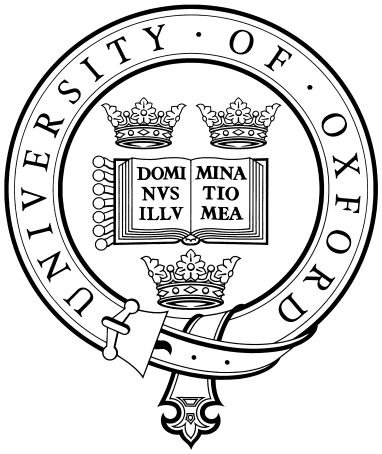
Formalisations and Applications of Business Process Modelling Notation



Peter Yung Ho Wong
Wolfson College
University of Oxford

A thesis submitted for the degree of
Doctor of Philosophy

Michaelmas 2011

To my parents, Eddie Wong and Emma Wong Chu, for their love and support.

Abstract

Business Process Modelling Notation (BPMN) is a standardised diagram notation for modelling interactive workflow processes graphically at the design stage. The primary objective of this thesis is to provide a framework for precise specifications and formal verifications of workflow processes modelled as BPMN diagrams. We provide two behavioural semantics for BPMN in the process algebra Communicating Sequential Processes (CSP). We apply existing CSP refinement orderings to both the refinement of business process diagrams and the verification of behavioural compatibility of business process collaborations. The first semantic model is an untimed model, focusing on the control flow and communication of business processes. The second semantic model extends the first one to capture the timing aspect of behaviour.

We also consider the applications of the semantic models. The secondary objective of this thesis is to apply BPMN and the semantic models to reason about long running empirical studies (e.g. laboratory experiments, clinical trials). We introduce a declarative workflow model `Empiricol` for recording trials and experiments precisely, and define bidirectional transformation functions between BPMN and `Empiricol`. Using the transformation functions, we make graphical specification, simulation, automation and verification of trials and experiments possible. We provide two case studies on the applications of BPMN's formalisations.

Acknowledgements

I am greatly indebted to my supervisor Jeremy Gibbons for his guidance and encouragement throughout the course of my DPhil. His constant positive attitude towards my research, and his unfailing, tireless and always-timely supports for guiding, reviewing and commenting my work have made this thesis possible. I would like to thank Jim Davies, Andrew Martin and Andrew Simpson for being my transfer and confirmation examiners, and for steering my research towards the right direction. I would also like to extend my gratitude and thanks towards my examiners Michael Butler and Andrew Simpson for their insightful comments and help in making this a better thesis.

I would like to thank my colleagues and friends at Oxford: Radu Calinescu for his helpful discussions on the CancerGrid trial model at the time of working on empirical studies modelling, Bill Roscoe for his useful advice on CSP responsiveness at the time of developing BPMN's relative timed semantics, Ib Holm Sørensen and Edward Crichton for spending time going through Z and Booster with me, and Philip Amstrong for his help on all sorts of problem to do with CSP. I would like to say thank you to members of the Software Engineering Research Group for always willing to help and making the fourth floor a much enjoyable place to work. I would especially like to thank John and Jun with whom I have tea breaks and lunches. Special thanks go to John Lyle for printing and submitting my thesis while I was away from Oxford!

Next I would like to thank my family, particularly my parents, Eddie and Emma, who have selflessly supported me in every aspect of my life, to my brother William and Lisa for their constant hospitality, to my in-laws, Amy, Danny, Margaret and Nancy, for being understanding during the course of my thesis. I would also like to thank my friends from University of Warwick, especially to Denis and Ant, for being very good friends and constant sources of laughter.

I would like to thank my colleagues at Fredhopper and friends in Amsterdam: Adrien and Noor, Andreas, Daniel, David and Nynke, José, Nikolay, and Stephan for their professional and moral support, for creating an exciting and warm environment to work and to live, and for helping me keep going during the writing up and correction periods of this thesis.

Special and huge thank you goes to my wife and best friend Wendy - for your love, continuous round-the-clock support and having undoubted confidence in me.

Finally, I wish to thank Microsoft Research for giving me financial support during the course of my DPhil, and Fredhopper for giving me the time to finish writing this thesis.

Dissemination

We have submitted and presented the following papers addressing the topics on specification of empirical studies, workflow modelling, and on formalising and verifying BPMN.

1. Peter Y.H. Wong and Jeremy Gibbons. **Property Specifications for Workflow Modellings**. In Special Issue of 7th International Conference on Integrated Formal Methods, Science of Computer Programming, October 2011 [WG11b]

2. Peter Y.H. Wong and Jeremy Gibbons. **Formalisations and Applications of BPMN**. In Special Issue of 7th International Workshop on the Foundations of Coordination Languages and Software Architectures, Science of Computer Programming, August 2011 [WG11a]

3. Peter Y.H. Wong and Jeremy Gibbons. **Property Specifications for Workflow Modellings**. In Proceedings of 7th International Conference on Integrated Formal Methods, February 2009 [WG09b]

   Invited for special issue in Science of Computer Programming.

4. Peter Y.H. Wong and Jeremy Gibbons. **A Process Semantics for BPMN**. In Proceedings of 10th International Conference on Formal Engineering Methods, October 2008 [WG08a]

5. Peter Y.H. Wong and Jeremy Gibbons. **Verifying Business Process Compatibility**. In Proceedings of 8th International Conference on Quality Software, August 2008 [WG08c]

   A preliminary version has been presented at 3rd International Workshop on Methods and Tools for Coordinating Concurrent, Distributed and Mobile Systems, June 2007.

   A shorter version of this paper has been presented at 2nd European Young Researchers Workshop on Service Oriented Computing, June 2007.

6. Peter Y.H. Wong and Jeremy Gibbons. **A Relative Timed Semantics for BPMN**. In Proceedings of 7th International Workshop on the Foundations of Coordination Languages and Software Architectures, July 2008 [WG09a].

   Invited for special issue in Science of Computer Programming.

   A shorter version of this paper has been presented at 3rd European Young Researchers Workshop on Service Oriented Computing, United Kingdom, June 2008.

7. Peter Y.H. Wong and Jeremy Gibbons. **On Specifying and Visualising Long-Running Empirical Studies**. In Proceedings of 1st International Conference on Model Transformation, July 2008 [WG08b]

8. Peter Y.H. Wong and Jeremy Gibbons. **A Process-Algebraic Approach to Workflow Specification and Refinement**. In Proceedings of 6th International Symposium on Software Composition, March 2007 [WG07].

9. Peter Y.H. Wong. **Towards A Unified Model for Workflow Processes**. Presented at 1st Service-Oriented Software Research Network workshop, June 2006. [Won06]

Contents

List of Figures

Chapter 1

Introduction

1.1 Workflow Technology

The concept of workflow has existed for a long time. A workflow procedure often began with some objectives to achieve, then someone would establish what needed to be done to achieve these objectives and thereafter assign these tasks to individuals. Usually these tasks are to be carried out in some predefined order.

With the prevalence of information technology, automated tools began to emerge to streamline workflow procedures that focus on paper-based administrative processes. Specifically, the goal of automated tools was to route the electronic version of these documents from one point to another. This was then the beginning of what is now known as workflow management systems.

As automated tools matured, and the need for enterprise application integration arose, in addition to document transfer among human participants, workflow management systems define data transfer among computer participants. They are used to define business logic, in particular to integrate heterogeneous and distributed systems. Workflows which implement business logic are called production workflows [ACKM03].

The emergence of workflow management systems offers support for composing, coordinating and monitoring the execution of manual tasks and automated services. The combination of loosely-coupled services and tasks allows workflows to define long running business processes that often exhibit concurrent behaviours. As a consequence, it is no longer possible to just hard-code this business logic into the execution environment or to implement routing procedures in an ad-hoc fashion, but instead workflow logic and coordination protocols have to be described precisely and declaratively. In order to share these descriptions across distributed environments, standards bodies [OMG, W3C, WFM] have made efforts to define a set of specification, modelling and description languages for designing and executing workflows.

Notable standards are WS-BPEL [BPE03] for services orchestration, and WSCI [W3C02] and its successor WS-CDL [KBR+05] for service choreography. These XML-based languages have well-defined syntax and vendors have subsequent implemented enactment engines. However, despite the emergence of workflow languages, no precise semantics has been provided and much related research has been focusing on formalising workflow languages using formalisms such as process algebras [BK85, Ros98, Mil99] and Petri Nets [Pet62]. Nevertheless these languages are textual and focused on *implementation* rather than *design*. A more prominent graphical design language from the standards bodies is the Business Process Modelling Notation (BPMN) [OMG08]. BPMN has been adopted by the Object Management Group [OMG] as a standardised graphical notation for modelling workflow processes. This leads to the first objective of this thesis:

Objective 1. *Our first objective is to provide a framework for formal specifications and verifications of workflow processes modelled using BPMN.*

Specifically we provide two process semantics for BPMN, both in the language of Communicating Sequential Processes (CSP) [Ros98]. Using CSP we show how the existing refinement orderings defined upon CSP processes can be applied to the refinement of business process diagrams and verification of the compatibility within a business collaboration; we first provide an untimed semantic model, focusing on the behaviour of control flow and concurrent interaction, we then provide a second semantic model that extends the first one with the notion of relative time in which the duration of a workflow activity is chosen nondeterministically from a bounded range.

1.2 Application: Empirical Studies

While BPMN has been employed extensively for modelling workflows of *service-centric systems* [ZMS07], which are composed of automated web services, we also consider the application of BPMN to modelling *long running empirical studies*. Empirical studies are plans describing a series of scientific procedures, which are interleaved with observations of the procedures over a period of time; these observations may be manually performed or automated, and are usually recorded in a calendar schedule. An example of a long-running empirical study is a clinical trial, where observations, specifically case report form submissions, are performed at specific points in the trial. In a clinical trial, observations are interleaved with clinical interventions on patients; precise descriptions of these observations and interventions are then recorded in a patient study calendar. For example, below is a schedule of drug administrations in a chemotherapeutic procedure adapted from the Neo-tAnGo clinical trial [ECH$^+$04]; we have omitted dosage for simplicity.

- Cyclophosphamide, every 14 days to 20 days
- Epirubicin, every 18 days to 21 days
- Paclitaxel, every 5 days to 10 days followed by Gemcitabine, up to 10 days

Currently information about observations and empirical procedures are specified in study planners such as trial designers either textually or via some XML-based data entry forms [CHG$^+$07]. However, the constraints on the order at which observations and scientific procedures are carried out may be complex, and their precise specification can be time consuming and prone to error. Going back to the example above, based on Hammond et al.'s set of safety principles [HSW95], one can envisage the safety rule: "No more than one dosage of Gemcitabine may be given after the administration of Cyclophosphamide and before Epirubicin." However current methods of specification do not lend themselves easily to capture such properties or to verify study plans against them. We believe the method of specification may be simplified and improved by allowing specifications to be constructed formally and graphically, and visualised as workflow instances. This leads to the second objective of this thesis:

Objective 2. *Our second objective is to introduce* `Empiricol`*, a declarative workflow model for recording empirical studies precisely, and to provide bi-directional transformation functions between* `Empiricol` *and BPMN, by which graphical specification, simulation, automation and verification of empirical studies are made possible.*

1.3 Thesis Contribution and Outline

The contribution of this thesis is to provide a CSP-based framework for declaratively and graphically specifying both *service-centric systems* and *empirical studies* described as BPMN diagrams, and verifying these workflow processes against abstract behavioural properties via automatic refinement checking. Figure 1.1 depicts the data flow of our proposed framework. The core framework is composed of nine modules, and is implemented using the Haskell functional programming language [Jon03]. These modules are grouped by a rounded dash line rectangle in the figure. Four of these modules – *XML To BPMN*, *BPMN To XML*, *XML To Empiricol* and *Empiricol To XML* – implement the XML (de)serialisations for our internal representation of BPMN and the empirical studies model. The module *To CSPm*, on the other hand, translates our internal representation of CSP definition to machine-readable CSP (CSPm) script, which may then be analysed using software tools such as the CSP model checker FDR [For98]. The rest of this section describes the structure of this thesis and the relationship between the chapters and the framework; a journal version of this thesis has been published in Science of Computer Programming [WG11a].

In Chapter 2 we give an informal presentation of the subset of BPMN considered in this thesis, and introduce a running example, which we refer to throughout this thesis; we then give a brief introduction to Haskell, which we use to implement our semantic functions; we also present the syntax of CSP along with its traces, stable failures and refusal traces models as well as their associated refinement orderings. In addition, assuming knowledge of set theory and predicate logic, we introduce Z's mathematical notation and its corresponding schema language; we use Z to formalise the syntax of BPMN.

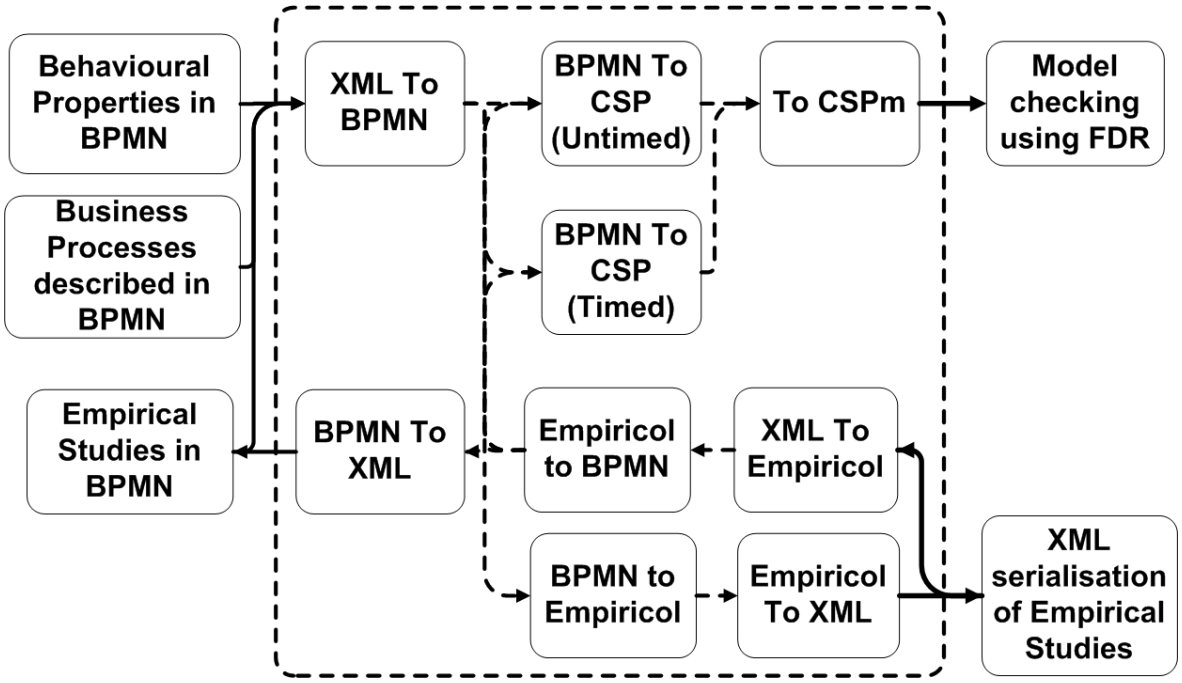


Figure 1.1: A framework for analysing service centric systems and empirical studies

Chapter 3 presents an overview of related work on formalising workflow descriptions, and specifying and visualising empirical studies.

Our main contribution starts in Chapter 4: in this chapter we provide a detailed study of the subset of BPMN syntax considered in this thesis, and provide it a formal specification in Z notation; we provide the semantic functions on this model. In this chapter we also use Z notation to formally specify a comprehensive set of operations for constructing BPMN diagrams.

In Chapter 5 we present a process semantics for BPMN in CSP. This semantics emphasizes the untimed behaviour associated with BPMN diagrams: we model each BPMN diagram as a parallel composition of CSP processes where each process models the untimed behaviour of a BPMN element contained in the diagram and the flow of control between elements is modelled as the synchronisation of the shared interface of the parallel composition. Our model permits hierarchical composition allowing formal reasoning at various levels of abstractions and semantic comparison of BPMN descriptions via CSP's traces and failures refinements. We also study the compositionality of the operations defined in Chapter 4 in the context of this semantics, and develop a notion of behavioural compatibility between BPMN processes interacting in a business collaboration. The semantic model described in this chapter corresponds to the module *BPMN To CSP (Untimed)* in Figure 1.1.

In Chapter 6 we present a relative timed model for BPMN: this model augments its untimed counterpart in Chapter 5 by extending it with the notion of relative time in which the duration of an activity is chosen nondeterministically from a bounded range. The semantic model described in this chapter corresponds to the module *BPMN To CSP (Timed)* in Figure 1.1.

In Chapter 7 we present a generic observation workflow model, Empiricol, an extended version of the CancerGrid trial workflow model [HC06], for modelling empirical studies declaratively. This model is intended to bridge the gap between BPMN and empirical studies. We also provide bidirectional transformation functions between Empiricol and BPMN. By leveraging BPMN's behavioural semantics provided in previous chapters, we show how empirical studies such as clinical trials may be specified graphically and verified against (oncological) safety properties. The implementation of Empiricol and its transformation functions correspond to modules *BPMN To Empiricol* and *Empiricol To BPMN* in Figure 1.1.

In Chapter 8 we present two case studies. The first case study considers the specification and

verification of an airline reservation systems. This system is adopted from the Web Service Choreography Interface specification document [W3C02]. The second case study considers the formal analysis of the participant workflow of the Neo-tAnGo clinical trial [ECH$^+$04] using the combination of Empiricol, BPMN and its CSP semantics.

In Chapter 9 we consider current approaches to formalising BPMN, and make comparisons between them and our formalisations where possible.

We conclude this thesis in Chapter 10 with a discussion of our contribution and possible future research.

Chapter 2

Preliminaries

In this chapter we give an informal description of BPMN, and introduce a running example; we then give a brief introduction to Haskell [Jon03], which we use to implement our semantic and transformation functions; we also present the syntax and the semantics of CSP [Ros98], which we use to define the semantics of BPMN, and Z [Spi92], which we use to formalise the syntax of BPMN. Specifically, Section 2.1 introduces BPMN; Sections 2.2 and 2.3 give a brief overview of Haskell and Z, and Section 2.4 presents CSP's syntax and semantics. In this thesis we use the `typewriter` font when referring to Haskell expressions and the *math* font when referring to non Haskell mathematical expressions, such as those defined using Z and CSP.

2.1 Business Process Modelling Notation

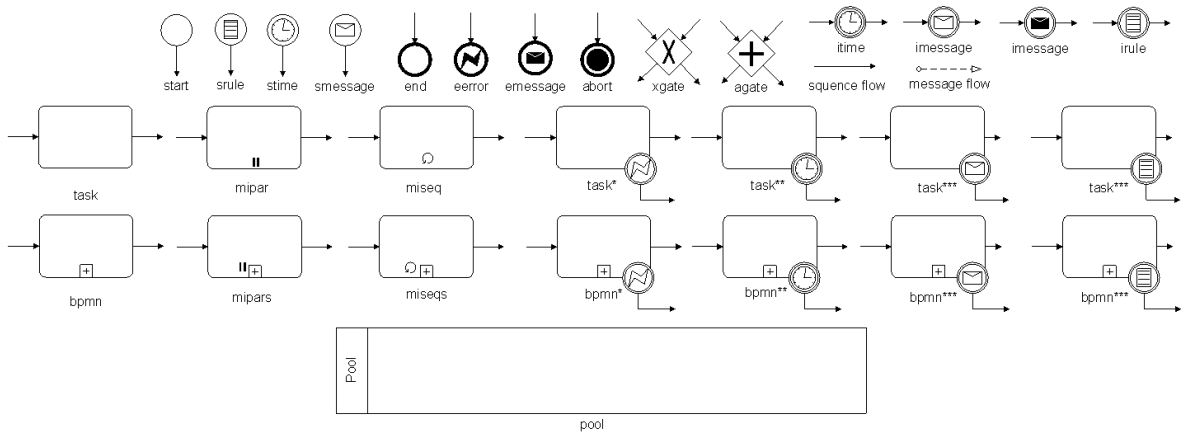


Figure 2.1: Our subset of BPMN elements

Business Process Modelling Notation (BPMN) [OMG08] is a graphical modelling language for business analysts to specify business processes as workflows. It is the language that bridges the gap between visualisation of the business processes and their executable implementation such as those defined in XML-based languages like the Business Process Execution Language [BPE03] (WS-BPEL) for implementing business processes using Web Services. We review related work on WS-BPEL in Chapter 3.

In this thesis we consider the subset of BPMN shown in Figure 2.1. Some BPMN elements have been omitted from this subset due to one of the following reasons.

1. The element is used specifically to express data flow or transactional behaviour.
2. The element may be semantically expressed using a combination of elements in the subset shown in Figure 2.1.

In this thesis we consider synchronous communications between elements in a BPMN diagram. We do not consider transactional behaviour: we believe that transactional behaviour should be studied with a formal modelling language, such as Compensating CSP [BHF05], that has transaction and compensation built into its syntax and semantics. Similarly, we do not consider data flow behaviour: data flow communications are asynchronous and should be studied with a formal modelling language, such as

Data-Flow Sequential Processes [Jos05], that has asynchronous interactions as its primitives. In the remaining section we describe the elements in this subset and justify why an element or a particular behaviour of an element is not selected.

A BPMN diagram is made up of a collection of graphical elements. Graphical elements in BPMN are categorised into flow objects, connecting objects, swimlanes and artifacts. A flow object is either an event, an activity or a gateway. We describe connecting objects in Section 2.1.1, flow objects in Sections 2.1.2, 2.1.3 and 2.1.4, and swimlanes and artifacts in Section 2.1.5. We introduce a running example to be used throughout this thesis in Section 2.1.6.

2.1.1 Connecting Objects

A connecting object is either a sequence flow, a message flow or an association. Each connecting object has the attributes Name, SourceRef and TargetRef. SourceRef defines a flow object as the source of the flow and TargetRef defines another flow object as the target of the flow [OMG08, Section 10.1.1]. A sequence flow is used to show the order in which activities, contained in a pool, will be performed [OMG08, Section 10.1.2]. A sequence flow is drawn as a solid line with a solid arrowhead and is depicted in Figure 2.1 by the element labelled sequence flow. A message flow is used to show the flow of messages between two participants (each represented by a BPMN pool) in a BPMN diagram. That is, it connects from a flow object of one BPMN pool to a flow object of another BPMN pool [OMG08, Section 10.1.3]. A message flow is drawn as a dashed line with a open arrowhead and is depicted in Figure 2.1 by the element labelled message flow An association is used to associate information and data with flow objects [OMG08, Section 10.1.4]. We do not consider associations in this thesis as our semantics abstract from the internal flow of data in an individual participant (business process) of a diagram.

The informal semantics in the official specification adopts the concept of "token" [OMG08, page 36] to facilitate the discussion of how sequence flows proceed within a BPMN process. We do not consider this concept to be compatible with the goal of our semantic definitions, as a diagram's behaviours can no longer be solely determined explicitly by the behaviour of elements it contains. For example, consider the participant Manufacturer in the BPMN diagram shown in Figure 2.2. After receiving authorisation

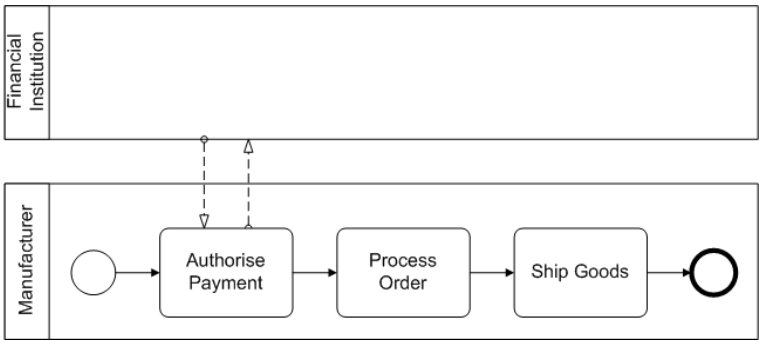


Figure 2.2: Interactions between two participant in a business process

(*Authorise Payment*), the manufacturer processes the order and ships out goods; this business process completes successfully as long as both of these tasks complete successfully. However this is no longer possible if, for instance, shipping out requested goods requires two tokens as a starting condition and only one token is passed through the sequence flow from processing the order to shipping out goods. While one might argue that two authorisations are required to process the order, we believe this should be represented explicitly, via multiple instance elements, for example. We therefore do not consider tokens.

2.1.2 Events

An event represents a "trigger" during the execution of the business process. Each event has a common attribute EventType, which takes the value Start, Intermediate or End denoting whether the event is at

the start, in the middle, or at the end of a process. Graphically, the types of event are distinguished by different line styles – start events are represented by a simple circle drawn with a thin black line (e.g. elements labelled as start and stime in Figure 2.1); end events are represented by a circle drawn with a single thick black line (e.g. elements labelled as end and emessage in Figure 2.1); and intermediate events are represented by a circle drawn with a double thin black line (e.g. elements labelled as itime and irule in Figure 2.1).

Intermediate events may be attached to the boundary of an activity element to represent exceptions that can interrupt that activity [OMG08, Section 10.2.2]. More than one intermediate event may be attached to a single activity element. The execution of this activity element may then be interrupted when one of these events is triggered, which in turn triggers an exception flow.

Each event is associated with a trigger [OMG08, Tables 9.4, 9.6 and 9.8]. Its value is represented by a corresponding marker. For our subset of BPMN we consider triggers None, Message, Timer, Rule and Error.

2.1.2.1 Message events

A message event either sends or receives a message from another BPMN pool (business process) in the same BPMN diagram. Message events are shown in Figure 2.1 as elements labelled smessage (start), imessage (intermediate) and emessage (end). While there are attributes associated with the content of messages and the messaging implementation, we do not consider them in this thesis. Each start (smessage) and intermediate (imessage) message event may have at most one incoming message flow, and each end (emessage) message event may have at most one outgoing message flow. Message events may also have no associated message flow, in particular when the BPMN diagram has only one BPMN pool.

2.1.2.2 Timer Events

A timer event denotes a time duration. It is triggered by either a time-stamp (e.g. 16:48, 15th August 2008) or a duration (e.g. 7 days). Timer events are shown in Figure 2.1 as elements labelled stime (start), and itime (intermediate); there is no end timer event in BPMN. A timer event has either a TimeDate attribute, specifying an absolute time stamp as the deadline upon which the event is triggered, or a TimeCycle attribute, specifying a relative period of delay after which the event is triggered. Since our timed model treats time relatively (see Chapter 6 for details of the timed model), we only consider the TimeCycle attribute.

2.1.2.3 Rule Events

A rule event specifies either a rule or a Boolean condition. It is triggered when its rule or condition becomes true. Rule events are shown in Figure 2.1 as elements labelled srule (start) and irule (intermediate). There is no end rule event in BPMN. Each rule event has a RuleName attribute to record its condition. While our semantic definitions abstract away from the evaluation of conditions, for the purpose of recording information about *empirical studies*, we record rule conditions syntactically. We consider empirical studies in Chapter 7.

2.1.2.4 None Events

A none event is an event without any triggers. These events are depicted in Figure 2.1 by elements labelled start and end respectively. They represent the start and the end of a business process respectively. In this thesis we refer to these events as non-trigger events. While BPMN also has a non-trigger intermediate event, we do not consider it because it does not have an explicit behavioural description.

2.1.2.5 Error Events

An error event represents an error during the business process execution. In BPMN an error event may be triggered at the end of or during a business process execution. In Figure 2.1, error events are depicted by elements labelled eerror (end) and ierror (intermediate). An error event has an ErrorCode attribute for identifying a particular error of the process containing the event and throws an error defined by that ErrorCode when triggered. An intermediate error event may be attached to the boundary of an activity

element, and in which case it catches a specific error defined by its ErrorCode and is thrown by an error event in that activity; if no ErrorCode is specified, an intermediate error may catch any error thrown from that activity.

2.1.3 Activities

An activity element denotes work in a business process. Each activity is drawn as a rectangle with rounded corners. An activity may be atomic or compound. An atomic activity is a task and is depicted in Figure 2.1 by the element labelled task. A task is drawn as a rectangle with at most one marker inside it to represent its type. A compound activity is a subprocess. A subprocess is itself another business process and is made up of other BPMN elements. A subprocess can be in a collapsed view, hiding its details, or it can be in an expanded view, showing its details.

To record an activity in our abstract syntax, we consider two attributes ActivityType and LoopType [OMG08, page 50]. ActivityType takes one of the values Task (atomic) or SubProcess (compound), and LoopType takes one of the values None and MultiInstance. The value None indicates that the activity is instantiated once every time it is triggered, while the value MultiInstance indicates that the activity is instantiated multiple times every time it is triggered.

An activity element also has attributes for recording its flow of data as well as other internal properties. For example, an activity element may be associated with one or more properties which are "local" to it and are only for use within the processing of the activity. We do not consider these attributes; our semantic definitions aim to model the communication of the activity and treat the execution of each atomic activity (task) as instantaneous.

A multiple instance activity repeats itself for a number of times according to the evaluation of its attributes. Each multiple instance activity has attributes MI_Condition, LoopCounter, MI_Ordering, MI_FlowCondition and ComplexMI_FlowCondition. Each multiple instance activity evaluates its loop condition (MI_Condition) once before the activity is performed; this expression returns the number of times that the activity is to be repeated [OMG08, Section 9.4.1.2]. There are two types of multiple instances: instances of the activity may be performed either sequentially or in parallel. This variation is determined by the attribute MI_ordering.

The element labelled miseq in Figure 2.1 depicts a sequential multiple instance task, and the element labelled miseqs depicts a sequential multiple instance subprocess. Similarly, the element labelled mispar depicts a parallel multiple instance task, and the element labelled mipars depicts a parallel multiple instance subprocess.

The attributes MI_FlowCondition and ComplexMI_FlowCondition are used to determine how a multiple instance activity's outgoing sequence flow is triggered. MI_FlowCondition may be set to the value None, One, All or Complex. If MI_FlowCondition is None then the activity's outgoing sequence flow is triggered every time after one of the activity instances has completed execution; if the value is One then the activity's outgoing sequence flow is triggered once after the first activity instance has completed execution; if the value is All then the outgoing sequence flow is triggered once after all activity instances have completed execution, and if the value is Complex then the expression defined by the attribute ComplexMI_FlowCondition determines when and how many times the activity's outgoing sequence flow is triggered. We do not consider the flow conditions None and Complex, since our semantic definitions abstract from process data.

2.1.4 Gateways

Gateway elements are used to control how sequence flows interact as they converge and diverge in a BPMN diagram [OMG08, Section 9.5]. A gateway is drawn as a diamond shape; different internal markers correspond to different types of gateways. Each gateway has a common attribute GateType, which takes the value XOR, OR, Complex or AND, representing exclusive, inclusive, complex or parallel respectively. Furthermore, each exclusive gateway has an attribute XORType, which may be set to Data or Event. All gateways in our subset must have at least one incoming sequence flow and at least one outgoing sequence flow. We now provide an analysis of the types of gateway.

2.1.4.1 Exclusive (XOR) and Parallel (AND) Gateways

A data-based XOR gateway is depicted in Figure 2.1 by the element labelled xgate. This type of gateway triggers one of its outgoing sequence flows depending on the internal evaluation of conditions associated to each outgoing sequence flow [OMG08, Section 9.5.2.3]. Our semantic definitions abstract from these conditions to capture the behaviour of all possible executions of BPMN diagrams.

An event-based XOR gateway is depicted in Figure 2.1 by the element labelled exgate. This type of gateway triggers one of its outgoing sequence flows depending on the events that occur at the target of the gateway's outgoing sequence flows; these could be receiving a message or an elapsed duration. As a consequence, the targets of its outgoing sequence flows may either be an intermediate event or a task. Both data-based and event-based exclusive gateway trigger their outgoing sequence flow when one of their incoming sequence flows is triggered [OMG08, Section 9.5.2].

An AND gateway triggers all of its outgoing sequence flows when all of its incoming sequence flows are triggered [OMG08, Section 9.5.5]. An AND gateway is depicted as the element labelled agate in Figure 2.1.

2.1.4.2 Inclusive (OR) and Complex gateways

We do not include complex or inclusive gateways. The complex gateway is a syntactic sugaring to combine a set of connected simple gateways [OMG08, page 82]. In an inclusive (OR) gateway, the evaluation of condition expressions of one of its outgoing sequence flows does not exclude the evaluation of condition expressions of its other outgoing sequence flows. This means it is not possible to provide a compositional behavioural model for the merging effect on the gateway's incoming sequence flows. Even the most efficient method that is currently available depends on a recursive algorithm to investigate the properties of elements that come before the source of the gateway's incoming sequence flows at runtime [DGHW07].

2.1.5 Swimlanes and Artifacts

A swimlane [OMG08, Section 9.6] is either a pool or a lane. A pool represents a participant in a business process; a participant may be an entity such as a company. A lane is a sub-partition within a pool. We do not consider lanes in this thesis as they neither influence the behaviour of elements in the pool nor its sequence and message flows. A BPMN pool is depicted in Figure 2.1 as the element labelled pool. A BPMN diagram, which represents a complete business process, consists of one or more BPMN pools. Each BPMN pool has the attribute Process that records the business process contained in that pool. While each diagram has other attributes, such as ModificationDate, for the purpose of formalising BPMN's syntax and semantics, our abstract syntax records only attributes Id, which identifies the BPMN diagram, and Pools, which records the set of pools in the diagram. Similar to lanes, artifacts provide the capabilities of showing additional information about a business process that is not directly related to the sequence or message flows of the business process [OMG08, Section 9.7].

2.1.6 Running Example

In this thesis we consider the BPMN diagram shown in Figure 2.3 as a running example. The diagram describes the business process of an online shop promoting a sale. Specifically, it is a business collaboration between an online shop and a customer.

2.1.6.1 Online Shop

The online shop business process begins by sending a message to the customer about a sale offer (Send Offer). It then waits until it receives either a confirmation (Receive Confirmation) or a decline (Receive Decline) from the customer. If a decline is received, the online shop business process ends. If a confirmation is received, the online shop receives payment from the customer, sends the invoice and dispatches the goods to her.

2.1.6.2 Customer

The customer's business process begins by receiving a message from the online shop about a certain promotion item. She may either accept (Accept Offer) or decline (Decline Offer) the offer. If she decides

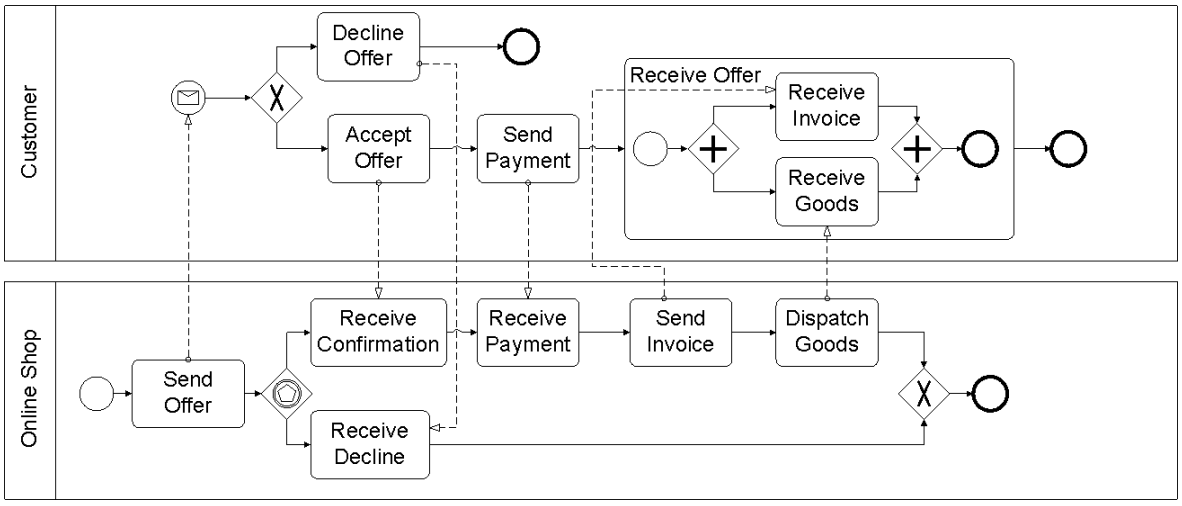


Figure 2.3: A running example of a BPMN diagram

to accept the offer, she sends payment to the shop (Send Payment), then waits for her goods (Receive Goods) and invoice (Receive Invoice) to arrive — in either order. If she declines the offer, the business process ends.

2.2 Haskell and Z

Haskell is a functional programming language [Jon03]. In this section we concentrate on introducing features that are required for defining BPMN's abstract syntax and semantic functions.

A Haskell program may be thought of as a collection of functions in which each function is given a type and a definition. For example below is the definition of the function `factorial`, which takes some positive integer n and returns its factorial $n!$.

```
factorial :: Integer -> Integer
factorial 0 = 1
factorial n = n * factorial (n-1)
```

Haskell allows functions to be defined recursively: in the case of the function `factorial` for each $n \neq 0$, `factorial(n)` is defined in terms of `factorial(n-1)`. It also a provides facility for pattern matching; in the above example, the function `factorial` tries to match each input value n of type `Integer` against the integer value 0; if that succeeds it returns the value 1; otherwise it matches anything else and returns the value `n * factorial (n-1)`.

A Haskell program may use any of the predefined *data types*. For example, the data type `Integer` is a predefined data type. It may also contain user-defined data types, a data type takes the form `data T a b ... = A | B a | C b a ...` where `T` is its name, parameterised by a finite (possibly empty) list of unique type parameters. Each data type is defined by a list of distinct constructors, each of which describes a different way of creating values of that type. Each constructor may contain zero or more parameters. For example the predefined data type `Maybe` is defined as `data Maybe a = Nothing | Just a`, where `Nothing` is a nullary constructor and `Just` is a constructor which takes a value of type `a`.

In general to retrieve the value of one of the type parameters of a given data type value, one could use *pattern matching*. For example, given `data D = D Int`, one may define function `f :: D -> Int` such that `f (D n) = n`. Alternatively one could define the data type as a record type of the following form,

```
data T a b ... = A | B {f :: a} | C {g :: b,h :: a} ...
```

where `f`, `g` and `h` are field names. For example `D` may be defined as `data D = f :: Int` and `f :: D -> Int` will be automatically provided as a function on `D` to retrieve the integer parameter.

In this thesis we use Haskell data types to implement the abstract syntax of BPMN and CSP. The semantic functions are implemented as transformations from one data type to another.

2.3 Z Notation

The Z notation [Spi92] is a state-based specification language. It is based on typed set theory coupled with a structuring mechanism: the schema. A schema is a pattern of declaration and constraint, and may be defined using the following syntax:

$$\begin{array}{|l} \textit{Name} \\ \hline \textit{components} \\ \hline \textit{predicate} \\ \hline \end{array}$$

or equivalently $\textit{Name} \mathrel{\widehat=} [\textit{components} \mid \textit{predicate}]$, where its predicate part may have constraints upon the values of its components. If S is a schema then θS is the characteristic binding of S in which each component is associated with its current value. Schemas can be used as declarations, for example, the lambda expression $\lambda\, S \bullet t$ is a function from the binding of schema S to the type of term expression t.

Z also provides a syntax for set expressions, predicates and definitions. A type can be either a basic type or a free type. A basic type is a maximal set within the specification and is defined by declaring its name $[\textit{Type}]$, and a free type is introduced by identifying each of its distinct constants and constructor functions.

$$\textit{Type} ::= \textit{element}_1 \mid ... \mid \textit{element}_m \mid \textit{fun}_1 \langle\langle \textit{term}_1 \rangle\rangle \mid \ldots \mid \textit{fun}_n \langle\langle \textit{term}_n \rangle\rangle$$

An alternative way to define an object in a Z specification is by abbreviation of the form $\textit{symbol} == \textit{term}$. An abbreviation introduces a new global constant. The value of the symbol on the left is given by the term on the right, and its type is the same as the type of the term.

An axiomatic definition introduces new symbols to denote elements of a certain type satisfying certain predicate constraint. For example, the following introduces a new symbol x, an element of S, satisfying predicate p.

$$\begin{array}{|l} x : S \\ \hline p \\ \hline \end{array}$$

We could use this feature to define functions, relations and predicates. For example, the function *double* is defined such that for every natural number m there is a unique number n which is double the value of m.

$$\begin{array}{|l} \textit{double} : \mathbb{N} \to \mathbb{N} \\ \hline \forall\, m, n : \mathbb{N} \bullet m \mapsto n \in \textit{double} \Leftrightarrow n = m + m \\ \hline \end{array}$$

Although we choose Haskell for implementing the abstract syntax of our subset of BPMN elements and its programming language constructs allow us to specify some of the structural constraints about the syntax, the schema calculus in Z allows us to specify constraints of the syntax without providing an implementation. For example, for any function f we could simply write $f^{\sim}$ for its inverse. The schema calculus also provides a natural way to group constraints about a particular type of element.

2.4 Communicating Sequential Processes

In CSP [Hoa85, Ros98], a process is defined as a pattern of possible behaviour; a behaviour consists of events which are atomic and synchronous between the environment and the process. The environment in this case can be another process. Events can be compound, constructed using '.' the dot operator; compound events may behave as channels communicating data objects synchronously between the process and the environment. For example $a.b$ is a compound event that communicates value b through channel

a. The set of events which process P can perform is P's alphabet, denoted as αP. Below is the syntax of CSP.

$$\begin{array}{rcl} P & ::= & P \,|||\, P \;\mid\; P \,[\![\, A \mid B \,]\!]\, Q \;\mid\; P \,[\![\, A \,]\!]\, P \;\mid\; P \setminus A \;\mid\; P \triangle P \;\mid \\ & & P \,\Box\, P \;\mid\; P \sqcap P \;\mid\; P \fatsemi P \;\mid\; e \to P \;\mid\; \mathit{Skip} \;\mid\; \mathit{Stop} \\ e & ::= & x \;\mid\; x.e \end{array}$$

Process $P \,|||\, Q$ denotes the interleaving of processes P and Q. Process $P \,[\![\, A \mid B \,]\!]\, Q$ denotes parallel composition, in which P and Q can evolve independently but must synchronise on every event in the set $A \cap B$; the set A is the alphabet of P and the set B is the alphabet of Q, and no event in A and B can occur without the cooperation of P and Q respectively. Process $P \,[\![\, A \,]\!]\, Q$ denotes the partial interleaving of processes P and Q, in which P and Q can evolve independently but synchronise on events in set A. We write $|||\, i : I \bullet P(i)$, $\|\, i : I \bullet \alpha P(i) \circ P(i)$ and $\|[A]\, i : I \bullet P(i)$ to denote an indexed interleaving, parallel combination and partial interleaving of processes $P(i)$ for i ranging over I. We provide the following usage example for parallel operators where $I = \{1, 2, 3\}$.

$$\begin{array}{rcl} |||\, i : \{1,2,3\} \bullet P(i) & \equiv & P(1) \,|||\, P(2) \,|||\, P(3) \\ \|\, i : \{1,2,3\} \bullet \alpha P(i) \circ P(i) & \equiv & P(1) \,[\![\, \alpha P(1) \mid \alpha P(2) \cup \alpha P(3) \,]\!]\, (P(2) \,[\![\, \alpha P(2) \mid \alpha P(3) \,]\!]\, P(3)) \\ \|[A]\, i : \{1,2,3\} \bullet P(i) & \equiv & P(1) \,[\![\, A \,]\!]\, P(2) \,[\![\, A \,]\!]\, P(3) \end{array}$$

Process $P \setminus A$ is obtained by hiding all occurrences of events of P in set A from the environment. Process $P \triangle Q$ denotes a process initially behaving as P, but may be interrupted at any time and behave as Q. Process $P \,\Box\, Q$ denotes the external choice between processes P and Q; the process is ready to behave as either P or Q. An external choice over a set of indexed processes is written as $\Box\, i : I \bullet P(i)$. Process $P \sqcap Q$ denotes the internal choice between processes P or Q, ready to behave as at least one of P and Q, the choice being made without cooperation from the environment. Similarly an internal choice over a set of indexed processes is written as $\sqcap\, i : I \bullet P(i)$.

Process $P \fatsemi Q$ denotes a process ready to behave as P; after P has successfully terminated, the process is ready to behave as Q. We write $\fatsemi\, i : S \bullet P(i)$ to denote an indexed sequential composition of processes $P(i)$ where i ranges over the set S. Process $e \to P$ denotes a process capable of performing event e, after which it behaves like process P. The process *Stop* is a deadlocked process and the process *Skip* is a successful termination.

2.4.1 Traces Model

CSP has three standard behavioural models for its semantics: the traces, stable failures and failures-divergences models; their respective refinement orderings are in order of increasing precision [Ros98]. Below is the trace semantics of individual CSP operators where $\mathcal{T}[\![\,\cdot\,]\!]$ is a semantic function such that $\mathcal{T}[\![P]\!]$ maps the CSP process P to its set of possible traces of type $\mathbb{P}(\operatorname{seq}\Sigma)$, and Σ is the set of all possible events. The special event $\checkmark$ represents a successful termination.

$$\begin{array}{rcl} \mathcal{T}[\![P \,|||\, Q]\!] & = & \bigcup \{ tr_p, tr_q : \operatorname{seq}\Sigma \mid tr_p \in \mathcal{T}[\![P]\!] \land tr_q \in \mathcal{T}[\![Q]\!] \bullet tr_p \,|||\, tr_q \} \\ \mathcal{T}[\![P \,[\![\, A \,]\!]\, Q]\!] & = & \bigcup \{ tr_p, tr_q : \operatorname{seq}\Sigma \mid tr_p \in \mathcal{T}[\![P]\!] \land tr_q \in \mathcal{T}[\![Q]\!] \bullet tr_p \,[\![\, A \,]\!]\, tr_q \} \\ \mathcal{T}[\![P \,[\![\, A \mid B \,]\!]\, Q]\!] & = & \{ tr : \operatorname{seq}\Sigma \mid \operatorname{ran} tr \subseteq A \cup B \land tr \upharpoonright A \in \mathcal{T}[\![P]\!] \land tr \upharpoonright B \in \mathcal{T}[\![Q]\!] \,\} \\ \mathcal{T}[\![P \setminus A]\!] & = & \{ tr : \operatorname{seq}\Sigma \mid tr \in \mathcal{T}[\![P]\!] \bullet tr \upharpoonright (\Sigma \setminus A) \} \\ \mathcal{T}[\![P \triangle Q]\!] & = & \mathcal{T}[\![P]\!] \cup \{ tr_p : \mathcal{T}[\![P]\!];\ tr_q : \mathcal{T}[\![Q]\!] \mid \checkmark \notin \operatorname{ran} tr_p \bullet tr_p \frown tr_q \} \\ \mathcal{T}[\![P \,\Box\, Q]\!] & = & \mathcal{T}[\![P]\!] \cup \mathcal{T}[\![Q]\!] \\ \mathcal{T}[\![P \sqcap Q]\!] & = & \mathcal{T}[\![P]\!] \cup \mathcal{T}[\![Q]\!] \\ \mathcal{T}[\![P \fatsemi Q]\!] & = & \{ tr_p : \mathcal{T}[\![P]\!] \mid \checkmark \notin \operatorname{ran} tr_p \} \cup \\ & & \{ tr_p : \mathcal{T}[\![P]\!];\ tr_q : \mathcal{T}[\![Q]\!] \mid \checkmark \notin \operatorname{ran} tr_p \land tr_p \frown \langle\checkmark\rangle \in \mathcal{T}[\![P]\!] \bullet tr_p \frown tr_q \} \\ \mathcal{T}[\![e \to P]\!] & = & \{\langle\rangle\} \cup \{ tr_p : \mathcal{T}[\![P]\!] \bullet \langle e \rangle \frown tr_p \} \\ \mathcal{T}[\![\mathit{Stop}]\!] & = & \{\langle\rangle\} \\ \mathcal{T}[\![\mathit{Skip}]\!] & = & \{\langle\rangle, \langle\checkmark\rangle\} \end{array}$$

The trace $t \upharpoonright R$ denotes t restricted to events in R, and is defined by the following clauses.

$$\begin{aligned} \langle\rangle \upharpoonright R &= \langle\rangle \\ (t \frown \langle a\rangle) \upharpoonright R &= (t \upharpoonright R) \frown \langle a\rangle \quad (a \in R) \\ (t \frown \langle a\rangle) \upharpoonright R &= t \upharpoonright R \quad (a \notin R) \end{aligned}$$

The term $s \mid\mid\mid t = s \,[\![\emptyset]\!]\, t$ where $s \,[\![R]\!]\, t$ the set of interleavings of traces s and t, synchronizing on events in R, and is defined as follows for all $s, t \in \operatorname{seq} \Sigma$, $x, x' \in R$ and $y, y' \in \Sigma \setminus R$.

$$\begin{aligned} s \,[\![R]\!]\, t &= t \,[\![R]\!]\, s \\ \langle\rangle \,[\![R]\!]\, \langle\rangle &= \{\langle\rangle\} \\ \langle\rangle \,[\![R]\!]\, \langle x\rangle &= \emptyset \\ \langle\rangle \,[\![R]\!]\, \langle y\rangle &= \{\langle y\rangle\} \\ \langle x\rangle \frown s \,[\![R]\!]\, \langle y\rangle \frown t &= \{u : \operatorname{seq} X \mid u \in \langle x\rangle \frown s \,[\![R]\!]\, t \bullet \langle y\rangle \frown u\} \\ \langle x\rangle \frown s \,[\![R]\!]\, \langle x\rangle \frown t &= \{u : \operatorname{seq} X \mid u \in s \,[\![R]\!]\, t \bullet \langle x\rangle \frown u\} \\ \langle x\rangle \frown s \,[\![R]\!]\, \langle x'\rangle \frown t &= \emptyset \quad (x \neq x') \\ \langle y\rangle \frown s \,[\![R]\!]\, \langle y'\rangle \frown t &= \{u : \operatorname{seq} X \mid u \in s \,[\![R]\!]\, \langle y'\rangle \frown t \bullet \langle y\rangle \frown u\} \cup \\ &\quad \{u : \operatorname{seq} X \mid u \in \langle y\rangle \frown s \,[\![R]\!]\, t \bullet \langle y'\rangle \frown u\} \end{aligned}$$

Using the traces semantics we can reason about safety properties of CSP processes. When considering safety properties, we are interested to know whether CSP processes are able to perform anything unsafe, that is, perform any behaviour that is outside their *safety specification*. Formally, process Q traces-refines P is expressed as follows:

$$P \sqsubseteq_{\mathcal{T}} Q \Leftrightarrow \mathcal{T}[\![P]\!] \supseteq \mathcal{T}[\![Q]\!]$$

For example let $P = a \rightarrow b \rightarrow \mathit{Stop}$ be our safety requirement, then CSP process $Q = a \rightarrow \mathit{Stop}$ would satisfy P. More precisely, any CSP processes satisfying this may either perform nothing or only perform a, or a followed by b. Semantically this means the traces of any process satisfying this requirement must be a subset of the traces of P.

Furthermore, two CSP processes are traces equivalent if and only if they traces-refine each other:

$$P \equiv_{\mathcal{T}} Q \Leftrightarrow P \sqsubseteq_{\mathcal{T}} Q \land Q \sqsubseteq_{\mathcal{T}} P$$

2.4.2 Stable Failures Model

In this thesis we concentrate on the stable failures model because the traces model does not record the availability of events. Notable is the semantic equivalence of processes $P \,\Box\, Q$ and $P \sqcap Q$ under the traces model. In order to distinguish these processes, it is necessary to record not only what a process can do, but also what it can *refuse* to do. This information is preserved in *refusal sets*, sets of events a process in a stable state can refuse to communicate. A *failure* of a process is a pair of which the first element is a trace of the process and the second is a refusal set of the process after that trace. The failures semantics of individual CSP operators is defined as follows, where $\mathcal{F}[\![\cdot]\!]$ is a semantic function that maps a CSP expression to its set of failures $\mathbb{P}(\operatorname{seq} \Sigma \times \mathbb{P}\, \Sigma)$.

$$\mathcal{F}[\![P \mid\mid\mid Q]\!] =$$
$$\{tr_p, tr_q, tr : \operatorname{seq}\Sigma;\ ref_p, ref_q, ref : \mathbb{P}\,\Sigma \mid$$
$$(tr_p, ref_p) \in \mathcal{F}[\![P]\!] \land (tr_q, ref_q) \in \mathcal{F}[\![Q]\!] \land tr \in tr_p \mid\mid\mid tr_q \land$$
$$ref_p \setminus \{\checkmark\} = ref_q \setminus \{\checkmark\} \land ref = ref_p \cup ref_q \bullet (tr, ref)\}$$

$$\mathcal{F}[\![P \,[\![\, A \,]\!]\, Q]\!] =$$
$$\{tr_p, tr_q, tr : \operatorname{seq}\Sigma;\ ref_p, ref_q, ref : \mathbb{P}\,\Sigma \mid$$
$$(tr_p, ref_p) \in \mathcal{F}[\![P]\!] \land (tr_q, ref_q) \in \mathcal{F}[\![Q]\!] \land tr \in tr_p \,[\![\, A \,]\!]\, tr_q \land$$
$$ref_p \setminus (A \cup \{\checkmark\}) = ref_q \setminus (A \cup \{\checkmark\}) \land ref = ref_p \cup ref_q \bullet (tr, ref)\}$$

$$\mathcal{F}[\![P \,[\![\, A \mid B \,]\!]\, Q]\!] =$$
$$\{\, tr : \operatorname{seq}\Sigma;\ ref_p, ref_q, ref : \mathbb{P}\,\Sigma \mid$$
$$\operatorname{ran} tr \subseteq A \cup B \cup \{\checkmark\} \land (tr \upharpoonright A, ref_p) \in \mathcal{F}[\![P]\!] \land (tr \upharpoonright B, ref_q) \in \mathcal{F}[\![Q]\!] \land$$
$$ref \cap (A \cup B \cup \{\checkmark\}) = (ref_p \cap (A \cup \{\checkmark\})) \cup (ref_q \cap (B \cup \{\checkmark\})) \bullet (tr, ref)\,\}$$

$$\mathcal{F}[\![P \setminus A]\!] =$$
$$\{tr : \operatorname{seq}\Sigma;\ ref : \mathbb{P}\,\Sigma \mid (tr, A \cup ref) \in \mathcal{F}[\![P]\!] \bullet (tr \upharpoonright (\Sigma \setminus A), ref)\}$$

$$\mathcal{F}[\![P \triangle Q]\!] =$$
$$\{tr_p, tr_q : \operatorname{seq}\Sigma;\ ref : \mathbb{P}\,\Sigma \mid$$
$$(tr_q = \langle\rangle \Rightarrow (tr_p, ref) \in \mathcal{T}[\![P]\!]) \land$$
$$(tr_q \neq \langle\rangle \Rightarrow \checkmark \notin \operatorname{ran} tr_p \land tr_p \in \mathcal{T}[\![P]\!]) \land (tr_q, ref) \in \mathcal{F}[\![Q]\!] \bullet (tr_p \frown tr_q, ref)\}$$

$$\mathcal{F}[\![P \,\Box\, Q]\!] =$$
$$\{ref : \mathbb{P}\,\Sigma \mid (\langle\rangle, ref) \in \mathcal{F}[\![P]\!] \cap \mathcal{F}[\![Q]\!] \bullet (\langle\rangle, ref)\} \cup$$
$$\{tr : \operatorname{seq}\Sigma;\ ref : \mathbb{P}\,\Sigma \mid tr \neq \langle\rangle \land (tr, ref) \in \mathcal{F}[\![P]\!] \cup \mathcal{F}[\![Q]\!] \bullet (tr, ref)\}$$

$$\mathcal{F}[\![P \sqcap Q]\!] = \mathcal{F}[\![P]\!] \cup \mathcal{F}[\![Q]\!]$$

$$\mathcal{F}[\![P \,{}^{\S}_{9}\, Q]\!] =$$
$$\{tr_p, tr_q : \operatorname{seq}\Sigma;\ ref : \mathbb{P}\,\Sigma \mid$$
$$\checkmark \notin \operatorname{ran} tr_p \land tr_q = \langle\rangle \land (tr_P, ref \cup \{\checkmark\}) \in \mathcal{F}[\![P]\!] \lor tr_p \frown \langle\checkmark\rangle \in \mathcal{T}[\![P]\!] \land$$
$$(tr_q, ref) \in \mathcal{F}[\![Q]\!] \bullet (tr_p \frown tr_q, ref)\}$$

$$\mathcal{F}[\![e \to P]\!] =$$
$$\{tr : \operatorname{seq}\Sigma;\ ref : \mathbb{P}\,\Sigma \mid tr = \langle\rangle \land e \notin ref \lor head(tr) = e \land (tail(tr), ref) \in \mathcal{F}[\![P]\!]\,\}$$

$$\mathcal{F}[\![Stop]\!] = \{ref : \mathbb{P}\,\Sigma \bullet (\langle\rangle, ref)\}$$

$$\mathcal{F}[\![Skip]\!] = \{tr : \operatorname{seq}\Sigma;\ ref : \mathbb{P}\,\Sigma \mid (tr = \langle\rangle \land \checkmark \notin ref) \lor tr = \langle\checkmark\rangle\}$$

Using the stable failures semantics, we can reason about liveness properties of CSP processes. When considering liveness properties, we are interested to know whether CSP processes are able to perform some specific useful activity, that is, that they cannot refuse behaviour specified by their liveness specification. Specifically process Q failures-refines process P if every failure of Q is also a failure of P.

$$P \sqsubseteq_{\mathcal{F}} Q \Leftrightarrow \mathcal{T}[\![P]\!] \supseteq \mathcal{T}[\![Q]\!] \land \mathcal{F}[\![P]\!] \supseteq \mathcal{F}[\![Q]\!]$$

For example let process $P = a \to b \to Stop$ be a liveness specification, then process $Q = a \to Stop$ would not satisfy this requirement; that is, Q does not *failures-refine* P. This is because while process P cannot refuse performing task b after a, Q terminates after a and so refuses everything, including b. Conversely, consider the process $R = a \to Stop \sqcap b \to Stop$ to be a liveness specification; it states that the process must either perform task a or b, the choice of which is nondeterministic. Therefore any process satisfying this must not be able to refuse the execution of both a and b. In this case Q would satisfy it, that is, it failures-refines R.

Similar to traces refinement, two processes are failures equivalent if and only if they failures-refine each other:

$$P \equiv_{\mathcal{F}} Q \Leftrightarrow Q \sqsubseteq_{\mathcal{F}} P \land P \sqsubseteq_{\mathcal{F}} Q$$

Note that we do not use the failures-divergences model in this thesis because we do not consider divergent behaviour in workflow processes.

2.4.3 Step-wise Refinement

Under the failures refinement, we say process P is deadlock-free if it failures-refines the characteristic process satisfying deadlock freedom. A process P is deadlock-free if none of its failures has the form $(s, \Sigma \cup \{\checkmark\})$, where $s \in traces(P)$ such that s is not of the form $t \frown \langle\checkmark\rangle$, that is, after every non terminating trace, it must not refuse all events. The characteristic deadlock free process, labelled DF, is defined as follows:

$$DF = (\sqcap e : \Sigma \bullet e \rightarrow DF) \sqcap Skip \tag{2.1}$$

and the deadlock freedom assertion on process P may be expressed as the following refinement.

$$DF \sqsubseteq_{\mathcal{F}} P$$

In fact, under both traces and failures refinements, the *characteristic* process of the specification is the most nondeterministic process satisfying the specification. CSP refinements permit a process to be a specification as well as a model of implementation. Another property that comes for free with both refinement orderings is *transitivity*, which means one could move gradually from specification to an acceptable implementation, a series of processes $P_1 \,..\, P_n$ such that P_{i+1} is a refinement of P_i for $i \in \{1 \,..\, n-1\}$.

$$P_1 \sqsubseteq \,..\, \sqsubseteq P_n$$

This is known as *step-wise* refinement [Wir71]. In general refinement assertions made on CSP processes with finite states may be automatically verified via model checking using the FDR tool [For98] for example. FDR is an explicit exhaustive finite-state exploration tools and has been used extensively in industrial applications [Law05, Cre05].

2.4.4 Implementation

As part of the development of our framework, we provide a Haskell implementation of our semantic definition, so that it is amenable to application. In this section we define a Haskell data type for a subset of CSP processes to which we define the semantic functions from the syntactic description of BPMN diagrams. We also present a Haskell function which maps this data type to the machine-readable version of CSP (CSPm) [Ros98]. This version allows CSP models to be fed into the FDR tool for refinement checks. This mapping corresponds to the module *To CSPm* in the framework shown in Figure 1.1. The derivation of the Haskell data type of the abstract syntax for BPMN is presented in Chapter 4.

Figures 2.4 and 2.5 contain the Haskell definition that captures a finite state subset of CSP processes. A finite state subset of CSP is sufficient since we consider business processes to be finite state systems. This is suitable as finite state models are more amenable to automatic refinement checks. Furthermore, we consider types `Event`, `Var` and `SetName` to be type synonyms of `String`, a predefined Haskell data type for string values. This is appropriate since each CSPm definition is supplied as ASCII text. In this thesis we use CSP and CSPm interchangeably.

```
data EventsType = Set | Seq
data Events = SName SetName | List EventsType [Event]
data Process =
    Stop | Skip | Prefix Event Process | Extern Process Process | ProcId ProcVar |
    Intern Process Process | Inter Process Process | Hide Process Events |
    Interrupt Process Process | SeqComp Process Process |
    Parcomp (Events,Process) (Events,Process) | Parinter Process Events Process |
    Indseqcomp (Var,Events) Process | Indextern (Var,Events) Process |
    Indintern (Var,Events) Process | Indparcomp (Var,Events) Events Process |
    Indinter (Var,Events) Process | Indparinter (Var,Events) Events Process
```

Figure 2.4: Haskell representation of CSP processes, Event sets and sequences

Each value of `Events` captures either a set or a sequence of CSP events. This is achieved either by enumerating its value via the constructor `List` that takes a `EventType` value and a list of `Event`, or

by referencing an existing set or sequence via the constructor `SName` that takes a `SetName` value as the reference. A value of `EventType` specifies the enumeration to be either a set (`Set`) or a sequence (`Seq`).

The data type `Process` captures the basic grammar of CSPm, where constants `Stop` and `Skip` denote the processes *Stop* and *Skip*; the constructor `Prefix` denotes the prefix operator ($\rightarrow$); `Extern` denotes the external choice operator ($\Box$); `Intern` denotes the internal choice operator ($\sqcap$); `Inter` denotes the interleaving operator ($|||$); `Hide` denotes the hiding operator ($\setminus$); `Parcomp` denotes the parallel composition operator ($[\![X \mid Y]\!]$); `Interrupt` denotes the interrupt operator ($\triangle$); `Parinter` denotes the partial interleaving operator ($[\![X]\!]$); `SeqComp` denotes the sequential composition operator ($\fatsemi$); and `ParId` denotes a reference to a process denoted by the name of type `ProcVar`. The constructors `Indseqcomp`, `Indextern`, `Indintern`, `Indparcomp`, `Indinter` and `Indparinter` are indexed versions of the sequential composition, external choice, internal choice, parallel composition, interleaving and partial interleaving operators respectively.

```
data DataType = DList DataName [String] | DSet DataName Events
data Data = DN DataName | DS Events
data Channel = NData [String] | TData [String] [Data]
data Local = LP (ProcVar,[Local],Process) | LS (SetName,Events)
data Model = T | F | FD | R
data Specification = Deter Model Process | Refine Model Process Process |
                     Deadlock Model Process | Livelock Process
data Script = Script [DataType] [Channel] [(ProcVar,[Local],Process)]
                     [(SetName,Events)] [Specification]
```

Figure 2.5: Haskell implementation of a finite subset of CSPm

Complete CSPm scripts are captured by the data type `Script`, shown in Figure 2.5. The constructor `Script` takes five arguments. The first and second arguments record data type and channel definitions used in CSP models as required by CSPm; they are recorded by the data types `DataType` and `Channel` respectively. A value of `DataType` captures either a named type or a data type in CSPm, while `Channel` captures channels in CSPm. Note that in CSPm a simple event e is considered a separate channel whereas a compound event $e.x$ is considered as channel e with value x of either a named type or a data type.

At implementation level we consider `DataName` to be a type synonym to `String`. The third argument defines a list of CSP processes, each of which is a triple where the first component of type `ProcVar` is the process name; the second component records a list of *local definitions*, of type `Local`; and the third component of type `Process` defines the process. The definitions recorded by the second component are local to its triple. A value of `Local` records either a process, a set or a sequence. The fourth component of `Script` is a list of pairs, each of which records either a set or a sequence that is global to the CSP model. The fifth component of `Script` is a list of `Specification`, which records a list of refinement assertions. For example, Figure 2.6 shows the Haskell representation of CSP definitions in Figure 2.7(a) and the corresponding CSPm translation in Figure 2.7(b).

```
Script [] [(NData ["a","b"])]
       [("A",[],Prefix "a" Stop),("B",[],Intern (Prefix "a" Stop) (Prefix "b" Stop)),
        ("C",[],Parinter (ProcId "A") (SName "S") (ProcId "B") ]
       [("S",List Set ["a","b"])] [Refine F (ProcId "B") (ProcId "A")]
```

Figure 2.6: An example of representing CSP definitions in Haskell

2.5 Summary

In this chapter we have provided an informal description of BPMN, and introduced a running example to be used throughout this thesis; we have given a brief introduction to Haskell, which we use to implement our semantic and transformation functions; we have presented the syntax and the semantics of CSP, which we use to define the semantics of BPMN, and Z [Spi92], which we use to formalise the syntax of BPMN.

$S = \{ a, b \}$
$A = a \rightarrow Stop$
$B = a \rightarrow Stop \sqcap b \rightarrow Stop$
$C = A \llbracket S \rrbracket B$
$B \sqsubseteq_{\mathcal{F}} A$

(a)

```
channel a,b
S = {a,b}
A = a -> STOP
B = a -> STOP |~| b -> STOP
C = A [| S |] B
assert B [F= A
```

(b)

Figure 2.7: An example of representing CSP definitions (a) in CSPm (b)

Chapter 3

Related Work

This chapter describes related research on modelling and formalising workflows. We consider those in both business and empirical domains.

In the business domain, with the increase in the complexity of business logic and the need for integrating software components, research effort has been directed to specification and verification of workflows.

The empirical domain may be subdivided into two areas: *in silico* (scientific) and clinical. In scientific workflow communities, more recent research effort has been focusing on the application of workflow technology to "in silico" scientific experiments. The aim is to provide technologies for precise specification and efficient execution of scientific experiments that require analysis of large volumes of data [LAB+06, DBG+04, OAF+04, CGH+06, BJA+08].

In the clinical domain, related research effort has been directed towards standardising clinical trial protocol structure and data management. It enables data sharing and automatic generation of software systems for data analysis and trial management [BCD+05, Cal06, Kus06]. There is also research work focusing on authoring trial protocols and verifying protocols correctness against safety medical properties [HSW95, HS96, MH03], and there is research focusing on the modelling, the automation and the correctness of clinical guidelines [PTB+03, PRO].

The rest of this chapter is structured as follows: in Section 3.1 we present related work in the area of specification and verification of business workflows; in Section 3.2 we present related work in the application of workflow systems in the life science community; and in Section 3.3 we describe related work in the modelling and the verification of clinical guidelines as well as the critiquing and standardisation of trial protocols. We give a more detail comparison of related works on formalising BPMN in Chapter 9.

3.1 Business Workflow

In the service-oriented computing paradigm, service components are *stateless* and *loosely coupled* which encourages re-usability. Standards bodies have proposed several specification and description languages for services *orchestration* and *choreography*.

Orchestration describes local, single participant viewpoint of the workflow model where services and components are composed. Notable is the Business Process Execution Language for Web Services (BPEL) [BPE03], which is an XML-based language designed for describing orchestration of services. Services orchestration described in BPEL may then be executed on a BPEL engine. A related topic to service orchestration is graphical task coordination, where activities in the workflow can be either manual or automated. While graphical task coordinations provide a richer syntax for expressing workflows than text-based orchestration languages like BPEL, they are generally not executable.

Choreography describes the collaboration between multiple workflows and elevates models of workflow behaviour to a global viewpoint such that the first class entity is external, observable behaviour. There exist two complementary approaches to choreography: to specify choreographies using an *interaction* model or to specify using an *interconnection* model. An interaction model explicitly describes the order of the interactive behaviour of a workflow collaboration, while an interconnection model specifies only the interaction point between workflows; the order in which these interactions take place is implied by the behaviour of individual workflows. One of the workflow languages that model service-based choreographies using interaction models is the Web Services Choreography Description Language (WS-CDL) [KBR+05], while one of workflow languages that model service-based choreographies using interconnection models is the Web Services Choreography Interface (WSCI) [W3C02]. Note that unlike orchestration, choreography takes a global viewpoint and is not executable.

Both orchestration and choreography languages focus on a particular viewpoint of workflow definitions and are primarily used for specifying (web) service compositions. They are commonly XML-based and

vendors have introduced customised graphical notations to assist workflow modelling [ORA, BIZ, ACT]. There are also workflow notations that extend graphical task coordination and provide a more holistic view of orchestration and choreography. BPMN, of which formalisations and applications are the focus of this thesis, is one such modelling notation. Unlike BPEL, WS-CDL and WSCI, BPMN is graphical and may be used to describe both orchestrations and choreographies. BPMN specifies choreographies using an interconnection model and is not designed to be executable. However, the BPMN specification document [OMG08, Appendix A] suggests how to map BPMN models to BPEL processes for execution, and recent research work [RM06, OvdADtH06] suggested this translation is partial and provided a formal account on the syntactic and semantic boundary within which BPMN models may be translated to BPEL processes.

The rest of this section considers existing workflow languages and their formalisations, and provides comparison against research contributions described in this thesis. Section 3.1.1 considers workflow languages for orchestration and related graphical task coordinations, and Section 3.1.2 considers workflow languages for choreography as well as graphical modelling notations with a holistic view of workflow definitions. We relegate the presentation of related work on BPMN to Chapter 9, where a more detailed and focused discussion is given.

3.1.1 Orchestration

The notion of orchestration is relatively recent, in which services are invoked in explicit order. BPEL is one of the prominent languages for describing orchestrations. BPEL processes can be complex, and it is necessary to formalise its semantics for reasoning about its behaviour against desired properties. Formalisations of BPEL are described in Section 3.1.1.1.

BPEL focuses on executable service-based orchestration; it exists primarily in the context of service-oriented computing. While BPEL is XML-based and does not focus on graphical representation, there are workflow languages that are not executable but are graphical. These languages focus on expressivity, and can be used to specify workflows that coordinate a mixture of manual tasks and automated functions. Formalisations of graphical languages are described in Section 3.1.1.2.

3.1.1.1 BPEL

BPEL has been given several behavioural semantics. One of the earlier models was provided by Koshkina [Kos03], who defined the BPE-calculus to express Web service orchestration. Model checking and preorder checking for deadlock freedom can be carried out on BPE-calculus processes using the verification tool CWB [CS96].Another formalisation is defined by Foster [Fos06], who provided a translation from BPEL to Finite States Processes (FSP) [MK99]. FSP is a process algebra, whose syntax is closely related to CSP; it has an operational semantics, using which behavioural equivalences are defined in terms of bisimulation. Both Foster's and Koshkina's approaches focused on tool support for behavioural analysis, while they did not considered the compositionality of their semantics. While Koshkina did not consider behavioural compatibility between BPEL processes, Foster provided a definition of behavioural compatibility in terms of deadlock freedom of interacting BPEL processes but did not consider its compositionality. Neither of their formalisations considers timing information.

3.1.1.2 Graphical Task Coordination

While BPEL is block-structured, workflows described using graphical task coordination languages may have a graph structure. While workflows specified by these languages may not be executable, it is important that they are not ambiguous. There are mainly two different approaches to address this issue: either to provide a formal semantics to an existing modelling notation such as BPMN, or to define a new modelling notation with a formal semantics; we defer the discussion of BPMN to Chapter 9.

One of the earliest formalisations of existing modelling notations has been given by ter Hofstede and Barros et al. [tHN93, BtH97, BtH99]. They provided a formal semantics to task structure diagrams for describing workflows that coordinate concurrent tasks. Their formalisation maps task structure diagrams to the process algebra ACP [BK85] and use ACP to prove behavioural equivalence between diagrams.

Other formalisations of existing modelling notations include formalisations [Esh02, BD00] of different versions of UML activity diagrams [UML04]. Eshuis [Esh02] provided two timed semantics on the syntax

of UML activity diagrams in Clocked transition systems [MP96]. These semantics are different in that one assumes perfect synchrony hypothesis and the other one does not. The perfect synchrony hypothesis states that a system must react immediately to external events and that the corresponding output must occur at the same time. The former model is more amenable to formal verification by model checking, while the latter model is more realistic.

Bolton and Davies provided a formalisation [BD00] of UML activity diagrams in CSP. The aim of this semantic model was to formally relate each UML activity diagram to its object model using CSP refinement orderings.

3.1.2 Choreography

This section considers related work on formal approaches to choreography. Section 3.1.2.1 presents related work on formalising existing choreography description languages, and Section 3.1.2.2 presents related work on formalising behavioural compatibility.

3.1.2.1 Choreography Description Languages

In this section we consider the formalisations of two choreography description languages – WS-CDL [KBR+05] and WSCI [W3C02]. Both languages are XML-based, based on the web service standards and require a separate language such as BPEL for describing orchestration. WS-CDL models choreography based on the interaction model, and WSCI models choreography based on the interconnection model.

Brogi et al. [BCPV04] provided a mapping from a subset of WSCI to the process algebra CCS, omitting the syntactic constructs in WSCI for defining transactional choreography. They suggested how this mapping may be used to study notions of compatibility for web services choreography by extending their earlier definition of behavioural compatibility for software architectures [CPT01].

WS-CDL is developed by W3C's WS-CDL Working Group [W3C] and several semantic models have been defined and studied. Notable models include Carbone et al.'s [CHY07] and Yang and Zhao et al.'s [ZYQ06, YZQ+06]. Carbone et al. studied a global calculus, which has its origin in WS-CDL, for describing global interaction behaviour between participants in a collaboration, and an end-point calculus, which is a typed π-Calculus, for modelling the local behaviour of each participant; the π-Calculus is a process algebra developed by Milner et al. as an advance over CCS [Mil89] to express mobility in concurrent processes [Mil99] and the basic notion of this calculus is channel passing between processes. The main result of Carbone et al.'s work is a theory of end-point projection, using which one could map a global description specified in global calculus to its end-points preserving session types and behaviour. In Yang and Zhao et al.'s formalisation, a single calculus CDL is provided as a semantic domain of WS-CDL. They provided an operational semantics to WS-CDL using this calculus. Using this calculus, WS-CDL descriptions may be translated to Promela, and model checked using the SPIN model checker [Hol03] against behavioural properties specified in Linear Temporal Logic [MP92]. While Yang and Zhao et al. discussed how to develop end-point projections using their calculus, they did not provide a formal account for this in their paper [ZYQ06].

3.1.2.2 Related Formal Approaches to Choreography and Compatibility

Related formal approaches in the context of choreography include the foundational study of compatibility, compliance and realisability.

In this thesis, we formalise behavioural compatibility as a (failures) refinement-closed property in the context of collaborations between BPMN processes. In particular, two processes are compatible if neither one of the processes may cause the other process to deadlock. Related studies of compatibility include Foster et al.'s [FUMK04, FUMK06] model-based approach, in which they translate both BPEL and WS-CDL into (parallel compositions of) finite states processes (FSP). General liveness behavioral properties such as deadlock freedom may be verified using a model checker. More specific properties such as *obligation* [FUMK06], which describes what activities a subject must or must not do to a set of target objects, require high-level specification of the corresponding policy using Message Sequence Charts [MSC96] (MSCs). MSCs are translated into FSPs, and verification can be carried out by showing behavioural equivalence between the respective FSP processes via model checking.

A stronger notion of compatibility is known as compliance. Bravetti and Zavattaro [BZ07b, BZ07a] formalised the notion of *strong service compliance*. A composition of services is strongly compliant if their composition is both deadlock and livelock free, and whenever one service is to initiate an interaction with another service (via messaging), this other service must be prepared to engage. They then further developed this formal notion by considering *service refinement*, in which they consider a suitable pre-order between services such that substitutions of individual services in a composition by their refinements preserve compliance. Note that in their paper [BZ07a], they use the term *strong subcontract pre-order* instead of service refinement, as services are modelled as contracts.

A related notion to end-point projection is *realisability*. Salaün and Bultan [SB09] defined realisability to indicate whether participants can be generated from a choreography such that they will behave exactly as formalized in its specification. If the specification is not realisable, they provide a technique for extending the behaviour of participants to realise the choreography. They use collaboration diagrams (called communication diagrams in UML [UML04]) for the specification of choreography and provide an encoding of the diagram's abstract syntax into LOTOS process algebra [LOT89]. Realisability of choreography is achieved by first generating participants (peers) in LOTOS via projection and then checking equivalence between the labelled transition system (LTS) of a LOTOS process describing the choreography and the LTS of a parallel composition of LOTOS processes, each modelling a projection in the choreography. If they are not equivalent, that is, the choreography is not realisable, additional interaction behaviours between participants are inserted to realise the choreography. Note that in their formalisation they consider both synchronous and bounded asynchronous interactions.

3.2 Scientific Workflows

Workflows in the scientific community have been used as a scalable mean to streamline the execution of *in silico* scientific experiments that process massive amount of data. While this type of workflow usage is not the focus of this thesis, we will nonetheless give a brief overview of related work in this area. Recent development of scientific workflow systems include Ludäscher et al.'s Kepler system [LAB+06]; Oinn et al.'s Taverna for building and executing bioinformatics workflows [OAF+04]; Barga et al's Trident scientific workflow workbench [BJA+08]; Churches et al.'s Triana architecture [CGH+06], and Deelman et al.'s Pegasus system for implementing scientific workflows in a Grid environment [DBG+04]. These systems aim to ease the process of integrating existing scientific applications through abstraction and encapsulation and to improve the implementation of data flow and the distribution of resources.

Ludäscher et al.'s Kepler system [LAB+06] provides support for execution of scientific experiments in the areas of bioinformatics, ecoinformatics and geoinformatics. The system has been implemented in Java [AGH05] and it comes with a graphical user interface, for constructing workflows and monitoring their executions. In particular Kepler defines two abstraction layers – *actor* and *director*. An actor forms an encapsulation layer for a wide variety of activities ranging from the instantiation of a web service operation to the execution of a Globus job [GLO]. A director, on the other hand, provides the semantics of interaction between actors and supplies objects, known as *receivers*, for implementing the communications. For example, whether the communication between two actors is buffered or synchronous is determined by the workflow's director, rather than individual actors performing the interaction. This approach provides exogenous coordination, similar to that of Reo [Arb04], which improves the reusability of actors themselves.

Oinn et al.'s Taverna tool [OAF+04] provides the mechanism for the composition and enactment of bioinformatics workflows for the life science community. Taverna contains a workbench which has a graphical user interface for the composition of workflows. A customized XML-based language, SCUFL, is provided as part of the Taverna tool, for specification of workflows. Specifically in a SCUFL workflow, each unit of activity is some form of transformation, known as a *processor*, which essentially accepts some input and produces a set of outputs. Each Scufl workflow has two types of linking, *data* and *coordination*. The former mediates the flow of data between a data source such as a processor's input, and a data sink such as a processor's output. The latter links two processors and controls their execution. For example, one processor could go from scheduled to running if another processor has the status 'complete'. Like many other scientific workflows, Taverna provides bindings of Scufl workflows to a wide range of third party applications that are used by scientists conducting experiments.

Barga et al.'s Trident scientific workflow workbench [BJA+08] is an extension of Microsoft Windows

Workflow [WWF], which is implemented as part of Project Neptune [NEP], to provide the facilities for scientists to explore and visualise oceanographic data in real-time and as well as the environment to compose, run and catalogue workflows.

Churches et al.'s Triana [CGH+06] provides an interface for the specification and composition of scientific applications. A component in a Triana workflow is then considered as a unit of execution. In Triana, a customized XML-based workflow language is provided for the specification of workflow. Triana also provides dynamic distribution mechanism for distributing a group of tasks specified in a Triana workflow across multiple machines either in parallel or in a pipeline. Similarly Deelman et al.'s Pegasus [DBG+04] is a framework for the specification of scientific workflows using a customized workflow language and the mapping of workflow specifications onto distributed resources like the Grid. Moreover, it supports various scheduling and replica selection algorithms, as well as partition-level failure recovery.

3.3 Modelling Clinical Trials and Guidelines

In Chapter 7 we consider how BPMN can be used to assist the specification and visualisation of long-running empirical studies. Moreover, our formalisations of BPMN allows models of empirical studies to be verified against behavioural properties to ensure correct execution of studies. In particular, we consider the clinical trial domain as a running example in the chapter. We also use a real clinical trial as the basis for one of our case studies in Chapter 8 to demonstrate the application of BPMN formalisations to empirical studies. In this section we describe some related research to standardise clinical trial protocols as well as to formalise clinical guidelines.

3.3.1 Clinical trial protocols

In oncology, chemotherapeutic treatments are often carried out within protocol-based clinical trials, in which data are collected to monitor the efficacy and the toxicity of treatment. This data is analysed statistically to evaluate the clinical objectives of the trial. For the trial to be scientifically valid, data must be complete and correct. To ensure the correctness of trial data, computer support has been introduced into trial management; in particular, authoring systems have been implemented to assist clinicians to construct trial protocols and to ensure information described in the trial protocols does not violate safety, structural and medical properties. Research work in the area of authoring and critiquing trial protocols includes Modgil and Hammond [MH03]'s Design-a-Trial. This is a decision support tool for critiquing the data supplied for trial specification based on expert knowledge, and subsequently outputting a protocol describing the trial.

More recently, informatics research in oncology has been directed towards standardising cancer clinical trials to ensure trial protocols are CONSORT-compliant [MSA01], and improving data sharing in cancer clinical trials by consistent use of standardised common data elements (CDE) and controlled vocabularies. Notable results include those from the CancerGrid project [CAN], whose aim was to develop open standards based solutions for clinical cancer informatics.

3.3.2 Clinical guidelines

Clinical guidelines are used in clinical care to reduce proneness to errors during the treatment of specific diseases [GTM+04]. However, there are two major issues surrounding the application of clinical guidelines. First, information contained in conventional text-based guidelines are difficult to access, modify and apply to patients during the consultation. Second, text-based guidelines are inherently ambiguous and may be incomplete, which could jeopardise heavily the quality of guidelines. To this end, research efforts have been directed to these two areas. Results in the first area include Bury, Fox, Sutton et al's PROforma [BFS00, SF03] language for authoring, publishing and executing clinical guidelines and Ciccaresea et al.'s GUIDE project [CCQS05, QSSF97], which provides a modelling language for integrating clinical guidelines with organisational workflows. Research work in the area of formalisation and verification of clinical guidelines includes the Protocure project [PRO]. In this project clinical guidelines are modelled using a time-oriented machine readable language, Asbru [MSJ96], and guideline models can be verified against medical properties via interactive theorem proving using Karlsruhe Interactive Verifier (KIV) [Rei95].

In comparison, this thesis considers a wider class of empirical workflows, but considers a smaller set of requirements, namely untimed temporal properties. While verifying clinical guidelines against general medical properties requires interactive theorem proving, we demonstrate in a case study in Chapter 8, that the verification of clinical workflows against oncological properties can be achieved by a combination of compositional reasoning and model checking.

3.4 Summary

In this chapter we considered the current research in formalising workflow systems. In particular, in Section 3.1 we studied formal approaches for reasoning with service orchestration and choreography. This included a survey of the formalisations of several XML-based workflow specification languages as well as graphical workflow modelling notations. In Sections 3.2 and 3.3, we gave an overview of related research in scientific and clinical workflows.

Chapter 4

BPMN Syntax

In this chapter we provide a formal specification of the abstract syntax of BPMN described in Section 2.1. The specification is provided in the language of Z. Figure 4.1 shows the specification's corresponding implementation in Haskell. We do not show the definition of all data types and type synonyms but only those that help to illuminate the formal specification of BPMN abstract syntax. For example, `Seqflow` and `Mgeflow` are type synonyms for the Haskell built-in data type `String`. The Haskell implementation provides the necessary type definitions for recording BPMN diagrams. We use it as the domain to implement the two semantic functions, which are formally defined in Chapters 5 and 6. Nevertheless, the Haskell implementation does not lend itself to specify constraints on data values, or the state space. For this reason, we turn to the language of Z. For example, the constructor `Compound` of data type `Element` is used to record subprocesses. The constructor takes as one of its arguments a list of `Element` values to record the BPMN elements contained in the subprocess. While the definition of data type `Element` permits any `Element` value to be in this list, BPMN official documentation describes some syntactic constraints between these elements, and we would like to formally specify these constraints. Throughout this chapter, we refer to both Haskell and Z definitions; we use the `typewriter` font when referring to Haskell expressions and the *math* font when referring Z expressions.

```
data Loops = Fix Int | Ndet Int
data FlowType = One | All
data Exception = Exception ErrorCode | AnyException

data Type = Itime Int | Stime Int | Ierror Exception | Eerror Exception |
            Srule BCondition | Irule BCondition | Start | End |
            Smessage (Maybe Mgeflow) | Imessage (Maybe Mgeflow) | Emessage (Maybe Mgeflow) |
            Agate | Xgate | Exgate | Task TaskName | SubProcess BName |
            Miseq TaskName Loops FlowType | Miseqs BName Loops FlowType |
            Mipar TaskName Loops FlowType | Mipars BName Loops FlowType

type Range = (Int,Int)

data Atom = Atom {etype :: Type, ins,outs :: [Seqflow],
                  exit :: [(Seqflow,Type)], range :: Range, receive, send :: [Mgeflow]}

data Element = Atomic Atom | Compound Atom [Element]
type Diagram = [(PoolId,[Element])]
```

Figure 4.1: Abstract syntax of BPMN subset in Haskell

This chapter is structured as follows. In Section 4.1 we provide some preliminary definitions that are common to all BPMN elements. In Sections 4.2, 4.3 and 4.4 we provide a formal specification of this syntax using the schema calculus that precisely describes the constraints on the types of BPMN elements and the relationship between them. In Section 4.5 we investigate the initialisation of BPMN elements, pools and diagrams. In Section 4.6 we present a set of syntactic operations for constructing BPMN processes. These operations are defined using the schema calculus and provides a syntactic vehicle for investigating our semantic models in Chapter 5. Note that some auxiliary definitions for the formalisation are only partially defined in this chapter. Full definition can be found in Appendix A.

4.1 Preliminaries

This section presents a specification of the BPMN syntax that is common to all BPMN elements. We first consider some modelling decisions about connecting objects.

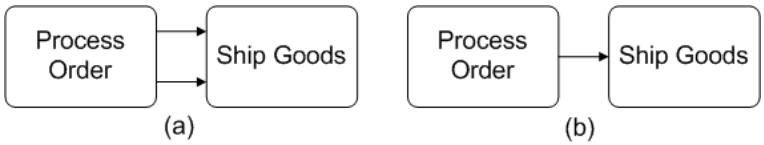


Figure 4.2: Two activities connected by (a) two identical sequence flows and (b) a single sequence flow

Specifically, we assume no two connecting object may have the same source and target. While this is not made explicit in the official specification, it is consistent with our view. For example, Figure 4.2(a) shows two task elements connected by two sequence flows. Since our semantic model abstracts from conditions and tokens, each connecting object is identified solely by its source and target elements. As a result it is not possible to distinguish between them, therefore one of the flows is redundant and can be removed as shown in Figure 4.2(b). The same applies to message flows since our model abstracts from message content.

While each BPMN element is identified as the source and/or target flow objects of some connecting object, we take an equivalent view of recording each flow object's incoming and outgoing sequence and message flows.

4.1.1 Haskell Syntax

A BPMN element in a BPMN pool is either atomic or compound. Atomic elements are event, task and gateway elements, while compound elements are subprocess elements that contain a collection of other BPMN elements. We define the data type `Element` to capture BPMN elements It has two constructors: `Atomic` for recording an atomic element, and `Compound` for recording a compound element. `Atomic` takes an `Atom` value that records the syntax of an atomic element such as the element's type, its sequence and message flows, and `Compound` takes an `Atom` value as well as a list of `Element` values that records the collection of elements the compound element contains.

4.1.2 Z Specification

Each element of type `Element` takes a value of type `Atom`. `Atom` has a single constructor `Atom` that takes eight parameters, and implements the following schema *Atom*.

Atom

$$
\begin{array}{l}
type : Type \\
in, out : \mathbb{F}\, Seqflow \\
exit : Seqflow \nrightarrow Type \\
range : Range \\
send, receive : \mathbb{F}\, Mgeflow \\
\hline
\mathsf{disjoint}\ \langle in, out, \mathrm{dom}\ exit \rangle \\
send \cap receive = \emptyset
\end{array}
$$

Specifically, component *type* has the type *Type*. It records the type of an element and is implemented by the Haskell field `etype`. The type *Type* is defined as a free type of the following form:

$$Type ::= itime\langle\langle \mathbb{N} \rangle\rangle \mid ... \mid imessage\langle\langle Message \rangle\rangle$$

where each constructor in *Type* is implemented by a constructor in the Haskell data type `Type`. For example, the constructor *imessage* is implemented by the constructor `Imessage` and records the type of

an intermediate message event. The free type *Message* records the option of having a message or not.

$$Message ::= message\langle\langle Mgeflow \rangle\rangle \mid nomessage$$

Components *in* and *out*, which are implemented by `Atom`'s fields `ins` and `outs`, record an element's incoming and outgoing sequence flows respectively. Similarly, components *receive* and *send* are implemented by `Atom`'s fields `receive` and `send` and record an element's incoming and outgoing message flows. Sequence flow and message flow are defined as basic types *Seqflow* and *Mgeflow*. Component *exit* records exception flows of an element and is implemented by `Atom`'s field `exit`, while component *range* records an element's timing information and is implemented by `Atom`'s field `range`.

The constraint part of schema *Atom* specifies that an element's incoming sequence flows, outgoing sequence flows and exception flows must be disjoint. This is because our semantic definition treats sequence flows to be internal to each BPMN diagram and cyclic elements would therefore lead to divergent behaviour. The same applies to its incoming and outgoing message flows.

The Haskell data type `Element` provides an implementation for the following free type *Element*.

$$Element ::= atomic\langle\langle Atom \rangle\rangle \mid compound\langle\langle Atom \times \mathbb{F}_1\, Element \rangle\rangle$$

To assist the specification we define the function *atom*, which takes a value of type *Element* and returns the first parameter of its constructor function.

$$atom : Element \rightarrow Atom$$

4.2 Events and Gateways

This section presents a specification of the syntax of BPMN events and gateways.

4.2.1 Haskell syntax

In our Haskell syntax in Figure 4.1, `Type` provides various constructors to record information about events and gateways. For message events, `Type` provides constructors `Smessage`, `Imessage` and `Emessage` to record the type of a start, an intermediate and an end message event respectively, each of which takes a value of type `Maybe Mgeflow` to record an optional message flow. For timer events, `Type` provides constructors `Stime` and `Itime` to record the type of a start and an intermediate timer event respectively, each of which takes a parameter of type `Int` to record a time duration. For rule events, `Type` provides constructors `Srule` and `Irule` to record the type of a start and an intermediate rule event respectively, each of which records a rule with a parameter of type `BCondition`. For non-trigger events, `Type` provides constructors `Start` and `End` to record the type of a start event and an end event respectively. For error events, `Type` provides constructors `Ierror` and `Eerror` to record the type of an intermediate and an end error event respectively, each of which records an optional error code with a parameter of type `Exception`. The data type `Exception` provides two constructors: `Exception` with a `String` parameter for recording the error code, and the nullary constructor `AnyException` representing no specific error code. For gateways, `Type` provides constructors `Xgate`, `Exgate` and `Agate` to record the type of a data-based XOR, an event-based XOR and an AND gateway respectively.

4.2.2 Formal Specifications

We define schema *NonActivity* to specify the common constraints on the syntax of an event or a gateway. We write $\#S$ to denote the size of set S.

NonActivity

$$ele : Element$$

$$ele \in \operatorname{ran} atomic$$
$$\#(atom\ ele).exit + \#(atom\ ele).send + \#(atom\ ele).receive = 0$$
$$first(atom\ ele).range = 0$$
$$second(atom\ ele).range = 0$$

Specifically, schema *NonActivity* states that all events and gateways are atomic, have no exception and message flow recorded in components *exit*, *send* and *receive*, and have zero duration recorded in component *range*. Each message event records its associated message flows using the parameter of its *type*'s constructor (e.g. *imessage*). Each timer event records its time duration using the parameter of its *type*'s constructor.

A start event has one or more outgoing sequence flows [OMG08, Section 9.3.2.3], where each flow "generates a separate parallel path"; a start event cannot have any incoming sequence flow. In our subset of BPMN, a non-gateway element with multiple incoming and outgoing sequence flows may be transformed into one with only one incoming and one outgoing sequence flow using appropriate *gateways*. For this reason, in our subset of BPMN a start event has *exactly one outgoing sequence flow*. We specify these constraints using the schema *Start*.

Start

$$NonActivity$$

$$(atom\ ele).type \in \{start\} \cup \operatorname{ran} stime \cup \operatorname{ran} smessage \cup \operatorname{ran} srule$$
$$\#(atom\ ele).in = 0 \land \#(atom\ ele).out = 1$$

The value *start*, of type *Type*, records the type of a non-trigger start event. Function constructors *stime*, *smessage* and *srule*, also of *Type*, record the types of a start timer, a start message and a start rule event respectively.

An end event may have one or more incoming sequence flows, and these flows may form alternate or parallel paths [OMG08, pages 42 – 43]. An end event cannot have any outgoing sequence flow. In our syntax, an end event has one incoming sequence flow; end events with multiple incoming sequence flows may be transformed using appropriate gateways. We specify these constraints using the schema *End*.

End

$$NonActivity$$

$$(atom\ ele).type \in \{\ end\ \} \cup \operatorname{ran} emessage \cup \operatorname{ran} eerror$$
$$\#(atom\ ele).in = 1 \land \#(atom\ ele).out = 0$$

The value *end*, of type *Type*, records the type of a non-trigger end event. Constructors *emessage* and *eerror*, also of *Type*, record the type of an end message and an end error event respectively.

An intermediate event may be attached to an activity element's boundary, creating an exception flow [OMG08, page 44]; we postpone the discussion of this functionality to Section 4.3 where the syntax of activity elements is discussed. Otherwise, an intermediate event has one or more incoming and outgoing sequence flows. In our syntax, an intermediate event has *one outgoing sequence flow and one incoming sequence flow*. Similar to start and end events, intermediate events with multiple sequence flows may be transformed using appropriate gateways. We specify these constraints using the schema *Inter*,

$$Inter \mathrel{\widehat=} [NonActivity \mid (atom\ ele).type \in inters] \land OneInOutFlow$$

where the value *inters* is the abbreviation for the set of all intermediate event types.

$$inters == \operatorname{ran} itime \cup \operatorname{ran} imessage \cup \operatorname{ran} ierror \cup \operatorname{ran} irule$$

Constructors *itime*, *imessage*, *ierror* and *irule*, of *Type*, record the type of an intermediate time, an intermediate message, an intermediate error and an intermediate rule event respectively. The schema *OneInOutFlow* states that the element has one incoming and one outgoing sequence flow.

We define the schema *Event* to capture an event in our BPMN subset.

$$Event \mathrel{\widehat=} Start \lor End \lor Inter$$

We provide schema *Gate* to specify the constraints that are common to all types of gateways in our subset.

Gate

NonActivity

$(atom\ ele).type \in \{agate, xgate, exgate\}$
$\#(atom\ ele).in \geq 1$
$\#(atom\ ele).out \geq 1$

We consider the syntactic constraints between an event-based XOR gateway and the targets of its outgoing sequence flows in Section 4.4.

4.3 Activities

4.3.1 Haskell Syntax

An activity in BPMN is either a task, a subprocess or one of their multiple instance variants. For a task, `Type` provides the constructor `Task` to record its type. The constructor takes a task name of type `TaskName`. For a subprocess, `Type` provides the constructor `SubProcess` to record its type. The constructor takes a subprocess name of type `BName`; we assume the values of `TaskName` and `BName` are disjoint.

For multiple instance variants, `Type` provides four constructors: `Miseq` records the type of a sequential multiple instance task, and `Miseqs` records the type of a sequential multiple instance subprocess. Similarly, `Mipar` records the type of a parallel multiple instance task, and `Mipars` records the type of a parallel multiple instance subprocess. All four constructors take as parameters the name of the activity, the number of instances and also a flow condition of type `FlowType`; the data type `FlowType` takes one of values `One` and `All`: `One` specifies that the multiple instance activity element triggers its outgoing sequence flow after one of it instances has completed execution, and value `All` specifies that the element triggers its outgoing sequence flow after all of its instances have completed execution.

4.3.2 Incoming and Outgoing Flows

In the official specification [OMG08, Sections 9.4.3.9 and 9.4.3.10], tasks and subprocesses in a BPMN process may have zero incoming sequence flows if there is no start event in that process, while they may have zero outgoing sequence flows if there is no end event in that process. In our syntax, however, a BPMN process has at least one start event and one end event, therefore each task and subprocess has exactly one incoming and one outgoing sequence flow; tasks and subprocesses with multiple incoming and outgoing sequence flows may be transformed using appropriate gateways. A task may have zero or more incoming message flows and zero or more outgoing message flows. If there is more than one incoming message flow then the task will receive messages from only one of them during its execution. Conversely, if there is more than one outgoing message flow then the task will send messages to all of them before the completing its execution. While the official specification defines a subprocess element to have zero or more incoming and outgoing message flows, it is not clear as to how these message flows affect the behaviour of the subprocess [OMG08, page 62]. We therefore do not consider message flows of subprocesses and instead consider the message flows of elements contained in subprocesses.

4.3.3 Timing Information

BPMN is an extensible notation. We extend the syntax of BPMN tasks to record timing information. This timing information may be ignored when considering only the untimed behaviour. Specifically each task element is associated with a bounded duration range recorded by the constructor `Atom`'s `Range` parameter. A duration range is a pair of time values, each of type `Time`. The first component records the task's minimum execution time and the second component records the task's maximum execution time. In the Z specification of the syntax, the schema *Atom* declares the component *range* to record this duration range.

4.3.4 Formal Specification of Task

In this section we provide a Z specification of tasks. We consider exception flows of tasks in Section 4.3.6.

We define schema *GenTask*, which states that a task is atomic, that is, has the type of either a task (*task*) or one of its multiple instance variants (*miseq*, *mipar*), and that it must not have a negative duration range.

GenTask

$$ele : Element$$

$$ele \in \operatorname{ran} atomic$$
$$(atom\ ele).type \in \operatorname{ran} task \cup \operatorname{ran} miseq \cup \operatorname{ran} mipar$$
$$first((atom\ ele).range) \leq second((atom\ ele).range)$$

A schema *NormalTask* is the conjunction of *GenTask* and *OneInOutFlow*, thereby also insisting that a task has exactly one incoming and one outgoing sequence flow.

$$NormalTask \mathrel{\widehat=} GenTask \land OneInOutFlow$$

4.3.5 Formal Specification of Subprocesses

In this section we provide a Z specification of subprocesses. We consider exception flows of subprocesses in Section 4.3.6. Since both subprocesses and pools are BPMN processes, we consider the syntactic constraints on BPMN processes such that we can reuse this specification when considering BPMN pools in Section 4.4.

We first provide some preliminaries to assist the specification of the relationship between a process and the elements it contains. A BPMN process is a nonempty finite set of BPMN elements. This is denoted by the abbreviation $Process == \mathbb{F}_1\ Element$. The following function *content* relates a subprocess to the collection of elements it contains.

$$content : Element \to \mathbb{F}\ Element$$

$$\forall\, e : Element \bullet$$
$$(e \in \operatorname{ran} compound \Rightarrow content(e) = second(compound^{\sim} e)) \land$$
$$(e \in \operatorname{ran} atomic \Rightarrow content(e) = \emptyset)$$

We introduce the set *contains* such that for any elements e and f, $(e, f) \in contains$ holds if and only if f is a compound element and $e \in content(f)$.

$$contains : Element \leftrightarrow Element$$

$$\forall\, e, f : Element \bullet (e, f) \in contains \Leftrightarrow (f \in \operatorname{ran} compound \land e \in content(f))$$

As BPMN processes are hierarchically structured, it is important to distinguish between the relations $(e, f) \in contains$ and $(e, f) \in contains^+$, where $contains^+$ is the transitive closure of *contains*. We therefore introduce the concept of containment formally.

Definition 4.1. ***Containment.*** *An element e is* ***contained*** *in some element f iff $(e, f) \in contains^+$. An element e is* ***directly contained*** *in some element f iff $(e, f) \in contains$.*

For example, in our running example in Figure 2.3 on Page 10, task *Receive Invoice* is directly contained in subprocess *Receive Offer*. We write $e \in_{\mathrm{e}} f$ if and only if $(e, f) \in contains^+$. In addition for some element e and a nonempty finite set of elements *es* (*Process*), we write $e \in_{\mathrm{p}} es$ if and only if either $e \in es$, or $e \in_{\mathrm{e}} f$ where $f \in es$.

$$\_ \in_{\mathrm{e}} \_ : Element \leftrightarrow Element$$
$$\_ \in_{\mathrm{p}} \_ : Element \leftrightarrow Process$$

We also introduce the set $edge(p)$ over some process p. Specifically, $edge(p)$ is a set of pairs of elements directly contained in p such that for each pair, one of the outgoing and exception flows of its

first component is an incoming sequence flow of its second component. We write $outs(e)$ to denote the set of e's outgoing and exception flows.

$$edge : Process \rightarrow \mathbb{P}(Element \times Element)$$

$$\forall p : Process \bullet edge(p) = \{e, f : p \mid outs(e) \cap (atom\, f).in \neq \emptyset\}$$

Using this definition of *edge*, we formally introduce the notion of predecessor and successor, which are used throughout this thesis.

Definition 4.2. ***Predecessor.*** *A BPMN element e precedes another BPMN element f in a process that directly contains a set of elements P iff $(e, f) \in edge(P)^+$ holds. The element e is a direct predecessor of f with respect to p iff $(e, f) \in edge(P)$.*

Definition 4.3. ***Successor.*** *A BPMN element e succeeds another BPMN element f in a process that directly contains a set of elements P iff $(f, e) \in edge(P)^+$ holds. The element e is a direct successor of f with respect to p iff $(f, e) \in edge(P)$.*

Using the preliminary definitions we provide a Z specification of the relationships between elements contained in a BPMN process. Specifically, we capture the following properties about elements in a BPMN process:

1. Elements must not share sequence or message flows.
2. Every non-start element must be preceded by a start event.
3. Every non-end element must be succeeded by an end event.
4. Every outgoing or exception sequence flow of an element must also be an incoming sequence flow of some other element in the process.
5. Every incoming sequence flow of an element must also be either an outgoing or an exception sequence flows of some other element in the process.

We first consider Property 1. We define function *getSd* to take an *Atom* part of an element and return its outgoing message flows; outgoing message flows are responsible for sending messages. Similarly we define function *getRec* to return an element's incoming message flows; incoming message flows are responsible for receiving messages. We write $getMsg == getSd \cup getRec$ to aggregate these two functions.

$$getSd, getRec, getMsg : Atom \rightarrow \mathbb{F}\, Mgeflow$$

We define the characteristic set *noOverLap* such that process p is a member if and only if the following hold:

- Elements directly contained in p do not share the same sequence or exception flow with elements directly contained in any other processes in p.
- Any two different elements contained in p do not share an incoming sequence flow, an outgoing sequence flow or an exception flow.
- Elements directly contained in any two different processes in p, do not share the same sequence or exception flow.
- Any two different elements contained in p do not share the same message flow.

$$noOverLap : \mathbb{P}\, Process$$

$$\begin{aligned}
&\forall p : Process \bullet p \in noOverLap \Leftrightarrow \\
&\quad (\forall e : \{g : Element \mid g \in_{\mathrm{p}} p\} \bullet \\
&\qquad \bigcup\{k : p \bullet (atom\, k).in \cup outs(k)\} \cap \bigcup\{k : content\, e \bullet (atom\, k).in \cup outs(k)\} = \emptyset \wedge \\
&\qquad (\forall f : \{g : Element \mid g \in_{\mathrm{p}} p\} \bullet e \neq f \Rightarrow \\
&\qquad\quad (atom\, e).in \cap (atom\, f).in = \emptyset \wedge \\
&\qquad\quad outs(e) \cap outs(f) = \emptyset \wedge \\
&\qquad\quad \bigcup\{k : content\, e \bullet (atom\, k).in \cup outs(k)\} \cap \\
&\qquad\quad \bigcup\{k : content\, f \bullet (atom\, k).in \cup outs(k)\} = \emptyset \wedge \\
&\qquad\quad getMsg\,(atom\, e) \cap getMsg\,(atom\, f) = \emptyset))
\end{aligned}$$

Property 2 states that every element which is not a start event must be preceded by a start event, while Property 3 states that every element which is not an end event must be succeeded by an end event. By taking the transitive closure of $edge(p)$ for any process p, we define the characteristic set of processes *endConnected* such that process p is a member if and only if Properties 2 and 3 hold for every element $e \in p$.

$$
\begin{array}{|l}
endsConnected : \mathbb{P}\, Process \\
\hline
\forall\, p : Process \bullet \\
\quad p \in endsConnected \Leftrightarrow \\
\qquad (\forall\, e : p \bullet \\
\qquad\quad (e \notin \{\, End \bullet ele \,\} \Rightarrow (\exists f : p \bullet f \in \{\, End \bullet ele \,\} \wedge (e, f) \in (edge\, p)^+)) \wedge \\
\qquad\quad (e \notin \{\, Start \bullet ele \,\} \Rightarrow (\exists f : p \bullet f \in \{\, Start \bullet ele \,\} \wedge (f, e) \in (edge\, p)^+)))
\end{array}
$$

We define the characteristic set of processes *noUnConnected* such that process p is a member if and only if Properties 4 and 5 hold for every element $e \in p$. Note that Property 1 specified by set *noOverLap* ensures that the existentially quantified variable f in set *noUnConnected* is unique.

$$
\begin{array}{|l}
noUnConnected : \mathbb{P}\, Process \\
\hline
\forall\, p : Process \bullet \\
\quad p \in noUnConnected \Leftrightarrow \\
\qquad (\forall\, e : p \bullet \\
\qquad\quad (\forall\, s : outs(e) \bullet \exists f : p \bullet s \in (atom\, f).in) \wedge \\
\qquad\quad (\forall\, s : (atom\, e).in \bullet \exists f : p \bullet s \in outs(f)))
\end{array}
$$

We now define the set *processSet* for capturing Properties 2, 3, 4 and 5.

$$
\begin{array}{|l}
processSet : \mathbb{P}\, Process \\
\hline
\forall\, p : Process \bullet p \in processSet \Leftrightarrow p \in endsConnected \cap noUnConnected
\end{array}
$$

The following schema *WFProcess* specifies the constraints about a BPMN process.

$$
\begin{array}{|l}
\textit{WFProcess} \\
proc : Process \\
\hline
proc \in noOverLap \cap processSet \\
\exists\, e, f : proc \bullet e \in \{Start \bullet ele\} \wedge f \in \{End \bullet ele\} \\
\neg\, (\exists\, e : proc \bullet e \in \{Inter \mid (atom\, ele).type \in \operatorname{ran} ierror \bullet ele\,\})
\end{array}
$$

The schema states that component *proc* must be a member of *noOverLap* and *processSet*, at least one start event and one end event are contained in *proc*, and there is no intermediate error event contained in *proc*. For example, consider Figure 4.3, which shows an intermediate error event directly contained in subprocess S. We observe that task B would never be performed due to the break in the flow of control by the intermediate error event that precedes it.

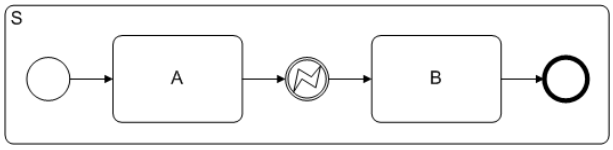


Figure 4.3: A subprocess containing an intermediate error event

Using *WFProcess* we define the schema *GenSub* to specify constraints that are specific to subprocesses.

GenSub

$ele : Element$

$ele \in \operatorname{ran} compound$
$\#(atom\ ele).send + \#(atom\ ele).receive = 0$
$(atom\ ele).type \in \operatorname{ran} subprocess \cup \operatorname{ran} mipars \cup \operatorname{ran} miseqs$
$content(ele) \in \{WFProcess \bullet proc\}$

We define schema *NormalSubProcess* as a conjunction of *GenSub* and *OneInOutFlow*, thereby insisting that a subprocess element must have one incoming and one outgoing sequence flows.

$$NormalSubProcess \mathrel{\widehat=} GenSub \land OneInOutFlow$$

4.3.6 Exception

An activity may have exception flows by attaching one or more BPMN intermediate events. Figure 4.4 shows an example of a task and a subprocess with an exception flow. Each intermediate event attached to an activity has an outgoing sequence flow. These sequence flows are the exception flows of that activity. In

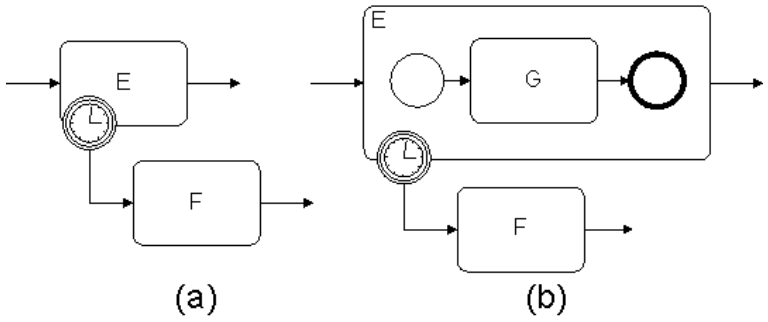


Figure 4.4: Representing interrupts by attaching an intermediate event to (a) a task and (b) a subprocess

the Haskell implementation, exception flows are recorded in `Atom`'s field `exit`, of type `[(Type,Seqflow)]`. In schema *Atom*, this is recorded by the component *exit*. The type of an intermediate event *ierror* takes a value of free type *Exception*.

$$Exception ::= exception\langle\langle \mathbb{N} \rangle\rangle \mid anyexception$$

Exception defines two constructors *anyexception* and *exception*: *anyexception* is a null constructor denoting that the error event is not associated to a particular type of error, and *exception* takes a natural number that identifies the particular type of error this event is associated to.

The schema *FullTask*0 specifies constraints about a task's exception flows. It states that every intermediate error event attached to a task element must have no associated error.

*FullTask*0

NormalTask

$\forall t : Type \bullet t \in (\operatorname{ran}((atom\ ele).exit) \cap \operatorname{ran} ierror) \Rightarrow (ierror^{\sim} t) = anyexception$

We define the set *errorCodeTypes* such that $t \in errorCodeTypes$ if and only if type t denotes an error event and records an error code.

$errorCodeTypes : \mathbb{P}\ Type$

$\forall t : Type \bullet$
$\quad t \in errorCodeTypes \Leftrightarrow$
$\quad\quad ((t \in \operatorname{ran} ierror \land (ierror^{\sim} t) \neq anyexception) \lor$
$\quad\quad (t \in \operatorname{ran} eerror \land (eerror^{\sim} t) \neq anyexception))$

We write $errorCode(t)$ to denote t's error code.

$$errorCode : errorCodeTypes \to \mathbb{N}$$

The schema *UniqueEndErrorSub* states that each end error event in a subprocess must have a unique error code.

UniqueEndErrorSub

$$NormalSubProcess$$

$$\begin{aligned}&\forall\, e, f : content(ele) \bullet \\ &\quad \{e, f\} \subseteq \{End \mid (atom\ ele).type \in (\text{ran}\ eerror \cap errorCodeTypes) \bullet ele\} \land \\ &\quad errorCode((atom\ e).type) = errorCode((atom\ f).type) \Rightarrow f = e\end{aligned}$$

The next two schemas relate end error events directly contained in a subprocess to intermediate error events attached to that subprocess. Schema *RelateEndIntErrorSub* states that if an end error event directly contained in a subprocess records an error code, then that code is also recorded by an intermediate error event attached to the subprocess.

RelateEndIntErrorSub

$$NormalSubProcess$$

$$\begin{aligned}&\forall\, e : content(ele) \bullet \\ &\quad (atom\ e).type \in (\text{ran}\ eerror \cap errorCodeTypes) \Rightarrow \\ &\qquad (\exists\, t : \text{ran}((atom\ ele).exit) \bullet \\ &\qquad\quad t \in (\text{ran}\ ierror \cap errorCodeTypes) \land \\ &\qquad\quad errorCode((atom\ e).type) = errorCode(t))\end{aligned}$$

Similarly, schema *RelateIntEndErrorSub* states that if an intermediate error event attached to a subprocess record an error code then the code is also recorded by an end error event directly contained in that subprocess.

RelateIntEndErrorSub

$$NormalSubProcess$$

$$\begin{aligned}&\forall\, t : \text{ran}((atom\ ele).exit) \bullet \\ &\quad (t \in (\text{ran}\ ierror \cap errorCodeTypes) \Rightarrow \\ &\qquad (\exists\, e : content(ele) \bullet \\ &\qquad\quad (atom\ e).type \in (\text{ran}\ eerror \cap errorCodeTypes) \land \\ &\qquad\quad errorCode((atom\ e).type) = errorCode(t)))\end{aligned}$$

Using the schemas defined above, we construct the schema *FullSub*0.

$$FullSub0 \mathrel{\widehat{=}} UniqueEndErrorSub \land RelateEndIntErrorSub \land RelateIntEndErrorSub$$

We complete the schema definition for tasks and subprocesses by insisting all events attached to their boundary must be intermediate events.

$$Attach \mathrel{\widehat{=}} [ele : Element \mid (\forall\, t : Type \bullet t \in \text{ran}((atom\ ele).exit) \Rightarrow t \in inters)]$$

and the full schema definitions are defined as follows.

$$FullTask \mathrel{\widehat{=}} FullTask0 \land Attach$$

$$FullSub \mathrel{\widehat{=}} FullSub0 \land Attach$$

Finally schema *Activity* specifies an activity element.

$$Activity \mathrel{\widehat{=}} FullTask \lor FullSub$$

4.4 Pools and Diagrams

4.4.1 Haskell Syntax

In the Haskell implementation, we defined type `Diagram` to be a list of pairs; each pair has the type `(PoolId,[Element])` and records a BPMN pool in a BPMN diagram. The first component identifies the BPMN pool, and the second component is a list of `Element` values that this pool directly contains.

4.4.2 Pools

In previous sections we specified the three categories of flow objects via schemas. We may now specify a flow object using the following schema *FlowObject*.

$$FlowObject \mathrel{\widehat=} Event \lor Activity \lor Gate$$

We provide a specification of BPMN pools. We define *onlyFlowObject* to be a characteristic set of processes, such that a process is in this set if and only if all elements contained in the process satisfy the constraints specified by the schema *FlowObject*.

$$\begin{array}{|l} onlyFlowObject : \mathbb{P}\, Process \\ \hline \forall\, ps : Process \bullet \\ \quad ps \in onlyFlowObject \Leftrightarrow (\forall\, e : \{g : Element \mid g \in_{\mathrm{p}} ps\} \bullet e \in \{FlowObject \bullet ele\}) \end{array}$$

The following schema *GenProc* states that a BPMN process satisfies the schema *WFProcess*, and that elements contained in the process satisfy schema *FlowObject*.

$$GenProc \mathrel{\widehat=} [WFProcess \mid proc \in onlyFlowObject]$$

We define schema *hasExgates* to specify the relationship between an event-based XOR gateway and the targets of its outgoing sequence flows in a BPMN process. Specifically, the schema states that each outgoing sequence flow of an event-based gateway contained in a BPMN pool must be an incoming sequence flow of either an intermediate event or a task. The set *eventgate* abbreviates the set of event-based XOR gateways. The set *sendelement* abbreviates the set of intermediate events and tasks, that is, all the elements in our BPMN subset that could have incoming message flows. The function *direct* returns either *proc* if $e \in proc$, or $content(f)$ where f is a subprocess contained in *proc* that directly contains e.

$$\begin{array}{|l} hasExgates \\ \hline proc : Process \\ \hline \forall\, f : \{g : Element \mid g \in_{\mathrm{p}} proc\} \bullet f \in eventgate \Rightarrow \\ \quad (\forall\, e : Element \bullet (f, e) \in edge(direct(proc, e)) \Rightarrow e \in sendelement) \end{array}$$

We specify a BPMN pool using schema *Pool*, and is defined as the following conjunction.

$$Pool \mathrel{\widehat=} GenProc \land hasExgates$$

4.4.3 Diagrams

A BPMN diagram is a collection of BPMN pools interacting via message flow communication.

We specify the structure of a BPMN diagram using the schema *GenDiagram*. This schema states that a diagram consists of one or more BPMN pools, with each pool being uniquely identified by some *PoolId* value i such that $pool(i)$ gives that diagram (where *PoolId* is a basic type).

$$GenDiagram \mathrel{\widehat=} [pool : PoolId \rightarrowtail Pool \mid pool \neq \emptyset]$$

In a BPMN diagram no two BPMN pools share a sequence flow. This is specified by the schema *SequenceFlows*,

$$SequenceFlows \mathrel{\widehat=} [\, GenDiagram \mid \forall\, p, q : \operatorname{ran} pool \bullet p \neq q \Rightarrow getSeqflows\ p.proc \cap getSeqflows\ q.proc = \emptyset \,]$$

where function *getSeqflows* is defined such that $getSeqflows(ps)$ returns sequence flows of all elements contained in ps.

$$getSeqflows : \mathbb{P}\, Element \to \mathbb{P}\, Seqflow$$

We also specify message flows of elements in a BPMN diagram. Functions *getSds* and *getRec* are defined such that $getSds(ps)$ and $getRec(ps)$ return outgoing and incoming messages flows of all elements contained in ps respectively.

$$getSds, getRecs : \mathbb{P}\, Element \to \mathbb{P}\, Mgeflow$$

Schema *MessageFlows* states that in a BPMN diagram, elements in two different BPMN pools do not share the same incoming and outgoing message flows. This schema also specifies that in a BPMN diagram a message flow is an outgoing message flow of an element contained in a BPMN pool if and only if it is also an incoming message flow of some element contained in a different BPMN pool of that diagram.

MessageFlows

$GenDiagram$

$$\begin{aligned}
&(\forall\, p, q : \operatorname{ran} pool \bullet \\
&\quad (p \neq q \Rightarrow getSds\ p.proc \cap getSds\ q.proc = \emptyset \wedge getRecs\ p.proc \cap getRecs\ q.proc = \emptyset)) \\
&(\forall\, p : \operatorname{ran} pool \bullet \\
&\quad (\forall\, m : getSds\ p.proc \bullet (\exists\, q : \operatorname{ran} pool \bullet (p \neq q \wedge m \in getRecs\ p.proc))) \wedge \\
&\quad (\forall\, m : getRecs\ p.proc \bullet (\exists\, q : \operatorname{ran} pool \bullet (p \neq q \wedge m \in getSds\ p.proc))))
\end{aligned}$$

The schema *Diagram* is the conjunction of the above two schemas, and specifies the relationships between elements in a BPMN diagram.

$$Diagram \mathrel{\widehat=} SequenceFlows \wedge MessageFlows$$

4.5 Initialisation Theorems

This section defines the initial states for BPMN elements, processes, pools and diagrams, and shows that they are consistent with respect to their Z specification.

4.5.1 BPMN Element

We first define the initial sequence flow, start and end event elements.

$$seq1 : Seqflow$$

$startatom, endatom : Atom$

$$\begin{aligned}
&startatom = \\
&\quad \langle\!| type \rightsquigarrow start, in \rightsquigarrow \emptyset, out \rightsquigarrow \{seq1\}, exit \rightsquigarrow \emptyset, range \rightsquigarrow (0,0), send \rightsquigarrow \emptyset, receive \rightsquigarrow \emptyset |\!\rangle \wedge \\
&endatom = \\
&\quad \langle\!| type \rightsquigarrow end, in \rightsquigarrow \{seq1\}, out \rightsquigarrow \emptyset, exit \rightsquigarrow \emptyset, range \rightsquigarrow (0,0), send \rightsquigarrow \emptyset, receive \rightsquigarrow \emptyset |\!\rangle
\end{aligned}$$

$startele, endele : Element$

$$startele = atomic\ startatom \wedge endele = atomic\ endatom$$

Schemas *StartAtomInit* and *EndAtomInit* specify *startatom* and *endatom* to be two possible initial states of *Atom*.

$$StartAtomInit \mathrel{\widehat=} [Atom' \mid \theta Atom' = startatom]$$
$$EndAtomInit \mathrel{\widehat=} [Atom' \mid \theta Atom' = endatom]$$

We show these states to be consistent with respect to schema *Atom*.

Theorem 4.4. $\exists\, Atom' \bullet StartAtomInit$

Proof. See Page 185 (Section A.5 in Appendix A). □

Theorem 4.5. $\exists\, Atom' \bullet EndAtomInit$

Proof. See Page 186 (Section A.5 in Appendix A). □

We now define *startele* and *endele* to two possible initial states of *FlowObject*.

$$StartInit \mathrel{\widehat=} [FlowObject' \mid ele' = startele]$$
$$EndInit \mathrel{\widehat=} [FlowObject' \mid ele' = endele]$$

We show these states to be consistent with respect to schema *FlowObject*.

Theorem 4.6. $\exists\, FlowObject' \bullet StartInit$

Proof. See Page 186 (Section A.5 in Appendix A). □

Theorem 4.7. $\exists\, FlowObject' \bullet EndInit$

Proof. See Page 187 (Section A.5 in Appendix A). □

4.5.2 BPMN Process

We define an initial state of a BPMN process by considering the constraints about elements contained in it. Using the start and end events *startele* and *endele*, we define the initial state of a process to be one that directly contains these two events. We first define process *initialproc* as follows.

$$initialproc : Process$$
$$\hline$$
$$initialproc = \{startele, endele\}$$

The following schema *GenProcInit* characterises the initial state of *GenProc*; *GenProc*, which is defined in Section 4.4.2, specifies properties about elements contained in a BPMN process.

$$GenProcInit \mathrel{\widehat=} [GenProc' \mid proc' = initialproc]$$

We show this state to be consistent with respect to schema *GenProc*.

Theorem 4.8. $\exists\, GenProc' \bullet GenProcInit$

Proof. See Page 188 (Section A.5 in Appendix A). □

4.5.3 BPMN Pool

We define schema *PoolInit* to specify the initial state of a BPMN pool *Pool*.

$$PoolInit \mathrel{\widehat=} [Pool' \mid proc' = initialproc]$$

We show this state to be consistent with respect to schema *Pool*.

Theorem 4.9. $\exists\, Pool' \bullet PoolInit$

Proof. See Page 191 (Section A.5 in Appendix A). □

4.5.4 BPMN Diagram

We define the initial state of a BPMN diagram to contain only one BPMN pool. This pool is initialised according to schema *PoolInit*. The value *initialpool* is a mapping from a pool identifier to a schema binding of *Pool* whose component *proc* has the value *initialproc*.

$$\begin{array}{|l} pool1 : PoolId \\ initialpool : PoolId \pfun Pool \\ \hline initialpool = \{pool1 \mapsto \langle\!| proc \leadsto initialproc |\!\rangle\} \end{array}$$

We define schema *DiagramInit* to specify the initial state of a BPMN diagram.

$$DiagramInit \mathrel{\widehat=} [Diagram' \mid pool' = initialpool]$$

We show this state to be consistent with respect to schema *Diagram*.

Theorem 4.10. $\exists\, Diagram' \bullet DiagramInit$

Proof. See Page 192 (Section A.5 in Appendix A). □

4.6 Diagrams Construction

Using the schema calculus we provide eight syntactic operations for constructing BPMN processes. Specifically these are operation schemas on the state schemas *Pool* and *Diagram*. They are partitioned into the six categories: sequential composition, split, join, iteration, interrupt and collaboration. These categories are described from Section 4.6.2 to Section 4.6.7.

While the syntactic operations defined in this section are not part of BPMN, we did not need to extend the existing syntax of BPMN for defining these operations. These operations are designed to provide following benefits:

- To provide the expressiveness and the flexibility to describe business processes. We have chosen a comprehensive set of operations to provide the expressiveness for describing business process similar to those in structured programming [DDH72]. All BPMN processes presented in this thesis can be constructed using operations defined in this section.
- To ensure the syntactic consistency of business processes. Syntactic consistency is ensured by calculating the preconditions of the operations defined in this section. These preconditions show that the application of these operations on a syntactically valid BPMN process guarantees a syntactically valid BPMN process. Precondition calculations are described in Section 4.6.9, while full calculations may be found in Appendix B.
- To assist the development of a compositional approach for describing and reasoning about the behaviour of business processes. Specifically in Chapter 5, we provide a process semantics to these operations using the process algebra CSP [Ros98]. Using this semantics we are able to give a compositional approach to reasoning about the behavioural correctness of complex BPMN business processes; this is achieved by exploiting the transitive and monotonic properties of CSP refinements [Ros98].

The rest of this section is structured as follows: In Section 4.6.1 we provide some preliminary definitions to assist the specification. Note that some auxiliary definitions are only partially defined in this section; full definitions can be found in Appendix A.6. From Section 4.6.2 to Section 4.6.7, we describe the eight syntactic operations using the Z schema calculus; when describing operations in Sections 4.6.2 to 4.6.7, we refer to Figure 4.6 for illustration purposes. Each subfigure in Figure 4.6 illustrates a single operation. The diagram on the left hand side of each subfigure depicts the operation's before state, and the diagram on the right hand side depicts its after state. In Section 4.6.8 we show an example of how to construct the customer business process of our online shop example in Figure 2.3, and in Section 4.6.9 we present the precondition calculations of the operations.

4.6.1 Preliminaries

4.6.1.1 States and Functions

We first describe informally schema definitions about subsets of BPMN elements. *InitialInter* specifies either an intermediate time event or an intermediate message event with no associated message flow. *InitialEnd* specifies a non-trigger end event. *NonEvSplit* specifies either a data-based XOR or an AND split gateway, *EventSplit* specifies an event-based XOR split gateway and *NonEvJoin* specifies a data-based XOR or an AND join gateway. *OneInOutAtom* specifies either an intermediate time event, an intermediate message event with no associated message flow, or a task with no message and exception flow. *OneInOutObject* specifies either an intermediate time event, an intermediate message event with no associated message flow, or an activity (task or subprocess) with no message and exception flow.

We provide the following auxiliary definitions.

$$alls, ends, activities, subs, tasks, eerrors, nonsends : Process \pfun (Seqflow \pfun Element)$$

Function *alls* takes some process p such that $alls\, p$ is a function that relates each incoming sequence flow of an element contained in p to that element. Functions *ends*, *activities*, *subs* and *eerror* are defined as follow: $ends\, p$ relates each incoming sequence flow of an end event contained in p to that event; $activities\, p$ relates each incoming sequence flow of an activity contained in p to that activity; $subs\, p$ relates each incoming sequence flow of a subprocess contained in p to that subprocess; $tasks\, p$ relates each incoming sequence flow of a task contained in p to that task; and *eerror* relates each incoming sequence flow of an end error event contained in p to that event. Function *nonsends* is defined such that $nonsends\, p$ relates each outgoing sequence flow of an element contained in p that is not an event-based XOR gateway to that element.

A nonempty finite set of elements belongs to the set *uniqueIns* if and only if elements in that set do not share incoming sequence flows.

$$uniqueIns : \mathbb{P}(\mathbb{F}_1\, Element)$$

$$\forall\, es : \mathbb{F}_1\, Element \bullet (es \in uniqueIns \Leftrightarrow (\forall\, e, f : es \bullet e \neq f \Rightarrow (atom\, e).in \cap (atom\, f).in = \emptyset))$$

We augment this definition to *uniqueEnds* such that a nonempty finite set of elements belongs to the set *uniqueEnds* if and only if the elements are non-trigger end events satisfying *InitialEnd* and that they do not share incoming sequence flows.

$$uniqueEnds : \mathbb{P}(\mathbb{F}_1\, Element)$$

The function *modify* takes a process ps and two finite sets of elements ns and rs. The function performs the following: if elements in rs are directly contained in ps, this operation returns the set $(ps \setminus rs) \cup ns$; otherwise it recursively searches for a subprocess element s that is directly contained in ps, and contains elements in rs and returns ps with content of s modified according to function application $modify(content(s), ns, rs)$. This is a partial function as it is only defined for inputs where process ps contains elements in rs. The expression $cont(ps, rs) = \{e : ps \mid rs \subseteq \{f : Element \mid f \in_{\mathrm{e}} e\}\}$. That is, it returns a set of subprocesses qs directly contained in ps such that the subprocesses contain all elements in rs. The expression $rep(s, modify(content(s), ns, rs))$ evaluates to the element $compound(atom\, s, modify(content(s), ns, rs))$. That is, subprocess s with elements it directly contains being replaced by elements in $modify(content(s), ns, rs)$.

$$modify : (Process \times \mathbb{F}\, Element \times \mathbb{F}_1\, Element) \pfun Process$$

$$\begin{aligned}
&\forall\, ps : Process;\ ns : \mathbb{F}\, Element;\ rs : \mathbb{F}_1\, Element \bullet \\
&\quad (rs \subseteq ps \Rightarrow modify(ps, ns, rs) = ((ps \setminus rs) \cup ns)) \land \\
&\quad (rs \subseteq (\{e : Element \mid e \in_{\mathrm{p}} ps\} \setminus ps) \Rightarrow \\
&\qquad modify(ps, ns, rs) = \\
&\qquad\quad ((ps \setminus cont(ps, rs)) \cup \{s : cont(ps, rs) \bullet rep(s, modify(content(s), ns, rs))\}))
\end{aligned}$$

4.6.1.2 Operation Schemas

We now consider operation schemas on a BPMN element. These operations help to construct the syntactic operations on *Pool* and *Diagram*. We first provide the following definition for introducing a new sequence or a message flow to a process.

Definition 4.11. ***Fresh flow.*** *A sequence flow s is fresh from an element e if and only if $s \notin getSeqflows\{e\}$. Likewise, a message flow m is fresh from an element e if and only $m \notin getMsgs\{e\}$. We say s (and m) is fresh from a set of elements ps if and only if s (and m) is fresh from all elements contained in ps.*

Operation *ChangeFlow* replaces an incoming sequence flow of a non-start BPMN element; all BPMN elements, except start elements, must have at least one incoming sequence flow. Here *to*? must be fresh from *ele* and *from*? must be an incoming sequence flow of *ele*. The expression $cge(ele, from?, to?)$ is element *ele* after replacing its incoming sequence flow *from*? with *to*?. An illustration of the operation *ChangeFlow* is shown in Figure 4.5(A).

ChangeFlow

$\Delta FlowObject$
$from?, to? : Seqflow$

$ele \notin \{Start \bullet ele\}$
$to? \notin getSeqflows\{ele\}$
$from? \in (atom\ ele).in$
$ele' = cge(ele, from?, to?)$

Operation *AddNoRelatedErrorExceptionSub* adds an exception flow to an activity element where the type of the exception flow is not associated with a specific error. The expression $ce(ele, (eflow?, etype?), \emptyset, \emptyset)$ is element *ele* after an exception flow, specified by the pair $(eflow?, etype?)$, is added. The set *nomsgserrors* is the set of intermediate events with no message and no associated with error codes.

$$nomsgserrors == \operatorname{ran} itime \cup \{imessage(nomessage)\} \cup \operatorname{ran} irule \cup \{i : \operatorname{ran} ierror \mid (ierror^{\sim})\ i \in \operatorname{ran} exception\}$$

Note that *eflow*? must be fresh from *ele*. An illustration of this operation is shown in Figure 4.5(B).

AddNoRelatedErrorExceptionSub

$\Delta Activity$
$etype? : Type$
$eflow? : Seqflow$

$etype? \in nomsgserrors$
$eflow? \notin getSeqflows\{ele\}$
$ele' = ce(ele, (eflow?, etype?), \emptyset, \emptyset)$

Operation *ChangeEndType* replaces the *type* component of an end event with the input end error type *type*?. The expression $ct(ele, type?)$ is element *ele* after replacing its type with the value *type*?. An illustration of this operation is shown in Figure 4.5(C).

ChangeEndType

ΔEnd
$type? : Type$

$type? \in \operatorname{ran} eerror \cap errorCodeTypes$
$(atom\ ele).type = end$
$ele' = ct(ele, type?)$

Operation *AddRelatedErrorExceptionSub* performs two steps:

1. Adds an exception flow, $(eflow?, etype?)$, to a subprocess.
2. Associates the type $etype?$ of the exception flow to an end error event directly contained in that subprocess.

The second step is achieved by replacing the type of a non-trigger end event with $type?$ such that $etype?$ and $type?$ refer to the same specific error. The expression $ce(ele, (eflow?, etype?), \{endele\}, \{endele'\})$ is element ele after the exception flow $(eflow?, etype?)$ is added and element $endele$ is replaced with $endele'$. An illustration of this operation is shown in Figure 4.5(D).

AddRelatedErrorExceptionSub

$$\Delta FullSub$$
$$ChangeEndType[endele/ele, endele'/ele']$$
$$etype? : Type$$
$$sflow?, eflow? : Seqflow$$

$$etype? \in (\text{ran } ierror \cap errorCodeTypes) \setminus (\text{ran}((atom\ ele).exit))$$
$$errorCode\ etype? = errorCode\ type?$$
$$eflow? \notin getSeqflows\{ele\}$$
$$endele \in (content\ ele)$$
$$(ends\ (content\ ele))sflow? = endele$$
$$ele' = ce(ele, (eflow?, etype?), \{endele\}, \{endele'\})$$

Operations *AddSendMgeFlowTask* and *AddReceiveMgeFlowTask* add an outgoing and an incoming message flow to a task respectively. The expression $cm(ele, \{msg?\}, \emptyset, (atom\ ele).exit)$ is element ele after adding message flow $msg?$ to ele's $send$ component. Conversely, the expression $cm(ele, \emptyset, \{msg?\}, (atom\ ele).exit)$ is ele after adding $msg?$ to the its $receive$ component. An illustration of *AddSendMgeFlowTask* and *AddReceiveMgeFlowTask* is shown in Figures 4.5(E) and (F) respectively.

$$AddSendMgeFlowTask \mathrel{\widehat=}$$
$$[\Delta FullTask;\ msg? : Mgeflow \mid$$
$$\quad msg? \notin getMsg(atom\ ele) \land ele' = cm(ele, \{msg?\}, \emptyset, (atom\ ele).exit)]$$

$$AddReceiveMgeFlowTask \mathrel{\widehat=}$$
$$[\Delta FullTask;\ msg? : Mgeflow \mid$$
$$\quad msg? \notin getMsg(atom\ ele) \land ele' = cm(ele, \emptyset, \{msg?\}, (atom\ ele).exit)]$$

Operation *AddExceptionMgeFlow* adds an incoming message flow to an intermediate message event attached to an activity. This operation insists that exactly one intermediate message event, which is not associated to a message flow $imessage(nomessage)$, is attached to the activity. The expression $rrange((atom\ ele).exit, imessage(nomessage), imessage(message(msg?)))$ returns the component $(atom\ ele).exit$ after all pairs $(s, imessage(nomessage)) \in (atom\ ele).exit$ are replaced with $(s, imessage(message(msg?)))$. An illustration of *AddExceptionMgeFlow* is shown in Figure 4.5(G).

AddExceptionMgeFlow

$$\Delta Activity$$
$$msg? : Mgeflow$$

$$msg? \notin getMsg(atom\ ele)$$
$$\#(((atom\ ele).exit) \rhd \{imessage(nomessage)\}) = 1$$
$$ele' = cm(ele, \emptyset, \emptyset, rrange((atom\ ele).exit, imessage(nomessage), imessage(message(msg?))))$$

Operation *AddMgeEvent* is the disjunction of schemas *AddSMgeEvent*, *AddIMgeEvent* and *AddEMgeEvent*.

$$AddMgeEvent \mathrel{\widehat=} AddSMgeEvent \lor AddIMgeEvent \lor AddEMgeEvent$$

The operation adds a message flow to a message event. An illustration of these operations is shown in Figures 4.5(H), (I) and (J).

$$AddSMgeEvent \mathrel{\widehat=}$$
$$[\Delta Start;\ msg? : Mgeflow \mid$$
$$(atom\ ele).type = smessage(nomessage) \land ele' = ct(ele, smessage(message(msg?)))]$$

$$AddIMgeEvent \mathrel{\widehat=}$$
$$[\Delta Inter;\ msg? : Mgeflow \mid$$
$$(atom\ ele).type = imessage(nomessage) \land ele' = ct(ele, imessage(message(msg?)))]$$

$$AddEMgeEvent \mathrel{\widehat=}$$
$$[\Delta End;\ msg? : Mgeflow \mid$$
$$(atom\ ele).type = emessage(nomessage) \land ele' = ct(ele, emessage(message(msg?)))]$$

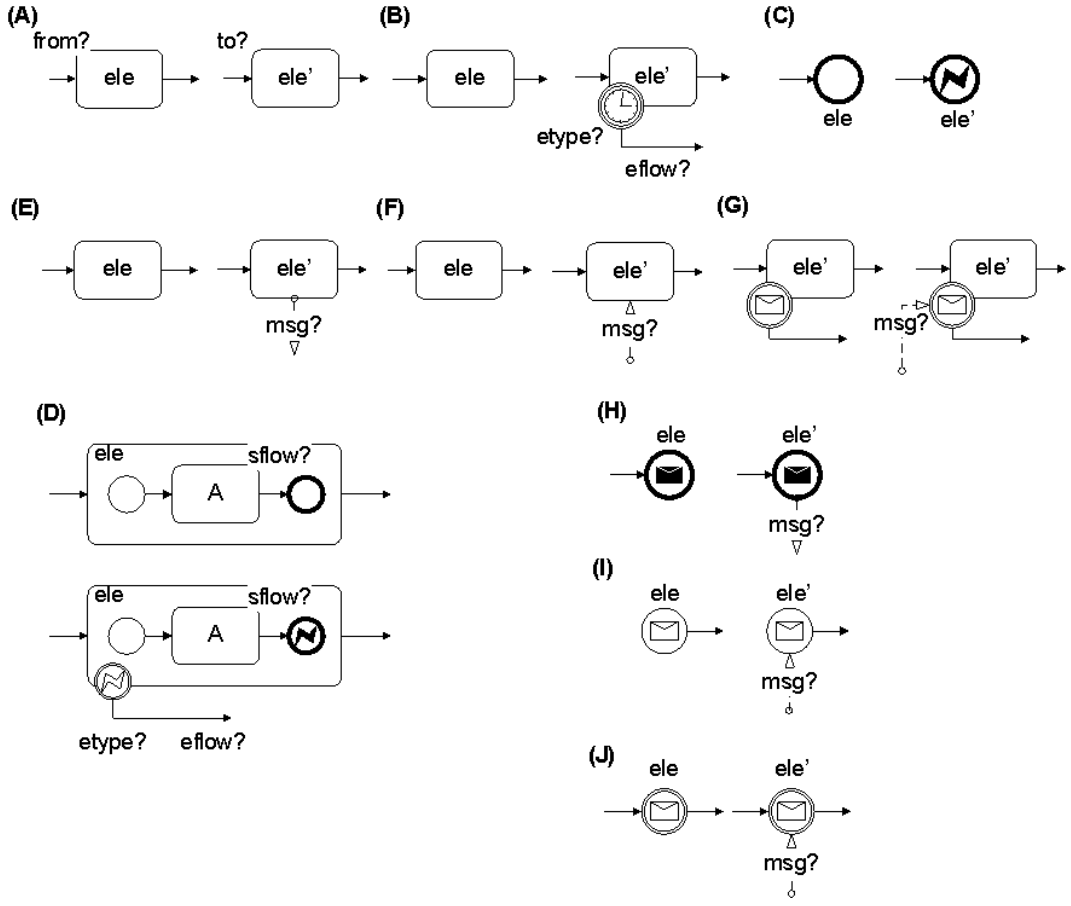


Figure 4.5: Before-and-after illustrations of operation schemas

4.6.2 Sequential Composition

We define the sequential composition by the operation schema *SeqComp*.

SeqComp

$\Delta Pool$
$CommonConstraints$
$end? : Element$

$new? \in \{ele : Element \mid OneInOutObject\}$
$end? \in \{ele : Element \mid InitialEnd\}$
$(atom\ end?).in = (atom\ new?).out$
$proc' = modify(proc, \{new?, end?\}, \{((ends\ proc)\ from?)\})$

This operation takes two elements *new*? and *end*?, and a sequence flow *from*? as inputs. An illustration of *SeqComp* is shown in Figure 4.6(A), where the end event labelled E is specified by the expression $((ends\ proc)\ from?)$. The illustration shows how this operation replaces element E with element *new*? and *end*?. We now consider the constraints specified by *SeqComp* in detail.

Specifically, *new*? is either an intermediate time or message event (with no message flow) or an activity. That is, *new*? has exactly one incoming and one outgoing sequence flow, and *end*? is a non-trigger end event. Furthermore, *SeqComp* includes the following schema *CommonConstraints*.

CommonConstraints

$Pool$
$new? : Element$
$from? : Seqflow$

$(outs\ new? \cup (getSeqflows\ (content\ new?))) \cap getSeqflows\ proc = \emptyset$
$from? \in (atom\ new?).in$
$(atom\ new?).in \subseteq \mathrm{dom}(ends\ proc)$

CommonConstraints states the following constraints:

- All outgoing sequence, exception flows of *news* as well as all sequence and exception flows of elements contained in *new*? must be fresh from the before state component *proc*.
- *from*? is an incoming sequence flow of *new*?.
- *from*? is also an incoming sequence flow of an end element contained in *proc*; $(ends\ proc)$ is a partial function such that $(ends\ proc)\ f$ returns an end event contained in *proc* and that has incoming sequence f.

4.6.3 Splits

There are two splits operations *Split* and *EventSplitOp*. *Split* adds either a data-based XOR or an AND split gateway to a BPMN pool, while *EventSplitOp* adds an event-based XOR split gateway to a BPMN pool. We first consider operation *Split*.

Split

$\Delta Pool$
$CommonConstraints$
$outs? : \mathbb{F}_1\ Element$

$new? \in \{NonEvSplit \bullet ele\}$
$outs? \in uniqueEnds$
$getIns(outs?) \cap getSeqflows\ proc = \emptyset$
$getIns(outs?) = (atom\ new?).out$
$proc' = modify(proc, \{new?\} \cup outs?, \{((ends\ proc)\ from?)\})$

This operation takes input components *outs*?, *new*? and *from*?. An illustration of *Split* is shown in Figure 4.6(B), where the end event labelled E is specified by the expression $((ends\ proc)\ from?)$. The

illustration shows how the operation replaces E with element *new*? and the set of elements *outs*?, which contains elements labelled F and G. We now consider the constraints specified by this operation in detail.

Split includes constraints specified by *CommonConstraints* about *new*?, *from*? and the before state *Pool*. In addition, it specifies the following constraints on all input components:

- *new*? must be either a data-based XOR or an AND split gateway
- *outs*? must be a non-empty set of end events, in which elements do not share incoming sequence flows.
- Incoming sequence flows of elements in *outs*? are fresh from the before state *Pool*.
- Incoming sequence flows of elements in *outs*? are exactly the outgoing sequence flows of *new*?.

We now consider the schema definition *EventSplitOp*.

EventSplitOp

$$\Delta Pool$$
$$CommonConstraints$$
$$EventSplitAux$$

$$new? \in \{EventSplit \bullet ele\}$$
$$getIns(events?) = (atom\ new?).out$$
$$getSeqflows(events? \cup ends?) \cap getSeqflows\ proc = \emptyset$$
$$proc' = modify(proc, \{new?\} \cup events? \cup ends?, \{((ends\ proc)\ from?)\})$$

This operation takes input components *events*?, *ends*?, *new*? and *from*?. An illustration of *EventSplitOp* is shown in Figure 4.6(C), where the end event labelled E is specified by the expression $((ends\ proc)\ from?)$. The illustration shows how the operation replaces E with element *new*? and two sets of elements *ends*? and *events*?. Here *end*? contains elements labelled F and G and *events*? contains elements labelled *eventA* and *eventB* respectively. This operation ensures each direct successors of the event-based split gateway is either a task or an intermediate event. We now consider the constraints specified by *EventSplitOp* in detail.

This operation includes constraints specified by *CommonConstraints* on *new*?, *from*? and the before state *Pool*. In addition, it includes the schema *EventSplitAux*.

EventSplitAux

$$events?, ends? : \mathbb{F}_1\ Element$$

$$events? \subseteq \{ele : Element \mid OneInOutAtom\}$$
$$ends? \subseteq \{ele : Element \mid InitialEnd\}$$
$$events? \cup ends? \in uniqueIns$$
$$\#events? = \#ends?$$
$$getOuts(events?) = getIns(ends?)$$

Specifically, schema *EventSplitOp* specifies the following constraints.

- *new*? must be an event-based XOR split gateway.
- Incoming sequence flows of elements in *events*? are exactly the outgoing sequence flows of *new*?.
- Sequence flows of elements in *events* and *ends*? are fresh from the before state *Pool*.
- Each element in *events*? is either an intermediate time event, a message event or an activity.
- *ends* is a set of non-trigger end events.
- Elements in *events*? and *ends*? do not share incoming sequence flows.
- *events*? and *ends*? have the same number of elements.
- The outgoing sequence flows of elements in *events*? are exactly the incoming sequence flows of elements in *ends*?.

4.6.4 Join

We define the join operation by the operation schema *JoinOp*. This operation adds either a data-based XOR or an AND join gateway to a BPMN pool.

JoinOp

$$\Delta Pool$$
$$gate?, end? : Element$$

$$gate? \in \{NonEvJoin \bullet ele\}$$
$$end? \in \{InitialEnd \bullet ele\}$$
$$(atom\ gate?).in \subseteq \mathrm{dom}(ends\ proc)$$
$$(proc, (ends\ proc)(\!|\ (atom\ gate?).in\ |\!)) \in together$$
$$(atom\ gate?).out = (atom\ end?).in$$
$$(atom\ end?).in \cap getSeqflows\ proc = \emptyset$$
$$proc' = modify(proc, \{gate?, end?\}, (ends\ proc)(\!|\ (atom\ gate?).in\ |\!))$$

JoinOp takes input components *gate*? and *end*?. An illustration of *JoinOp* is shown in Figure 4.6(D), where the end events labelled E and F are specified by the expression $(ends\ proc)(\!|\ (atom\ gate?).in\ |\!)$. The illustration shows how the operation replaces elements E and F with elements *gate*? and *end*?. Here *end*? is labelled G in the illustration. Specifically, *JoinOp* defines the following constraints:

- *gate*? is either a data-based XOR or an AND join gateway.
- *end*? is a non-trigger end event.
- *gate*?'s incoming sequence flows are incoming sequence flows of some end events contained in the before state *Pool*, this constraint is specified by the membership $(proc, (ends\ proc)(\!|\ (atom\ gate?).in\ |\!)) \in together$, where *together* is a binary relation defined as follows.

  $$together : (Process \leftrightarrow \mathbb{F}_1\ Element)$$

  $$\forall\, p : Process;\ es : \mathbb{F}_1\ Element \bullet$$
  $$(p, es) \in together \Leftrightarrow es \subseteq p \vee \#\{e : Element \mid e \in_{\mathrm{p}} p \wedge es \subseteq content(e)\} = 1$$

- Outgoing sequence flows of *gate*? are exactly the incoming sequence flows of *end*?
- Incoming sequence flows of *end*? are fresh from the before state *Pool*.

4.6.5 Iteration

There are two iteration operations *Loop* and *EventLoop*. We first consider *Loop*. An illustration of *Loop* is shown in Figure 4.6(E). The illustration shows how iteration is constructed using a (non event-based) split and a join gateway. We first consider the constraints on the split gateway. These are provided by the following schema *ConnectSplit*.

ConnectSplit

$$CommonConstraints[split?/new?]$$
$$connect? : Seqflow$$
$$end? : Element$$

$$split? \in \{NonEvSplit \bullet ele\}$$
$$end? \in \{ele : Element \mid InitialEnd\}$$
$$connect? \notin (atom\ end?).in$$
$$(atom\ split?).out = \{connect?\} \cup (atom\ end?).in$$

This schema specifies the following constraints between the input components *split*?, *end*?, *connect*? and *from*?.

- *split*? is either a data-based XOR or an AND split gateway.
- *end*? is a non-trigger end event.
- Sequence flow *connect*? is fresh from *end*?.
- Sequence flows of set $\{connect?\} \cup (atom\ end?).in$ are the outgoing sequence flows of *split*.

The constraints specified in *CommonConstraint* ensure the outgoing sequence flows of *split*? are fresh from the before state *Pool*.

The following schema *ConnectJoin* specifies the constraints on the non event-based join gateway.

ConnectJoin

$$Pool$$
$$ChangeFlow[change/ele, change'/ele', f2?/from?, t2?/to?]$$
$$ConnectJoin0$$

$$t2? \notin getSeqflows\ proc$$
$$(nonsends\ proc)\ f2? = change$$

This schema takes input element *join*?, and sequence flows *connect*?, $f2?$ and $t2?$. This includes the operation schema *ChangeFlow*, which replaces the incoming sequence flow $f2?$ of the element $(nonsends\ proc)\ f2?$ with $t2?$. The expression $(nonsends\ proc)\ f2?$ ensures this element is not a direct successor of an event-based XOR gateway. A detail description of *ChangeFlow* is provided on Page 39 in Section 4.6.1.2. *ConnectJoin* also insists that $t2?$ is fresh from the before state *Pool*. In addition, this schema includes the schema *ConnectJoin*0.

*ConnectJoin*0

$$join? : Element$$
$$connect?, f2?, t2? : Seqflow$$

$$join? \in \{NonEvJoin \bullet ele\}$$
$$connect? \neq t2?$$
$$(atom\ join?).in = \{f2?, connect?\}$$
$$outs\ join? = \{t2?\}$$

*ConnectJoin*0 specifies the following constraints on the input components independently from the before state.

- *join*? is either a join gateway.
- Sequence flows *connect*? and $t2?$ are not the same.
- $f2?$ and *connect*? are the only incoming sequence flows of *join*?; since $f2?$ is not fresh from the before state and *connect*? is, this ensures that $f2?$ and *connect*? are not the same flow.
- $t2?$ is the only outgoing sequence flow of *join*?.

Having specified independent constraints on the split and join gateways for constructing the iteration, the following schema *Connect* specifies the interdependent constraints on the gateways.

$$\begin{aligned}
&Connect \mathrel{\widehat=} \\
&\quad [\Delta Pool;\ from?, f2?, t2? : Seqflow;\ change, change', split?, join?, end? : Element \mid \\
&\qquad from? \neq f2? \land \\
&\qquad t2? \notin (atom\ end?).in \land \\
&\qquad (proc, \{((ends\ proc)\ from?), change\}) \in together \land \\
&\qquad (change, ((ends\ proc)\ from?)) \in (edge(direct(proc, change)))^+ \land \\
&\qquad proc' = modify(proc, \{split?, join?, end?, change'\}, \{((ends\ proc)\ from?), change\})]
\end{aligned}$$

Specifically, it insists on the following constraints.

- Sequence flow *from*? is not the same as sequence flow $f2?$; this ensures the sequence flows of the split and join gateways do not intersect.
- Sequence flow $t2?$ is fresh from the end event *end*?.
- Both end event ((*ends proc*) *from*?) and element *change* are directly contained in the same subprocess of the before state; this ensures the iteration is constructed within a single process.
- End event ((*ends proc*) *from*?) is a successor of element *change*; this ensures that the iteration is constructed along a continuous sequence of sequence flows within the process.

The last line defines the after state by replacing elements ((*ends proc*) *from*?) and *change* with elements *split*?, *join*?, *end*? and *change*′. Figure 4.6(E) shows how the end event ((*ends proc*) *from*?), labelled E in the figure, is replaced by elements *split*?, *end*? and *join*?, where *end*? is labelled F in the figure. The figure also shows how element *change* is replaced by its after state *change*′ and *join*?, and how *split*? and *join*? are connected by *connect*?.

The iteration operation *Loop* is defined as a conjunction of *ConnectSplit*, *ConnectJoin* and *Connect*. The components *change* and *change*′ are hidden as they are defined in terms of other input components and constraints specified in the schema.

$$Loop \mathrel{\widehat=} (ConnectSplit \land ConnectJoin \land Connect) \setminus (change, change')$$

We now consider the iteration operation *EventLoop*. This operation constructs an iteration on a BPMN pool using an event-based XOR split gateway and a (non event-based) join gateway. An illustration of *EventLoop* is shown in Figure 4.6(F). Note that by using an event-based split gateway, this operation also needs to ensure that each direct successor of the split gateway is either a task or an intermediate event. Similar to operation *Loop*, *EventLoop* is defined as a conjunction of *ConnectEventSplit*, *ConnectJoin* and *EventConnect*.

$$EventLoop \mathrel{\widehat=} (ConnectEventSplit \land ConnectJoin \land EventConnect) \setminus (change, change')$$

ConnectEventSplit specifies the constraints on the event-based split gateway. *ConnectJoin*, which has already been defined, specifies the constraints on the join gateway. *EventConnect* specifies the interdependent constraints on the gateways and the operation on the state *Pool*. We first consider the constraints on the event-based split gateway. These are provided by the following schema *ConnectEventSplit*.

ConnectEventSplit

$\Delta Pool$
$CommonConstraints[split?/new?]$
$EventLoopEvents$
$end? : Element$
$connect? : Seqflow$

$split? \in \{EventSplit \bullet ele\}$
$end? \in \{ele : Element \mid InitialEnd\}$
$getIns(events?) = (atom\ split?).out$
$getOuts\ events? = \{connect?\} \cup (atom\ end?).in$
$getSeqflows\ events? \cap getSeqflows\ proc = \emptyset$
$(\{connect?\} \cup (atom\ end?).in) \cap getIns\ events? = \emptyset$
$connect? \notin (atom\ end?).in$

As well as including schema *CommonConstraints*, which specifies the constraints on *split*? and *from*? with respect to the before state *Pool*, *ConnectEventSplit* also includes the following schema *EventLoopEvents*.

EventLoopEvents

$events? : \mathbb{F}_1\ Element$

$events? \subseteq \{ele : Element \mid OneInOutAtom\}$
$events? \in uniqueIns$
$\#events? = 2$

Specifically, *ConnectEventSplit* describes the following constraints on the input components *split*?, *end*?, *connect*?, *events*? and *from*?.

- *split* is an event-based XOR split gateway.
- *end*? is a non-trigger end event.
- Incoming sequence flows of elements in *events*? are *split*?'s outgoing sequence flows.
- Outgoing sequence flows of elements in *events*? are the flow *connect*? and the incoming sequence flows of *end*?.
- Sequence flows of elements in *events*? are fresh from the before state.
- The set of incoming sequence flows of elements in *events*? is disjoint from $\{connect?\} \cup (atom\ end?).in$.
- *connect*? is not an incoming sequence flow of *end*?.
- Elements in *events*? do not share incoming sequence flows.
- Set *events*? has two elements, each of which is either a task or an intermediate event.

We now consider *EventConnect*, which is defined as follows.

$$
\begin{array}{l}
EventConnect \mathrel{\widehat=} \\
\quad [\Delta Pool;\ f2?, t2?, from? : Seqflow;\ events? : \mathbb{F}_1\ Element; \\
\quad\ change, change', end?, split?, join? : Element \mid \\
\qquad from? \neq f2? \land \\
\qquad t2? \notin getSeqflows(events?) \land \\
\qquad (proc, \{((ends\ proc)\ from?), change\}) \in together \land \\
\qquad (change, ((ends\ proc)\ from?)) \in (edge(direct(proc, change)))^+ \land \\
\qquad proc' = modify(proc, \{split?, join?, change', end?\} \cup events?, \{((ends\ proc)\ from?), change\})]
\end{array}
$$

This schema specifies the following constraints between the gateways and the operation on the state *Pool*.

- Sequence flow *from*? is not the same as sequence flow $f2?$; this ensures the sequence flows of the split and join gateways do not intersect.
- Sequence flow $t2?$ is fresh from elements in *events*?; this implies that $t2?$ is also fresh from *end*? and *split*?.
- Both end event $((ends\ proc)\ from?)$ and element *change* are directly contained in the same subprocess of the before state; this ensures that the iteration is constructed within a single process.
- End event $((ends\ proc)\ from?)$ is a successor of element *change*; this ensures that the iteration is constructed along a continuous sequence of sequence flows within the process.

The last line defines the after state by replacing elements $((ends\ proc)\ from?)$ and *change* with elements *split*?, *join*?, *end*?, $change'$ and the set of elements *events*?. Figure 4.6(F) shows exactly this, where *events*? consists of the event elements *eventA* and *eventB*, and *end*? is labelled *F* in the figure.

4.6.6 Interrupt

The interrupt operation attaches an intermediate event to an activity, thereby creating an exception flow for that activity. The interrupt operation is specified by the schema *AddException* and is defined as a disjunction of two operation schemas *AddNoRelatedErrorException* and *AddRelatedErrorException*.

$$
\begin{array}{l}
AddException \mathrel{\widehat=} \\
\quad (AddNoRelatedErrorException \lor AddRelatedErrorException) \setminus (change, change')
\end{array}
$$

Schema *AddNoRelatedErrorException* adds an exception flow to any activity element such that the exception is either a time lapse (*itime*), the arrival of a message (*imessage*) or an unspecified error in the activity (*ierror*(*anyexception*)). An illustration of *AddNoRelatedErrorException* is shown in Figure 4.6(G).

Conversely, schema *AddRelatedErrorException* creates an exception flow to a subprocess such that the exception can only be triggered by an error thrown by an end error event element directly contained in that subprocess. An illustration of *AddRelatedErrorException* is shown in Figure 4.6(H).

We now consider schema *AddNoRelatedErrorException* in detail.

$$AddNoRelatedErrorException \mathrel{\widehat=} GeneralException \land (AddNoRelatedErrorExceptionSub[change/ele, change'/ele'])$$

This operation is defined as a conjunction of schemas *GeneralException* and *AddNoRelatedErrorExceptionSub*. Schema *AddNoRelatedErrorExceptionSub* defines the after state of the activity element *change'* by adding the exception flow, that is the pair (*eflow*?, *etype*?), to component *exit* of before state *change*. Details of this operation have already been described on Page 39 in Section 4.6.1.2. We now consider the schema *GeneralException*.

GeneralException

$\Delta Pool$
$change, change' : Element$
$eflow?, loc? : Seqflow$
$end? : Element$

$eflow? \notin getSeqflows\ proc$
$end? \in \{InitialEnd \bullet ele\}$
$(atom\ end?).in = \{eflow?\}$
$(activities\ proc)\ loc? = change$
$proc' = modify(proc, \{end?, change'\}, \{change\})$

This schema defines the after state *Pool* and specifies the following constraints between the activity element *change*, the exception flow, (*eflow*?, *etype*?) to be added to *change* and the before state *Pool*.

- Sequence flow *eflows*? is fresh from the before state *Pool*.
- Element *end*? is a non-trigger end event element
- Sequence flow *eflow*? is the only incoming sequence flow of *end*?
- Element *change* is an activity contained in the before state and has an incoming sequence flow *loc*?.

The after state component *proc'* is then defined by replacing *change* with *end*? and *change'*. Figure 4.6(G) shows that *change'* is defined by adding an exception flow to *change* and both *change'* and *end*? are added to the after state *Pool'*.

We now consider schema *AddRelatedErrorException*. This operation creates an exception flow to a subprocess element such that the exception is an error thrown by an end error event element directly contained in that subprocess. This is defined as a conjunction of three schemas *GeneralException*, *CanAddExit* and *AddRelatedErrorExceptionSub*.

$$AddRelatedErrorException \mathrel{\widehat=} (GeneralException \land CanAddExit \land (AddRelatedErrorExceptionSub[change/ele, change'/ele'])) \setminus (endele, endele')$$

We have already described *GeneralException* and a description of *AddRelatedErrorExceptionSub* can be found on Page 40 in Section 4.6.1.2. Schema *CanAddExit* ensures that *change* is a subprocess contained in the before state *Pool* and has an incoming sequence flow *loc*?.

$$CanAddExit \mathrel{\widehat=} [proc : Process;\ change : Element;\ loc? : Seqflow \mid (subs\ proc)\ loc? = change]$$

Figure 4.6(H) shows that subprocess $change'$ is defined by adding an exception flow to *change* and updating the non-trigger end event labelled *endele* directly contained in *change* to the corresponding end error event. We hide components *endele* and $endele'$ as they are defined in terms of the other input components and the before state

4.6.7 Collaboration

Operations described so far are defined on state schema *Pool* (BPMN pool), The collaboration operation, on the other hand, is defined on the state schema *Diagram* (BPMN diagrams), which contains one or more BPMN pools. Specifically, this operation is defined by the following schema *ConnectMgeFlowDiagram*.

$$\begin{aligned}&ConnectMgeFlowDiagram \mathrel{\widehat=} \\ &\quad \exists \Delta Pool[proc1/proc, proc1'/proc'] \bullet (\exists \Delta Pool[proc2/proc, proc2'/proc'] \bullet \\ &\qquad AddSendMgeFlow[proc1/proc, proc1'/proc', tos?/to?] \land \\ &\qquad AddReceiveMgeFlow[proc2/proc, proc2'/proc', tor?/to?] \land DiagramPromote)\end{aligned}$$

This operation is defined by promoting local operations on two separate BPMN pools in the before state *Diagram*. These local operations are *AddSendMgeFlow* and *AddReceiveMgeFlow*, and the promotion schema is *DiagramPromote*. The operation *AddSendMgeFlow* updates a BPMN pool by adding an outgoing message flow to an element contained in that pool. We describe this in Section 4.6.7.1. Conversely, the operation *AddReceiveMgeFlow* updates a BPMN pool by adding an incoming message flow to an element contained in that pool. This operation is described in Section 4.6.7.2. The schema *DiagramPromote* defines the promotion of these two operations to a BPMN diagram. The promotion schema is described in Section 4.6.7.3.

We now provide some preliminary definitions to assist the construction of the collaboration operation. The schemas *SMgeEvent*, *IMgeEvent* and *EMgeEvent* define a start, an intermediate and an end message event that has no associated message flow respectively.

$$SMgeEvent \mathrel{\widehat=} [Start \mid (atom\ ele).type = smessage(nomessage)]$$

$$IMgeEvent \mathrel{\widehat=} [Inter \mid (atom\ ele).type = imessage(nomessage)]$$

$$EMgeEvent \mathrel{\widehat=} [End \mid (atom\ ele).type = emessage(nomessage)]$$

We also provide two functions *msgrecs* and *msgsends*.

$$msgrecs, msgsends : Process \to (Seqflow \nrightarrow Element)$$

Function *msgrecs* is defined such that $msgrecs(p)$ returns a function that associates end message events contained in process p with their incoming sequence flows. Conversely, function *msgsends* is defined such that $msgsends(p)$ returns a function that associates start and intermediate message events contained in process p with their outgoing sequence flows.

4.6.7.1 Adding an Outgoing Message Flow

The operation *AddSendMgeFlow* is defined as follows.

$$\begin{aligned}&AddSendMgeFlow \mathrel{\widehat=} \\ &\quad ((AddSendEventMgeFlow \lor AddSendTaskMgeFlow) \land MgeFlowConstraints) \setminus (change, change')\end{aligned}$$

This operation adds an outgoing message flow to either an end message event, which is defined by *AddSendEventMgeFlow*, or a task, which is defined by *AddSendTaskMgeFlow*.

AddSendEventMgeFlow

$$\begin{aligned}&AddMgeEvent[change/ele, change'/ele'] \\ &proc : Process \\ &to? : Seqflow \\ &\hline &(msgsends\ proc)\ to? = change\end{aligned}$$

Operation *AddSendEventMgeFlow* includes operation schema *AddMgeEvent* that adds a message flow to message event *change*. *AddSendEventMgeFlow* ensures that *change* is an end message event contained in *proc* and has an incoming sequence flow *to?*. *AddMgeEvent* is described in Section 4.6.1.

AddSendTaskMgeFlow

$$AddSendMgeFlowTask[change/ele, change'/ele']$$
$$proc : Process$$
$$to? : Seqflow$$
$$(tasks\ proc)\ to? = change$$

Similarly, operation *AddSendTaskMgeFlow* includes operation schema *AddSendMgeFlowTask* that adds an outgoing message flow to task *change*. *AddSendTaskMgeFlow* ensures that *change* is a task contained in *proc* and has an incoming sequence flow *to?*. *AddSendMgeFlowTask* is described in Section 4.6.1.

The operation *AddSendMgeFlow* also includes the following schema *MgeFlowConstraints*.

MgeFlowConstraints

$$\Delta Pool$$
$$change, change' : Element$$
$$msg? : Mgeflow$$
$$msg? \notin getMsgs(proc)$$
$$proc' = modify(proc, \{change'\}, \{change\})$$

This schema ensures that the new message flow is fresh from the before state and that the after state is defined by replacing *change* with its after state *change'*.

4.6.7.2 Adding an Incoming Message Flow

The operation *AddReceiveMgeFlow* is defined as follows.

$$AddReceiveMgeFlow \mathrel{\widehat=} ((AddRecEventMgeFlow \lor AddRecTasktMgeFlow \lor AddRecExpMgeFlow) \land MgeFlowConstraints) \setminus (change, change')$$

This operation adds an incoming message flow to either a start or an intermediate message event, which is defined by *AddRecEventMgeFlow*, or a task, which is defined by *AddSendTaskMgeFlow*, or an intermediate message event attached to an activity, which is defined by *AddRecExpMgeFlow*.

AddRecEventMgeFlow

$$AddMgeEvent[change/ele, change'/ele']$$
$$proc : Process$$
$$to? : Seqflow$$
$$(msgrecs\ proc)\ to? = change$$

Operation *AddRecEventMgeFlow* includes operation schema *AddMgeEvent* that adds a message flow to message event *change*. The constraint part of *AddRecEventMgeFlow* ensures *change* is either a start or an intermediate message event contained in *proc* and has an outgoing sequence flow *to?*. *AddMgeEvent* is described in Section 4.6.1.

AddRecTasktMgeFlow

$$AddReceiveMgeFlowTask[change/ele, change'/ele']$$
$$proc : Process$$
$$to? : Seqflow$$
$$(tasks\ proc)\ to? = change$$

Operation *AddRecTasktMgeFlow* includes operation schema *AddReceiveMgeFlowTask* that adds an incoming message flow to a task *change*. The constraint part of *AddRecTasktMgeFlow* ensures *change* is a task contained in *proc* and has an incoming sequence flow *to*?. *AddReceiveMgeFlowTask* is described in Section 4.6.1.

AddRecExpMgeFlow

$$
\begin{array}{l}
AddExceptionMgeFlow[change/ele, change'/ele'] \\
proc : Process \\
to? : Seqflow \\
\hline
(activities\ proc)\ to? = change
\end{array}
$$

Operation *AddRecExpMgeFlow* includes operation schema *AddExceptionMgeFlow* that adds an incoming message flow to an intermediate message event attached to element *change*. The constraint part of *AddRecExpMgeFlow* ensures *change* is an activity contained in *proc* and has an incoming sequence flow *to*?. *AddExceptionMgeFlow* has been described on Page 40 in Section 4.6.1.

4.6.7.3 Promoting to *Diagram*

Message flows form interactions between BPMN pools in a BPMN diagram. Having defined the operations that add message flows to *Pool*, we promote them for *Diagram*. This is provided by the following schema *DiagramPromote*.

DiagramPromote

$$
\begin{array}{l}
\Delta Diagram \\
\Delta Pool[proc1/proc, proc1'/proc'] \\
\Delta Pool[proc2/proc, proc2'/proc'] \\
id1?, id2? : PoolId \\
msg? : Mgeflow \\
\hline
id1? \neq id2? \\
msg? \notin getMsgs(\bigcup\{p : \mathrm{ran}\ pool \bullet p.proc\}) \\
\{id1?, id2?\} \subseteq \mathrm{dom}\ pool \\
\{id1?, id2?\} \lhd pool = \{id1?, id2?\} \lhd pool' \\
pool\ id1? = \langle\!| proc \rightsquigarrow proc1 |\!\rangle \\
pool'\ id1? = \langle\!| proc \rightsquigarrow proc1' |\!\rangle \\
pool\ id2? = \langle\!| proc \rightsquigarrow proc2 |\!\rangle \\
pool'\ id2? = \langle\!| proc \rightsquigarrow proc2' |\!\rangle
\end{array}
$$

This schema specifies the following global constraints:

- *id*1? identifies a BPMN pool in the before state *Diagram*, to which *msg*? is added as an outgoing message flow; this is specified by the binding the after state $\langle proc \rightsquigarrow proc1' \rangle$.
- *id*2? identifies a BPMN pool in the before state, to which *msg*? is added as an incoming message flow; this is specified by the binding the after state $\langle proc \rightsquigarrow proc2' \rangle$.
- Identifiers *id*1? and *id*2? are different, thereby ensuring this operation uses *msg*? to connect two different pools in the before state *Diagram*.
- *msg*? must be fresh from the before state *Diagram*.

Figure 4.6(I) shows how BPMN pools labelled *id*1? and *id*2? are connected. It shows that *msg*? is added as an outgoing message flow to a task contained in pool *id*1?, while it is also added as an outgoing message flow to a task contained in pool *id*2?.

4.6.8 Example

We now demonstrate how to construct the customer business process of our online shop example in Figure 2.3. A step-by-step illustration of the business process construction is shown in Figure 4.7. The following describes the steps shown in the figure.

- Starts with a subprocess' initial state, as defined by the schema *GenProcInit* on Page 35 (Step 1).
- Adds a data-based XOR split gateway using *Split* (Step 2).
- Applies *SeqComp* four times to add three task elements and one subprocess element. The subprocess element is in an initial state, as defined by the schema *GenProcInit* (Steps 3, 4, 5, 6).
- Applies *Split* to add a data-based XOR split gateway inside the subprocess element that was added in Step 6 (Step 7).
- Adds two task elements inside the subprocess element using *SeqComp* two times (Steps 8, 9).
- Joins two end elements inside the subprocess element using *JoinOp* (Step 10).

4.6.9 Preconditions

In Sections 4.6.2 – 4.6.7 we defined eight operations on BPMN pool (*Pool*) and diagram (*Diagram*). In this section we investigate their consistency with respect to state schema *Pool* and *Diagram*. Specifically, for operation schemas *SeqComp*, *Split*, *EventSplitOp*, *JoinOp*, *Loop*, *EventLoop*, *AddException* and *ConnectMgeFlowDiagram*, we show that given a before state *Pool* and constraints on the input components specified by the operation schemas, the after state would satisfy *Pool*. For operation schema *ConnectMgeFlowDiagram*, we show that, given before state *Diagram* and constraints on the input components specified by the operation, the after state would satisfy *Diagram*. We therefore calculate the preconditions of the operation schemas.

For operations *SeqComp*, *Split*, *EventSplitOp* and *JoinOp*, the strategy for precondition calculation is as follows: We first expand the precondition expression pre *Op* where *Op* is one of *SeqComp*, *Split*, *EventSplitOp* and *JoinOp*. We then apply the one-point rule [Spi92] to eliminate the existential quantifications over the after state. Finally, we show that the expression that defines the after state follows from the before state and the constraints on the input components. As an example, we informally describe the precondition calculation of operations *SeqComp* and *AddNoRelatedErrorException*. Full precondition calculations for all operations may be found in Appendix B.

4.6.9.1 Calculating pre *SeqComp*

We first expand pre *SeqComp* and apply the one-point rule to arrive at the following schema.

$$
\begin{array}{l}
[Pool;\ new?, end? : Element;\ from? : Seqflow \mid \\
\quad (outs\ new? \cup (getSeqflows\ (content\ new?))) \cap getSeqflows\ proc = \emptyset \land \\
\quad from? \in (atom\ new?).in \land \\
\quad (atom\ new?).in \subseteq \mathrm{dom}(ends\ proc) \land \\
\quad new? \in \{ele : Element \mid OneInOutObject\} \land \\
\quad end? \in \{ele : Element \mid InitialEnd\} \land \\
\quad (atom\ end?).in = (atom\ new?).out \land \\
\quad modify(proc, \{new?, end?\}, \{((ends\ proc)\ from?)\}) \in \{proc : Process \mid Pool\}]
\end{array}
$$

We then show the constraint

$$modify(proc, \{new?, end?\}, \{((ends\ proc)\ from?)\}) \in \{proc : Process \mid Pool\}$$

follows from the before state *Pool* and the constraints on input components *from*?, *new*? and *end*?. Since *modify* is defined recursively, we apply induction on the depth of the following expression.

$$modify(proc, \{new?, end?\}, \{((ends\ proc)\ from?)\})$$

For the base case, where $((ends\ proc)\ from?) \in ps$, the function *modify* returns the set of elements $((ps \cup \{new?, end?\}) \setminus \{((ends\ proc)\ from?)\})$. From the before state we know that $ps \in \{proc : Process \mid Pool\}$, we therefore expand the constraints specified by schema *Pool* and show that the expression satisfies the *proc* component of *Pool*.

For the inductive step where $((ends\ proc)\ from?) \in (\{e : Element \mid e \in_{\mathrm{p}} ps\} \setminus ps)$ we show the following implication holds.

$$(\forall s : cs \bullet mf(s) \in \{proc : Process \mid Pool\}) \Rightarrow \\ ((ps \setminus cs) \cup \{s : cs \bullet rep(s, mf(s))\}) \in \{proc : Process \mid Pool\}$$

where cs and $mf(s)$ are defined as follows.

$$cs = cont(ps, \{((ends\ proc)\ from?)\}) \\ mf(s) = modify(content(s), \{new?, end\}, \{((ends\ proc)\ from?)\})$$

Specifically, given ps is a process satisfying *proc* component of *Pool* and that for all subprocess elements $s \in ps$, expressions $mf(s)$ are processes satisfying the *proc* component of *Pool*, we show that the substituting them into the content of s ($content(s)$), that is the consequent, would also be a process satisfying the *proc* component of *Pool*. Here we refer to $mf(s)$ as the *substitution process* for subprocess s. For example, we consider Step 10 of our step-by-step example in Figure 4.7. In this step, the process in the subprocess element is substituted with another process by applying operation *JoinOp*. Assuming the customer business process after Step 9 as well as the substitution process for the subprocess satisfy *Pool*, we show the business process after Step 10 also satisfies *Pool*.

We observe in general for any given process ps, and element s contained in ps, the implication holds if the substitution process qs for s satisfies the following four constraints.

- Processes ps and qs satisfy the *proc* component of *Pool*.
- Element s is a compound element.
- Both sequence and message flows of elements contained in qs are fresh from elements that are contained in ps but are not contained in subprocess s.
- The substitution process qs and the original process $content(s)$ directly contain the same end error events.

We formulate these constraints into the following lemma.

Lemma 4.12.

$$\forall ps, qs : Process \bullet \\ ((ps \in \{proc : Process \mid Pool\} \land qs \in \{proc : Process \mid Pool\}) \Rightarrow \\ (\forall s : ps \bullet \\ s \in \mathrm{ran}\ compound \land \\ getSeqflows(qs) \cap (getSeqflows(ps \setminus \{s\}) \cup (atom\ s).in \cup outs(s)) = \emptyset \land \\ getMsgs(qs) \cap (getMsgs(ps \setminus \{s\}) \cup getMsg\ (atom\ s)) = \emptyset \land \\ \{e : qs \mid (atom\ e).type \in \mathrm{ran}\ eerror\} = \{e : content(s) \mid (atom\ e).type \in \mathrm{ran}\ eerror\} \Rightarrow \\ ((ps \setminus \{s\}) \cup \{rep(s, qs)\}) \in \{proc : Process \mid Pool\}))$$

Proof. See Page 197 (Section B.1 in Appendix B). □

Using this result we simplify the precondition of *SeqComp* into the following schema, labelled *PreSeqComp*, where *CommonConstraint* has already been described in Section 4.6.2. Full derivation of *PreSeqComp* can be found in Section B.9.

PreSeqComp

$Pool$
$CommonConstraints$
$end? : Element$

$new? \in \{ele : Element \mid OneInOutObject\}$
$end? \in \{ele : Element \mid InitialEnd\}$
$(atom\ end?).in = (atom\ new?).out$

The preconditions of *Split*, *EventSplitOp* and *JoinOp* are calculated using the same strategy. Their precondition calculations are documented in Sections B.10, B.11 and B.12 respectively.

4.6.9.2 Calculating pre *AddNoRelatedErrorException*

In this section we consider one of the interrupt operations *AddNoRelatedErrorException*, which was defined in Section 4.6.6. We first expand pre *AddNoRelatedErrorException* and apply the one-point rule to arrive at the following schema.

$$
\begin{array}{l}
[Pool;\ etype? : Type;\ eflow?, loc? : Seqflow;\ end? : Element \mid \\
\quad \textbf{let}\ ed == (activities\ proc)\ loc? \bullet \\
\quad\quad \textbf{let}\ ch == ce(ed, (eflow?, etype?), \emptyset, \emptyset) \bullet \\
\quad\quad\quad \textbf{let}\ md == modify(proc, \{end?, ch\}, \{ed\}) \bullet \\
\quad \{ed, ch\} \subseteq \{ele : Element \mid Activity\} \land \\
\quad etype? \in \operatorname{ran} itime \cup \{imessage(nomessage)\} \cup \{i : \operatorname{ran} ierror \mid (ierror^{\sim})\ i \notin \operatorname{ran} exception\} \land \\
\quad eflow? \notin getSeqflows\ proc \land \\
\quad end? \in \{InitialEnd \bullet ele\} \land \\
\quad (atom\ end?).in = \{eflow?\} \land \\
\quad ed \in Element \land \\
\quad md \in \{proc : Process \mid Pool\}]
\end{array}
$$

Unlike *SeqComp*, this operation updates the activity $ed == (activities\ proc)\ loc?$ contained in the before state *Pool* by adding to it a new exception flow. The after state of ed is defined by the abbreviation $ch == ce(ed, (eflow?, etype?), \emptyset, \emptyset)$. This update is specified by operation schema *AddNoRelatedErrorExceptionSub*. We first show the constraint

$$\{ed, ch\} \subseteq \{ele : Element \mid Activity\} \tag{4.1}$$

follows from the before state *Pool* and the constraints on input components $loc?$, $etype?$, $eflow?$ and $end?$. By definition of *Pool* and function *activities*, we know ed satisfies the constraints on the ele component of schema *Activity*. We also need to show that ch satisfies the constraints on the ele component of *Activity*. To do this we first obtain the following schema as the precondition of *AddNoRelatedErrorExceptionSub*, labelled as *PreAddNoRelatedErrorExceptionSub*. The full derivation of *PreAddNoRelatedErrorExceptionSub* can be found in Section B.3.

PreAddNoRelatedErrorExceptionSub

$Activity$
$etype? : Type$
$eflow? : Seqflow$

$etype? \in nomsgserrors$
$eflow? \notin getSeqflows\{ele\}$

Here we can see constraints on component $etype?$ and $eflow?$ specified by the expanded schema of pre *AddNoRelatedErrorException* satisfy those specified by pre *AddNoRelatedErrorExceptionSub*. We therefore conclude that the constraint defined in Equation 4.1 follows from the before state *Pool* and the constraints on input components.

We then show the constraint $md \in \{proc : Process \mid Pool\}$ follows from the before state *Pool* and the constraints on input components, where $md == modify(proc, \{end?, ch\}, \{ed\})$. Similar to the previous example, we apply induction on md. We obtain the following precondition of *AddNoRelatedErrorException*, labelled *PreAddNoRelatedErrorException*. The full derivation of *PreAddNoRelatedErrorException* can be found in Section B.15.1.

PreAddNoRelatedErrorException

$Pool$
$etype? : Type$
$eflow?, loc? : Seqflow$
$end? : Element$

$loc? \in \mathrm{dom}(activities\ proc)$
$etype? \in nomsgserrors$
$eflow? \notin getSeqflows\ proc$
$end? \in \{InitialEnd \bullet ele\}$
$(atom\ end?).in = \{eflow?\}$

The preconditions of *Loop*, *EventLoop*, *AddRelatedErrorException* and *ConnectMgeFlowDiagram* are calculated using this strategy. Their precondition calculations are documented in Sections B.13, B.14, B.15.2 and B.16 respectively.

4.7 Summary

This chapter provided a detailed analysis of BPMN syntax and presented a corresponding implementation in Haskell. We specified initial states of BPMN pools and diagrams, and defined a set of schema operations for constructing BPMN diagrams. Our aim is to provide semantics on this syntax to formalise the behaviour of communications between elements in a BPMN diagram. This leads naturally to our semantic constructions, which are presented in Chapters 5 and 6.

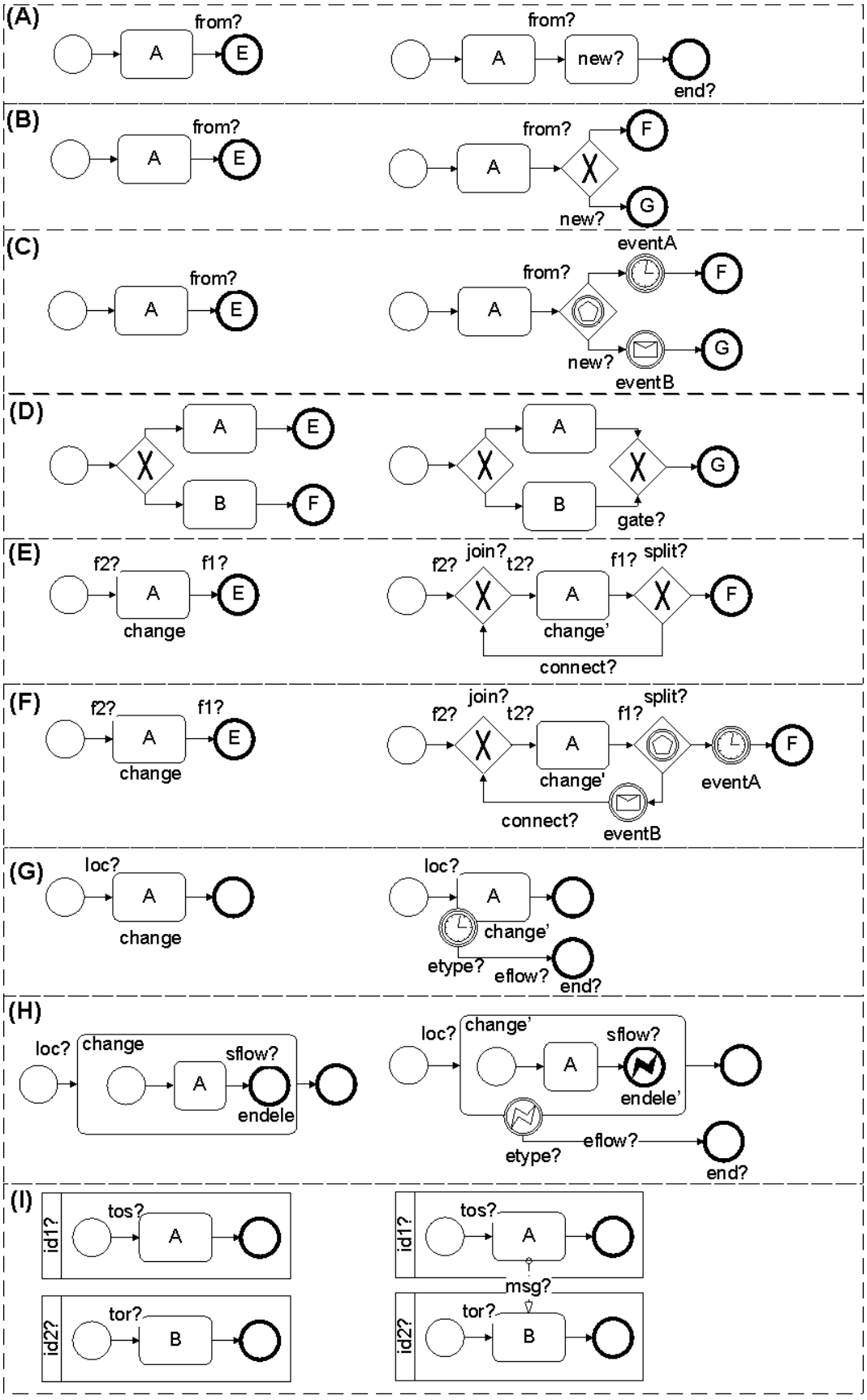


Figure 4.6: Before-and-after illustrations of operation schemas

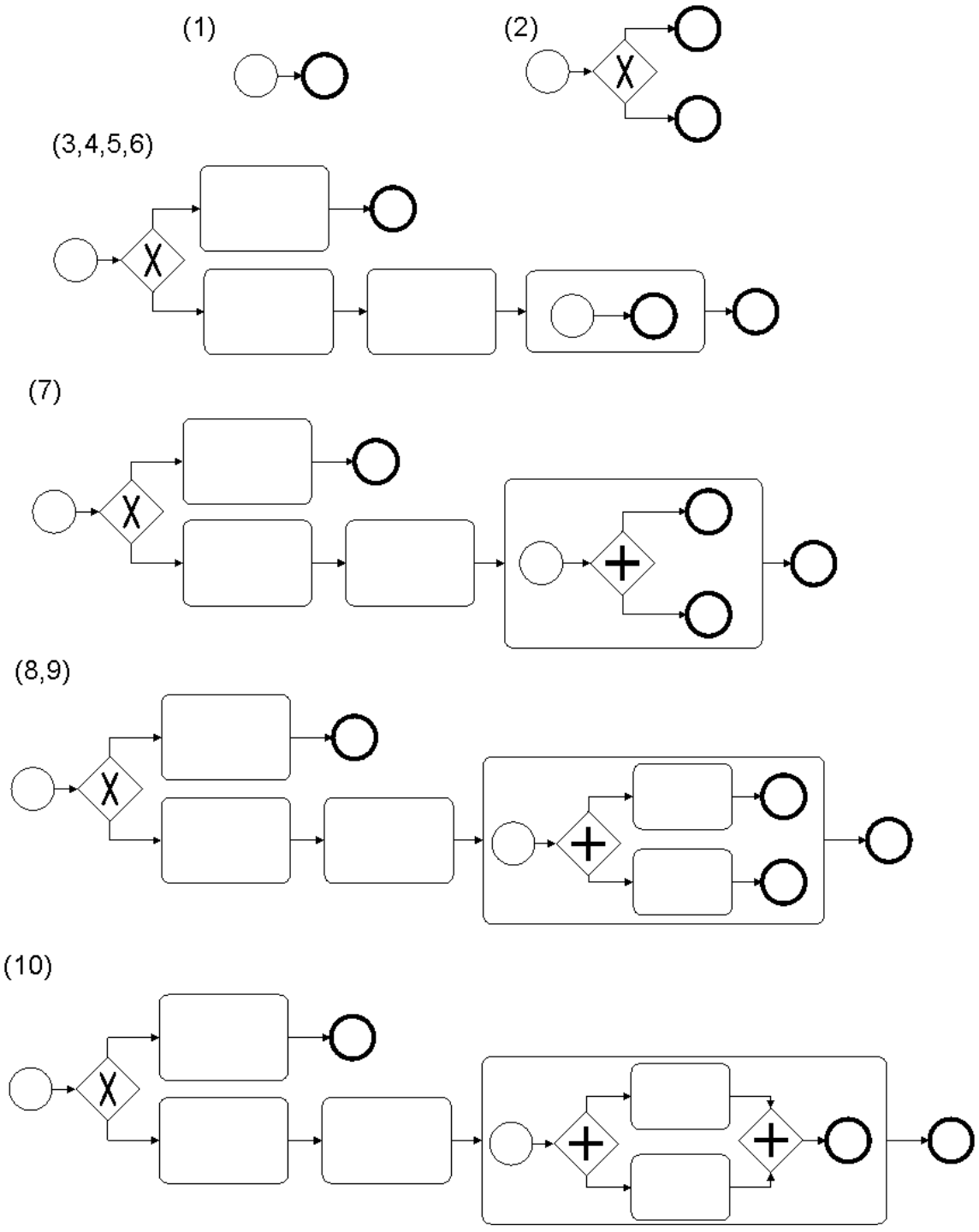


Figure 4.7: Syntactic construction of the customer business process

Chapter 5

Process Semantics

5.1 Introduction

In this chapter we define a process semantics for our subset of BPMN using the process algebra CSP [Ros98]. We show how CSP refinement orderings can be applied to the specification and the verification of business process behaviour. Our semantic definition can be readily analysed using the FDR tool [For98].

5.1.1 Approach

For each type of BPMN element, we provide an informal description of its behaviour and then present its semantic definition. We implement the semantic function in Haskell; for presentation purposes, some functions are partially presented in this chapter — their full definitions may be found in Appendix D. More specifically, we define the function `bToc` in this chapter.

```
bToc :: Diagram -> Script
```

This function takes a BPMN diagram and returns a CSPm definition that models communications between elements in that diagram. We model each BPMN diagram as a parallel composition of processes, each corresponding to a BPMN pool in that diagram. Similarly, we model each BPMN pool as a parallel composition of processes, each corresponding to an element the pool directly contains. The output value of `bToc` has the type `Script` and it records the resulting CSPm definition; the definition of `Script` may be found in Section 2.4.4.

5.1.2 Structure

The rest of this chapter is structured as follows. In Section 5.2 we define functions to associate each sequence flow, message flow, element and diagram to its possible behaviours. Section 5.3 presents the semantics of atomic elements. Sections 5.4 and 5.5 present the semantics of compound elements. In Section 5.6 we show how to specify safety and liveness properties of BPMN processes and to verify BPMN processes against these properties. In Section 5.7 we provide a CSP semantics to the syntactic operations defined in Section 4.6. In Section 5.9 we study the notion of compatibility between business processes. We summarise the contribution of this chapter in Section 5.10.

5.2 Alphabet

In CSP the alphabet of a process is the set of events that process can perform. We therefore associate each object in BPMN to a set of CSP events to denote its possible behaviours.

We first consider sequence, message and exception flows. We associate each sequence flow and message flow to a CSP event. This is defined by the following functions `seqflow` and `mgeflow`,

```
seqflow :: Seqflow -> Event
seqflow s = "s."++s
mgeflow :: Mgeflow -> Event
mgeflow m = "m."++m
```

where the operator `++` concatenates two strings together. At the implementation level, each sequence and message flow is uniquely identified by a Haskell `String` value. Similarly, each CSP event in CSPm is also a `String` value. As a result we map each sequence flow to a CSP event by prefixing it with "`s.`" and each message flow to a CSP event by prefixing it with "`m.`". For example the set $\{\!|s|\!\}$ (or `{|s|}` in CSPm) refers to all events associated with sequence flows.

Semantically an exception flow can be modelled in the same way as a sequence flow, that is as a CSP event with "`s.`" as its prefix. The difference lies in how an exception flow is triggered. Specifically, an activity triggers an exception flow if an error occurs during its execution. In particular, an end error event contained in a subprocess throws a specific error of type `ErrorCode` when triggered. We denote errors using the function `except`, which maps an `ErrorCode` value to a CSP event.

```
except :: ErrorCode -> Event
except e = "e."++e
```

We now consider elements in BPMN. We are interested in modelling communications between elements in a BPMN diagram. As a result gateways, message, none, rule and timer events may be characterised by their sequence and message flows. This is because their interactions with their environment are only those associated with their incoming and outgoing flows. We also model work done in task elements. This is because we consider activities in tasks to be visible, so that we can analyse the effects on them caused by the communication between elements in a diagram; we define function `task` to map each task name of type `TaskName` to a CSP event. Here the prefix "`w.`" groups all events associated with work done.

```
task :: TaskName -> Event
task t = "w."++t
```

Note that we do not explicitly associate CSP events to activities in subprocesses and pools; their behaviours are characterised by the behaviours of the elements they contain. We define function `alpha` to map each element to its alphabet.

```
alpha :: Element -> [Event]
alpha (Atomic (Atom d t i o e t r s)) =
       let
         te = if (istask t) then [(task . taskname) t] else []
         ec = if (iserror t) && (hasexcept t) then [(except . errorcode) t] else []
         ms = if (ismessage t) && (hasmessage t) then [(mgeflow . eventmge) t] else []
       in (map mgeflow (r++s)) ++ (map seqflow (i++o)) ++ te ++ ec ++ ms
alpha (Compound a es) = (alpha (Atomic a)) ++ (concatMap alpha es)
```

Functions `istask`, `iserror` and `ismessage` take a `Type` value and check if it is associated with a task element, an error event or a message event respectively; the function `hasexcept` checks if a `Type` value records an error code, and functions `taskname` and `errorcode` map a `Type` value to a taskname of type `TaskName` and an error code of type `ErrorCode` respectively.

```
istask, iserror, ismessage, hasexcept :: Type -> Bool
taskname :: Type -> TaskName
errorcode :: Type -> ErrorCode
```

Function `alpha` is defined using predefined Haskell functions `map` and `concatMap`: `map f xs` is the list obtained by applying `f` to each value in `xs`, while `concatMap f xs` is the list obtained by applying `f` over each value in list `xs` and concatenating the results.

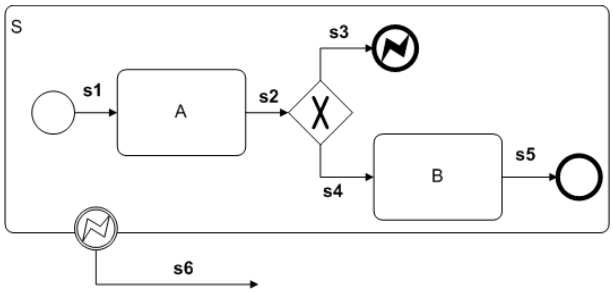


Figure 5.1: A simple BPMN subprocess

Figure 5.1 shows a simple BPMN subprocess; each element, and indeed the whole diagram, is associated with an alphabet. For example, the alphabet of task element A is `["w.A","s.s1","s.s2"]`; that for task B is `["w.B","s.s4","s.s5"]`; and for the data-based XOR gateway is `["s.s2","s.s3","s.s4"]`.

5.3 Atomic Elements

Atomic elements are events, gateway, and task elements. We model an atomic element as a sequential composition of its incoming flows behaviour, its work done and its outgoing flows behaviour. For example,

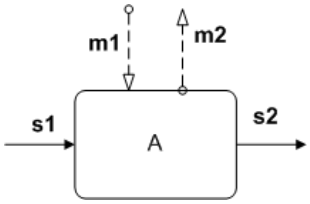


Figure 5.2: An atomic task element

Figure 5.2 shows an atomic task element; its behaviour is modelled by the following CSP process A.

$$A = (m.m1 \rightarrow \mathit{Skip} \;|||\; s.s1 \rightarrow \mathit{Skip}) \fatsemi w.A \rightarrow \mathit{Skip} \fatsemi m.m2 \rightarrow \mathit{Skip} \fatsemi s.s2 \rightarrow \mathit{Skip} \tag{5.1}$$

This process is defined by sequentially composing processes describing the behaviour of its incoming message and sequence flows, work done and outgoing message and sequence flows. This process does not insist on the order of incoming sequence flows and message flows, as it is not specified in the official document [OMG08].

We define mappings from each part of an element to a process that corresponds to its behaviour. We consider sequence flows in Section 5.3.1, and message flows in Section 5.3.2. We present the semantics of events, gateways and tasks in Sections 5.3.3, 5.3.4 and 5.3.5 respectively, and the semantics of task's exception flows in Section 5.3.6.

5.3.1 Sequence Flows

Based on the Z specification of the BPMN syntax, events, activities and split gateways have one incoming sequence flow, while join gateways have multiple incoming sequence flows. A XOR join gateway is triggered when one of its incoming sequence flows is triggered, and an AND join gateway, on the other hand, is triggered when all of its incoming sequence flows are triggered.

In CSP, we model the incoming sequence flows of a XOR gateway as a process that externally chooses one of the events associated with the flows, and the incoming sequence flows of an AND gateway as a process that interleaves the events associated with the flows.

We define two functions `seqext` and `seqpar` to model the behaviour of incoming sequence flows. They take a list of sequence flows (`[Seqflow]`) and return a CSP process of type `Process`. The constructors `Indextern` and `Indinterl` represent the indexed external choice and interleaving operators in CSP respectively.

```
seqext,seqpar :: [Seqflow] -> Process
seqext [] = Skip
seqext ss = Indextern ("s",(List Set (map seqflow ss))) (Prefix "s" Skip)
seqpar ss = Indinterl ("s",(List Set (map seqflow ss))) (Prefix "s" Skip)
```

The function `seqext` returns a process that externally chooses one of the flows and performs the event that is associated with that flow. For example, given a list of sequence flows (`["s1","s2"]`), `seqext` returns the process $\Box\, s : \{\, s.s1, s.s2 \,\} \bullet s \rightarrow \mathit{Skip}$. In fact `seqext` defines the incoming sequence flow semantics for all BPMN elements except the AND gateway, which synchronises its incoming sequence flows. Start event elements have no incoming sequence flow, and this is modelled using the process *Skip*.

The function `seqpar` returns a process that interleaves the performance of the events associated with the flows. For example given the list `["s1","s2"]`, `seqpar` returns the process $|||\, s : \{\, s.s1, s.s2 \,\} \bullet s \rightarrow \mathit{Skip}$.

Events, activities and join gateways have one outgoing sequence flow, while split gateways have multiple outgoing sequence flows. A data-based XOR split gateway chooses one of its outgoing flows to trigger internally, while an event-based XOR split gateway chooses one of its outgoing flows to trigger

based on the behaviour of the gateway's succeeding elements. An AND split gateway triggers all of its outgoing sequence flows; at this level of abstraction, we do not restrict the order in which an AND split gateway triggers its outgoing sequence flows.

In CSP, we reuse function `seqpar` to model the behaviour of an AND split gateway's outgoing sequence flows and `seqext` to model that of an event-based XOR split gateway's outgoing sequence flows. Conversely, we define the function `seqint`, which takes a list of sequence flows (`[Seqflow]`) and returns a CSP process of type `Process`, to model the behaviour of a data-based XOR outgoing sequence flows; the constructor `Indintern` represents CSP's indexed internal choice operator.

```
seqint :: [Seqflow] -> Process
seqint ss = Indintern ("s",(List Set (map seqflow ss))) (Prefix "s" Skip)
```

The function `seqint` returns a process that internally chooses one of the flows and performs that flow's associated event. For example given the list `["s1","s2"]`, it returns the process $\sqcap s : \{ s.s1, s.s2 \} \bullet s \rightarrow Skip$.

5.3.2 Message Flows

Message flows represent the communications between different BPMN pools of the same diagram. Based on the Z specification of the BPMN syntax, message events and tasks have zero or more incoming and outgoing message flows. A start message event has at most one incoming message flow; an intermediate message event has at most one message flow, either incoming or outgoing, and an end message event has at most one outgoing message flow. Task elements, on the other hand, may have zero or more incoming and outgoing message flows. Specifically, a task element receives a message from one of its incoming message flows and sends a message to all of its outgoing message flows.

We define the function `mgeext` to model the behaviour of an element's incoming message flows and the function `mgepar` to model that of an element's outgoing message flows. Similar to `seqext`, `mgeext` returns the process *Skip* to model zero message flows.

```
mgeext, mgepar :: [Mgeflow] -> Process
mgeext [] = Skip
mgeext mm = Indextern ("m",(List Set (map mgeflow mm))) (Prefix "m" Skip)
mgepar mm = Indinterl ("m",(List Set (map mgeflow mm))) (Prefix "m" Skip)
```

5.3.3 Events

In this section we consider the behaviour of events. We define function `event` to take a value of type `Atom`, recording the syntactic information of an event and return a CSP process of type `Process` that models the event's behaviour.

```
event :: Atom -> Process
event (Atom n t i o e r im om)
 | ismessage t = mgesem t i o
 | iserror t = errsem t i o
 | otherwise = othersem t i o
```

The guards `ismessage t` and `iserror t` check whether or not the input atomic element is a message event or an error event respectively.

```
ismessage, ierror :: Type -> Bool
```

For a message event, `event` applies function `mgesem` to its type, incoming and outgoing sequence flows, and for an error event, the function `errsem` is applied similarly. If the event is neither a message nor an error event, the function `othersem` is applied. We now consider function `mgesem`,

```
mgesem :: Type -> [Seqflow] -> [Seqflow] -> Process
mgesem (Smessage m) is os = SeqComp ((mgeext . getvalue) m) (seqext os)
mgesem (Imessage m) is os = SeqComp (Inter ((mgeext . getvalue) m) (seqext is)) (seqext os)
mgesem (Emessage m) is os = SeqComp (seqext is) ((mgepar . getvalue) m)
```

where `getvalue` is a generic function that takes a `Maybe` value and returns either a singleton list or the empty list.

```
getvalue :: Maybe a -> [a]
```

The function `mgesem` defines the semantics of a message event as the sequential composition of its incoming message/sequence flows and outgoing message/sequence flows. We do not insist on the triggering order between incoming sequence and message flows in an intermediate message event because while the official documentation [OMG08] does not specify an explicit order, it does include these flows as the necessary conditions to trigger the event.

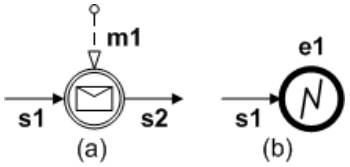


Figure 5.3: (a) message catch event, (b) error event

For example the semantics of the message event in Figure 5.3(a) is given by the following CSP process.

$$(m.m1 \rightarrow \mathit{Skip} \;|||\; s.s1 \rightarrow \mathit{Skip}) \fatsemi s.s2 \rightarrow \mathit{Skip}$$

We now consider the function `errsem`.

```
errsem :: Type -> [Seqflow] -> [Seqflow] -> Process
errsem (Ierror AnyException) is os = seqext os
errsem (Ierror (Exception e)) is os = Prefix (error e) (seqext os)
errsem (Eerror (Exception e)) is os = SeqComp (seqext is) (Prefix (error e) Skip)
```

This function defines the semantics of an error event. For an intermediate error event, it prefixes the event's outgoing sequence flow with a CSP event to denote an exception occurrence, thereby modelling the behaviour of catching an exception. We do not consider intermediate error events with incoming sequence flows; this is because intermediate error event may only be attached to an activity's boundary.

For an end error event, the function prefixes the event's exception behaviour with its incoming sequence flow, thereby modelling the behaviour of throwing an exception. If an intermediate error event is attached to an activity and is not associated with a specific error, its behaviour is then simply that of its outgoing sequence flows. For example the semantics of the end error event in Figure 5.3(b) is captured by the following CSP process, assuming the error code is $e1$.

$$s.s1 \rightarrow \mathit{Skip} \fatsemi e.e1 \rightarrow \mathit{Skip}$$

We now consider the function `othersem`, which defines the semantics of either a timer or a rule event.

```
othersem t is os = SeqComp (seqext is) (seqext os)
```

Our model abstracts from the timing information of a timer event and the rule condition of a rule event. Timing information is considered in Chapter 6, while rule conditions are used to record syntactic information when modelling empirical studies in Chapter 7. As a result, `othersem` models the semantics of one of these events as simply the sequential composition of the behaviour of events' incoming and outgoing sequence flows.

5.3.4 Gateways

In this section we consider gateways. We define function `gateway` to take an `Atom` value that describes a gateway, and return a `Process` value that records the CSP process modelling the gateway's behaviour.

```
gateway :: Atom -> Process
gateway a =
  case etype a of
    Xgate -> SeqComp ((seqext . ins) a) ((seqint . outs) a)
    Exgate -> SeqComp ((seqext . ins) a) ((seqext . outs) a)
    Agate -> SeqComp ((seqpar . ins) a) ((seqpar . outs) a)
```

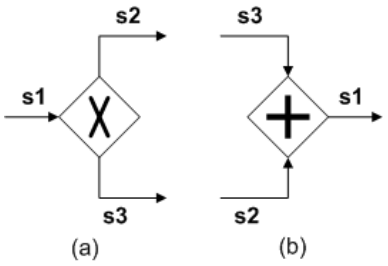


Figure 5.4: (a) XOR gateway and (b) AND gateway

For a data-based XOR gateway, `gateway` returns a process that first offers one of the gateway's incoming sequence flows and then internally chooses one of its outgoing sequence flows. For example the semantics of the data based XOR gateway shown in Figure 5.4(a) is defined by the following CSP process.

$$s.s1 \rightarrow \mathit{Skip} \fatsemi (s.s2 \rightarrow \mathit{Skip} \sqcap s.s3 \rightarrow \mathit{Skip})$$

We use the CSP internal choice operator to model the data based XOR gateway because our semantic model abstracts from process data, as a result the resolution of the exclusive choice condition is internal from the environment. Conversely, for an event-based XOR gateway, `gateway` returns a process that offers one of the gateway's incoming sequence flows and then offers one of its outgoing sequence flows. We use the CSP external choice operator to model this behaviour and this allows the environment to choose which of its outgoing flows to trigger. For an AND gateway, `gateway` returns a process that interleaves the gateway's incoming sequence flows, and then interleaves the gateway's outgoing sequence flows. For example the semantics of the AND gateway shown in Figure 5.4(b) is defined by the following CSP process.

$$(s.s2 \rightarrow \mathit{Skip} \mid\mid\mid s.s3 \rightarrow \mathit{Skip}) \fatsemi s.s1 \rightarrow \mathit{Skip}$$

5.3.5 Tasks

A task represents an atomic piece of work done in a business process; its type is defined by the constructor `Task`. We model an atomic piece of work as a single CSP event. According to the official specification [OMG08, Section 9.4.3.4], a task with outgoing message flows completes execution after all outgoing message flows are triggered. For example, consider again the task shown in Figure 5.2. It performs work A and sends a message along message flow $m2$. We model this as the process $w.A \rightarrow \mathit{Skip} \fatsemi m.m2 \rightarrow \mathit{Skip}$. As a result, we define the function `stask` that takes an `Atom` value describing a task, and returns a CSP process that models the behaviour of the task's work.

```
stask :: Atom -> Process
stask (Atom n t i o e r im om l) = SeqComp (Prefix (taskname t) Skip) (mgepar om)
```

There are also multiple instance tasks. A sequential multiple instance task, whose type is defined by the constructor `Miseq`, repeats its work sequentially, while a parallel multiple instance task, whose type is defined by the constructor `Mipar`, repeats its work concurrently and independently.

According to the BPMN's official documentation [OMG08], when a multiple instance task is triggered, it evaluates an expression that returns an integer value, and repeats its work for that number of times. Since our model abstracts from process data, we model a multiple instance task to choose this value nondeterministically ranging from one to some number recorded in the element's type constructor.

In addition, a multiple instance task records a `FlowType` value. It takes one of values `One` and `All`. If the value is `One`, the element triggers its outgoing sequence flow after one of its work instances has been executed; if the value is `All`, the element triggers its outgoing sequence flow after all of its work instances have been executed.

In the remaining part of this section, we consider the behaviour of multiple instance tasks. Note that there are also multiple instance subprocesses. While a subprocess's work is defined in terms of

the elements it directly contains, similar to a multiple instance task, each work instance in a multiple instance subprocess is independently executed. As a result, our semantic definition of multiple instance applies to both task and subprocess.

We present the behaviour of multiple instance in the following categories: fixed number of sequential instances is described in Section 5.3.5.1; nondeterministic number of sequential instances is described in Section 5.3.5.2; fixed number of parallel instances is described in Section 5.3.5.3; and nondeterministic number of parallel instances is described in Section 5.3.5.4.

5.3.5.1 Fixed Number of Sequential Instances

A multiple instance element in this category executes its work instances i times sequentially, where i is a finite non-zero positive number specified by the value `Fix i` recorded in the element's type. We define the function `fixseq`, which takes the number of iterations, the element's *flow type*, the element's outgoing sequence flows and the process that models the behaviour of its work, and returns a process that sequentially iterates the element's work and then triggers the element's outgoing sequence flows. Here `replicate n p` returns a list of n copies of p.

```
fixseq :: Int -> FlowType -> [Seqflow] -> Process -> ([Local],Process)
fixseq 1 f os p = sact os p
fixseq n One es os p = ([],SeqComp (SeqComp p (seqext os)) (seqcomps (replicate (n-1) p)))
fixseq n All es os p = ([],SeqComp (seqcomps (replicate n p)) (seqext os))
```

Specifically, this function is defined for three different combinations of iteration number and flow type: one iteration, multiple iterations with flow type value `One` and multiple iterations with flow type value `All`.

- For one iteration, the activity's flow type is irrelevant. We provide function `sact` to model the behaviour of one work iteration. For example Figure 5.5(a) shows a sequential multiple instance task. If it defines one iteration, the behaviour after triggering its incoming sequence flow is modelled by the CSP process $w.A \rightarrow \mathit{Skip} \fatsemi s.s2 \rightarrow \mathit{Skip}$. Note that for one iteration, parallel multiple instance elements are modelled in the same way.

  ```
  sact :: [Seqflow] -> Process -> ([Local],Process)
  sact o p = ([],SeqComp p (seqext o))
  ```

- Given the multiple instance specifies n iterations where $n > 1$ and the flow type `One`, `fixseq` returns a sequential composition of processes, such that it first performs one instance of the activity's work, then triggers its outgoing sequence flow, and then performs its remaining $n-1$ work instances. For example, we consider the sequential multiple instance task shown in Figure 5.5(a). If it specifies three iterations and flow type `One`, `fixseq` returns the following CSP process.

  $$w.A \rightarrow \mathit{Skip} \fatsemi s.s2 \rightarrow \mathit{Skip} \fatsemi w.A \rightarrow \mathit{Skip} \fatsemi w.A \rightarrow \mathit{Skip}$$

- Given the multiple instance specifies n iterations where $n > 1$ and the flow type `All`, `fixseq` returns a sequential composition of processes, such that it first performs n instances of the activity's work, and then triggers its outgoing sequence flow. For example, if the sequential multiple instance task shown in Figure 5.5(a) specifies three iterations and flow type `All`, `fixseq` returns the following CSP process.

  $$w.A \rightarrow \mathit{Skip} \fatsemi w.A \rightarrow \mathit{Skip} \fatsemi w.A \rightarrow \mathit{Skip} \fatsemi s.s2 \rightarrow \mathit{Skip}$$

5.3.5.2 Nondeterministic Number of Sequential Instances

A multiple instance element in this category executes its work instance n times sequentially, where n is nondeterministically chosen from the range $\{1 \mathinner{..} i\}$ for some non-zero finite positive number i specified by the value `Ndet i` recorded in the element's type. We define function `ndetseq` to model the semantics of activities in this category. This function has the same function type as `fixseq`, and is defined for the same three combinations.

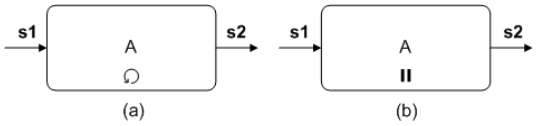


Figure 5.5: (a) sequential and (b) parallel multiple instance task

- For one iteration, `ndetseq` returns the same process as that of `fixseq`.
- Given the multiple instance element specifies at most i iterations where $i > 1$ and flow type `One`, `ndetseq` returns a process that has the following form,

  $$(P \fatsemi Q \fatsemi ((((\fatsemi\ n : \langle 1..i{-}1\rangle \bullet P \sqcap a \to \mathit{Skip}) \fatsemi a \to \mathit{Skip}) \triangle b \to \mathit{Skip}) [\![\{a, b\}]\!] a \to b \to \mathit{Skip})) \setminus \{a, b\}$$

  where: P performs an instance of the activity's work; Q performs the activity's outgoing sequence flow; and events a and b are hidden from the environment. Specifically, this process first performs one instance of the activity's work, then triggers its outgoing sequence flows, and then performs n iterations of the activity's work, where $0 \leq n < i$. For example, if the sequential multiple instance task in Figure 5.5(a) specifies at most three iterations of work instances and flow type `One`, function `ndetseq` returns the following CSP process,

  $$((w.A \to \mathit{Skip} \fatsemi s.s2 \to \mathit{Skip}) \fatsemi ((N \triangle b \to \mathit{Skip}) [\![\{a, b\}]\!] a \to b \to \mathit{Skip})) \setminus \{a, b\}$$

  where $N = (w.A \to \mathit{Skip} \sqcap a \to \mathit{Skip}) \fatsemi (w.A \to \mathit{Skip} \sqcap a \to \mathit{Skip}) \fatsemi a \to \mathit{Skip}$.
- Given the multiple instance element specifies at most i iterations where $i > 1$ and flow type `All`, `ndetseq` returns a process that has following form,

  $$((((P \fatsemi (\fatsemi\ n : \langle 1 \,..\, i-1\rangle \bullet P \sqcap a \to \mathit{Skip})) \triangle b \to \mathit{Skip}) [\![\{a, b\}]\!] a \to b \to \mathit{Skip}) \fatsemi Q) \setminus \{a, b\}$$

  where: P performs an instance of the activity's work; Q performs the activity's outgoing sequence flow; and events a and b are hidden from the environment. Specifically, the process first performs n instances of the activity's work, where $1 \leq n \leq i$, and then triggers its outgoing sequence flow. For example, if the sequential multiple instance task in Figure 5.5(a) specifies at most three iterations of work instances and flow type `All`, function `ndetseq` returns the following CSP process,

  $$((((w.A \to \mathit{Skip} \fatsemi N) \triangle b \to \mathit{Skip}) [\![\{a, b\}]\!] a \to b \to \mathit{Skip}) \fatsemi s.s2 \to \mathit{Skip}) \setminus \{a, b\}$$

  where $N = (w.A \to \mathit{Skip} \sqcap a \to \mathit{Skip}) \fatsemi (w.A \to \mathit{Skip} \sqcap a \to \mathit{Skip}) \fatsemi a \to \mathit{Skip}$.

5.3.5.3 Fixed Number of Parallel Instances

A multiple instance element in this category interleaves i copies of its work instance, where i is a finite non-zero positive number specified by `Fix i` recorded in the element's type. We define the function `fixpar` to model the semantics of activities in this category. This function has to the same function type as `fixseq`, and is defined for the same three combinations.

- For one iteration, `fixpar` returns the same process as that of `fixseq`.
- Given the multiple instance specifies i iterations where $i > 1$ and flow type `One`, `fixpar` returns a process of the following form,

  $$((\,\|\|[\{a\}]\, n : \{1 \,..\, i\} \bullet (P \fatsemi ((Q \fatsemi a \to \mathit{Skip}) \Box a \to \mathit{Skip}))) [\![\alpha Q \cup \{a\}]\!] (Q \fatsemi a \to \mathit{Skip})) \setminus \{a\}$$

  where: P performs an instances of the activity's work; Q performs the activity's outgoing sequence flows; and event a is hidden from the environment. Specifically, this process interleaves i instances of the activity's work, and after one instance is performed, the process may trigger the activity's outgoing sequence flow. Note that after the completion of one instance, the performance of outgoing

sequence flows cannot be refused while other instances may continue to progress. For example, we consider the parallel multiple instance task shown in Figure 5.5(b). Suppose its fixed number of iterations is three and its flow type is `One`; its behaviour is modelled by the following CSP process,

$$((N \;[\![\, \{a\} \,]\!]\; N \;[\![\, \{a\} \,]\!]\; N) \;[\![\, \{s.s2, a\} \,]\!]\; (s.s2 \rightarrow \mathit{Skip} \fatsemi a \rightarrow \mathit{Skip})) \setminus \{a\}$$

where $N = w.A \rightarrow \mathit{Skip} \fatsemi (s.s2 \rightarrow \mathit{Skip} \fatsemi a \rightarrow \mathit{Skip} \;\Box\; a \rightarrow \mathit{Skip})$.

- Given the multiple instance specifies i iterations where $i > 1$ and flow type `All`, `fixpar` returns a process of the form $(|||\, n : \{1..i\} P) \fatsemi Q$, where P performs an instances of the activity's work and Q triggers the activity's outgoing sequence flow. Specifically, this process first interleaves i instances of the activity's work and then performs the activity's outgoing sequence flow. For example, the parallel multiple instance element in Figure 5.5(b) specifies three iterations and the flow type `All`, its behaviour is modelled by the following CSP process.

$$(w.A \rightarrow \mathit{Skip} \;|||\; w.A \rightarrow \mathit{Skip} \;|||\; w.A \rightarrow \mathit{Skip}) \fatsemi s.s2 \rightarrow \mathit{Skip}$$

5.3.5.4 Nondeterministic Number of Parallel Instances

A multiple instance element in this category interleaves n copies of its work instance, where n is nondeterministically chosen from the range $\{1 \,..\, i\}$ from some finite non-zero positive number i specified by the value `Ndet i` recorded in the element's type. We define function `ndetpar` to model the semantics of activities in this category. This function has the same function type as `fixseq`, and is defined for the same three combinations.

- For one iteration, `ndetpar` returns the same process as that of `fixseq`.
- Given the multiple instance specifies i iterations, where $i > 1$, and flow type `One`, `ndetpar` returns a process of the form $(M \;[\![\, \alpha Q \cup \{a, b\} \,]\!]\; N) \setminus \{a, b\}$, where M and N are defined as follows.

$$M = |||\, n : \{1 \,..\, i\} \bullet ((P \fatsemi ((Q \fatsemi a \rightarrow \mathit{Skip}) \;\Box\; a \rightarrow \mathit{Skip})) \;\Box\; b \rightarrow \mathit{Skip})$$
$$N = Q \fatsemi (|||\, n : \{1 \,..\, i\} \bullet (a \rightarrow \mathit{Skip} \;\Box\; b \rightarrow \mathit{Skip}))$$

  Here: P performs an instances of the activity's work; Q performs the activity's outgoing sequence flow; and events a and b are hidden from the environment. Specifically, this process interleaves n work instances, where $1 \leq n \leq i$, and after one instance is performed, the performance of outgoing sequence flows cannot be refused, while the remaining $n-1$ instances may progress independently. For example, we consider the parallel multiple instance task shown in Figure 5.5(b). If it specifies a maximum of three iterations and flow type `One`, its behaviour is modelled by the following CSP process,

$$((G \;|||\; G \;|||\; G) \;[\![\, \{s.s2, a, b\} \,]\!]\; (s.s2 \rightarrow (|||\, n : \{1 \,..\, 3\} \bullet (b \rightarrow \mathit{Skip} \;\Box\; a \rightarrow \mathit{Skip})))) \setminus \{a, b\}$$

  where $G = (w.A \rightarrow \mathit{Skip} \fatsemi (s.s2 \rightarrow \mathit{Skip} \fatsemi a \rightarrow \mathit{Skip} \;\Box\; a \rightarrow \mathit{Skip})) \;\Box\; b \rightarrow \mathit{Skip}$.

- Given the multiple instance specifies i iterations, where $i > 1$, and flow type `All`, `ndetpar` returns a process of the form $((M \;[\![\, \{a, b\} \,]\!]\; N) \fatsemi Q) \setminus \{a, b\}$, where M and N are defined as follows.

$$M = |||\, n : \{1 \,..\, i\} \bullet ((P \fatsemi a \rightarrow \mathit{Skip}) \;\Box\; b \rightarrow \mathit{Skip})$$
$$N = a \rightarrow (|||\, n : \{1 \,..\, i-1\} \bullet (a \rightarrow \mathit{Skip} \;\Box\; b \rightarrow \mathit{Skip}))$$

  Here: P performs an instances of the activity's work; Q performs the activity's outgoing sequence flow; and events a and b are hidden from the environment. Specifically, this process first interleaves n work instances, where $1 \leq n \leq i$, and then performs the activity's outgoing sequence flow. For example, we consider the parallel multiple instance task element shown in Figure 5.5(b). Suppose it specifies a maximum of three iterations and flow type `All`, its behaviour is modelled by the following CSP process,

$$(((G \;|||\; G \;|||\; G) \;[\![\, \{a, b\} \,]\!]\; (a \rightarrow (|||\, i : \{1, 2\} \bullet (b \rightarrow \mathit{Skip} \;\Box\; a \rightarrow \mathit{Skip})))) \fatsemi s.s2 \rightarrow \mathit{Skip}) \setminus \{a, b\}$$

  where $G = (w.A \rightarrow \mathit{Skip} \fatsemi a \rightarrow \mathit{Skip}) \;\Box\; b \rightarrow \mathit{Skip}$.

5.3.6 Exception Flows

Attaching an intermediate event to an activity adds an exception flow to that element. We define the function `encap` to model the behaviour of exception flows. This function takes the following three arguments:

1. An `Atom` value recording the activity's type, sequence, exception and message flows.
2. A list of `Event` values, recording the activity's alphabet.
3. The process definition modelling the behaviour the activity's work and outgoing sequence flows.

The function then returns a process definition that models the activity work, outgoing sequence flows and exception flows.

```
encap :: Atom -> [Event] -> ([Local],Process) -> ([Local],Process)
```

Specifically, this function is defined for the following categories of activity elements:

1. An activity with no exception flow.
2. A task with exception flows.
3. A subprocess with exception flows.

For an activity with no exception flow, the function simply returns the third input argument. We relegate the consideration of subprocesses with exception flows to Section 5.4.4 after we have considered the semantics of subprocesses.

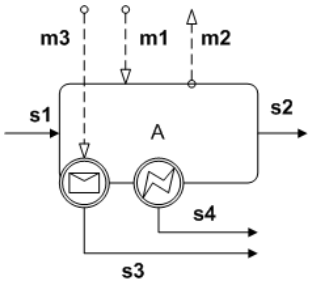


Figure 5.6: A task element with exception flows

For a task with exception flows, the function `encap` returns the process of the form $(W \triangle C)[\![\alpha M]\!]M$, where W is the CSP process defined in the second component of the third argument; this process models the behaviour of the task's work and outgoing sequence flows. This is a task's normal flow [OMG08, Section 10.2.1]. For example consider the task in Figure 5.6; process W for this task, which is shown below, is the sequential composition of the task's work, outgoing message flow and outgoing sequence flow.

$$W = w.A \rightarrow \mathit{Skip} \fatsemi m.m2 \rightarrow \mathit{Skip} \fatsemi s.s2 \rightarrow \mathit{Skip} \tag{5.2}$$

At any time during the normal flow, an interrupt might occur, which halts the normal flow and triggers one of the task's exception flows. This is modelled by the process $W \triangle C$, where process C models the task's possible exception flows, that is, an external choice of processes, each modelling an intermediate event attached to the task. For example consider the task in Figure 5.6; process C for this task, which is shown below, is the external choice of processes modelling the message and error events attached to the task.

$$C = m.m3 \rightarrow s.s3 \rightarrow \mathit{Skip} \Box s.s4 \rightarrow \mathit{Skip} \tag{5.3}$$

The process $W \triangle C$ is partially interleaved with process M synchronising on αM. M initially offers either the event that models the task's work or one of the task's outgoing sequence flow and exception flows. After performing the work done event, the process behaves as M again, while after performing one

of task's outgoing sequence flow and exception flows the process terminates. This composition ensures that the task cannot trigger both its outgoing sequence flows and exception flows. For example consider the task in Figure 5.6; process M for this task, which is shown below, is the external choice of the task's outgoing sequence flow and exception flows.

$$M = w.A \rightarrow M \;\Box\; m.m3 \rightarrow s.s3 \rightarrow \mathit{Skip} \;\Box\; s.s4 \rightarrow \mathit{Skip} \;\Box\; s.s2 \rightarrow \mathit{Skip} \tag{5.4}$$

5.4 Subprocesses

A subprocess is a compound activity [OMG08, Section 9.4.2]. The work of a subprocess is defined in terms of BPMN elements it directly contains. An example of a subprocess is shown in Figure 5.7. We use this subprocess as a running example throughout this section. This section is structured as follows. In Section 5.4.1 we describe and motivate our approach to modelling interaction between elements contained in a subprocess; in Sections 5.4.2 and 5.4.3 we describe extensions to our existing semantics for precisely capturing the event-based gateway's behaviour, looping and completion; in Section 5.4.4 we consider the subprocess's exception flows behaviour; and in Section 5.4.5 we give an overview of the Haskell implementation of the subprocess' semantics.

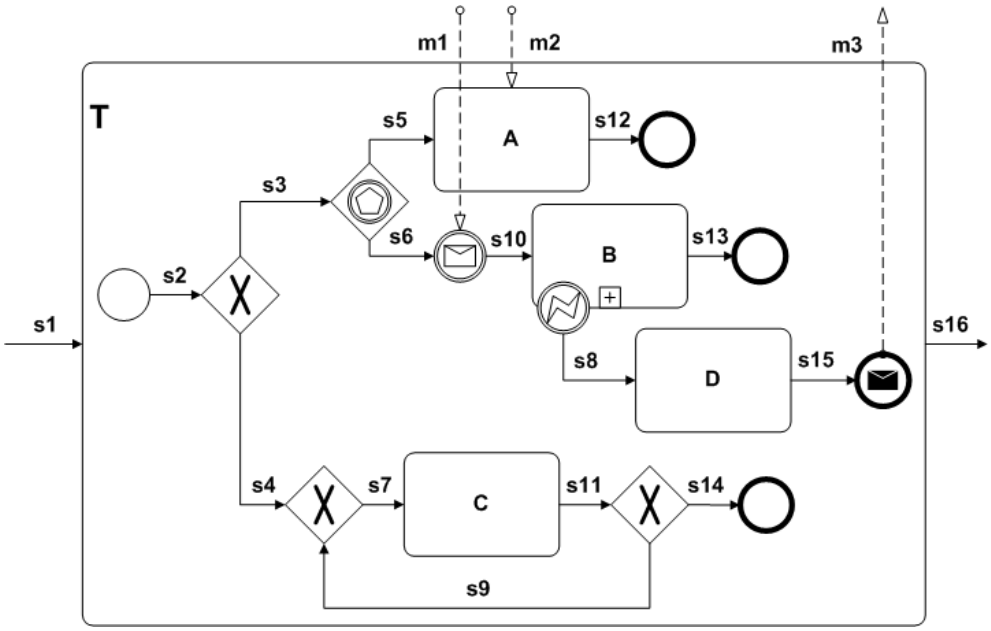


Figure 5.7: A subprocess

5.4.1 Containment

A BPMN process is defined in terms of the behaviour of elements it directly contains and their control flows. Two elements contained in a BPMN process establish a flow of control by sharing a sequence flow. That is to say one element's outgoing sequence flow or exception flow is the other element's incoming sequence flow. Semantically this is represented by the parallel composition of the two processes that model the elements' behaviour, each synchronising on its own alphabet. This method of constructing the semantics via parallel operators is a natural representation in process algebras [CPT01]. For example the semantics of the subprocess shown in Figure 5.7 can be described by the following CSP process, $s.s1 \rightarrow \mathit{Skip} \;⨟\; T \;⨟\; s.s16 \rightarrow \mathit{Skip}$, where process T is defined by Equation 5.5. Here we identify each end event by its incoming sequence flow. The full definition of CSP process $P(B)$ is given by Equation 5.14. Note that the process definitions provided in Equation 5.5 apply the extension considered in Sections 5.4.2, 5.4.3 and 5.4.4.

$$
\begin{array}{rcl}
P(s1) & = & s.s2 \rightarrow \mathit{Skip} \fatsemi E \\
P(g1) & = & (s.s2 \rightarrow \mathit{Skip} \fatsemi (s.s3 \rightarrow \mathit{Skip} \sqcap s.s3 \rightarrow \mathit{Skip}) \fatsemi P(g1)) \Box E \\
P(g2) & = & (s.s3 \rightarrow \mathit{Skip} \fatsemi (m.m2 \rightarrow s.s5 \rightarrow \mathit{Skip} \Box m.m1 \rightarrow s.s6 \rightarrow \mathit{Skip}) \fatsemi P(g2)) \Box E \\
P(g3) & = & ((s.s4 \rightarrow \mathit{Skip} \Box s.s9 \rightarrow \mathit{Skip}) \fatsemi s.s7 \rightarrow \mathit{Skip} \fatsemi P(g3)) \Box E \\
P(g4) & = & (s.s11 \rightarrow \mathit{Skip} \fatsemi (s.s14 \rightarrow \mathit{Skip} \sqcap s.s9 \rightarrow \mathit{Skip}) \fatsemi P(g4)) \Box E \\
P(A) & = & ((s.s5 \rightarrow \mathit{Skip} \,|||\, m.m2 \rightarrow \mathit{Skip}) \fatsemi w.A \rightarrow \mathit{Skip} \fatsemi s.s12 \rightarrow \mathit{Skip} \fatsemi P(A)) \Box E \\
P(B) & = & \textbf{let } BC = (\Box\, x : (\alpha BP \setminus \{s.s13, e.s19, s.s8\}) \bullet x \rightarrow BC) \Box \\
 & & \qquad s.s13 \rightarrow \mathit{Skip} \Box e.s19 \rightarrow s.s8 \rightarrow \mathit{Skip} \\
 & & \quad BP = (\, \|\, i : \{\mathit{start}, S, \mathit{gate}, s19, s20\} \bullet \alpha P(i) \circ P(i)) \fatsemi s.s13 \rightarrow \mathit{Skip} \\
 & & \textbf{in } (s.s10 \rightarrow \mathit{Skip} \fatsemi ((BP \triangle s.s8 \rightarrow \mathit{Skip}) \,[\![\, \alpha BP \,]\!]\, BC) \fatsemi P(B)) \Box E \\
P(C) & = & (s.s7 \rightarrow \mathit{Skip} \fatsemi w.C \rightarrow \mathit{Skip} \fatsemi s.s11 \rightarrow \mathit{Skip} \fatsemi P(C)) \Box E \\
P(D) & = & (s.s8 \rightarrow \mathit{Skip} \fatsemi w.D \rightarrow \mathit{Skip} \fatsemi s.s15 \rightarrow \mathit{Skip} \fatsemi P(D)) \Box E \\
P(i1) & = & ((s.s6 \rightarrow \mathit{Skip} \,|||\, m.m1 \rightarrow \mathit{Skip}) \fatsemi s.s10 \rightarrow \mathit{Skip} \fatsemi P(i1)) \Box E \\
P(s12) & = & (s.s12 \rightarrow \mathit{Skip} \fatsemi c.s12 \rightarrow \mathit{Skip}) \Box (\Box\, i : \{c.s13, c.s14, c.s15\} \bullet i \rightarrow \mathit{Skip}) \\
P(s13) & = & (s.s13 \rightarrow \mathit{Skip} \fatsemi c.s13 \rightarrow \mathit{Skip}) \Box (\Box\, i : \{c.s12, c.s14, c.s15\} \bullet i \rightarrow \mathit{Skip}) \\
P(s14) & = & (s.s14 \rightarrow \mathit{Skip} \fatsemi c.s14 \rightarrow \mathit{Skip}) \Box (\Box\, i : \{c.s12, c.s13, c.s15\} \bullet i \rightarrow \mathit{Skip}) \\
P(s15) & = & (s.s15 \rightarrow \mathit{Skip} \fatsemi m.m3 \rightarrow \mathit{Skip} \fatsemi c.s15 \rightarrow \mathit{Skip}) \Box (\Box\, i : \{c.s12, c.s13, c.s14\} \bullet i \rightarrow \mathit{Skip}) \\
E & = & \Box\, i : \{c.s12, c.s13, c.s14, c.s15\} \bullet i \rightarrow \mathit{Skip} \\
T & = & \|\, i : \{A, B, C, D, g1, g2, g3, g4, s1, i1, s12, s13, s14, s15\} \bullet \alpha P(i) \circ P(i)
\end{array} \tag{5.5}
$$

5.4.2 Event-based Gateways

In Section 5.3.3, message events are modelled with interleaving incoming sequence and message flows. However, this has an effect when considering the control flows between elements in a BPMN process. For example, the subprocess in Figure 5.7 contains an event-based exclusive gateway with incoming sequence flow $s3$ and outgoing sequence flows $s5$ and $s6$. Upon triggering $s3$, the gateway triggers $s5$ if task A receives a message first, and $s6$ if the intermediate message event receives a message first.

However, this behaviour cannot be captured by the current semantics of message events. Consider the following processes $P(\mathit{exgate})$ and $P(\mathit{msg})$ that model the behaviour of the gateway and the message event.

$$
\begin{array}{rcl}
P(\mathit{exgate}) & = & s.s3 \rightarrow \mathit{Skip} \fatsemi (s.s5 \rightarrow \mathit{Skip} \Box s.s6 \rightarrow \mathit{Skip}) \\
P(\mathit{msg}) & = & (m.m1 \rightarrow \mathit{Skip} \,|||\, s.s6 \rightarrow \mathit{Skip}) \fatsemi s.s10 \rightarrow \mathit{Skip}
\end{array} \tag{5.6}
$$

Specifically, the message event (msg) does not insist on the order of incoming sequence and message flows, and the gateway (exgate) does not insist that the message flow $m1$ must be triggered before sequence flow $s6$. As a result we cannot ensure $s6$ gets triggered after $m1$ is triggered. we therefore extend the semantics of event-based XOR gateways by prefixing each of its outgoing sequence flows with the corresponding incoming message flows of the gateway's direct succeeding elements. The following process illustrates this proposed revision to the process semantics in Definition 5.6.

$$
P(\mathit{exgate}) = s.s3 \rightarrow \mathit{Skip} \fatsemi (m.m2 \rightarrow s.s5 \rightarrow \mathit{Skip} \Box m.m1 \rightarrow s.s6 \rightarrow \mathit{Skip}) \tag{5.7}
$$

5.4.3 Looping and Completion

BPMN allows sequence flow looping [OMG08, Section 10.2.1.7]. For example in Figure 5.7, task C may be repeatedly performed by sequence flow looping. However, we have so far modelled a BPMN element's behaviour as a terminating process. To capture sequence flow looping, we introduce recursion into our model. For example, task C in Figure 5.7 is modelled by the following process.

$$
P(c) = s.s7 \rightarrow \mathit{Skip} \fatsemi w.C \rightarrow \mathit{Skip} \fatsemi s.s11 \rightarrow \mathit{Skip} \fatsemi P(c) \tag{5.8}
$$

Note that this extension is not applied to start and end events, this is because sequence flow looping can only be defined on elements that have both incoming and outgoing sequence flows.

While the recursive process models sequence flow looping, a BPMN process terminates upon a successful completion. Specifically, a BPMN process signifies its completion when one of the end events it directly contains is triggered [OMG08, Table 9.6]. For example, we consider the following normal flow of the subprocess shown in Figure 5.7: upon triggering the start event in the subprocess shown in Figure 5.7, the subprocess internally chooses to trigger flow $s4$, and performs task C. Afterwards, the subprocess triggers the task's succeeding gateway, then chooses to trigger flow $s14$, and arrives at the non-trigger end event. At this point the subprocess completes its normal flow, hence it reaches a completion and terminates.

Naturally we model a BPMN process's termination as the CSP process *Skip*, and a parallel process may behave as *Skip* if and only if each of its composed processes can also behave as *Skip*. As a result we provide some extensions to our model. In the following definitions, set E indexes all end events directly contained that BPMN process and for all $e \in E$, we let CSP event $c.e$ to denote the completion of the end event e.

- Given a recursive CSP process of the form $P = T \fatsemi P$ that models an activity or a gateway in a BPMN process, we extend P as follows.

$$P = (T \fatsemi P) \Box (\Box\, e : E \bullet c.e \rightarrow \mathit{Skip}) \tag{5.9}$$

  With this extension, the process initially offers the choice to either behave as the intended BPMN element or cooperate with an end event to terminate, and repeats the same choice after each time behaving as that element.

  For example, the following process $P(C)$ extends the process in Equation 5.8 and shows how we model the completion of task C in Figure 5.7.

$$P(C) = ((s.s7 \rightarrow \mathit{Skip} \fatsemi (w.C \rightarrow \mathit{Skip} \fatsemi (s.s11 \rightarrow \mathit{Skip}))) \fatsemi P(c)) \Box (\Box\, e : E \bullet c.e \rightarrow \mathit{Skip}) \tag{5.10}$$

- Given a non-recursive CSP process of the form $P = S$ that models a start event in a BPMN process, we extend P as follows.

$$P = (S \fatsemi (\Box\, e : E \bullet c.e \rightarrow \mathit{Skip})) \Box (\Box\, e : E \bullet c.e \rightarrow \mathit{Skip}) \tag{5.11}$$

  With this extension, the process initially offers the choice to either behave as the start event or cooperate with an end event to terminate. After behaving as the start event, it waits until an end event is triggered and cooperates with it to terminate.

- Given a non-recursive CSP process of the form $P = S$ that models some end event f in a BPMN process, we extend P as follows,

$$P = (S \fatsemi c.f \rightarrow \mathit{Skip}) \Box (\Box\, e : E \setminus \{f\} \bullet c.e \rightarrow \mathit{Skip}) \tag{5.12}$$

  With this extension, the process initially offers the choice to either behave as the end event or cooperate with one of the other end events contained in the same BPMN process to terminate. After behaving as the end event, it may signal completion.

  For example, the following process $P(e1)$ defines the behaviour of the non-trigger end event in Figure 5.7 that has incoming sequence flow $s14$. Here $e1$ identifies the end event.

$$P(e1) = (s.s14 \rightarrow \mathit{Skip} \fatsemi c.e1 \rightarrow \mathit{Skip}) \Box (\Box\, e : E \setminus \{e1\} \bullet c.e \rightarrow \mathit{Skip}) \tag{5.13}$$

5.4.4 Exception Flows

In this section we consider the behaviour of a subprocess's exception flows. For example, Figure 5.8 expands the collapsed subprocess B from Figure 5.7. Upon triggering flow $s10$, subprocess B triggers the start event it contains. The flow leads to task S. After performing S, the subprocess either completes

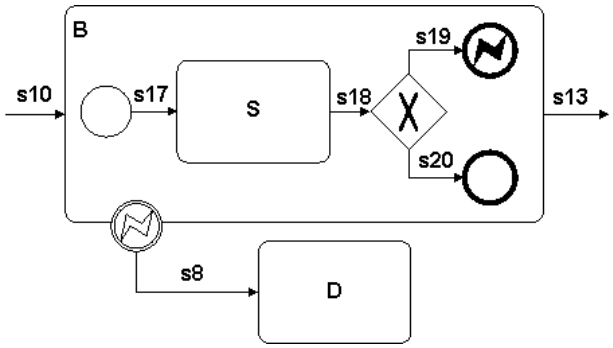


Figure 5.8: A BPMN subprocess with exception flows

successfully or arrives at the end error event via flow $s19$, which throws an error. This error is then caught by the intermediate event attached to B, thereby triggering exception flow $s8$.

Specifically, an end error event records an error code, representing an occurrence of a specific error. An error thrown from a subprocess is caught by an intermediate error event that is attached to the subprocess and has the same error code.

Recall that in Section 5.3.6, we introduced function `encap` and described how it models a task's exception flows. We now describe how it models a subprocess' exception flows. Given some subprocess S with exception flows, the function `encap` returns the process of the form $(P \triangle (E \,\Box\, F'))[\![\alpha P \cup \alpha E \cup \alpha F]\!] C$, where C is defined as follows.

$$C = (\Box\, x : (\alpha P \setminus (\alpha O \cup \alpha E \cup \alpha F)) \bullet x \rightarrow C) \,\Box\, O \,\Box\, E \,\Box\, F$$

Here P is the CSP process that models the interaction between elements directly contained in S and S's outgoing sequence flows; O models S's outgoing sequence flows, and E, F and F' model S's exception flows in the following way:

- E defines the external choice over the intermediate events attached to S such that each of the events is either a timer event, a message event, a rule event or an error event that is not associated with a specific error.
- F defines the external choice over the intermediate events attached to S such that each of the events is an error event that is associated with a specific error.
- F' defines the external choice over the outgoing sequence flow of intermediate events attached to S such that each of the events is an error event that is associated with a specific error.

Specifically, `encap` returns a process that behaves as P, the normal flow of the subprocess, and at any point the normal flow may be interrupted due to one of the following behaviours:

1. A message arrives at an intermediate message event attached to the subprocess.
2. The duration specified by an intermediate timer event attached to the subprocess elapses.
3. The condition specified by an intermediate rule event attached to the subprocess becomes true.
4. An unspecified error occurs, which is then caught by an intermediate error event attached to the subprocess.
5. A specific error thrown by an end error event directly contained in the subprocess. This is then caught by the corresponding intermediate error event attached to that subprocess.

The exception flows of the first four types of behaviour are modelled by process E. Note that our model abstracts from both time and process data, as a result, exceptions due to the second and third types of behaviour are modelled as unspecified errors. The exception flows of the fifth type of behaviour are

modelled by process F'. The process $P \triangle (E \Box F')$ is partially interleaved with C, synchronising on the alphabet of $P \triangle (E \Box F')$. The composed process ensures that if an outgoing sequence flow is triggered then no exception flows may be triggered, and vice versa.

For example, consider subprocess B in Figure 5.8; consider also the following CSP processes model B and the elements it contains. Here we identify each end event by its incoming sequence flow.

$$
\begin{aligned}
P(start) &= s.s17 \rightarrow Skip \fatsemi (c.s19 \rightarrow Skip \Box c.s20 \rightarrow Skip) \\
P(S) &= (s.s17 \rightarrow Skip \fatsemi w.S \rightarrow Skip \fatsemi s.s18 \rightarrow Skip \fatsemi P(S)) \Box (c.s19 \rightarrow Skip \Box c.s20 \rightarrow Skip) \\
P(gate) &= (s.s18 \rightarrow Skip \fatsemi (s.s19 \rightarrow Skip \sqcap s.s20 \rightarrow Skip) \fatsemi P(gate)) \Box \\
&\quad (c.s19 \rightarrow Skip \Box c.s20 \rightarrow Skip) \\
P(s19) &= (s.s19 \rightarrow Skip \fatsemi e.s19 \rightarrow Skip \fatsemi c.s19 \rightarrow Skip) \Box c.s20 \rightarrow Skip \\
P(s20) &= (s.s20 \rightarrow Skip \fatsemi c.s20 \rightarrow Skip) \Box c.s19 \rightarrow Skip \\
P(B) &= \textbf{let } BC = (\Box\, x : (\alpha BP \setminus \{s.s13, e.s19, s.s8\})) \bullet x \rightarrow BC) \Box s.s13 \rightarrow Skip \Box e.s19 \rightarrow s.s8 \rightarrow Skip \\
&\qquad BP = (\, \|\, i : \{start, S, gate, s19, s20\} \bullet \alpha P(i) \circ P(i)) \fatsemi s.s13 \rightarrow Skip \\
&\quad \textbf{in } (s.s10 \rightarrow Skip \fatsemi ((BP \triangle s.s8 \rightarrow Skip) \,[\![\, \alpha BP \,]\!]\, BC) \fatsemi P(B)) \Box \\
&\qquad (\Box\, i : \{s12, s13, s14, s15\} \bullet c.i \rightarrow Skip)
\end{aligned} \tag{5.14}
$$

5.4.5 Implementation

We provide function `embed` for modelling the work behaviour of a subprocess.

```
embed :: String -> [Element] -> [ProcessDef]
embed nm es = comps1 ++ [(spterm nm, [LS ("I",index)] ,proc)]
  where index = List Set (map name es)
        proc = Indparcomp ("i",SName "I") (SName (aterm "i")) (ProcId (pterm "i"))
        comps = map ((par id element) . split) es
        comps1 = concat [ [(complete es (atom e) p)]++ps | (e,(p:ps)) <- comps ]
```

This function takes the subprocess's identifier of type `String` and a list of elements, of type `[Element]`, directly contained in that subprocess, and returns a list of process definitions, of type `[ProcessDef]`, that models the subprocess's behaviour. This list of process definitions consists of processes that model individual elements contained in the subprocess as well as a process that defines their parallel combination. Specifically, `embed` takes each BPMN element directly contained in the subprocess and first applies the function `element`, which returns a CSP process that models a single instance of element's behaviour. `embed` then applies the function `complete` to extend this process for sequence flow looping and process completion as defined in Section 5.4.3.

5.5 Pools and Diagrams

In this section we consider the behaviour of BPMN pools and diagrams. We consider the semantics of BPMN pools in Section 5.5.1 and the semantics of BPMN diagrams in Section 5.5.2.

5.5.1 Pools

Similar to a subprocess, the semantics of a BPMN pool is defined in terms of the behaviour of the elements.

For example, Figure 5.9 shows BPMN pool P. P initially triggers subprocess M and task N. P completes execution when N completes successfully and that M either completes successfully or throws an error. The following CSP process $PL(P)$ models this behaviour,

$$
PL(P) = \, \|\, i : \{sP, s1, s2, s3, s4, s6, s8\} \bullet \alpha P(i) \circ P(i) \tag{5.15}
$$

where sP identifies the start event directly contained in P, and all other elements directly contained in P are identified by their incoming sequence flows.

We provide function `pool` for modelling BPMN pool's behaviour.

```
pool :: (PoolId,[Element]) -> Script
```

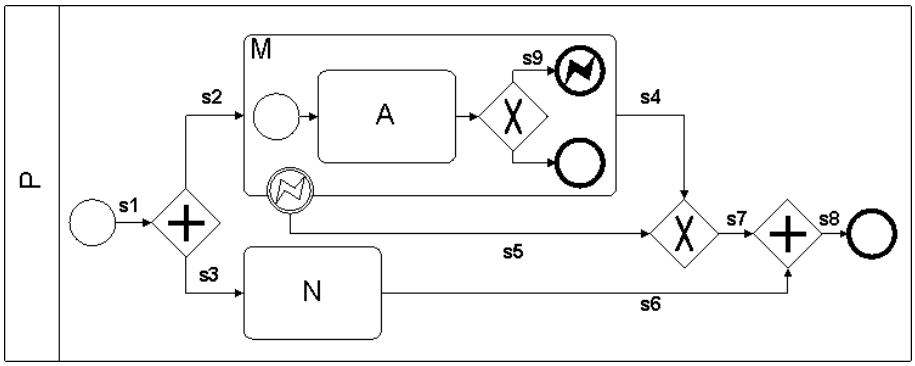


Figure 5.9: A BPMN pool

This function takes the pool's name and the list of elements the pool contains, and returns a `Script` value, recording the datatype, process and set definitions in CSPm, which together define the process semantics of the BPMN pool.

5.5.2 Diagrams

A BPMN diagram describes the interactions between business processes, each represented by a BPMN pool.

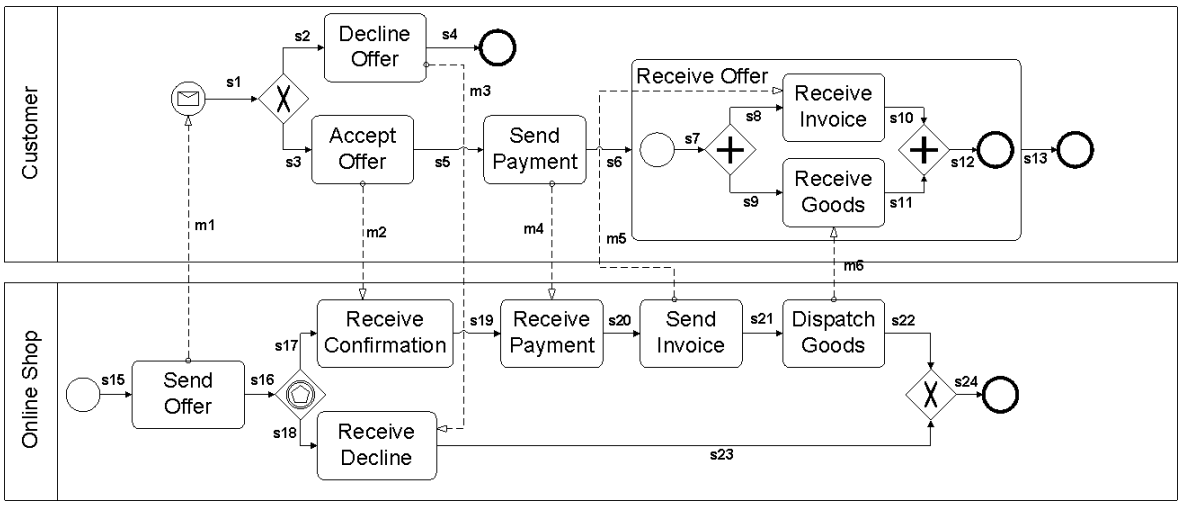


Figure 5.10: An online shop business process

For example, we consider our online shop running example of Figure 2.3. Figure 5.10 replicates the same BPMN diagram and labels its sequence and message flows. In this example, the customer and online shop BPMN pools communicate to each other via message flows $m1$, $m2$, $m3$, $m4$, $m5$ and $m6$. We define CSP process CP in Equation 5.16 to model the behaviour of the *Customer* BPMN pool. The start element is identified by sCP while all other BPMN elements in the pool are identified by their

incoming sequence flows.

$$
\begin{aligned}
EP &= c.s13 \rightarrow Skip \;\Box\; c.s4 \rightarrow Skip \\
P(sCP) &= (m.m1 \rightarrow Skip \fatsemi s.s1 \rightarrow Skip \fatsemi EP) \;\Box\; EP \\
P(s1) &= (s.s1 \rightarrow Skip \fatsemi (s.s2 \rightarrow Skip \sqcap s.s3 \rightarrow Skip) \fatsemi P(s1)) \;\Box\; EP \\
P(s2) &= (s.s2 \rightarrow Skip \fatsemi w.DO \rightarrow Skip \fatsemi m.m3 \rightarrow Skip \fatsemi s.s4 \rightarrow Skip \fatsemi P(s2)) \;\Box\; EP \\
P(s3) &= (s.s3 \rightarrow Skip \fatsemi w.AO \rightarrow Skip \fatsemi m.m2 \rightarrow Skip \fatsemi s.s5 \rightarrow Skip \fatsemi P(s3)) \;\Box\; EP \\
P(s4) &= (s.s4 \rightarrow Skip \fatsemi c.s4 \rightarrow Skip) \;\Box\; c.s13 \rightarrow Skip \\
P(s5) &= (s.s5 \rightarrow Skip \fatsemi w.SP \rightarrow Skip \fatsemi m.m4 \rightarrow Skip \fatsemi s.s6 \rightarrow Skip \fatsemi P(s4)) \;\Box\; EP \\
P(s6) &= (s.s6 \rightarrow Skip \fatsemi RO \fatsemi s.s13 \rightarrow Skip \fatsemi P(s6)) \;\Box\; EP \\
P(s13) &= (s.s13 \rightarrow Skip \fatsemi c.13 \rightarrow Skip) \;\Box\; c.s4 \rightarrow Skip \\
CP &= \Vert\, i : \{sCP, s1, s2, s3, s4, s5, s6, s13\} \bullet \alpha P(i) \circ P(i)
\end{aligned} \tag{5.16}
$$

We abbreviate each task name using the first letter of each word in its name. For example, the CSP event $w.DO$ represents the work done of task Decline Offer. The process $P(s6)$ models the behaviour of the Receive Offer subprocess; the definition of RO is defined in Equation 5.17.

$$
\begin{aligned}
P(sRO) &= (s.s7 \rightarrow Skip \fatsemi c.s12 \rightarrow Skip) \;\Box\; c.s12 \rightarrow Skip \\
P(s7) &= (s.s7 \rightarrow Skip \fatsemi (s.s8 \rightarrow Skip \;|||\; s.s9 \rightarrow Skip) \fatsemi P(s7)) \;\Box\; c.s12 \rightarrow Skip \\
P(s8) &= ((s.s8 \rightarrow Skip \;|||\; m.m5 \rightarrow Skip) \fatsemi w.RI \rightarrow Skip \fatsemi s.s4 \rightarrow Skip \fatsemi P(s8)) \;\Box\; c.s12 \rightarrow Skip \\
P(s9) &= ((s.s9 \rightarrow Skip \;|||\; m.m6 \rightarrow Skip) \fatsemi w.RG \rightarrow Skip \fatsemi s.s11 \rightarrow Skip \fatsemi P(s9)) \;\Box\; c.s12 \rightarrow Skip \\
P(s10) &= ((s.s10 \rightarrow Skip \;|||\; s.s11 \rightarrow Skip) \fatsemi s.s12 \rightarrow Skip \fatsemi P(s10)) \;\Box\; c.s12 \rightarrow Skip \\
P(s12) &= s.s12 \rightarrow Skip \fatsemi c.12 \rightarrow Skip \\
RO &= \Vert\, i : \{sRO, s7, s8, s9, s10, 12\} \bullet \alpha P(i) \circ P(i)
\end{aligned} \tag{5.17}
$$

Similarly we define CSP process OS in Equation 5.18 that models the behaviour of the *OnlineShop* BPMN pool. Its start element is identified by sOS while all other BPMN elements in the pool are identified by their incoming sequence flows.

$$
\begin{aligned}
P(sOS) &= (s.s15 \rightarrow Skip \fatsemi c.s24 \rightarrow Skip) \;\Box\; c.s24 \rightarrow Skip \\
P(s15) &= (s.s15 \rightarrow Skip \fatsemi w.SO \rightarrow Skip \fatsemi m.m1 \rightarrow Skip \fatsemi s.s16 \rightarrow Skip \fatsemi P(s15)) \;\Box\; c.s24 \rightarrow Skip \\
P(s16) &= (s.s16 \rightarrow Skip \fatsemi (m.m2 \rightarrow Skip \fatsemi s.s17 \rightarrow Skip \;\Box\; m.m3 \rightarrow Skip \fatsemi s.s18 \rightarrow Skip) \fatsemi P(s16)) \;\Box \\
&\quad\; c.s24 \rightarrow Skip \\
P(s17) &= ((s.s17 \rightarrow Skip \;|||\; m.m2 \rightarrow Skip) \fatsemi w.RC \rightarrow Skip \fatsemi s.s19 \rightarrow Skip \fatsemi P(s17)) \;\Box\; c.s24 \rightarrow Skip \\
P(s18) &= ((s.s18 \rightarrow Skip \;|||\; m.m3 \rightarrow Skip) \fatsemi w.RD \rightarrow Skip \fatsemi s.s23 \rightarrow Skip \fatsemi P(s18)) \;\Box\; c.s24 \rightarrow Skip \\
P(s19) &= ((s.s19 \rightarrow Skip \;|||\; m.m4 \rightarrow Skip) \fatsemi w.RP \rightarrow Skip \fatsemi s.s20 \rightarrow Skip \fatsemi P(s19)) \;\Box\; c.s24 \rightarrow Skip \\
P(s20) &= (s.s20 \rightarrow Skip \fatsemi w.SI \rightarrow Skip \fatsemi m.m5 \rightarrow Skip \fatsemi s.s21 \rightarrow Skip \fatsemi P(s20)) \;\Box\; c.s24 \rightarrow Skip \\
P(s21) &= (s.s21 \rightarrow Skip \fatsemi w.DG \rightarrow Skip \fatsemi m.m6 \rightarrow Skip \fatsemi s.s22 \rightarrow Skip \fatsemi P(s21)) \;\Box\; c.s24 \rightarrow Skip \\
P(s22) &= ((s.s22 \rightarrow Skip \;\Box\; s.s23 \rightarrow Skip) \fatsemi s.s24 \rightarrow Skip \fatsemi P(s13)) \;\Box\; c.s24 \rightarrow Skip \\
P(s24) &= s.s24 \rightarrow Skip \fatsemi c.24 \rightarrow Skip \\
OS &= \Vert\, i : \{sOS, s15, s16, s17, s18, s19, s20, s21, s22, s24\} \bullet \alpha P(i) \circ P(i)
\end{aligned} \tag{5.18}
$$

Equation 5.19 defines CSP process *Example* that models the interaction between the customer and the online shop business processes. Specifically, we model a BPMN diagram as a parallel composition of processes, each corresponds to a BPMN pool's behaviour and synchronises on its own alphabet.

$$
Example = CP \;[\![\, \alpha CP \mid \alpha OS \,]\!]\; OS \tag{5.19}
$$

We hence devise the semantic function `bToc` that was introduced at the beginning of this chapter. This function takes the syntax of the BPMN diagram, of type `Diagram`, and returns a `Script` value,

recording the datatype, process and set definitions in CSPm, that defines the process semantics of the BPMN diagram. In our abstract syntax each BPMN diagram is recorded using the type synonym `Diagram`, which is a list of pairs, where each pair records the name and the syntax of a BPMN pool in the diagram.

5.6 Safety and Liveness

Using CSP's refinement orderings, we formally compare behaviours between BPMN diagrams and are able to verify behavioural properties of BPMN diagrams. In particular behavioural properties may be specified either as CSP processes or as BPMN diagrams. To simplify our notation in this section we write $D_p[\![P]\!]$ to denote the semantics of BPMN process P and $D[\![X]\!]$ to denote the semantics of BPMN diagram X.

5.6.1 Safety

In general a BPMN diagram is a traces refinement of another BPMN diagram precisely when the traces of the former is a subset of the traces of the latter:

Definition 5.1. ***Traces Refinement.*** *BPMN diagram P is a traces refinement of BPMN diagram Q if and only if $D[\![Q]\!] \sqsubseteq_{\mathcal{T}} D[\![P]\!]$.*

Using the traces refinement, we can also assert that a BPMN diagram meets a particular safety property expressed as a CSP process or as another BPMN diagram. We consider the online shop business process running example, whose behaviour is modelled by process *Example* defined in Equation 5.19. One of the business process's behavioural requirements is as follows:

> Goods must not be dispatched after the shop has sent a sale offer until the customer has accepted the offer and then made the required payment.

We model this requirement as the CSP process DP defined in the following Equation 5.20.

$$DP = w.SO \rightarrow w.AO \rightarrow w.SP \rightarrow w.DG \rightarrow Stop \tag{5.20}$$

Here we are interested in this requirement as a safety property, we therefore check the refinement assertion expressed in the following Equation 5.21.

$$DP \sqsubseteq_{\mathcal{T}} Example \fatsemi Stop \setminus (\Sigma \setminus \alpha DP) \tag{5.21}$$

Note that we hide event $\checkmark$ by sequentially composing the process modelling the BPMN diagram with the process *Stop*. This refinement states that the behaviour of $Example \fatsemi Stop \setminus (\Sigma \setminus \alpha DP)$ is a subset of the following set of traces.

$$\{\langle\rangle, \langle w.SO\rangle, \langle w.SO, w.AO\rangle, \langle w.SO, w.AO, w.SP\rangle, \langle w.SO, w.AO, w.SP, w.DG\rangle\}$$

The refinement ensures that bad behaviours such as the online shop dispatching goods before the customer has sent the payment are not possible.

By giving a semantics to BPMN in CSP, we can also express the above safety property using the BPMN diagram *Dispatch* shown in Figure 5.11. As a safety property, we check the refinement assertion

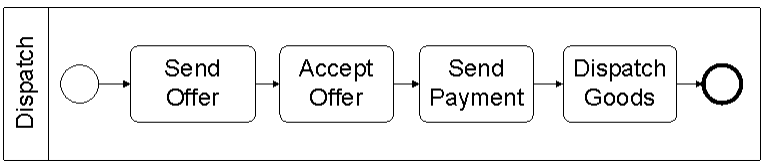


Figure 5.11: A BPMN process modelling Equation 5.20

expressed in Equation 5.22.

$$D[\![Dispatch]\!] \fatsemi Stop \setminus (\Sigma \setminus \{\!|w|\!\}) \sqsubseteq_{\mathcal{T}} Example \fatsemi Stop \setminus (\Sigma \setminus (\alpha D[\![D]\!] \cap \{\!|w|\!\})) \tag{5.22}$$

Again, we hide event ✓ by sequentially composing the process modelling the BPMN diagram with process *Stop*. Moreover, this safety property concerns only tasks in the diagram and we therefore use the CSP hiding operator to conceal events associated with completion, sequence and message flows. Both traces refinement assertions hold.

Using the traces refinement, we can also express more general properties about BPMN diagrams such as *the absence of un-triggered elements* property [DDO08], that is, given a BPMN diagram we specify that there are no element of the diagram can never be triggered. Specifically, an AND join gateway is triggered if all of its incoming sequence flows are performed, while other element is triggered is one of its incoming sequence flows is performed.

We first define the characteristic set *trg* such that for element e contained in some BPMN diagrams d, $(e, d) \in trg$ if e can be triggered in d. Specifically, if e is an AND join gateway, $(e, d) \in trg$ if and only if all incoming sequence flows of e appear in one of $D[\![d]\!]$'s traces regardless of the order in which they appear, and if e is a XOR join gateway, $(e, d) \in trg$ if and only if one of incoming sequence flows of e appear in one of $D[\![d]\!]$'s traces, otherwise $(e, d) \in trg$ if and only if each incoming sequence flow of e appears in one of $D[\![d]\!]$'s traces. Here iseq s is a set of injective finite sequences over set s.

$$\begin{array}{|l} trg : \mathbb{P}(Element \times Diagram) \\ \hline \forall\, e : Element;\ d : Diagram \bullet \\ \quad (atom\ e).type = agate \land \#(atom\ e).in > 1 \Rightarrow \\ \quad\quad (e, d) \in trg \Leftrightarrow (\exists\, ms : \{ts : \text{iseq}((atom\ e).in) \mid \#ts = \#(atom\ e).in\} \bullet \\ \quad\quad\quad D[\![d]\!] \setminus (\Sigma \setminus \text{ran}\, ms) \sqsubseteq_{\mathcal{T}} (\fatsemi\, m : ms \bullet s.m \rightarrow Skip) \fatsemi Stop) \land \\ \quad (atom\ e).type \neq agate \land \#(atom\ e).in > 1 \Rightarrow \\ \quad\quad (e, d) \in trg \Leftrightarrow (\exists\, i : (atom\ e).in \bullet D[\![d]\!] \setminus (\Sigma \setminus \{s.i\}) \sqsubseteq_{\mathcal{T}} s.i \rightarrow Stop) \\ \quad (atom\ e).type \neq agate \lor \#(atom\ e).in \leq 1 \Rightarrow \\ \quad\quad (e, d) \in trg \Leftrightarrow (\forall\, i : (atom\ e).in \bullet D[\![d]\!] \setminus (\Sigma \setminus \{s.i\}) \sqsubseteq_{\mathcal{T}} s.i \rightarrow Stop) \end{array}$$

Since each BPMN diagram contains a finite number of BPMN elements, the characteristic set *trg* casts the absence of un-triggered elements properties into a finite number of traces refinement checks. For example, given a diagram d containing an AND join gateway e with the set of incoming sequence flows $\{s1, s2\}$, then $(e, d) \in trg$ if and only if one of following two traces refinements hold, that is, all of e's incoming sequence flows are performed in one of $D[\![d]\!]$'s traces.

$$\begin{aligned} D[\![d]\!] \setminus (\Sigma \setminus \{s.s1, s.s2\}) &\sqsubseteq_{\mathcal{T}} s.s2 \rightarrow s.s1 \rightarrow Stop \\ D[\![d]\!] \setminus (\Sigma \setminus \{s.s1, s.s2\}) &\sqsubseteq_{\mathcal{T}} s.s1 \rightarrow s.s2 \rightarrow Stop \end{aligned}$$

Conversely, if e is a XOR join gateway then $(e, d) \in trg$ if and only if one of following two traces refinements holds, that is, one of e's incoming sequence flows is performed in one of $D[\![d]\!]$'s traces.

$$\begin{aligned} D[\![d]\!] \setminus (\Sigma \setminus \{s.s1\}) &\sqsubseteq_{\mathcal{T}} s.s1 \rightarrow Stop \\ D[\![d]\!] \setminus (\Sigma \setminus \{s.s2\}) &\sqsubseteq_{\mathcal{T}} s.s2 \rightarrow Stop \end{aligned}$$

Finally, if e is a task element with incoming sequence flow $s1$ then $(e, d) \in trg$ if and only if the following traces refinement holds, that is, e's incoming sequence flow is performed in one of $D[\![d]\!]$'s traces.

$$D[\![d]\!] \setminus (\Sigma \setminus \{s.s1\}) \sqsubseteq_{\mathcal{T}} s.s1 \rightarrow Stop$$

We now define the characteristic set *trgs* such that $d \in trgs$ asserts the absence of un-triggered elements in a BPMN diagram d.

$$\begin{array}{|l} trgs : \mathbb{P}\ Diagram \\ \hline \forall\, d : Diagram \bullet d \in trgs \Leftrightarrow (\forall\, e : Element \bullet e \in_{\text{p}} \bigcup \{p : \text{ran}\, d.pool \bullet p.proc\} \Rightarrow (e, d) \in trg) \end{array}$$

Going back to our running example, we mechanically verify that the online shop business process is indeed a member of the set *trgs* using the FDR tool.

5.6.2 Liveness

In general a BPMN diagram is a failures refinement of another BPMN diagram precisely when the failures of former is a subset of the failures of latter:

Definition 5.2. ***Failures Refinement.*** *BPMN diagram P is a failures refinement of BPMN diagram Q if and only if $D[\![Q]\!] \sqsubseteq_{\mathcal{F}} D[\![P]\!]$.*

Using the failures refinement, we can also assert that a BPMN diagram meets a particular liveness property expressed as a CSP process or another BPMN diagram. Going back to our running example, we now consider the requirement expressed as the CSP process DP defined in Equation 5.21 as a liveness property, and in this case we check the following failures refinement assertion:

$$DP \sqsubseteq_{\mathcal{F}} Example \fatsemi Stop \setminus (\Sigma \setminus \alpha DP)$$

We observe this assertion does not hold and this is because the customer business process may decline the online shop's sale offer, that is, after event $w.SO$, process *Example* may perform events $w.DO$ and $w.RD$ and terminate. As a result the process $Example \fatsemi Stop \setminus (\Sigma \setminus \alpha DP)$ has the failure $(\langle w.SO \rangle, \{w.AO\})$, that is, it refuses $w.AO$ after performing $w.SO$. However any process that failures-refines DP must not refuse to perform $w.AO$ after the trace $\langle w.SO \rangle$.

Using the failures refinement, we can also express more general properties about BPMN diagrams such as deadlock freedom. The characteristic deadlock free process is defined by the CSP process DP defined in Equation 2.1. Here we reproduce the process definition.

$$DF = (\sqcap e : \Sigma \bullet e \rightarrow DF) \sqcap Skip$$

Process DF either offers at least one event recursively or terminates. The deadlock freedom assertion on BPMN diagram P is then expressed as the following refinement.

$$DF \sqsubseteq_{\mathcal{F}} D[\![P]\!]$$

Going back to our running example, we verify that the business process is deadlock free, that is, *Example* failures-refines DF.

We have shown how the refinement orderings defined over traces and failures semantics of CSP enable BPMN to be a specification language as well as a modelling language for implementation. Our process semantics also induces an equivalence relationship in which two BPMN processes are equivalent when each failures-refines the other. The notion of equivalence is formally defined as follows:

Definition 5.3. ***Equivalence.*** *Two BPMN processes P and Q are equivalent, denoted as $P \equiv_{BPMN} Q$ if and only if $D[\![Q]\!] \sqsubseteq_{\mathcal{F}} D[\![P]\!] \wedge D[\![P]\!] \sqsubseteq_{\mathcal{F}} D[\![Q]\!]$.*

5.7 Semantics of Composition

In this section we provide a CSP semantics for the syntactic operations defined in Section 4.6. For presentation purposes we use the following notational convention: For each syntactic operation Op that takes some component s of the before state and a list of input components ss, we write $Op(s, ss)$ to denote the component s' of the corresponding after state. For example, $SeqComp(proc, new?, from?, end?)$ denotes the BPMN pool constructed by applying the sequential composition operation *SeqComp* over the BPMN pool *proc* with input components *new*?, *from*? and *end*?.

We assume that all input components satisfy the operation's precondition. Given some BPMN element *ele*, we write $P(ele)$ to refer to the CSP process that models the behaviour of *ele*. We write CSP event $s.f$ for sequence flow f, $m.g$ for message flow g, and $c.e$ for the completion of end event e.

We provide the following definitions on BPMN elements to assist semantic constructions.

$$
\begin{aligned}
extin(ele) &= \Box\, i : (atom\ ele).in \bullet s.i \rightarrow Skip \\
parin(ele) &= |||\, i : (atom\ ele).in \bullet s.i \rightarrow Skip \\
intot(ele) &= \sqcap\, i : (atom\ ele).out \bullet s.i \rightarrow Skip \\
extot(ele) &= \Box\, i : (atom\ ele).out \bullet s.i \rightarrow Skip \\
parot(ele) &= |||\, i : (atom\ ele).out \bullet s.i \rightarrow Skip \\
extrc(ele) &= \Box\, i : (atom\ ele).receive \bullet m.i \rightarrow Skip \\
parsn(ele) &= |||\, i : (atom\ ele).send \bullet m.i \rightarrow Skip
\end{aligned}
$$

For some set of BPMN elements es, we define $alphas(es)$ to return the alphabet of the process semantics of elements in es, and $procs(es)$ to return the parallel composition of processes, each modelling the behaviour of an element in es.

$$
\begin{aligned}
alphas(es) &= \bigcup \{ i : es \bullet \alpha P(i) \} \\
procs(es) &= \|\, i : es \bullet \alpha P(i) \circ P(i)
\end{aligned}
$$

All operations described in this section add one or more end events to the BPMN process. Semantically, each BPMN element in the process must cooperate with those end events' completion. Specifically, let process $P(i)$ model the behaviour of BPMN element i directly contained in BPMN process B; $P(i)$ has the following form,

$$
P(i) = \begin{cases}
(C(i) \fatsemi (\Box\, j : E \bullet c.j \rightarrow Skip)) \,\Box\, (\Box\, j : E \bullet c.j \rightarrow Skip) & \text{if } i \text{ is a start event} \\
(C(i) \fatsemi c.i \rightarrow Skip) \,\Box\, (\Box\, j : (E \setminus \{i\}) \bullet c.j \rightarrow Skip) & \text{if } i \text{ is an end event} \\
(C(i) \fatsemi P(i)) \,\Box\, (\Box\, j : E \bullet c.j \rightarrow Skip) & \text{otherwise}
\end{cases}
$$

where $C(i)$ models i's sequence, message and exception flows. If i is an activity element, $C(i)$ also models its activity. Here set E is the set of end events directly contained in B. Let F be the set of new end events that is to be added to B, the following defines $addend(P(i), F)$ to model the composition of the completion event of end events in F with $P(i)$.

$$
addend(P(i), F) = \begin{cases}
(C(i) \fatsemi (\Box\, j : E \cup F \bullet c.j \rightarrow Skip)) \,\Box & \text{if } i \text{ is a start event} \\
(\Box\, j : E \cup F \bullet c.j \rightarrow Skip) & \\
(C(i) \fatsemi c.i \rightarrow Skip) \,\Box & \text{if } i \text{ is an end event} \\
(\Box\, j : (E \cup F) \setminus \{i\} \bullet c.j \rightarrow Skip) & \\
(C(i) \fatsemi W) \,\Box & \text{otherwise} \\
(\Box\, j : E \cup F \bullet c.j \rightarrow Skip) &
\end{cases}
$$

From Sections 5.7.1 to 5.7.5, we assume the before state component $proc$ to be some BPMN process directly containing elements $i \in I$, and set $E \subset I$ to identify the set of end events contained in $proc$. We also use the BPMN process shown in Figure 5.12 as a running example to illustrate the semantics of operations. We label this process $proc$. Equation 5.24 defines CSP process $P(proc)$ that models the behaviour of the BPMN process in the figure, where Equation 5.24 defines the behaviour of elements directly contained in subprocess B.

$$
\begin{aligned}
P(start) &= (s.s1 \rightarrow Skip \fatsemi (c.s4 \rightarrow Skip \Box c.s5 \rightarrow Skip)) \Box c.s4 \rightarrow Skip \Box c.s5 \rightarrow Skip \\
P(s1) &= (s.s1 \rightarrow Skip \fatsemi (s.s2 \rightarrow Skip \sqcap s.s3 \rightarrow Skip) \fatsemi P(s1)) \Box c.s4 \rightarrow Skip \Box c.s5 \rightarrow Skip \\
P(s2) &= (s.s2 \rightarrow Skip \fatsemi w.A \rightarrow Skip \fatsemi s.s4 \rightarrow Skip \fatsemi P(s2)) \Box c.s4 \rightarrow Skip \Box c.s5 \rightarrow Skip \\
P(s3) &= (s.s3 \rightarrow Skip \fatsemi B \fatsemi s.s5 \rightarrow Skip \fatsemi P(s3)) \Box c.s4 \rightarrow Skip \Box c.s5 \rightarrow Skip \\
P(s4) &= (s.s4 \rightarrow Skip \fatsemi c.s4 \rightarrow Skip) \Box c.s5 \rightarrow Skip \\
P(s5) &= (s.s5 \rightarrow Skip \fatsemi c.s5 \rightarrow Skip) \Box c.s4 \rightarrow Skip \\
P(proc) &= \Vert\, i : \{start, s1, s2, s3, s4, s5\} \bullet \alpha P(i) \circ P(i) \qquad (5.23)
\end{aligned}
$$

$$
\begin{aligned}
P(startB) &= (s.s6 \rightarrow Skip \fatsemi c.s7 \rightarrow Skip) \Box c.s7 \rightarrow Skip \\
P(s6) &= (s.s6 \rightarrow Skip \fatsemi w.C \rightarrow Skip \fatsemi s.s7 \rightarrow Skip \fatsemi P(s6)) \Box c.s7 \rightarrow Skip \\
P(s7) &= s.s7 \rightarrow Skip \fatsemi c.s7 \rightarrow Skip \\
B &= \Vert\, i : \{startB, s6, s7\} \bullet \alpha P(i) \circ P(i) \qquad (5.24)
\end{aligned}
$$

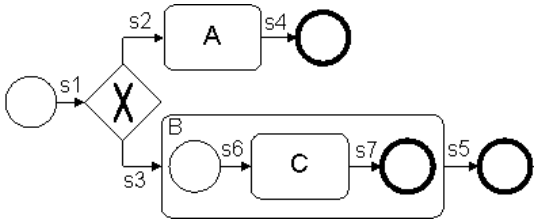


Figure 5.12: A BPMN process

5.7.1 Sequential Composition

The sequential composition operation *SeqComp* is defined in Section 4.6.2. Operation *SeqComp* replaces the end event identified by the expression (*ends proc*) *from*? and contained in *proc* with activity element *new*? and end event *end*?. The semantics of *new*? and *end*? are provided by the following processes,

$$
\begin{aligned}
P(new?) &= (extin(new?) \fatsemi N \fatsemi extot(new?) \fatsemi P(new?)) \Box (\Box\, i : (E \cup \{end?\}) \bullet c.i \rightarrow Skip) \\
P(end?) &= (extin(end?) \fatsemi c.end? \rightarrow Skip) \Box (\Box\, i : (E \setminus \{e\}) \bullet c.i \rightarrow Skip)
\end{aligned}
$$

where we let e to abbreviate (*ends proc*) *from*?, and process N models *new*?'s activity and is defined as follows.

$$
N = \begin{cases} w.new? \rightarrow Skip & \text{if } new? \text{ is a task} \\ procs(J) & \text{if } new? \text{ is a subprocess, directly containing set of elements } J \end{cases}
$$

Assuming that $P(e)$ models the behaviour of e, the semantics of $SeqComp(proc, new?, from?, end?)$ is given by the following process,

$$OB \,[\![\, AB \mid alphas(\{new?, end?\})\,]\!]\, (P(new?) \,[\![\, \alpha P(new?) \mid \alpha P(end?)\,]\!]\, P(end?))$$

where OB and AB are defined as follows.

$$
\begin{aligned}
OB &= \Vert\, i : I \setminus \{e\} \bullet ((\alpha P(i) \setminus \{c.e\}) \cup \{c.end?\}) \circ addend(P(i), \{end?\}) \\
AB &= (alphas(I \setminus \{e\}) \setminus \{c.e\}) \cup \{c.end?\}
\end{aligned}
$$

For example, we consider the BPMN process shown in Figure 5.13. It is the after state component $proc'$ of operation $SeqComp(proc, D, s4, e)$, where *proc* is the BPMN process shown in Figure 5.12

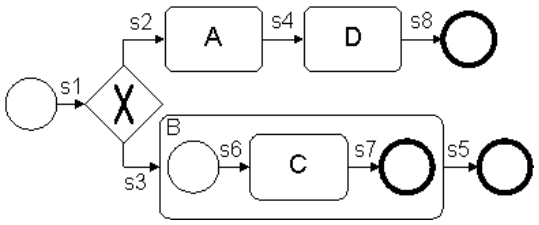


Figure 5.13: Applying *SeqComp*

and is modelled by CSP process $P(proc)$ defined in Equation 5.24. D identifies input task $new?$ and e identifies the input end event $end?$. Equation 5.25 defines the process that models $proc'$ of $SeqComp(proc, D, s4, s8)$, where B has already been defined in Equation 5.24.

$$
\begin{aligned}
P'(start) &= (s.s1 \rightarrow Skip \fatsemi (c.e \rightarrow Skip \Box c.s5 \rightarrow Skip)) \Box c.e \rightarrow Skip \Box c.s5 \rightarrow Skip \\
P'(s1) &= (s.s1 \rightarrow Skip \fatsemi (s.s2 \rightarrow Skip \sqcap s.s3 \rightarrow Skip) \fatsemi P'(s1)) \Box c.e \rightarrow Skip \Box c.s5 \rightarrow Skip \\
P'(s2) &= (s.s2 \rightarrow Skip \fatsemi w.A \rightarrow Skip \fatsemi s.s4 \rightarrow Skip \fatsemi P'(s2)) \Box c.e \rightarrow Skip \Box c.s5 \rightarrow Skip \\
P'(s3) &= (s.s3 \rightarrow Skip \fatsemi B \fatsemi s.s5 \rightarrow Skip \fatsemi P'(s3)) \Box c.e \rightarrow Skip \Box c.s5 \rightarrow Skip \\
P'(D) &= (s.s4 \rightarrow Skip \fatsemi w.D \rightarrow Skip \fatsemi s.e \rightarrow Skip \fatsemi P'(D)) \Box c.e \rightarrow Skip \Box c.s5 \rightarrow Skip \\
P'(e) &= (s.e \rightarrow Skip \fatsemi c.e \rightarrow Skip) \Box c.s5 \rightarrow Skip \\
P'(s5) &= (s.s5 \rightarrow Skip \fatsemi c.s5 \rightarrow Skip) \Box c.e \rightarrow Skip \\
P(proc') &= \Vert\, i : \{start, s1, s2, s3, s5, e, D\} \bullet \alpha P'(i) \circ P'(i) \qquad (5.25)
\end{aligned}
$$

5.7.2 Splits

There are two splits operations: *Split* and *EventSplitOp*. Both operations are defined in Section 4.6.3. We first consider operation *Split*. $Splits(proc, new?, from?, outs?)$ replaces the end event identified by $(ends\ proc)\ from?$ and contained in the before state component $proc$ with element $new?$, which is either a data-based XOR or an AND split gateway, and the set of end events $outs?$. The semantics of $new?$ are provided by the following process,

$$P(new?) = \textbf{let}\ W = (extin(new?) \fatsemi S \fatsemi W) \Box (\Box\, i : ((E \setminus \{e\}) \cup outs?) \bullet c.i \rightarrow Skip)\ \textbf{in}\ W$$

where e abbreviates $(ends\ proc)\ from?$ and S is defined as follows.

$$S = \begin{cases} intot(new?) & \text{if } new? \text{ is a data-based XOR split gateway} \\ parot(new?) & \text{if } new? \text{ is an AND split gateway} \end{cases}$$

From the definition of *Split* in Section 4.6.3, elements in $outs?$ have disjoint sets of incoming sequence flows. Moreover, the union of of these sets is the outgoing sequence flows of $new?$. As a result the semantics of elements in $outs?$ is modelled by the following parallel combination of processes QS,

$$QS = \Vert\, o : outs? \bullet \alpha Q(o) \circ Q(o)$$

where each process $Q(o)$ is defined as follows.

$$Q(o) = (extin(o) \fatsemi c.o \rightarrow Skip) \Box (\Box\, n : ((outs? \cup E) \setminus \{o, e\}) \bullet c.n \rightarrow Skip)$$

The semantics of $Splits(proc, new?, from?, outs?)$ is then given by the following process,

$$OB \,[\![\, AB \mid alphas(outs? \cup \{new?\}) \,]\!]\, (P(new?) \,[\![\, \alpha P(new?) \mid alphas(outs?) \,]\!]\, QS)$$

where OB and AB are defined as follows.

$$
\begin{aligned}
OB &= \Vert\, i : I \setminus \{e\} \bullet ((\alpha P(i) \setminus \{c.e\}) \cup \{o : outs? \bullet c.o\}) \circ addend(P(i), outs?) \\
AB &= (alphas(I \setminus \{e\}) \setminus \{c.e\}) \cup \{o : outs? \bullet c.o\}
\end{aligned}
$$

We now consider operation *EventSplitOp*. *EventSplitOp*(*proc*, *new*?, *from*?, *events*?, *ends*?) replaces the end event identified by (*ends proc*) *from*? and contained in the before state component *proc* with the event-based XOR split gateway *new*?, the set *events*? containing tasks and intermediate events and the set of end events *end*?. The semantics of *new*? is provided by the following process.

$$P(new?) \quad = \quad \textbf{let } W = (extin(new?) \fatsemi extot(new?) \fatsemi W) \Box (\Box\, i : ((E \setminus \{e\}) \cup ends?) \bullet c.i \rightarrow Skip) \textbf{ in } W$$

According to the definition of *EventSplitOp* in Section 4.6.3, the incoming sequence flows of elements in *events*? are the outgoing sequence flows of *new*?. Both *events*? and *ends*? have the same number of elements, and the outgoing sequence flows of elements in *events*? are the incoming sequence flows of elements in *ends*?. As a result the behaviour of elements in *events*? is modelled by the following parallel composition of processes QS.

$$QS = \parallel v : events? \bullet \alpha Q(v) \circ Q(v)$$

Here each process $Q(v)$ is defined as follows,

$$Q(v) = \textbf{let } W = (extin(v) \fatsemi S \fatsemi extot(v) \fatsemi W) \Box (\Box\, n : ((ends? \cup E) \setminus \{e\}) \bullet c.n \rightarrow Skip) \textbf{ in } W$$

where S is defined as follows.

$$S = \begin{cases} w.v \rightarrow Skip & \text{if } v \text{ is a task element} \\ Skip & \text{otherwise} \end{cases}$$

The semantics of the set of end event *ends* is defined by the following parallel combination of processes RS,

$$RS = \parallel o : ends? \bullet \alpha R(o) \circ R(o)$$

where each process $R(o)$ is defined as follows.

$$R(o) = (extin(o) \fatsemi c.o \rightarrow Skip) \Box (\Box\, n : ((ends? \cup E) \setminus \{o, e\}) \bullet c.n \rightarrow Skip)$$

Hence the semantics of *EventSplitOp*(*proc*, *new*?, *from*?, *events*?, *ends*?) is given by the following process,

$$\begin{array}{l} OB \,[\![\, AB \mid alphas(ends? \cup events? \cup \{new?\})]\!] \\ \quad (P(new?) \,[\![\, \alpha P(new?) \mid alphas(ends? \cup events?)]\!]\, (QS[alphas(events?)][alphas(ends?)]RS)) \end{array}$$

where OB and AB are defined as follows.

$$\begin{array}{rcl} OB & = & \parallel i : I \setminus \{e\} \bullet ((\alpha P(i) \setminus \{c.e\}) \cup \{o : ends? \bullet c.o\}) \circ addend(P(i), ends?) \\ AB & = & (alphas(I \setminus \{e\}) \setminus \{c.e\}) \cup \{o : ends? \bullet c.o\} \end{array}$$

For example, we consider the BPMN process shown in Figure 5.14. This process is the after state component *proc*′ of operation $Splits(proc, gate, s4, \{e, f\})$, where *proc* is the BPMN process shown in Figure 5.12 and is modelled by the CSP process $P(proc)$ defined in Equation 5.24. Here *gate* identifies input gateway *new*?; e identifies the end event in the figure with incoming sequence flow $s9$, while f identifies the end event with incoming sequence flow $s8$. The set $\{e, f\}$ therefore identifies the set *outs*?. Equation 5.26 defines $P(proc')$ that models *proc*′, where B has been defined in Equation 5.24.

$$\begin{array}{rcl} P'(start) & = & (s.s1 \rightarrow Skip \fatsemi \Box\, e : \{s8, s5.s9\} \bullet c.e \rightarrow Skip) \Box (\Box\, e : \{s8, s5.s9\} \bullet c.e \rightarrow Skip) \\ P'(s1) & = & (s.s1 \rightarrow Skip \fatsemi (s.s2 \rightarrow Skip \sqcap s.s3 \rightarrow Skip) \fatsemi P'(s1)) \Box (\Box\, e : \{s8, s5.s9\} \bullet c.e \rightarrow Skip) \\ P'(s2) & = & (s.s2 \rightarrow Skip \fatsemi w.A \rightarrow Skip \fatsemi s.s4 \rightarrow Skip \fatsemi P'(s2)) \Box (\Box\, e : \{s8, s5.s9\} \bullet c.e \rightarrow Skip) \\ P'(s3) & = & (s.s3 \rightarrow Skip \fatsemi B \fatsemi s.s5 \rightarrow Skip \fatsemi P'(s3)) \Box (\Box\, e : \{s8, s5.s9\} \bullet c.e \rightarrow Skip) \\ P'(gate) & = & (s.s4 \rightarrow Skip \fatsemi (s.s8 \rightarrow Skip \sqcap s.s9 \rightarrow Skip) \fatsemi P'(gate)) \Box (\Box\, e : \{s8, s5.s9\} \bullet c.e \rightarrow Skip) \\ P'(s5) & = & (s.s5 \rightarrow Skip \fatsemi c.s5 \rightarrow Skip) \Box c.s8 \rightarrow Skip \Box c.s9 \rightarrow Skip \\ P'(e) & = & (s.s8 \rightarrow Skip \fatsemi c.s8 \rightarrow Skip) \Box c.s5 \rightarrow Skip \Box c.s9 \rightarrow Skip \\ P'(f) & = & (s.s9 \rightarrow Skip \fatsemi c.s9 \rightarrow Skip) \Box c.s5 \rightarrow Skip \Box c.s8 \rightarrow Skip \\ P(proc') & = & \parallel i : \{start, s1, s2, s3, s5, e, f, gate\} \bullet \alpha P'(i) \circ P'(i) \end{array} \tag{5.26}$$

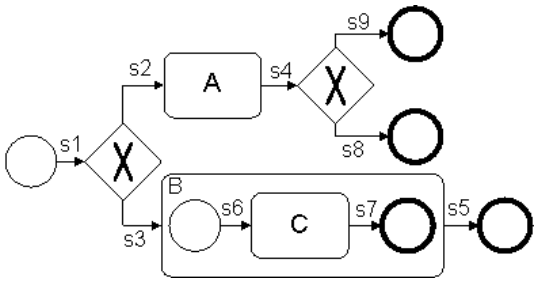


Figure 5.14: Applying *Splits*

5.7.3 Join

The join operation *JoinOp* is defined such that $JoinOp(proc, gate?, end?)$ replaces a set of end events identified by the expression $(ends\ proc)(\!|\ (atom\ gate?).in\ |\!)$ and contained in the before state component *proc* with the end event *end*? and either a data-based XOR or an AND join gateway *gate*?. This operation was defined in Section 4.6.4 on Page 44. We let R abbreviate the set of end events $(ends\ proc)(\!|\ (atom\ gate?).in\ |\!)$ and provide the semantics of R by the following parallel composition of processes RP,

$$RP = \ \|\ r : R \bullet \alpha P(r) \circ P(r)$$

where each process $P(r)$ models element $r \in R$ and has the following form.

$$P(r) = (extin(r) \fatsemi c.r \rightarrow Skip) \ \Box\ (\Box\ i : E \setminus \{r\} \bullet c.i \rightarrow Skip)$$

We provide the semantics of *gate*? and *end*? by the following processes,

$$\begin{array}{rcl} P(gate?) & = & \textbf{let}\ W = (S \fatsemi extot(gate?) \fatsemi W)\ \Box\ (\Box\ i : ((E \setminus R) \cup \{end?\}) \bullet c.i \rightarrow Skip)\ \textbf{in}\ W \\ P(end?) & = & (extin(end?) \fatsemi c.end? \rightarrow Skip)\ \Box\ (\Box\ i : (E \setminus R) \bullet c.i \rightarrow Skip) \end{array}$$

where process S is defined as follows.

$$S = \begin{cases} extin(gate?) & \text{if } new? \text{ is a XOR join gateway element} \\ parin(gate?) & \text{if } new? \text{ is an AND join gateway element} \end{cases}$$

Hence, the semantics of $JoinOp(proc, gate?, end?)$ is given by the following process,

$$OB\ [\![\ AB \mid alphas(\{gate?, end?\})\ ]\!]\ (P(gate?)\ [\![\ \alpha P(gate?) \mid \alpha P(end?)\ ]\!]\ P(end?))$$

where OB and AB are defined as follow.

$$\begin{array}{rcl} OB & = & \|\ i : I \setminus R \bullet ((\alpha P(i) \setminus \{r : R \bullet c.r\}) \cup \{c.end?\}) \circ addend(P(i), \{end?\}) \\ AB & = & (alphas(I \setminus R) \setminus \{r : R \bullet c.r\}) \cup \{c.end?\} \end{array}$$

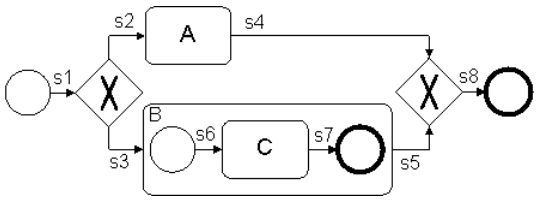


Figure 5.15: Applying *JoinOp*

For example, we consider the BPMN process shown in Figure 5.15. This process models the after state component $proc'$ of operation $JoinOp(proc, gate, s8)$, where $proc$ is the BPMN process shown in Figure 5.12 and is modelled by the process $P(proc)$ defined in Equation 5.24. Here $gate$ identifies the gateway $gate?$, and $s8$ identifies the end event $end?$. Equation 5.27 defines process $P(proc')$ that models $proc'$, where B is defined in Equation 5.24.

$$\begin{array}{rcl}
P'(start) & = & (s.s1 \rightarrow Skip \fatsemi c.s8 \rightarrow Skip) \Box c.s8 \rightarrow Skip \\
P'(s1) & = & (s.s1 \rightarrow Skip \fatsemi (s.s2 \rightarrow Skip \sqcap s.s3 \rightarrow Skip) \fatsemi P'(s1)) \Box c.s8 \rightarrow Skip \\
P'(s2) & = & (s.s2 \rightarrow Skip \fatsemi w.A \rightarrow Skip \fatsemi s.s4 \rightarrow Skip \fatsemi P'(s2)) \Box c.s8 \rightarrow Skip \\
P'(s3) & = & (s.s3 \rightarrow Skip \fatsemi B \fatsemi s.s5 \rightarrow Skip \fatsemi P'(s3)) \Box c.s8 \rightarrow Skip \\
P'(gate) & = & ((s.s4 \rightarrow Skip \Box s.s5 \rightarrow Skip) \fatsemi s.s8 \rightarrow Skip \fatsemi P'(gate)) \Box c.s8 \rightarrow Skip \\
P'(s8) & = & (s.s8 \rightarrow Skip \fatsemi c.s8 \rightarrow Skip) \\
P(proc') & = & \|\ i : \{start, s1, s2, s3, s8, gate\} \bullet \alpha P'(i) \circ P'(i)
\end{array} \tag{5.27}$$

5.7.4 Iteration

There are two iteration operations: *Loop* and *EventLoop*. We first consider *Loop*.

The operation *Loop* takes sequence flows $connect?$, $f2?$, $t2?$ and $from?$, and elements $split?$, $join?$ and $end?$ and applies the following three steps to the before state component $proc$:

1. Replace the end event, identified by $(ends\ proc)\ from?$, contained in $proc$ with either an AND or a data-based XOR split gateway $split?$ and the end event $end?$.
2. Add either an AND or a data-based XOR join gateway $join?$ to $proc$.
3. Replace the incoming sequence flow $f2?$ from the element, identified by the expression $(nonsends\ proc)\ f2?$, contained in $proc$ with $t2?$.

A syntactic definition of *Loop* is provided in Section 4.6.5. The semantics of $proc$, $split?$, $join?$ and $end?$ are provided by the following processes,

$$\begin{array}{rcl}
D[\![proc]\!] & = & (\ \|\ i : I \setminus J \bullet \alpha P(i) \circ P(i))\ [\![\ alphas(I \setminus J) \mid alphas(J)\ ]\!]\ (\ \|\ i : J \bullet \alpha P(i) \circ P(i)) \\
P(split?) & = & \textbf{let}\ W = (s.from? \rightarrow S \fatsemi W) \Box (\Box\ i : ((E \setminus \{e\}) \cup \{end?\}) \bullet c.i \rightarrow Skip)\ \textbf{in}\ W \\
P(join?) & = & \textbf{let}\ W = (T \fatsemi s.t2? \rightarrow W) \Box (\Box\ i : ((E \setminus \{e\}) \cup \{end?\}) \bullet c.i \rightarrow Skip)\ \textbf{in}\ W \\
P(end?) & = & (extin(end?) \fatsemi c.end? \rightarrow Skip) \Box (\Box\ i : (E \setminus \{e\}) \bullet c.i \rightarrow Skip)
\end{array}$$

where $J = \{e, m\}$, and processes S and T are defined as follows.

$$S = \begin{cases} intot(split?) & \text{if } split? \text{ is a data-based XOR split gateway} \\ parot(split?) & \text{if } split? \text{ is an AND split gateway} \end{cases}$$

$$T = \begin{cases} s.f2? \rightarrow Skip \Box s.connect? \rightarrow Skip & \text{if } join? \text{ is a data-based XOR join gateway} \\ s.f2? \rightarrow Skip \ |||\ s.connect? \rightarrow Skip & \text{if } join? \text{ is an AND join gateway} \end{cases}$$

Here e abbreviates $(ends\ proc)\ from?$ and process $P(e)$ models the behaviour of e. Similarly, we use m to abbreviate the expression $(nonsends\ proc)\ f2?$ and process $P(m)$ models the behaviour of m. Hence, the semantics of $Loop(proc, split?, join?, end?, connect?, from?, f2?, t2?)$ is defined by the following process,

$$\begin{array}{l}
OB\ [\![\ alphas(I \setminus \{e, m\}) \cup \{c.end?\} \mid alphas(\{split?, join?, end?, m'\})]\!] \\
\quad (P(m')\ [\![\ \alpha P(m') \mid alphas(\{split?, join?, end?\})]\!] \\
\quad\quad (P(split?)\ [\![\ \alpha P(split?) \mid alphas(\{join?, end?\})]\!] \\
\quad\quad\quad (P(join?)\ [\![\ \alpha P(join?) \mid \alpha P(end?)\ ]\!]\ P(end?))))
\end{array}$$

where OB and $P(m')$ are defined as follows; $P(m)[s.f2? \leftarrow s.t2?]$ renames all occurrences of $s.f2?$ to $s.t2?$ in $P(m)$.

$$\begin{array}{rcl} OB & = & \|\ i : I \setminus \{e, m\} \bullet ((\alpha P(i) \setminus \{e\}) \cup \{c.end?\}) \circ addend(P(i), \{end?\}) \\ P(m') & = & P(m)[s.f2? \leftarrow s.t2?] \end{array}$$

We now consider the operation *EventLoop*. *EventLoop* takes sequence flows *connect*?, $f2?$, $t2?$ and *from*?, elements *split*?, *join*? and *end*?, and the set *events*?, which consists of two elements, and applies the following three steps to the before state component *proc*:

1. Replace the end event, identified by (*ends proc*) *from*?, in *proc* with the event-based XOR split gateway *split*?, the set of elements *events*? and the end event *end*?; each element in the set *events*? is either a task or an intermediate event.
2. Add either a join gateway *join*? to *proc*.
3. Replace the incoming sequence flow $f2?$ of the element, identified by expression (*nonsends proc*) $f2?$, contained in *proc* with $t2?$.

A syntactic definition of *EventLoop* is provided in Section 4.6.5. We let e abbreviate (*ends proc*) *from*? and m to abbreviate (*nonsends proc*) $f2?$, and first model the semantics of the set of elements *events*? using the following parallel composition of processes EP,

$$EP = \|\ v : events? \bullet \alpha P(v) \circ P(v)$$

where each process $P(v)$ is defined as follows,

$$P(v) = \textbf{let } W = (extin(v) \fatsemi S \fatsemi extot(v) \fatsemi W) \Box (\Box\ n : ((ends? \cup E) \setminus \{e\}) \bullet c.n \rightarrow Skip) \textbf{ in } W$$

and S is defined as follows.

$$S = \begin{cases} w.v \rightarrow Skip & \text{if } v \text{ is a task} \\ Skip & \text{otherwise} \end{cases}$$

We provide the semantics of *proc*, *split*?, *join*? and *end*? using the following processes,

$$\begin{array}{rcl} D[\![proc]\!] & = & (\ \|\ i : I \setminus \{e, m\} \bullet \alpha P(i) \circ P(i)) \\ & & \quad [\![alphas(I \setminus \{e, m\}) \mid alphas(\{e, m\})\ ]\!]\ (P(e)\ [\![\alpha P(e) \mid \alpha P(m)\ ]\!]\ P(m)) \\ P(split?) & = & \textbf{let } W = (s.from? \rightarrow extot(split?) \fatsemi W) \Box (\Box\ i : ((E \setminus \{e\}) \cup \{end?\}) \bullet c.i \rightarrow Skip) \textbf{ in } W \\ P(join?) & = & \textbf{let } W = (T \fatsemi s.t2? \rightarrow W) \Box (\Box\ i : ((E \setminus \{e\}) \cup \{end?\}) \bullet c.i \rightarrow Skip) \textbf{ in } W \\ P(end?) & = & (extin(end?) \fatsemi c.end? \rightarrow Skip) \Box (\Box\ i : (E \setminus \{e\}) \bullet c.i \rightarrow Skip) \end{array}$$

where process T is defined as follows.

$$T = \begin{cases} s.f2? \rightarrow Skip \ \Box\ s.connect? \rightarrow Skip & \text{if } join? \text{ is a data-based XOR join gateway} \\ s.f2? \rightarrow Skip \ |||\ s.connect? \rightarrow Skip & \text{if } join? \text{ is an AND join gateway} \end{cases}$$

We let process $P(e)$ model the behaviour of e and $P(m)$ model that of m, the semantics of $EventLoop(proc, split?, join?, end?, events?, connect?, from?, f2?, t2?)$ is hence defined by the following process,

$$\begin{array}{l} OB\ [\![\ alphas(I \setminus \{e, m\}) \cup \{c.end?\} \mid alphas(\{split?, join?, end?, m'\} \cup events?)]\!] \\ \quad (P(m')\ [\![\ \alpha P(m') \mid alphas(\{split?, join?, end?\} \cup events?)]\!] \\ \quad\quad (EP\ [\![\ alphas(events?) \mid alphas(\{split?, join?, end?\})]\!] \\ \quad\quad\quad (P(split?)\ [\![\ \alpha P(split?) \mid alphas(\{join?, end?\})]\!] \\ \quad\quad\quad\quad (P(join?)\ [\![\ \alpha P(join?) \mid \alpha P(end?)\ ]\!]\ P(end?))))) \end{array}$$

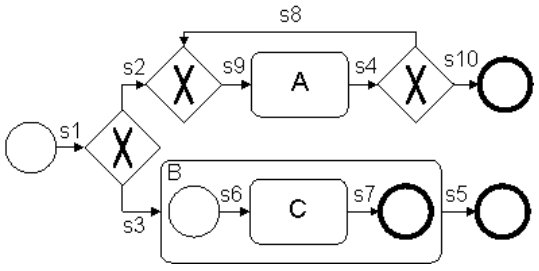


Figure 5.16: Applying *Loop*

where OB and $P(m')$ are defined as follows.

$$\begin{aligned} OB &= \|\ i : I \setminus \{e, m\} \bullet ((\alpha P(i) \setminus \{e\}) \cup \{c.end?\}) \circ addend(P(i), \{end?\}) \\ P(m') &= P(m)[s.f2? \leftarrow s.t2?] \end{aligned}$$

For example, consider the BPMN process shown in Figure 5.16. This process is the after state component $proc'$ of operation $Loop(proc, g1, g2, s10, s8, s4, s2, s9)$, where $proc$ is the BPMN process shown in Figure 5.12 and is modelled by the process $P(proc)$ defined in Equation 5.24. Here $g1$ identifies the input gateway *split*? while $g2$ identifies *join*?, $s10$ identifies the input end event *end*?, and $s8$, $s4$, $s2$, $s9$ identify sequence flows *connect*?, *from*?, $f2?$ and $t2?$ respectively. Equation 5.28 defines process $P(proc')$ that models $proc'$, where B is defined in Equation 5.24.

$$\begin{aligned} P'(start) &= (s.s1 \rightarrow Skip \fatsemi (c.s10 \rightarrow Skip \Box c.s5 \rightarrow Skip)) \Box c.s10 \rightarrow Skip \Box c.s5 \rightarrow Skip \\ P'(s1) &= (s.s1 \rightarrow Skip \fatsemi (s.s2 \rightarrow Skip \sqcap s.s3 \rightarrow Skip) \fatsemi P'(s1)) \Box c.s10 \rightarrow Skip \Box c.s5 \rightarrow Skip \\ P'(s3) &= (s.s3 \rightarrow Skip \fatsemi B \fatsemi s.s5 \rightarrow Skip \fatsemi P'(s3)) \Box c.s10 \rightarrow Skip \Box c.s5 \rightarrow Skip \\ P'(g2) &= ((s.s2 \rightarrow Skip \Box s.s8 \rightarrow Skip) \fatsemi s.s9 \rightarrow Skip \fatsemi P'(g2)) \Box c.s10 \rightarrow Skip \Box c.s5 \rightarrow Skip \\ P'(s2) &= (s.s9 \rightarrow Skip \fatsemi w.A \rightarrow Skip \fatsemi s.s4 \rightarrow Skip \fatsemi P'(s2)) \Box c.s10 \rightarrow Skip \Box c.s5 \rightarrow Skip \\ P'(g1) &= (s.s4 \rightarrow Skip \fatsemi (s.s8 \rightarrow Skip \sqcap s.s10 \rightarrow Skip) \fatsemi P'(g1)) \Box c.s10 \rightarrow Skip \Box c.s5 \rightarrow Skip \\ P'(s10) &= (s.s10 \rightarrow Skip \fatsemi c.s10 \rightarrow Skip) \Box c.s5 \rightarrow Skip \\ P'(s5) &= (s.s5 \rightarrow Skip \fatsemi c.s5 \rightarrow Skip) \Box c.s10 \rightarrow Skip \\ P(proc') &= \|\ i : \{start, s1, s2, s3, s5, g1, g2, s10\} \bullet \alpha P'(i) \circ P'(i) \end{aligned} \tag{5.28}$$

5.7.5 Exception

The interrupt operation specifies how to attach an intermediate event to an activity element to create an exception flow. The interrupt operation is specified by the schema *AddException*, which is defined as a disjunction of two operations: *AddNoRelatedErrorException* and *AddRelatedErrorException*. The schemas are defined in Section 4.6.6.

The operation *AddNoRelatedErrorException* takes sequence flows *eflow*? and *loc*?, the end event *end*? and the element type *etype*?, and adds an exception flow to the activity element identified by (*activities proc*) *loc*? and contained in *proc*. This exception flow is triggered by either a time lapse (*itime*), the arrival of a message (*imessage*) or an unspecified error during the activity's execution (*ierror*(*anyexception*)). The expression (*activities proc*) *loc*? ensures that the activity element, to which the exception flow is added, has an incoming sequence flow *loc*?.

We provide the semantics of *proc* and *end*? with the following processes,

$$\begin{aligned} D[\![proc]\!] &= (\ \|\ i : I \setminus \{m\} \bullet \alpha P(i) \circ P(i)) \ [\![\ alphas(I \setminus \{m\}) \mid \alpha P(m)\ ]\!]\ P(m) \\ P(end?) &= (extin(end?) \fatsemi c.end? \rightarrow Skip) \Box (\Box\ i : (E \setminus \{e\}) \bullet c.i \rightarrow Skip) \end{aligned}$$

where m abbreviates (*activities proc*) *loc*?. Process $P(m)$ models the behaviour of m and has the following

form,

$$\begin{array}{rcl} P(m) & = & \textbf{let} \quad X = ((N \fatsemi extot(m)) \triangle C) \,[\![\, \alpha M \,]\!]\, M \\ & & \qquad W = (s.loc? \rightarrow X \fatsemi W) \,\Box\, (\Box\, i : E \bullet c.i \rightarrow Skip) \;\textbf{in}\;\; W \end{array}$$

where process N denotes the activity of m and is defined as follows:

$$N = \begin{cases} w.m \rightarrow Skip & \text{if } m \text{ is a task} \\ procs(J) & \text{if } m \text{ is a subprocess, directly containing set of elements } J \end{cases}$$

Process C describes the exception flows of m and M coordinates the outgoing sequence flows and exception flows of m as possible "outcomes" of m. The semantics of $AddNoRelatedErrorException(proc, end?, etype?, eflow?, loc?)$ is hence defined by the following process,

$$OB \,[\![\, alphas(I \setminus \{m\}) \cup \{c.end?\} \mid alphas(\{end?, m'\}) \,]\!]\, (P(m') \,[\![\, \alpha P(m') \mid \alpha P(end?) \,]\!]\, P(end?))$$

where processes OB and $P(m')$ are defined as follow,

$$\begin{array}{rcl} OB & = & \|\, i : I \setminus \{m\} \bullet (\alpha P(i) \cup \{c.end?\}) \circ addend(P(i), \{end?\}) \\ P(m') & = & \textbf{let} \quad X = ((N \fatsemi extot(m)) \triangle C') \,[\![\, \alpha M \cup \{s.eflow?\} \,]\!]\, M' \\ & & \qquad C' = C \,\Box\, s.eflow? \rightarrow Skip \\ & & \qquad M' = M \,\Box\, s.eflow? \rightarrow Skip \\ & & \qquad W = (s.loc? \rightarrow X \fatsemi W) \,\Box\, (\Box\, i : (E \cup \{end?\}) \bullet c.i \rightarrow Skip) \;\textbf{in}\;\; W \end{array}$$

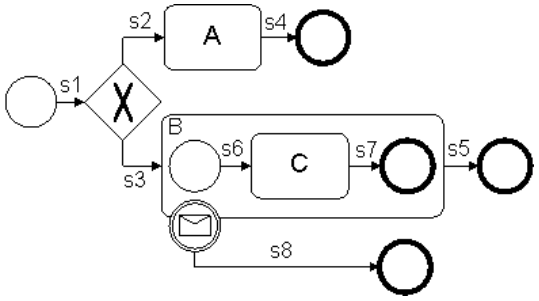


Figure 5.17: Applying *AddNoRelatedErrorException*

For example, we consider the BPMN process shown in Figure 5.17. This process is the after state component $proc'$ of operation $AddNoRelatedErrorException(proc, e, ierror(anyexception), s8, s2)$, where $proc$ is the BPMN process shown in Figure 5.12 and is modelled by the process $P(proc)$ defined in Equation 5.24. Here e identifies end event $end?$, $etype?$ is the value $eerror(anyexception)$, and $s8$ and $s2$ identify sequence flows $eflow?$ and $loc?$ respectively. Equation 5.29 defines $P(proc')$ that models the behaviour of $proc'$, where B is defined in Equation 5.24.

$$\begin{array}{rcl} P'(start) & = & (s.s1 \rightarrow Skip \fatsemi \Box\, i : \{s4, s5, e\} \bullet c.i \rightarrow Skip) \,\Box\, (\Box\, i : \{s4, s5, e\} \bullet c.i \rightarrow Skip) \\ P'(s1) & = & (s.s1 \rightarrow Skip \fatsemi (s.s2 \rightarrow Skip \sqcap s.s3 \rightarrow Skip) \fatsemi P'(s1)) \,\Box\, (\Box\, i : \{s4, s5, e\} \bullet c.i \rightarrow Skip) \\ P'(s2) & = & (s.s2 \rightarrow Skip \fatsemi w.A \rightarrow Skip \fatsemi s.s4 \rightarrow Skip \fatsemi P'(s2)) \,\Box\, (\Box\, i : \{s4, s5, e\} \bullet c.i \rightarrow Skip) \\ P'(s3) & = & \textbf{let}\; C = (\Box\, i : \alpha B \setminus \{s.s5, s.s8\} \bullet i \rightarrow C) \,\Box\, s.s8 \rightarrow Skip \,\Box\, s.s5 \rightarrow Skip \\ & & \qquad K = (B \fatsemi s.s5 \rightarrow Skip) \triangle s.s8 \rightarrow Skip \\ & & \textbf{in}\; (s.s3 \rightarrow Skip \fatsemi (K \,[\![\, \alpha C \,]\!]\, C) \fatsemi P'(s3)) \,\Box\, (\Box\, i : \{s4, s5, e\} \bullet c.i \rightarrow Skip) \\ P'(s4) & = & (s.s4 \rightarrow Skip \fatsemi c.s4 \rightarrow Skip) \,\Box\, c.s5 \rightarrow Skip \,\Box\, c.e \rightarrow Skip \\ P'(s5) & = & (s.s5 \rightarrow Skip \fatsemi c.s5 \rightarrow Skip) \,\Box\, c.s4 \rightarrow Skip \,\Box\, c.e \rightarrow Skip \\ P'(e) & = & (s.s8 \rightarrow Skip \fatsemi c.e \rightarrow Skip) \,\Box\, c.s4 \rightarrow Skip \,\Box\, c.s5 \rightarrow Skip \\ P(proc') & = & \|\, i : \{start, s1, s2, s3, s4, s5, e\} \bullet \alpha P'(i) \circ P'(i) \end{array} \tag{5.29}$$

The operation *AddRelatedErrorException* takes as input components sequence flows *eflow*?, *sflow*? and *loc*?, end event *end*?, and element types *etype*? and *type*?. The operation identifies some subprocess element m in *proc* by the expression (*activities proc*) *loc*?; the expression (*activities proc*) *loc*? ensures that m is an activity and has an incoming sequence flow *loc*?. Moreover, the precondition of *AddRelatedErrorException* ensures m is a subprocess. Specifically, the operation performs the following two steps:

1. Add an exception flow, that is a pair (*eflow*?, *etype*?), to m, where *etype*? is a type of an intermediate error event.
2. Replace the type of an non-trigger end event directly contained in m with an end error event type such that it has the same error code as *etype*?.

We provide the semantics of *proc* and *end*? with the following processes,

$$\begin{aligned} D[\![proc]\!] &= (\ \|\ i : I \setminus \{m\} \bullet \alpha P(i) \circ P(i)) \ [\![\ alphas(I \setminus \{m\}) \mid \alpha P(m)\ ]\!]\ P(m) \\ P(end?) &= (extin(end?) \fatsemi c.end? \rightarrow Skip) \ \Box\ (\Box\ i : (E \setminus \{e\}) \bullet c.i \rightarrow Skip) \end{aligned}$$

where $P(m)$ is the process semantics of m and has the following form.

$$\begin{aligned} P(m) = \ \textbf{let}\ & X = ((N\ [\![\ alphas(J \setminus \{k\}) \mid \alpha P(k)\ ]\!]\ P(k)) \fatsemi extot(m)) \triangle C)\ [\![\ \alpha M\ ]\!]\ M \\ & N = \ \|\ j : (J \setminus \{k\}) \bullet \alpha P(j) \circ P(j) \\ & W = (s.loc? \rightarrow X \fatsemi W) \ \Box\ (\Box\ i : E \bullet c.i \rightarrow Skip)\ \textbf{in}\ \ W \end{aligned}$$

Here element $k \in J$ is an end event identified by the expression (*ends* (*content ele*)) *sflow*? where *ele* = (*activities proc*) *loc*? is the subprocess m. Specifically, k is an end event whose type is replaced with an end error event type as described in Step ii of the operation.

Similar to the definition of the semantics of *AddNoRelatedErrorException*, process C describes the exception flows of m and M coordinates the outgoing sequence flows and exception flows of m as possible "outcomes" of m. The semantics of *AddRelatedErrorException*(*proc*, *end*?, *etype*?, *eflow*?, *sflow*?, *loc*?) is hence defined by the following process,

$$OB\ [\![\ alphas(I \setminus \{m\}) \cup \{c.end?\} \mid alphas(\{end?, m'\})\ ]\!]\ (P(m')\ [\![\ \alpha P(m') \mid \alpha P(end?)\ ]\!]\ P(end?))$$

where processes OB and $P(m')$ are defined as follows,

$$\begin{aligned} OB &= \ \|\ i : I \setminus \{m\} \bullet (\alpha P(i) \cup \{c.end?\}) \circ addend(P(i), \{end?\}) \\ P(m') = \ \textbf{let}\ & X = (((N\ [\![\ alphas(J \setminus \{k\}) \mid \alpha K\ ]\!]\ K \fatsemi extot(m)) \triangle C')\ [\![\ \alpha M \cup \{s.eflow?\}\ ]\!]\ M' \\ & C' = C \ \Box\ s.eflow? \rightarrow Skip \\ & M' = M \ \Box\ e.(errorCode\ etype?) \rightarrow s.eflow? \rightarrow Skip \\ & K = (extin(k) \fatsemi e.(errorCode\ etype?) \rightarrow c.k \rightarrow Skip) \ \Box\ (\Box\ i : F \setminus \{k\} \bullet c.i \rightarrow Skip) \\ & W = (s.loc? \rightarrow X \fatsemi W) \ \Box\ (\Box\ i : (E \cup \{end?\}) \bullet c.i \rightarrow Skip)\ \textbf{in}\ \ W \end{aligned}$$

and $F \subseteq J$ is the set of end events directly contained in subprocess m.

For example, we consider the BPMN process shown in Figure 5.18. This process is the after state component *proc*′ of operation *AddRelatedErrorException*(*proc*, e, *ierror*(*exception*($e1$)), $s8$, $s2$), where *proc* is the BPMN process shown in Figure 5.12 and is modelled by the process P(*proc*) defined in Equation 5.24. Here e identifies end event *end*?, *etype*? is the value *ierror*(*exception*($e1$)), for error code $e1$, and $s8$ and $s2$ identify sequence flows *eflow*? and *loc*? respectively. Equation 5.30 defines process P(*proc*′) that models *proc*′, where B' is defined in Equation 5.31.

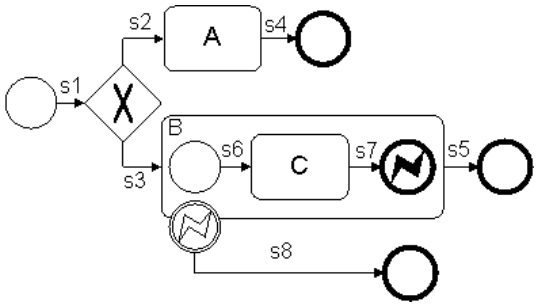


Figure 5.18: Applying *AddRelatedErrorException*

$$
\begin{array}{rcl}
P'(start) & = & (s.s1 \rightarrow Skip \fatsemi \Box\, i : \{s4, s5, e\} \bullet c.i \rightarrow Skip) \Box (\Box\, i : \{s4, s5, e\} \bullet c.i \rightarrow Skip) \\
P'(s1) & = & (s.s1 \rightarrow Skip \fatsemi (s.s2 \rightarrow Skip \sqcap s.s3 \rightarrow Skip) \fatsemi P'(s1)) \Box (\Box\, i : \{s4, s5, e\} \bullet c.i \rightarrow Skip) \\
P'(s2) & = & (s.s2 \rightarrow Skip \fatsemi w.A \rightarrow Skip \fatsemi s.s4 \rightarrow Skip \fatsemi P'(s2)) \Box (\Box\, i : \{s4, s5, e\} \bullet c.i \rightarrow Skip) \\
P'(s3) & = & \textbf{let}\ C = (\Box\, i : \alpha B' \setminus \{s.s5, e.e1, s.s8\} \bullet i \rightarrow C) \Box e.e1 \rightarrow s.s8 \rightarrow Skip \Box s.s5 \rightarrow Skip \\
 & & \quad K = (B' \fatsemi s.s5 \rightarrow Skip) \triangle s.s8 \rightarrow Skip \\
 & & \textbf{in}\ (s.s3 \rightarrow Skip \fatsemi (K \,[\![\, \alpha C \,]\!]\, C) \fatsemi P'(s3)) \Box (\Box\, i : \{s4, s5, e\} \bullet c.i \rightarrow Skip) \\
P'(s4) & = & (s.s4 \rightarrow Skip \fatsemi c.s4 \rightarrow Skip) \Box c.s5 \rightarrow Skip \Box c.e \rightarrow Skip \\
P'(s5) & = & (s.s5 \rightarrow Skip \fatsemi c.s5 \rightarrow Skip) \Box c.s4 \rightarrow Skip \Box c.e \rightarrow Skip \\
P'(e) & = & (s.s8 \rightarrow Skip \fatsemi c.e \rightarrow Skip) \Box c.s4 \rightarrow Skip \Box c.s5 \rightarrow Skip \\
P(proc') & = & \|\, i : \{start, s1, s2, s3, s4, s5, e\} \bullet \alpha P'(i) \circ P'(i)
\end{array} \tag{5.30}
$$

$$
\begin{array}{rcl}
P'(startB) & = & (s.s6 \rightarrow Skip \fatsemi c.s7 \rightarrow Skip) \Box c.s7 \rightarrow Skip \\
P'(s6) & = & (s.s6 \rightarrow Skip \fatsemi w.C \rightarrow Skip \fatsemi s.s7 \rightarrow Skip \fatsemi P'(s6)) \Box c.s7 \rightarrow Skip \\
P'(s7) & = & s.s7 \rightarrow Skip \fatsemi e.e1 \rightarrow Skip \fatsemi c.s7 \rightarrow Skip \\
B' & = & \|\, i : \{startB, s6, s7\} \bullet \alpha P'(i) \circ P'(i)
\end{array} \tag{5.31}
$$

5.7.6 Collaboration

The collaboration operation is defined on the state schema *Diagram* (BPMN diagrams), which contains one or more BPMN pools. Specifically, this operation is defined by the schema *ConnectMgeFlowDiagram* which promotes the local operations *AddSendMgeFlow* and *AddReceiveMgeFlow* from the state schema *Pool* to the state schema *Diagram* using the promotion schema *DiagramPromote*; the schemas are defined in Section 4.6.7.

The operation *AddSendMgeFlow* takes a message flow *msg*? and a sequence flow *tos*?, and adds *msg*? as an outgoing message flow to one of the following elements contained in the before state component *proc*: an end message event identified by the expression (*msgsends proc*) *tos*? and a task identified by the expression (*activities proc*) *tos*?. Conversely, the operation *AddReceiveMgeFlow* takes a message flow *msg*? and a sequence flow *tor*?, and adds *msg*? as an incoming message flow to one of the following elements contained *proc*: a message event identified by the expression (*msgrecs proc*) *tor*? and an activity identified by the expression (*activities proc*) *tor*?. The promotion schema *DiagramPromote* ensures that *msg*? is fresh from the before state *Diagram*, and that *AddSendMgeFlow* is applied to the before state *Pool* identified by *id*1? and *AddReceiveMgeFlow* is applied to the before state *Pool* identified by *id*2?, such that *id*1? and *id*2? are not the same.

We first consider the operation *AddSendMgeFlow*. We let I be the set of elements contained in the before state component *proc*, $m \in I$ be the element with incoming sequence flow *tos*?, and $E \subset I$ be the

set of end events in I. The semantics of $AddSendMgeFlow(proc, tos?, msg?)$ is defined by the following process $AS(proc, tos?, msg?)$,

$$AS(proc, tos?, msg?) = procs(I \setminus \{m\}) \,[\![\, alphas(I \setminus \{m\}) \mid \alpha P(m')\}) \,]\!]\, P(m')$$

where process $P(m')$ is defined as follows:

$$P(m') = \begin{cases} T(m') & \text{if } m \text{ a task} \\ V(m') & \text{if } m \text{ an end message event} \end{cases}$$

Here processes $T(m')$ and $V(m')$ have the following forms, where process C describes the exception flows of m and M coordinates the outgoing sequence flows and exception flows of m.

$$\begin{array}{lcl} T(m') & = & \textbf{let} \;\; X = ((w.m \rightarrow ((parrc(m) \;|||\; m.msg? \rightarrow Skip) \fatsemi extot(m))) \triangle C) \,[\![\, \alpha M \,]\!]\, M \\ & & \qquad W = ((s.tos? \rightarrow Skip \;|||\; parrc(m)) \fatsemi X \fatsemi W) \,\Box\, (\Box\, i : E \bullet c.i \rightarrow Skip) \;\textbf{in} \;\; W \\ V(m') & = & (s.tos? \rightarrow m.msg? \rightarrow c.m \rightarrow Skip) \,\Box\, (\Box\, n : E \setminus \{m\} \bullet c.n \rightarrow Skip) \end{array}$$

We now consider the operation $AddReceiveMgeFlow$. We let J be the set of elements contained in the before state component $proc$, $n \in J$ be either the activity with incoming sequence flow $tor?$ or the start event with the outgoing sequence flow $tor?$, and set $F \subset J$ be the set of end events in J. The semantics of $AddReceiveMgeFlow(proc, tor?, msg?)$ is defined by the following process $AR(proc, tor?, msg?)$,

$$AR(proc, tor?, msg?) = procs(J \setminus \{n\}) \,[\![\, alphas(J \setminus \{n\}) \mid \alpha P(n')\}) \,]\!]\, P(n')$$

where process $P(n')$ is defined as follows:

$$P(n') = \begin{cases} S(n') & \text{if } n \text{ is a task and } msg? \text{ is added as its incoming message flow} \\ R(n') & \text{if } n \text{ is an activity and } msg? \text{ is added to a message event attached to } n \\ U(n') & \text{if } n \text{ is a start message event} \\ Q(n') & \text{if } n \text{ is a intermediate message event} \end{cases}$$

Here processes $S(n')$, $R(n')$, $U(n')$ and $Q(n')$ have the following forms,

$$\begin{array}{lcl} S(n') & = & \textbf{let} \;\; X = ((w.n \rightarrow parrc(n) \fatsemi extot(n))) \triangle C) \,[\![\, \alpha M \,]\!]\, M \\ & & \qquad W = ((s.tor? \rightarrow Skip \;|||\; parrc(n) \;|||\; m.msg? \rightarrow Skip) \fatsemi X \fatsemi W) \,\Box\, FP \;\textbf{in} \;\; W \\ R(n') & = & \textbf{let} \;\; X = ((N \fatsemi parrc(n) \fatsemi extot(n))) \triangle C) \,[\![\, \alpha M \,]\!]\, M \\ & & \qquad C = D \,\Box\, m.msg? \rightarrow extot(O) \\ & & \qquad M = L \,\Box\, m.msg? \rightarrow extot(O) \\ & & \qquad W = ((s.tor? \rightarrow Skip \;|||\; parrc(n)) \fatsemi X \fatsemi W) \,\Box\, FP \;\textbf{in} \;\; W \\ U(n') & = & (m.msg? \rightarrow extot(n) \fatsemi FP) \,\Box\, FP \\ Q(n') & = & \textbf{let} \;\; W = ((s.tor? \rightarrow Skip \;|||\; m.msg? \rightarrow Skip) \fatsemi extot(n) \fatsemi W) \,\Box\, FP \;\textbf{in}\; W \end{array}$$

where $FP = \Box\, i : F \bullet c.i \rightarrow Skip$, N is defined as follows.

$$N = \begin{cases} w.n \rightarrow Skip & \text{if } n \text{ is a task} \\ procs(K) & \text{if } n \text{ is a subprocess, directly containing set of elements } K \end{cases}$$

Specifically, for $S(n')$, process C describes the exception flows of n and M coordinates the outgoing sequence flows and exception flows of n. For $R(n')$, on the other hand, we let O be the intermediate message event attached to n, to which $msg?$ is added, then process $D \,\Box\, extot(O)$ describes the exception flows of n and $L \,\Box\, extot(O)$ coordinates the outgoing sequence flows and exception flows of n.

We now consider $ConnectMgeFlowDiagram$, we define DP to be the CSP process that models the collaboration of BPMN pools defined by the $pools$ component of the before state $Diagram$, where for each $i \in \mathrm{dom}\, pools$, we define $PL(i) = procs(pools(i).proc))$.

$$\begin{array}{lcl} OB & = & \|\, i : ((\mathrm{dom}\, pools) \setminus \{id1?, id2?\}) \bullet \alpha PL(i) \circ PL(i) \\ DP & = & OB \,[\![\, \alpha OB \mid \alpha PL(id1?) \cup \alpha PL(id2?) \,]\!]\, (PL(id1?) \,[\![\, \alpha PL(id1?) \mid \alpha PL(id2?) \,]\!]\, PL(id2?)) \end{array}$$

The semantics of $ConnectMgeFlowDiagram(pools, id1?, id2?, msg?, tos?, tor?)$ is then defined by the following process.

$$
\begin{array}{l}
OB \,[\![\, \alpha OB \mid \alpha PL(id1?) \cup \alpha PL(id2?) \cup \{m.msg?\} \,]\!] \\
\quad (AS(pools(id1?).proc, tos?, msg?) \,[\![\, \alpha PL(id1?) \cup \{m.msg?\} \mid \alpha PL(id2?) \cup \{m.msg?\} \,]\!] \\
\quad\quad AR(pools(id2?).proc, tor?, msg?))
\end{array}
$$

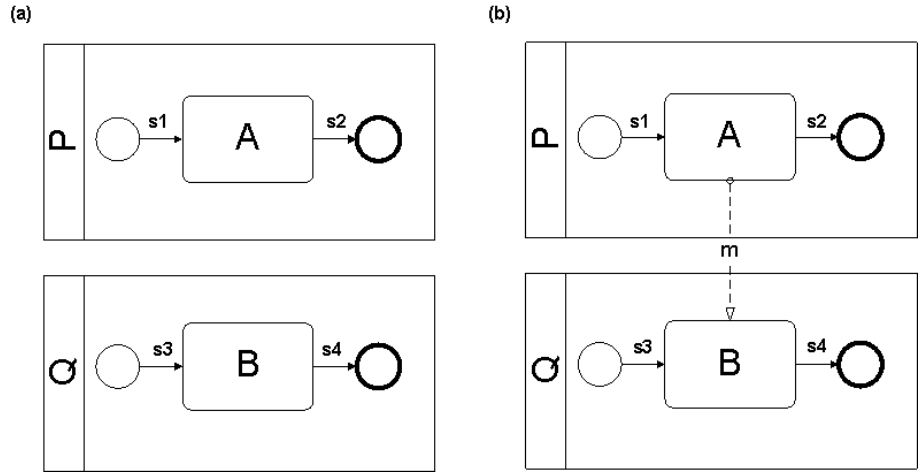


Figure 5.19: Applying *ConnectMgeFlowDiagram*

For example, we consider the BPMN processes shown in Figure 5.19. Figure 5.19(a) shows the before component *pools* of operation $ConnectMgeFlowDiagram(pools, p, q, m, s1, s3)$, where p identifies BPMN pool P and q identifies BPMN pool Q, m identifies the message flow $msg?$, and $s1$ and $s3$ identify sequence flows $tos?$ and $tor?$ respectively. Equation 5.32 defines process $D[\![pools]\!]$ that models *pools*. On the other hand, Figure 5.19(b) shows the after component *pools* of operation $ConnectMgeFlowDiagram(pools, p, q, m, s1, s3)$. Equation 5.33 defines process $D[\![pools']\!]$ that models $pools'$.

$$
\begin{array}{rcl}
P(start1) & = & (s.s1 \rightarrow Skip \fatsemi c.s2 \rightarrow Skip) \,\Box\, c.s2 \rightarrow Skip \\
P(s1) & = & (s.s1 \rightarrow Skip \fatsemi w.A \rightarrow Skip \fatsemi s.s2 \rightarrow Skip \fatsemi P(s1)) \,\Box\, c.s2 \rightarrow Skip \\
P(s2) & = & s.s2 \rightarrow Skip \fatsemi c.s2 \rightarrow Skip \\
PL(p) & = & \|\, i : \{start1, s1, s2\} \bullet \alpha P(i) \circ P(i) \\
P(start1) & = & (s.s3 \rightarrow Skip \fatsemi c.s4 \rightarrow Skip) \,\Box\, c.s4 \rightarrow Skip \\
P(s3) & = & (s.s3 \rightarrow Skip \fatsemi w.B \rightarrow Skip \fatsemi s.s4 \rightarrow Skip \fatsemi P(s3)) \,\Box\, c.s4 \rightarrow Skip \\
P(s4) & = & s.s4 \rightarrow Skip \fatsemi c.s4 \rightarrow Skip \\
PL(q) & = & \|\, i : \{start2, s3, s4\} \bullet \alpha P(i) \circ P(i) \\
D[\![pools]\!] & = & PL(p) \,[\![\, \alpha PL(p) \mid \alpha PL(q) \,]\!]\, PL(q)
\end{array}
\tag{5.32}
$$

$$
\begin{aligned}
P(start1) &= (s.s1 \rightarrow Skip \fatsemi c.s2 \rightarrow Skip) \,\Box\, c.s2 \rightarrow Skip \\
P(s1) &= (s.s1 \rightarrow Skip \fatsemi w.A \rightarrow Skip \fatsemi m.m \rightarrow Skip \fatsemi s.s2 \rightarrow Skip \fatsemi P(s1)) \,\Box\, c.s2 \rightarrow Skip \\
P(s2) &= s.s2 \rightarrow Skip \fatsemi c.s2 \rightarrow Skip \\
PL(p) &= \|\, i : \{start1, s1, s2\} \bullet \alpha P(i) \circ P(i) \\
P(start1) &= (s.s3 \rightarrow Skip \fatsemi c.s4 \rightarrow Skip) \,\Box\, c.s4 \rightarrow Skip \\
P(s3) &= ((s.s3 \rightarrow Skip \,|||\, m.m \rightarrow Skip) \fatsemi w.B \rightarrow Skip \fatsemi s.s4 \rightarrow Skip \fatsemi P(s3)) \,\Box\, c.s4 \rightarrow Skip \\
P(s4) &= s.s4 \rightarrow Skip \fatsemi c.s4 \rightarrow Skip \\
PL(q) &= \|\, i : \{start2, s3, s4\} \bullet \alpha P(i) \circ P(i) \\
D[\![pools']\!] &= PL(p) \,[\![\, \alpha PL(p) \mid \alpha PL(q) \,]\!]\, PL(q)
\end{aligned}
\tag{5.33}
$$

5.8 Compositional Development

Our semantic model permits mechanical verification of BPMN processes against behavioural properties via FDR. Nevertheless, when business processes become large, their behaviour would become complex and model checking might not be feasible due to the *state space explosion*. Compositional development allows one to verify behavioural correctness of a complex system by exploiting the transitive and monotonic properties of refinements [Ros98].

Specifically, we would like to show that the composition operations considered in Section 5.7 to be monotonic with respect to failures refinement ($\sqsubseteq_{\mathcal{F}}$). However, we observe that in general these operations are not monotonic with respect to $\sqsubseteq_{\mathcal{F}}$.

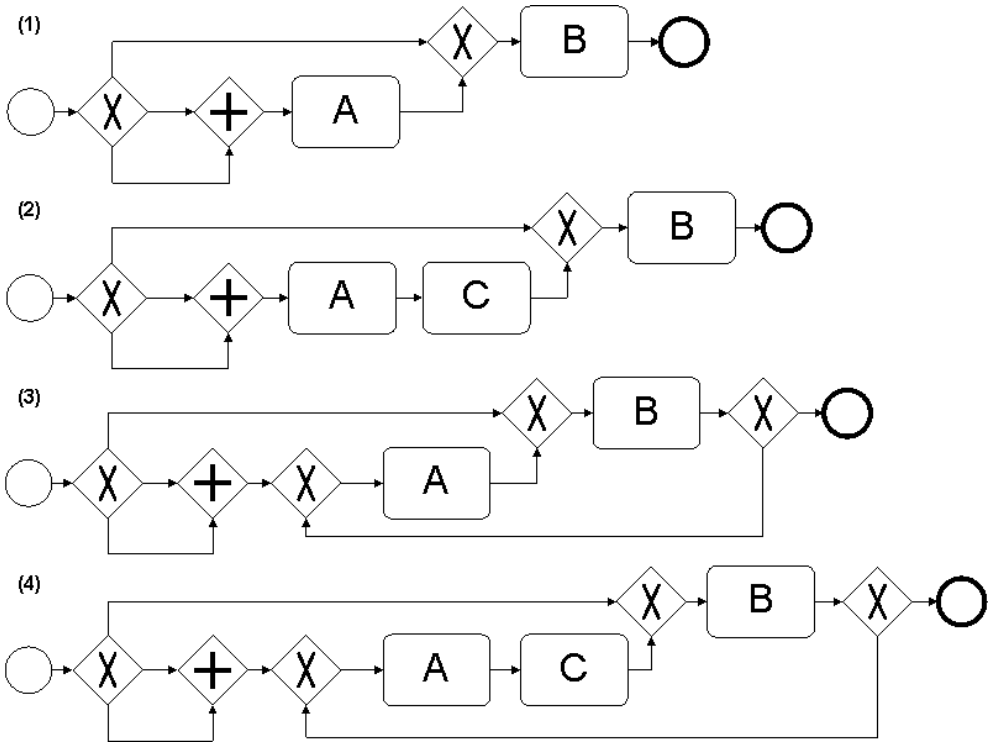


Figure 5.20: A non-monotonic scenario

Consider the BPMN processes in Figure 5.20. They are constructed by a combination of operations *SeqComp*, *Split*, *JoinOp* and *Loop*. We let $P1$, $P2$, $P3$ and $P4$ denote the BPMN processes shown in Figures 5.20(a), (b), (c) and (d) respectively, and a, b and c denote task elements A, B and C of the

processes. We observe that both $P1$ and $P2$ deadlock, because in both cases not all of the AND join gateway's incoming sequence flows can be triggered. As a result we have $\{(a, P1), (a, P2), (b, P2)\} \cap trg = \emptyset$. Due to the un-triggered elements, we also observe that $P1$ and $P2$ admit the same behaviour, that is, $P1 \equiv_{BPMN} P2$. We now consider $P3$ and $P4$ that are constructed by applying the operation *Loop* to $P1$ and $P2$ respectively. We observe that unlike $P1$ and $P2$, a can be triggered in both $P3$ and $P4$. However, after performing a, $P4$ can trigger element b, while $P3$ cannot, as a result we have $P3 \not\equiv_{BPMN} P4$. This shows that in general not only the operations are non-monotonic, but more importantly the equivalence $\equiv_{BPMN}$ is not congruent with respect to these operations. The reason is because $P1$ and $P2$ contain elements that are not reachable, that is, $P1 \notin trgs$ and $P2 \notin trgs$. To ensure the composition operations can be applied monotonically, we assume the following conditions about BPMN diagrams and processes and state that the composition operations are monotonic with respect to the failures refinement:

(a) Given any two BPMN processes X and Y, such that X directly contains the set of elements I and Y directly contains the set of elements J, if we have $D_p[\![X]\!] \sqsubseteq_{\mathcal{F}} D_p[\![Y]\!]$, where $D_p[\![X]\!] = \Vert\, i : I \bullet \alpha P(i) \circ P(i)$ and $D_p[\![Y]\!] = \Vert\, j : J \bullet \alpha P(j) \circ P(j)$, then the set of elements $I \setminus J$ can be partitioned into two sets: $\langle A, B\rangle$ partition $(I \setminus J)$:

  (i) Each element $e \in A$ is either a data-based XOR split gateway, a subprocess, a nondeterministic sequential multiple instance activity or a nondeterministic parallel multiple instance activity such that there exists an element $e' \in J \setminus I$ where $P(e) \sqsubseteq_{\mathcal{F}} P(e')$.

  (ii) For each element $f \in B$, there exists some data-based XOR split gateway $e \in A$, such that there exists some element $e' \in J \setminus I$ where $P(e) \sqsubseteq_{\mathcal{F}} P(e')$. Moreover, $atom(e').out \subset atom(e).out$ and either $atom(f).in \subset atom(e).out \setminus atom(e').out$ or there exists an element $g \in B$ such that $atom(g).in \subset atom(e).out \setminus atom(e').out$ and $(g, f) \in edge(X)^+$.

  Furthermore, $J \setminus I$ can be partitioned into two sets: $\langle M, N\rangle$ partition $J \setminus I$:

  (iii) For each element $m \in M$, there exists exactly one element $e \in A$ such that $P(e) \sqsubseteq_{\mathcal{F}} P(m)$,

  (iv) Each element $e \in N$ is a XOR join gateway such that there exists a XOR join gateway $f \in B$ with the same outgoing sequence flow and whose set of incoming sequence flows is a superset of e's.

(b) Given any two BPMN diagrams X and Y, such that X is the collaboration of the set of pools I and Y is the collaboration of the set of pools J. If we have $D_p[\![X]\!] \sqsubseteq_{\mathcal{F}} D_p[\![Y]\!]$, where $D[\![X]\!] = \Vert\, i : \mathrm{dom}\, X.pools \bullet \alpha Pl(X, i) \circ Pl(X, i)$ and $D[\![Y]\!] = \Vert\, j : \mathrm{dom}\, Y.pools \bullet \alpha Pl(Y, i) \circ Pl(Y, i)$, and $Pl(X, i) = D_p[\![X.pools(i).proc]\!]$, we have $\mathrm{dom}\, X.pools = \mathrm{dom}\, Y.pools$ and $\forall\, i : \mathrm{dom}\, X.pools \bullet Pl(X, i) \sqsubseteq_{\mathcal{F}} Pl(Y, i)$.

Condition a is appropriate: according to our process semantics, only data-based XOR split gateways, subprocesses, nondeterministic sequential multiple instance activities and nondeterministic parallel multiple instance activities have nondeterministic behaviour. Moreover, Condition a relaxes the condition that the comparing BPMN processes must satisfy the absence of un-triggered elements property. Condition b is also appropriate: it is reasonable to compare behaviour of business collaborations if they have same participants during compositional development.

Theorem 5.4. ***Monotonicity.*** *Assuming BPMN diagrams satisfy Conditions a and b, the operations SeqComp, Split, EventSplitOp, JoinOp, Loop, EventLoop, AddException and ConnectMgeFlowDiagram are monotonic with respect to the failures refinement.*

Proof. See Page 257 (Section C.1 in Appendix C). □

We use Condition a to show that *SeqComp*, *Split*, *EventSplitOp*, *JoinOp*, *Loop*, *EventLoop* and *AddException* are monotonic. We now informally describe how we show that *SeqComp* is monotonic using two BPMN processes in Figure 5.21 as an example. We let X and Y denote the BPMN processes in Figure 5.21(1) and 5.21(2) respectively, and $I = \{e1, x1, O, Q, S, x2, e2\}$ and $J = \{e1, x3, O, S', x4, e2\}$ be the set of elements directly contained in X and Y. We see that $P(x1) \sqsubseteq_{\mathcal{F}} P(x3)$ and assume that $P(S) \sqsubseteq_{\mathcal{F}} P(S')$, then due to Condition a, we can partition $I \setminus J$ into $A = \{x1, S\}$ and $B = \{Q, x2\}$, and

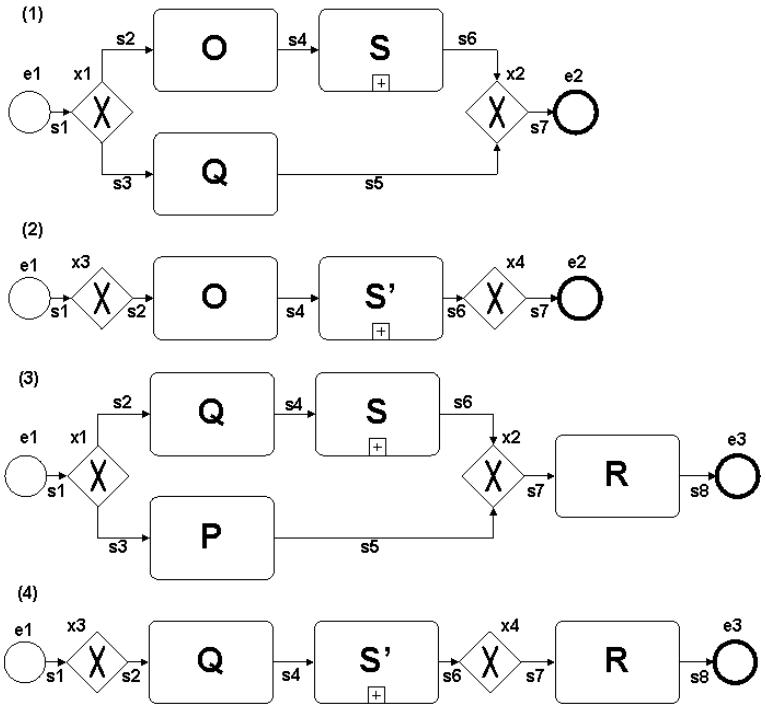


Figure 5.21: An illustration of Condition a

$J \setminus I$ into $M = \{x3, S'\}$ and $N = \{x4\}$. As a result the BPMN processes X and Y satisfy Condition a. Now we consider *SeqComp* that adds a new task R. This operation replaces the end event $e2$ with R and a new end event $e3$. The results are illustrated in Figures 5.21(3) and (4) respectively. where $e3$'s incoming sequence flow is R's outgoing sequence flow, and R's incoming sequence flow is that of $e2$.

Due to Condition a, we can reconstruct X and Y stepwise, starting with composing $\{P(e1), P(O), P(S), P(x1)\}$ in parallel for X and $\{P(e1), P(O), P(S'), P(x3)\}$ for Y. Since $P(S')$ and $P(x3)$ refine $P(S)$ and $P(x1)$, Y refines X. Now we consider elements P, $x2$, $x4$ and $e2$. We observe that composing Q and $x2$ in parallel onto Y is semantically equivalent to composing $x4$ in parallel onto Y. The reason is as follows: The process $P(Q)$ initially offers to either perform the sequence flow $s3$ or cooperate with an end event to completion and terminate. Since $P(x3)$ is composed in parallel for Y synchronising on the same alphabet as that of $P(x1)$ for X, we observe that $P(Q)$ can only cooperate with an end event to completion and terminate. Similarly the only difference between $P(x4)$ and $P(x3)$ are that $P(x3)$ initially also offers to perform sequence flow $s5$. However, since $P(Q)$ can only cooperate with an end event to completion and terminate, sequence flow $s5$ can never be triggered. As a result, we arrive at the parallel composition of $\{P(e1), P(O), P(S), P(x1), P(Q), P(x3)\}$ for X and that of $\{P(e1), P(O), P(S'), P(x3), P(Q), P(x3)\}$ for Y, and by definition of the parallel operator, we have Y refining X. Finally the operation *SeqComp* semantically composes $P(R)$ and $P(e3)$ in parallel onto X and Y, giving the resulting parallel composition of $\{P(e1), P(O), P(S), P(x1), P(Q), P(x3), P(R), P(e3)\}$ for X and that of $\{P(e1), P(O), P(S'), P(x3), P(Q), P(x3), P(R), P(e3)\}$ for Y. Again using the monotonic property of the parallel operator, we have the result that Y refines X.

Going back to the BPMN processes $P1$ and $P2$ on Figures 5.20, we observe that $P1$ and $P2$ do not satisfy Condition a: If we let I and J be the set of elements directly contained in $P1$ and $P2$, we notice that $c \in J \setminus I$ but c can be characterised in neither the partition N (aiii) nor the partition M (aiv).

We use Condition b to show *ConnectMgeFlowDiagram* is monotonic. Recall that the semantics of a BPMN diagram is given by the parallel composition of CSP processes, such that each of the processes corresponds to the behaviour of a BPMN element in that diagram. Due to the monotonicity of the CSP parallel composition operator. Informally given two BPMN diagrams X and Y such that Y refines X, we follow the same strategy for showing *SeqComp* monotonic and reconstruct X and Y stepwise to arrive

as the after state of *ConnectMgeFlowDiagram* while maintaining the refinement relation between Y and X.

In general, for any subprocess s, the CSP process that models s's behaviour can be generalised as $C[S]$ where S is the CSP process that models elements directly contained in s, and $C[.]$ is a CSP process context that models the incoming, outgoing sequence and message flows as well as exception flows of s. The following result shows that refinements are preserved from S to $C[S]$.

Theorem 5.5. *Given any subprocess s satisfying Conditions a and b, and such that its behaviour is modelled by CSP process $C[S]$, where S is the CSP process that models elements directly contained in s and $C[.]$ is a CSP process context that models s's sequence, message and exception flows. Let t be any subprocess whose behaviour is modelled by CSP process $C[T]$ and T is the CSP process that models elements directly contained in t. If we have both $S \sqsubseteq_{\mathcal{F}} T$ and $C[S] \sqsubseteq_{\mathcal{F}} C[T]$, then we have $C[S'] \sqsubseteq_{\mathcal{F}} C[T']$ where S' and T' are the results of applying any one of the operations SeqComp, Split, EventSplitOp, JoinOp, Loop, EventLoop, and AddException on S and T respectively.*

Proof. See Page 257 (Section C.1 in Appendix C). □

A BPMN process's behaviour is defined as a parallel composition of CSP processes, each modelling the behaviour of an element directly contained in the BPMN process. A consequence of Theorems 5.4 and 5.5 is that refinement is preserved between a subprocess and the BPMN process that directly contains it. The following result lifts operations *SeqComp*, *Split*, *EventSplitOp*, *JoinOp*, *Loop*, *EventLoop*, and *AddException* to BPMN pools and diagrams, and follows immediately from the fact that the CSP parallel operator $\|$ is monotonic with respect to refinements.

Corollary 5.6. *Given a BPMN process X that directly contains some subprocess s, such that X's behaviour is modelled by the CSP process $P(s) \,[\![\, \alpha P(s) \mid \alpha D \,]\!]\, D$, where D models the behaviour of other elements directly contained in X. For any s' such that $P(s) \sqsubseteq_{\mathcal{F}} P(s')$ we have*

$$P(s) \,[\![\, \alpha P(s) \mid \alpha D \,]\!]\, D \sqsubseteq_{\mathcal{F}} P(s') \,[\![\, \alpha P(s') \mid \alpha D \,]\!]\, D$$

Due to monotonicity, the equivalence $\equiv_{BPMN}$ defined in Definition 5.3 is a congruence with respect to the composition operations described above.

Corollary 5.7. *Assuming BPMN processes satisfy Conditions a and b, the equivalence $\equiv_{BPMN}$ is a congruence with respect to operations SeqComp, Split, EventSplitOp, JoinOp, Loop, EventLoop, AddException and ConnectMgeFlowDiagram.*

A congruence relationship allows one to substitute one part of a BPMN process with another that is semantically equivalent and obtain the same BPMN process.

5.8.1 Running Example

We now revisit our running example. Figure 5.22 shows an optimistic version of the customer business process that is originally shown at the top of Figure 5.10. It is modelled by BPMN pool *OpCustomer*. After receiving an offer from the online shop, the customer spends some time deciding, which is modelled by the intermediate timer event, but always accepts the offer. Equation 5.34 defines process *OCP* that

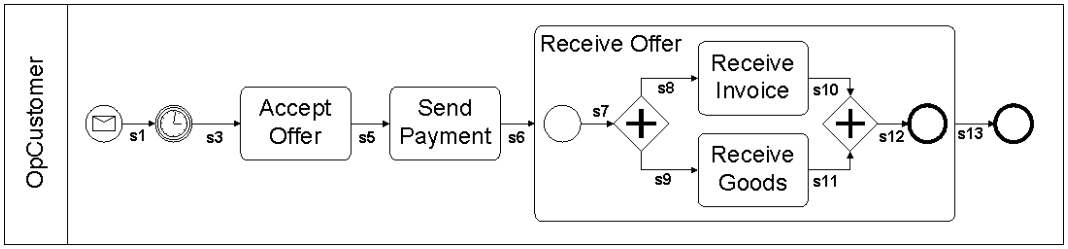


Figure 5.22: An optimistic customer

models pool *OpCustomer*, where for all $i \in \{sCP, s3, s5, s6, s13\}$ process $P(i)$ is defined in Equation 5.16.

$$\begin{aligned} P(s1) &= (s.s1 \rightarrow \mathit{Skip} \fatsemi (s.s2 \rightarrow \mathit{Skip} \sqcap s.s3 \rightarrow \mathit{Skip}) \fatsemi P(s1)) \,\Box\, c.s14 \rightarrow \mathit{Skip} \\ OCP &= \Big\| i : \{sCP, s1, s3, s5, s6, s13\} \bullet \alpha P(i) \setminus \{c.s4\} \circ P(i) \end{aligned} \tag{5.34}$$

In fact the optimistic customer process is a refinement of the customer process; we verify this by checking the refinement $CP \sqsubseteq_{\mathcal{F}} OCP$ using FDR. Suppose we extend the customer's business process with the following return policy.

> After receiving the goods and the invoice, the customer may decide to either keep the goods or return them for repair. Depending on the policy of the online shop, if the customer chooses to return her goods for repair, the shop may either provide a full refund, or repair the goods and deliver them back to the customer. After every repair, the customer has the choice to send the goods back again if further repairs are required.

Figure 5.23 shows the result of extending the optimistic customer business process with the above return policy. Here we observe that this extension can be constructed by a combination of operations *SeqComp*, *Split* and *EventSplitOp*, *JoinOp* and *EventLoop*. Note that the same combination of operations can be applied to the original customer business process to model this return policy. If we let CP' be the resulting CSP process modelling the extended version of CP and OCP' be that of OCP, by Theorem 5.4, we have $CP' \sqsubseteq_{\mathcal{F}} OCP'$.

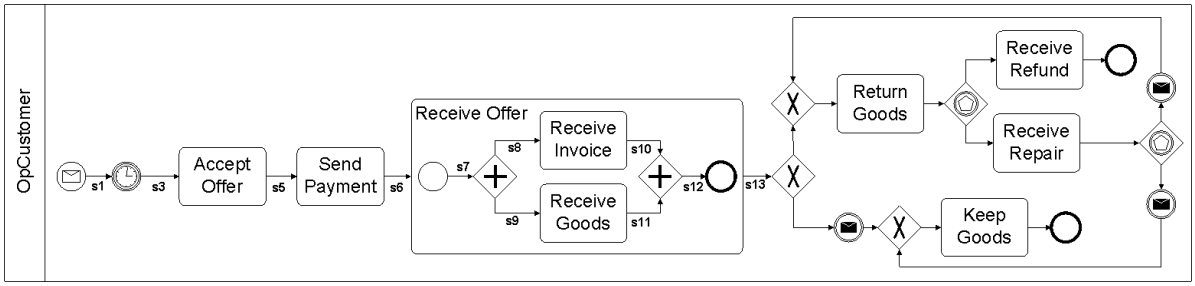


Figure 5.23: Extending the customer business process

5.9 Behavioural Compatibility

BPMN diagrams with more than one pool represent business collaborations, in which each BPMN pool is a business participant. An example of a business collaboration is our online shop running example shown in Figure 5.10.

5.9.1 Responsiveness

Generally in a system of interacting components, individual components may use *services* provided externally and will require assurance both of appropriate functionality and of *responsiveness* [RRS05]. In particular the question of responsiveness of interoperating components, modelled in CSP, has been studied by Reed, Sinclair and Roscoe [RSR04]. Informally, the question of responsiveness is such that, given a component modelled by some process P, which is itself deadlock-free, and placing it in parallel with another component, modelled by some process Q, whether Q could cause P to block. Here we consider it to be reasonable to combine interoperating components, modelled as CSP processes, using the parallel operator [CPT01]. Formally Reed et al. provided two binary relations *RespondsToLive* and *RespondsTo* over CSP processes under the failures model as follow:

Definition 5.8. *For any processes P and Q, Q RespondsToLive P on A for $A \subseteq J^{\checkmark}$ if and only if for all traces $s \in \mathrm{seq}(\alpha P \cup \alpha Q)$ we have $(s, A) \in \mathcal{F}[\![P \,[\![\, J^{\checkmark} \,]\!]\, Q]\!] \Rightarrow (s \upharpoonright \alpha P, A) \in \mathcal{F}[\![P]\!]$ where $A^{\checkmark}$ is a set of events $A \cup \{\checkmark\}$. We write $s \upharpoonright A$ for the sequence whose members are those of s which are in A. We say Q RespondsToLive P if and only if Q RespondsToLive P on $J^{\checkmark}$.*

Definition 5.9. *For any processes P and Q where there exists a set J of shared events, Q RespondsTo P if and only if for all traces $s \in \text{seq}(\alpha P \cup \alpha Q)$ and event sets X, such that $u = s \upharpoonright \alpha P$ and $t = s \upharpoonright \alpha Q$ and $(u, X) \in \mathcal{F}[\![P]\!] \wedge (initials(P/u) \cap J^{\checkmark}) \setminus X \neq \emptyset \Rightarrow (t, (initials(P/u) \cap J^{\checkmark}) \setminus X) \notin \mathcal{F}[\![Q]\!]$ where $initials(P/s)$ is the set of possible events for P after trace s.*

The condition Q *RespondsToLive* P on A says that the parallel combination $P \parallel Q$ may refuse the set of events A after trace s if P could refuse A after the trace $s \upharpoonright \alpha P$. Note that *RespondsTo* is the weakest refinement-closed strengthening of *RespondsToLive*, and we say Q is a responsive plugin to P if Q *RespondsTo* P holds. A relation $\oplus$ is refinement-closed if and only if for all processes P and Q such that if $P \oplus Q$, then it is the case that $P' \oplus Q'$ for all $P \sqsubseteq P'$ and $Q \sqsubseteq Q'$. Furthermore Reed et al. have shown the following two results [RSR04]:

Theorem 5.10. *Q RespondsTo $P \Leftrightarrow Q'$ RespondsToLive P' for all Q' and P' such that $P \sqsubseteq_{\mathcal{F}} P'$ and $Q \sqsubseteq_{\mathcal{F}} Q'$.*

Theorem 5.11. *Suppose $N = \big\| i : \{1 \,..\, N\} \bullet \alpha P(i) \circ P(i)\}$ is a network of CSP processes. Suppose that Q is a plug-in process whose alphabet J is disjoint from $\alpha P(i) \cap \alpha P(j)$ for each $i \neq j$, then if N is deadlock free, $J \cap \alpha P(j) \neq \emptyset$ for at least one j and Q RespondsToLive $P(i)$ for each i with $\alpha P(i) \cap J \neq \emptyset$, we have $N \,[\![\, \alpha N \mid \alpha Q \,]\!]\, Q$ is also deadlock free.*

As a direct consequence of the above two theorems, we may establish the following theorem.

Theorem 5.12. *Given a network of components $N = \{i : I \bullet P(i)\}$, indexed by I, where each component is modelled by process $P(i)$ for $i \in I$ and the network $PS = \big\| i : I \bullet \alpha P(i) \circ P(i)$ is deadlock-free. Suppose process Q such that $\alpha Q \cap (\alpha P(i) \cap \alpha P(j)) = \emptyset$ for $i, j : I$ and $i \neq j$, then if*

$$\exists\, i : I \bullet \alpha Q \cap \alpha P(i) \neq \emptyset \wedge \forall\, i : I \bullet \alpha Q \cap \alpha P(i) \neq \emptyset \Rightarrow Q \text{ RespondsTo } P(i)$$

then $PS \,[\![\, \bigcup\{i : I \bullet \alpha P(i)\} \mid \alpha Q \,]\!]\, Q$ is also deadlock-free.

Proof. From Theorem 5.10 we know *RespondsTo* implies *RespondsToLive* and from Theorem 5.11 this holds for *RespondsToLive*, then by transitivity, this also holds for *RespondsTo*. □

Furthermore, Reed et al. demonstrate in their paper how both binary relations may be verified mechanically using the FDR tool. Here we outline the procedure for translating the binary relation *RespondsTo* into a refinement check that can be checked by the FDR tool; the complete procedure can be found in Reed et al.'s paper [RSR04, Appendix B]. Briefly, to check if Q *RespondsTo* P on the alphabet J, one constructs a process P' that alternates between performing events from a given trace of P and the corresponding trace of P^*, where $P^* = P[a : \alpha P \bullet a \leftarrow a^*]$ is P with all its events augmented with $*$, and that if P refuses X after an odd length trace $s \frown \langle a^* \rangle$, P' refuses $X \setminus \{a\}$. This process P' is then composed with Q' synchronised on events in J, where Q' is Q with event $\Sigma \setminus J$ hidden. As a result Q *RespondsTo* P on J if and only if $P' \,[\![\, J \,]\!]\, Q'$ has no deadlock after an odd-length trace whose last member is in $J^* = \{a : J \bullet a^*\}$. This check can then be specified as a refinement check $Spec \sqsubseteq_{\mathcal{F}} P' \,[\![\, J \,]\!]\, Q'$ that can be fed into the FDR tool, where *Spec* is defined as follows:

$$\begin{aligned} Spec \;=\; & (\Box\, a : J \bullet a^* \rightarrow ((\sqcap\, b : J \bullet b \rightarrow Spec) \,\Box\, (Stop \sqcap Skip \sqcap \Box\, b : \alpha P \setminus J \bullet b \rightarrow Spec))) \\ & (\Box\, a : \alpha P \setminus J \bullet a^* \rightarrow (Stop \sqcap Skip \sqcap (\Box\, a : \alpha P \bullet a \rightarrow Spec))) \sqcap Skip \sqcap Stop \end{aligned}$$

We consider to be reasonable to consider deadlock-freedom to be a basic notion of correctness in business processes, and adopt Reed et al.'s theory of responsiveness to study compatibility between participants in a business collaboration.

5.9.2 Compatibility

While in the theory of responsiveness the relation Q *RespondsTo* P holds even if P could deadlock, causing $P \parallel Q$ also to deadlock, in the study of a business collaboration we wish to establish the notion of compatibility whereby each participant is itself deadlock-free and given any pair of interacting participants (via message flows), one participant must be a responsive plugin to the other participant. Furthermore, in the theory of responsiveness both binary relations Q *RespondsTo* P and Q *RespondsToLive* P

hold on the assumption that both P and Q may only perform events from $J = \alpha P \cap \alpha Q$, that is, their joint alphabet. To relax this assumption we eagerly abstract the behaviour not relevant in the joint alphabet. We formalise the notion of compatibility between two BPMN process as follows:

Definition 5.13. ***Compatibility.*** *BPMN processes p and q that interact via the set of message flows M and both satisfy the state schema Pool are compatible, denoted by the predicate $compatible(p, q)$, if and only if $DF \sqsubseteq_{\mathcal{F}} P \wedge DF \sqsubseteq_{\mathcal{F}} Q \wedge (P \setminus H \; RespondsTo \; Q \setminus H \vee Q \setminus H \; RespondsTo \; P \setminus H)$, where $H = \Sigma \setminus \{s : M \bullet m.s\}$, $P = D_p[\![p]\!]$ and $Q = D_p[\![q]\!]$.*

Theorem 5.14. *For any BPMN processes p and q, if $compatible(p, q)$ then their collaboration is deadlock-free, that is, $compatible(p, q) \Rightarrow DF \sqsubseteq_{\mathcal{F}} (D_p[\![p]\!] \,[\![\, \alpha D_p[\![p]\!] \mid \alpha D_p[\![q]\!] \,]\!]\, D_p[\![q]\!])$.*

Proof. See Page 258 (Section C.2 in Appendix C). □

For example, in the online shop example shown in Figure 5.10, where CSP process *CP* models the behaviour of BPMN pool *Customer* and *OS* models that of *OnlineShop*. We first verify that processes *CP* defined in Equation 5.16 and *OS* defined in Equation 5.18 to be deadlock free by checking the refinements $DF \sqsubseteq_{\mathcal{F}} CP$ and $DF \sqsubseteq_{\mathcal{F}} OS$, where *DF* is the characteristic deadlock free process defined in Equation 5.6.2. We also check that $OS \setminus H \; RespondsTo \; CP \setminus H$ where $H = \Sigma \setminus \{m.m1, m.m2, m.m3, m.m4, m.m5, m.m6\}$ is the set of events associated with message flows and is shared between *CP* and *OS*. By Definition 5.13, $compatible(OnlineShop, Customer)$, and by Theorem 5.14, their collaboration is deadlock-free.

One pleasing result is that compatibility is *refinement-closed*.

Theorem 5.15. *For any BPMN processes P and Q, if $compatible(P, Q)$, then for all BPMN processes P' and Q' such that $D_p[\![P]\!] \sqsubseteq_{\mathcal{F}} D_p[\![P']\!]$ and $D_p[\![Q]\!] \sqsubseteq_{\mathcal{F}} D_p[\![Q']\!]$, $compatible(P', Q')$.*

Proof. See Page 258 (Section C.2 in Appendix C). □

This refinement closure property encourages independent compositional development using monotonic operations defined in the previous section. For example, we know that the original customer business process *Customer* is compatible with the online shop business process *OnlineShop*, that is, $compatible(OnlineShop, Customer)$. In Section 5.8.1, we showed the optimistic version of the customer business process *OpCustomer* to be a refinement of *Customer*, that is, $CP \sqsubseteq_{\mathcal{F}} OCP$, where *OCP* models the behaviour of pool *OpCustomer*. By Theorem 5.15, we know that *OpCustomer* is also compatible with *OnlineShop*, that is, $compatible(OnlineShop, OpCustomer)$.

In general, if we let G be one of the monotonic operations *SeqComp*, *Split*, *EventSplitOp*, *JoinOp*, *Loop*, *EventLoop*, and *AddException*, if $compatible(G(P), G(Q))$, for all $D_p[\![P]\!] \sqsubseteq_{\mathcal{F}} D_p[\![P']\!]$ and $D_p[\![Q]\!] \sqsubseteq_{\mathcal{F}} D_p[\![Q']\!]$, $compatible(G(P'), G(Q'))$.

Naturally one would like to extend the property of compatibility to business collaborations with any finite number of participants. Given an existing deadlock-free business collaboration we would like to know if the collaboration is still deadlock-free if we add another compatible participant. This allows us to construct business collaboration *incrementally*, even in a complex situation. We formalise this notion as follows:

Theorem 5.16. *Given a collaboration of business participants (BPMN processes) $B = \{i : I \bullet B(i)\}$, indexed by I, where the semantics of each business participant is denoted by process $P(i) = D_p[\![B(i)]\!]$ for $i \in I$ and the collaboration $C = \Big\| \, i : I \bullet \alpha P(i) \circ P(i)$ is deadlock-free. Suppose there is a new business participant R, denoted by $Q = D_p[\![R]\!]$ such that $\alpha Q \cap (\alpha P(i) \cap \alpha P(j)) = \emptyset$ for $i, j : I$ and $i \neq j$ (an appropriated assumption as by definition each message flow connects exactly two participants). Then if*

$$\exists\, i : I \bullet \alpha Q \cap \alpha P(i) \neq \emptyset \wedge \forall\, i : I \bullet (\alpha Q \cap \alpha P(i) \neq \emptyset \Rightarrow compatible(R, B(i)))$$

then $E = C \,[\![\bigcup \{i : I \bullet \alpha P(i)\} \mid \alpha Q \,]\!]\, Q$ is also deadlock-free.

Proof. See Page 258 (Section C.2 in Appendix C). □

For example, recall that we extended the customer business process with a return policy in Section 5.8.1. While the online shop business process is responsible for the shop's sales, the shop has a separate repair and maintenance department. This is modelled by the BPMN pool *Maintenance* shown at the bottom of Figure 5.24. This figure also shows how to add *Maintenance* as a new participant to the collaboration. By Theorem 5.16, this new collaboration is deadlock free if $compatible(OnlineShop, Customer)$, $compatible(Customer, Maintenance)$ and $compatible(OnlineShop, Maintenance)$.

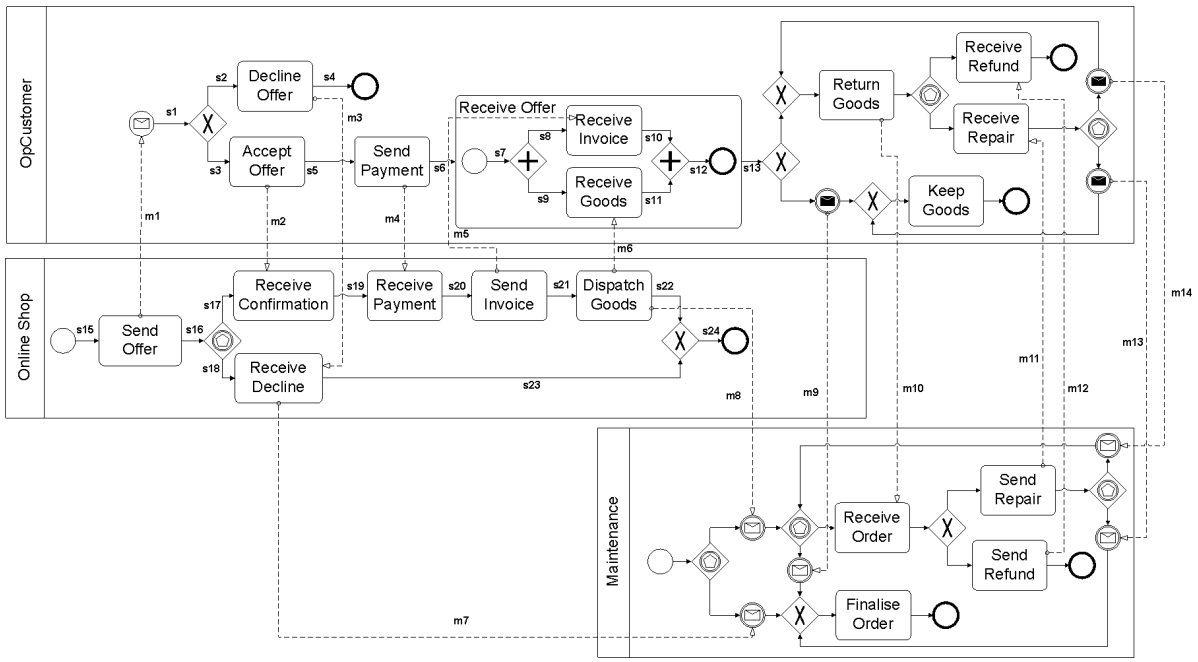


Figure 5.24: Adding the maintenance business process

5.10 Summary

In this chapter, we considered the behavioural semantics for BPMN. Specifically, from Sections 5.2 to 5.5, we presented a CSP semantics for the subset of BPMN specified in Chapter 4. In Section 5.6, we considered safety and liveness specification of BPMN processes via traces and failures refinements. In Section 5.7, we provided a CSP semantics for BPMN composition operations introduced in Chapter 4, and in Section 5.8, we considered how to apply compositional development approach to construct business processes using the CSP semantics of composition operations and BPMN. In Section 5.9, we studied Reed et al.'s theory of responsiveness for interoperating components in a complex system and applied it to develop a formal notion of *compatibility*. We were able to show compatibility between deadlock-free business processes ensures their collaboration's deadlock-freedom, and that adding a compatible deadlock-free business process to any complex deadlock-free business collaboration preserves the deadlock freedom of that collaboration.

A more detailed comparison of our approach with related work is provided in Chapter 9. Note that the CSP semantics of a BPMN process can be constructed automatically from a simple syntactic presentation of the diagram defined in Chapter 4. We do not expect the designer to write in this syntax directly, but to generate it from the diagrammatic notation.

Chapter 6

Modelling Relative Time

6.1 Introduction

In this chapter we consider timed behaviour in business processes, in particular, we provide an extension to the process semantics defined in Chapter 5 for modelling and reasoning about timed behaviour in BPMN. Specifically, we provide the following extensions to BPMN.

- We extend the semantics of timer events, such that each timer event waits for a duration of time when triggered.
- We introduce duration range into tasks, such that a task's execution time is chosen nondeterministically over a double-bounded range.

6.1.1 Running Example

As a running example we consider a production business process in a product line. This product line consists of four types of artifacts: $A1$, $A2$, $B1$ and $B2$. Each member of the product line is composed of one or more artifacts. A BPMN pool for this production business process is shown in Figure 6.1, where each of tasks $A1$, $A2$, $B1$ and $B2$ is responsible for assembling its respective artifact for a product. Each

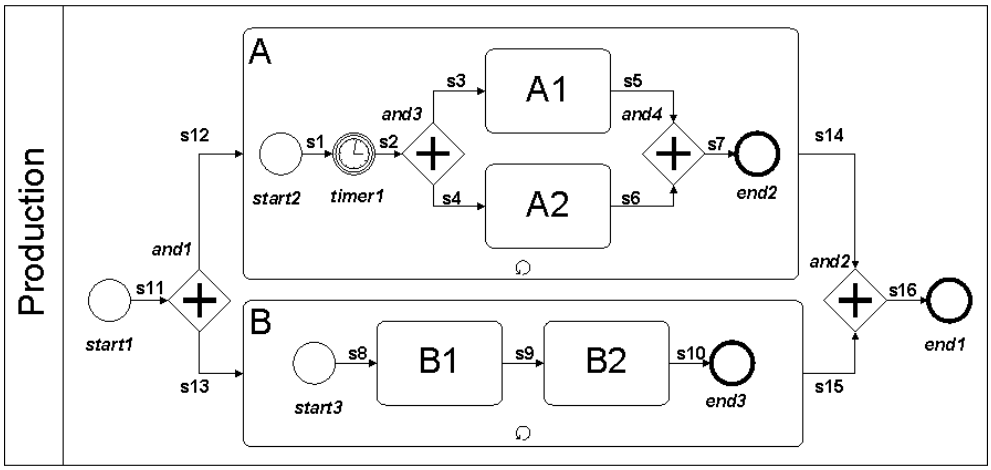


Figure 6.1: A production business process

task is allocated with timing information, specifying the minimum and the maximum duration between which the task is executed, and each timer event is specified with a duration to wait. By allocating different timing information to tasks and timer events as well as the number of iterations the multiple instance subprocesses A and B perform in the BPMN pool, we schedule the production business process for different products. Table 6.1 shows an example schedule of the production business process.

While a schedule is designed for a product, all valid schedules should satisfy the following requirement of the production business process.

At least one of artifacts $A1$ and $A2$ must be composed in between artifacts $B1$ and $B2$.

Element	**Iterations**
Subprocess A	1
Subprocess B	2
	Timing
Task $A1$	1 - 2 hours
Task $A2$	1 - 1.5 hours
Task $B1$	2 - 2.5 hours
Task $B2$	2 - 3 hours
Timer event	30 minutes

Table 6.1: An example schedule

We model this requirement with the following CSP process, where the work of tasks $A1$, $A2$, $B1$ and $B2$ are modelled by CSP events $w.A1$, $w.A2$, $w.B1$ and $w.B2$ respectively.

$$\begin{aligned} \mathit{Spec}0 &= (\sqcap x : \Sigma \setminus \{w.B1\} \bullet x \rightarrow \mathit{Spec}0) \sqcap w.B1 \rightarrow \mathit{Spec}1 \\ \mathit{Spec}1 &= w.A1 \rightarrow \mathit{Spec}2 \sqcap w.A2 \rightarrow \mathit{Spec}2 \sqcap (\sqcap x : \Sigma \setminus \{w.B2, w.A1, w.A2\} \bullet x \rightarrow \mathit{Spec}1) \\ \mathit{Spec}2 &= (\sqcap x : \Sigma \setminus \{w.B1, w.B2\} \bullet x \rightarrow \mathit{Spec}2) \sqcap w.B1 \rightarrow \mathit{Spec}1 \sqcap w.B2 \rightarrow \mathit{Spec}0 \end{aligned}$$

By extending the process semantics, we can model how timing information influences a business process's behaviour, and verify the production business process against behavioural properties such as process $\mathit{Spec}0$ above.

Notation. For the rest of this chapter we identify each task and subprocess in Figure 6.1 by its activity's name, and each of the other types of element by its label shown in *italic* font in the figure. For each task element A, the CSP event $w.A$ denotes A's work, and for any element e, we provide the following abbreviations to refer to the CSP events denoting e's incoming, outgoing and exception flows, where CSP event $s.f$ denotes sequence flow f.

$$\begin{aligned} \mathit{in}(e) &== \{f : (\mathit{atom}\ e).\mathit{in} \bullet s.f\} \\ \mathit{out}(e) &== \{f : (\mathit{atom}\ e).\mathit{out} \bullet s.f\} \\ \mathit{exit}(e) &== \{f : \mathrm{dom}(\mathit{atom}\ e).\mathit{exit} \bullet s.f\} \end{aligned}$$

We also define $\mathit{work}(e)$ such that if e is a task with name n, then $\mathit{work}(e)$ is a singleton set, containing the CSP event $w.n$, where $w.n$ denotes e's work. Otherwise $\mathit{work}(e)$ is the empty set.

$$\mathit{work}(e) = \begin{cases} \{w.(\mathit{task}^{\sim}(\mathit{atom}\ e).\mathit{type})\} & e \text{ is a task} \\ \emptyset & \text{otherwise} \end{cases}$$

Furthermore, we define $\mathit{min}(e)$ and $\mathit{max}(e)$ to refer to timed element e's minimum and maximum durations.

$$\mathit{min}(e) = \begin{cases} \mathit{stime}^{\sim}(\mathit{atom}\ e).\mathit{type} & e \text{ is a start timer event} \\ \mathit{itime}^{\sim}(\mathit{atom}\ e).\mathit{type} & e \text{ is an intermediate timer event} \\ \mathit{first}((\mathit{atom}\ e).\mathit{range}) & \text{otherwise} \end{cases}$$

$$\mathit{max}(e) = \begin{cases} \mathit{stime}^{\sim}(\mathit{atom}\ e).\mathit{type} & e \text{ is a start timer event} \\ \mathit{itime}^{\sim}(\mathit{atom}\ e).\mathit{type} & e \text{ is an intermediate timer event} \\ \mathit{second}((\mathit{atom}\ e).\mathit{range}) & \text{otherwise} \end{cases}$$

6.1.2 Contribution

In this chapter we extend the semantics defined in Chapter 5 to model relative time. This extension adopts the classical two-phase functioning approach to modelling real-time systems illustrated by languages such as Lustre [CPHP87] and Statecharts [Har87], and more recently to coordination models such as Linda [LJBB06]. We summarise this approach as follows [LJBB06]. In the first phase, elementary actions of statements are executed. They are assumed to be atomic in the sense that they take no time. Similarly, composition operators are assumed to be executed at no cost. In the second phase, when no actions can be reduced or when all the components encounter a special timed action, time progresses by one unit. We provide this extension in the form of transition rules. Similar to the process semantics defined in Chapter 5, this extension permits automatic verification via the FDR tool.

While our relative time extension permits the specification of real timing information about the behaviour of BPMN elements, at the level of CSP, our extension essentially uses this timing information to constrain the order of CSP events. As a result we focus on high level behavioural properties about the relative ordering of such events rather than timed properties.

6.1.3 Structure

The structure of this chapter is as follows. In Section 6.2 we give an overview of the relative timed extension and present preliminary definitions to assist the formal definition of the extension later in the chapter. We provide a formal definition of the extension in Sections 6.3, 6.4 and 6.5. Similar to Chapter 5, we have implemented the extension in Haskell, and for presentation purposes, some functions are partially presented in this chapter – their full definitions may be found in Appendix E. In Section 6.6 we analyse the relative timed extension. We summarise the contribution of this chapter in Section 6.7.

6.2 Preliminaries

6.2.1 Approach

In Chapter 5, we provided a CSP process semantics for BPMN. We modelled a BPMN pool as a parallel composition of CSP processes, each corresponding to an element directly contained in that pool. Throughout this chapter we refer to this parallel composition as the pool's enactment process. We extend the process semantics with relative time by composing the enactment process in parallel with a coordination process. This coordination process is a CSP process that coordinates the enactment process's timed behaviour.

In general, given a BPMN pool p, we define the following CSP process $T(p)$,

$$T(p) = PL(p) \,[\![\Sigma]\!]\, CP(p) \tag{6.1}$$

where $PL(p)$ is the enactment process of p and $CP(p)$ is the coordination of process of p. In this chapter we describe $CP(p)$ in terms of the labelled transition system (LTS) of the enactment process $PL(p)$.

A LTS of CSP process P is a set of nodes and for each event $e \in \alpha P \cup \{\tau\}$, a relation $\xrightarrow{e}$ between nodes. Specifically, a LTS of P is an edge-labelled directed graph where each edge represents an event being performed by P. Here we introduce some basic vocabulary for describing LTSs:

- $S \xrightarrow{e} T$ denotes $(S, T) \in (\xrightarrow{e})$;
- $S \xrightarrow{e}$ denotes there exists a state T such that $(S, T) \in (\xrightarrow{e})$;
- $S \rightarrow T$ denotes there exists some event e such that $S \xrightarrow{e} T$;
- $S \xrightarrow{\tau} T$ denotes an invisible, internal transition from S to T;
- $S \xRightarrow{t} T$ denotes $S \xrightarrow{e_0} .. \xrightarrow{e_n} T$ for some nonempty $t = \langle e_0 .. e_n \rangle$;
- $S \Longrightarrow T$ denotes there exists some nonempty sequence of events t such that $S \xRightarrow{t} T$.

We also consider states of BPMN pools: a state of BPMN pool p is a snapshot of its execution. Here each state of pool p corresponds to a node in the LTS of $PL(p)$.

Let $PL(p)$ be the enactment process of BPMN pool p, and A and B be p's states; we say A is reachable from B, if and only if there exists some trace $s \frown t \in traces(PL(p))$ such that $S_0 \stackrel{s}{\Longrightarrow} B \wedge B \stackrel{t}{\Longrightarrow} A$, where S_0 is an initial state of p. The coordination process $CP(p)$ restricts $PL(p)$'s behaviour such that the resulting process $T(p)$ only performs behaviour according to the definition of BPMN pool p and its timing specification. As a result pool p's timing specification restricts the set of p's reachable states.

We now introduce some terminologies to describe states in a BPMN pool. To model timed behaviour in a BPMN pool, we partition BPMN elements into timed and untimed:

Definition 6.1. ***Timed Element.*** *Timer events and atomic tasks are timed elements. A timed element takes a positive amount of time to execute. This implies that a BPMN process containing tasks and timer events also takes a positive amount of time to execute. All other types of elements are untimed.*

For example, in the BPMN pool shown in Figure 6.1, tasks $A1$, $A2$, $A3$ and $A4$ and the timer event are timed elements. At a reachable state of a BPMN pool, an element contained in the pool is either inactive or active.

Definition 6.2. ***Active.*** *A BPMN element is active if and only if one of its incoming sequence flows is triggered and not all of its necessary outgoing sequence flows are triggered. A start event is active as soon as the process directly containing it becomes active.*

For example, in the BPMN pool shown in Figure 6.1, task $A1$ is active after sequence flow $s3$ is triggered and before $s5$ is triggered. Similarly, the AND join gateway with incoming sequence flows $s5$ and $s6$ is active after one of $s5$ and $s6$ is triggered and before its outgoing sequence flow $s7$ is triggered.

Reachable states of a BPMN pool are defined in terms of the elements that the pool can execute. We partition the set of reachable states into untimed, time stable and timed states. We first give an overview of these various types of states; formal definitions are given from Section 6.3 onwards in the chapter. Specifically, at an untimed state, only active untimed elements can be executed, while in a timed state only active timed elements can be executed. A time stable state is a state between untimed and timed states where the set of timed elements for execution in the timed states are determined. For example, in the BPMN pool shown in Figure 6.1, the pool reaches a time stable state after triggering sequence flows $s1$ and $s8$, and at which point intermediate timer event $timer1$ and task $B1$ become active.

At a reachable state of a BPMN pool, an active element may be enactable. Here we give a definition of an enactable element.

Definition 6.3. ***Enactable.*** *An active untimed BPMN element is enactable at an untimed state if and only if all of its required incoming sequence flows are triggered, and not all of its required outgoing sequence flows are triggered. Similarly a timed element at a timed state is enactable if and only if it has zero minimum duration, all of its required incoming sequence flows are triggered and not all of its required outgoing sequence flows are triggered.*

$$\overbrace{S^0 \stackrel{s_0}{\Longrightarrow}}^{\text{X}} \overbrace{A^0 \stackrel{\tau}{\rightarrow}}^{\text{Y}} \overbrace{T^0 \stackrel{t_0}{\Longrightarrow}}^{\text{Z}} \overbrace{S^1 \stackrel{s_1}{\Longrightarrow}}^{\text{X}} \overbrace{A^1 \stackrel{\tau}{\rightarrow}}^{\text{Y}} \overbrace{T^1 \stackrel{t_1}{\Longrightarrow}}^{\text{Z}} .. \overbrace{S^{j-1} \stackrel{s_{j-1}}{\Longrightarrow}}^{\text{X}} \overbrace{A^{j-1} \stackrel{\tau}{\rightarrow}}^{\text{Y}} \overbrace{T^{j-1} \stackrel{t_{j-1}}{\Longrightarrow}}^{\text{Z}} \overbrace{S^j \stackrel{s_j}{\Longrightarrow}}^{\text{X}} C$$

Figure 6.2: An example relative timed execution

A relative timed execution of a BPMN pool p based on its timing information is a sequence of p's states that leads to completion. In CSP, this is a trace of process $PL(p)$ restricted by p's coordination process $CP(p)$. Figure 6.2 shows an example relative timed execution of a BPMN pool. Here S^0 is an initial state, and C is an ending state with no outgoing edge; states $A^0 .. A^{j-1}$ are time stable states. We observe $\frown/\langle s_0, t_0, s_1, t_1 .. s_{j-1}, t_{j-1}, s_j \rangle$ is a trace of $PL(p)$, where $\frown/\langle a_0, a_1 .. a_n \rangle$ concatenates the n sequences $a_0, a_1 .. a_n$. The behaviour of the coordination process $CP(p)$ can therefore be described as iterations over the following three steps.

1. All enactable untimed elements are executed until a time stable state of the BPMN pool is reached. Consider the example execution in Figure 6.2; the corresponding sequences of states representing this step are labelled X. We describe this step in Section 6.3.
2. At a time stable state, time progresses uniformly until at least either one active task's minimum duration is reached or one active timer event's duration is reached. Consider the example execution in Figure 6.2, the corresponding state transitions representing this step are labelled Y. Note that since no elements are executed, this step is modelled as τ transitions. We describe this step in Section 6.4.
3. All enactable timed elements are executed. Consider the example execution in Figure 6.2; the corresponding sequences of states representing this step are labelled Z. We describe this step in Section 6.5.

6.2.2 Syntactic Assumptions

In this chapter we provide a relative timed extension to the CSP semantics of BPMN pools. Here we describe some syntactic assumptions when applying this extension. These assumptions allow us to focus on the timing aspects of business process behaviour.

Start event Our relative time extension assumes each BPMN pools and subprocesses to have only one start event. Pools and subprocesses with multiple start events can be expressed using a combination of a start event, intermediate events and split/join gateways.

Sequence flow looping Our relative time extension considers only finite traces, we therefore do not sequential flow looping because in the untimed semantics it can generate infinite traces.

Parallel multiple instances Our relative time extension does not consider parallel multiple instances. Instead we model n parallel instances of activity T as the CSP process $|||\, i : \{1\,..\,n\} \bullet P(T)$, where $P(T)$ models an instance of T.

Message flows Our relative timed extension only considers single BPMN pools and as such we do not consider message flows, message events or event-based XOR gateways in this chapter.

Flow type In our relative timed extension, a multiple instance activity triggers its outgoing sequence flow after all of its instances have completed execution.

Error events In our relative timed extension, we only consider exception flows caused by intermediate timer events.

6.2.3 Implementation

We implement Haskell function `pTot` to take a BPMN pool and return a CSP process that models the relative timed behaviour of the pool.

```
pTot :: (PoolId,[Element]) -> Script
pTot (id,es) = Script d c (p++[coord(id,es),(tpterm id,[],tproc)]) e s
  where tproc = Parinter (ProcId (cterm id)) (SName "Events") (ProcId (plterm id))
        (Script d c p e s) = pool(id,es)
```

The returned CSP process is the partial interleaving of the enactment and coordination processes: the enactment process, which models the untimed CSP semantics of the pool, is defined by function `pool`, and the coordination process, which models the relative time extension, is defined by function `coord`. Function `coord` is defined as follows.

```
coord :: (PoolId,[Element]) -> ProcessDef
coord (id,ss) = ((cterm id),[],stable state)
  where state = (ss,[],(filter (isstart.etype.atom) ss),[],(filter (isstime.etype.atom) ss))
```

Here, `cterm id` identifies the coordination process, and the expression

```
stable (ss,[],(filter (isstart.etype.atom) ss),[],(filter (isstime.etype.atom) ss))
```

returns the coordination process by taking the set of start events directly contained in the pool as these elements become enactable immediately at the start of the pool's execution; the function `stable` implements the first step of the coordination procedure and is presented in the next section.

6.3 Coordinating Untimed States

6.3.1 Introduction

This section describes Step 1 of the coordination introducted on Page 103. This step executes all enactable untimed elements to reach a time stable state. A reachable state of a BPMN pool is time stable if and only if all its active elements are timed. Definitions 6.4 and 6.5 formally define untimed and time stable states respectively.

An initial state of a BPMN pool is then the tuple $(S, \emptyset, I, \emptyset, J)$ where $I = S \cap \{Start \bullet ele\}$ and $J = S \cap \{Start \mid (atom\ ele).type \in \mathrm{ran}\, stime \bullet ele\}$. For example, we consider the production business process shown in Figure 6.1. The only initial state of the BPMN pool is given by the tuple $(\{start1, and1, A, B, and2, end1\}, \emptyset, \{start1\}, \emptyset, \emptyset)$. Similarly, an ending state of a BPMN pool is the tuple $(S, \emptyset, \emptyset, \emptyset, \emptyset)$.

Definition 6.4. ***Untimed State.*** *An untimed state of a BPMN pool P is a five-tuple (S, M, E, W, T), where:*

- *S is the set of elements directly contained in P.*
- *$M : Element \nrightarrow \mathbb{N}$ is a partial function that takes a multiple instance subprocess and returns the number of remaining instances the subprocess can execute.*
- *E is the set of enactable untimed elements.*
- *W is the set of active untimed elements that are not enactable such that $W \cup E$ is the set of all active untimed elements and that $W \cap E = \emptyset$.*
- *T is the set of active timed elements; we write $Tm(X)$ to denote T of X.*

Definition 6.5. ***Time Stable State.*** *A time stable state is an untimed state such that E is empty.*

For example, we again consider the business process in Figure 6.1. One possible time stable state of the BPMN pool is given by the tuple $(\{start1, and1, A, B, and2, end1\}, \emptyset, \emptyset, \emptyset, \{B1, timer1\})$, at which point task $B1$ and intermediate timer event $timer1$ become enactable.

We define transition rules to model the coordination procedure at untimed states. We categorise transition rules according to the types of elements that enable the transitions. Table 6.2 lists the category prefixes and their corresponding descriptions. For example, transition rules, whose name begins with S, define transitions to perform an outgoing sequence flow of an AND split gateway with multiple outgoing sequence flows.

Category prefix	Transition description
S	perform an outgoing sequence flow of an AND split gateway with multiple outgoing sequence flows
U	perform an outgoing sequence flow of an AND split gateway with one outgoing sequence flow or any other types of untimed element
E	perform the completion event of an end event
M	perform an outgoing sequence flow of a multiple instance subprocess
L	move a multiple instance subprocess to its next iteration

Table 6.2: Categories of transition rules at untimed states

We first consider categories S, U and E in Section 6.3.2; these rules define the coordination of atomic elements at untimed states. We then consider categories M and L in Section 6.3.3; these rules define the coordination of compound elements at untimed states. We provide an overview of our Haskell implementation for this coordination step in Section 6.3.5.

6.3.2 Coordinating Atomic Elements

Figures 6.3, 6.4 and 6.5 show the transition rules that coordinate atomic elements at an untimed state. Specifically, they define transitions between any two untimed states due to the behaviour of atomic elements. We write $rm(e, s)$ to denote the BPMN element such that for BPMN element e and CSP event s, the following equation is satisfied:

$$atom(rm(e, s)) = (\mu\, RmSeqflow \mid \theta Atom = atom(e) \land s = s.sf? \bullet \theta Atom') \tag{6.2}$$

Here CSP event $s.sf?$ denotes sequence flow $sf?$, and $RmSeqflow$ is an operation schema on the $Atom$ part of an element that removes the sequence flow $sf?$ from that element.

$$\begin{aligned} RmSeqflow \mathrel{\widehat=} [&\Delta Atom;\ sf? : Seqflow \mid \\ &in' = in \setminus \{\, sf? \,\} \land out' = out \setminus \{\, sf? \,\} \land exit' = \{\, sf? \,\} \lhd exit \land \\ &receive' = receive \land send' = send \land type' = type] \end{aligned}$$

Given $r \in E$ is an AND split gateway, $\#out(r) > 1$ and $s \in out(r)$:

$$\frac{\begin{array}{l} e \in_{\mathrm{p}} S \text{ is untimed, either } s \in in(e) \text{ and if it is an AND join gateway then } e \in W \\ \text{and } \#in(e) = 1 \text{ or } c \in_{\mathrm{p}} S \text{ is a subprocess, } s \in in(c),\ e \text{ is a start event and } (e, c) \in contains \end{array}}{(S,M,E,W,T) \xrightarrow{s} (S,M,(E \setminus \{r\}) \cup \{rm(r,s), e\}, W \setminus \{e\}, T)} \quad [\text{ S-U }]$$

$$\frac{e \in_{\mathrm{p}} S \text{ or } e \in W \text{ is an AND join gateway, } s \in in(e) \text{ and } \#in(e) > 1}{(S,M,E,W,T) \xrightarrow{s} (S,M,(E \setminus \{r\}) \cup \{rm(r,s)\}, (W \setminus \{e\}) \cup \{rm(e,s)\}, T)} \quad [\text{ S-J }]$$

$$\frac{\begin{array}{l} c \in_{\mathrm{p}} S \text{ is a multiple instance subprocess, has } l \text{ instances and } s \in in(c) \\ e \text{ is an untimed start event and } (e, c) \in contains \end{array}}{(S,M,E,W,T) \xrightarrow{s} (S, M \cup \{(c, l-1)\}, (E \setminus \{r\}) \cup \{rm(r,s), e\}, W, T)} \quad [\text{ S-M }]$$

$$\frac{\begin{array}{l} c \in_{\mathrm{p}} S \text{ is a multiple instance subprocess, has } l \text{ instances and } s \in in(c) \\ e \text{ is an start timer event and } (e, c) \in contains \end{array}}{(S,M,E,W,T) \xrightarrow{s} (S, M \cup \{(c, l-1)\}, (E \setminus \{r\}) \cup \{rm(r,s)\}, W, T \cup \{e\})} \quad [\text{ S-M' }]$$

$$\frac{\begin{array}{l} e \in_{\mathrm{p}} S \text{ is timed, either } s \in in(e) \text{ or} \\ c \in_{\mathrm{p}} S \text{ is a subprocess, } s \in in(c),\ e \text{ is a start timer event and } (e, c) \in contains \end{array}}{(S,M,E,W,T) \xrightarrow{s} (S, M, (E \setminus \{r\}) \cup \{rm(r,s)\}, W, T \cup \{e\})} \quad [\text{ S-T }]$$

Figure 6.3: Transition rules (1)

We now consider rules shown in Figure 6.3. We first describe the rules' similarities before describing their differences. These rules define transitions that perform outgoing sequence flow s of some AND split gateway r with more than one outgoing sequence flow. After triggering s, r is still enactable as its other outgoing sequence flows still need to be triggered. As a result r is changed to $rm(r, s)$ in E to keep a record of r's remaining outgoing sequence flows to be triggered.

The differences between Rules S-U, S-J, S-M, S-M' and S-T shown in Figure 6.17 are as follows:

- Rule S-U defines transitions such that e is one of the following: an AND join gateway with one incoming sequence flow s; a start event in a subprocess c with an incoming sequence flow s; and

an untimed element that is not an AND join gateway and has s as one of its incoming sequence flows. For these cases, e becomes enactable after s is triggered, therefore e is added to E.

- Rule S-J defines transitions such that e is an AND join gateway that has more than one incoming sequence flow and where s is one of its incoming sequence flows. In this case e is active but not enactable since its other incoming sequence flows have not been triggered, therefore $rm(e, s)$ is replaced with r in W; note that r needs not be in W initially.
- Rule S-M defines transitions such that e is a start event directly contained in some multiple instance subprocess c, and that c has incoming sequence flow s and specifies l instances. In this case e becomes enactable and is added to E. To keep track of the number of c's remaining instances, the pair $(c, l-1)$ is added to M.
- Rule S-M' defines transitions such that e is a start timer event directly contained in some multiple instance subprocess c, and that c has incoming sequence flow s and specifies l instances. In this case e cannot be executed in an untimed state and therefore is added to T. To keep track of the number of c's remaining instances, the pair $(c, l-1)$ is added to M.
- Rule S-T defines transitions such that e is a timed element and has incoming sequence flow s. In this case e cannot be executed in an untimed state and therefore is added to T.

Given $r \in E$ is untimed, $s \in out(r)$ and if it is an AND split gateway, $\#in(e) = 1$:

$$\frac{\begin{array}{l}e \in_{\mathrm{p}} S \text{ is untimed, either } s \in in(e) \text{ and if it is an AND join gateway then } e \in W \\ \text{and } \#in(e) = 1 \text{ or } c \in_{\mathrm{p}} S \text{ is a subprocess, } s \in in(c),\ e \text{ is a start event and } (e, c) \in \mathit{contains}\end{array}}{(S,M,E,W,T) \xrightarrow{s} (S,M,(E\backslash\{r\})\cup\{e\},W\backslash\{e\},T)} \quad [\text{ U-U }]$$

$$\frac{e \in_{\mathrm{p}} S \text{ or } e \in W \text{ is an AND join gateway, } s \in in(e) \text{ and } \#in(e) > 1}{(S,M,E,W,T) \xrightarrow{s} (S,M,E\backslash\{r\},(W\backslash\{e\})\cup\{rm(e,s)\},T)} \quad [\text{ U-J }]$$

$$\frac{\begin{array}{l}c \in_{\mathrm{p}} S \text{ is a multiple instance subprocess, has } l \text{ instances and } s \in in(c) \\ e \text{ is a start event and } (e, c) \in \mathit{contains}\end{array}}{(S,M,E,W,T) \xrightarrow{s} (S,M\cup\{(c,l-1)\},(E\backslash\{r\})\cup\{e\},W,T)} \quad [\text{ U-M }]$$

$$\frac{\begin{array}{l}c \in_{\mathrm{p}} S \text{ is a multiple instance subprocess, has } l \text{ instances and } s \in in(c) \\ e \text{ is an start timer event and } (e, c) \in \mathit{contains}\end{array}}{(S,M,E,W,T) \xrightarrow{s} (S,M\cup\{(c,l-1)\},E\backslash\{r\},W,T\cup\{e\})} \quad [\text{ U-M' }]$$

$$\frac{\begin{array}{l}e \in_{\mathrm{p}} S \text{ is timed, either } s \in in(e) \text{ or} \\ c \in_{\mathrm{p}} S \text{ is a subprocess, } s \in in(c),\ e \text{ is a start timer event and } (e, c) \in \mathit{contains}\end{array}}{(S,M,E,W,T) \xrightarrow{s} (S,M,E\backslash\{r\},W,T\cup\{e\})} \quad [\text{ U-T }]$$

Figure 6.4: Transition rules (2)

We now consider rules shown in Figure 6.4. These rules define transitions that perform outgoing sequence flow s of untimed element r such that if it is an AND split gateway then it has only one outgoing sequence flow. After triggering s, r is not longer enactable as it completes its execution,

therefore r is removed from E. The differences between Rules U-U, U-J, U-M, U-M' and U-T shown in Figure 6.4 correspond to those between Rules S-U, S-J, S-M, S-M' and S-T shown in Figure 6.3.

We now consider rules shown in Figure 6.5. These rules define transitions that trigger completion event $c.e$ of some end event e. Rule E-C defines transitions such that e is an end event directly contained in the BPMN pool. In this case, no other element is active, otherwise the pool is deadlocked, and after $c.e$ is performed, e has completed its execution and is removed from E; this then leads to an ending state. Rule E-M defines transitions such that e is an end event directly contained in some compound element c. In this case, e has completed execution and is removed from E, and c becomes enactable and is added to E.

Table 6.3 shows some examples of applying the transition rules shown in Figures 6.3, 6.4 and 6.5 to BPMN processes shown in Figure 6.6.

$$\frac{c.e \text{ is a completion event for end event } e \in E \text{ and } e \in S}{(S,\emptyset,\{e\},\emptyset,\emptyset) \xrightarrow{c.e} (S,\emptyset,\emptyset,\emptyset,\emptyset)} \quad [\text{ E-C }]$$

$$\frac{\begin{array}{l} c.e \text{ is a completion event for end event } e \in E \text{ such that } e \in_{\mathrm{p}} S, \text{ and } (e,c) \in \textit{contains} \text{ for compound} \\ \text{element } c \in_{\mathrm{p}} S \text{ where } e \text{ is the only element in } E \text{ that is contained in } c \text{ and no element in } W \cup T \\ \text{is contained in } c \end{array}}{(S,M,E,W,T) \xrightarrow{c.e} (S,M,(E \setminus \{e\}) \cup \{c\},W,T)} \quad [\text{ E-M }]$$

Figure 6.5: Transition rules (3)

Figure	Rule	Transition
6.6(1)	$U-J$	$(S, M, E, W, T) \xrightarrow{s.f} (S, M, E \setminus \{r\}, (W \setminus \{e\}) \cup \{rm(e,s)\}, T)$
6.6(2)	$U-M$	$(S, M, E, W, T) \xrightarrow{s.f} (S, M \cup \{(c, l_c)\}, (E \setminus \{r\}) \cup \{e\}, W, T)$
6.6(3)	$U-T$	$(S, M, E, W, T) \xrightarrow{s.f} (S, M, E \setminus \{r\}, W, T \cup \{e\})$
6.6(4)	$A-T$	$(S, M, E, W, T) \xrightarrow{s.f} (S, M, (E \setminus \{r\}) \cup \{rm(r,s)\}, W, T \cup \{e\})$
6.6(5)	$E-M$	$(S, M, E, W, T) \xrightarrow{c.e} (S, M, (E \setminus \{e\}) \cup \{c\}, W, T)$

Table 6.3: Applications of transition rules

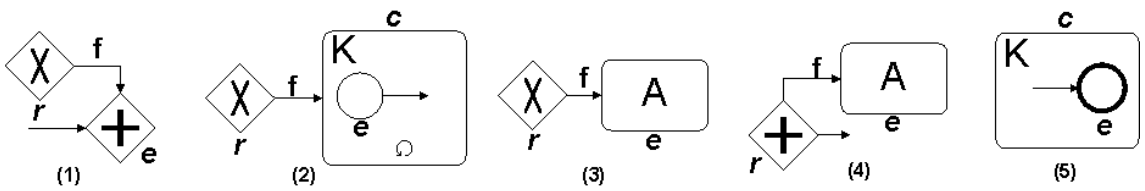


Figure 6.6: Illustrations of coordinating atomic elements

6.3.3 Coordinating Compound Elements

Figures 6.7 and 6.8 define the transition rules to coordinate compound elements at an untimed state. Specifically, these rules define transitions that are made available after an end event contained in a compound element has performed its completion event, that is, after a transition defined by Rule E-M.

Given $s \in out(m)$ for some compound element $m \in E$, where $m \in_{\mathrm{p}} S$ is either a subprocess or a multiple instance subprocess such that $M(m) = l$ and $l = 0$, let $ex(m, X) = \{x : X \mid out(x) \subseteq exit(m)\}$:

$$\frac{\begin{array}{l}e \in_{\mathrm{p}} S \text{ is untimed, either } s \in in(e) \text{ and if it is an AND join gateway then } e \in W \\ \text{and } \#in(e) = 1 \text{ or } c \in_{\mathrm{p}} S \text{ is a subprocess, } s \in in(c),\ e \text{ is a start event and } (e, c) \in contains\end{array}}{(S,M,E,W,T) \xrightarrow{s} (S, M \setminus \{(m,l)\}, (E \setminus \{m\}) \cup \{e\}, W \cup \{e\}, T \setminus ex(m,T))} \quad [\text{ M-U }]$$

$$\frac{e \in_{\mathrm{p}} S \text{ or } e \in W \text{ is an AND join gateway, } s \in in(e) \text{ and } \#in(e) > 1}{(S,M,E,W,T) \xrightarrow{s} (S, M \setminus \{(m,l)\}, E \setminus \{m\}, (W \setminus \{e\}) \cup \{rm(e,s)\}, T \setminus ex(m,T))} \quad [\text{ M-J }]$$

$$\frac{\begin{array}{l}c \in_{\mathrm{p}} S \text{ is a multiple instance subprocess, has } l_c \text{ instances and } s \in in(c) \\ e \text{ is an untimed start event and } (e, c) \in contains\end{array}}{(S,M,E,W,T) \xrightarrow{s} (S, (M \setminus \{(m,l)\}) \cup \{(c, l_c - 1)\}, (E \setminus \{m\}) \cup \{e\}, W, T \setminus ex(m,T))} \quad [\text{ M-M }]$$

$$\frac{\begin{array}{l}c \in_{\mathrm{p}} S \text{ is a multiple instance subprocess, has } l_c \text{ instances and } s \in in(c) \\ e \text{ is an start timer event and } (e, c) \in contains\end{array}}{(S,M,E,W,T) \xrightarrow{s} (S, (M \setminus \{(m,l)\}) \cup \{(c, l_c - 1)\}, E \setminus \{m\}, W, (T \setminus ex(m,T)) \cup \{e\})} \quad [\text{ M-M' }]$$

$$\frac{\begin{array}{l}e \in_{\mathrm{p}} S \text{ is timed, either } s \in in(e) \text{ or} \\ c \in_{\mathrm{p}} S \text{ is a subprocess, } s \in in(c),\ e \text{ is a start timer event and } (e, c) \in contains\end{array}}{(S,M,E,W,T) \xrightarrow{s} (S, M \setminus \{(m,l)\}, E \setminus \{m\}, W, (T \setminus ex(m,T)) \cup \{e\})} \quad [\text{ M-T }]$$

Figure 6.7: Transition rules (4)

Given multiple instance subprocess $m \in E$ such that $M(m) = l$ and $l > 0$:

$$\frac{e \text{ is an untimed start event such that } (e, m) \in contains}{(S,M,E,W,T) \xrightarrow{\tau} (S, (M \oplus \{m \mapsto l-1\}, (E \setminus \{m\}) \cup \{e\}, W, T)} \quad [\text{ L-U }]$$

$$\frac{e \in_{\mathrm{p}} S \text{ is a start timer event such that } (e, m) \in contains}{(S,M,E,W,T) \xrightarrow{\tau} (S, (M \oplus \{m \mapsto l-1\}, E \setminus \{m\}, W, T \cup \{e\})} \quad [\text{ L-T }]$$

Figure 6.8: Transition rules (5)

The rules shown in Figure 6.7 define transitions that perform outgoing sequence flow s of some compound element $m \in E$, where m is either a subprocess or a multiple instance subprocess specifying l instances but has no remaining instance to be executed. For these rules, m is removed from E, and the pair (m, l) is removed from M. These rules also remove the set $ex(m, T)$ of intermediate events that specifies timed exception flows of m from the set of active timed elements T, where $ex(m, X)$ is the set $\{x : X \mid out(x) \subseteq exit(m)\}$. Detailed presentations of specifying and coordinating timed exception flows are given in Sections 6.4 and 6.5. The differences between Rules M-U, M-J, M-M, M-M' and M-T

$$
\begin{array}{rcll}
(ES, \emptyset, \{start1\}, \emptyset, \emptyset) & \stackrel{s.s11}{\rightarrow} & (ES, \emptyset, \{and1\}, \emptyset, \emptyset) & \text{U-U} \\
(ES, \emptyset, \{and1\}, \emptyset, \emptyset) & \stackrel{s.s12}{\rightarrow} & (ES, \{(A, i_A - 1)\}, \{and1', start2\}, \emptyset, \emptyset) & \text{S-M} \\
(ES, \emptyset, \{and1\}, \emptyset, \emptyset) & \stackrel{s.s13}{\rightarrow} & (ES, \{(B, i_B - 1)\}, \{and1'', start3\}, \emptyset, \emptyset) & \text{S-M} \\
(ES, \{(A, i_A - 1)\}, \{and1', start2\}, \emptyset, \emptyset) & \stackrel{s.s13}{\rightarrow} & (ES, MS, \{start2, start3\}, \emptyset, \emptyset) & \text{S-M} \\
(ES, \{(A, i_A - 1)\}, \{and1', start2\}, \emptyset, \emptyset) & \stackrel{s.s1}{\rightarrow} & (ES, \{(A, i_A - 1)\}, \{and1'\}, \emptyset, \{timer1\}) & \text{U-T} \\
(ES, \{(B, i_B - 1)\}, \{and1'', start3\}, \emptyset, \emptyset) & \stackrel{s.s12}{\rightarrow} & (ES, MS, \{start2, start3\}, \emptyset, \emptyset) & \text{S-M} \\
(ES, \{(B, i_B - 1)\}, \{and1'', start3\}, \emptyset, \emptyset) & \stackrel{s.s8}{\rightarrow} & (ES, \{(B, i_B - 1)\}, \{and1''\}, \emptyset, \{B1\}) & \text{U-T} \\
(ES, MS, \{start2, start3\}, \emptyset, \emptyset) & \stackrel{s.s1}{\rightarrow} & (ES, MS, \{start3\}, \emptyset, \{timer1\}) & \text{U-T} \\
(ES, MS, \{start2, start3\}, \emptyset, \emptyset) & \stackrel{s.s8}{\rightarrow} & (ES, MS, \{start2\}, \emptyset, \{B1\}) & \text{U-T} \\
(ES, \{(A, i_A - 1)\}, \{and1'\}, \emptyset, \{timer1\}) & \stackrel{s.s13}{\rightarrow} & (ES, MS, \{start3\}, \emptyset, \{timer1\}) & \text{U-M} \\
(ES, \{(B, i_B - 1)\}, \{and1''\}, \emptyset, \{B1\}) & \stackrel{s.s12}{\rightarrow} & (ES, MS, \{start2\}, \emptyset, \{B1\}) & \text{U-M} \\
(ES, MS, \{start2\}, \emptyset, \{B1\}) & \stackrel{s.s1}{\rightarrow} & (ES, MS, \emptyset, \emptyset, \{timer1, B1\}) & \text{U-T} \\
(ES, MS, \{start3\}, \emptyset, \{timer1\}) & \stackrel{s.s8}{\rightarrow} & (ES, MS, \emptyset, \emptyset, \{timer1, B1\}) & \text{U-T}
\end{array}
$$

Figure 6.9: Coordinating untimed states

correspond to those between Rules S-U, S-J, S-M, S-M' and S-T shown in Figure 6.3.

The rules L-U and L-T shown in Figure 6.8 define transitions that trigger a new instance of some multiple instance subprocess m such that m directly contains some start event e. Since e has no incoming sequence flow, this behaviour is modelled by the τ transition. Specifically, in Rules L-U and L-T, m is removed from E, while the pair (m, l) in M is replaced by $(m, l-1)$ to model the behaviour of triggering of a new instance of m. For Rule L-U, e is untimed, therefore e becomes enactable and is added to E, while for Rule L-T, e is timed and is only enactable in a timed state, therefore it is added to T.

6.3.4 Example

For example, Figure 6.9 shows the coordination of the BPMN pool shown in Figure 6.1 from its initial state to its first time stable state according to transition rules defined so far, where multiple instance subprocess A has i_A instances and B has i_B instances; element $and1' = rm(and1, s.s12)$ is the AND split gateway $and1$ with sequence flow $s12$ removed, and element $and1'' = rm(and1, s.s13)$ is $and1$ with $s13$ removed; set $ES = \{start1, and1, A, B, and2, end1\}$ is the set of elements directly contained in the BPMN pool; and set $MS = \{(A, i_A - 1), (B, i_B - 1)\}$ is the set of pairs recording the number of remaining instances to be triggered for subprocesses A and B.

This coordination can then be expressed as the CSP process $U0$ of the process definitions in Equation 6.3. Specifically, $U0$ defines the coordination process of the BPMN pool from its initial state to its first time stable state, where process $T0$ coordinates the behaviour of the BPMN pool at the time stable state. The coordination of time stable states and the definition of $T0$ are given in Section 6.4.

$$
\begin{array}{lll}
U0 = s.s11 \rightarrow U1 & U3 = s.s12 \rightarrow U4 \;\Box\; s.s8 \rightarrow U6 & U6 = s.s12 \rightarrow U8 \\
U1 = s.s12 \rightarrow U2 \;\Box\; s.s13 \rightarrow U3 & U4 = s.s1 \rightarrow U7 \;\Box\; s.s8 \rightarrow U8 & U7 = s.s8 \rightarrow T0 \\
U2 = s.s13 \rightarrow U4 \;\Box\; s.s1 \rightarrow U5 & U5 = s.s13 \rightarrow U7 & U8 = s.s1 \rightarrow T0
\end{array} \tag{6.3}
$$

6.3.5 Implementation

We provide function `stable` to implement the coordination at untimed states; it is defined as follows,

```
stable :: UntimedState -> Process
stable (ss,[],[],[],[]) = Skip
stable (ss,ms,[],ae,te) = timer ss ms ae te
stable (ss,ms,ue,ae,te) = exts [ branch (ss,ms,ue,ae,te) e | e <- ue ]
```

where we model an untimed state using the following type synonym.

```
type UntimedState = ([Element], [(Element,Int)], [Element], [Element], [Element])
```

Specifically, function `stable` takes a tuple `(ss,ms,ae,ue,te)` recording an untimed state (S, M, E, W, T): `ss` models component S, `ms` models M, `ue` models E, `ae` models W and `te` models T.

Function `stable` implements transition rules by evaluating the following clauses. The first clause matches an ending state and returns the CSP process *Skip*. The second clause matches a time stable state and evaluates the expression `timer ss ms ae te`; function `timer` implements the coordination step at a time stable state, which is presented in Section 6.4. The third clause matches any other untimed states. At this point, the coordination is at an untimed state with enactable elements, and `stable` returns a CSP process that defines the external choice over all possible transitions. Here each transition is implemented by the function `branch`; the full definition of `branch` as well as other functions used to implement the transition rules defined in this section can be found in Section E.2.

6.4 Calculating Time Progression

This section describes Step 2 of the coordination introduced on Page 103. The step calculates the minimum amount of time that needs to progress at a time stable state before either an active task's minimum duration or an active timer event's duration is lapsed. We first use an example to describe our approach to calculate time progression. From the example we provide a general structure to the rest of this section.

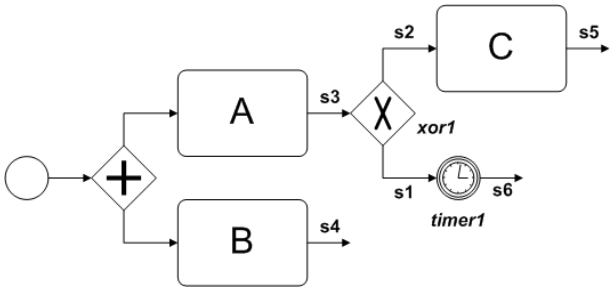


Figure 6.10: Illustration of postponement and delay

We use the partial BPMN process in Figure 6.10 as an example to describe informally our approach to calculating time progression. We write $d(w) = [a, b]$ to denote the duration of task w ranging between a and b, where $a \leq b$, and $t(m)$ to denote the duration of timer event m. Specifically, let tasks $d(A) = [i, j]$ and $d(B) = [k, l]$ such that $i < j < k$. At the first time stable state, the set of active timed elements is $T = \{A, B\}$; we label this time stable state X in this example. Since task A has the shortest minimum duration i, time progresses for duration i. Time progression is denoted by an internal transition $X \xrightarrow{\tau} R$, where R is the initial timed state. At R, $d(A) = [0, j - i]$ and $d(B) = [k - i, l - i]$, therefore A may be executed, while B, with a non-zero minimum duration, is postponed until the next time stable state.

At timed state R, A may be either executed or delayed until the next time stable state. We first consider the case in which A is executed and sequence flow $s3$ is triggered. In this case the coordination then reaches an untimed state, where gateway $xor1$ is enactable, and therefore either flow $s2$ or $s1$ can be triggered. If $s2$ is triggered, another time stable state is reached, where $T = \{B, C\}$.

Conversely, if A's execution is delayed, no untimed element can be executed, therefore the coordination reaches the next time state, where $T = \{A, B\}$ and $d(A) = [0, j - i]$ and $d(B) = [k - i, l - i]$; we label this time stable state Y in this example. Since $j < k$, time progresses $j - i$ at Y; this is denoted by the internal transition $Y \xrightarrow{\tau} U$ to timed state U. At U, $d(A) = [0, 0]$ and $d(B) = [k - j, l - j]$, therefore A is executed, since B has a non-zero minimum duration, its execution is postponed until the next time stable state.

In general, in order to calculate time progression the following two pieces of information are required.

1. The possible exception flows caused by intermediate timer events attached to active tasks and subprocesses.

2. The possible activity instances that can be triggered by active multiple instance tasks and subprocesses.

We describe how to obtain these two pieces of information in Sections 6.4.2 and 6.4.3 respectively. We define the procedure to calculate time progression in Section 6.4.4, and we give an overview of our Haskell implementation of this coordination step in Section 6.4.5. We first provide some preliminary definitions in Section 6.4.1 to assist defining time progression calculation.

6.4.1 Preliminaries

We consider two adjacent time stable states $X \Longrightarrow Y$ and provide the following definitions about timed elements at time stable state Y.

Definition 6.6. ***Postponement.*** *A timed element $t \in Tm(Y)$ is a postponed element at time stable state Y if and only if $t \in Tm(X)$ and $d(t) = [i, j]$ where $i > 0$. The set of postponed elements at Y is denoted by $Pp(Y) \subseteq Tm(Y)$.*

Definition 6.7. ***Delay.*** *A timed element $t \in Tm(Y)$ is a delayed element at time stable state Y if and only if $t \in Tm(X)$ and $d(t) = [0, j]$. The set of delayed elements at Y is denoted by $De(Y) \subseteq Tm(Y)$.*

Definition 6.8. ***Freshness.*** *A timed element $t \in Tm(Y)$ is fresh at time stable state Y if and only if $t \notin Tm(X)$. The set of fresh timed elements at Y is denoted by $F(Y)$ such that $Fr(Y) = Tm(Y) \setminus (Pp(Y) \cup De(Y))$.*

As described in the example above, time progression at some time stable state X is denoted as a single internal transition τ such that $X \xrightarrow{\tau} T$, where T is a timed state at which enactable timed elements can be executed. We provide a formal definition of timed states. Transition rules between timed states are defined in Section 6.5.

Definition 6.9. ***Timed State.*** *A timed state of a BPMN pool P is a seven-tuple (S, M, E, W, P, C, O), where:*

- *S is the set of elements directly contained in P.*
- *M : Element $\nrightarrow \mathbb{N}$ is a partial function that takes a multiple instance subprocess and returns the number of remaining instances the subprocess can execute.*
- *E is the set of enactable untimed elements; elements in this set will not be executed until P's coordination has reached an untimed state.*
- *W is the set of active untimed elements that are not enactable.*
- *P is the set of postponed elements and active multiple instance task elements.*
- *C is the set of enactable timed elements.*
- *O is the set of task and multiple instance task elements whose outgoing sequence flows can be triggered.*

The (initial) timed state that follows immediately from a time stable state is then characterised by the tuple $(S, M, \emptyset, W, P, C, \emptyset)$.

Definition 6.10. ***Initial Timed State.*** *An initial timed state is a timed state (S, M, E, W, P, C, O) such that E and O are empty.*

While components S, M and W of an initial timed state are defined by its immediate preceding time stable state $(S, M, \emptyset, W, T)$, by calculating time progression, we determine components P and C.

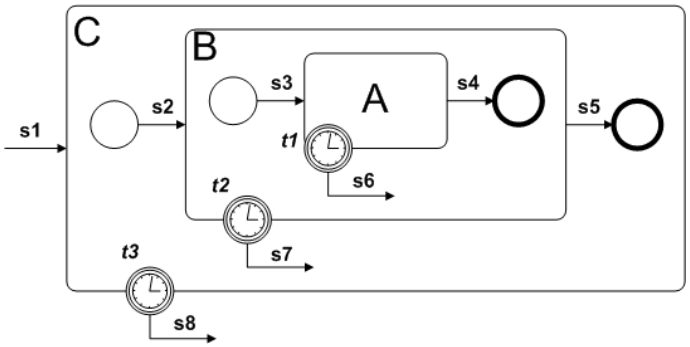


Figure 6.11: Timed exception associations

6.4.2 Timed Exception Association

In this section we describe the procedure to calculate the associated timed exception flows of a timed element at a time stable state. Associated time exception flows were first introduced in Section 6.3.3, where the transition rules for coordinating an outgoing sequence flow of a compound element are defined.

For example, consider subprocess C in Figure 6.11, which contains task A. If $A \in Tm(X)$ is an active timed element at some time stable state X where $d(A) = [i, j]$, and if the duration of intermediate timer event $t2$ is at most i, that is, $t(t2) \leq i$, then $t2$ may interrupt A's execution and trigger exception flow $s7$. The same applies to events $t1$ and $t3$. As a result the calculation of time progression must consider these exception flows. We now provide a definition of exception flow associations.

Definition 6.11. ***Timed exception association.*** *An associated timed exception flow of an element e is a pair (f, t) such that either $(f, t) \in (atom\ e).exit$ and e is an activity element, or $(f, t) \in (atom\ c).exit$ for some compound element c such that $(e, c) \in contains^+$. Formally the set of e's associated timed exception flows is defined as follows:*

$$assoc(e) = (atom\ e).exit \cup \bigcup\{c : Element \mid (e, c) \in contains^+ \bullet (atom\ c).exit\}$$

We record timed exception associations from the set of elements in a time stable state in the following way: for each timed exception flow (f, t), we construct an intermediate timer event that has type t and an outgoing sequence flow f. We write $exp(f, t)$ to denote this element; an illustration of this is shown in Figure 6.12.

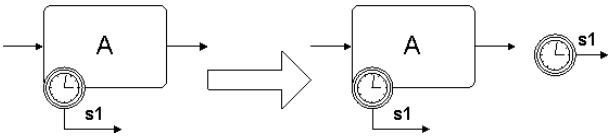


Figure 6.12: Specifying a timed exception association

6.4.3 Multiple Instance

In this section we consider the calculation of multiple instance tasks. For each active multiple instance task $m \in Tm(X)$ in time stable state X, we construct a task element t to model a single instance of m, and a resulting multiple instance n to model m with the number of available instances decreased by one. We write $splitsq(m)$ to denote the pair (n, t). Figure 6.13 shows an illustration of splitting a multiple instance task element.

We record information about multiple instance tasks in some time stable state X in the following way: given a pair $(n, t) = splitsq(m)$ of multiple instance task m at X, we record $n \in Pp(X)$ as a

postponed element at X. We also partition the set of active multiple instance tasks in a time stable state into the following four sets:

1. Set of fresh active multiple instance tasks $M_f(X) \subseteq Fr(X)$.
2. Set of active multiple instance tasks $M_p(X) \subseteq Pp(X)$, whose instances have been postponed, that is, for all $n \in M_p(X)$ where $(n, t) = splitsq(m)$ for some multiple instance task m, we have $t \in Pp(X)$.
3. Set of active multiple instance tasks $M_d(X) \subseteq Pp(X)$, whose instances have been delayed, that is, for all $n \in M_d(X)$ where $(n, t) = splitsq(m)$ for some multiple instance task m, we have $t \in De(X)$.
4. Set of active multiple instance tasks $M_n(X) \subseteq Pp(X)$, whose instances have been neither delayed nor postponed, that is, for all $n \in M_n(X)$, where $(n, t) = splitsq(m)$ for some multiple instance task m, we have $t \notin Pp(X) \cup De(X)$.

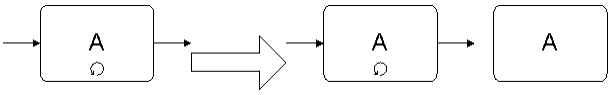


Figure 6.13: An illustration of splitting a multiple instance task

6.4.4 Minimal Time Progression

Transition rule A-E shown in Figure 6.14 defines transitions that progress time from a time stable state to a timed state. Functions *dmin* and *dmax* in the rule are defined as follows: given some element e and duration d, $dmin(e, d)$ denotes the element derived from decreasing duration d from $min(e)$ and $dmax(e, d)$ denotes the element derived from decreasing d from $max(e)$. Note that the minimum and maximum durations of a timer event are the same, such that $dmax(e, d)$ becomes the identity function if e is a timer event. We now provide the following textual description of the transition rule where X is a time stable state.

1. Set $Tm(X)$ of timed elements is partitioned into sets of delayed $De(X)$, postponed $Pp(X)$ and fresh $Fr(X)$ elements as defined in Section 6.4.1.
2. Set Nr is the union of the set of fresh elements $Fr(X)$ and the set of timer events modelling associated timed exception flows of elements in $Fr(X)$ such that timer events in this set are not postponed elements. The construction of timer events recording associated timed exception flows is defined in Section 6.4.2.
3. Set Er is composed from the set Nr without elements in the set $M_f(X)$ of fresh multiple instance tasks, and the set of task instances of multiple instance task elements from sets $M_f(X)$ and $M_n(X)$; the specification of task instances is defined in Section 6.4.3. Set Er contains all active fresh timed elements at X.
4. Set Os is the set of postponed elements that are not multiple instance task. Specifically, set Os contains active postponed timed elements at X.
5. The minimum time progression $dur(X)$ is defined as follows.

   $$dur(X) = minimum(\{e : Os \cup Er \bullet min(e)\} \cup \{e : De(X) \bullet max(e)\})$$

   Here $min(e)$ and $max(e)$ return the minimum and maximum duration of timed element e. Specifically, $dur(X)$ is the smallest value from the set constructed from the minimum duration of fresh Nr and postponed Os elements, and the maximum duration of delayed De elements. $dur(X)$ is the least amount of time that needs to progress before either one active task's minimum duration is reached or one active timer event's duration is reached.

Let X be the time stable state $(S, M, \emptyset, W, T)$:

$$\frac{\begin{array}{l}\langle De(X), Pp(X), Fr(X)\rangle \text{ partition } T \\ Nr = Fr(X) \cup (\bigcup\{e : Fr(X) \bullet exp(\!| \, assoc(e) \, |\!)\} \setminus Pp(X)) \\ Er = (Nr \setminus M_f(X)) \cup \{m : M_f(X) \cup M_n(X) \bullet second(splitsq(m))\} \\ Os = Pp(X) \setminus (M_n(X) \cup M_p(X) \cup M_d(X)) \\ dur(X) = minimum(\{e : Os \cup Er \bullet min(e)\} \cup \{e : De(X) \bullet max(e)\}) \\ As = \{e : Os \cup Er \bullet dmax(dmin(e, dur(X)), dur(X))\} \cup \{e : De(X) \bullet dmax(e, dur(X))\} \\ Ac = \{e : As \mid min(e) > 0\}\end{array}}{(S,M,\emptyset,W,T) \xrightarrow{\tau} (S,M,\emptyset,W,Ac \cup M_p(X) \cup M_d(X) \cup \{m : M_n(X) \cup M_f(X) \bullet first(splitsq(m))\}, As \setminus Ac, \emptyset)} \quad [\text{ A-E }]$$

Figure 6.14: Transition rule (6)

6. Set As contains timed elements derived decreasing duration $dur(X)$ from elements in sets Nr, De and Os. This models minimum time progression at X.

7. Set Ac is a subset of As such that each element $e \in Ac$ has a non-zero positive minimum duration $min(e) > 0$. This set contains elements that are to be postponed until the next time stable state as their minimum durations have not been reached.

Transition rule A-E shown in Figure 6.14 therefore specifies the following transition from time stable state X to the subsequent initial timed state,

$$(S, M, \emptyset, W, T) \xrightarrow{\tau} (S, M, \emptyset, W, P, As \setminus Ac, \emptyset)$$

where set $As \setminus Ac$ is the set of timed elements with zero minimum duration, and set P is the set of postponed elements and active multiple instance tasks, and is defined as follows.

$$P = Ac \cup M_p(X) \cup M_d(X) \cup \{m : M_n(X) \cup M_f(X) \bullet first(splitsq(m))\}$$

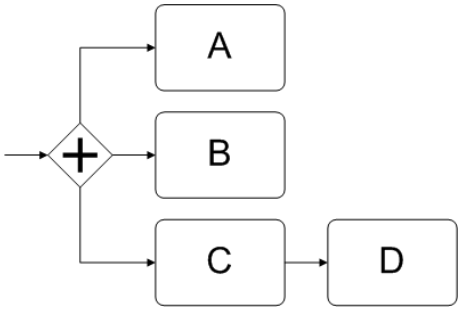


Figure 6.15: A BPMN subprocess element illustrating time progression

For example, we consider the partial BPMN process in Figure 6.15. We assume there exists a time stable state S such that the set of fresh timed elements are tasks A, B and C where $d(A) = [i, j]$, $d(B) = [k, l]$ and $d(C) = [m, n]$ and $m < i$ and $m < k$. By Rule A-E, $dur(S) = m$, hence both A and B are postponed for duration m, and during this duration, task C may either be executed or delayed nondeterministically.

If task C is delayed, then in the subsequent time stable T, where $S \Longrightarrow T$, $De(T) = \{C\}$ and $Pp(T) = \{A, B\}$ such that $d(A) = [i - m, j - m]$, $d(B) = [k - m, l - m]$ and $d(C) = [0, n - m]$. If $i < k$ and $i < n$ then after duration $i - m$, A becomes enactable. If $k < i$ and $k < n$, then after duration

$k - m$, B becomes enactable, and if $n < j$ and $n < k$ then after duration $n - m$, C can no longer be delayed.

If task C is executed, in the subsequent time stable T, $Fr(T) = \{D\}$ and $Pp(T) = \{A, B\}$, where $d(D) = [o, p]$. If $o < i - m$ and $o < k - m$, after duration o, D may either be executed or delayed.

We now return to our running example. Given time stable state shown at the bottom of Figure 6.9, if we assume the example schedule shown in Table 6.1, and apply Rule A-E, we yield the transition $(ES, MS, \emptyset, \emptyset, \{timer1, B1\}) \xrightarrow{\tau} (ES, MS, \emptyset, \emptyset, \{B1'\}, \{timer1'\}, \emptyset)$, such that $timer1' = dmin(timer1, 30)$ and $B1' = dmax(dmin(B1, 30), 30)$, where 30 minutes is the minimum duration and minute is the smallest unit of time used in the schedule in Table 6.1.

6.4.5 Implementation

Record that function `stable` evaluates the expression `timer ss ms ae rn`, and function *timer* is defined as follows.

```
timer :: [Element] -> [(Element,Int)] -> [Element] -> [Element] -> Process
timer ss ms ae rn =
  let (de,ps,fr) = fdexcept ss rn
      ((nm,ni),op) = par (unzip.(map splitseq)) id (parpost (union ps de) ps)
      (fm,fi) = (unzip.(map splitseq).(filter (ismiseq.etype.atom))) fr
      (pm,as) = sorting op de (unions [(fr \\ fm),fi,ni])
      (en,ac) = break ((> 0).minrange) as
  in exe (ss,ms,[],ae,(unions [ac,pm,nm,fm]),en,[])
```

The first argument `ss` is the list of elements contained in the BPMN process; the second argument `ms` is the list of pairs recording active multiple instance subprocesses and their remaining iterations; the third argument `ae` is the list of active untimed elements that are not enactable; and the fourth argument `rn` is the list of active time elements.

The full definition of `timer` can be found in Section E.3. We now provide an overview of how `timer` implements transition rule A-E shown in Figure 6.14. Specifically, function `timer` contains several local definitions, which match the conditions of transition rule A-E. From top to bottom, given some time stable state X:

- The first local definition is the triple of lists of elements `(de,ps,fr)`. The list `de` corresponds to the set of delayed elements $De(X)$, `ps` corresponds to the set of postponed elements $Pp(X)$, and `fr` corresponds to the set of fresh timed elements $Fr(X)$.
- The second local definition is the pair `((nm,ni),op)`. The list `nm` corresponds to the set $\{m : M_n(X) \bullet first(splitsq(m))\}$, `ni` corresponds to the set $\{m : M_n(X) \bullet second(splitsq(m))\}$, and `op` corresponds to the set $Pp(X) \setminus M_n(X)$.
- The third local definition is the pair of lists `(fm,fi)`. The list `fm` corresponds to the set $\{m : M_f(X) \bullet first(splitsq(m))\}$, and `fi` corresponds to the set $\{m : M_f(X) \bullet second(splitsq(m))\}$.
- The fourth local definition is the pair of lists `(pm,as)`. The value `pm` corresponds to the set $M_p(X)$, and `as` corresponds to the set As defined in transition rule A-E.
- The fifth local definition is the pair of lists `(en,ac)`. The list `en` corresponds to the set $As \setminus Ac$, and `ac` corresponds to the set Ac. Both are defined in Rule A-E.

After evaluating the local definitions, function `timer` evaluates the expression `exe (ss,ms,[],ae,(unions [ac,pm,nm,fm]),en,[])`, where `exe` implements the coordination step at a timed state and is presented in Section 6.5. The tuple `(ss,ms,[],ae,(unions [ac,pm,nm,fm]),en,[])` records the initial timed state as defined in Definition 6.10, where the set `(unions [ac,pm,nm,fm])` is the set of postponed elements and active multiple instance tasks.

6.5 Coordinating Timed States

6.5.1 Introduction

This section describes Step 3 of the coordination introduced on Page 103. This step executes all enactable timed elements to reach an untimed state.

In Section 6.4.1 we gave a definition of timed states (Definition 6.9) and initial timed states (Definition 6.10). At a timed state, where all enactable timed elements have either completed execution, delayed or interrupted, the coordination procedure proceeds to an untimed state, thereby commencing the untimed state coordination. Untimed state coordination is Step 1 of the coordination procedure on Page 103 and was described in Section 6.3. Definition 6.12 defines the transition rule T-C from such a timed state to the subsequent untimed state.

Definition 6.12. ***Timed Complete Transition Rule*** **T-C.** *Given a timed state* (S, M, W, E, P, C, O) *such that* $C = \emptyset$ *and* $O = \emptyset$*, the timed complete transition rule* T-C *is defined as* $(S, M, E, W, P, C, O) \xrightarrow{\tau} (S, M, E, W, P)$.

Running Example. As a running example, we consider the partial BPMN process in Figure 6.16. We assume an initial timed state X such that its component C, the set of enactment timed elements, contains the following: Task E, multiple instance task B contained in the multiple instance subprocess A, associated timed exception flows e and f, and intermediate timer event g.

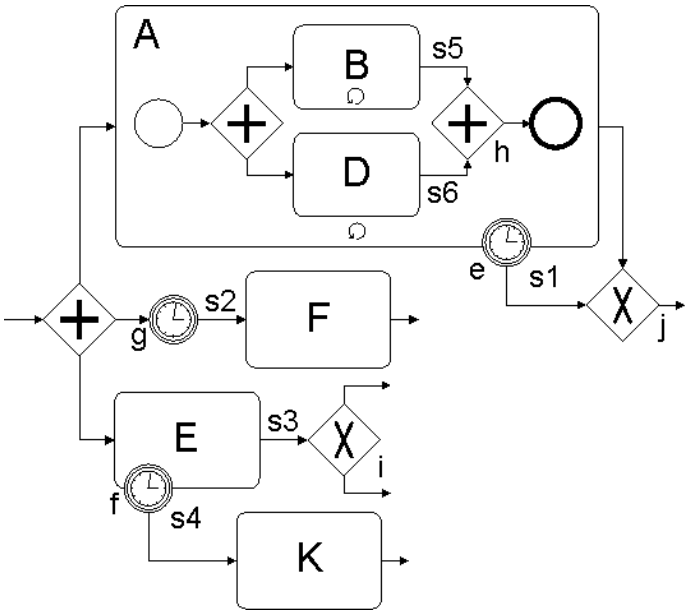


Figure 6.16: An illustration of timed elements coordination

Similar to untimed state coordination, we define transition rules to model the coordination procedure at timed states. We categorise transition rules according to the types of elements that enable the transitions. Table 6.4 lists the category prefixes and their corresponding descriptions. For example transition rules whose name begin with V define transitions that perform an outgoing sequence flow of either a start timer event or an intermediate timer event that is not an associated timed exception flow. We first consider category V in Section 6.5.2, category D in Section 6.5.3, and category X in Section 6.5.4. We provide an overview of our Haskell implementation of this coordination step in Section 6.5.6.

6.5.2 Coordinating Timer Events

Transition rules shown in Figure 6.17 define transitions to trigger an outgoing sequence flow of either a start timer event or an intermediate timer event that is not an associated timed exception flow.

Category prefix	Transition description
V	perform an outgoing sequence flow of either a start timer event or an intermediate timer event that is not an associated timed exception flow
T	perform an outgoing sequence flow of a task or a multiple instance task, or perform the work of a task or a multiple instance task
X	perform an associated timed exception flow

Table 6.4: Transition rule categories for timed states

We first describe the rules' similarities before describing their differences. Rules shown in Figure 6.17 define transitions to trigger an outgoing sequence flow s of a timer event r. After performing s, r completes its execution, and therefore is removed from C.

Given $r \in C$ is a start timer event or an intermediate timer event such that $in(r) \neq \emptyset$, and $s \in out(r)$:

$$\frac{\begin{array}{l}e \in_{\mathrm{p}} S \text{ is untimed, either } s \in in(e) \text{ and if it is an AND join gateway then } e \in W \\ \text{and } \#in(e) = 1 \text{ or } c \in_{\mathrm{p}} S \text{ is a subprocess, } s \in in(c), \ e \text{ is a start event and } (e, c) \in \mathit{contains}\end{array}}{(S,M,E,W,P,C,O) \xrightarrow{s} (S,M,E \cup \{e\}, W \setminus \{e\}, P, C \setminus \{r\}, O)} \quad [\text{ V-U }]$$

$$\frac{e \in_{\mathrm{p}} S \text{ or } e \in W \text{ is an AND join gateway, } s \in in(e) \text{ and } \#in(e) > 1}{(S,M,E,W,P,C,O) \xrightarrow{s} (S,M,E,(W \setminus \{e\}) \cup \{rm(e,s)\}, P, C \setminus \{r\}, O)} \quad [\text{ V-J }]$$

$$\frac{\begin{array}{l}c \in_{\mathrm{p}} S \text{ is a multiple instance subprocess, has } l \text{ instances and } s \in in(c) \\ e \text{ is a start event and } (e, c) \in \mathit{contains}\end{array}}{(S,M,E,W,P,C,O) \xrightarrow{s} (S, M \cup \{(c, l-1)\}, E \cup \{e\}, W, P, C \setminus \{r\}, O)} \quad [\text{ V-M }]$$

$$\frac{\begin{array}{l}c \in_{\mathrm{p}} S \text{ is a multiple instance subprocess, has } l \text{ instances and } s \in in(c) \\ e \text{ is an start timer event and } (e, c) \in \mathit{contains}\end{array}}{(S,M,E,W,P,C,O) \xrightarrow{s} (S, M \cup \{(c, l-1)\}, E, W, P \cup \{e\}, C \setminus \{r\}, O)} \quad [\text{ V-M' }]$$

$$\frac{\begin{array}{l}e \in_{\mathrm{p}} S \text{ is timed, either } s \in in(e) \text{ or} \\ c \in_{\mathrm{p}} S \text{ is a subprocess, } s \in in(c), \ e \text{ is a start timer event and } (e, c) \in \mathit{contains}\end{array}}{(S,M,E,W,P,C,O) \xrightarrow{s} (S,M,E,W,P \cup \{e\}, C \setminus \{r\}, O)} \quad [\text{ V-T }]$$

Figure 6.17: Transition rules (7)

The differences between Rules V-U, V-J, V-M, V-M' and V-T shown in Figure 6.17 are as follows.

- Rule V-U transitions such that e is one of the following: an AND join gateway with one incoming sequence flow s; a start event in a subprocess c with an incoming sequence flow s; and an untimed element that is not an AND join gateway and has s as one of its incoming sequence flows. For these cases, e becomes enactable (in an untimed state) after s is triggered, therefore e is added to E.
- Rule V-J defines transitions such that e is an AND join gateway that has more than one incoming sequence flow and where s is one of its incoming sequence flows. In this case e is active but not

enactable since its other incoming sequence flows have not been triggered, therefore $rm(e, s)$ is replaced with r in W; note that r needs not be in W initially.

- Rule V-M defines transitions such that e is a start event directly contained in some multiple instance subprocess c, and that c has incoming sequence flow s and specifies l instances. In this case e becomes enactable (at an untimed state) and is added to E. To keep track of the number of c's remaining instances, the pair $(c, l-1)$ is added to M.
- Rule V-M' defines such that e is a start timer event directly contained in some multiple instance subprocess c, and that c has incoming sequence flow s and specifies l instances. In this case e is a timed element and therefore is added to P. Also to keep track of the number of c's remaining instances, the pair $(c, l-1)$ is added to M.
- Rule V-T defines transitions such that e is a timed element and has incoming sequence flow s. In this case e is a timed element and is therefore added to P.

For example, consider the BPMN process in Figure 6.16. According to Rule V-T, the intermediate timer event g completes its execution by triggering its outgoing sequence flow $s2$, which is also the incoming sequence flow of task element F. This is modelled by the following transition.

$$(S, M, E, W, P, C, O) \xrightarrow{s.s2} (S, M, E, W, P \cup \{F\}, C \setminus \{g\}, O)$$

6.5.3 Coordinating Task Elements

Figures 6.18 and 6.19 define the transition rules to coordinate enactable task and multiple instance task elements.

Unlike timer events, executing a task element consists of performing two CSP events that denote its work and its outgoing sequence flow. For task t with outgoing sequence flow f, the CSP event $w.t$ denotes t's work and event $s.f$ denotes t's outgoing sequence flow f. Moreover, a task element has a duration range, and may nondeterministically choose to delay its execution until its maximum duration is lapsed.

$$\frac{\begin{array}{l}e \in C \text{ is a task or an instance of some multiple instance task } m \in P \\ \text{The maximum duration of } e,\ second((atom\ e).range), \text{ is larger than 0}\end{array}}{(S,M,E,W,P,C,O) \xrightarrow{\tau} (S,M,E,W,P \cup \{e\}, C \setminus \{e\}, O)} \quad [\text{ T-D }]$$

$$\frac{e \in C \text{ is a task and } w.e \in work(e)}{(S,M,E,W,P,C,O) \xrightarrow{w.e} (S,M,E,W,P,C \setminus \{e\}, O \cup \{e\})} \quad [\text{ T-E }]$$

$$\frac{\begin{array}{l}e \in C \text{ is an instance of some multiple instance task } m \in P \text{ where } w.e \in work(e) \text{ and} \\ m \text{ has 0 instance.}\end{array}}{(S,M,E,W,P,C,O) \xrightarrow{w.e} (S,M,E,W,P \setminus \{m\}, C \setminus \{e\}, O \cup \{m\})} \quad [\text{ T-I }]$$

$$\frac{\begin{array}{l}e \in C \text{ is an instance of some multiple instance task } m \in P \text{ and } w.e \in work(e) \\ m \text{ has } l \text{ number of instances, where } l > 0\end{array}}{(S,M,E,W,P,C,O) \xrightarrow{w.e} (S,M,E,W,P,C \setminus \{e\}, O)} \quad [\text{ T-I' }]$$

Figure 6.18: Transition rules (8)

We consider the transition rules shown in Figure 6.18. These rules define transitions to either perform a task's work or delay its execution. Rule T-D defines transitions to delay the execution of some element

e that is either a task element or a task instance of a multiple instance task element. Transitions defined by this rule perform the τ event to model this nondeterministic choice and move element e from the set of enactable timed elements C to set P; P is the set of timed elements that are either postponed or delayed. Rule T-E defines transitions to perform the work of some task element e. Transitions defined by this rule perform the CSP event $w.e$ to denote e's work and move e from the set of enactable timed element C to set O; O is a set of task elements and multiple instance tasks whose outgoing sequence flows are available to be triggered. Rules T-I and T-I' define transitions to perform the work of an instance e of some multiple instance task $m \in P$. In Rule T-I, m has 0 remaining instances, while in Rule T-I', m has one or more remaining instances. Both transition rules perform the CSP event $w.e$, to denote e's work and remove e from the set of enactable elements C. Rule T-I also moves m from P to O as m has executed all its instances and therefore its outgoing sequence flow is available to be triggered. For Rule T-I', however, m has one or more instances to execute and therefore remains in set P.

For example, consider the BPMN process in Figure 6.16. According to Rule T-D, if task E has a non-zero maximum duration, it can delay its execution, yielding the following transition.

$$(S, M, E, W, P, C, O) \xrightarrow{tau} (S, M, E, W, P \cup \{e\}, C \setminus \{e\}, O)$$

Here we use e to denote task E to avoid the confusion with the state's component E. Conversely, according to Rule T-E, E can perform its work, yielding the following transition,

$$(S, M, E, W, P, C, O) \xrightarrow{w.e} (S, M, E, W, P, C \setminus \{e\}, O \cup \{e\})$$

where we use e to denote task E and CSP event $w.e$ denotes e's work; Similarly, according to Rule T-I, if multiple instance task $B \in P$ has one remaining instance b and b performs its work, yielding the following transition,

$$(S, M, E, W, P, C, O) \xrightarrow{w.B} (S, M, E, W, P \setminus \{B\}, C \setminus \{b\}, O \cup \{B\})$$

where CSP event $w.B$ denotes B's work.

We now consider transition rules shown in Figure 6.19; these rules define transitions to trigger the outgoing sequence flow s of element $r \in O$ that is either a task or a multiple instance task. After triggering s, r completes its execution and is removed from O. Since r completes its execution, transitions defined by these rules also remove r's timed exception flows from P and C. The differences between Rules T-U, T-J, T-M, T-M' and T-T shown in Figure 6.19 correspond to those between Rules V-U, V-J, V-M, V-M' and V-T shown in Figure 6.17.

For example, we consider the BPMN process in Figure 6.16. According to Rule T-U, task element E can trigger outgoing sequence flow $s3$ to complete its execution; $s3$ is also the incoming sequence flow of XOR split gateway i. This behaviour yields the following transition,

$$(S, M, E, W, P, C, O) \xrightarrow{s.s3} (S, M, E \cup \{i\}, W, P, C \setminus \{f\}, O \setminus \{e\})$$

where we use e to denote task E to avoid confusion with component E of the timed state, and where f is E's timed exception flow.

Similarly, according to Rule T-J, if task $D \in P$ is postponed, multiple instance task $B \in O$ can trigger outgoing sequence flow $s5$ to complete its execution; $s5$ is also the incoming sequence flow of AND join gateway h. Since h has two incoming sequence flows, namely $\{s5, s6\}$, this behaviour yields the following transition.

$$(S, M, E, W, P, C, O) \xrightarrow{s.s5} (S, M, E, (W \setminus \{h\}) \cup \{rm(s5, h)\}, P, C, O \setminus \{B\})$$

Conversely, if task D has already triggered sequence flow $s6$, and multiple instance task B triggers sequence flow $s5$, then according to Rule T-U, this behaviour yields the following transition.

$$(S, M, E, W, P, C, O) \xrightarrow{s.s5} (S, M, E \cup \{h\}, W \setminus \{h\}, P, C, O \setminus \{B\})$$

Given $r \in O$ is a task or a multiple task instance that has 0 instance, such that $s \in out(r)$:

$$\frac{\begin{array}{l}e \in_{\mathrm{p}} S \text{ is untimed, either } s \in in(e) \text{ and if it is an AND join gateway then } e \in W \\ \text{and } \#in(e) = 1 \text{ or } c \in_{\mathrm{p}} S \text{ is a subprocess, } s \in in(c),\ e \text{ is a start event and } (e, c) \in \mathit{contains}\end{array}}{(S,M,E,W,P,C,O) \xrightarrow{s} (S,M,E \cup \{e\}, W \setminus \{e\}, P \setminus ex(r,P), C \setminus ex(r,C), O \setminus \{r\})} \quad [\text{ T-U }]$$

$$\frac{e \in_{\mathrm{p}} S \text{ or } e \in W \text{ is an AND join gateway, } s \in in(e) \text{ and } \#in(e) > 1}{(S,M,E,W,P,C,O) \xrightarrow{s} (S,M,E,(W \setminus \{e\}) \cup \{rm(e,s)\}, P \setminus ex(r,P), C \setminus ex(r,C), O \setminus \{r\})} \quad [\text{ T-J }]$$

$$\frac{\begin{array}{l}c \in_{\mathrm{p}} S \text{ is a multiple instance subprocess, has } l \text{ instances and } s \in in(c) \\ e \text{ is a start event and } (r, C) \in \mathit{contains}\end{array}}{(S,M,E,W,P,C,O) \xrightarrow{s} (S, M \cup \{(c, l-1)\}, E \cup \{e\}, W, P \setminus ex(r,P), C \setminus ex(r,C), O \setminus \{r\})} \quad [\text{ T-M }]$$

$$\frac{\begin{array}{l}c \in_{\mathrm{p}} S \text{ is a multiple instance subprocess, has } l \text{ instances and } s \in in(c) \\ e \text{ is an start timer event and } (r, C) \in \mathit{contains}\end{array}}{(S,M,E,W,P,C,O) \xrightarrow{s} (S, M \cup \{(c, l-1)\}, E, W, (P \setminus ex(r,P)) \cup \{e\}, C \setminus ex(r,C), O \setminus \{r\})} \quad [\text{ T-M' }]$$

$$\frac{\begin{array}{l}e \in_{\mathrm{p}} S \text{ is timed, either } s \in in(e) \text{ or} \\ c \in_{\mathrm{p}} S \text{ is a subprocess, } s \in in(c),\ e \text{ is a start timer event and } (r, C) \in \mathit{contains}\end{array}}{(S,M,E,W,P,C,O) \xrightarrow{s} (S,M,E,W,(P \setminus ex(r,P)) \cup \{e\}, C \setminus ex(r,C), O \setminus \{r\})} \quad [\text{ T-T }]$$

Figure 6.19: Transition rules (9)

6.5.4 Coordinating Timed Exception Flows

This section describes the coordination of timed exception flows. After an exception flow attached to a task or a multiple instance task is triggered, the execution of that task or multiple instance task is interrupted. Similarly, after an exception flow attached to a subprocess or a multiple instance subprocess is triggered, the executions of that subprocess or multiple instance subprocess as well as all elements contained in that subprocess are interrupted. To model the interruption caused by an exception flow, we provide some preliminaries in Definition 6.13.

Definition 6.13. *Given a timed state (S, M, E, W, P, C, O), the functions $assocs(f,t)$, $mults(f,t)$ and $timers(f,t)$ are defined as follows:*

- *$assocs(f,t) = \{e : Element \mid e \in_p S \wedge (f,t) \in assoc(e)\}$ takes a timed exception flow (f,t) and returns the set containing all elements that have the associated timed exception flow (f,t).*
- *$mults(f,t) = assocs(f,t) \lhd M$ is a subset of M, such that the first component of each element in that subset records a multiple instance subprocess that has the associated timed exception flow (f,t).*
- *$timers(f,t) = \{c : C \cup P \mid out(c) \subseteq exit(\!| assocs(f,t) |\!)\}$ is the set of associated timed exception flows attached to elements in $assocs(f,t)$.*

Figure 6.20 shows the transition rules for coordinating timed exception flows. Specifically, the rules define transitions that perform the outgoing sequence flow f of some intermediate timer event $r \in C$, which models the associated exception flow (f,t). Specifically, after performing sequence flow f, all elements recorded in the timed state, which are associated timed exception flow (f,t), are interrupted.

Given $r \in C$ is an intermediate timer event $exp(f,t)$ for some associated timed exception flow $(f,t) \in assoc(e)$ of some activity element e. Let s be the CSP event to denote the sequence flow f and A, U and V be abbreviations for sets $assocs(f,t)$, $mults(f,t)$ and $timers(f,t)$ respectively.

$$\frac{\begin{array}{l}e \in_{\mathrm{p}} S \text{ is untimed, either } s \in in(e) \text{ and if it is an AND join gateway then } e \in W \\ \text{and } \#in(e) = 1 \text{ or } c \in_{\mathrm{p}} S \text{ is a subprocess, } s \in in(c),\ e \text{ is a start event and } (e,c) \in contains\end{array}}{(S,M,E,W,P,C,O) \xrightarrow{s} (S,M \setminus U,(E \setminus A) \cup \{e\},W \setminus (\{e\} \cup A),P \setminus (A \cup V),C \setminus (A \cup V),O \setminus A)} \quad [\text{ X-U }]$$

$$\frac{e \in_{\mathrm{p}} S \text{ or } e \in W \text{ is an AND join gateway, } s \in in(e) \text{ and } \#in(e) > 1}{(S,M,E,W,P,C,O) \xrightarrow{s} (S,M \setminus U,E \setminus A,(W \setminus (\{e\} \cup A)) \cup \{rm(e,s)\},P \setminus (A \cup V),C \setminus (A \cup V),O \setminus A)} \quad [\text{ X-J }]$$

$$\frac{\begin{array}{l}c \in_{\mathrm{p}} S \text{ is a multiple instance subprocess, has } l \text{ instances and } s \in in(c) \\ e \text{ is a start event and } (e,c) \in contains\end{array}}{(S,M,E,W,P,C,O) \xrightarrow{s} (S,(M \setminus U) \cup \{(c,l-1)\},(E \setminus A) \cup \{e\},W \setminus A,P \setminus (A \cup V),C \setminus (A \cup V),O \setminus A)} \quad [\text{ X-M }]$$

$$\frac{\begin{array}{l}c \in_{\mathrm{p}} S \text{ is a multiple instance subprocess, has } l \text{ instances and } s \in in(c) \\ e \text{ is an start timer event and } (e,c) \in contains\end{array}}{(S,M,E,W,P,C,O) \xrightarrow{s} (S,(M \setminus U) \cup \{(c,l-1)\},E \setminus A,W \setminus A,(P \setminus (A \cup V)) \cup \{e\},C \setminus (A \cup V),O \setminus A)} \quad [\text{ X-M' }]$$

$$\frac{\begin{array}{l}e \in_{\mathrm{p}} S \text{ is timed, either } s \in in(e) \text{ or} \\ c \in_{\mathrm{p}} S \text{ is a subprocess, } s \in in(c),\ e \text{ is a start timer event and } (e,c) \in contains\end{array}}{(S,M,E,W,P,C,O) \xrightarrow{s} (S,M \setminus U,E \setminus A,W \setminus A,(P \setminus (A \cup V)) \cup \{e\},C \setminus (A \cup V),O \setminus A)} \quad [\text{ X-T }]$$

Figure 6.20: Transition rules (10)

As a result the following modifications are applied to timed state (S, M, E, W, P, C, O), where sets A, U and V are abbreviations for sets $assocs(f,t)$, $mults(f,t)$ and $timers(f,t)$:

- Elements in M, whose first component is a multiple instance subprocess that has the associated exception flow (f,t), are removed. The result is the set $M \setminus U$.
- Elements with associated exception flows (f,t) are removed from W. The result is the set $W \setminus A$.
- Elements with associated exception flows (f,t) are removed from E. The result is the set $E \setminus A$.
- Elements with associated exception flows (f,t) as well as intermediate timer events modelling timed exception flows (g,h) attached to one of those elements are removed from P. The result is the set $P \setminus (A \cup V)$.
- Elements with associated exception flows (f,t) as well as intermediate timer events modelling timed exception flows (g,h) attached to one of those elements are removed from C. The result is the set $C \setminus (A \cup V)$.
- Elements with associated exception flows (f,t) are removed from O. The result is the set $O \setminus A$.

The differences between Rules X-U, X-J, X-M, X-M' and X-T shown in Figure 6.20 correspond to those between Rules V-U, V-J, V-M, V-M' and V-T shown in Figure 6.17.

For example, consider the BPMN process in Figure 6.16. According to Rule X-T, the intermediate timer event f may interrupt task E's execution and trigger sequence flow $s4$; $s4$ is also the incoming

sequence flow of task K. This behaviour yields the following transition, where we use x to denote task E to avoid confusion:

$$(S, M, E, W, P, C, O) \xrightarrow{s.s4} (S, M, E, W, P \cup \{K\}, C \setminus \{x, f\}, O \setminus \{x\})$$

If task $D \in P$ is postponed and the intermediate timer event e may interrupt subprocess A's execution and trigger sequence flow $s1$; $s1$ is also the incoming sequence flow of XOR join gateway j. This behaviour yields the following transition,

$$(S, M, E, W, P, C, O) \xrightarrow{s.s1} (S, M \setminus \{(A, l_A)\}), E \cup \{j\}, W, P \setminus \{D, B\}, C \setminus \{b, e\}, O \setminus \{b\})$$

where l_A records the number of multiple instance subprocess A's remaining instances, and b is a task instance of multiple instance task B.

6.5.5 Example

Figure 6.21 shows parts of the relative timed coordination of the BPMN pool shown in Figure 6.1. Specifically, it defines the following steps.

1. Transition 1 models minimal time progression from the first time stable state $(ES, MS, \emptyset, \emptyset, \{timer1, B1\})$. Minimal time progression is defined by the transition rule shown on Figure 6.14 and this particular transition has been illustrated at the end of Section 6.4.4, where the minimum duration is 30 minutes such that $timer1' = dmin(timer1, 30)$ and $B1' = dmax(dmin(B1, 30), 30)$.
2. Transition 2 defines the timed state coordination of the intermediate timer event $timer1$.
3. Transition 3 is the timed complete transition to the subsequent untimed state, defined by the transition rule in Definition 6.12.
4. Transitions 4 – 7 define untimed state coordination of the AND split gateway $and3$ to the next time stable state $(ES, MS, \emptyset, \emptyset, \{B1', A1, A2\})$.
5. Transition 8 models the minimal time progression from the time stable state $(ES, MS, \emptyset, \emptyset, \{B1', A1, A2\})$ to timed state $(ES, MS, \emptyset, \emptyset, \{B1''\}, \{A1', A2'\}, \emptyset)$, where the minimum duration is 60 minutes; $A1' = dmax(dmin(A1, 60), 60)$, $A2' = dmax(dmin(A2, 60), 60)$ and $B1'' = dmax(dmin(B1', 60), 60)$.
6. Transitions 9 – 32 define the timed state coordination of tasks $A1'$ and $A2'$.
7. Transitions 33 – 36 define the possible timed complete transitions to the next untimed state.

This coordination is expressed as the CSP process $T0$ defined in Equation 6.4, where $\rhd$ is a derived CSP operator [Ros98] and is defined as follows:

$$P \rhd Q = (P \,\Box\, a \rightarrow Q) \setminus \{a\} \qquad [a \text{ does not appear in } P \text{ or } Q.]$$

Specifically, $T0$ defines the coordination process of the BPMN pool from the first time stable state to four subsequent possible untimed states; these four untimed states are shown as the ending states of transitions 33 – 36 and are denoted by process terms $U21$, $U22$, $U23$ and $U24$ in Equation 6.4.

$$\begin{aligned}
T0 &= s.s2 \rightarrow U9 & T14 &= (w.A1 \rightarrow T17 \,\Box\, s.s6 \rightarrow T19) \rhd T15 \\
U9 &= s.s3 \rightarrow U10 \,\Box\, s.s4 \rightarrow U11 & T15 &= s.s6 \rightarrow U21 \\
U10 &= s.s4 \rightarrow T10 & T16 &= s.s5 \rightarrow U22 \\
U11 &= s.s3 \rightarrow T10 & T17 &= s.s5 \rightarrow T20 \,\Box\, s.s6 \rightarrow T21 \\
T10 &= (w.A1 \rightarrow T13 \,\Box\, w.A2 \rightarrow T14) & T18 &= w.A2 \rightarrow T20 \rhd U22 \\
&\quad \rhd (T11 \sqcap T12) & & \\
T11 &= w.A2 \rightarrow T15 \rhd U24 & T19 &= w.A2 \rightarrow T21 \rhd U21 \\
T12 &= w.A1 \rightarrow T16 \rhd U24 & T20 &= s.s6 \rightarrow U23 \\
T13 &= (w.A2 \rightarrow T17 \,\Box\, s.s5 \rightarrow T18) \rhd T16 & T21 &= s.s5 \rightarrow U23
\end{aligned} \tag{6.4}$$

1	$(ES, MS, \emptyset, \emptyset, \{timer1, B1\})$	$\xrightarrow{\tau}$	$(ES, MS, \emptyset, \emptyset, \{B1'\}, \{timer1'\}, \emptyset)$	A-E
2	$(ES, MS, \emptyset, \emptyset, \{B1'\}, \{timer1'\}, \emptyset)$	$\xrightarrow{s.s2}$	$(ES, MS, \{and3\}, \emptyset, \{B1'\}, \emptyset, \emptyset)$	V-U
3	$(ES, MS, \{and3\}, \emptyset, \{B1'\}, \emptyset, \emptyset)$	$\xrightarrow{\tau}$	$(ES, MS, \{and3\}, \emptyset, \{B1'\})$	T-C
4	$(ES, MS, \{and3\}, \emptyset, \{B1'\})$	$\xrightarrow{s.s3}$	$(ES, MS, \{and3'\}, \emptyset, \{B1', A1\})$	S-T
5		$\xrightarrow{s.s4}$	$(ES, MS, \{and3''\}, \emptyset, \{B1', A2\})$	S-T
6	$(ES, MS, \{and3'\}, \emptyset, \{B1', A1\})$	$\xrightarrow{s.s3}$	$(ES, MS, \emptyset, \emptyset, \{B1', A1, A2\})$	U-T
7	$(ES, MS, \{and3''\}, \emptyset, \{B1', A2\})$	$\xrightarrow{s.s4}$	$(ES, MS, \emptyset, \emptyset, \{B1', A1, A2\})$	U-T
8	$(ES, MS, \emptyset, \emptyset, \{B1', A1, A2\})$	$\xrightarrow{\tau}$	$(ES, MS, \emptyset, \emptyset, \{B1''\}, \{A1', A2'\}, \emptyset)$	A-E
9	$(ES, MS, \emptyset, \emptyset, \{B1''\}, \{A1', A2'\}, \emptyset)$	$\xrightarrow{\tau}$	$(ES, MS, \emptyset, \emptyset, \{B1'', A1'\}, \{A2'\}, \emptyset)$	T-D
10		$\xrightarrow{\tau}$	$(ES, MS, \emptyset, \emptyset, \{B1'', A2'\}, \{A1'\}, \emptyset)$	T-D
11		$\xrightarrow{w.A1}$	$(ES, MS, \emptyset, \emptyset, \{B1''\}, \{A2'\}, \{A1'\})$	T-E
12		$\xrightarrow{w.A2}$	$(ES, MS, \emptyset, \emptyset, \{B1''\}, \{A1'\}, \{A2'\})$	T-E
13	$(ES, MS, \emptyset, \emptyset, \{B1'', A1'\}, \{A2'\}, \emptyset)$	$\xrightarrow{w.A2}$	$(ES, MS, \emptyset, \emptyset, \{B1'', A1'\}, \emptyset, \{A2'\})$	T-E
14		$\xrightarrow{\tau}$	$(ES, MS, \emptyset, \emptyset, \{B1'', A2', A1'\}, \emptyset, \emptyset)$	T-D
15	$(ES, MS, \emptyset, \emptyset, \{B1'', A2'\}, \{A1'\}, \emptyset)$	$\xrightarrow{w.A1}$	$(ES, MS, \emptyset, \emptyset, \{B1'', A2'\}, \emptyset, \{A1'\})$	T-E
16		$\xrightarrow{\tau}$	$(ES, MS, \emptyset, \emptyset, \{B1'', A2', A1'\}, \emptyset, \emptyset)$	T-D
17	$(ES, MS, \emptyset, \emptyset, \{B1''\}, \{A2'\}, \{A1'\})$	$\xrightarrow{w.A2}$	$(ES, MS, \emptyset, \emptyset, \{B1''\}, \emptyset, \{A1', A2'\})$	T-E
18		$\xrightarrow{s.s5}$	$(ES, MS, \{and4'\}, \emptyset, \{B1''\}, \{A2'\}, \emptyset)$	T-J
19		$\xrightarrow{\tau}$	$(ES, MS, \emptyset, \emptyset, \{B1'', A2'\}, \emptyset, \{A1'\})$	T-D
20	$(ES, MS, \emptyset, \emptyset, \{B1''\}, \{A1'\}, \{A2'\})$	$\xrightarrow{w.A1}$	$(ES, MS, \emptyset, \emptyset, \{B1''\}, \emptyset, \{A1', A2'\})$	T-E
21		$\xrightarrow{s.s6}$	$(ES, MS, \{and4''\}, \emptyset, \{B1''\}, \{A1'\}, \emptyset)$	T-J
22		$\xrightarrow{\tau}$	$(ES, MS, \emptyset, \emptyset, \{B1'', A1'\}, \emptyset, \{A2'\})$	T-D
23	$(ES, MS, \emptyset, \emptyset, \{B1'', A1'\}, \emptyset, \{A2'\})$	$\xrightarrow{s.s6}$	$(ES, MS, \emptyset, \{and4''\}, \{B1'', A1'\}, \emptyset, \emptyset)$	T-J
24	$(ES, MS, \emptyset, \emptyset, \{B1'', A2'\}, \emptyset, \{A1'\})$	$\xrightarrow{s.s5}$	$(ES, MS, \emptyset, \{and4'\}, \{B1'', A2'\}, \emptyset, \emptyset)$	T-J
25	$(ES, MS, \emptyset, \emptyset, \{B1''\}, \emptyset, \{A1', A2'\})$	$\xrightarrow{s.s5}$	$(ES, MS, \emptyset, \{and4'\}, \{B1''\}, \emptyset, \{A2'\})$	T-J
26		$\xrightarrow{s.s6}$	$(ES, MS, \emptyset, \{and4''\}, \{B1''\}, \emptyset, \{A1'\})$	T-J
27	$(ES, MS, \emptyset, \{and4'\}, \{B1''\}, \{A2'\}, \emptyset)$	$\xrightarrow{w.A2}$	$(ES, MS, \emptyset, \{and4'\}, \{B1''\}, \emptyset, \{A2'\})$	T-E
28		$\xrightarrow{\tau}$	$(ES, MS, \emptyset, \{and4'\}, \{B1'', A2'\}, \emptyset, \emptyset)$	T-D
29	$(ES, MS, \emptyset, \{and4''\}, \{B1''\}, \{A1'\}, \emptyset)$	$\xrightarrow{w.A1}$	$(ES, MS, \emptyset, \{and4''\}, \{B1''\}, \emptyset, \{A1'\})$	T-E
30		$\xrightarrow{\tau}$	$(ES, MS, \emptyset, \{and4''\}, \{B1'', A1'\}, \emptyset, \emptyset)$	T-D
31	$(ES, MS, \emptyset, \{and4'\}, \{B1''\}, \emptyset, \{A2'\})$	$\xrightarrow{s.s6}$	$(ES, MS, \{and4'''\}, \emptyset, \{B1''\}, \emptyset, \emptyset)$	T-U
32	$(ES, MS, \emptyset, \{and4''\}, \{B1''\}, \emptyset, \{A1'\})$	$\xrightarrow{s.s5}$	$(ES, MS, \{and4'''\}, \emptyset, \{B1''\}, \emptyset, \emptyset)$	T-U
33	$(ES, MS, \emptyset, \{and4''\}, \{B1'', A1'\}, \emptyset, \emptyset)$	$\xrightarrow{\tau}$	$(ES, MS, \emptyset, \{and4''\}, \{B1'', A1'\})$	T-C
34	$(ES, MS, \emptyset, \{and4'\}, \{B1'', A2'\}, \emptyset, \emptyset)$	$\xrightarrow{\tau}$	$(ES, MS, \emptyset, \{and4'\}, \{B1'', A2'\})$	T-C
35	$(ES, MS, \{and4'''\}, \emptyset, \{B1''\}, \emptyset, \emptyset)$	$\xrightarrow{\tau}$	$(ES, MS, \{and4'''\}, \emptyset, \{B1''\})$	T-C
36	$(ES, MS, \emptyset, \emptyset, \{B1'', A2', A1'\}, \emptyset, \emptyset)$	$\xrightarrow{\tau}$	$(ES, MS, \emptyset, \emptyset, \{B1'', A2', A1'\})$	T-C

Figure 6.21: Coordinating BPMN process shown in Figure 6.1

6.5.6 Implementation

We provide function `exe` to implement the coordination of timed states and is defined as follows,

```
exe :: TimedState -> Process
exe (ss,ms,es,ae,ps,[],[]) = stable (ss,ms,es,ae,ps)
exe (ss,ms,es,ae,ps,cs,os) =
  exts ([ csbranch (ss,ms,es,ae,ps,cs,os) e | e <- cs] ++
        [ osbranch (ss,ms,es,ae,ps,cs,os) e | e <- os])
```

where we model a timed state using the following type synonym.

```
type TimedState = ([Element],[(Element,Int)],[Element],[Element],[Element],[Element],[Element])
```

Specifically, function `exe` takes a tuple `(ss,ms,ae,es,ps,cs,os)` recording a timed state. Here `ss` models component S, `ms` models M, `ue` models E, `ae` models W, `ps` models P, `cs` models C and `os` models O.

Function `exe` implements transition rules shown in Figures 6.17, 6.18, 6.19 and 6.20 by evaluating the following clauses. The first clause implements timed complete transition rule T-C, and the second clause matches any other timed states. At this point, the coordination is at a timed state with enactable elements, and `exe` returns a CSP process that defines the external choice over all possible transitions. Transition rules shown in Figures 6.17, 6.18 and 6.20 are implemented by the function `csbranch`, while transition rules shown in Figures 6.19 are implemented by the function `osbranch`. The definition of these functions can be found in Section E.4.

6.6 Analysis

We provided an extension to model BPMN process's relative timed behaviour in the form of transition rules. Given a BPMN pool p, we defined the CSP process $T(p) = PL(p) \,[\![\Sigma]\!]\, CP(p)$, where $PL(p)$ is the enactment process of p and $CP(p)$ is the coordination process of p; $PL(p)$ models the untimed behaviour of p, while $CP(p)$ coordinates p's timed behaviour and is calculated using the transition rules. Given any BPMN pool p, we establish the following two properties about p's coordination process $CP(p)$:

1. $CP(p)$ ensures time progression between any two consecutive time stable state.
2. If p's enactment process $PL(p)$ is deadlock free, then $PL(p) \,[\![\,\Sigma\,]\!]\, CP(p)$ is also deadlock free.

Property 1 ensures the coordination process $CP(p)$ performs some timed element's behaviour between any two adjacent time stable states; here a timed element's behaviour is either performing a task element/instance's work, delaying a task element/instance, triggering a task element/instance's outgoing sequence flow, or triggering a timed exception flow. We provide the following definition of time progression between two adjacent time stable states $X \Longrightarrow Y$, that is, X proceeds to Y via a sequence of timed states followed by a sequence of untimed states.

Definition 6.14. ***Time Progression.*** *Given any two consecutive time stable states X and Y such that $X \Longrightarrow Y$, time progresses from X to Y, denoted as $pg(X, Y)$, if and only if at least one of the following holds.*

1. $Pp(X) \not\subseteq Pp(Y)$
2. $De(X) \not\subseteq De(Y)$
3. $(Fr(X) \cup M_f(X)) \not\subseteq Pp(Y)$

Definition 6.14 states the following: Time progresses from time stable state X to state Y, if either (1) at least one postponed element at X becomes enactable at Y, or (2) at least one delayed element at X is executed before Y, or (3) not all fresh elements have been postponed at Y. The following result establishes Property 1.

Lemma 6.15. *Given two consecutive time stable states X and Y, such that $X \Longrightarrow Y$, we have $pg(X, Y)$.*

Proof. See Page 258 (Section C.3). □

Consider again the example described in Section 6.4.4. Here we assume the execution of the partial BPMN process shown in Figure 6.15 has reached time stable state S such that its set of fresh timed elements are tasks A, B and C where $d(A) = [i, j]$, $d(B) = [k, l]$ and $d(C) = [m, n]$ and $m < i$ and $m < k$. By Rule A-E, $dur(S) = m$, and both A and B are postponed for duration m, during which task C can be either executed or delayed. If task C is delayed, then in the subsequent time stable T, where $S \Longrightarrow T$, $De(T) = \{C\}$ and $Pp(T) = \{A, B\}$ such that $d(A) = [i - m, j - m]$, $d(B) = [k - m, l - m]$ and $d(C) = [0, n - m]$. If $i < k$ and $i < n$ then after duration $i - m$, A becomes enactable, therefore time progresses from S to T.

Property 2 ensures that given any BPMN pool p, $CP(p)$ cannot cause $PL(p)$ to deadlock. Theorem 6.16 establishes this property.

Theorem 6.16. ***Deadlock Freedom.*** *Given any BPMN pool p, such that if process $PL(p)$, modelling p's untimed behaviour, is deadlock free, then process $PL(p) \,[\![\, \Sigma \,]\!]\, CP(p)$ is also deadlock free.*

Proof. See Page 261 (Section C.3). □

Property 2 also shows that deadlock freedom is an untimed property, that is, if p's untimed behaviour is deadlock free, p's relative timed behaviour is also deadlock free, regardless of the timing information specified.

We now return to the running example. The following CSP process *Product* models the relative timed behaviour of the production business process diagram shown in Figure 6.1.

$$Product = EP \,[\![\, \Sigma \,]\!]\, CP$$

The process *Product* is the parallel composition of the enactment process *EP* and the coordination process *CP*. The definition of *EP* is straightforward, and has been omitted; the definition of process *CP* is given by the set of process equations (6.5) on Page 127 at the end of the chapter.

By Theorem 6.16, we check whether *Product* is deadlock-free by checking the following refinement assertion using the FDR tool.

$$DF \sqsubseteq_{\mathcal{F}} EP$$

Here process *DF* is the characteristic deadlock free process defined in Equation 5.6.2 on Page 77.

We also check if *Product* satisfies the requirement introduced at the beginning of this chapter. The requirement states that "at least one of artifacts $A1$ and $A2$ must be composed in between artifacts $B1$ and $B2$". The following process, *Spec*0, models this requirement, where the work of tasks $A1$, $A2$, $B1$ and $B2$ are modelled by CSP events $w.A1$, $w.A2$, $w.B1$ and $w.B2$ respectively.

$$\begin{aligned}
Spec0 &= (\sqcap x : \Sigma \setminus \{w.B1\} \bullet x \rightarrow Spec0) \sqcap w.B1 \rightarrow Spec1 \\
Spec1 &= w.A1 \rightarrow Spec2 \sqcap w.A2 \rightarrow Spec2 \sqcap (\sqcap x : \Sigma \setminus \{w.B2, w.A1, w.A2\} \bullet x \rightarrow Spec1) \\
Spec2 &= (\sqcap x : \Sigma \setminus \{w.B1, w.B2\} \bullet x \rightarrow Spec2) \sqcap w.B1 \rightarrow Spec1 \sqcap w.B2 \rightarrow Spec0
\end{aligned}$$

We check if *Product* satisfies the requirement by checking the following refinement assertion using the FDR tool.

$$Spec0 \sqsubseteq_{\mathcal{F}} Product \setminus (\Sigma \setminus \{w.A1, w.A2, w.B1, w.B2\})$$

This refinement does not hold and the trace $\langle w.A2, w.A1, w.B1, w.B2 \rangle$ is given as a counterexample by FDR. This counterexample shows the possibility of performing tasks $A1$ and $A2$ before task $B1$ and that task $B2$ can be performed after task $B1$ without performing any of tasks $A1$ and $A2$.

6.7 Summary

In this chapter we provided an extension to the process semantics presented in Chapter 5 to model BPMN process's relative timed behaviour. We adopted a variant of the two-phase functioning approach widely used in real-time systems and timed coordination languages [LJBB06] and defined a set of transition rules to coordinate the timed behaviour of BPMN processes. We formalised the coordination procedure

by showing that the coordination procedure yields CSP processes that cannot cause their untimed counterpart to deadlock. We have also illustrated by examples how to use the extension and described an implementation in Haskell.

$$
\begin{array}{ll}
CP = s.s11 \rightarrow (C01 \,\Box\, C02) & C46 = w.A1 \rightarrow C71 \rhd w.A2 \rightarrow C34 \\
C01 = s.s12 \rightarrow (C03 \,\Box\, C04) & C47 = C77 \,\Box\, s.s9 \rightarrow C22 \,\Box\, s.s6 \rightarrow C31 \\
C02 = s.s13 \rightarrow (C05 \,\Box\, C06) & C48 = w.B2 \rightarrow s.s10 \rightarrow c.end3 \rightarrow C92 \\
C03 = s.s1 \rightarrow s.s13 \rightarrow s.s8 \rightarrow C14 & C83 = C80 \rhd w.A2 \rightarrow s.s6 \rightarrow C44 \\
C04 = s.s13 \rightarrow (C07 \,\Box\, C08) & C87 = w.A2 \rightarrow C30 \,\Box\, s.s5 \rightarrow w.A2 \rightarrow s.s6 \rightarrow C63 \\
C05 = s.s8 \rightarrow s.s12 \rightarrow s.s1 \rightarrow C14 & C88 = (C76 \rhd (C35 \,\Box\, s.s6 \rightarrow C36)) \,\Box\, (C72 \rhd C42) \,\Box\, C45 \\
C06 = s.s12 \rightarrow (C09 \,\Box\, C10) & C90 = (w.B1 \rightarrow C20 \rhd s.s6 \rightarrow C63) \,\Box\, C56 \\
C07 = s.s8 \rightarrow s.s1 \rightarrow C14 & C91 = (w.B1 \rightarrow C49 \rhd s.s5 \rightarrow C63) \,\Box\, C23 \\
C53 = s.s9 \rightarrow s.s7 \rightarrow C59 & C49 = s.s5 \rightarrow C53 \,\Box\, s.s9 \rightarrow C60 \\
C09 = s.s1 \rightarrow s.s8 \rightarrow C14 & C96 = s.s6 \rightarrow (w.A1 \rightarrow C39 \rhd (C68 \,\Box\, C24)) \\
C60 = s.s5 \rightarrow s.s7 \rightarrow C59 & C13 = C81 \rhd (C19 \rhd (C85 \,\Box\, C84 \,\Box\, w.A2 \rightarrow C88)) \\
C11 = s.s4 \rightarrow s.s3 \rightarrow (C18 \,\Box\, C13) & C52 = (w.B1 \rightarrow C25 \rhd w.A2 \rightarrow s.s6 \rightarrow C63) \,\Box\, w.A2 \rightarrow C90 \\
C12 = s.s3 \rightarrow s.s4 \rightarrow (C18 \,\Box\, C13) & C08 = s.s1 \rightarrow s.s8 \rightarrow s.s2 \rightarrow (C11 \,\Box\, C12) \\
C51 = s.s5 \rightarrow (s.s6 \rightarrow C63) & C93 = s.s8 \rightarrow w.B1 \rightarrow s.s9 \rightarrow C48 \\
C14 = s.s2 \rightarrow (C11 \,\Box\, C12) & C55 = w.B1 \rightarrow s.s9 \rightarrow w.B2 \rightarrow s.s10 \rightarrow c.end3 \rightarrow C93 \\
C68 = w.B1 \rightarrow C31 \rhd C36 & C59 = c.end2 \rightarrow s.s14 \rightarrow w.B2 \rightarrow s.s10 \rightarrow c.end3 \rightarrow C93 \\
C57 = w.A2 \rightarrow s.s6 \rightarrow C63 & C17 = w.A1 \rightarrow ((C79 \rhd C87) \,\Box\, C15 \,\Box\, s.s5 \rightarrow C52) \\
C89 = w.B1 \rightarrow C32 \rhd C30 & C56 = s.s6 \rightarrow (w.B1 \rightarrow C53 \rhd C63) \\
C19 = w.A1 \rightarrow s.s5 \rightarrow C52 & C18 = C75 \rhd (C73 \rhd ((C70 \rhd C67) \,\Box\, C78 \,\Box\, C84)) \\
C20 = s.s6 \rightarrow C53 \,\Box\, s.s9 \rightarrow C62 & C10 = s.s8 \rightarrow s.s1 \rightarrow s.s2 \rightarrow (C11 \,\Box\, C12) \\
C21 = s.s5 \rightarrow w.B1 \rightarrow C53 & C28 = w.B1 \rightarrow (w.A1 \rightarrow C49 \,\Box\, C61) \\
C22 = w.A1 \rightarrow C40 \rhd C34 & C25 = w.A2 \rightarrow C20 \,\Box\, s.s9 \rightarrow w.A2 \rightarrow C62 \\
C85 = C70 \rhd C67 & C63 = s.s7 \rightarrow c.end2 \rightarrow s.s14 \rightarrow C55 \\
C24 = w.A1 \rightarrow C91 \rhd C37 & C23 = s.s5 \rightarrow (w.B1 \rightarrow C53 \rhd C63) \\
C62 = s.s6 \rightarrow s.s7 \rightarrow C59 & C65 = w.A1 \rightarrow (w.B1 \rightarrow C49 \,\Box\, s.s5 \rightarrow w.B1 \rightarrow C53) \\
C27 = C65 \,\Box\, C28 & C66 = w.A2 \rightarrow (C35 \,\Box\, s.s6 \rightarrow C36) \\
C61 = s.s9 \rightarrow w.A1 \rightarrow C60 & C67 = C66 \,\Box\, (w.A1 \rightarrow C87 \rhd w.A2 \rightarrow s.s6 \rightarrow C27) \\
C29 = s.s9 \rightarrow C34 & C15 = w.A2 \rightarrow (C89 \,\Box\, C82 \,\Box\, s.s5 \rightarrow C90) \\
C86 = w.A2 \rightarrow C42 & C69 = w.A1 \rightarrow C41 \rhd (w.A2 \rightarrow C43 \,\Box\, s.s9 \rightarrow w.A2 \rightarrow C34) \\
C84 = C17 \rhd (C83 \,\Box\, C86) & C70 = w.B1 \rightarrow (w.A2 \rightarrow C47 \,\Box\, C69 \,\Box\, C94) \\
C73 = w.A2 \rightarrow C45 & C71 = w.A2 \rightarrow C40 \,\Box\, s.s5 \rightarrow w.A2 \rightarrow C62 \\
C33 = w.A2 \rightarrow C34 & C72 = w.A1 \rightarrow ((w.B1 \rightarrow C32 \rhd C30) \,\Box\, s.s5 \rightarrow C90 \,\Box\, C82) \\
C34 = s.s6 \rightarrow w.A1 \rightarrow C60 & C32 = s.s5 \rightarrow C20 \,\Box\, s.s6 \rightarrow C49 \,\Box\, s.s9 \rightarrow C40 \\
C35 = w.A1 \rightarrow C30 \rhd s.s6 \rightarrow C27 & C30 = s.s5 \rightarrow s.s6 \rightarrow C63 \,\Box\, s.s6 \rightarrow s.s5 \rightarrow C63 \\
C36 = w.A1 \rightarrow s.s5 \rightarrow C63 \rhd C27 & C75 = w.A1 \rightarrow (C95 \,\Box\, s.s5 \rightarrow (C57 \rhd C52)) \\
C37 = w.B1 \rightarrow C61 \rhd C44 & C44 = C28 \,\Box\, w.A1 \rightarrow (w.B1 \rightarrow C49 \,\Box\, C21) \\
C38 = s.s6 \rightarrow C39 & C39 = s.s5 \rightarrow s.s7 \rightarrow c.end2 \rightarrow s.s14 \rightarrow C55 \\
C77 = w.A1 \rightarrow C32 \rhd C43 & C41 = w.A2 \rightarrow C32 \,\Box\, s.s5 \rightarrow C25 \,\Box\, s.s9 \rightarrow C71 \\
C40 = s.s5 \rightarrow C62 \,\Box\, s.s6 \rightarrow C60 & C42 = s.s6 \rightarrow C37 \,\Box\, (w.B1 \rightarrow C43 \rhd s.s6 \rightarrow C44) \\
C78 = w.A2 \rightarrow C88 & C80 = w.B1 \rightarrow (w.A2 \rightarrow C43 \,\Box\, s.s9 \rightarrow w.A2 \rightarrow C34) \\
C79 = w.B1 \rightarrow C41 & C81 = w.A2 \rightarrow ((w.A1 \rightarrow (C51 \,\Box\, C38) \rhd C45) \,\Box\, C96) \\
C43 = s.s6 \rightarrow C61 \,\Box\, C29 & C31 = (w.A1 \rightarrow C49 \rhd C61) \,\Box\, C61 \\
C76 = w.B1 \rightarrow C47 & C94 = s.s9 \rightarrow (w.A2 \rightarrow C22 \,\Box\, C46) \\
C45 = s.s6 \rightarrow (C68 \,\Box\, C24) & C95 = w.A2 \rightarrow (s.s6 \rightarrow C39 \,\Box\, C51) \rhd s.s5 \rightarrow C52 \\
C82 = s.s6 \rightarrow C91 & C92 = s.s15 \rightarrow s.s16 \rightarrow c.end1 \rightarrow \mathit{Skip}
\end{array}
\tag{6.5}
$$

Chapter 7

Modelling Empirical Studies

7.1 Introduction

In this chapter we consider extending the application scope of BPMN by investigating an alternative class of workflows. Specifically we propose a declarative model for *empirical studies*, and a method for transforming this model into BPMN so that one may leverage both BPMN's graphical expressiveness and the formal semantics which we have provided. We first provide the following definition of an empirical study and motivate our approach with an example.

Definition 7.1. ***Empirical Study.*** *An empirical study is a plan consisting of a series of scientific procedures interleaved with a set of observations performed over a period of time; these observations may be manually performed or automated.*

7.1.1 Motivating Example

A clinical trial is a typical long-running empirical study. In a clinical trial, observations are made as a series of case report form submissions detailing the health of the patient. Case reports are performed at specific times in the trial, and are interleaved with clinical interventions. The precise description of both reports and interventions are then recorded in a patient study calendar similar to the one shown in Figure 7.1 [CAL].

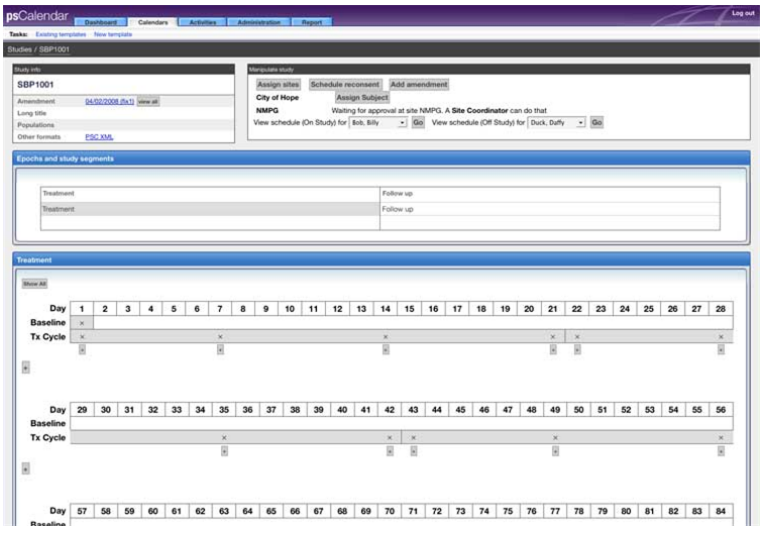


Figure 7.1: A screenshot of the patient study calendar

Specification. Currently clinical trial designers (or 'study planners') supply both schedule and procedural information about case reports and interventions either textually or by inputting textual information and selecting options about them on XML-based data entry forms [CHG+07]. An example of this is shown in Figure 7.2. However, the ordering constraints on observations and scientific procedures may be complex, and a precise specification of this information is time consuming and prone to error.

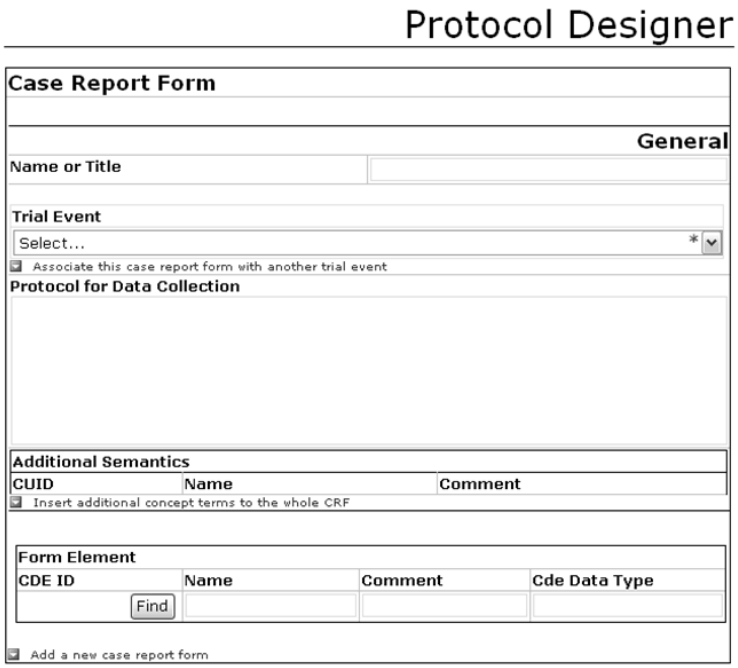


Figure 7.2: An XML-based data entry form

Verification. Table 7.1 shows the schedule, dosage and method of administration for each drug administration in an intervention of a clinical trial. During an intervention it is important that drugs are given to patients safely and effectively. Specifically, clinical interventions *must* satisfy a set of oncological safety principles [HSW95]. For example, the safety principle *Sequencing* ensures that an intervention "order(s) (essential) actions temporally for good effect and least harm". Here we may be interested in the following particular instance of this principle for the intervention schedule in Table 7.1.

> No more than one dosage of Gemcitabine (TG_G) may be given after the administration of Cyclophosphamide (EC_C) and before Epirubicin (EC_E).

Without a precise declarative specification of the schedule, it is difficult to guarantee the safety of the intervention.

Name	Code	Schedule	Dosage	Method
Cyclophosphamide	EC_C	every 14 to 20 days	$175mg/m^2$	Intravenous bolus
Epirubicin	EC_E	every 18 to 21 days	$2000mg/m^2$	Intravenous infusion
Paclitaxel	TG_T	every 5 to 10 days	$90mg/m^2$	Slow push/ fast drip
Gemcitabine	TG_G	within 10 days after TG	$600mg/m^2$	3 hours infusion

Table 7.1: Drug administration schedule for an example clinical trial

This requirement of precise specification suggests the use of a graphical modelling language, while the verification problem may be addressed by a language with a formal semantics and accompanying mechanical verification tools. We believe both these requirements can be addressed by specifying empirical schehdules as workflow instances in BPMN.

Figure 7.3 shows a BPMN diagram describing the clinical intervention described in Table 7.1. Note specific details (e.g. dosage) of each drug administration are recorded in the syntax of the subprocess element. The construction procedure for this will be described later in the chapter.

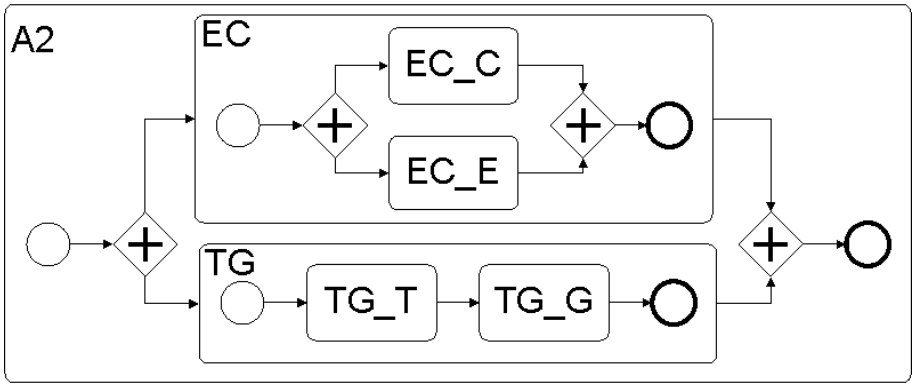


Figure 7.3: A set of clinical interventions

7.1.2 Contributions and Structure of Chapter

The main contributions of this chapter are as follows:

- We introduce a generic empirical studies model, `Empiricol`, an extension of the clinical trial workflow model defined in the CancerGrid project [CHG+07], for modelling empirical studies declaratively.
- We describe bidirectional transformation functions between `Empiricol` and BPMN. While the transformation from BPMN to `Empiricol` provides a medium for empirical studies to be specified graphically as workflows, transforming from `Empiricol` to BPMN allows graphical visualisation, simulation and verification.

The rest of this chapter is structured as follows. Section 7.2 describes the abstract syntax and the semantics of the empirical studies model `Empiricol`. Section 7.3 presents the bidirectional transformation functions between `Empiricol` and BPMN. Complete definitions of the transformation functions can be found in Appendices F and G. Section 7.4 discusses how this transformation allows simulation and automation of empirical studies (verification of empirical studies is discussed via a case study in Chapter 8). Section 7.5 discusses related work and summarises the contribution of this chapter.

7.2 Abstract Syntax of `Empiricol`

In this section we describe the syntax of the empirical studies model `Empiricol`. This model generalises the clinical trial workflow model defined in the CancerGrid project [CHG+07]. An empirical workflow is a list of parameterised generic activity interdependence sequence rules, with each rule modelling the dependency between the prerequisite and the dependent observations. Figure 7.4 shows the abstract syntax of `Empiricol`. `Empiricol` is a list of `EventSequencing` values, where each value records a sequence rule. The data type `EventSequencing` has four constructors, with each constructor recording a type of sequence rules for a particular part of the empirical workflow.

```
type Empiricol = [EventSequencing]
data EventSequencing = Start DptEvent | NStop PreAct |
                       Event ActivityId PreAct DptEvent Condition Condition DptAct [RepeatExp] Works
data ActivityId = START | NSTOP | Id String
```

Specifically an empirical workflow has exactly one starting sequence rule that models the start of the workflow; the starting sequence rule is recorded using the constructor `Start`. An empirical workflow has also exactly one ending sequence rule that models the end of the workflow; the ending sequence rule is recorded by the constructor `NStop`. An empirical workflow has one or more sequence rules defining activities in the workflow; each activity is captured by the constructor `Event`. Unless stated otherwise, we assume the term sequence rules for referring to activity-defining sequences, that is, those captured by `Event`.

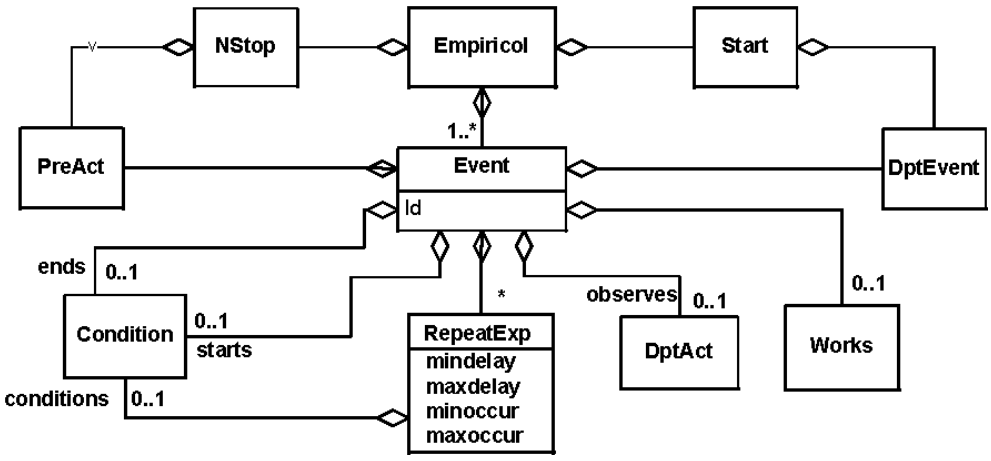


Figure 7.4: Abstract syntax of `Empiricol`

We now describe each argument encapsulated by `Event`:

- The `ActivityId` value records a unique name, for identifying the sequence rule. The data type `ActivityId` defines three constructors: `START` and `NSTOP` identify the starting and terminating sequences respectively, and `Id` records the unique name of a sequence rule in its `String` parameter.
- The `PreAct` value records the sequence rule's *prerequisite*; prerequisites are presented in Section 7.2.2.
- The `DptEvent` value records the sequence rule's *dependency* rule; dependencies are presented in Section 7.2.2.
- The fourth and fifth values of type `Condition` record the starting and terminating conditions of the sequence rule; conditions are presented in Section 7.2.5.
- The `DptAct` value records the observation group of the sequence rule; the construction of observation groups is presented in Section 7.2.1.
- The `[RepeatExp]` value is a list of observation repetitions defined by the sequence rule; repetitions is explained in Section 7.2.3.
- The `Works` value records the work group of the sequence rule; work groups are explained in Section 7.2.4.

7.2.1 Observation Group

We define a single observation by the tuple type `Observation`,

```
type Observation = (ActivityId,Duration,Duration,Repeat,Condition,ActType)
```

where the first component is a unique name from a set of names (`ActivityId`) distinct from those identifying sequence rules. Each observation specifies a delay; a delay has the type `Duration`. It is a double bounded range, of which the minimum and maximum are specified by the second and third components of `Observation`.

```
data Duration = UNBOUNDED | Dur String
```

Each duration records a string value, whose format is in accordance with the XML schema duration datatype [XS004, Section 3.2.6]. For example in a clinical trial, a follow-up case report is made between two and three months after all case reports associated with the end of the treatment have been carried out. Each observation may be repeated one or more times, and this is specified by the fourth component, `Repeat`. We explain the data type `Repeat` in Section 7.2.3. Each observation is conditioned by

the `Condition` value such that the observation must be terminated if the condition is satisfied. Each observation may either be performed manually or automated, and this information is recorded by the sixth component, `ActType` of `Observation`.

Each rule defines a workflow of observations, which is recorded by the data type `DptAct`.

```
type DptAct = Swf
data Swf = Choice [Swf] | Par [Swf] | Seq [Swf] | Single Acts | NoFlow
```

The workflows recorded by the datatype `DptAct` structurally conforms to a subclass of Kiepuszewski's *structure workflow model* [Kie02, Section 4.1.3]. Informally a structure workflow is a workflow where each XOR-split has a corresponding XOR-join, and each AND-split has a corresponding AND-join, and arbitrary cycles are not allowed. Our data type further restricts structure workflow such that each workflow is acyclic; we call this model the *acyclic structure workflow model*.

We provide the following definition of the acyclic workflow model based on the data type `Swf`:

Definition 7.2. ***Acyclic Structure Workflow.*** *An acyclic structure workflow (ASW) is inductively defined as follows:*

1. `NoFlow` *defines a null activity and is an ASW.*
2. *If* `a :: Acts` *is a single activity, then* `Single a :: Swf` *defines an ASW. Here* `a` *is both the initial and final activities. This workflow yields to completion when the activity identified by* `a` *has been executed.*
3. *Let* `ws :: [Swf]` *be a list of n ASWs where* $n \geq 1$*; their sequential composition* `Seq ws :: Swf` *defines an ASW. Given an observation group* `Seq ws`*, we describe its semantics inductively:*
   (a) *The initial ASW is* `head ws`*, its initial activity may be performed when all activities associated with the containing rule's prerequisites have been satisfied; prerequisites are explained in Section 7.2.2.*
   (b) *For any ASW* `ws!!i` *where* `i` *ranges over* `1..n-1`*, its initial activity may be performed when the final activity from* `ws!!i-1` *has completed execution.*
   (c) `Seq ws` *yields to completion when activities from ASW* `last ws` *have completed execution.*
4. *Let* `ws :: [Swf]` *be a list of n ASWs, an application of exclusive choice over them* `Choice ws :: Swf` *is an ASW. It yields to completion when the final activity from one of* `ws!!i` *where* `i` *ranges over* `0..n-1` *has completed execution.*
5. *Let* `ws :: [Swf]` *be a list of n ASWs; their interleaving* `Par ws :: Swf` *is an ASW. It yields to completion when the final activities from all of ASWs from the list have been completed.*
6. *Nothing else defines an ASW.*

Kiepuszewski showed that structure workflow models are deadlock-free [Kie02, Section 5.3].

Theorem 7.3. *Structured Workflow Models are deadlock-free.*

Since individual activity in an ASW may terminate if and only if either it has completed execution or the condition of the activity is satisfied, ASW is a subset of structure workflow, a trivial consequence of this is that ASW is also deadlock-free.

Corollary 7.4. *ASWs are deadlock-free.*

Each activity `a` in the ASW definition `Single a` is defined by the data type `Acts`,

```
data Acts = Dp Observation | Wk Work | Wu WorkSUnit
```

where observation groups restrict each `a` in `Single a` to be `Dp o` for some observation `o :: Observation`. The set of all observation groups is a subset of the AWF model and we define `DptAct` as a type synonym to `Swf`. Note that there are also two subsets of the ASW model, restricting activity `a` to the other two constructors of `Acts`. These will be described in Section 7.2.4.

7.2.2 Prerequisites and Dependencies

A sequence rule may only be evaluated if its prerequisite is satisfied. A prerequisite is a collection of names for identifying sequence rules and is organised as a tree. Prerequisites are recorded in the second argument of `Event` with the type `PreAct`. The completion of a sequence rule's evaluation triggers the evaluation of the rule's dependencies. Similar to a prerequisite, a dependency is a collection of names for identifying sequence rules and is also organised as a tree. Dependencies are recorded in the third argument with the type `DptEvent`. Both `PreAct` and `DptEvent` are defined as follows.

```
type PreAct = Tree
type DptEvent = Tree
data Tree = All [Tree] | OneOf [Tree] | Leaf ActivityId
```

Both `DptEvent` and `PreAct` are specialisations of the data type `Tree`; a `Tree` value records a tree structure. Here we provide the inductive definition of construction and satisfiability of `Tree`; while construction is the same across different specialisations, satisfiability depends on the semantics of the specific specialisation.

Definition 7.5. ***Tree.*** *The construction and satisfiability of a tree is inductively defined as follows:*

1. *If* `id :: ActivityId` *is a unique name that identifies a particular sequence rule, then* `Leaf id :: Tree` *defines a tree. The tree is satisfied if and only if the sequence rule* `id` *is satisfied.*
2. *If* `ts :: [Tree]` *is a list of trees, then* `All ts :: Tree` *is a tree and is satisfied if and only if all trees* `ts` *are satisfied.*
3. *If* `ts :: [Tree]` *is a list of trees, then* `OneOf ts :: Tree` *is a tree and is satisfied if and only if one of the trees* `ts` *is satisfied.*
4. *Nothing else defines a tree.*

We now describe the satisfiability of trees. A prerequisite tree identifies the preceding sequence rules that must have been evaluated before its containing sequence rule may be evaluated. In particular, the prerequisite tree `Leaf i` is satisfied if and only if the preceding sequence rule identified by `i` has been evaluated. A dependency tree, on the other hand, identifies the succeeding sequence rules, each of whose prerequisite identifies the containing sequence rule. In particular, the dependency tree `Leaf i` is satisfied if and only if the succeeding sequence rule identified by `i` is triggered.

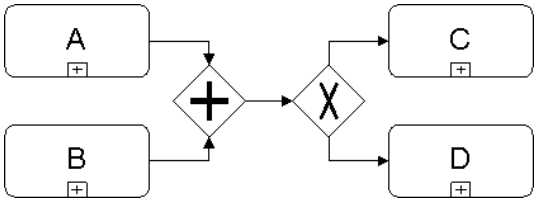


Figure 7.5: An illustration of prerequisite and dependence

For example, Figure 7.5 shows a partial BPMN diagram modelling a part of an empirical workflow, each (collapsed) subprocess element modelling a distinct sequence rule. In particular, both subprocesses C and D have the prerequisite tree `All [A,B]` where `A` and `B` are `Leaf` values identifying the sequence rules modelled by subprocesses A and B. Conversely, both subprocesses A and B have the dependency tree `OneOf [C,D]` where `C` and `D` are `Leaf` values identifying the sequence rules modelled by subprocesses C and D.

7.2.3 Repetition

Each sequence rule also defines a (possibly empty) list of *repeat* clauses; this is recorded by the sixth argument of `Event`, and has the type `[RepeatEx]`. These clauses are evaluated sequentially over the list after one default iteration of performing the rule's observations.

```
type RepeatExp = (Duration,Duration,Repeat,Repeat,Condition)
type Repeat = Int
```

Each clause has the type `RepeatExp` and contains a terminating condition specified by the fifth component with the type `Condition`. The construction and the evaluation of conditions is described in Section 7.2.5. Each clause also records the minimum and maximum numbers of repetitions, specified by the third and fourth components of `RepeatExp` respectively, and a delay between a minimal period of duration and a maximal period of duration, by the first and second components of `RepeatExp` respectively. Specifically while the condition of the clause holds and the number of repetitions still lies in between minimum and maximum, all observations defined in the containing sequence rule are repeated with the delay specified by the clause. We now define the evaluation of a list of repeat clauses.

Definition 7.6. ***Evaluation of Repeat Clause List.*** *Given a list of repeat clauses* `res :: [RepeatExp]` *defined in some sequence rule* `sr`*, its evaluation is inductively defined as follows:*

1. *If* `res` *is empty, it terminates.*
2. *The initial repeat clause is* `head res`*. It is evaluated after the default iteration of* `sr`*'s observations have completed, or the condition defined by the fifth component of* `head res` *is satisfied.*
3. *A given repeat clause terminates if either its maximum number of repetitions has been reached, or its minimum number of repetitions has been reached and the terminating condition is satisfied.*
4. *For any clause* `res!!n` *where n ranges over* `[1..(length res - 1)]`*, it may be evaluated after the evaluation of the clause* `res!!(n-1)` *terminates.*
5. `res` *terminates when* `last res` *terminates.*

For example, the follow up sequence rule of a clinical trial might specify that follow up case report should be made every three months for three times after the initial case reports have been made, after which case reportings should be performed every six months for four times.

7.2.4 Work Group

Each sequence rule might include work units, recorded by the last argument of the constructor `Event`. Each work unit represents an empirical procedure such as administering a medical treatment on a patient in a clinical trial. In each sequence rule, the procedures defined by the rule's work units are interleaved with the rule's observations. Each collection of work units is a *procedure workflow* defined by the data type `Works`.

```
type Works = Swf
```

This is a specialisation of the ASW model. In particular, the specialisation restricts each `a` in `Single a` in a procedure workflow to be `Wk w` for some procedure `w :: Work`. We present the definition of `Work`.

```
data Work = WkS WorkMUnit | WkM WorkMBlock
type WorkMBlock = (WorkId,WorkBlock,Duration,Repeat)
type WorkMUnit = (WorkId,Procedure,Duration,Duration,Repeat)
```

The data type `Work` defines two constructors, `WkS` and `WkM`. A `WkS w` value records a single repeatable procedure, while a `WkM w` value records a repeatable workflow of procedures. We first consider single repeatable procedures, whose definition is provided as follows.

Definition 7.7. ***Single Repeatable Procedure.*** *A single repeatable procedure (SRP) is a quintuple* (I, P, M, N, R) *such that:*

I *uniquely identifies the procedure;*

P *records the definition of the procedure;*

M *records the minimum duration before the procedure may be executed;*

N *records the maximum duration after which the procedure must be terminated; and*

R records the repeat clause, which defines if and how the procedure should be repeated.

Each SRP is defined by a tuple type `WorkMUnit`, and its procedural definition is recorded using the data type `Procedure`. A `Procedure` value is either the null constructor `DetailHidden` representing unknown procedural information or one of the other constructors each capturing a different type of procedure.

```
data Procedure = DetailHidden | ProcedureA Treatment | ProcedureB ...
data Treatment = Treatment Name Quantity Method
```

In our current definition the constructor `ProcedureA t` stores a `Treatment` value recording the name of the drug, its dosage and method of administration. Here the ellipsis `...` indicates the data type `Procedure` may be extended to cater for other types of empirical studies.

We now consider repeatable workflow of procedures, whose definition is provided as follows:

Definition 7.8. ***Repeatable Workflow of Procedures.*** *A repeatable workflow of procedures (RWP) is a quadruple* (I, W, M, R) *such that:*

I uniquely identifies the procedure;

W records an ASW defining a workflow of procedures;

M records the minimum duration before the workflow may be triggered; and

R records the repeat clause, which defines if and how the workflow should be repeated.

Each RWP is defined by a tuple type `WorkMBlock`. In particular its workflow of procedures is recorded by the type `WorkBlock`, and is a specialisation of `Swf`, restricting each `a` in `Single a` in the workflow to be `Wu w` for some single non-repeatable procedure `w :: WorkSUnit`.

```
type WorkBlock = Swf
type WorkSUnit = (WorkId,Procedure,Duration,Duration)
```

The definition of a single non-repeatable procedure is provided as follows:

Definition 7.9. ***Single Non-Repeatable Procedure.*** *A single non-repeatable procedure (SNP) is a quadruple* (I, P, M, N) *such that:*

I uniquely identifies the procedure;

P records the definition of the procedure;

M records the minimum duration before the procedure may be executed; and

N records the maximum duration after which the procedure must be terminated.

7.2.5 Conditions

The third and fourth arguments of a sequence rule are two conditional statements, each of type `Condition`. The third argument defines the condition for triggering the evaluation the sequence rule, and the fourth argument defines the condition for terminating the evaluation of the sequence rule.

Our definition of `Condition` extends the *skip logic* used in the CancerGrid Workflow Model [CHG+07]. Specifically, its syntax captures expressions in conjunctive normal form.

```
data Condition = NoCond| Ands [Ors]
data Ors = Ors [SCondition]
type SCondition = (Range,Property)
data Range = Bound RangeBound RangeBound | Emu [String]
data RangeBound = NoBound | Abdate Duration | Abdec Float | Abint Int |
                  Rldate Property Duration | Rldec Property Float | Rlint Property Int
```

Each condition `c :: Condition` yields a Boolean value and is either true by default, denoted by the constructor `NoCond`, or defined as the conjunction of clauses, each of which is a disjunction of Boolean conditions, of type `SCondition`. The type `SCondition` is satisfied if the value of the specified property (typed `Property`) falls into the specified range (typed `Range`) at the time of evaluation. A `Property`

value is a name that identifies a particular property in the domain of the empirical study. A `Range` is either an enumeration of `String` values via the constructor `Emu`, or an interval of two numeric values via the constructor `Range` over two arguments of type `RangeBound`, which may be absolute or relative to a property.

7.3 Transformation

In this section we describe the bidirectional transformation between empirical workflows of type `Empiricol` and their corresponding BPMN diagrams. Specifically we have implemented a total function transforming `Empiricol` to `BPMN`, the data type for recording the abstract syntax of BPMN, and a partial function, transforming `BPMN` to `Empiricol`.

```
eTob :: Empiricol -> BPMN      bToe :: BPMN -> Empiricol
```

We first describe the transformation from a single sequence rule to a BPMN subprocess, by considering the transformation over each of the components that makes up the tuple of a sequence rule. We then describe the composition of sequence rules to model a complete observation workflow. Full definitions of the transformations can be found in Appendices F and G.

7.3.1 Activities

This section considers the transformation of activities (`Single`) defined in an ASW. The transformation of ASW is presented in Section 7.3.2.

7.3.1.1 Observation

We consider the transformation of an observation. Each observation is graphically represented by either a BPMN multiple instance task (`Miseq`), or an atomic task (`Task`) if the repeat value is one. In both cases, the minimum and maximum durations of an observation are mapped to the `Range` value of the BPMN element and both the observation identifier and its `ActType` value are mapped to the name of the task element via the function `idToTName`. For example Observation `A` with the `ActType` value `Manual` is translated to the `TaskName` value `A_Manual`. Conversely the function `idsTname` takes a task name and returns the corresponding observation identifier and `ActType` value.

```
idToTn :: ActvityId -> ActType -> TaskName
idsTname :: TaskName -> (ActivityId,ActType)
```

The condition of the observation may be translated into a rule exception flow, that is, an intermediate rule (`Irule`) element attached to the task element.

We have implemented two functions to assist the transformation between an observation and its corresponding task element. In particular, the function `dptact` takes a task element and returns a `Observation` value recording the observation modelled by that element. The function `mkdpt` takes a sequence flow (`Seqflow`) and an observation, and returns a pair where the first component is either an atomic task element or a multiple instance task element, and the second component is a fresh sequence flow; a sequence flow is *fresh* if it has not been declared so far in the transformation process; a sequence flow is fresh with respect to a list of elements if it has not been declared in that list.

```
dptact :: Element -> Observation
mkdpt :: Observation -> Seqflow -> (Element,Seqflow)
```

7.3.1.2 Procedure

We consider the transformation of a SRP, a SNP and a RWP. A SRP is modelled by either a multiple instance task or an atomic task if the repeat value is one. In both cases, the minimum and maximum durations of a SRP are mapped to the `Range` value of the BPMN element and the procedure identifier is mapped to the name of the task element via the function `wkToTk`. Conversely the function `tkTowk` takes a task name and returns the corresponding procedure identifier.

```
wkToTk :: WorkId -> TaskName      tkTowk :: TaskName -> WorkId
```

A SNP is modelled by an atomic task as it is not repeatable. Similar to SRP, the minimum and maximum durations of a SNP are mapped to the `Range` value of the BPMN element and the procedure identifier is mapped to the name of the task element via the function `wkToTk`.

A RWP is modelled by either a multiple instance subprocess or an atomic subprocess if the repeat value is one. Figure 7.6 shows an example BPMN subprocess modelling a RWP. Note that the RWP's

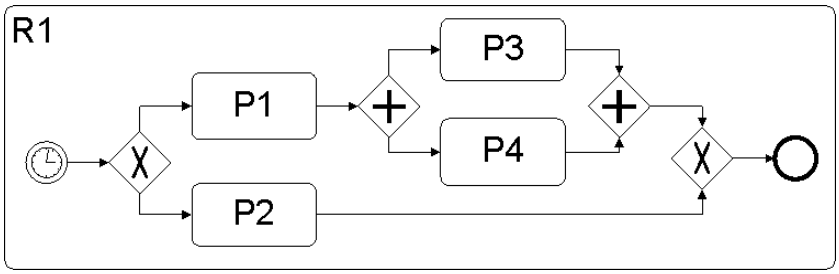


Figure 7.6: A BPMN subprocess element modelling a RWP

procedure workflow, denoted by W in Definition 7.8, is modelled by the BPMN elements directly contained in the subprocess $R1$. In Figure 7.6 the elements contained in the subprocess model the procedure workflow `Choice [Seq [P1,Par [P3,P4]],P2]`. The transformation of the procedure workflow, which is an ASW, is presented in Section 7.3.2. The minimum duration of the RWP (M in Definition 7.8) is modelled by the start timer event contained in the subprocess.

We implement four functions for the transformation between a procedure and its corresponding BPMN task. The function `wk` takes a BPMN task and returns a `Work` value encapsulating either a SRP or a RWP, while function `wb` takes a BPMN task and returns a `WorkSUnit` value recording the corresponding SNP.

```
wk :: Element -> Work       wb :: Element -> WorkSUnit
```

The function `mkwk` takes either a SRP (`WorkMUnit`) or a SNP (`WorkSUnit`) and a sequence flow (`Seqflow`) and returns a pair where the first component is either an atomic task or a multiple instance task and the second component is a fresh sequence flow. The function `mkwb` takes a RWP (`WorkMBlock`) and a sequence flow, and returns a subprocess modelling the RWP and a fresh sequence flow.

```
mkwk :: Either WorkMUnit WorkSUnit -> Seqflow -> (Element,Seqflow)
mkwb :: WorkMBlock -> Seqflow -> (Element,Seqflow)
```

7.3.2 Acyclic Structured Workflows

Each sequence rule defines one observation workflow and at most one procedure workflow specifying the work units of the sequence rule. Each of these workflows defines an ASW, whose syntax has been presented in Section 7.2.1. While the previous section described the transformation between individual workflow activities and their respective BPMN constructs, this section describes the transformation of ASWs.

An ASW specifies the control flow of a collection of activities. Figure 7.6 showed a BPMN subprocess modelling a RWP, which is a collection of procedures. Figure 7.7, on the other hand, shows a BPMN subprocess modelling a procedure workflow, where the subprocess $R1$ in Figure 7.6 is shown as a collapsed subprocess in Figure 7.7. Specifically the figure depicts a BPMN subprocess modelling the procedure `Par[Seq[R1,Choice[R3,R4],R2]]`.

We implement two functions `swfsub` and `swfs` for the transformation between ASW and BPMN. We first consider `swfsub`.

The function `swfsub` takes a `WfType` value, for identifying whether the ASW specifies an observation workflow, a procedure workflow or a RWP, a sequence flow, a `Struct` value, and returns a BPMN subprocess modelling that ASW and a fresh sequence flow.

```
type Struct = Either (Swf,ActivityId) (String,Swf,Duration,Repeat)
data WfType = Dpt | Wks | Wbs
```

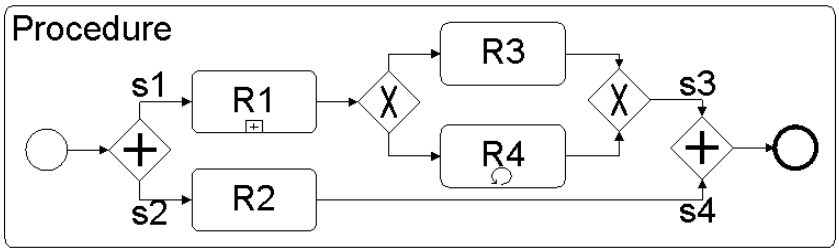


Figure 7.7: A BPMN subprocess element modelling a procedure workflow

```
swfsub :: WfType -> Seqflow -> Struct -> (Element,Seqflow)
swfsub w s d = (Compound a (start:end:sf), sflow e)
```

The `Struct` value contains either a pair or a quadruple of values. If it is a pair of an ASW (`Swf`) and the sequence rule's name (`ActivityId`), then the ASW specifies either an observation workflow or a procedure workflow and the rule's name is used for constructing the name of the subprocess. If it is a quadruple of `String` value, an ASW, a `Duration` and a `Repeat` value, then the quadruple corresponds to a RWP. Specifically `swfsub` returns a subprocess element, and a fresh sequence flow. The subprocess element has the form `Compound a (start:end:sf)`, in which `start` and `end` denotes the start and end events contained in that subprocess; `sf` is a list of elements defined by the expression `extswf (sflow s) wf`, where `wf` is the ASW and `sflow s` is the outgoing sequence flow of the start event `start`. The function `extswf` is defined as follows:

```
extswf :: Seqflow -> Swf -> ([Element],Seqflow)
extswf s NoFlow = ([],s)
extswf s (Choice ws) = extswfm Xgate s ws
extswf s (Par ws) = extswfm Agate s ws
extswf s (Seq ws) = ((par concat last).unzip.(swfm id s)) ws
extswf s (Single g) = sg s g
```

Specifically given a sequence flow and an ASW, the function returns a pair, where the first component is a list of BPMN elements modelling the ASW and the second component is a fresh sequence flow with respect to that list of elements. Note that the value `sg s g` evaluates the activity using the functions defined in Section 7.3.1. The function `extswf` is defined in terms of two other functions: `swfm` and `extswfm`. We first consider `swfm`, which is defined as follows:

```
swfm :: (Seqflow -> Seqflow) -> Seqflow -> [Swf] -> [([Element],Seqflow)]
swfm f _ [] = []
swfm f s (d:ds) = (g,t):(swfm f (f t) ds) where (g,t) = extswf s d
```

It takes a function `f` of type `Seqflow -> Seqflow`, a sequence flow and a list of ASWs, and returns a list of pairs, where each pair consists of a list of BPMN elements and a fresh sequence flow from the list. Essentially, function `swfm` applies `extswf` to every ASW from the list (third argument) and for every ASW in that applies `f` to the sequence flow in the second argument.

If the list of ASWs are sequentially composed, then the function `f = id` is the identity function. This is because for every ASW evaluated, the outermost outgoing sequence flow of the BPMN construct modelling the ASW must be the outermost incoming sequence flow of the BPMN construct modelling the subsequent ASW; Figure 7.8 shows a partial BPMN process specifying the `Seq` construct. In this example the outermost sequence flows are denoted by $s1$ and $s2$.

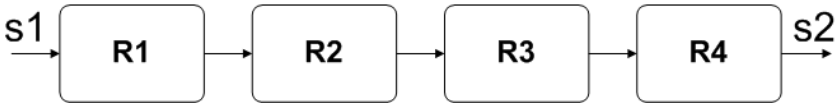


Figure 7.8: A BPMN model of the `Seq` construct

Conversely if the list of ASWs are interleaved or the exclusively chosen from, then function `f = sflow`, which takes a sequence flow and creates a fresh sequence flow. This is because each ASW in the list is executed on a separate flow and requires a fresh sequence flow.

We now consider `extswfm`, which is defined as follows:

```
extswfm :: Type -> Seqflow -> [Swf] -> ([Element],Seqflow)
extswfm t s wf = ((g:j:(concat fe)),(head.outs.atom) j)
  where (g,_) = node (Left [s]) (Left (concatMap (ins.atom.head) fe)) "" ("Gate"++(si s)) t
        (fe,fs) = unzip (swfm sflow (sflow s) wf)
        (j,_) = node (Left fs) (Right 1) "" ("Gate"++((si.head) fs)) t
```

It takes a BPMN `Type` value, a sequence flow and a list of ASWs, and returns a pair of a list of BPMN elements and a fresh sequence flow. The `Type` value either is `Agate`, the type of a parallel gateway, if the list of ASWs are interleaved, or `Xgate`, the type of a data-based XOR gateway, if the exclusive choice is applied to the list of ASWs. Specifically, function `extswfm` returns a list of BPMN elements and a fresh sequence flow such that the list contains a (AND/XOR) split gateway, a (AND/XOR) join gateway, and a list of BPMN elements modelling the list of ASWs. In particular, the outgoing sequence flows of the split gateway correspond to the outermost incoming sequence flows of the BPMN elements, each of which models an ASW from the list, while the incoming sequence flows of the join gateway correspond to their outermost outgoing sequence flows. Figure 7.7 shows a BPMN representation of a `Par` value. In particular, $s1$ and $s2$ are the outgoing sequence flows of the AND split gateway and are also the outermost incoming sequence flows of ASWs `Seq [R1,Choice[R3,R4]]` and `R2`, while $s3$ and $s4$ are the incoming sequence flows of the AND join gateway and are also the outermost outgoing sequence flows of the ASWs.

We now consider the transformation from BPMN to ASW. The function `swfs` takes a `WfType` value and a BPMN subprocess, and returns an ASW modelled by that BPMN subprocess.

```
swfs :: WfType -> Element -> Swf
swfs w e = maybe NoFlow ((gswf w es).head.outs.atom) (find (isstart.etype.atom) es)
  where es = embedding e
```

Specifically, it locates the start event directly contained in the subprocess. If this element exists, then by definition of ASW it is unique in the subprocess, and the function evaluates the expression `gswf w es s` where `s` is the outgoing sequence flow of the start element. If the subprocess does not contain a start event, the function returns `NoFlow`, signifying an empty ASW. The function `gswf` is defined as follows:

```
gswf :: WfType -> [Element] -> Seqflow -> Swf
gswf w es s = maybe NoFlow (\x -> Seq (fst (seqswf w s es))) (findsucceed0 es s)
```

Specifically the function `gswf` checks whether there exists a directed successor of the start event. If so, the function applies `Seq` over the expression `fst (seqswf w s es)`. This expression uses the function `seqswf`, which essentially applies the function `seqswfs1` over the same argument as `gswf` with the addition of the element `e`, the direct successor of the start event. The definition of `seqswfs1` is provided as follows:

```
seqswfs1 :: WfType -> Seqflow -> [Element] -> Element -> [(Swf,Maybe Seqflow)]
seqswfs1 w s es e
  | isxs e = [(Choice f,n)]++cs
  | isas e = [(Par f,n)]++cs
  | disjs [isxj,isaj] e = [(NoFlow,(listToMaybe.outs.atom) e)]
  | (inb w) e =
      [(single w e,(listToMaybe.outs.atom) e)]++
      (seqswfs w ((head.outs.atom) e) es)
  | otherwise = []
  where (f,n) = swfmult w es ((outs.atom) e)
        cs = maybe [] (\s -> seqswfs w s es) n
```

If `e` is a (data-based) XOR split gateway, the function returns the list `[(Choice f,n)]++cs`, and if `e` is an AND split gateway, the function returns the list `[(Par f,n)]++cs`. The value `f` is the list of ASWs modelled by the succeeding elements of the gateway, `n` is the outermost outgoing sequence flow of the BPMN elements that model either `Par f` or `Choice f`, and `cs` contains subsequent ASWs in the `Seq` defined by `gswf`.

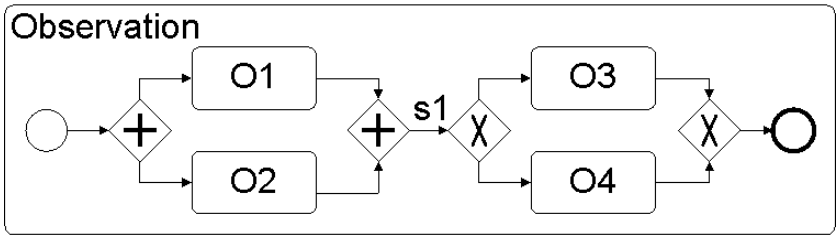


Figure 7.9: A BPMN subprocess modelling an observation workflow

Figure 7.9 shows a BPMN subprocess modelling an observation workflow, where the sequence flow $s1$ is the outermost outgoing sequence flow of the interleaving of tasks $O1$ and $O2$, capturing the ASW `Par [O1,O2]`. The calculation of `n` allows the function to locate the subsequent BPMN construct, which is the exclusive choice of tasks $O3$ and $O4$, capturing the ASW `Choice [O3,O4]`. The complete ASW hence is `Seq [Par [O1,O2],Choice [O3,O4]]`, where the application of `Seq` is defined by `gswf`.

7.3.3 Repeat Clauses

This section describes the transformation between a list of repeat clauses, each of type `RepeatExp`, and its corresponding BPMN subprocess. A single repeat clause specifies how many repetitions an observation workflow should perform. Since an observation workflow is modelled by a BPMN subprocess, a repeat clause can be naturally modelled as a sequential multiple instance subprocess.

Figure 7.10(a) shows a BPMN subprocess modelling a single repeat clause. According to the semantics of a repeat clause, a repeat clause in a sequence rule repeats all observations defined in that rule; the number of repetitions from a clause ranges between a minimum and a maximum values, and there is a delay, ranging between a minimum and a maximum durations, before a repetition can start.

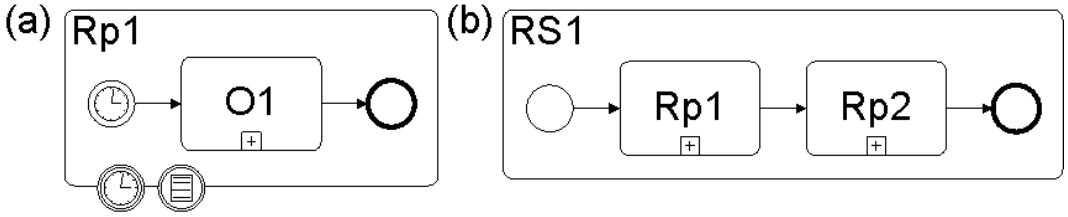


Figure 7.10: (a) A BPMN subprocess modelling a repeat clause and (b) a BPMN subprocess modelling a list of two repeat clauses

Specifically we model repeated observations as a subprocess according to the transformation of observation workflows in Section 7.3.2. Each repeat clause is modelled as a sequential multiple instance subprocess, such that it has one incoming and one outgoing sequence flow, denoted as the outermost incoming and outgoing sequence flows respectively. The type of a sequential multiple instance subprocess element, `Miseqs`, takes an integer value, which specifies the maximum number of repetitions, while the minimum duration is modelled by the start timer event. Both the terminating condition and the maximum duration are modelled by attaching intermediate events to the subprocess, thereby creating exception flows that model the behaviour.

Below shows an example of how to model the clause's repetition range and terminating condition,

```
rep = Miseqs "Treatment" (Fix 2) RepeatB BPMN.All
cond = And [cond01,cond02]
cond01 = Sgl (Lt (Pty (Nm "LoopCounter" InT)) (Rge (Inv 5 10)))
cond02 = Sgl (Eq (Pty (Nm "Abnormal Blood Count" EmT))
               (Rge (Emv ["VHigh","VLow"])))
```

where the element type `rep` records information about a repetitive observation during a study of an

effect of a medical intervention. The condition `cond` is a conjunction of Conditions `cond01` and `cond02`: Condition `cond01` is satisfied if the current number of repetitions is less than a value chosen over the closed interval `[5..10]`, and Condition `cond02` is satisfied if the abnormal blood count of the patient is either very high or very low.

A list of repeat clauses is therefore transformed iteratively over each clause starting from head of the list. Figure 7.10(b) shows a BPMN subprocess modelling a list of two repeat clauses, where individual repeat clauses are shown as collapsed subprocesses.

7.3.4 `Empiricol` Workflows

We have so far described the transformation for repeat clauses, observation and procedure workflows, constituting the sixth, seventh and eighth components of a sequence rule. In this section we consider the transformation of a sequence rule and an empirical workflow.

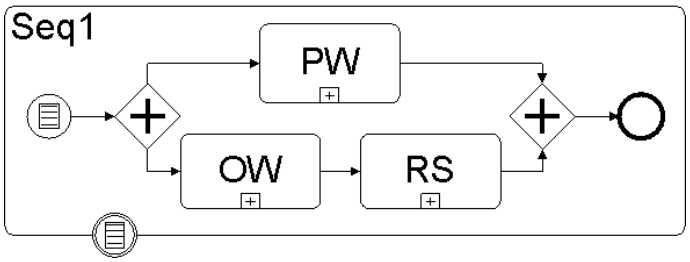


Figure 7.11: A BPMN subprocess representing a single sequence rule

Figure 7.11 shows a BPMN subprocess modelling a single sequence rule. The subprocess directly contains three other subprocesses in collapsed view that model observations, procedures and repeat clauses of the sequence rule. A sequence rule first performs its observations once, which are modelled by the subprocess OW, after which the list of repeat clauses, which are modelled by the subprocess RS, is evaluated and performed. As explained in Section 7.2, procedures are interleaved with observations, we therefore use an AND split gateway to initialise both observations and procedures. We do not place explicit control flow constraints on the interleaving of the procedure and observation workflows as the particular order of execution is determined by their timing information. If no procedure is defined in the sequence rule, the corresponding subprocess will not have parallel gateways and will be represented by a sequential composition of OW and RS subprocess elements.

Finally the start rule event and the attached intermediate rule event model the starting and the terminating conditions of the sequence rule. For example, in a clinical trial a sequence rule may be defined for administering a new drug on a patient, where the insulin level of the patient is to be monitored before and during the treatment. The following two conditions can be specified,

```
start = Ands [Ors [(Emu ["Normal"],"Insulin Levels")]]
terminating = Ands [Ors [(Emu ["Low"],"Insulin Levels")]]
```

where `start` is modelled by the start rule event and `terminating` is modelled by the attached intermediate rule event.

An empirical workflow, of type `Empiricol`, is a list of sequence rules connected according to each rule's prerequisite and dependency. Figure 7.12 shows a single empirical workflow modelled as a BPMN pool.

Note that the order of execution of individual sequence rules also conforms an ASW. We first consider the transformation of sequence rules to a BPMN process.

7.3.4.1 From Empiricol Workflow to BPMN

We define the function `flow`, which takes an empirical workflow, a list of subprocess elements, each modelling a sequence rule in the workflow, and a sequence flow fresh from the list of subprocesses, and returns a list of all elements, such that these elements are directly contained in a BPMN pool that models the empirical workflow.

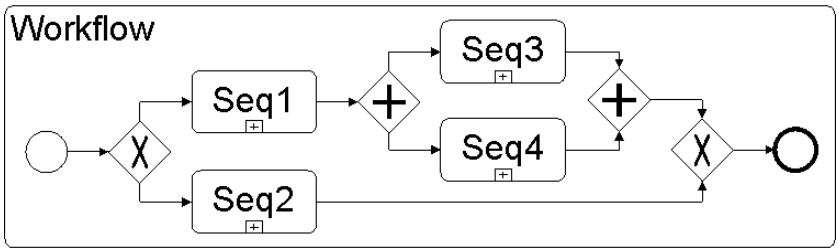


Figure 7.12: A BPMN process describing an empirical workflow

```
flow :: Empiricol -> [Element] -> Seqflow -> [Element]
flow ws es s = pres ws ne (sflow ns) where (ne,ns) = dpts ws es s
```

The function first evaluates the expression `dpts ws es s`, where `dpts` is defined as follows:

```
dpts :: Empiricol -> [Element] -> Seqflow -> ([Element],Seqflow)
dpts [] es s = (es,s)
dpts ((NStop _):ws) es s = dpts ws es s
dpts (w:ws) es s = dpts ws (update es (a:nss)) ts
  where (nss,ts) = btree Dpts es b ((normtree.getDe) w)
        (a,b) = cg Dpts (Left w) es (Nothing,Just s)
```

This function takes a list of sequence rules, a list of subprocesses and a sequence flow, and connects each BPMN subprocess to its direct successors by evaluating the dependency tree of the sequence rule that the subprocess models. Specifically for each non terminating sequence rule, that is, each rule which is not a `NStop` value, the function returns a list of subprocess elements `nss` such that they are connected by sequence flows according to the rule's dependency and a fresh sequence flow. This list of subprocesses is then used to update the list of BPMN elements `es` before proceeding to the next sequence rule. The calculation of dependency is defined by the function `btree Dpts`, which is applied over the outgoing sequence flow of the subprocess that models the sequence rule, and the rule's dependency. `Dpts` is a null constructor of type `TreeType`, which specifies the given `Tree` value to be a rule's dependency. The functional composition `normtree.getDe` selects and normalises the rule's dependency. Normalisation ensures constructors `All` and `OneOf` are never applied over a singleton list.

The function `btree` is defined as follows:

```
btree :: TreeType -> [Element] -> Seqflow -> Tree -> ([Element],Seqflow)
btree y es s t =
  if (length.gtree) t > 1 then ((g:(concat fs)),(sflow.newest) ks)
  else (par (:[]) sflow) (cg y (Right (treeid t)) es (sf y (treeid t) s))
  where (fs,ks) = unzip (btree2 y es (sflow s) (gtree t))
        g | y == Pres = fst (gate y t s (concatMap (outs.atom.head) fs))
          | y == Dpts = fst (gate y t s (concatMap (ins.atom.head) fs))
```

Specifically if the dependency tree is a `Leaf` value (single leaf node), the function returns a singleton list and a sequence flow such that the list contains the element, identified by the `ActivityId` value encapsulated by the leaf node, the sequence flow is an incoming sequence flow of the element in the list.

Otherwise if the dependency tree is either a `OneOf` or an `All` value, the function applies the function `btree2` over the list of dependency trees recorded by the constructor. The function `btree2` is defined as follows:

```
btree2 :: TreeType -> [Element] -> Seqflow -> [Tree] -> [([Element],Seqflow)]
btree2 _ _ _ [] = []
btree2 y es s (t:ts) =
  if (length.gtree) t > 1 || y == Dpts then ((n,u):(btree2 y (update es n) u ts))
  else (([(fromJust.(findele (Right (treeid t)))) es],s):(btree2 y es s ts))
  where (n,u) = btree y es s t
```

This function takes a specification type (`TreeType`), the list of BPMN elements constructed so far, a sequence flow and a list of trees, and returns a list of pairs, where each pair is a list of elements and a sequence flow such that the elements model one of the trees and the sequence flow is fresh from the list of elements. Note that each tree is evaluated by applying the function `btree`. Specifically each pair

represents a branch and these branches are connected via corresponding split and join gateways by the function `btree`.

After constructing parts of the BPMN pool that model each sequence rule's dependency, the function `flow` evaluates the expression `pres ws ne (sflow ns)`, where `ne` from the expression is the list of subprocesses updated with dependency information, `sflow ns` is a sequence flow fresh from the list of subprocesses, and function `pres` is defined as follows:

```
pres :: Empiricol -> [Element] -> Seqflow -> [Element]
pres [] es s = es
pres ((Empiricol.Start _):ws) es s = pres ws es s
pres (w:ws) es s
  | (length.gtree.getPr) w == 1 = pres ws es s
  | otherwise = pres ws (update es (a:mss)) rs
  where (mss,rs) = btree Pres es ((head.ins.atom) a) ((normtree.getPr) w)
        a = setflows ((fromJust.(findele (Left w))) es) [s] []
```

This function takes a list of sequence rules, a list of BPMN elements and a sequence flow, and connects BPMN elements using sequence flows according to the prerequisite of the given sequence rules. Specifically for each sequence rule which is not a `Start` value, it applies the function `btree Pres` over the incoming sequence flow of the subprocess, that models the rule, and the rule's prerequisite tree. `Pres` is a null constructor of type `TreeType`, which specifies the given `Tree` value to be a rule's prerequisite.

7.3.4.2 From BPMN to Empiricol Workflow

We now consider the transformation from BPMN to empirical workflow. Since the transformations from BPMN subprocess to a sequence rule's observation workflow, repeat clauses and procedure workflow have been considered, this section focuses on the construction of prerequisite and dependency trees.

We provide the following generic function `tree` for constructing both types of trees:

```
tree :: TreeType -> [Element] -> Seqflow -> Tree
tree t (e:es) s = ((\x -> g x es s).fromJust.(find f)) es
  where (f,g) | t == Pres = ((elem s).allouts,preap)
              | t == Dpts = ((elem s).ins.atom,dptap)
```

Specifically the function takes a `TreeType` value, the list of BPMN elements modelling the empirical workflow and a sequence flow, and returns one of the following:

- a prerequisite tree if and only if `TreeType` value is `Pres` and the sequence flow is the incoming sequence flow of a subprocess modelling the sequence rule of which the prerequisite tree is defined; or
- a dependence tree if and only if `TreeType` value is `Dpts` and the sequence flow is the outgoing sequence flow or an exception flow of a subprocess modelling the sequence rule of which the dependency tree is defined.

To construct a prerequisite tree, the function applies `preap` over the element, which has `s` as one of its outgoing sequence and exception flows (the direct predecessor of the subprocess that models the sequence rule), the list of BPMN elements modelling the empirical workflow and `s`. The definition of `preap` is defined as follows:

```
preap :: Element -> [Element] -> Seqflow -> Tree
preap e es s | (isstart.etype) a = Leaf START
             | isxj e = OneOf (map (trees Pres (es++[e])) (ins a))
             | isaj e = Empiricol.All (map (trees Pres (es++[e])) (ins a))
             | disjs [isxs,isas] e = tree Pres (es++[e]) ((head.ins) a)
             | (iscompound.etype) a = Leaf (ename e)
             where a = atom e
```

This function `preap` checks if element `e` is one of the following:

- If `e` is a start event, then a leaf node of the prerequisite tree has been reached, therefore the function returns `Leaf START`.

- If `e` is a XOR join gateway (a join gateway has multiple incoming sequence flows and one outgoing sequence flow), then semantically the satisfaction of one of the gateway's direct predecessors would satisfy the prerequisite. Therefore the function returns `OneOf ss` where `ss` is the evaluation of the expression `map (trees Pres (es++[e])) (ins a)`.
- If `e` is an AND join gateway, then semantically only the satisfaction of all of the gateway's direct predecessors would satisfy the prerequisite. Therefore the function returns `All ss` where `ss` is the evaluation of the expression `map (trees Pres (es++[e])) (ins a)`.
- If `e` is a split gateway (a split gateway has one incoming sequence flow and multiple outgoing sequence flows), then the satisfaction of its unique predecessor would satisfy the prerequisite. Therefore the function applies `tree` over the incoming sequence flow of `e`.
- If `e` is a subprocess, then it would model a sequence rule and the completion of the rule's evaluation would satisfy the prerequisite. Therefore the function returns `Leaf (ename e)` where `ename e` returns the unique identifier of the sequence rule modelled by `e`.

Conversely to construct a dependency tree, the function applies `dptap` over the element `e`, which has `s` as one of its incoming sequence flows (the direct successor of the subprocess that models the sequence rule), the list of BPMN elements `es`, modelling the empirical workflow and `s`. The definition of `dptap` is defined as follows:

```
dptap :: Element -> [Element] -> Seqflow -> Tree
dptap e es s | (isend.etype) a = Leaf NSTOP
             | (isabort.etype) a = Leaf ASTOP
             | isxs e = OneOf (map (trees Dpts (es++[e])) (outs a))
             | isas e = Empiricol.All (map (trees Dpts (es++[e])) (outs a))
             | disjs [isaj,isxj] e = tree Dpts (es++[e]) ((head.outs) a)
             | (iscompound.etype) a = Leaf (ename e)
             where a = atom e
```

It is easy to see the definition of this function mirrors that of `preap`.

7.4 On Simulation and Automation

In this section we discuss briefly the application of business process management techniques to empirical studies. We describe informally, via a simple example, how modelling empirical studies in BPMN allows their study plans to be simulated and partially automated by translating the BPMN diagrams into executable BPEL processes. A more comprehensive empirical study example is presented in Chapter 8, using which we show how transforming empirical workflows to BPMN assists verifying the correctness of empirical studies

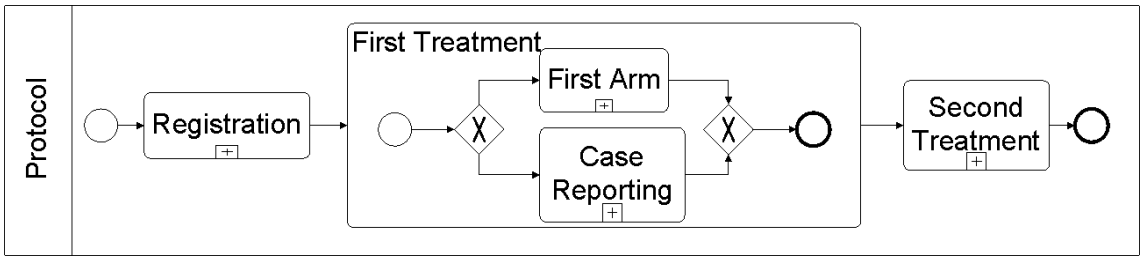


Figure 7.13: A BPMN pool describing the workflow of a clinical trial

As useful as it is to visualise and formally specify a complete study plan, it is also beneficial to validate the plan before its execution phase, especially if the study has a long running duration; it would be undesirable to run into an error three months into the study. One method of validating a study is by simulation. When considering either simulating or automating a portion of a study, we assume the observations specified in that portion can be appropriately simulated or automated; an observation might

define the action of recording a measurement from a display interfacing with a software application or submitting a web form to a web service for analysis.

Figure 7.13 shows a BPMN pool modelling the workflow of a phase III chemotherapy clinical trial. In this trial there are two sets of interventions, each interleaved with observation consisting of submitting forms about specific medical conditions of the patients (case reporting). The following observation workflow is defined by the sequence rule modelled by the subprocess *Case Reporting* in the figure. An expanded view of this subprocess is shown in Figure 7.14.

```
Seq [Single (Id "Hypertensity Report", Dur "P7D",Dur "P7D",None,Manual),
     Choice [Dp (Id "Tumour Measurement Report", Dur "P1D",Dur "P1D",
                Ands [Ors [(Emu ["low"],"blood pressure")]],Manual),
             Dp (Id "Toxicity Review", Dur "P1D",Dur "P1D",
                Ands [Ors [(Emu ["high"],"blood pressure")]],Manual)]]
```

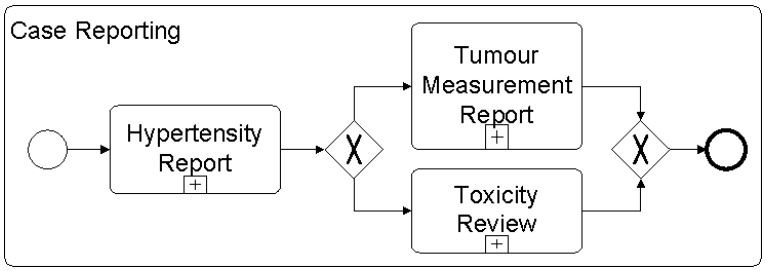


Figure 7.14: A BPMN subprocess of an observation block

While submitting a report form is a manual task, due to the transformation, it is possible to simulate this action by translating its corresponding BPMN subprocess into a sequence of BPEL activities according the mapping provided by the BPMN official documentation [OMG08, Annex A].

```
<sequence>
 <wait for="PT7M"><operation name="sendHypertensityReport">
   <input message="hypertensityMessage" /></operation></wait>
 <switch>
  <case condition="getVariableData('blood pressure') == high">
   <wait for="PT1M"><operation name="sendToxicityReview">
    <input message="toxicityMessage" /></operation></wait></case>
  <case condition="getVariableData('blood pressure') == low">
   <wait for="PT1M"><operation name="sendTumourReport">
    <input message="tumourMessage" /></operation></wait></case>
 </switch></sequence>
```

Here each *wait* activity is an invocation upon the elapse of a specified duration. Since the derived BPEL process is for simulation, we scale down the specified duration of each observation. For example operation `sendHypertensityReport` simulates the hypertensity report being sent. While the observation workflow above specifies this report to be sent after seven days, we scale down the duration to seven minutes (`PT7M`). Note that each invocation in a BPEL process is necessarily of a web service; if the specified observation defines an action to invoke a web service, e.g. uploading a web form, the translated BPEL operation will also be invoking that web service, and otherwise, for simulation purposes, a "dummy" web service could be used for merely receiving appropriate messages. Similarly, partial automation is also possible by translating appropriate observations into BPEL processes which may be executed during the execution phase of the study.

7.5 Summary

Specifications of long running empirical studies are complex; the production of a complete specification can be time-consuming and prone to error. We have described a graphical method to assist this type of specification. We have introduced a workflow model suitable for specifying empirical studies, which then can be populated onto a calendar for scheduling, and described bidirectional transformations, which allow

empirical studies to be modelled graphically using BPMN, and to be simulated and partially automated as BPEL processes. Transforming empirical studies to BPMN also assists verifying the correctness of empirical studies against (clinical) safety properties.

To the best of our knowledge, this chapter describes the first attempt to apply graphical workflow technology to empirical studies and calendar scheduling, while large amounts of research [LAB+06, DBG+04, OAF+04, CGH+06, BJA+08] have focused on the application of workflow notations and implementations to "in silico" scientific experiments. Similarly, research effort has been directed towards effective planning of *specific types* of long running empirical studies, namely clinical trials and guidelines. Related works in scientific workflows and clinical trial planning have been considered in Chapter 3.

Chapter 8

Case Studies

8.1 Introduction

This chapter presents two comprehensive case studies illustrating the semantics defined in Chapters 5 and 6, and the empirical studies model `Empiricol` presented in Chapter 7. In Section 8.2 we consider a collaborative business process describing an airline ticket reservation system, shown in Figure 8.12. This business process has been adopted from the WSCI specification document [W3C02]. We investigate some general behavioural requirements such as deadlock freedom and compatibility, and some specific requirements by applying the notion of compositionality and CSP refinements. In Section 8.3 we consider an empirical workflow specification based on Neo-tAnGo [ECH+04], a phase III breast cancer clinical trial. In particular we investigate the construction of the empirical study using the empirical studies model `Empiricol`, and how transforming it into BPMN assists visualisation and simulation of the study. We also investigate the correctness of the study via a set of oncological safety principles [HSW95]. We summarise the contribution of this chapter in Section 8.4.

8.2 Ticket Reservation Process

This section considers the airline ticket reservation business process shown in Figure 8.12 on Page 171. This business process is adapted from the WSCI specification document [W3C02]. The business process describes the collaboration of three participants: a traveller, a travel agent, and an airline reservation system, which are specified by BPMN pools *Traveller*, *Agent* and *Airline* respectively.

We model the workflow of individual participants in Sections 8.2.1, 8.2.2 and 8.2.3; in Section 8.2.4 we model their collaboration; in Section 8.2.5 we investigate behavioural properties concerning individual participants as well as their collaboration; and in Section 8.2.6 we consider composition development of the business process.

8.2.1 Traveller

The traveller can order a trip by setting up an itinerary for airline tickets, she can then reserve the seats and subsequently proceed with the booking, after which the travel agent and the airline will send her the invoice and the tickets respectively. Figure 8.1 shows the BPMN pool *Traveller* specifying the workflow of the traveller participant.

Specifically, the traveller may choose her travel plan (from a catalogue), and submit her choice to the travel agent via some local web service (e.g. web form) (*Order_Trip*). For various reasons the traveller may choose to change her itinerary (*Change_Itinerary*); she may also decide not to take the trip, in which case she may cancel her order (*Cancel_Itinerary*). In case she decides to accept the proposed itinerary, she may proceed to reserve this itinerary (*Send_Confirmation*) and provide her credit card information to the travel agent. After reservation the traveller may either purchase her tickets (*Book_Tickets*) or cancel them (*Cancel_Tickets*); if she chooses to cancel her tickets, she will have to wait for the cancellation to be processed (*Accept_Cancellation*). Also if she takes too long at purchasing her tickets, a time out will occur; the tickets will then be released from the traveller and she will receive a cancellation notification (*Accept_Cancellation*). From the traveller's point of view, the time restriction would normally be enforced by the travel agent, therefore the time out is modelled as a message exception flow attached to the task *Book_Tickets*. After the traveller has purchased her tickets, she will receive the invoice from the travel agent (*Receive_Invoice*) and tickets from the airline (*Receive_Tickets*). From the point of view of the traveller's workflow, the order in which she receives the invoice and tickets is not important.

The BPMN pool *Traveller* is constructed by applying the following sequence of syntactic operations:

1. *SeqComp* adds the task *Order_Trip*;
2. *Split* adds a XOR split gateway;
3. *SeqComp* adds the task *Change_Itinerary*;
4. *JoinOp* adds a XOR join gateway to join the XOR split gateway and task *Change_Itinerary*;
5. *Loop* adds a sequential loop back to the task *Change_Itinerary*;
6. *Split* adds a XOR split gateway;
7. Two *SeqComp* add two tasks *Send_Confirmation* and *Cancel_Itinerary*;
8. *Split* adds a XOR split gateway;
9. Two *SeqComp* add two tasks *Book_Tickets* and *Cancel_Tickets*;
10. *AddException* creates an exception flow by attaching an intermediate message event to *Book_Tickets*;
11. *JoinOp* adds a XOR join gateway to join the exception flow from *Book_Tickets* and *Cancel_Tickets*;
12. *SeqComp* adds task *Accept_Cancellation*;
13. *Split* adds an AND split gateway after task *Book_Tickets*;
14. Two *SeqComp* add two tasks *Receive_Tickets* and *Receive_Invoice*; and
15. *JoinOp* adds an AND join gateway to synchronise tasks *Receive_Tickets* and *Receive_Invoice*.

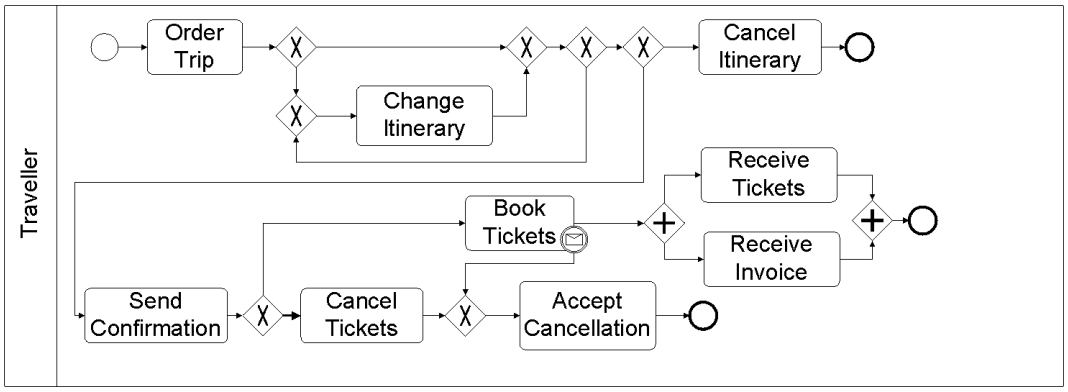


Figure 8.1: Traveller

8.2.2 Travel Agent

The main purpose of the travel agent is to mediate interactions between the traveller who wants to buy airline tickets and the airline who supplies them. Figure 8.2 shows the BPMN pool *Agent* specifying the workflow of the travel agent participant.

Once the travel agent receives an initial order from the traveller (*Receive_Order*), the agent verifies seating availability with the airline; this is modelled by a message flow (dashed line) connecting task *Receive_Order* from the travel agent to task *Verify_Order* from the airline reservation system; this is shown in Figure 8.12. In order to cater for the possibility of the traveller making changes to her itinerary, the travel agent verifies with the airline the availability of the seats (*Receive_Changes*) every time there is a change to the itinerary. There is a period of time between each change to the itinerary, and this is modelled by the intermediate timer event. Once the traveller has agreed upon a particular itinerary, the

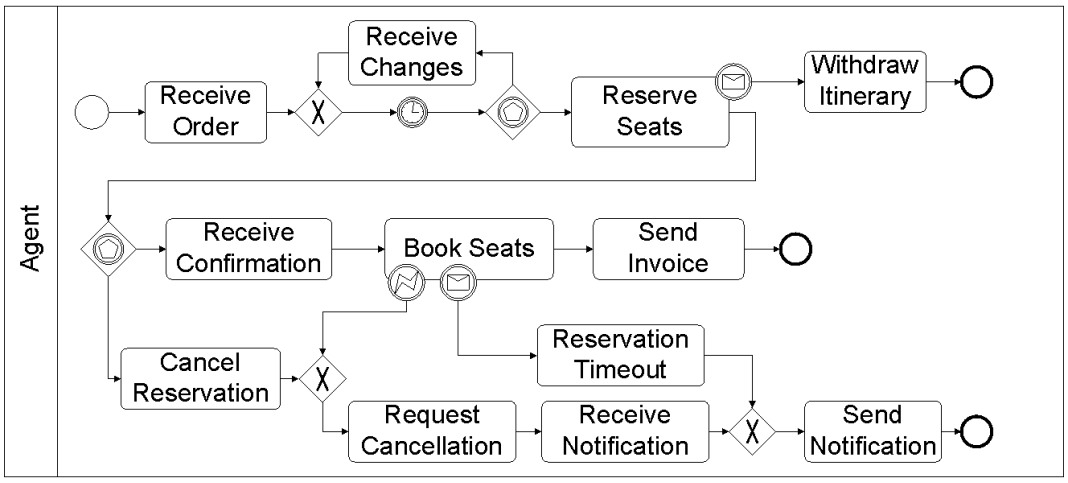


Figure 8.2: Travel Agent

travel agent will reserve the seats for the traveller (*Reserve_Seats*). During the reservation period the traveller may cancel her itinerary, thereby withdrawing her itinerary (*Withdraw_Itinerary*); cancellation is modelled as a message exception flow of task *Reserve_Seats*.

Once the reservation has been completed, the travel agent receives either a confirmation notice from the traveller (*Receive_Confirmation*), in which case he receives the credit card information from the traveller and proceeds with the booking (*Book_Seats*), or a cancellation request (*Cancel_Reservation*), in which case the travel agent will first request cancellation from the airline (*Request_Cancellation*), then wait for a notification confirming the cancellation from the airline (*Receive_Notification*), and then send the notification to the traveller (*Send_Notification*). Also during the booking phase, if either an error (e.g. incorrect card information), modelled as an error exception flow, or a time out (*Reservation_Timeout*) occurs, a cancellation notification will be sent to the traveller. Otherwise, the booking invoice will be sent to the traveller for billing (*Send_Invoice*). Note that from the point of view of the travel agent, time restriction on booking is determined by the particular airline, therefore the time out is modelled as a message exception flow attached to task *Book_Seats*.

The BPMN pool *Agent* is constructed by applying the following sequence of syntactic operations:

1. Two *SeqComp* add the task *Receive_Order* and the intermediate timer event;
2. *EventLoop* adds an event-based XOR split gateway, tasks *Receive_Order* and *Reserve_Seats*, and a XOR join gateway;
3. *AddException* creates an exception flow by attaching an intermediate message event to task *Reserve_Seats*;
4. *SeqComp* adds task *Withdraw_Itinerary*;
5. *EventSplitOp* adds an event-based XOR gateway, and tasks *Receive_Confirmation* and *Cancel_Reservation*;
6. Two *SeqComp* add tasks *Book_Seats* and *Send_Invoice*;
7. *AddException* creates two exception flows by attaching an intermediate message event and an intermediate error event to *Book_Seats*;
8. *SeqComp* adds task *Reservation_Timeout*;
9. *JoinOp* adds a XOR join gateway to merge together the exception flow created by the intermediate error event and task *Cancel_Reservation*;

10. Two *SeqComp* add tasks *Request_Cancellation* and *Receive_Notification*;
11. *JoinOp* adds a XOR join gateway to merge tasks *Reservation_Timeout* and *Receive_Notification*; and
12. *SeqComp* adds task *Send_Notification*.

8.2.3 Airline Reservation System

The airline reservation system receives a request to check for seating availability, it then checks for seating availability, reserves the seats, completes the booking and delivers the tickets. Figure 8.3 shows the BPMN pool *Airline* describing the workflow of the reservation system.

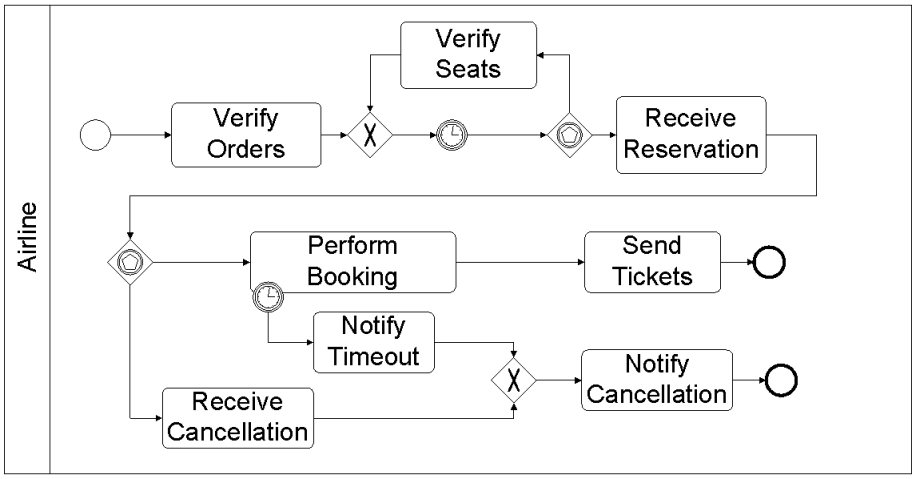


Figure 8.3: Reservation System

Specifically, upon receiving a seating order, the airline verifies the order (*Verify_Orders*). The airline then receives request to change the seating order zero or more times, and at each time the reservation system verifies the change (*Verify_Seats*). Similar to the travel agent, there is a period of time between each seating change, and this is modelled by the intermediate timer event, after which the reservation system can receive the message to reserve the seats and proceed to reserving the seats (*Receive_Reservation*). At this point the reservation system receives either the order to finalise the booking (*Perform_Booking*) or a reservation cancellation notification. In the latter case, the reservation system removes the seat reservations (*Receive_Cancellation*) and sends a cancellation notification (*Notify_Cancellation*). Note that a time out occurs if the booking is not finalised within the time limit after reservations have been made; this is modelled by the timer exception flow (*Itimer*) attached to the task *Perform_Booking*. If a time out occurs, the booking is cancelled and a time out notification (*Notify_Timeout*) and a cancellation notification (*Notify_Cancellation*) will be sent out. Otherwise once the booking is finalised, the seat tickets will be sent out (*Send_Tickets*).

The BPMN pool *Airline* is constructed by applying the following sequence of syntactic operations:

1. Two *SeqComp* add the task *Verify_Orders* and the intermediate timer event;
2. *EventLoop* adds an event-based XOR split gateway, tasks *Verify_Seats* and *Receive_Reservation*, and a XOR join gateway;
3. *EventSplitOp* adds an event-based XOR gateway, and tasks *Perform_Booking* and *Receive_Cancellation*;
4. *SeqComp* adds tasks *Send_Tickets*;
5. *AddException* creates an exception flow by attaching an intermediate timer event to *Perform_Booking*;
6. *SeqComp* adds task *Notify_Timeout*;

7. *JoinOp* adds a XOR join gateway to merge tasks *Notify_Timeout* and *Receive_Cancellation*; and
8. *SeqComp* adds task *Notify_Cancellation*.

8.2.4 Collaboration

Figure 8.12 shows the BPMN diagram modelling the collaboration of *Traveller*, *Agent* and *Airline* in the ticket reservation business process. Using the process semantics defined in Chapter 5, we translate the behaviour of the BPMN diagram into the following CSP process *Collab*.

$$\begin{aligned} Collab \quad = \quad & P(Traveller) \,[\![\, \alpha P(Traveller) \mid \alpha P(Agent) \cup \alpha P(Airline) \,]\!] \\ & (P(Agent) \,[\![\, \alpha P(Agent) \mid \alpha P(Airline) \,]\!]\, P(Airline)) \end{aligned} \tag{8.1}$$

The behaviour of BPMN pools *Traveller*, *Agent* and *Airline* are modelled by CSP processes $P(Traveller)$, $P(Agent)$ and $P(Airline)$ respectively. Here we provide the definition of $P(Traveller)$.

$$\begin{aligned} P(Start1) &= (s.Order\_Trip \rightarrow EP) \,\Box\, EP \\ P(Order\_Trip) &= (s.Order\_Trip \rightarrow w.Order\_Trip \rightarrow m.m1 \rightarrow s.Xsplit1 \rightarrow \\ &\quad P(Order\_Trip)) \,\Box\, EP \\ P(Xsplit1) &= ((s.Xsplit1 \rightarrow (s.Xjoin1 \rightarrow Skip \sqcap s.Xjoin2 \rightarrow Skip)) \fatsemi P(Xsplit1)) \,\Box\, EP \\ P(Xjoin1) &= ((s.Xjoin1 \rightarrow Skip \,\Box\, s.Xjoin12 \rightarrow Skip) \fatsemi s.Xsplit2 \rightarrow P(Xjoin1)) \,\Box\, EP \\ P(Xjoin2) &= ((s.Xjoin2 \rightarrow Skip \,\Box\, s.Xjoin22 \rightarrow Skip) \fatsemi s.Change\_Itinerary \rightarrow \\ &\quad P(Xjoin2)) \,\Box\, EP \\ P(Xsplit2) &= ((s.Xsplit2 \rightarrow (s.Xjoin22 \rightarrow Skip \sqcap s.Xsplit3 \rightarrow Skip)) \fatsemi P(Xsplit2)) \,\Box\, EP \\ P(Xsplit3) &= ((s.Xsplit3 \rightarrow (s.Cancel\_Itinerary \rightarrow Skip \sqcap \\ &\quad s.Send\_Confirmation \rightarrow Skip)) \fatsemi P(Xsplit3)) \,\Box\, EP \\ P(Change\_Itinerary) &= (s.Change\_Itinerary \rightarrow w.Change\_Itinerary \rightarrow m.m2 \rightarrow s.Xjoin12 \rightarrow \\ &\quad P(Change\_Itinerary)) \,\Box\, EP \\ P(Cancel\_Itinerary) &= (s.Cancel\_Itinerary \rightarrow w.Cancel\_Itinerary \rightarrow m.m3 \rightarrow s.EP \rightarrow \\ &\quad P(Cancel\_Itinerary)) \,\Box\, EP \end{aligned}$$

$$
\begin{aligned}
P(\mathit{Send\_Confirmation}) &= (s.\mathit{Send\_Confirmation} \rightarrow w.\mathit{Send\_Confirmation} \rightarrow m.m4 \rightarrow s.\mathit{Xsplit}4 \rightarrow \\
&\quad P(\mathit{Send\_Confirmation})) \,\Box\, \mathit{EP} \\
P(\mathit{Xsplit}4) &= ((s.\mathit{Xsplit}4 \rightarrow (s.\mathit{Cancel\_Ticket} \rightarrow \mathit{Skip} \sqcap s.\mathit{Book\_Ticket} \rightarrow \mathit{Skip})) \fatsemi \\
&\quad P(\mathit{Xsplit}4)) \,\Box\, \mathit{EP} \\
P(\mathit{Cancel\_Tickets}) &= (s.\mathit{Cancel\_Tickets} \rightarrow w.\mathit{Cancel\_Tickets} \rightarrow m.m5 \rightarrow s.\mathit{Xjoin}31 \rightarrow \\
&\quad P(\mathit{Cancel\_Tickets})) \,\Box\, \mathit{EP} \\
P(\mathit{Book\_Tickets}) &= \mathbf{let}\ \ W = w.\mathit{Book\_Tickets} \rightarrow m.m6 \rightarrow s.\mathit{Asplit}1 \rightarrow \mathit{Skip} \\
&\qquad\quad C = m.m8 \rightarrow s.\mathit{Xjoin}32 \rightarrow \mathit{Skip} \\
&\qquad\quad M = m.m8 \rightarrow s.\mathit{Xjoin}32 \rightarrow \mathit{Skip} \,\Box\, s.\mathit{Asplit}1 \rightarrow \mathit{Skip} \\
&\quad \mathbf{in}\ (s.\mathit{Book\_Tickets} \rightarrow (W \triangle C) \,[\![\, \{m.m8, s.\mathit{Xjoin}32, s.\mathit{Asplit}1\} \,]\!]\, M) \fatsemi \\
&\qquad P(\mathit{Book\_Tickets})) \,\Box\, \mathit{EP} \\
P(\mathit{Xjoin}3) &= ((s.\mathit{Xjoin}3 \rightarrow \mathit{Skip} \,\Box\, s.\mathit{Xjoin}32 \rightarrow \mathit{Skip}) \fatsemi \\
&\quad s.\mathit{Accept\_Cancellation} \rightarrow P(\mathit{Xjoin}3)) \,\Box\, \mathit{EP} \\
P(\mathit{Accept\_Cancellation}) &= ((s.\mathit{Accept\_Cancellation} \rightarrow \mathit{Skip} \,|||\, m.m7 \rightarrow \mathit{Skip}) \fatsemi \\
&\quad w.\mathit{Accept\_Cancellation} \rightarrow s.E2 \rightarrow P(\mathit{Accept\_Cancellation})) \,\Box\, \mathit{EP} \\
P(\mathit{Asplit}1) &= ((s.\mathit{Asplit}1 \rightarrow (s.\mathit{Receive\_Tickets} \rightarrow \mathit{Skip} \,|||\, s.\mathit{Receive\_Invoice} \rightarrow \mathit{Skip})) \fatsemi \\
&\quad P(\mathit{Asplit}1)) \,\Box\, \mathit{EP} \\
P(\mathit{Receive\_Tickets}) &= ((s.\mathit{Receive\_Tickets} \rightarrow \mathit{Skip} \,|||\, m.m10 \rightarrow \mathit{Skip}) \fatsemi w.\mathit{Receive\_Tickets} \rightarrow \\
&\quad s.\mathit{Ajoin}1 \rightarrow P(\mathit{Receive\_Tickets})) \,\Box\, \mathit{EP} \\
P(\mathit{Receive\_Invoice}) &= ((s.\mathit{Receive\_Invoice} \rightarrow \mathit{Skip} \,|||\, m.m9 \rightarrow \mathit{Skip}) \fatsemi w.\mathit{Receive\_Invoice} \rightarrow \\
&\quad s.\mathit{Ajoin}12 \rightarrow P(\mathit{Receive\_Invoice})) \,\Box\, \mathit{EP} \\
P(\mathit{Ajoin}1) &= ((s.\mathit{Ajoin}1 \rightarrow \mathit{Skip} \,|||\, s.\mathit{Ajoin}12 \rightarrow \mathit{Skip}) \fatsemi s.E3 \rightarrow P(\mathit{Ajoin}1)) \,\Box\, \mathit{EP} \\
P(E1) &= (s.E1 \rightarrow \mathit{Skip} \fatsemi c.E1 \rightarrow \mathit{Skip}) \,\Box\, c.E2 \rightarrow \mathit{Skip} \,\Box\, c.E3 \rightarrow \mathit{Skip} \\
P(E2) &= s.E2 \rightarrow \mathit{Skip} \fatsemi c.E2 \rightarrow \mathit{Skip}) \,\Box\, c.E1 \rightarrow \mathit{Skip} \,\Box\, c.E3 \rightarrow \mathit{Skip} \\
P(E3) &= s.E3 \rightarrow \mathit{Skip} \fatsemi c.E3 \rightarrow \mathit{Skip}) \,\Box\, c.E1 \rightarrow \mathit{Skip} \,\Box\, c.E2 \rightarrow \mathit{Skip} \\
\mathit{EP} &= c.E1 \rightarrow \mathit{Skip} \,\Box\, c.E2 \rightarrow \mathit{Skip} \,\Box\, c.E3 \rightarrow \mathit{Skip} \\
P(\mathit{Traveller}) &= \mathbf{let}\ S = \{\mathit{Start}1, \mathit{Order\_Trip}, \mathit{Xsplit}1, \mathit{Xsplit}2, \mathit{Xsplit}3, \mathit{Xsplit}4, \mathit{Xjoin}1, \\
&\qquad\quad \mathit{Xjoin}2, \mathit{Xjoin}3, \mathit{Asplit}1, \mathit{Ajoin}1, \mathit{Change\_Itinerary}, \\
&\qquad\quad \mathit{Cancel\_Itinerary}, \mathit{Send\_Confirmation}, \mathit{Cancel\_Tickets}, \\
&\qquad\quad \mathit{Book\_Tickets}, \mathit{Accept\_Cancellation}, \mathit{Receive\_Tickets}, \\
&\qquad\quad \mathit{Receive\_Invoice}, E1, E2, E3\} \\
&\quad \mathbf{in}\ \ \|\, i : S \bullet \alpha P(i) \circ P(i) \qquad (8.2)
\end{aligned}
$$

8.2.5 Requirements

Table 8.1 shows two general requirements (G1 and G2) and five specific requirements (S1, S2, S3, S4 and S5), against which we verify the ticket reservation process.

8.2.5.1 Requirement G1

Requirement G1 states that the workflow of an individual participant is deadlock-free, that is, we are interested to know if any possible order of execution could lead to a situation where the business transaction has not been completed and yet no more progress can be made. In our process semantics, the property of deadlock freedom may be asserted by the following property,

$$\forall\, i : \{\mathit{Travel}, \mathit{Agent}, \mathit{Airline}\} \bullet \mathit{DF} \sqsubseteq_{\mathcal{F}} P(i) \qquad (8.3)$$

where $P(i)$ denotes the process semantics of participant i, and DF is the characteristic deadlock freedom process (Equation 5.6.2 on Page 77.). Property 8.3 may be split into individual refinement assertions

Code	Description
G1	All participants must be deadlock-free.
G2	The collaboration must be deadlock-free.
S1	Once the traveller has confirmed her order, she receives either both tickets and invoice or a notification of cancellation.
S2	It is not possible for the travel agent to request cancellation and for the traveller to receive tickets.
S3	If the traveller issues a cancellation, she must receive a notification.
S4	If the airline reservation system issues a time out during the booking process, the traveller must receive a notification of cancellation.
S5	The travel agent must not allow any kind of cancellation after the traveller has booked her tickets, if an invoice is to be sent to the traveller.

Table 8.1: Requirements

and verified using the FDR tool. Requirement G1 is satisfied.

8.2.5.2 Requirement G2

Requirement G2 states that the collaboration is deadlock-free. Since we know individual participants are themselves deadlock-free, it is sufficient to show that they are behaviourally compatible.

Recall that we write $\mathit{compatible}(P, Q)$ if BPMN processes P and Q are compatible. Since this binary relation *compatible* is symmetric by Definition 5.13, due to Theorem 5.16, it suffices to prove the following predicate expression.

$$\mathit{compatible}(\mathit{Travel}, \mathit{Agent}) \wedge \mathit{compatible}(\mathit{Travel}, \mathit{Airline}) \wedge \mathit{compatible}(\mathit{Agent}, \mathit{Airline})$$

By using the mechanical procedure devised for responsiveness, we initially find that the first conjunct, $\mathit{compatible}(\mathit{Travel}, \mathit{Agent})$, does not hold. The FDR tool returns a counter example in which the CSP process $P(\mathit{Travel}) \,[\![\, \alpha P(\mathit{Travel}) \mid \alpha P(\mathit{Agent}) \,]\!]\, P(\mathit{Agent})$ deadlocks after the following trace restricted to the set of events $\{\!|w|\!\}$ that model work done in BPMN tasks.

$$\langle w.\mathit{Order\_Trip}, w.\mathit{Cancel\_Itinerary}, w.\mathit{Receive\_Order}, w.\mathit{Reserve\_Seat}\rangle$$

The trace shows that after the reservation process has completed, it is still possible for the traveller to cancel her itinerary (*Cancel_Itinerary*) by sending a message to the travel agent's reservation process (*Reserve_Seat*), which causes the deadlock. One possible set of changes to the collaboration between the traveller and the travel agent is shown in Figure 8.4. We describe the changes as follows:

- Traveller – First insert two new tasks *Place_Reservation* and *Confirm_Reservation* before task *Send_Confirmation*, then move the original XOR gateway between *Cancel_Itinerary* and *Send_Confirmation* to between *Cancel_Itinerary* and *Confirm_Reservation*.
- Agent – Replace task *Reserve_Seats* with an intermediate message event and a task *Send_Reservation*, and adds an event-based XOR gateway, choosing between task *Commit_Reservation* and another intermediate message event.
- Interactions – Connect the following message flows:
  - from *Place_Reservation* to the intermediate message event that directly precedes task *Send_Reservation*
  - from *Confirm_Reservation* to *Commit_Reservation*

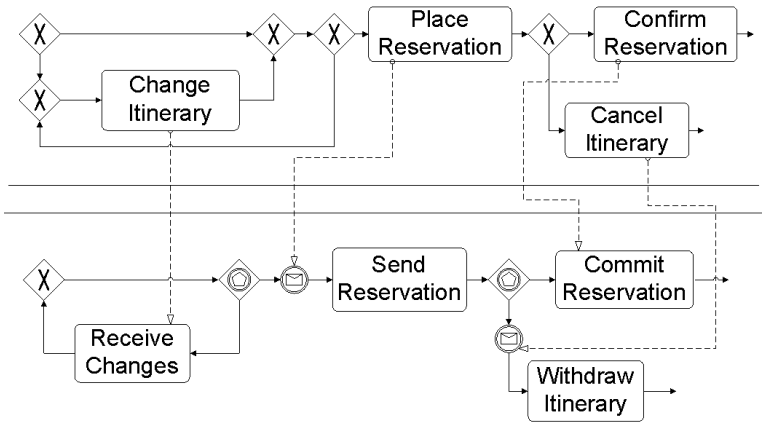


Figure 8.4: Correcting the reservation phase

– from *Cancel_Itinerary* to the intermediate message event that directly precedes task *Withdraw_Itinerary*

After the above changes, we observe that the interaction between the workflows of traveller and travel agent deadlocks after the following trace restricted to set of events $\{\!|w|\!\}$ that model work done in BPMN tasks.

$\langle\, w.Order\_Trip, w.Place\_Reservation, w.Receive\_Order, w.Confirm\_Reservation,$
$w.Send\_Confirmation, w.Send\_Reservation, w.Commit\_Reservation, w.Book\_Tickets,$
$w.Receive\_Confirm, w.Reservation\_Timeout, w.Receive\_Tickets\,\rangle$

The trace shows that while a time out (*Reservation_Timeout*) occurs during task *Book_Seats*, the travel agent is unable to communicate this information to the traveller, hence the traveller assumes the booking is successful; the fact that *w.Receive_Tickets* is performed is irrelevant as task *Receive_Tickets* does not interact with the travel agent. Assuming the traveller is using a web site to carry out the booking, this misinformation could lead to a HTTP 404 error response, which gives no information about the current transaction.

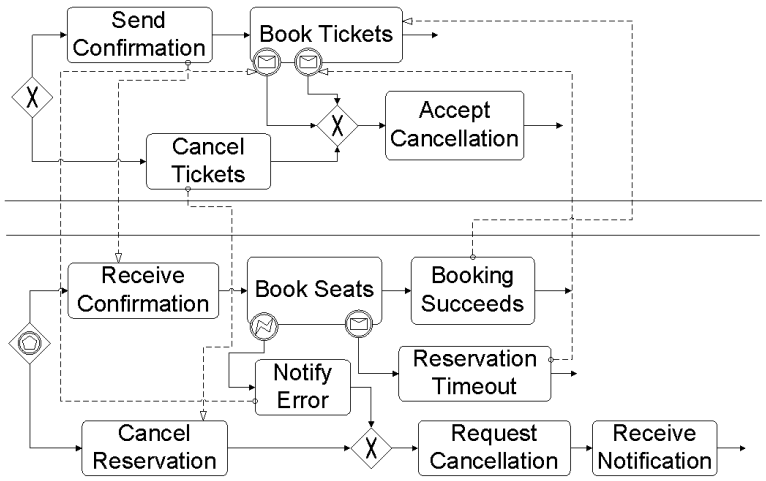


Figure 8.5: Correcting the booking phase

Figure 8.5 shows one possible set of changes to the workflows of the traveller and travel agent; the description of the changes is similar to those described for Figure 8.4 and thus has been omitted.

Figure 8.13 shows a modified BPMN diagram describing the airline ticket reservation business process that satisfies Requirements G1 and G2. The modification is based on further investigation to establish $compatible(Travel, Airline)$ and $compatible(Agent, Airline)$. We have omitted the textual description of the modification. Note that the changes may not reflect those shown in Figures 8.4 and 8.5 as the modification in Figure 8.13 has also taken into account compatibilities between *Agent* and *Airline*, and between *Traveller* and *Airline*.

Specially in Figure 8.13, BPMN pools *CTraveller*, *CAgent* and *CAirline* model the corrected version of the traveller, the travel agent and the airline business processes. The CSP process *CCollab* in the following Equation 8.4 models the behaviour of the BPMN diagram of the ticket reservation business process shown in Figure 8.13.

$$\begin{aligned} CCollab \quad = \quad & P(CTraveller) \,[\![\, \alpha P(CTraveller) \mid \alpha P(CAgent) \cup \alpha P(CAirline) \,]\!] \\ & (P(CAgent) \,[\![\, \alpha P(CAgent) \mid \alpha P(CAirline) \,]\!]\, P(CAirline)) \end{aligned} \tag{8.4}$$

8.2.5.3 Requirement S1

Requirement S1 states that if the traveller has confirmed her order, she receives either both tickets and invoice or a notification of cancellation. Here we abstract the CSP process modelling the BPMN diagram from irrelevant events. Specifically, Requirement S1 concerns events in the following set A,

$$A = \{w.Send\_Confirmation, w.Accept\_Cancellation, w.Receive\_Tickets, w.Receive\_Invoice\}$$

and the following CSP process *Spec* models Requirement S1,

$$\begin{aligned} Spec &= Spec0 \sqcap Spec1 \\ Spec0 &= Proceed(\{w.Send\_Confirmation\}, Spec0 \sqcap Spec1) \\ Spec1 &= w.Send\_Confirmation \rightarrow Spec2 \sqcap Spec3 \sqcap Spec4 \\ Spec2 &= w.Receive\_Tickets \rightarrow Spec5 \\ Spec3 &= w.Receive\_Invoice \rightarrow Spec6 \\ Spec4 &= w.Accept\_Cancellation \rightarrow Spec0 \sqcap Spec1 \\ Spec5 &= w.Receive\_Invoice \rightarrow Spec0 \sqcap Spec1 \\ Spec6 &= w.Receive\_Tickets \rightarrow Spec0 \sqcap Spec1 \end{aligned}$$

where *Proceed* is defined as follows:

$$Proceed(X, P) = Stop \sqcap Skip \sqcap (\sqcap x : \Sigma \setminus X \bullet x \rightarrow P) \tag{8.5}$$

We verify the BPMN diagram against Requirement S1 by checking the refinement assertion $Spec \sqsubseteq_{\mathcal{F}} CCollab \setminus (\Sigma \setminus A)$ using the FDR tool. Requirement S1 holds.

8.2.5.4 Requirement S2

Requirement S2 states that it is not possible for the travel agent to request cancellation and for the traveller to receive tickets. The following CSP process *Spec* models Requirement S2,

$$\begin{aligned} Spec &= Spec0 \sqcap Spec1 \sqcap Spec2 \\ Spec0 &= Proceed(\{w.Cancel\_Itinerary, w.Cancel\_Tickets\}, Spec0 \sqcap Spec1 \sqcap Spec2) \\ Spec1 &= w.Cancel\_Tickets \rightarrow Spec3 \sqcap Spec1 \sqcap Spec2 \\ Spec2 &= w.Cancel\_Itinerary \rightarrow Spec3 \sqcap Spec1 \sqcap Spec2 \\ Spec3 &= Proceed(A, Spec3 \sqcap Spec1 \sqcap Spec2) \end{aligned}$$

where the following set A contains events that are relevant to this requirement.

$$A = \{w.Cancel\_Itinerary, w.Cancel\_Tickets, w.Receive\_Tickets\}$$

We verify the BPMN diagram against Requirement S2 by checking the refinement assertion $Spec \sqsubseteq_{\mathcal{F}} CCollab \setminus (\Sigma \setminus A)$ using the FDR tool. Requirement S2 holds.

8.2.5.5 Requirement S3

Requirement S3 states that if the traveller issues a cancellation, she must receive a notification. The following CSP process *Spec* models Requirement S3.

$$\begin{aligned} Spec &= Spec0 \sqcap Spec1 \\ Spec0 &= Proceed(\{w.Cancel\_Tickets\}, Spec0 \sqcap Spec1) \\ Spec1 &= w.Cancel\_Tickets \rightarrow Spec2 \\ Spec2 &= w.Accept\_Cancellation \rightarrow Spec0 \sqcap Spec1 \end{aligned}$$

We verify the BPMN diagram against Requirement S3 by checking the refinement assertion $Spec \sqsubseteq_{\mathcal{F}} CCollab \setminus (\Sigma \setminus \{w.Cancel\_Tickets, w.Accept\_Cancellation\})$. using FDR. Requirement S3 holds.

8.2.5.6 Requirement S4

Requirement S4 states that if airline reservation system issues a time out during the booking process, the traveller must receive a cancellation notification. The following CSP process *Spec* models Requirement S4.

$$\begin{aligned} Spec &= Spec0 \sqcap Spec1 \\ Spec0 &= Proceed(\{w.Notify\_Timeout\}, Spec0 \sqcap Spec1) \\ Spec1 &= w.Notify\_Timeout \rightarrow Spec2 \\ Spec2 &= w.Accept\_Cancellation \rightarrow Spec0 \sqcap Spec1 \end{aligned}$$

We verify the BPMN diagram against Requirement S4 by checking the refinement assertion $Spec \sqsubseteq_{\mathcal{F}} CCollab \setminus (\Sigma \setminus \{w.Notify\_Timeout, w.Accept\_Cancellation\})$ using FDR. Requirement S4 holds.

8.2.5.7 Requirement S5

Requirement S5 states that the travel agent must not allow any kind of cancellation after the traveller has booked her tickets, if an invoice is to be sent to the traveller. The following CSP process *Spec* models Requirement S5,

$$\begin{aligned} Spec &= Spec0 \sqcap Spec1 \\ Spec0 &= Proceed(\{\ w.Book\_Seats\ \}, Spec0 \sqcap Spec1) \\ Spec1 &= w.Book\_Seats \rightarrow (Spec2 \sqcap Spec3 \sqcap Spec4 \sqcap Spec5 \sqcap Spec6) \\ Spec2 &= Proceed(\{\ w.Book\_Seats, w.Send\_Invoice\ \}, Spec7 \sqcap Spec1) \\ Spec3 &= w.Send\_Invoice \rightarrow (Spec0 \sqcap Spec1) \\ Spec4 &= w.Book\_Seats \rightarrow (Spec2 \sqcap Spec4 \sqcap Spec8 \sqcap Spec9) \\ Spec5 &= Proceed(A \setminus \{\ w.Send\_Invoice\ \}, Spec3) \\ Spec6 &= w.Book\_Seats \rightarrow Spec3 \\ Spec7 &= Proceed(\{\ w.Book\_Seats, w.Send\_Invoice\ \}, Spec0 \sqcap Spec1) \\ Spec8 &= Proceed(A, Spec3) \\ Spec9 &= w.Book\_Seats \rightarrow (Spec3) \end{aligned}$$

where the following set A contains events that are relevant to this requirement.

$$A = \{\ w.Book\_Seats, w.Booking\_Error, w.Booking\_Timeout, w.Send\_Invoice\ \}$$

We verify the BPMN diagram against Requirement S5 by checking the refinement assertion $Spec \sqsubseteq_{\mathcal{F}} CCollab \setminus (\Sigma \setminus A)$ using FDR. Requirement S5 holds.

8.2.6 Compositional Development

Figure 8.6 shows four BPMN pools: *CTraveller*, *Traveller*1, *Traveller*2 and *Traveller*3. BPMN pool *CTraveller* describes the corrected version of the traveller participant. Equation 8.6 defines process $P(CTraveller)$ that models BPMN pool *CTraveller* shown in Figure 8.6. Here for all i in the following set I, process $P(i)$ is the same as that defined in the original Equation 8.2.

$$
\begin{array}{rcl}
I & = & \{Start1, Order\_Trip, Xsplit1, Xjoin1, Xjoin2, Asplit1, Ajoin1, \\
 & & Change\_Itinerary, Cancel\_Itinerary, Cancel\_Tickets, \\
 & & Accept\_Cancellation, Receive\_Tickets, Receive\_Invoice, E1, E2, E3\} \\
I' & = & I \cup \{Xsplit2, Xsplit3, Xsplit4, Xjoin3, Place\_Reservation, \\
 & & Confirm\_Reservation, Send\_Confirmation, Book\_Tickets\}
\end{array}
$$

$$
\begin{array}{rcl}
P(Xsplit2) & = & ((s.Xsplit2 \rightarrow (s.Xjoin22 \rightarrow Skip \sqcap s.Place\_Reservation \rightarrow Skip)) \fatsemi \\
 & & P(Xsplit2)) \,\Box\, EP \\
P(Place\_Reservation) & = & (s.Place\_Reservation \rightarrow w.Place\_Reservation \rightarrow m.m11 \rightarrow s.Xsplit3 \rightarrow \\
 & & P(Place\_Reservation)) \,\Box\, EP \\
P(Xsplit3) & = & ((s.Xsplit3 \rightarrow (s.Cancel\_Itinerary \rightarrow Skip \sqcap \\
 & & s.Confirm\_Reservation \rightarrow Skip)) \fatsemi P(Xsplit3)) \,\Box\, EP \\
P(Confirm\_Reservation) & = & (s.Confirm\_Reservation \rightarrow w.Confirm\_Reservation \rightarrow m.m12 \rightarrow \\
 & & s.Xsplit4 \rightarrow P(Confirm\_Reservation)) \,\Box\, EP \\
P(Xsplit4) & = & ((s.Xsplit4 \rightarrow (s.Cancel\_Ticket \rightarrow Skip \sqcap \\
 & & s.Send\_Confirmation \rightarrow Skip)) \fatsemi P(Xsplit4)) \,\Box\, EP \\
P(Send\_Confirmation) & = & (s.Send\_Confirmation \rightarrow w.Send\_Confirmation \rightarrow m.m4 \rightarrow \\
 & & s.Book\_Ticket \rightarrow P(Send\_Confirmation)) \,\Box\, EP \\
P(Book\_Tickets) & = & \textbf{let} \;\; W = w.Book\_Tickets \rightarrow m.m6 \rightarrow s.Asplit1 \rightarrow Skip \\
 & & \qquad C = m.m8 \rightarrow s.Xjoin32 \rightarrow Skip \,\Box\, m.m13 \rightarrow s.Xjoin33 \rightarrow Skip \\
 & & \qquad M = m.m8 \rightarrow s.Xjoin32 \rightarrow Skip \,\Box\, m.m13 \rightarrow s.Xjoin33 \rightarrow Skip \,\Box \\
 & & \qquad\quad s.Asplit1 \rightarrow Skip \\
 & & \qquad S = \{m.m8, s.Xjoin32, m.m13, s.Xjoin33, s.Asplit1\} \\
 & & \textbf{in} \; ((s.Book\_Tickets \rightarrow (W \,\triangle\, C) \,[\![\, S \,]\!]\, M) \fatsemi P(Book\_Tickets)) \,\Box\, EP \\
P(Xjoin3) & = & ((s.Xjoin3 \rightarrow Skip \,\Box\, s.Xjoin32 \rightarrow Skip \,\Box\, s.Xjoin33 \rightarrow Skip) \fatsemi \\
 & & s.Accept\_Cancellation \rightarrow P(Xjoin3)) \,\Box\, EP \\
P(CTraveller) & = & \|\, i : I' \bullet \alpha P(i) \circ P(i) \qquad (8.6)
\end{array}
$$

8.2.6.1 Modification 1

BPMN pool *Traveller*1 in Figure 8.6 models a traveller behaviour that changes her itinerary at most once. Equation 8.7 defines CSP process $P(Traveller1)$ that models BPMN pool *Traveller*1, where $P(Xsplit2')$ is a refinement of process $P(Xsplit2)$ and function S is defined as follows:

$$
S(x) = \begin{cases} \alpha P(Xsplit2) & \text{if } x = Xsplit2' \\ \alpha P(x) & \text{otherwise} \end{cases}
$$

$$
\begin{array}{rcl}
I'' & = & (I' \setminus \{XSplit2\}) \cup \{XSplit2'\} \\
P(Xsplit2') & = & (s.Xsplit2 \rightarrow s.Place\_Reservation \rightarrow P(Xsplit2')) \,\Box\, EP \\
P(Traveller1) & = & \|\, i : I'' \bullet S(i) \circ P(i) \qquad (8.7)
\end{array}
$$

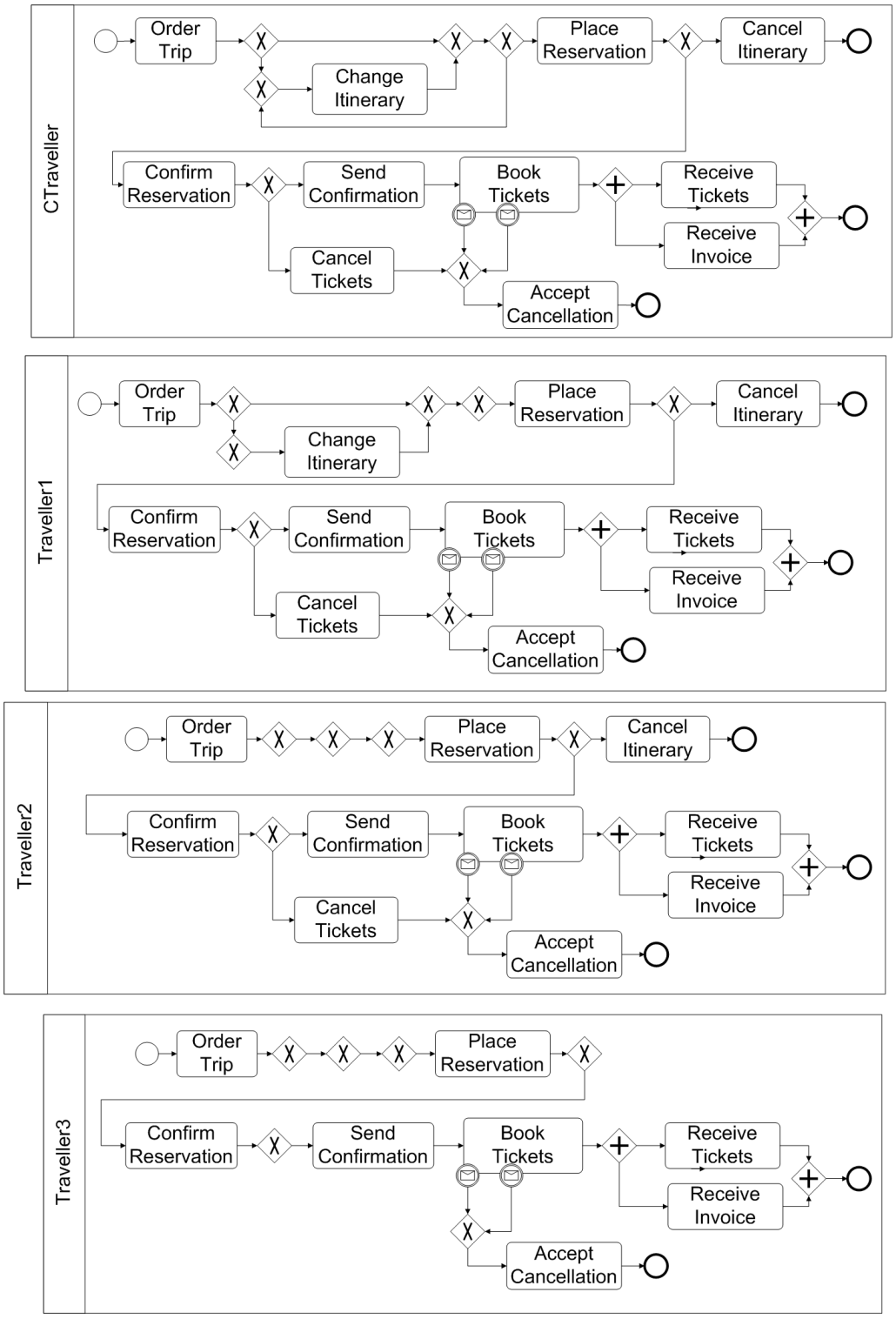


Figure 8.6: Variations of the Traveller workflow

Since the parallel composition operator ($\|$) is monotonic over $\mathcal{F}$, $P(CTraveller) \sqsubseteq_{\mathcal{F}} P(Traveller1)$, that is, $P(Traveller1)$ is a refinement of $P(CTraveller)$, and as a result $P(Traveller1)$ preserves deadlock freedom. Since compatibility is a refinement-closed property, BPMN pool $Traveller1$ is compatible with

BPMN pools *CAirline* and *CAgent* of the BPMN diagram in Figure 8.13. The following CSP process *Collab*1 models the behaviour of the ticket reservation business process with the modified traveller participant *Traveller*1.

$$\begin{aligned} Collab1 &= P(Traveller1) \,[\![\, \alpha P(CTraveller) \mid \alpha P(CAgent) \cup \alpha P(CAirline) \,]\!] \\ &\quad (P(CAgent) \,[\![\, \alpha P(CAgent) \mid \alpha P(CAirline) \,]\!]\, P(CAirline)) \end{aligned} \tag{8.8}$$

By the monotonicity of $\|$, $CCollab \sqsubseteq_{\mathcal{F}} Collab1$, as a result *Collab*1 preserves Requirements S1, S2, S3, S4 and S5.

8.2.6.2 Modification 2

BPMN pool *Traveller*2 on Figure 8.6 models a traveller who does not make any changes to her itinerary; this change of behaviour can be due to additional cost incurred upon changing an existing itinerary. Equation 8.9 defines process $P(Traveller2)$ which models BPMN pool *Traveller*2, where $P(Xsplit1')$ is a refinement of $P(Xsplit1)$, and function S' is defined as follows:

$$S'(x) = \begin{cases} \alpha P(Xsplit1) & \text{if } x = Xsplit1' \\ S(x) & \text{otherwise} \end{cases}$$

$$\begin{aligned} I''' &= (I'' \setminus \{Xsplit1\}) \cup \{Xsplit1'\} \\ P(Xsplit1') &= (s.Xsplit1 \rightarrow s.Xjoin1 \rightarrow P(Xsplit1')) \,\Box\, EP \\ P(Traveller2) &= \|\, i : I''' \bullet S'(i) \circ P(i) \end{aligned} \tag{8.9}$$

By the monotonicity of operator $\|$, $P(Traveller1) \sqsubseteq_{\mathcal{F}} P(Traveller2)$, that is, $P(Traveller2)$ is a refinement of $P(Traveller1)$, and as a result $P(Traveller2)$ preserves deadlock freedom. Since compatibility is a refinement-closed property, BPMN pool *Traveller*2 is compatible with BPMN pools *CAirline* and *CAgent* shown in Figure 8.13. The following CSP process *Collab*2 models the behaviour of the ticket reservation business process with the modified traveller participant *Traveller*2.

$$\begin{aligned} Collab2 &= P(Traveller2) \,[\![\, \alpha P(CTraveller) \mid \alpha P(CAgent) \cup \alpha P(CAirline) \,]\!] \\ &\quad (P(CAgent) \,[\![\, \alpha P(CAgent) \mid \alpha P(CAirline) \,]\!]\, P(CAirline)) \end{aligned} \tag{8.10}$$

Likewise, due to the monotonicity of $\|$, $CCollab \sqsubseteq_{\mathcal{F}} Collab2$, as a result *Collab*2 preserves Requirements S1, S2, S3, S4 and S5.

8.2.6.3 Modification 3

BPMN pool *Traveller*3 on Figure 8.6 models a traveller who does not cancel her orders; this change of behaviour may be due to restrictions applied to different classes of tickets. This modification is achieved by removing tasks *Cancel_Itinerary* and *Cancel_Ticket* from BPMN pool *Traveller*2. Equation 8.11 defines process $P(Traveller3)$ that models BPMN pool *Traveller*3, where $P(Xsplit3')$ and $P(Xsplit4')$ are refinements of processes $P(Xsplit3)$ and $P(Xsplit4)$ respectively, and function S'' is defined as follows:

$$S''(x) = \begin{cases} \alpha P(Xsplit3) & \text{if } x = Xsplit3' \\ \alpha P(Xsplit4) & \text{if } x = Xsplit4' \\ S'(x) & \text{otherwise} \end{cases}$$

$$\begin{aligned} I'''' &= (I''' \setminus \{Xsplit3, Xsplit4\}) \cup \{Xsplit3', Xsplit4'\} \\ P(Xsplit3') &= ((s.Xsplit3 \rightarrow s.Confirm\_Reservation \rightarrow Skip) \fatsemi P(Xsplit3')) \,\Box\, EP \\ P(Xsplit4') &= ((s.Xsplit4 \rightarrow s.Send\_Confirmation \rightarrow Skip) \fatsemi P(Xsplit4')) \,\Box\, EP \\ P(Traveller3) &= \|\, i : I'''' \bullet S''(i) \circ P(i) \end{aligned} \tag{8.11}$$

By the monotonicity of operator $\|$, $P(Traveller2) \sqsubseteq_{\mathcal{F}} P(Traveller3)$, as a result $P(Traveller3)$ preserves deadlock freedom and is compatible with BPMN pools *CAirline* and *CAgent* shown in Figure 8.13.

The following CSP process *Collab*3 models the behaviour of the ticket reservation business process with the modified traveller participant *Traveller*3.

$$\begin{aligned} Collab3 \quad = \quad & P(Traveller3) \,[\![\, \alpha P(CTraveller) \mid \alpha P(CAgent) \cup \alpha P(CAirline) \,]\!] \\ & (P(CAgent) \,[\![\, \alpha P(CAgent) \mid \alpha P(CAirline) \,]\!]\, P(CAirline)) \end{aligned} \tag{8.12}$$

Again, by the monotonicity of operator $\|$, $Collab2 \sqsubseteq_{\mathcal{F}} Collab3$, as a result *Collab*3 preserves Requirements S1, S2, S3, S4 and S5.

8.2.6.4 Extension

We extend the ticket reservation business process with refunding and upgrading processes. The refunding and upgrading processes are managed by the customer service department of the airline. This is modelled by the BPMN pool *CustomerService* shown at the bottom half of Figure 8.7. The BPMN pool *ETraveller* on the top of the figure models the traveller's workflow extended with the interaction with the customer service department. Figure 8.14 shows the complete ticket reservation process extended with refunding and upgrading processes.

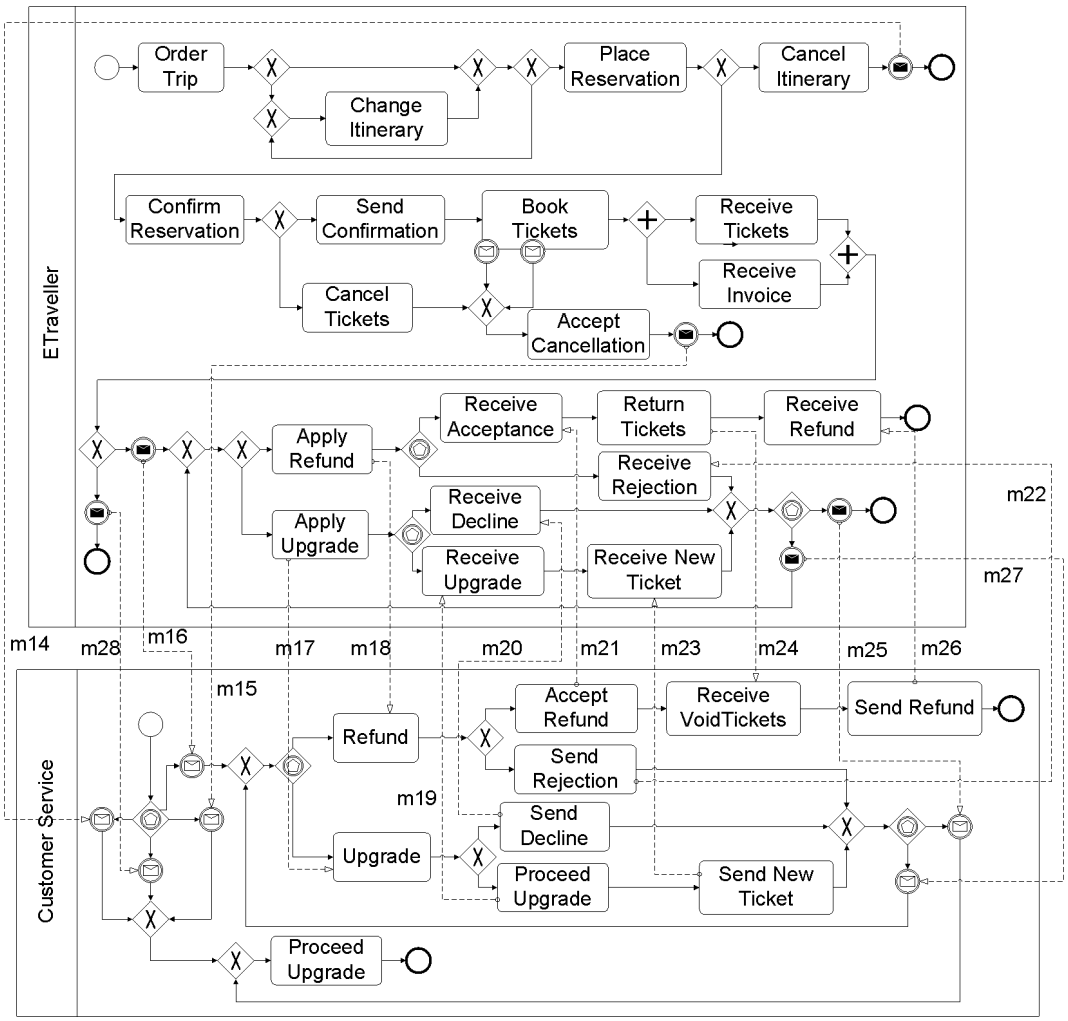


Figure 8.7: Customer Service

After the traveller receives the tickets and the invoice, she applies to refund or upgrade her airline tickets zero or more times. If she decides to apply for a refund, the customer service department would

receive such a request (*Refund*), and the department internally decides if the refund application is to be accepted. If the refund application is accepted (*Accept_Refund*), the department first receives the tickets (*Receive_VoidTickets*) before sending the refund (*Send_Refund*), thereby completing the order. Conversely, if the refund application is not accepted, a rejection notification is sent (*Send_Rejection*). If the traveller decides to apply for an upgrade (*Upgrade*), the customer service department internally decides if the upgrade application is to be accepted. If the upgrade application is accepted (*Proceed_Upgrade*), the department sends out new tickets (*Send_New_Ticket*). Conversely, if the upgrade application is not accepted, a decline notification is sent (*Send_Decline*).

The following CSP process *ECollab* models the behaviour of the extended ticket reservation business process, where $P(CustomerService)$ denotes the CSP process that models the behaviour of BPMN pool *CustomerService*; here we have omitted the definition of process $P(CustomerService)$.

$$\begin{aligned} ECollab \;=\; & P(CustomerService) \\ & [\![\alpha P(CustomerService) \mid \alpha P(ETraveller) \cup \alpha P(CAgent) \cup \alpha P(CAirline)]\!] \\ & (P(ETraveller) \; [\![\alpha P(ETraveller) \mid \alpha P(CAgent) \cup \alpha P(CAirline)]\!] \\ & (P(CAgent) \; [\![\alpha P(CAgent) \mid \alpha P(CAirline)]\!] \; P(CAirline))) \end{aligned} \tag{8.13}$$

Since BPMN pool *CustomerService* does not directly interact with pools *CAgent* and *CAirline*, we can verify that the extended ticket reservation business process shown in Figure 8.14 is deadlock free if $compatible(CAgent, ETraveller)$, $compatible(CAirline, ETraveller)$, $compatible(CAgent, CAirline)$ and $compatible(ETraveller, CustomerServce)$. We use the FDR tool to verify that the extended ticket reservation business process satisfies Requirements S1, S2, S3, S4 and S5.

This extension of the traveller's workflow is constructed using a combination of syntactic operations presented in Chapters 4 and 5. As a result applying the same sequence of operations various versions of the traveller's workflow shown in Figure 8.6 preserve their refinement. Since compatibility is a refinement-closed property, the extended versions of the traveller's workflow are also compatible with BPMN pools *CAgent*, *CAirline* and *CustomerService*. Furthermore we know the parallel composition operator is monotonic over $\mathcal{F}$, therefore their collaborations also satisfy Requirements S1, S2, S3, S4 and S5.

8.3 Clinical Trial Protocol

In this section we consider an empirical study adopted from a phase III breast cancer clinical trial – *NeotAnGo* [ECH$^+$04][1], a neoadjuvant study for the treatment of high risk early breast cancer with molecular profiling, proteomics and candidate gene analysis. The following is the description of the trial's primary objective from the official textual protocol document [ECH$^+$04]:

> A phase III, randomised trial with two-by-two factorial design addressing both the role of Gemcitabine in a sequential neoadjuvant chemotherapy regimen of Epirubicin/Cyclophosphamide and Paclitaxel, and the role of sequencing of these treatment components in terms of short-term and long-term outcome in women presenting with high risk early breast cancer.

8.3.1 Specification

Neoadjuvant chemotherapy refers to the delivery of chemotherapy prior to the definitive surgical or other local (e.g. radiotherapeutic) procedure. The rationale here is to shrink the primary tumour and reduce the chance of it showering off malignant cells into the circulation (metastases), allowing the subsequent operation or radiotherapeutic procedure to carry a better chance of achieving cure.

Specifically this clinical trial employs a 20-week neoadjuvant chemotherapy regimen; 800 patients are randomised to receive Gemcitabine or not, together with a randomisation for sequence of chemotherapy. In particular, each patient receives either $EC + T$ or $EC + TG$, with a second randomisation to sequence, that is, T before G, or G before T, where E, C, T and G denote Epirubicin, Cyclophosphamide, Paclitaxel and Gemcitabine respectively. Each possible sequence of chemotherapy is known as an *arm* in the trial. Figure 8.8 shows the schema of the trial [ECH$^+$04, Page 4], where each part of the trial is detailed in the list below. Note that in this case study we are interested in the chemotherapeutic

[1] Some details of the trial have been modified for purpose of the case study

interventions of the trial and therefore do not intend to model the core biopsy, the selection (eligibility criteria) and randomisation processes.

- Arm A:
  - Epirubicin (90 mg/m^2 by slow push into fast drip), every 2 weeks for 5 cycles;
  - Cyclophosphamide (600 mg/m^2 by slow push infusion), every 3 weeks for 4 cycles;
  - Paclitaxel (175 mg/m^2 by 3 hour infusion), every 2 weeks for 4 cycles.
- Arm B1:
  - Epirubicin (90 mg/m^2 by slow push into fast drip), every 2 weeks for 5 cycles;
  - Cyclophosphamide (600 mg/m^2 by slow push infusion), every 3 weeks for 4 cycles;
  - Paclitaxel (175 mg/m^2 by 3 hour infusion), followed by Gemcitabine (2000 mg/m^2 by 60 minute infusion) within 2 days, every 2 weeks for 4 cycles.
- Arm B2:
  - Epirubicin (90 mg/m^2 by slow push into fast drip), every 2 weeks for 5 cycles;
  - Cyclophosphamide (600 mg/m^2 by slow push infusion), every 3 weeks for 4 cycles;
  - Gemcitabine (2000 mg/m^2 by 60 minute infusion), followed by Paclitaxel (175 mg/m^2 by 3 hour infusion), within 2 days, every 2 weeks for 4 cycles.

During chemotherapy, associated case report forms must be submitted according to some predefined schedule. In this case study, case reporting takes place periodically every 2 weeks for 5 cycles.

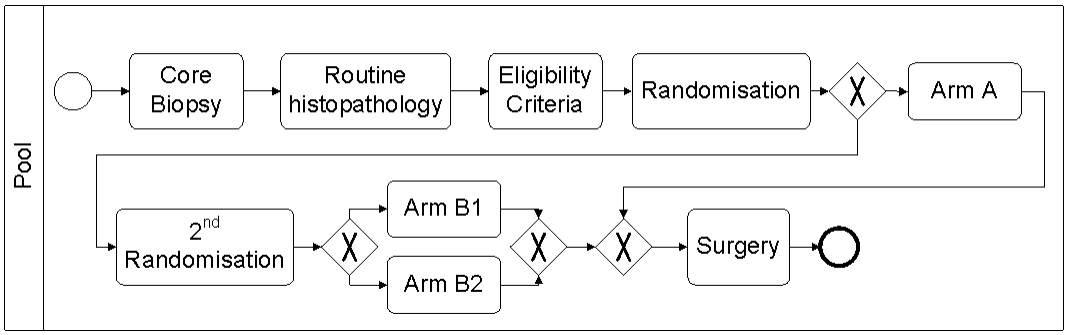


Figure 8.8: Neo-tAnGo trial schema

8.3.2 Modelling

We describe how to model the clinical trial using the empirical workflow model `Empiricol`. We map each arm of the trial to a sequence rule, that is, each arm corresponds to a single `EventSequencing` value. In particular, each sequence rule has the form,

```
Event (Id i) (Leaf START) (Leaf NSTOP) NoCond NoCond
     (Single (Dp (Id i,Dur "P14D",Dur "P14D",5,NoCond,Manual))) [] fw(i)
```

where `fw(i)` returns the procedural workflow for the rule identified by `i`. Given `i` ranges over values *A*, *B1* and *B2*, we enumerate the values of `fw(i)`,

```
fw("A") =
  Par [Single (Wk (WkS ("A_E",te,Dur "P14D",Dur "P14D",5))),
       Single (Wk (WkS ("A_C",tc,Dur "P21D",Dur "P21D",4))),
       Single (Wk (WkS ("A_T",tt,Dur "P14D",Dur "P14D",4)))]
fw("B1") =
  Par [Single (Wk (WkS ("B1_E",te,Dur "P14D",Dur "P14D",5))),
```

```
      Single (Wk (WkS ("B1_C",tc,Dur "P21D",Dur "P21D",4))),
      Single (Wk (WkM ("B1_GT",
                 Seq [Single (Wu ("B1_G",tt,Dur "P0D",Dur "P0D")),
                      Single (Wu ("B1_T",tt,Dur "P0D",Dur "P2D"))],Dur "P14D",4)))]
fw("B2") =
  Par [Single (Wk (WkS ("B2_E",te,Dur "P14D",Dur "P14D",5))),
      Single (Wk (WkS ("B2_C",tc,Dur "P21D",Dur "P21D",4))),
      Single (Wk (WkM ("B2_GT",
                 Seq [Single (Wu ("B1_T",tt,Dur "P0D",Dur "P0D")),
                      Single (Wu ("B1_G",tt,Dur "P0D",Dur "P2D"))],Dur "P14D",4)))]
```

where the values `te`, `tc`, `tt` and `tg` are defined as follows:

```
te = ProcedureA (Treatment "Epirubicin" "90" "slow push into fast drip")
tc = ProcedureA (Treatment "Cyclophosphamide" "600" "slow push infusion")
tt = ProcedureA (Treatment "Paclitaxel" "175" "3 hour infusion")
tg = ProcedureA (Treatment "Gemcitabine" "2000" "60 minute infusion")
```

The constructor `Treatment` encapsulates details of `ProcedureA` such that `Treatment n d m` records the administration of `d` units (mg/m^2) of drug `n` using method `m`.

We extend this list of sequence rules with the appropriate starting (`Start`) and ending (`NStop`) sequences, such that starting sequence's dependency tree is the same as the ending sequence's prerequisite tree and is defined as `OneOf [Leaf (Id "A"),Leaf (Id "B1"),Leaf (Id "B2")]`.

Given the above description of the empirical workflow, we employ the function `etb`, to transform the trial to the BPMN diagram shown in Figure 8.9.

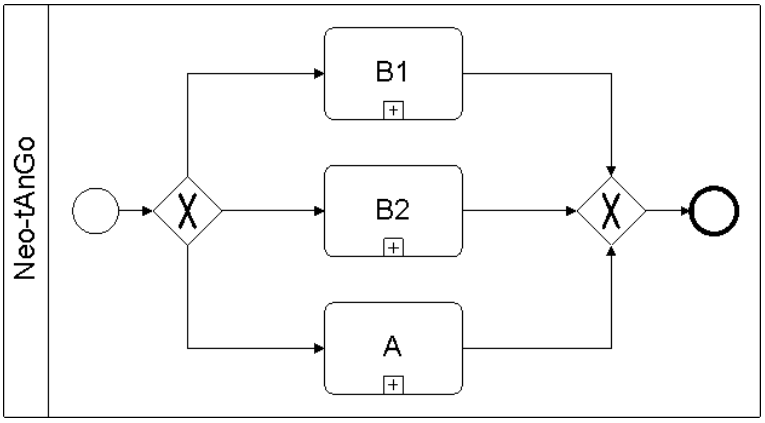


Figure 8.9: BPMN model of Neo-tAnGo chemotherapy

In this figure, each collapsed subprocess models a sequence rule. For example, subprocess A, which is shown in expanded view in Figure 8.10, corresponds to sequence rule A.

Chapter 7 motivates the need to simulate empirical workflows and demonstrates how a workflow can be simulated by transforming its BPMN model to an executable BPEL process. Here we consider how this technique can be applied to the Neo-tAnGo clinical trial. The following BPEL code implements the BPMN model in Figure 8.9 according to the mapping provided by the BPMN's official documentation [OMG08, Annex A].

```
<pick><onMessage operation="chooseA">activityA</onMessage>
      <onMessage operation="chooseB1">activityB1</onMessage>
      <onMessage operation="chooseB2">activityB2</onMessage></pick>
```

Specifically it implements an exclusive choice via the `<pick>` construct, respecting the fact that each patient undergoes treatment from precisely one arm during the trial. The values `activityA`, `activityB1` and `activityB2` are placeholders for implementation of sequence rules A, *B1* and *B2* respectively.

For illustration purposes we consider sequence rule *B2* (`activityB2`) while other rules may be simulated in similar manner. The interleaving of the procedural and observation workflows of rule *B2* is implemented in BPEL as follows:

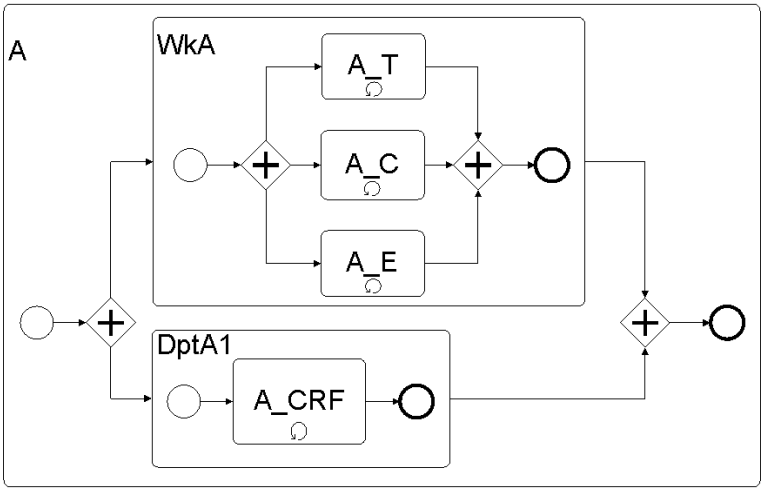


Figure 8.10: BPMN model of treatment arm *A*

```
<flow> procedureB2 observationB2 </flow>
```

where `procedureB2` implements the procedural workflow and `observationB2` implements the observation workflow. We first consider the procedural workflow, which is implemented in BPEL as follows:

```
<sequence>
  <flow>
    <while condition="B2_E_lc < 5">
      <sequence>
        <wait><for>"P14D"</for></wait><invoke operation="B2_E" />...</invoke>
        <assign name="B2_E_incr_lc">... </assign>
      </sequence>
    </while>
    <while condition="B2_C_lc < 4">
      <sequence>
        <wait><for>"P21D"</for></wait><invoke operation="B2_C" />...</invoke>
        <assign name="B2_C_incr_lc">... </assign>
      </sequence>
    </while>
    <while condition="B2_GT_lc < 4">
      <sequence>
        <wait><for>"P14D"</for></wait><invoke operation="B2_T" />...</invoke>
        <wait><for>"P2D"</for></wait><invoke operation="B2_G" />...</invoke>
        <assign name="B2_GT_incr_lc">... </assign>
      </sequence>
    </while>
  </flow>
</sequence>
```

Here, each invocation activity of the form `<invoke operation="B2_T" />...</invoke>` represents a single chemotherapy session. For the purpose of simulation, it suffices to use "dummy" services to simulate these activities. Specifically the BPEL mapping above implements the interleaving of three `<while>` activities using the `<flow>` activity. The first `<while>` activity implements a SRP (single repeatable procedure) that models the Epirubicin chemotherapy; the second `<while>` implements the SRP a SRP that models the Cyclophosphamide chemotherapy; and the third `<while>` implements a RWP (Repeatable Workflow Procedure) that models the Paclitaxel/Gemcitabine chemotherapy. It is easy to see that similar transformations can be performed on the procedural workflow of the other rules.

We now consider observation workflows. The following shows the BPEL mapping of the observation workflow specified by sequence rule *B2*; this mapping is in accordance with BPMN's official documentation [OMG08].

```
<sequence>
  <while condition="DptB2_lc < 5">
```

```
      <sequence>
        <wait><for>"P14D"</for></wait>
        <invoke operation="B2_CRF" inputVariable="reportCRF" />...</invoke>
        <assign name="DptB2_incr_lc">... </assign>
      </sequence>
    </while>
  </sequence>
```

This BPEL mapping defines a `<sequence>` activity, which encapsulates a `<while>` activity that implements the observation workflows of *B2*'s case reporting; observation workflow of the other rules can be transformed in the similar way.

8.3.3 Requirements

For treatment in a clinical trial, compliance with the trial protocol is important to ensure *efficacy* and *safety*. Table 8.2 shows a set of generic oncological safety principles derived by Hammond et al. [HSW95].

Types	Description	Behaviour
Warning	Warn about hazards due to inadequate execution of essential actions	P
Reaction	React appropriately to ameliorate detected hazards	P
Exacerbation	Avoid exacerbating anticipated hazards	P
Monitoring	Monitor responses which herald hazardous situations	P+O
Efficacy	Ensure that overall plans are efficacious in pursuing stated objectives	P
Sequencing	Order (essential) actions temporally for good effect and least harm	P
Diminution	Avoid undermining the benefits of essential actions	P
Critiquing	Critique the proposal of certain hazardous actions even if they are well motivated	P
Prevention	Prevent or ameliorate hazards before executing an essential action	P

Table 8.2: Safety principles in oncology

The third column of the table partitions the set of principles into two groups – principles in group P+O concerns both procedural and observational behaviours, while principles in group P concerns only procedural behaviour. The application of safety principles requires the existence of an expert knowledge base on drug efficacy and toxicity; we assume this knowledge base and derive the following set of behavioural requirements for chemotherapy treatments.

N1 Paclitaxel must be administered after the administrations of Epirubicin and before that of Cyclophosphamide.

N2 No Gemcitabine should be administered between the administrations of Epirubicin and Cyclophosphamide.

N3 After every application of Paclitaxel, data must be collected via case reporting.

While Requirements N1 and N2 concern only procedural behaviour, Requirement N3 concerns both procedural and observational behaviours. Before addressing the requirements in detail, in the following section we consider how the structure and the semantics of `Empiricol` permit *compositional verification*.

8.3.3.1 Compositional Verification

We show how to verify empirical workflow compositionally by verifying its sequence rules individually. We first consider compositional verification of CSP processes and then show how these results can be applied to empirical workflows

Consider some CSP process Q, R and S, and some set of CSP events A. Using the definition of failures refinement, we infer the following:

$$S \sqsubseteq_{\mathcal{F}} (Q \sqcap R) \setminus A$$

$$
\begin{aligned}
&\Leftrightarrow S \sqsubseteq_{\mathcal{F}} (Q \setminus A \sqcap R \setminus A) && \text{[hide-}\sqcap\text{-dist]}\\
&\Leftrightarrow (\mathcal{F}[\![Q \setminus A]\!] \cup \mathcal{F}[\![R \setminus A]\!]) \subseteq \mathcal{F}[\![S]\!] && \text{[def of } \sqcap\text{]}\\
&\Leftrightarrow \mathcal{F}[\![Q \setminus A]\!] \subseteq \mathcal{F}[\![S]\!] \wedge \mathcal{F}[\![R \setminus A]\!] \subseteq \mathcal{F}[\![S]\!] && \text{[set theory]}\\
&\Leftrightarrow S \sqsubseteq_{\mathcal{F}} Q \setminus A \wedge S \sqsubseteq_{\mathcal{F}} R \setminus A && \text{[def of } \mathcal{F}\text{]}
\end{aligned}
$$

This means Q and R refine S if and only if $Q \sqcap R$ refines S. This gives us the following law (labelled refine-$\sqcap$-dist).

$$S \sqsubseteq_{\mathcal{F}} Q \setminus A \wedge S \sqsubseteq_{\mathcal{F}} R \setminus A \Leftrightarrow S \sqsubseteq_{\mathcal{F}} (Q \sqcap R) \setminus A \qquad \text{[refine-}\sqcap\text{-dist]}$$

We now consider the CSP process $Q \,|||\, R$. If we assume the following conditions:

1. $\alpha Q \cap \alpha R = \emptyset$
2. $\alpha S \subseteq \alpha Q$
3. $\mathit{Skip} \equiv_{\mathcal{F}} R \setminus \Sigma$: R is deadlock free and contains only finite traces.

Then given $A = \Sigma \setminus \alpha S$, we infer the following:

$$
\begin{aligned}
&S \sqsubseteq_{\mathcal{F}} (Q \,|||\, R) \setminus A\\
&\quad\Leftrightarrow S \sqsubseteq_{\mathcal{F}} (Q \setminus A \,|||\, R \setminus A) && \text{[hide-}|||\text{-dist]}\\
&\quad\Leftrightarrow S \sqsubseteq_{\mathcal{F}} ((Q \setminus A) \,|||\, \mathit{Skip}) && \text{[Assum. 1, 2, 3]}\\
&\quad\Leftrightarrow S \sqsubseteq_{\mathcal{F}} Q \setminus A && \text{[}|||\text{-unit]}
\end{aligned}
$$

This means if $Q \setminus A$ refines S if and only if $(Q \,|||\, R) \setminus A$ refines S given the assumptions above. This gives us the following law (labelled refine-$|||$-dist).

$$S \sqsubseteq_{\mathcal{F}} Q \setminus A \Leftrightarrow S \sqsubseteq_{\mathcal{F}} (Q \,|||\, R) \setminus A \qquad \text{[refine-}|||\text{-dist]}$$

We now consider these two laws can be applied to the `Empiricol`. Consider the BPMN diagram in Figure 8.9: after the start event is triggered, the XOR split gateway only triggers one of three subprocesses denoting the three separate treatment arms. By the process semantics of XOR split gateway, this choice is internal. Let CSP process *Trial* denote the process that models the behaviour of the diagram, and CSP processes AP, $B1P$ and $B2P$ denote the relative timed models of the subprocesses. By Law refine-$\sqcap$-dist above, we can see for any specification process S where $A = \Sigma \setminus \alpha S$:

$$S \sqsubseteq_{\mathcal{F}} \mathit{Trial} \setminus A \Leftrightarrow S \sqsubseteq_{\mathcal{F}} AP \setminus A \wedge S \sqsubseteq_{\mathcal{F}} B1P \setminus A \wedge S \sqsubseteq_{\mathcal{F}} B2P \setminus A \tag{8.14}$$

We now consider each subprocess in Figure 8.9 individually. Each subprocess models a sequence rule; after the start event of the subprocess is triggered, the parallel gateway triggers both subprocesses modelling the rule's procedural workflow and observation workflow. In terms of semantics, the behaviour of both workflows are interleaved. This is because by the definition of the diagram's process semantics, their process alphabets are disjoint and their process behaviours only synchronise on their own alphabets. Furthermore, by Corollary 7.4 both workflows have finite behaviour and are deadlock free. If we let processes AWP, $B1\,WP$ and $B2\,WP$ to describe behaviour of the BPMN subprocesses modelling the procedural workflow of treatment Arms A (WkA), $B1$ ($WkB1$) and $B2$ ($WkB2$) respectively. By Law refine-$|||$-dist above, and Equivalence 8.14, we have the following logical equivalence.

$$S \sqsubseteq_{\mathcal{F}} \mathit{Trial} \setminus A \Leftrightarrow S \sqsubseteq_{\mathcal{F}} APW \setminus A \wedge S \sqsubseteq_{\mathcal{F}} B1PW \setminus A \wedge S \sqsubseteq_{\mathcal{F}} B2PW \setminus A \tag{8.15}$$

Following the relative timed extension presented in Chapter 6, we consider the semantics of procedural workflow of $B1$. Let CSP events $w.G$, $w.T$, $w.C$ and $w.E$ model Gemcitabine, Paclitaxel, Cyclophosphamide and Epirubicin chemotherapy procedures. Equation 8.16 shows the definition of process $B1PW$. $B1PW$ is defined by parallel composing the enactment process $TP(B1P)$, modelling untimed behaviour of procedural workflow of treatment Arm $B1$, and the coordination process $CP(B1P)$, coordinating the relative timed behaviour of the enactment process $TP(B1P)$. $TP(B1P)$ is defined in Equation 8.17 and

$CP(B1P)$ is defined in Equation 8.18; the behaviour of the procedural workflow of the treatment Arms A and $B2$ can be defined in the similar way.

$$B1PW = TP(B1P) \,[\![\, \Sigma \,]\!]\, CP(B1P) \tag{8.16}$$

$$\begin{array}{rcl}
P(start1) & = & (s.ASplit \rightarrow c.endB1 \rightarrow Skip) \,\Box\, c.endB1 \rightarrow Skip \\
P(ASplit) & = & ((s.ASplit \rightarrow (s.C \rightarrow Skip \;|||\; s.E \rightarrow Skip \;|||\; s.Sub \rightarrow Skip)) \fatsemi \\
 & & P(ASplit)) \,\Box\, c.endB1 \rightarrow Skip \\
P(Sub) & = & \textbf{let}\;\; P(start2) = (s.T \rightarrow c.endB12 \rightarrow Skip) \,\Box\, c.endB12 \rightarrow Skip \\
 & & \quad P(T) = (s.T \rightarrow w.T \rightarrow s.G \rightarrow P(T)) \,\Box\, c.endB12 \rightarrow Skip \\
 & & \quad P(G) = (s.G \rightarrow w.G \rightarrow s.endB12 \rightarrow P(G)) \,\Box\, c.endB12 \rightarrow Skip \\
 & & \quad P(endB12) = s.endB12 \rightarrow c.endB12 \rightarrow Skip \\
 & & \quad SP = \;\|\; i : \{start2, T, G, endB12\} \bullet \alpha P(i) \circ P(i) \\
 & & \textbf{in}\; ((s.Sub \rightarrow (SP \fatsemi SP \fatsemi SP \fatsemi SP)) \fatsemi s.Ajoin3 \rightarrow P(Sub)) \,\Box\, c.endB1 \rightarrow Skip \\
P(E) & = & ((s.E \rightarrow (\fatsemi\; i : \langle 1 \,..\, 5 \rangle \bullet w.E \rightarrow Skip)) \fatsemi s.Ajoin1 \rightarrow P(E)) \,\Box\, c.endB1 \rightarrow Skip \\
P(C) & = & ((s.C \rightarrow (\fatsemi\; i : \langle 1 \,..\, 5 \rangle \bullet w.C \rightarrow Skip)) \fatsemi s.Ajoin2 \rightarrow P(C)) \,\Box\, c.endB1 \rightarrow Skip \\
P(Ajoin) & = & ((s.Ajoin1 \rightarrow Skip \;|||\; s.Ajoin2 \rightarrow Skip \;|||\; s.Ajoin3 \rightarrow Skip) \fatsemi \\
 & & s.endB1 \rightarrow P(Ajoin)) \,\Box\, c.endB1 \rightarrow Skip \\
P(endB1) & = & s.endB1 \rightarrow c.endB1 \rightarrow Skip \\
TP(B1P) & = & \;\|\; i : \{start1, ASplit, Sub, E, C, Ajoin, endB1\} \bullet \alpha P(i) \circ P(i)
\end{array} \tag{8.17}$$

$$\begin{array}{rcl}
CP(B1P) & = & s.ASplit \rightarrow (C16 \,\Box\, C17 \,\Box\, C18) \\
C01 & = & w.G \rightarrow C03 \\
C02 & = & w.C \rightarrow C03 \\
C03 & = & s.endB12 \rightarrow c.endB12 \rightarrow C05 \\
C04 & = & s.endB12 \rightarrow w.C \rightarrow c.endB12 \rightarrow C05 \\
C05 & = & s.Ajoin3 \rightarrow w.E \rightarrow s.Ajoin1 \rightarrow w.C \rightarrow s.Ajoin2 \rightarrow s.endB1 \rightarrow \\
 & & c.endB1 \rightarrow Skip \\
C06 & = & w.C \rightarrow w.E \rightarrow C08 \\
C07 & = & w.E \rightarrow w.C \rightarrow C08 \\
C08 & = & s.T \rightarrow w.T \rightarrow s.G \rightarrow w.G \rightarrow s.endB12 \rightarrow c.endB12 \rightarrow \\
 & & w.E \rightarrow s.T \rightarrow w.T \rightarrow s.G \rightarrow C09 \\
C09 & = & (w.C \rightarrow C01) \,\Box\, ((w.G \rightarrow (C02 \,\Box\, C04)) \rhd (w.C \rightarrow C01)) \\
C10 & = & w.E \rightarrow s.T \rightarrow C12 \\
C11 & = & s.T \rightarrow w.E \rightarrow C12 \\
C12 & = & w.T \rightarrow s.G \rightarrow w.G \rightarrow s.endB12 \rightarrow c.endB12 \rightarrow w.C \rightarrow w.E \rightarrow \\
 & & s.T \rightarrow w.T \rightarrow s.G \rightarrow w.G \rightarrow s.endB12 \rightarrow c.endB12 \rightarrow (C06 \,\Box\, C07)
\end{array}$$

$$\begin{array}{rcl}
C13 & = & s.E \rightarrow (C11 \,\Box\, C10) \\
C14 & = & s.C \rightarrow (C11 \,\Box\, C10) \\
C15 & = & s.Sub \rightarrow (C11 \,\Box\, C10) \\
C16 & = & s.C \rightarrow ((s.E \rightarrow C15) \,\Box\, (s.Sub \rightarrow C13)) \\
C17 & = & s.E \rightarrow ((s.C \rightarrow C15) \,\Box\, (s.Sub \rightarrow C14)) \\
C18 & = & s.Sub \rightarrow ((s.C \rightarrow C13) \,\Box\, (s.E \rightarrow C14))
\end{array} \tag{8.18}$$

8.3.3.2 Requirement N1

Requirement N1 states that Paclitaxel must be administered between the administrations of Epirubicin and Cyclophosphamide. The following CSP process *Spec* models Requirement N1.

$$
\begin{aligned}
Spec &= Spec0 \sqcap Spec1 \\
Spec0 &= Proceed(\{w.E\}, Spec) \\
Spec1 &= w.E \rightarrow Spec2 \sqcap Spec3 \sqcap Spec4 \sqcap Spec5 \\
Spec2 &= Proceed(\{w.C, w.E\}, Spec6 \sqcap Spec1) \\
Spec3 &= w.E \rightarrow Spec7 \sqcap Spec3 \sqcap Spec4 \sqcap Spec8 \\
Spec4 &= Proceed(\{w.C, w.E\}, Spec9) \\
Spec5 &= w.T \rightarrow (Spec10 \sqcap Spec11) \\
Spec6 &= Proceed(\{w.C, w.E\}, Spec12 \sqcap Spec1) \\
Spec7 &= Proceed(\{w.C, w.E\}, Spec6 \sqcap Spec1) \\
Spec8 &= w.T \rightarrow Spec13 \\
Spec9 &= w.T \rightarrow Spec10 \\
Spec10 &= w.C \rightarrow Spec \\
Spec11 &= Proceed(\{w.E\}, Spec10) \\
Spec12 &= Proceed(\{w.C, w.E\}, Spec) \\
Spec13 &= Proceed(\{w.C, w.E\}, Spec10)
\end{aligned}
$$

Since Requirement N1 does not impose behavioural constraints on the observation workflows, by Equivalence 8.15, we verify the diagram's behaviour against N1 by verifying the behaviour of individual procedural workflow of the subprocesses. Formally we have the following logical equivalence.

$$Spec \sqsubseteq_{\mathcal{F}} Trial \setminus H \Leftrightarrow Spec \sqsubseteq_{\mathcal{F}} APW \setminus H \wedge Spec \sqsubseteq_{\mathcal{F}} B1PW \setminus H \wedge Spec \sqsubseteq_{\mathcal{F}} B2PW \setminus H$$

Here, processes AWP, $B1\,WP$ and $B2\,WP$ describe the behaviour of the BPMN subprocesses modelling the procedural workflow of treatment Arms A (WkA), $B1$ ($WkB1$) and $B2$ ($WkB2$) respectively. We abstract irrelevant behaviour by hiding the set of events $H = \Sigma \setminus \alpha Spec$.

We check whether the BPMN diagram satisfies this property by checking the above refinement assertion using the FDR tool. We first consider whether process $AWP \setminus H$ is a refinement of *Spec*. This refinement does not hold and the trace $\langle w.T, w.E, w.C \rangle$ is given as a counterexample by FDR. This counterexample shows the possibility of administering Epirubicin and then Cyclophosphamide without administering Paclitaxel in between. While there are several solutions to overcome this problem, such as changing the timing specification of the trial plan, thereby requiring revision to the trial protocol, deciding which solution is preferable would require clinical insight.

8.3.3.3 Requirement N2

Requirement N2 states that no Gemcitabine should be administered in between the administrations of Epirubicin and Cyclophosphamide. The following CSP process *Spec* models Requirement N2.

$$\begin{aligned}
Spec &= Spec0 \sqcap Spec1 \\
Spec0 &= Proceed(\{w.E\}, Spec) \\
Spec1 &= w.E \rightarrow Spec2 \sqcap Spec3 \sqcap Spec4 \sqcap Spec5 \sqcap Spec6 \\
Spec2 &= Proceed(\{w.E, w.C\}, Spec7 \sqcap Spec1) \\
Spec3 &= w.C \rightarrow Spec \\
Spec4 &= w.E \rightarrow Spec2 \sqcap Spec4 \sqcap Spec8 \sqcap Spec9 \\
Spec5 &= Proceed(\{w.E, w.G\}, Spec3) \\
Spec6 &= w.E \rightarrow (Spec3) \\
Spec7 &= Proceed(\{w.E, w.C\}, Spec) \\
Spec8 &= Proceed(\{w.E, w.C, w.G\}, Spec3) \\
Spec9 &= w.E \rightarrow Spec3
\end{aligned}$$

The administrations of Epirubicin and Cyclophosphamide in all treatment arms are interleaved, that is, there is no explicit ordering imposed by sequence flows. While process *Spec* only considers the behaviour after the administration of Epirubicin chemotherapy and before that of Cyclophosphamide, we need to also consider the behaviour after the administration of Cyclophosphamide chemotherapy and before that of Epirubicin. We capture this via the following CSP process $Spec'$.

$$\begin{aligned}
Spec' &= Spec'0 \sqcap Spec'1 \\
Spec'0 &= Proceed(\{w.C\}, Spec') \\
Spec'1 &= w.C \rightarrow Spec'2 \sqcap Spec'3 \sqcap Spec'4 \sqcap Spec'5 \sqcap Spec'6 \\
Spec'2 &= Proceed(\{w.C, w.E\}, Spec'7 \sqcap Spec'1) \\
Spec'3 &= w.E \rightarrow Spec' \\
Spec'4 &= w.C \rightarrow Spec'2 \sqcap Spec'4 \sqcap Spec'8 \sqcap Spec'9 \\
Spec'5 &= Proceed(\{w.C, w.G\}, Spec'3) \\
Spec'6 &= w.C \rightarrow (Spec'3) \\
Spec'7 &= Proceed(\{w.C, w.E\}, Spec') \\
Spec'8 &= Proceed(\{w.C, w.E, w.G\}, Spec'3) \\
Spec'9 &= w.C \rightarrow Spec'3
\end{aligned}$$

We verify the diagram's behaviour compositionally by considering each treatment arm individually. Moreover, it is sufficient to verify the behaviour of each treatment arm by considering its procedural workflow, and since treatment Arm A does not contain any application of Gemcitabine, we only need to consider arms $B1$ and $B2$. By Equivalence 8.15 we have the following logical equivalence, where process terms *Trial*, $B1PW$ and $B2PW$ are defined as before. Irrelevant behaviours are abstracted by hiding the set of events $H = \Sigma \setminus \alpha Spec$.

$$\begin{aligned}
&Spec \sqsubseteq_{\mathcal{F}} Trial \setminus H \wedge Spec' \sqsubseteq_{\mathcal{F}} Trial \setminus H \Leftrightarrow \\
&\quad Spec \sqsubseteq_{\mathcal{F}} B1PW \setminus H \wedge Spec' \sqsubseteq_{\mathcal{F}} B1PW \setminus H \wedge Spec \sqsubseteq_{\mathcal{F}} B2PW \setminus H \wedge Spec' \sqsubseteq_{\mathcal{F}} B2PW \setminus H
\end{aligned}$$

Similar to the verification of Requirement N1, we verify the behaviour of the BPMN diagram in Figure 8.9 by checking individual assertions at the right hand side of the equivalence. We first consider whether process $B1PW \setminus H$ is a failure refinement of *Spec*. This refinement does not hold and the trace $\langle w.G, w.C \rangle$ is given as a counterexample. This counterexample shows the possibility to administer Gemcitabine after Epirubicin and before Cyclophosphamide. Again deciding how to correct this behaviour would require clinical insight.

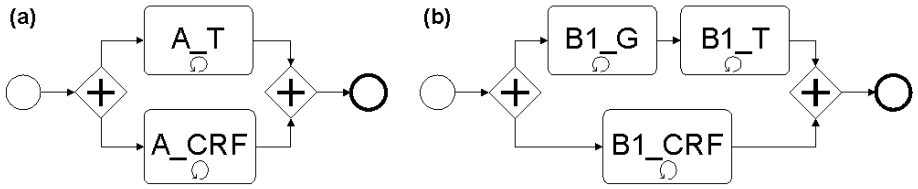


Figure 8.11: Abstraction of the BPMN model of treatment arm (a) A and (b) $B1$

8.3.3.4 Requirement N3

Requirement N3 states that after every application of Paclitaxel, data must be collected via case reporting. Here we express this requirement as the following CSP process $Spec$, where events $w.T$ and $w.S$ model administering Paclitaxel and case reporting respectively.

$$\begin{aligned} Spec &= Spec0 \sqcap Spec1 \\ Spec0 &= Proceed(\{w.T\}, Spec0 \sqcap Spec1) \\ Spec1 &= w.T \rightarrow Spec2 \\ Spec2 &= w.S \rightarrow Spec0 \sqcap Spec1 \end{aligned}$$

The verification of the diagram's behaviour against Requirement N3 is achieved by verifying the behaviour of individual subprocesses. Note that this requirement concerns the behaviour of both procedure (Paclitaxel chemotherapy) and observation (case reporting), therefore it is necessary to consider the behaviour of both workflows together. However, it is sufficient to verify the behaviour of each treatment arm by considering the interleaving behaviour of the Paclitaxel chemotherapy procedure and case reporting. To illustrate this, consider the subprocess in Figure 8.10 on Page 164, which models sequence rule A. We observe that the behaviour of multiple instances tasks A_C (Cyclophosphamide chemotherapy) and A_E (Epirubicin chemotherapy) do not interfere with the behavioural constraints between tasks A_T and A_CRF. Figure 8.11(a) shows how we abstract behaviour of the BPMN subprocess A in Figure 8.10. For subprocesses $B1$ and $B2$, however, it is necessary to consider the behaviour of $B1\_G$ and $B2\_G$ (Gemcitabine chemotherapy) as this behaviour is a dependency for the behaviour of $B1\_T$ and $B2\_T$ respectively. For example, Figure 8.11(b) shows how we abstract behaviour of the BPMN subprocess $B1$. By Equivalence 8.15, we have the following logical equivalence, where processes AS, $B1S$ and $B2S$ describe the behaviour of the abstracted versions of BPMN subprocesses A, $B1$ and $B2$ respectively.

$$Spec \sqsubseteq_{\mathcal{F}} Trial \setminus H \Leftrightarrow Spec \sqsubseteq_{\mathcal{F}} AS \setminus H \wedge Spec \sqsubseteq_{\mathcal{F}} B1S \setminus H \wedge Spec \sqsubseteq_{\mathcal{F}} B2S \setminus H$$

We verify the behaviour of the BPMN diagram in Figure 8.9 by checking individual assertions at the right hand side of the equivalence. We first consider if process $AS \setminus H$ is a refinement of $Spec$. This refinement does not hold. The trace $\langle w.S, w.T, w.T \rangle$ is returned as a counterexample by FDR. This trace shows the possibility of administering two dosages of Paclitaxel without case reporting. Clinical insight is again required to choose appropriate correction to this behaviour.

8.4 Summary

In this chapter we presented two case studies. The first case study was a collaborative business process describing an airline ticket reservation system, in which we investigated the compatibility and compositional development. The second case study was an empirical workflow specification based on a phase III breast cancer clinical trial. We studied the modelling of the trial protocol using our empirical workflow, the specification of oncological safety requirements and the verification of the protocol against these requirements by applying the relative timed extension, compositional reasoning and model checking.

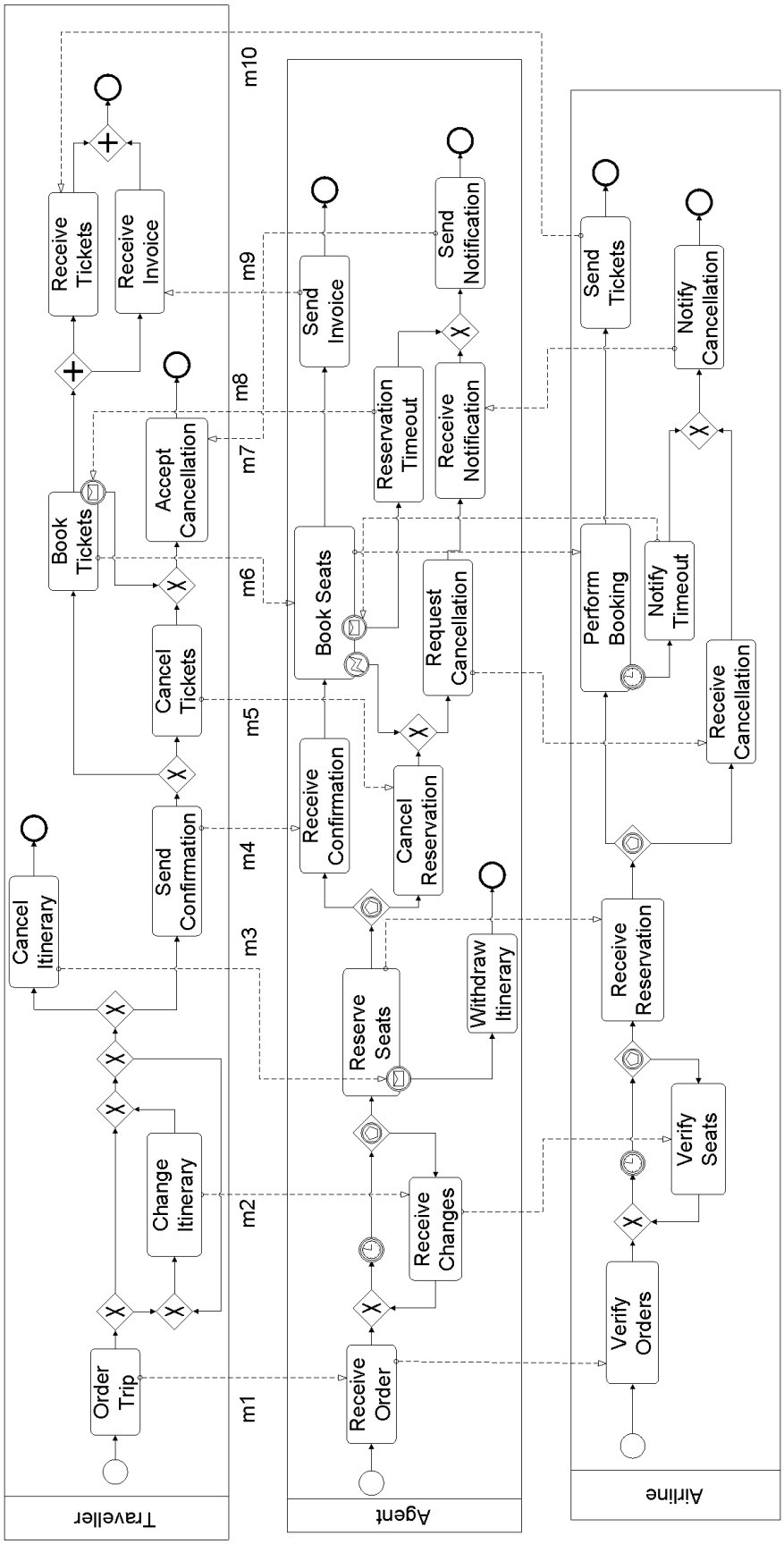


Figure 8.12: Airline Ticket Reservation

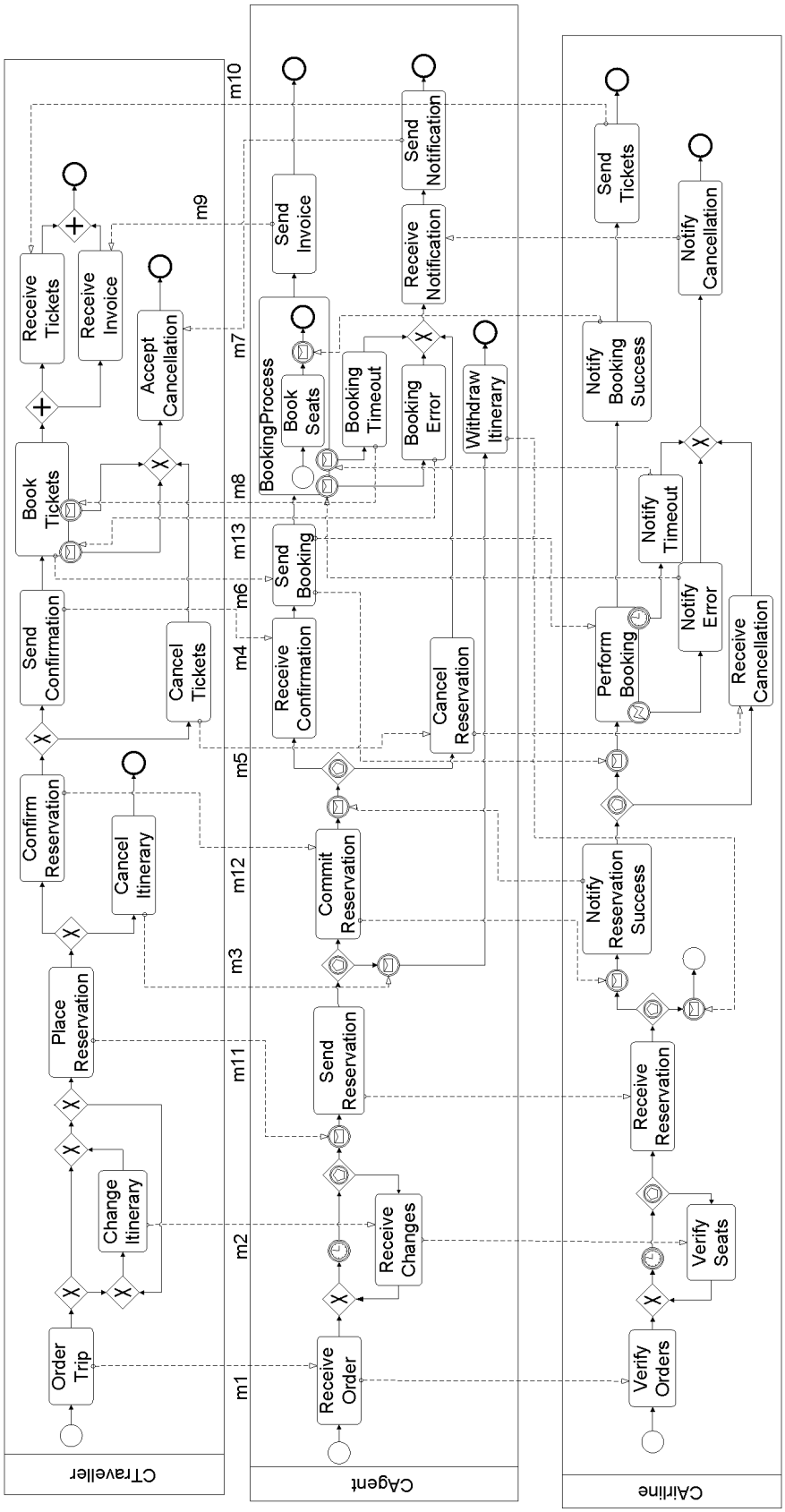


Figure 8.13: Corrected Airline Ticket Reservation

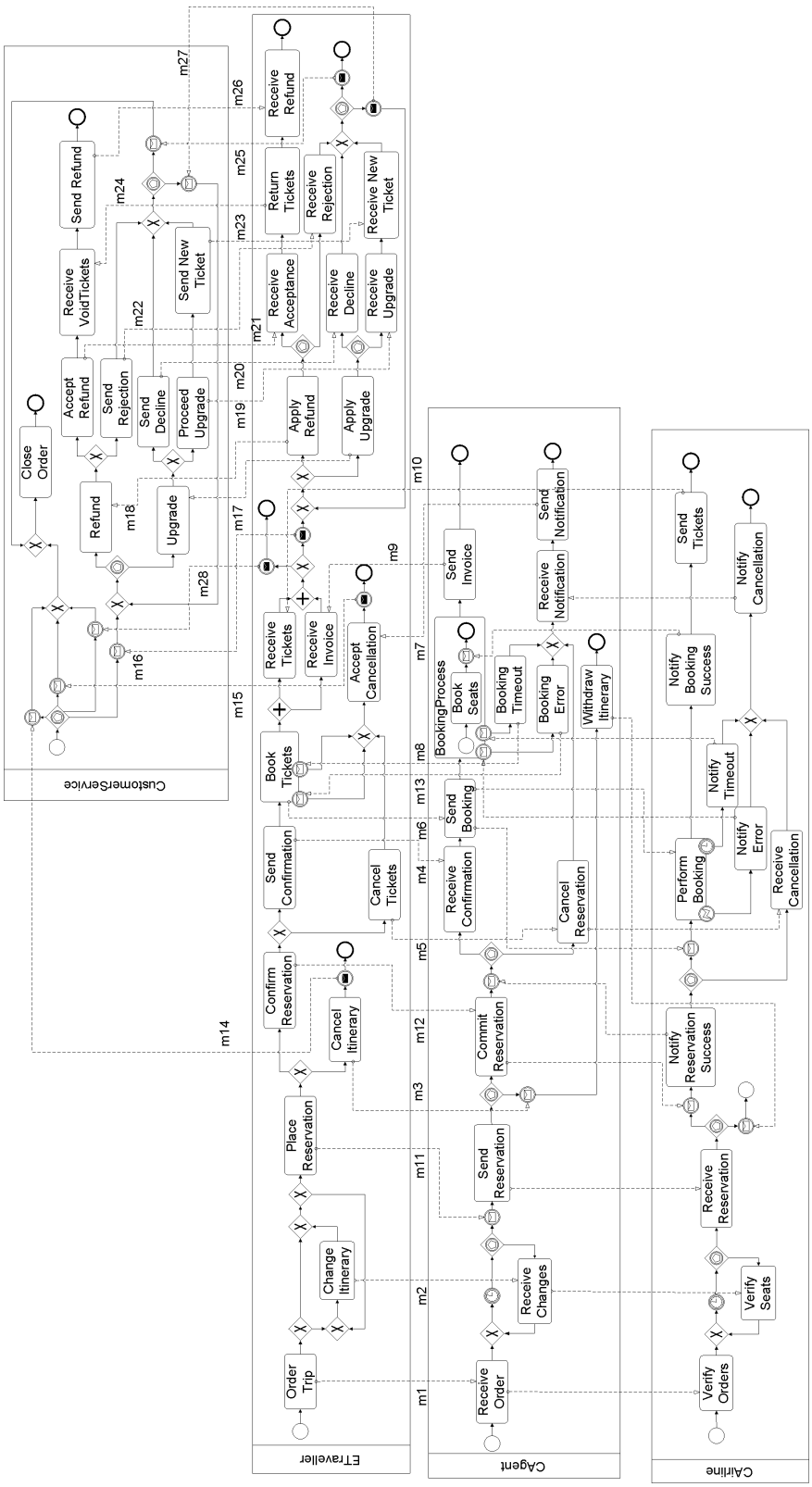


Figure 8.14: Extended Airline Ticket Reservation

Chapter 9

Formalising BPMN

This chapter reports some of the current approaches to formalising BPMN. Comparisons between them and our formalisations are given where possible. Related research efforts on BPMN have been aiming to provide a suitable formalisations for BPMN. At the time of our development of BPMN semantics, the most prominent ones in this area included Dijkman et al.'s Petri net semantics [DDO08], and Dumas et al.'s rule-based semantics focusing on BPMN's inclusive (OR) join gateway [DGHW07]. Since then, there have been other attempts to provide alternative semantic models to BPMN. These include, but are not limited to, Arbab et al.'s formalisation in Reo [AKS08, KA09], Prandi et al.'s translation from BPMN to Calculus for Orchestrating Web Services (COWS) [PQZ08], and Ye et al.'s translation from BPMN to Yet Another Workflow Language (YAWL) [YSSW08].

This chapter is structured as follows. In Sections 9.1, 9.2, 9.3 and 9.4 we discuss related work on formalising BPMN using Petri nets, YAWL, Reo, COWS respectively. We summarize the contribution of this chapter in Section 9.5.

9.1 Petri nets

Petri nets originated from the early work of Petri [Pet62]. Petri nets have been used quite extensively to study the behaviour of workflow [vdAtHKB03, vdAtH05, Kie02, DDO08]. Specifically a Petri net is a directed bipartite graph with two node types called *places* and *transitions*. Places are denoted by circles such that each place can contain a non-negative number of tokens; transitions are denoted by rectangles. The nodes are connected via directed arcs. Here we present the formal definition of Petri nets [vdA97].

Definition 9.1. ***Petri Net.*** *A Petri net is a triple $N = (P, T, F)$ where*

- *P is a finite set of places;*
- *T is a finite set of transitions ($P \cap T = \emptyset$);*
- *$F \subseteq (P \times T) \cup (T \times P)$ is a set of arcs (flow relation).*

Place p is called an input place of transition t if and only if there exists a directed arc $(p, t) \in F$ from p to t. Place p is called an output place of transition t if and only if there exists a directed arc from t to p. The set $\bullet t$ denotes the set of input places for transition t . The sets $t\bullet$, $\bullet p$ and $p\bullet$ hence have similar meanings. A Petri net is strongly connected if and only if, for every pair of nodes $x, y \in P \cup T$, there is a directed path leading from x to y.

During the execution of a Petri net, each place holds zero or more tokens, and a state of a Petri net, called a marking, is a function from each place in the net to a number of tokens it holds. In a given marking, a transition t is enabled if every input place in $\bullet t$ has at least one token; firing t removes a token from every input place in $\bullet t$ and adds a token to every output place in $t\bullet$. The state of the Petri net transitions from one to the next by firing any one of the enabled transitions. A marking of a Petri net is dead if it does not enable any transition, and a transition in a Petri net is dead if and only if the net has no marking that enables it.

Dijkman et al. [DDO08] provide a semantics for BPMN in Petri nets. In particular, they provide a semantics that maps BPMN pools into Workflow nets. A Workflow net is a Petri net such that there is a unique source place i ($\bullet i = \emptyset$), a unique sink place o ($o\bullet = \emptyset$), and every other place and transition is on a directed path from the unique source place to the unique sink place.

Definition 9.2. ***Workflow net.*** *A Petri net $N = (P, T, F)$ is a Workflow net if and only if:*

- *N has two special places: i and o. Place i is a source place: $\bullet i = \emptyset$. Place o is a sink place: $o\bullet = \emptyset$.*
- *If we add a transition t to N which connects place o with i (i.e. $\bullet t = \{o\}$ and $t\bullet = \{i\}$), then the resulting Petri net is strongly connected.*

There are similarities and differences between their semantics and that presented in Chapter 5. Specifically, their Petri net semantics models behaviour of tasks, events, gateways and message flows to be instantaneous, and message flows to be synchronous communications between BPMN pools. A BPMN task or an intermediate event is mapped to a transition with one input place and one output place, where the transition models the execution of the element. A start or an end event is mapped to a similar set of nodes where the transition is used to signal when the process starts or ends. AND and data-based XOR gateways are mapped to Petri-net nodes with silent transitions capturing their routing behaviour. For example, an AND split gateway is mapped to a silent transition t that has a single input place $\#\bullet t = 1$ and multiple output places $\#t\bullet > 1$, where each output place models an outgoing sequence flow of the gateway. A data-based XOR split gateway, on the other hand, is mapped to several silent transitions ts such that each $t \in ts$ has a single output place $\#t\bullet = 1$ modelling an outgoing sequence flow of the gateway and all silent transitions in ts have a common single input place $\#\bullet t = 1$ modelling the exclusive choice over outgoing sequence flows. These decisions are the same as those we have made for our process semantics; they give rise to a suitable level of abstraction for reasoning about interaction behaviour between BPMN elements.

There are also differences: Dijkman et al.'s Petri net semantics models multiple instance activities implicitly by translating each multiple instance activity into individual activities and gateways. Our semantic definition, on the other hand, models the behaviour of multiple instance activities directly. Moreover, Dijkman et al. only consider multiple instance activities with a fixed numbers of instances, that is, the number of instances is known before the execution of the activity. In contrast, our semantics permits both fixed and nondeterministic numbers of instances, modelling situations in which the number of instances is only known at run time. Note that multiple instance activity elements with nondeterministic number of instances cannot be implicitly modelled compositionally using activity and gateway elements.

BPMN processes, which are translated to Petri nets using Dijkman et al.'s semantics, can be mechanically verified using the process mining tool ProM [vDdMV+05] against general behavioural properties such as deadlock/livelock freedom and the absence of dead tasks. A BPMN process is deadlock free if its corresponding Petri net has no dead marking, while a BPMN process has no dead task if the corresponding Petri net has no dead transition. Similarly, using the FDR tool, our process semantics also permits mechanical verification of BPMN processes against these general behavioural properties; general behavioural properties are modelled as CSP processes and verification corresponds to refinement checks.

Extensive research works have been carried to investigate properties of Petri nets and Workflow nets. For example, Workflow nets have been analysed with respect to liveness, boundedness and soundness properties [vdA97, vdAvHtH+10]. Furthermore a timed formalisation of workflow nets using timed Petri nets has subsequently been provided by Ling et al. [LS00]. While it would be interesting to see how these properties can be exploited in the context of BPMN, they were not considered in Dijkman et al.'s formalisation [DDO08]. Moreover, neither compositional reasoning nor the notion of behavioural compatibility were considered in their work.

9.2 YAWL

Yet Another Workflow Language (YAWL), a workflow language developed by van der Aalst et al. [vdAtH05], is based on the investigation of workflow patterns [vdAtHKB03] and the expressiveness of existed workflow modelling notations. Its semantics is provided in terms of extended Workflow nets. Here we present the formal definition of an extended Workflow net [vdA97].

Definition 9.3. ***Extended Workflow net.*** *An extended Workflow net N is a tuple $(C, i, o, T, F, split, join, rem, nofi)$ such that:*

- *C is a set of conditions;*
- *$i \in C$ is the input condition;*
- *$o \in C$ is the output condition;*
- *T is a set of tasks;*
- *$F \subseteq (C \setminus \{o\} \times T) \cup (T \times C \setminus \{i\}) \cup (T \times T)$ is the flow relation; every node in the graph $(C \cup T, F)$ is on a directed path from i to o;*
- *$split : T \to \{AND, XOR, OR\}$ specifies the split behaviour of each task;*
- *$join : T \to \{AND, XOR, OR\}$ specifies the join behaviour of each task;*
- *$rem : T \to \mathbb{P}(T \cup C \setminus \{i, o\})$ specifies additional tokens to be removed by emptying a part of the workflow; and*
- *$nofi : T \to \mathbb{N} \times \mathbb{N}^{inf} \times \mathbb{N}^{inf} \times \{dynamic, static\}$ specifies the multiplicity of each task (minimum, maximum, threshold of continuation, and dynamic/static creation of instances).*

An extended Workflow net is composed of conditions and tasks. In terms of Petri nets (Definition 9.1), conditions correspond to places and tasks correspond to transitions. In particular from the definition of an extended Workflow net, the tuple (C, T, F) corresponds to a Petri net. Similar to Workflow nets (Definition 9.2), an extended Workflow net has a unique input and output conditions. Unlike Petri nets, the flow relation F of an extended Workflow net can connect two tasks directly. In terms of Petri nets, this is interpreted as placing a "hidden" place between the two transitions. Functions *split* and *join* specify split and join behaviour of each transition, *rem* specifies the additional tokens to be removed by emptying a part of the Workflow net, and *nofi* specifies the multiplicity of each transition.

Extended Workflow nets provide the expressivity for modelling workflows containing multiple instance, inclusive splits/joins and cancellation patterns [vdAtHKB03]; these patterns have been known to be difficult to model with existing modelling notations [vdAtH05].

In YAWL, a task is either atomic or composite, and a workflow specification in YAWL is a set of extended Workflow nets, such that each composite task refers to a unique extended Workflow net in that set. In formal analysis, van der Aalst et al. [vdAtH05] provide some compositionality results on the soundness property of YAWL [vdA97]. Specifically, a workflow net is sound if and only if it terminates without deadlock and is absence of dead tasks. van der Aalst et al. showed that YAWL is compositional with respect to soundness: a workflow specification in YAWL is sound if and only if each extended Workflow net in the the workflow specification's set is sound.

Ye et al. [YSSW08] provided a translation from BPMN to YAWL. Unlike other formalisms that have been used to model BPMN, YAWL is itself a workflow language and many constructs in YAWL fit naturally with BPMN elements and objects. This is reflected in Ye et al.'s approach. Specifically, a YAWL workflow specification represents a BPMN diagram; YAWL atomic tasks correspond to BPMN task elements; composite tasks correspond to subprocesses; functions *split* and *join* correspond to the behaviour of AND, XOR and OR split/join gateways; function *rem* corresponds to the behaviour of exception flows; and function *nofi* corresponds to the behaviour of multiple instance activities. Note that the distinction between sequence and message flows, as well as the concept of pools are not explicit. Unlike the other formalisation presented in this chapter, Ye et al. also consider the semantics of ad-hoc subprocesses by mapping them to YAWL's composite tasks.

To enable automatic verification of BPMN processes modelled in YAWL, Ye et al. developed a BPMN2YAWL plugin for the process mining tool ProM [vDdMV$^+$05]. While YAWL offers a compositional way to verify YAWL workflow with respect to the soundness properties [vdA97], this compositionality result has not been applied in Ye et al.'s mapping. Furthermore, neither any other compositional reasoning nor the notion of behavioural compatibility was considered in their approach.

9.3 Reo

Reo [Arb04] is a formal language for constructing channel-based models that exhibit exogenous coordination. In Reo, a system consists of a number of components executing at one or more locations, communicating through connectors that coordinate their activities. Each connector in Reo is constructed compositionally out of simpler connectors which are in turn composed out of channels.

Specifically a channel has two directed ends, through which components refer to and manipulate that channel and the data it carries. There are two types of channel ends: sources and sinks. A source channel end accepts data into its channel. A sink channel end dispenses data out of its channel. Typical channel types include FIFO1, synchronous and lossy synchronous: a FIFO1 channel represents an asynchronous channel with one buffer cell that can hold one data item; a synchronous channel has a source and a sink ends with no buffer. It accepts a data item through its source end if and only if it can simultaneously dispense it through its sink, and a lossy synchronous channel is similar to synchronous channel except that it always accepts data items from its source and a data item is transferred if it can be dispensed through the channel's sink, otherwise the item is lost.

The semantics of Reo is based on Constraint Automata. A Constraint Automaton, introduced by Baier et al. [BSAR06], is conceptually a generalization of probabilistic automata where data constraints, instead of probabilities, influence applicable state transitions. We present a formal definition of Constraint Automata from Baier et al.'s work [BSAR06].

Definition 9.4. ***Constraint Automata.*** *A constraint automaton is a tuple* $A = (S, S_0, N, \rightarrow)$ *where*

- S *is a set of states*
- $S_0 \subseteq S$ *is the set of initial states*
- N *is a set of names*
- $\rightarrow \subseteq (S \times \mathbb{P}\, N \times G \times S)$ *is the transition relation of A, where G is possible guards of the transition in the relation. For* $(q, n, g, p) \in \rightarrow$, $n \in \mathbb{P}\, N$ *is the name-set and* $g \in G$ *the guard of the transition.*

There exist several extensions to Constraint Automata for modelling and automated analysis of time, resource and QoS aspects of Reo coordination models [ACMM07, ABdBR07].

Arbab et al. [AKS08] provide a translation from BPMN to Reo. Due to the extensions to Reo's Constraint Automata semantics, this translation permits formal reasoning about BPMN processes against time and resource properties.

In their Reo semantics, the interactions between BPMN elements are modelled as data flows through Reo channels. A BPMN task is modelled in Reo as a simple FIFO1 channel, where data flow in the source end of the channel corresponds to the start of the task, data flow in the sink end of the channel corresponds to the end of the task, while the data token residing in the channel buffer implies that the task is being executed. A subprocess is modelled using a Reo connector that preserves the number of the subprocess' incoming and outgoing flows. An event with no trigger is modelled as a single Reo node, while events with triggers are modelled using Reo channels. For example, a message event with an incoming message flow is modelled as a synchronous drain. An AND split gateway is composed of multiple diverging synchronous channels, each represents one of the gateway's outgoing sequence flows. An AND join gateway consists of multiple synchronous channels representing the gateway's incoming sequence flows; these channels are synchronized using a synchronous drain, while the gateway's outgoing sequence flow is modelled using lossy synchronous channels.

The Reo translation described so far yields the same level of abstraction as our process semantics in which the focus is on the behaviour due to the interaction between elements in a BPMN process. We now consider a number of differences in their modelling decision. In their Reo semantics, a data-based XOR split gateway is modelled using a synchronous channel to model the gateway's incoming flow, and multiple filter channels with a common source model its outgoing flows. Filter transition conditions are defined by Boolean expressions that are used to determine which outgoing sequence flow to take. Data dependence allows one to specify exactly the condition under which an outgoing sequence flow is triggered, however, this means it is important to ensure the Boolean conditions are mutually exclusive, otherwise it can lead to deadlocks or dead tasks. In our process semantics, the choice between outgoing

sequence flows of a XOR split gateway is modelled using the nondeterministic choice operator, thereby abstracting process data and ensuring only one flow is triggered.

Nevertheless, using data dependent Boolean conditions, an OR split gateway can be modelled similarly to the data-based XOR gateway by relaxing the need for the Boolean conditions to be mutually exclusive. Again, incorrect Boolean conditions can lead to deadlock or dead tasks. Additionally, using Reo, various complex gateways can be constructed. Due to data abstraction OR split gateway and other complex gateways were not considered in our formalisation.

Another difference in their modelling decisions is to allow both synchronous and asynchronous communications via message flows between BPMN pools. This provides a high level of flexibility to model communications between participants in the business collaboration. However, the authors did not clearly present how to reason about the behaviour of BPMN processes compositionally in the presence of both synchronous and asynchronous communications.

In their more recent work ([KA09]), they generalised the representation of business processes and presented a framework for modeling and verifying service-based business processes against external compliance regulations such as Segregation of Duties (SoD) and privacy protection policies. The verification of a (BPMN) business process modelled in Reo against compliance regulations is achieved using the Vereofy model checker [BBKK09]. One of the input languages of Vereofy is the Reo Scripting Language (RSL), a textual version of Reo. Vereofy supports linear and branching-time model checking. Compliance regulations about the business process are specified as behavioural properties about the Reo circuit that models the business process. These properties can be expressed in several formalisms such as Linear Temporal Logic (LTL) and Computation Tree Logic (CTL). Similar to their analysis approach, we have considered the problem of property specification for business processes using temporal logic [WG11b], in particular we generalised Dwyer et al.'s Property Specification Patterns (PSP) [DAC99], and translated PSP properties into a bounded positive fragment of LTL [Low08]; these LTL formulae are then automatically translated into CSP processes for simple refinement checks using the FDR tool [Low08].

Alternatively, a compliance requirement can be seen as an informal description of the characteristic business process satisfying the requirement, which can be formally modelled as a Reo model. The verification of business process modelled in Reo against this requirement can then be achieved by checking bisimulation equivalence between their Reo models. Similarly, in this thesis a behavioural property about a BPMN processes is specified as a CSP process and the verification is achieved by using the FDR tool to automatically check if the CSP process modelling the behaviour of the BPMN process refines that expressing the behavioural property. Note that neither compositional reasoning nor the notion of behavioural compatibility was considered for Arbab et al.'s Reo semantics.

9.4 COWS

The Calculus of Orchestrating Web Services (COWS) [LPT07] is a process algebra that combines elements well-known to process algebras with constructs in BPEL [BPE03]. The computation unit of COWS is a service. Here we provide the grammar of the COWS language [LPT07].

$$s ::= u!w \mid [d]s \mid g \mid s \parallel s \mid \{\!|s|\!\} \mid \mathbf{kill}(k) \mid s$$
$$g ::= \mathbf{0} \mid p?w.s \mid g + g$$

Specifically, $u!w$ is a service that invokes an activity over endpoint u with parameter w; $[d]s$ denotes the scope of d to be s; g is a guarded command, which can either be the empty activity $\mathbf{0}$, or a service $p?w.s$ that halts until communication is received via endpoint p with the possible instantiation of parameter w before proceeding to become s, or a choice between two guarded commands $g + g$. $s \parallel s$[1] is the parallel composition of two services; $\{\!|s|\!\}$ protects service s from termination while $\mathbf{kill}(k)$ forces termination of unprotected parallel services that are in the scope of k, $*s$ is the replication of s and behaves as $*s \parallel s$. High level imperative constructs such as conditional statements and sequential composition can be encoded using COWS primitive operators as follow,

$$\begin{aligned} \mathbf{if}\ c\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2 &= [p](p!\hat{c} \parallel (p?\mathbf{true}.s_1 + p?\mathbf{false}.s_2)) \\ s_1;\ s_2 &= [p_{s_1\_done}](s1 \parallel p_{s_1\_done}?.s_2) \end{aligned}$$

[1] The original grammar uses $s \mid t$ to denote the parallel composition of s and t, we have replaced it with $s \parallel t$ to avoid clashes with the BNF |.

where $\hat{c}$ evaluates the condition c and can either assume the value **true** or the value **false**. For sequential composition, p_{s_1-done} is a special endpoint for termination of service s_1. COWS has an operation semantics and is given in terms of a structural congruence and a labelled transition relation [LPT07].

Prandi et al. [PQZ08] provided a translation from a subset of BPMN to COWS. Their choice of BPMN elements to model is a subset of that considered in this thesis with the addition of OR split and join gateways. Specifically, they do not consider subprocesses and multiple instances. Similar to our process semantics, they model a BPMN process as a parallel composition of COWS services, each corresponding to the behaviour of a BPMN element contained in that process. Incoming sequence and message flows are modelled as request activities, while outgoing sequence and message flows are modelled as invoking activities. A task with an incoming sequence flow i, outgoing sequence flow o and incoming message flow m is modelled as the service $*([w]i?.m?w.o!)$; an AND split gateway with an incoming sequence flow i, and two outgoing sequence flows $o1$ and $o2$ is modelled as the service $*(i?.(o1! \parallel o2!))$; and an AND join gateway with incoming sequence flows $i1$ and $i2$, and outgoing sequence flow o is modelled as the service $*((i1? \parallel i2?);\ o!)$. Similar to Reo's semantics presented in Section 9.3, Prandi et al.'s semantics associates sequence flows of choice gateways with conditional statements. A XOR split gateway with incoming sequence flow i, and two outgoing sequence flows $o1$ and $o2$ is modelled as service $*(i?.(\textbf{if}\ c1\ \textbf{then}\ o1!\ \textbf{else}\ (\textbf{if}\ c2\ \textbf{then}\ o2!)))$ where $c1$ and $c2$ are conditions associated with flows $o1$ and $o2$ respectively. An OR split gateway with incoming sequence flow i, and two outgoing sequence flows $o1$ and $o2$ is modelled as service $*(i?.(\textbf{if}\ c1\ \textbf{then}\ o1! \mid (\textbf{if}\ c2\ \textbf{then}\ o2!)))$. Similar to the Reo semantics, incorrect conditions can lead to deadlock or dead tasks. Moreover, for the XOR split gateway, if conditions $c1$ and $c2$ are not exclusive and are true at the same time, outgoing sequence flow $o1$ would always be chosen according to Prandi et al.'s COWS semantics.

In their paper [PQZ08] Prandi et al. considered quantitative analysis of BPMN processes using a stochastic extension of COWS [PQ07]. Using this extension, COWS service modelling the behaviour of a BPMN process can also be expressed as Continuous Time Markov Chains for analysis. Specifically, each basic action is associated with a random duration governed by a negative exponential distribution that is characterized by a unique rate r, therefore the probability that an action is performed within a period of time of length t is $1 - e^{-rt}$. To enable automated analysis of BPMN processes modelled in COWS, Prandi et al. implemented the semantics of COWS in PRISM [HKNP06], a stochastic model checker. Note that neither compositional reasoning nor the notion of behavioural compatibility was considered in Prandi et al.'s formalisation.

9.5 Summary

In this chapter we looked at some of the current approaches to formalise BPMN. In Section 9.1 we looked at Dijkman et al.'s Petri nets approach; in Section 9.2 we considered van der Aalst et al.'s YAWL workflow language and Ling et al.'s mapping from BPMN to YAWL; in Section 9.3 we presented Arbab et al's Reo coordination language and their mapping from BPMN to Reo, which facilitated the analysis of BPMN process with respect to compliance requirements; and in Section 9.4 we presented Lapadula et al.'s process algebra COWS and Prandi et al.'s translation from BPMN to COWS, which facilitated quantitative analysis of BPMN processes.

Chapter 10

Conclusion

In this final chapter we reflect on the contributions presented in this thesis. We first provide a summary and draw some conclusions; we discuss some limitation of our research and propose possible directions in future research to overcome them.

10.1 Summary of Contributions

In this thesis we have provided a CSP-based formal framework for declaratively and graphically specifying both *service-centric systems* and *empirical studies* described in Business Process Modelling Notation, and verifying these workflow processes against abstract behavioural properties via automatic refinement checking. We achieved this by formalising the syntax of BPMN using the Z schema language and defining two new process semantic models on that syntax in the language of CSP. One considers purely untimed behaviours of BPMN diagrams, while the other one extends the first one with timing information. We exploited CSP's process-based specification and refinement orderings to capture abstract behavioural properties, against which workflow processes described in BPMN may be verified.

In Chapter 4 we studied in detail the syntax as well as the informal semantics of BPMN, and provided a formalisation of a BPMN subset in Z. We studied several syntactic operations for constructing BPMN diagrams, provided corresponding Z formalisations, and using which we investigated their logical preconditions.

In Chapter 5 we considered an untimed process semantic model for BPMN. In this semantics, each BPMN element is represented as a CSP process, where each element's sequence flows and message flows are represented as CSP events. Consequently each BPMN diagram may be represented as a parallel composition of CSP processes where the flow of control between elements is modelled by synchronising the shared interface of the parallel composition. This semantics permits hierarchical composition allowing formal reasoning at various levels of abstractions as well as semantic comparisons between BPMN diagrams via CSP's refinement. We studied the semantics and the compositionality of the syntactic operations defined in Chapter 4. Specifically we showed these operations to be monotonic with respect to CSP's failures refinement – this encourages compositional development of workflows.

We also considered Reed et al.'s CSP theory of responsiveness for interoperating components in a complex system and using which we developed a formal notion of behavioural compatibility in the context of our semantic model. We were able to show that compatibility between deadlock-free business processes ensures their interaction to be deadlock free. This property was then extended such that the composition of a compatible business process to a deadlock-free business collaboration guarantees the overall collaboration to be also free of deadlock.

In Chapter 6 we introduced a timed model for BPMN by augmenting its untimed counterpart with the notion of relative time in the form of delays chosen non-deterministically over a double-bounded range. This model adopts a variant of the two-phase functioning approach widely used in real-time systems and timed coordination languages [LJBB06]. We have formalised the coordination procedure, and established a formal relationship between the timed coordination and the untimed enactment processes for BPMN. Specifically we have shown the coordination procedure yields coordination processes that do not cause their enactment counterpart to deadlock. Using this model, BPMN diagrams could be used to describe concurrent behaviour with timing restrictions.

Chapter 7 studied the application of BPMN and the semantic models defined in this thesis to the specification and analysis of empirical studies. We achieved this by defining a generic workflow model `Empiricol` for modelling empirical studies declaratively; specifically this model generalised the workflow model implemented in the CancerGrid trial model [CHG+07]. We provided bidirectional transformation functions between `Empiricol` and BPMN. The transformation from BPMN to `Empiricol` provided a

medium for empirical studies to be specified graphically as workflows, while transforming `Empiricol` to BPMN permits graphical visualisation, simulation and verification of empirical studies.

In Chapter 8, two comprehensive case studies were presented. The first one was on a collaborative business process describing an airline ticket reservation system, in which we considered compatibility between participants in the business process and verified behavioural properties using a combination of refinement checking and compositional reasoning. The second one was on the empirical workflow specification of a phase III breast cancer clinical trial. We studied the modelling of the trial protocol using `Empiricol`, the specification of oncological safety requirements and the verification of the protocol against these requirements by employing our relative timed extension, compositional reasoning and refinement checking.

10.2 Discussions

10.2.1 Process Semantics

During the development of the CSP-based semantic model for BPMN described in Chapter 5, we had to first choose and formalise the syntax of a subset of BPMN; this was provided in Chapter 4. By choosing only a subset of BPMN, we were able to focus on a particular aspect of workflow behaviour. Specifically we focused on the control flow behaviour of BPMN. We denote the behaviour of a BPMN diagram by a parallel composition of CSP processes, each modelling the semantics of individual BPMN elements contained in the diagram. While our approach allows compositional modelling and reasoning, it turned out to be relatively more difficult to model the control flow behaviour of termination and exception flows. This is because to model termination and exception flows we are required to use the CSP interrupt operator. The use of the interrupt operator considerably increases the state space of the CSP model and hence would make model checking less efficient. In retrospect, should we have chosen another process-algebraic model, such as π-calculus or ACP, and provided a compositional semantic definition similar to the one defined in this thesis, the same issue would arise. On the other hand, we could have considered a Petri net-based approach similar to that of Dijkman et al.'s [DDO08]. However, as demonstrated by Dijkman et al.'s model [DDO08, Figure 9], a Petri net model of exception flows would be complex. Moreover a Petri-net model does not lend itself to compositional reasoning. Furthermore, none of these formalisms provide natural refinement orders which we could exploit to allow formal comparison of BPMN diagrams. While we might have possibly overcome this issue by providing a semantics directly to BPMN, we did not choose this option. This is because we aim to develop a framework for reasoning about BPMN models automatically and mechanically, and existing formalisms would provide the necessary foundational theories and associated tool support.

10.2.2 Modelling Relative Time

In this thesis we have chosen to study relative timed behaviour of BPMN processes. We have provided an extension to the process semantics to model relative timed behaviour of BPMN processes. This has allowed us to a) reuse the process semantic definition and b) to show that given a BPMN diagram, its coordination process is at least a responsive plugin-in to its enactment process. Due to properties of responsiveness, liveness properties such as deadlock freedom may be preserved from the untimed model to the timed one. However, one of the limitations this model has is that it does not consider infinite behaviours. In retrospect, focusing on alternatives that would also have allowed us to study correspondences with the untimed model, a primary candidate formalism, which we could have chosen, is Timed-CSP [Sch00]. Timed-CSP is a timed variant of CSP and has been given a timed failures semantics, which is a continuous timed model and has a natural projection to CSP's failures semantics. Due to Ouaknine's extension of the digitization technique [Oua01], it has become possible to model check Timed-CSP processes using FDR.

10.2.3 Modelling Empirical Studies

In Chapter 7 we studied the application of BPMN and the semantic models for the specification and analysis of empirical studies. In particular we defined a generic empirical studies model `Empiricol` for modelling empirical studies and bidirectional transformation functions between `Empiricol` and BPMN.

While we have chosen Haskell to implement the transformation procedures, other existing transformation languages could suffice to implement the transformation procedures. However, since we have chosen Haskell to implement the semantic models defined in this thesis, Haskell was a natural choice for implementing the transformation functions.

10.3 Limitations and Future Research

10.3.1 Efficiency and Methodology

The semantic models and the specification technique described in this thesis aim to support automatic verification and therefore focus on analyses of workflow processes whose behaviours may be modelled using finite states machines. However, it is well-known that automatic verification techniques such as model checking for finite state systems with many parallel components suffer from the problem of combinatorial explosion. To deal with this problem, several approaches have been taken in the model checking community as a whole; for refinement checking CSP there are binary decision diagrams (BDDs) [Yan96], and hierarchical compression [Ros98]. However, these techniques alone are not enough. Modelling large complex concurrent systems not only requires a thorough understanding of the syntax and semantics of the modelling language used, but also the *methodology* for composing and abstracting models so that formal analyses may be carried out compositionally and cost-effectively. This methodological requirement becomes especially important when considering analyses of workflow processes. The reason is twofold. Firstly workflow designers cannot be assumed to have the required experience to directly apply compositional and abstraction techniques developed in the formal engineering methods community, and secondly for these techniques to be usable for workflow development, they would require to be lifted to the level of the workflow modelling language. As a result, we believe it is necessary to investigate and develop a suitable methodology which applies the core idea of composition and abstraction in formal engineering methods but at the same time is amenable to workflow designers so that they may be applied directly to workflow modelling languages like BPMN.

10.3.2 Executable Semantics

While the semantic models developed in this thesis lend themselves to formal development of workflow processes, they are denotational and not operational. Operational semantics, on the other hand, interpret programs as transition systems, and are relatively closer to implementations. As such an operational semantics based on transition systems might be provided to BPMN as an alternative mathematical formalisation for some implementation strategy. Specifically by giving BPMN an operational semantics that is *congruent* to the models presented in this thesis, it would provide a precise understanding of the execution of BPMN. It also would be possible to carry out compositional reasoning at the (denotational) model level and transfer verified properties to the implementation level. Moreover, an executable semantics for BPMN would allow simulation of business processes and encourage the use of other transition-system-based verification tools for formal analyses. There are a number of verification tools for analysing labelled transition systems. Closely related to the language of CSP are Sun et al.'s Process Analysis Toolkit [SLD08], and Kramer et al.'s Labelled Transition System Analyser [MK99].

10.3.3 Completeness

This thesis only studied a subset of BPMN constructs. In particular we did not consider BPMN's transactional and data flow behaviours. For reasons of completeness, a natural extension to our study would be to investigate the semantics of these constructs. However, to ensure compactness and focus of our models, we believe both transactional and data flow behaviours should be formalised using alternative models. In particular transactional behaviour in BPMN may be modelled using Butler et al.'s Compensating CSP [BHF05, BR05], which has a traces semantics and a congruent operation semantics for modelling compensation actions, while data flow behaviour in BPMN may be modelled using Josephs's Dataflow Sequential Processes [Jos05], which has been given several denotational models in the style of those for CSP. By choosing these variants, one could aim provide a more holistic model of business processes as well as study particular aspects of their behaviours in isolation.

10.3.4 Runtime Verification

Our proposed framework focuses on static verifications; however, runtime verification has also been an important area of research for ensuring correctness of systems [FF95]. It is well known that not all desired requirements of a software system may be verified statically and this is also true even if one could show the system does meet its requirements prior to deployment. This is because during the execution of the system, there could be unpredicted changes to the environment of the system or simply the system might fail to anticipate the behaviour of all the agents, including humans, interacting with it [SM06]. We therefore propose, as future work, to extend our current framework to provide a runtime monitoring facility, and in particular behavioural properties specified during static verification should be reused as the source of requirements to be monitored at run time.

10.3.5 Schedulability

In Chapter 6, we presented a relative timed model for BPMN. In this model each atomic task element is annotated with a minimum and a maximum time such that the task nondeterministically takes a duration bounded by them to complete. This formalisation of timed behaviour could be extended to study the notion of *schedulability*. In schedulability analysis, each task is annotated with a *deadline* and a system can be scheduled if there exists a scheduling strategy such that all possible sequences of activities in the system may be completed within their deadlines. Notable work in formalising schedulability includes Ferman et al.'s task automata approach [FaPPY07]. A task automaton is an extended timed automaton and one of its characteristics is that it may be used to describe tasks that may have interval execution times representing the best case and the worst case execution times.

10.4 Summary

We have presented a summary and drawn conclusions on the contributions made in this thesis. We have also discussed possible directions for future research.

Appendix A

Z Specification of BPMN Syntax

A.1 Preliminaries

This section provides the basic and free type definitions for our Z specification.

$$[TaskName, BName, PoolId, FlowType, Loops, BCondition, Seqflow, Mgeflow]$$

$$Exception ::= exception\langle\langle \mathbb{N} \rangle\rangle \mid anyexception$$

$$Message ::= message\langle\langle Mgeflow \rangle\rangle \mid nomessage$$

$$Range == \mathbb{N} \times \mathbb{N}$$

$$\begin{aligned} Type ::= {} & itime\langle\langle \mathbb{N} \rangle\rangle \mid stime\langle\langle \mathbb{N} \rangle\rangle \mid ierror\langle\langle Exception \rangle\rangle \mid eerror\langle\langle Exception \rangle\rangle \mid \\ & srule\langle\langle BCondition \rangle\rangle \mid irule\langle\langle BCondition \rangle\rangle \mid start \mid end \mid \\ & smessage\langle\langle Message \rangle\rangle \mid imessage\langle\langle Message \rangle\rangle \mid emessage\langle\langle Message \rangle\rangle \mid \\ & agate \mid xgate \mid exgate \mid task\langle\langle TaskName \rangle\rangle \mid subprocess\langle\langle BName \rangle\rangle \mid \\ & miseq\langle\langle TaskName \times (Loops \times FlowType) \rangle\rangle \mid miseqs\langle\langle BName \times (Loops \times FlowType) \rangle\rangle \mid \\ & mipar\langle\langle TaskName \times (Loops \times FlowType) \rangle\rangle \mid mipars\langle\langle BName \times (Loops \times FlowType) \rangle\rangle \end{aligned}$$

A.2 Events

OneInOutFlow

$$ele : Element$$

$$\#(atom\ ele).in = 1 \land \#(atom\ ele).out = 1$$

A.3 Activities

$$Process == \mathbb{F}_1\ Element$$

$$\_ \in_{\mathrm{e}} \_ : Element \leftrightarrow Element$$

$$\forall e, f : Element \bullet e \in_{\mathrm{e}} f \Leftrightarrow (e, f) \in contains^+$$

$$\_ \in_{\mathrm{p}} \_ : Element \leftrightarrow Process$$

$$\forall e : Element;\ es : Process \bullet e \in_{\mathrm{p}} es \Leftrightarrow e \in es \lor (\exists f : Element \bullet f \in es \land e \in_{\mathrm{e}} f)$$

$$outs : Element \to \mathbb{F}\ Seqflow$$

$$\forall e : Element \bullet outs\ e = (atom\ e).out \cup \mathrm{dom}((atom\ e).exit)$$

$$getmsg : Type \nrightarrow Message$$

$$\begin{aligned} & \forall t : Type \bullet \\ & \quad (t \in \mathrm{ran}\ smessage \Rightarrow getmsg\ t = smessage^{\sim} t) \land \\ & \quad (t \in \mathrm{ran}\ imessage \Rightarrow getmsg\ t = imessage^{\sim} t) \land \\ & \quad (t \in \mathrm{ran}\ emessage \Rightarrow getmsg\ t = emessage^{\sim} t) \end{aligned}$$

$$getSd, getRec : Atom \to \mathbb{F}\ Mgeflow$$

$$\begin{aligned}
&\forall\, a : Atom \bullet \\
&\quad (a.type \in \operatorname{ran} emessage \land getmsg\ a.type \neq nomessage \Rightarrow \\
&\qquad getSd\ a = \{message^{\sim}(getmsg\ a.type)\}) \land \\
&\quad (a.type \notin \operatorname{ran} emessage \lor getmsg\ a.type = nomessage \Rightarrow \\
&\qquad getSd\ a = a.send) \land \\
&\quad (a.type \in \operatorname{ran} smessage \cup \operatorname{ran} imessage \land getmsg\ a.type \neq nomessage \Rightarrow \\
&\qquad getRec\ a = \{message^{\sim}(getmsg\ a.type)\}) \land \\
&\quad (a.type \notin \operatorname{ran} smessage \cup \operatorname{ran} imessage \lor getmsg\ a.type = nomessage \Rightarrow \\
&\qquad getRec\ a = \\
&\qquad\quad a.receive \cup \\
&\qquad\quad \{t : \operatorname{ran} a.exit \mid t \in \operatorname{ran} imessage \land getmsg\ t \neq nomessage \bullet message^{\sim}(getmsg\ t)\})
\end{aligned}$$

$$getMsg : Atom \to \mathbb{F}\ Mgeflow$$

$$getMsg = getSd \cup getRec$$

$$errorCode : errorCodeTypes \to \mathbb{N}$$

$$\begin{aligned}
&\forall\, t : errorCodeTypes \bullet \\
&\quad (t \in \operatorname{ran} ierror \Rightarrow errorCode(t) = (exception^{\sim}(ierror^{\sim} t))) \land \\
&\quad (t \in \operatorname{ran} eerror \Rightarrow errorCode(t) = (exception^{\sim}(eerror^{\sim} t)))
\end{aligned}$$

A.4 Pools and Diagrams

$$eventgate == \{Gate \mid (atom\ ele).type = exgate \bullet ele\}$$
$$sendelement == \{ele : Element \mid Inter \lor FullTask\}$$

$$direct : (Process \times Element) \rightarrowtail Process$$

$$\begin{aligned}
&\forall\, p : Process;\ e : Element \bullet \\
&\quad e \in p \Rightarrow direct(p, e) = p \land \\
&\quad e \in_{\mathrm{p}} p \land e \notin p \land (\exists_1 f : Element \bullet f \in_{\mathrm{p}} p \land (e, f) \in contains) \Rightarrow \\
&\qquad direct(p, e) = (\mu f : Element \mid f \in_{\mathrm{p}} p \land (e, f) \in contains \bullet content(f))
\end{aligned}$$

$$getIns, getOuts, getSeqflows : \mathbb{P}\ Element \to \mathbb{P}\ Seqflow$$

$$\begin{aligned}
&\forall\, es : \mathbb{P}\ Element \bullet \\
&\quad getIns\ es = \bigcup\{e : Element \mid e \in_{\mathrm{p}} es \bullet (atom\ e).in\} \land \\
&\quad getOuts\ es = \bigcup\{e : Element \mid e \in_{\mathrm{p}} es \bullet outs(e)\} \land \\
&\quad getSeqflows = getIns \cup getOuts
\end{aligned}$$

$$getSds, getRecs, getMsgs : \mathbb{P}\ Element \to \mathbb{P}\ Mgeflow$$

$$\begin{aligned}
&\forall\, es : \mathbb{P}\ Element \bullet \\
&\quad getSds\ es = \bigcup\{e : Element \mid e \in_{\mathrm{p}} es \bullet getSd(atom\ e)\} \land \\
&\quad getRecs\ es = \bigcup\{e : Element \mid e \in_{\mathrm{p}} es \bullet getRec(atom\ e)\} \land \\
&\quad getMsgs = getRecs \cup getSds
\end{aligned}$$

A.5 Initialisation Theorems

A.5.1 Start and End Elements

Theorem (4.4 Initialisation Theorem for $Atom$)**.** $\exists\, Atom' \bullet StartAtomInit$

Proof.

$$
\begin{array}{ll}
& \exists\, Atom' \bullet StartAtomInit \\
\Leftrightarrow & \exists\, Atom' \bullet \theta Atom' = startatom \qquad \text{[def of } StartAtomInit \text{ and schema quantification]} \\
\Leftrightarrow & \exists\, type' : Type;\ in', out' : \mathbb{F}\, Seqflow;\ exit' : Seqflow \rightarrowtail\!\!\!\!\!\!\twoheadrightarrow Type; \quad \text{[def of } Atom' \text{ and schema binding]} \\
& \quad range' : Range;\ send', receive' : \mathbb{F}\, Mgeflow \bullet \\
& \quad \mathsf{disjoint}\ \langle in', out', \mathrm{dom}\, exit' \rangle \wedge \\
& \quad send' \cap receive' = \emptyset \wedge \\
& \quad type' = start \wedge \\
& \quad in' = \emptyset \wedge \\
& \quad out' = \{seq1\} \wedge \\
& \quad exit' = \emptyset \wedge \\
& \quad range' = (0,0) \wedge \\
& \quad send' = \emptyset \wedge \\
& \quad receive' = \emptyset \\
\Leftrightarrow & \quad \mathsf{disjoint}\ \langle \emptyset, \{seq1\}, \mathrm{dom}\, \emptyset \rangle \wedge \qquad \text{[one-point x 7]} \\
& \quad \emptyset \cap \emptyset = \emptyset \wedge \\
& \quad start \in Type \wedge \\
& \quad \emptyset \in \mathbb{F}\, Seqflow \wedge \\
& \quad \{seq1\} \in \mathbb{F}\, Seqflow \wedge \\
& \quad \emptyset \in Seqflow \rightarrowtail\!\!\!\!\!\!\twoheadrightarrow Type \wedge \\
& \quad (0,0) \in Range \wedge \\
& \quad \emptyset \in \mathbb{F}\, Mgeflow \wedge \\
& \quad \emptyset \in \mathbb{F}\, Mgeflow \\
\Leftrightarrow & (0,0) \in Range \qquad \text{[property of } \mathsf{disjoint} \text{ and } \cap\text{, def of } \mathbb{F}\text{, } \rightarrowtail\!\!\!\!\!\!\twoheadrightarrow \text{ and } Type] \\
\Leftrightarrow & true \qquad [0 \in \mathbb{N} \text{ and } Range == \mathbb{N} \times \mathbb{N}]
\end{array}
$$

□

Theorem (4.5 Initialisation Theorem for $Atom$). $\exists\, Atom' \bullet EndAtomInit$

Proof. Similar to the proof of Theorem 4.4. □

The following is the initialisation theorem for *FlowObject*.

Theorem (4.6 Initialisation Theorem for $FlowObject$). $\exists\, FlowObject' \bullet StartInit$

Proof.

$$
\begin{array}{ll}
& \exists\, FlowObject' \bullet StartInit \\
\Leftrightarrow & \exists\, FlowObject' \bullet ele' = startele \qquad \text{[def of } StartInit \text{ and schema quantification]} \\
\Leftrightarrow & \exists\, ele' : Element \bullet (Activity' \lor Gate' \lor Event') \wedge ele' = startele \qquad \text{[def of } FlowObject'] \\
\Leftrightarrow & ((\exists\, ele' : Element \bullet Activity' \wedge ele' = startele) \lor \qquad \text{[def of } Event\text{, } \wedge\text{-}\lor\text{-dist and } \exists\text{-}\lor\text{-dist]} \\
& \quad (\exists\, ele' : Element \bullet Gate' \wedge ele' = startele) \lor \\
& \quad ((\exists\, ele' : Element \bullet Start' \wedge ele' = startele) \lor \\
& \quad (\exists\, ele' : Element \bullet Inter' \wedge ele' = startele) \lor \\
& \quad (\exists\, ele' : Element \bullet End' \wedge ele' = startele)))
\end{array}
$$

It is sufficient to show one of the disjuncts is true. We consider the third disjunct.

$$\exists\, ele' : Element \bullet Start' \wedge ele' = startele$$

$\Leftrightarrow \exists\, ele' \bullet$ [def of $Start'$ and $NonActivity'$]
$ele' \in \operatorname{ran} atomic \land$
$\#(atom\ ele').exit + \#(atom\ ele').send + \#(atom\ ele').receive = 0 \land$
$first(atom\ ele').range = 0 \land$
$second(atom\ ele').range = 0 \land$
$(atom\ ele').type \in \{start\} \cup \operatorname{ran} stime \cup \operatorname{ran} smessage \cup \operatorname{ran} srule \land$
$\#(atom\ ele').in = 0 \land$
$\#(atom\ ele').out = 1 \land$
$ele' = startele$

$\Leftrightarrow$ $startele \in \operatorname{ran} atomic \land$ [one-point]
$\#(atom\ startele).exit + \#(atom\ startele).send + \#(atom\ startele).receive = 0 \land$
$first(atom\ startele).range = 0 \land$
$second(atom\ startele).range = 0 \land$
$(atom\ startele).type \in \{start\} \cup \operatorname{ran} stime \cup \operatorname{ran} smessage \cup \operatorname{ran} srule \land$
$\#(atom\ startele).in = 0 \land$
$\#(atom\ startele).out = 1 \land$
$startele \in Element$

$\Leftrightarrow$ $startele \in \operatorname{ran} atomic \land$ [def of $atom$]
$\#startatom.exit + \#startatom.send + \#startatom.receive = 0 \land$
$first\ startatom.range = 0 \land$
$second\ startatom.range = 0 \land$
$startatom.type \in \{start\} \cup \operatorname{ran} stime \cup \operatorname{ran} smessage \cup \operatorname{ran} srule \land$
$\#startatom.in = 0 \land$
$\#startatom.out = 1 \land$
$startele \in Element$

$\Leftrightarrow$ $true$ [def of $startele$ and $startatom$, and Theorem 4.4.]

□

Theorem (4.7 Initialisation Theorem for $FlowObject$)**.** $\exists\, FlowObject' \bullet EndInit$

Proof. Similar to the proof of Theorem 4.6 by applying Theorem 4.5 and proving the expression $\exists\, ele' : Element \bullet End' \land ele' = endele$. □

A.5.2 Process

Here we prove some properties about the values *startele*, *endele* and *initialproc* that will be useful when proving the initialisation theorem for schema *GenProc*.

Lemma A.1. $(edge\ \{startele, endele\})^+ = \{(startele, endele)\}$

Proof. We first enumerate and label the following facts about *startele* and *endele*.

$$outs(startele) \neq \emptyset \land (atom\ endele).in \neq \emptyset \land (atom\ startele).in = \emptyset \land outs(endele) = \emptyset \tag{A.1}$$

$(edge\ \{startele, endele\})^+$
$= (\{e, f : \{startele, endele\} \mid ((atom\ e).out \cup \operatorname{dom}(atom\ e).exit) \cap (atom\ f).in \neq \emptyset\})^+$ [def of $edge$]
$= \{(startele, endele)\}^+$ [Expression A.1]
$= \{(startele, endele)\}$ [$endele \notin \operatorname{dom}\{(startele, endele)\}$]

□

Lemma A.2. $\{g : Element \mid g \in_p initialproc\} = initialproc$

Proof.

$$
\begin{aligned}
&\{g : Element \mid g \in_{\mathrm{p}} initialproc\} \\
&= \{g : Element \mid g \in initialproc \lor (\exists_1 f : Element \bullet f \in initialproc \land g \in_{\mathrm{e}} f)\} && [\text{def of } \in_{\mathrm{p}}] \\
&= \{g : Element \mid && [\text{def of } \in_{\mathrm{e}}] \\
&\quad g \in initialproc \lor (\exists_1 f : Element \bullet f \in initialproc \land (g,f) \in contains^+)\} \\
&= \{g : Element \mid g \in initialproc\} \cup && [\text{set-theory}] \\
&\quad \{g : Element \mid (\exists_1 f : Element \bullet f \in initialproc \land (g,f) \in contains^+)\} \\
&= initialproc \cup \{g : Element \mid (\exists_1 f : Element \bullet f \in initialproc \land (g,f) \in contains^+)\} && [\text{def of } \in] \\
&= initialproc \cup && [\text{def of } ^+] \\
&\quad \{g : Element \mid (\exists_1 f : Element \bullet f \in initialproc \land \\
&\qquad (g,f) \in \bigcap\{Q : Element \leftrightarrow Element \mid contains \subseteq Q \land Q \fatsemi Q \subseteq Q\} \\
&= initialproc \cup && [\text{def of } contains] \\
&\quad \{g : Element \mid (\exists_1 f : Element \bullet f \in initialproc \land \\
&\qquad (g,f) \in \bigcap\{Q : Element \leftrightarrow Element \mid \\
&\qquad\quad \{j,k : Element \bullet k \in \operatorname{ran} compound \land j \in content(k)\} \subseteq Q \land Q \fatsemi Q \subseteq Q\} \\
&= \{startele, endele\} \cup && [\text{def of } initialproc] \\
&\quad \{g : Element \mid (\exists_1 f : Element \bullet f \in \{startele, endele\} \land \\
&\qquad (g,f) \in \bigcap\{Q : Element \leftrightarrow Element \mid \\
&\qquad\quad \{j,k : Element \bullet k \in \operatorname{ran} compound \land j \in content(k)\} \subseteq Q \land Q \fatsemi Q \subseteq Q\} \\
&= \{startele, endele\} \cup \emptyset && [startele \notin \operatorname{ran} compound \land endele \notin \operatorname{ran} compound] \\
&= initialproc && [\cup\text{-unit and def of } initialproc]
\end{aligned}
$$

□

We now prove the initialisation theorem for schema *GenProc*.

Theorem (4.8 Initialisation Theorem for *GenProc*)**.** $\exists\, GenProc' \bullet GenProcInit$

Proof.

$$
\begin{aligned}
&\exists\, GenProc' \bullet GenProcInit \\
&\Leftrightarrow \exists\, GenProc \bullet proc' = \{startele, endele\} && [\text{def of } GenProcInit, initialproc, \text{schema quantification}] \\
&\Leftrightarrow \exists\, proc' : Process \bullet && [\text{def of } GenProc', WFProcess] \\
&\quad proc' \in processSet \cap noOverLap \land \\
&\quad (\exists\, e,f : proc' \bullet e \in \{Start \bullet ele\} \land f \in \{End \bullet ele\}) \land \\
&\quad \neg\, (\exists\, e : proc' \bullet e \in \{Inter \mid (atom\ ele).type \in \operatorname{ran} ierror \bullet ele\}) \land \\
&\quad proc' \in onlyFlowObject \land \\
&\quad proc' = \{startele, endele\} \\
&\Leftrightarrow \{startele, endele\} \in processSet \cap noOverLap \land && [\text{one-point}] \\
&\quad (\exists\, e,f : \{startele, endele\} \bullet e \in \{Start \bullet ele\} \land f \in \{End \bullet ele\}) \land \\
&\quad \neg\, (\exists\, e : \{startele, endele\} \bullet e \in \{Inter \mid (atom\ ele).type \in \operatorname{ran} ierror \bullet ele\}) \land \\
&\quad \{startele, endele\} \in onlyFlowObject \land \\
&\quad \{startele, endele\} \in Process \\
&\Leftrightarrow \{startele, endele\} \in noOverLap \land && [\text{def of } processSet \text{ and intersection}] \\
&\quad \{startele, endele\} \in endsConnected \land \\
&\quad \{startele, endele\} \in noUnConnected \land
\end{aligned}
$$

$$
\begin{array}{l}
(\exists\, e, f : \{startele, endele\} \bullet e \in \{Start \bullet ele\} \land f \in \{End \bullet ele\}) \land \\
\neg\, (\exists\, e : \{startele, endele\} \bullet e \in \{Inter \mid (atom\ ele).type \in \operatorname{ran} ierror \bullet ele\}) \land \\
\{startele, endele\} \in onlyFlowObject \land \\
\{startele, endele\} \in Process
\end{array}
$$

We consider each of the seven conjuncts individually. We now consider the first conjunct.

$$
\begin{array}{ll}
& \{startele, endele\} \in noOverLap \\
\Leftrightarrow & \forall\, e : \{g : Element \mid g \in_{\mathrm{p}} \{startele, endele\}\} \bullet \qquad \text{[def of } noOverLap\text{]} \\
& \quad \bigcup\{k : \{startele, endele\} \bullet (atom\ k).in \cup outs(k)\} \cap \\
& \quad \bigcup\{k : content\ e \bullet (atom\ k).in \cup outs(k)\} = \emptyset \land \\
& \quad (\forall\, f : \{g : Element \mid g \in_{\mathrm{p}} \{startele, endele\}\} \bullet e \neq f \Rightarrow \\
& \qquad ((atom\ e).in \cap (atom\ f).in = \emptyset \land \\
& \qquad outs(e) \cap outs(f) = \emptyset \land \\
& \qquad \bigcup\{k : content\ e \bullet (atom\ k).in \cup outs(k)\} \cap \bigcup\{k : content\ f \bullet (atom\ k).in \cup outs(k)\} = \emptyset \land \\
& \qquad getMsg\,(atom\ e) \cap getMsg\,(atom\ f) = \emptyset)) \\
\Leftrightarrow & \forall\, e, f : \{startele, endele\} \bullet \qquad \text{[Lemma A.2 and property of } \forall\text{]} \\
& \quad \bigcup\{k : \{startele, endele\} \bullet (atom\ k).in \cup outs(k)\} \cap \\
& \quad \bigcup\{k : content\ e \bullet (atom\ k).in \cup outs(k)\} = \emptyset \land \\
& \quad (e \neq f \Rightarrow \\
& \qquad ((atom\ e).in \cap (atom\ f).in = \emptyset \land \\
& \qquad outs(e) \cap outs(f) = \emptyset \land \\
& \qquad \bigcup\{k : content\ e \bullet (atom\ k).in \cup outs(k)\} \cap \bigcup\{k : content\ f \bullet (atom\ k).in \cup outs(k)\} = \emptyset \land \\
& \qquad getMsg\,(atom\ e) \cap getMsg\,(atom\ f) = \emptyset)) \\
\Leftrightarrow & (startele \neq endele \Rightarrow \qquad \text{[}\forall\text{-elim, } content\ startele \cup content\ endele = \emptyset \text{ and commutativity]} \\
& \quad ((atom\ startele).in \cap (atom\ endele).in = \emptyset \land \\
& \quad (((atom\ startele).out \cup \operatorname{dom}(atom\ startele).exit) \cap \\
& \quad ((atom\ endele).out \cup \operatorname{dom}(atom\ endele).exit) = \emptyset) \land \\
& \quad getMsg\,(atom\ startele) \cap getMsg\,(atom\ endele) = \emptyset)) \\
\Leftrightarrow & ((startatom.in \cap endatom.in = \emptyset \land \qquad \text{[def of } startele \text{ and } endele\text{]} \\
& \quad ((startatom.out \cup \operatorname{dom} startatom.exit) \cap (endatom.out \cup \operatorname{dom} endatom.exit) = \emptyset) \land \\
& \quad getMsg\ startatom \cap getMsg\ endatom = \emptyset)) \\
\Leftrightarrow & ((\emptyset \cap \{seq1\} = \emptyset \land \qquad \text{[def of } startatom \text{ and } endatom\text{]} \\
& \quad ((\{seq1\} \cup \operatorname{dom} \emptyset) \cap (\emptyset \cup \operatorname{dom} \emptyset) = \emptyset) \land \\
& \quad getMsg\ startatom \cap getMsg\ endatom = \emptyset)) \\
\Leftrightarrow & getMsg\ startatom \cap getMsg\ endatom = \emptyset \qquad \text{[property of } \cap, \cup, \_(\!|\ \_\ |\!)\text{]} \\
\Leftrightarrow & (getSd\ startatom \cup getRec\ startatom) \cap \qquad \text{[property of } \land \text{ and def of } getMsg\text{]} \\
& \quad (getSd\ endatom \cup getRec\ endatom) = \emptyset \\
\Leftrightarrow & \qquad \text{[def of } getSd \text{ and } getRec\text{, and } startatom.type = start \text{ and } endatom.type = end\text{]} \\
& \quad (startatom.send \cup startatom.receive) \cap (endatom.send \cup endatom.receive) = \emptyset \\
\Leftrightarrow & true \qquad \text{[def of } startatom \text{ and } endatom\text{]}
\end{array}
$$

We now consider the second conjunct.

$$
\begin{array}{ll}
& \{startele, endele\} \in endsConnected \\
\Leftrightarrow & (\forall\, e : \{startele, endele\} \bullet \qquad \text{[def of } endsConnected\text{]} \\
& \quad (e \notin \{\, End \bullet ele\,\} \Rightarrow
\end{array}
$$

$$
\begin{array}{ll}
 & \quad (\exists f : \{startele, endele\} \bullet f \in \{\, End \bullet ele \,\} \land (e, f) \in (edge\,\{startele, endele\})^+)) \land \\
 & (e \notin \{\, Start \bullet ele \,\} \Rightarrow \\
 & \quad (\exists f : \{startele, endele\} \bullet f \in \{\, Start \bullet ele \,\} \land (f, e) \in (edge\,\{startele, endele\})^+))) \\
\Leftrightarrow & (\exists f : \{startele, endele\} \bullet \qquad [\forall\text{-elim, Theorems 4.6 and 4.7, def of } startele,\ endele \text{ and } \Rightarrow] \\
 & \quad f \in \{\, End \bullet ele \,\} \land (startele, f) \in (edge\,\{startele, endele\})^+) \land \\
 & (\exists f : \{startele, endele\} \bullet \\
 & \quad f \in \{\, Start \bullet ele \,\} \land (f, endele) \in (edge\,\{startele, endele\})^+) \\
\Leftrightarrow & ((startele \in \{\, End \bullet ele \,\} \land (startele, startele) \in (edge\,\{startele, endele\})^+) \lor \qquad [\exists\text{-elim}] \\
 & (endele \in \{\, End \bullet ele \,\} \land (startele, endele) \in (edge\,\{startele, endele\})^+)) \land \\
 & ((startele \in \{\, Start \bullet ele \,\} \land (startele, endele) \in (edge\,\{startele, endele\})^+) \lor \\
 & (endele \in \{\, Start \bullet ele \,\} \land (endele, endele) \in (edge\,\{startele, endele\})^+)) \\
\Leftrightarrow & true \qquad [\text{Lemma A.1, } startele \in \{Start \bullet ele\} \text{ and } endele \in \{End \bullet ele\}]
\end{array}
$$

We now consider the third conjunct.

$$
\begin{array}{ll}
 & \{startele, endele\} \in noUnConnected \\
\Leftrightarrow & (\forall\, e : \{startele, endele\} \bullet \\
 & \quad (\forall\, s : outs(e) \bullet \exists f : \{startele, endele\} \bullet s \in (atom\, f).in) \land \\
 & \quad (\forall\, s : (atom\, e).in \bullet \exists f : \{startele, endele\} \bullet s \in outs(f))) \\
\Leftrightarrow & ((\forall\, s : outs(startele) \bullet \exists f : \{startele, endele\} \bullet s \in (atom\, f).in) \land \qquad [\forall\text{-elim}] \\
 & \quad (\forall\, s : (atom\, startele).in \bullet \exists f : \{startele, endele\} \bullet s \in outs(f))) \land \\
 & ((\forall\, s : outs(endele) \bullet \exists f : \{startele, endele\} \bullet s \in (atom\, f).in) \land \\
 & \quad (\forall\, s : (atom\, endele).in \bullet \exists f : \{startele, endele\} \bullet s \in outs(f))) \\
\Leftrightarrow & \qquad [outs(endele) = \emptyset \text{ and } (atom\, startele).in = \emptyset] \\
 & (\forall\, s : outs(startele) \bullet \exists f : \{startele, endele\} \bullet s \in (atom\, f).in) \land \\
 & (\forall\, s : (atom\, endele).in \bullet \exists f : \{startele, endele\} \bullet s \in outs(f)) \\
\Leftrightarrow & \qquad [\text{def of } startele \text{ and } endeled, \text{ and } \forall\text{-elim}] \\
 & \exists f : \{startele, endele\} \bullet seq1 \in (atom\, f).in \land \\
 & \exists f : \{startele, endele\} \bullet seq1 \in outs(f) \\
\Leftrightarrow & \qquad [\text{def of } startele \text{ and } endeled, \text{ and } \exists\text{-elim}] \\
 & seq1 \in (atom\, endele).in \land \\
 & seq1 \in outs(startele) \\
\Leftrightarrow & true
\end{array}
$$

We now consider the fourth conjunct.

$$
\begin{array}{ll}
 & \exists\, e, f : \{startele, endele\} \bullet e \in \{Start \bullet ele\} \land f \in \{End \bullet ele\} \\
\Leftrightarrow & (\exists\, e : \{startele, endele\} \bullet e \in \{Start \bullet ele\}) \land \qquad [f \text{ is not free in } e \in \{Start \bullet ele\}] \\
 & (\exists f : \{startele, endele\} \bullet f \in \{End \bullet ele\}) \\
\Leftrightarrow & (startele \in \{Start \bullet ele\} \lor endele \in \{Start \bullet ele\}) \land \qquad [\text{def of } \exists] \\
 & (startele \in \{End \bullet ele\} \lor endele \in \{End \bullet ele\}) \\
\Leftrightarrow & true \qquad [\text{Theorems 4.6 and 4.7}]
\end{array}
$$

We consider the fifth conjunct.

$$
\begin{array}{ll}
 & \neg\, (\exists\, e : \{startele, endele\} \bullet e \in \{Inter \mid (atom\, ele).type \in \operatorname{ran} ierror \bullet ele\,\}) \\
\Leftrightarrow & \forall\, e : \{startele, endele\} \bullet e \notin \{Inter \mid (atom\, ele).type \in \operatorname{ran} ierror \bullet ele\,\} \quad [\text{property of } \exists \text{ and } \neg\,] \\
\Leftrightarrow & startele \notin \{Inter \mid (atom\, ele).type \in \operatorname{ran} ierror \bullet ele\,\} \land \qquad [\text{def of } \forall]
\end{array}
$$

$$
\begin{array}{ll}
& endele \notin \{Inter \mid (atom\ ele).type \in \operatorname{ran} ierror \bullet ele\} \\
\Leftrightarrow & true \qquad [(atomic\ startele).type = start \text{ and } (atomic\ endele).type = end]
\end{array}
$$

We now consider the sixth conjunct.

$$
\begin{array}{ll}
& \{startele, endele\} \in onlyFlowObject \\
\Leftrightarrow & \forall e : \{g : Element \mid g \in_{\mathrm{p}} \{startele, endele\}\} \bullet e \in \{FlowObject \bullet ele\} \qquad [\text{def of } onlyFlowObject] \\
\Leftrightarrow & startele \in \{FlowObject \bullet ele\} \land startele \in \{FlowObject \bullet ele\} \qquad [\forall\text{-elim}] \\
\Leftrightarrow & true \qquad [\text{Theorem 4.6 and 4.7}]
\end{array}
$$

We finally consider the seventh conjunct.

$$
\begin{array}{ll}
& \{startele, endele\} \in Process \\
\Leftrightarrow & \{startele, endele\} \in \mathbb{F}_1\ Element \qquad [\text{def of } Process] \\
\Leftrightarrow & true \qquad [startele \in Element \text{ and } endele \in Element]
\end{array}
$$

The completes the proof. □

A.5.3 Pool

Theorem (4.9 Initialisation Theorem for *Pool*)**.** $\exists\, Pool' \bullet PoolInit$

Proof.

$$
\begin{array}{ll}
& \exists\, Pool' \bullet PoolInit \\
\Leftrightarrow & \exists\, Pool' \bullet proc' = initialproc \qquad [\text{def of } PoolInit \text{ and schema quantification}] \\
\Leftrightarrow & \exists\, proc' : Process \bullet \qquad [\text{def of } Pool'] \\
& \quad proc' \in processSet \cap noOverLap \\
& \quad (\exists\, e, f : proc' \bullet e \in \{Start \bullet ele\} \land f \in \{End \bullet ele\}) \land \\
& \quad \neg\, (\exists\, e : proc' \bullet e \in \{Inter \mid (atom\ ele).type \in \operatorname{ran} ierror \bullet ele\}) \land \\
& \quad proc' \in onlyFlowObject \land \\
& \quad proc' \in \{proc : Process \mid hasExgates\} \land \\
& \quad proc' = initialproc \\
\Leftrightarrow & \quad \{startele, endele\} \in processSet \cap noOverLap \land \qquad [\text{one-point and def of } initialproc] \\
& \quad (\exists\, e, f : \{startele, endele\} \bullet e \in \{Start \bullet ele\} \land f \in \{End \bullet ele\}) \land \\
& \quad \neg\, (\exists\, e : \{startele, endele\} \bullet e \in \{Inter \mid (atom\ ele).type \in \operatorname{ran} ierror \bullet ele\}) \land \\
& \quad \{startele, endele\} \in onlyFlowObject \land \\
& \quad \{startele, endele\} \in \{proc : Process \mid hasExgates\} \land \\
& \quad \{startele, endele\} \in Process \\
\Leftrightarrow & \{startele, endele\} \in \{proc : Process \mid hasExgates\} \land \qquad [\text{Theorem 4.8}] \\
& \{startele, endele\} \in Process
\end{array}
$$

We consider these two conjuncts individually. We consider the first conjunct.

$$
\begin{array}{ll}
& \{startele, endele\} \in \{proc : Process \mid hasExgates\} \\
\Leftrightarrow & \forall f : \{g : Element \mid g \in_{\mathrm{p}} \{startele, endele\}\} \bullet \qquad [\text{def of } hasExgates] \\
& \quad f \in eventgate \Rightarrow \\
& \qquad (\forall\, e : Element \bullet (f, e) \in edge(direct(\{startele, endele\}, e)) \Rightarrow e \in sendelement) \\
\Leftrightarrow & \quad (startele \in eventgate \Rightarrow \qquad [\text{Lemma A.2, } \forall\text{-elim and def of } direct] \\
& \qquad (\forall\, e : Element \bullet (startele, e) \in edge(direct(\{startele, endele\}, e)) \Rightarrow e \in sendelement)) \land \\
& \quad (endele \in eventgate \Rightarrow
\end{array}
$$

$$
\begin{array}{ll}
 & (\forall\, e : Element \bullet (endele, e) \in edge(direct(\{startele, endele\}, e)) \Rightarrow e \in sendelement)) \\
\Leftrightarrow & (startele \in \{Gate \mid (atom\ ele).type = exgate \bullet ele\} \Rightarrow & \text{[def of } eventgate\text{]} \\
 & \quad (\forall\, e : Element \bullet (startele, e) \in edge(direct(\{startele, endele\}, e)) \Rightarrow e \in sendelement)) \wedge \\
 & (endele \in \{Gate \mid (atom\ ele).type = exgate \bullet ele\} \Rightarrow \\
 & \quad (\forall\, e : Element \bullet (endele, e) \in edge(direct(\{startele, endele\}, e)) \Rightarrow e \in sendelement)) \\
\Leftrightarrow & true & [(atom\ startele).type = start \text{ and } (atom\ endele).type = end, \text{ def of } \Rightarrow \text{ and } \wedge]
\end{array}
$$

We now consider the second conjunct.

$$
\begin{array}{ll}
 & \{startele, endele\} \in Process \\
\Leftrightarrow & \{startele, endele\} \in \mathbb{F}_1\, Element & \text{[def of } initialproc \text{ and } Process\text{]} \\
\Leftrightarrow & true & [startele \in Element \text{ and } endele \in Element]
\end{array}
$$

The completes the proof. □

A.5.4 Diagram

Here we first prove some properties about *initialproc* that will be useful for proving the initialisation theorem for schema *Diagram*.

Theorem (4.10 Initialisation Theorem for *Diagram*)**.** $\exists\, Diagram' \bullet DiagramInit$

Proof.

$$
\begin{array}{ll}
 & \exists\, Diagram' \bullet DiagramInit \\
\Leftrightarrow & \exists\, Diagram' \bullet [Diagram' \mid pool' = initialpool] & \text{[def of } DiagramInit\text{]} \\
\Leftrightarrow & \exists\, pool' : PoolId \rightarrowtail Pool \bullet & \text{[schema quantification and def of } Diagram'\text{]} \\
 & \quad pool' \neq \emptyset \wedge \\
 & \quad (\forall\, p, q : \operatorname{ran} pool' \bullet p \neq q \Rightarrow getSeqflows\ p.proc \cap getSeqflows\ q.proc = \emptyset) \wedge \\
 & \quad (\forall\, p, q : \operatorname{ran} pool' \bullet \\
 & \qquad (p \neq q \Rightarrow getSds\ p.proc \cap getSds\ q.proc = \emptyset \wedge getRecs\ p.proc \cap getRecs\ q.proc = \emptyset)) \\
 & \quad (\forall\, p : \operatorname{ran} pool' \bullet \\
 & \qquad ((\forall\, m : getSds\ p.proc \bullet (\exists\, q : \operatorname{ran} pool' \bullet p \neq q \wedge m \in getRecs\ p.proc)) \wedge \\
 & \qquad (\forall\, m : getRecs\ p.proc \bullet (\exists\, q : \operatorname{ran} pool' \bullet (p \neq q \wedge m \in getSds\ p.proc))))) \wedge \\
 & \quad pool' = initialpool \\
\Leftrightarrow & & \text{[one-point and } initialpool \neq \emptyset\text{]} \\
 & \quad (\forall\, p, q : \operatorname{ran} initialpool \bullet p \neq q \Rightarrow getSeqflows\ p.proc \cap getSeqflows\ q.proc = \emptyset) \wedge \\
 & \quad (\forall\, p, q : \operatorname{ran} initialpool \bullet \\
 & \qquad (p \neq q \Rightarrow getSds\ p.proc \cap getSds\ q.proc = \emptyset \wedge getRecs\ p.proc \cap getRecs\ q.proc = \emptyset)) \\
 & \quad (\forall\, p : \operatorname{ran} initialpool \bullet \\
 & \qquad ((\forall\, m : getSds\ p.proc \bullet (\exists\, q : \operatorname{ran} initialpool \bullet p \neq q \wedge m \in getRecs\ p.proc)) \wedge \\
 & \qquad (\forall\, m : getRecs\ p.proc \bullet (\exists\, q : \operatorname{ran} initialpool \bullet (p \neq q \wedge m \in getSds\ p.proc))))) \wedge \\
 & \quad initialpool \in PoolId \rightarrowtail Pool
\end{array}
$$

We consider each conjunct individually. We first consider the first conjunct.

$$
\begin{array}{ll}
 & \forall\, p, q : \operatorname{ran} initialpool \bullet p \neq q \Rightarrow getSeqflows\ p.proc \cap getSeqflows\ q.proc = \emptyset \\
\Leftrightarrow & \forall\, p, q : \operatorname{ran}\{pool1 \mapsto \langle\!| proc \rightsquigarrow initialproc |\!\rangle\} \bullet & \text{[def of } initialpool\text{]} \\
 & \quad p \neq q \Rightarrow getSeqflows\ p.proc \cap getSeqflows\ q.proc = \emptyset \\
\Leftrightarrow & \forall\, p, q : \{\langle\!| proc \rightsquigarrow initialproc |\!\rangle\} \bullet & \text{[def of ran]}
\end{array}
$$

$$
\begin{array}{ll}
& p \neq q \Rightarrow getSeqflows\ p.proc \cap getSeqflows\ q.proc = \emptyset \\
\Leftrightarrow & \langle proc \rightsquigarrow initialproc \rangle \neq \langle proc \rightsquigarrow initialproc \rangle \Rightarrow & [\forall\text{-elim}] \\
& getSeqflows\ initialproc \cap getSeqflows\ initialproc = \emptyset \\
\Leftrightarrow & false \Rightarrow getSeqflows\ initialproc \cap getSeqflows\ initialproc = \emptyset & [p \neq p] \\
\Leftrightarrow & true & [\text{def of } \Rightarrow]
\end{array}
$$

We now consider the second conjunct.

$$
\begin{array}{ll}
& (\forall\, p, q : \operatorname{ran} initialpool \bullet \\
& \quad (p \neq q \Rightarrow getSds\ p.proc \cap getSds\ q.proc = \emptyset \wedge getRecs\ p.proc \cap getRecs\ q.proc = \emptyset)) \\
\Leftrightarrow & (\forall\, p, q : \{\langle proc \rightsquigarrow initialproc \rangle\} \bullet & [\text{def of } initialpool \text{ and ran}] \\
& \quad (p \neq q \Rightarrow getSds\ p.proc \cap getSds\ q.proc = \emptyset \wedge getRecs\ p.proc \cap getRecs\ q.proc = \emptyset)) \\
\Leftrightarrow & (\langle proc \rightsquigarrow initialproc \rangle \neq \langle proc \rightsquigarrow initialproc \rangle \Rightarrow & [\forall\text{-elim}] \\
& \quad getSds\ proc \cap getSds\ proc = \emptyset \wedge getRecs\ proc \cap getRecs\ proc = \emptyset) \\
\Leftrightarrow & (false \Rightarrow & [\langle proc \rightsquigarrow initialproc \rangle = \langle proc \rightsquigarrow initialproc \rangle] \\
& \quad getSds\ proc \cap getSds\ proc = \emptyset \wedge getRecs\ proc \cap getRecs\ proc = \emptyset) \\
\Leftrightarrow & true & [\text{def of } \Rightarrow]
\end{array}
$$

We now consider the third conjunct.

$$
\begin{array}{ll}
& \forall\, p : \operatorname{ran} initialpool \bullet \\
& \quad ((\forall\, m : getSds\ p.proc \bullet (\exists\, q : \operatorname{ran} initialpool \bullet (p \neq q \wedge m \in getRecs\ p.proc))) \wedge \\
& \quad (\forall\, m : getRecs\ p.proc \bullet (\exists\, q : \operatorname{ran} initialpool \bullet (p \neq q \wedge m \in getSds\ p.proc)))) \\
\Leftrightarrow & \forall\, p : \{\langle proc \rightsquigarrow initialproc \rangle\} \bullet & [\text{def of } initialpool \text{ and ran}] \\
& \quad ((\forall\, m : getSds\ p.proc \bullet (\exists\, q : \{\langle proc \rightsquigarrow initialproc \rangle\} \bullet (p \neq q \wedge m \in getRecs\ p.proc))) \wedge \\
& \quad (\forall\, m : getRecs\ p.proc \bullet (\exists\, q : \{\langle proc \rightsquigarrow initialproc \rangle\} \bullet (p \neq q \wedge m \in getSds\ p.proc)))) \\
\Leftrightarrow & \quad (\forall\, m : getSds\ initialproc \bullet & [\forall\text{-elim}] \\
& \qquad (\exists\, q : \{\langle proc \rightsquigarrow initialproc \rangle\} \bullet (\langle proc \rightsquigarrow initialproc \rangle \neq q \wedge m \in getRecs\ initialproc))) \wedge \\
& \quad (\forall\, m : getRecs\ initialproc \bullet \\
& \qquad (\exists\, q : \{\langle proc \rightsquigarrow initialproc \rangle\} \bullet (\langle proc \rightsquigarrow initialproc \rangle \neq q \wedge m \in getSds\ initialproc))) \\
\Leftrightarrow & true & [\text{def of } initialproc,\ \emptyset \cup \emptyset = \emptyset \text{ and } \forall\, x : \emptyset \bullet p \text{ is true}]
\end{array}
$$

We now consider the fourth conjunct.

$$
\begin{array}{ll}
& initialpool \in PoolId \rightarrowtail Pool \\
\Leftrightarrow & \{pool1 \mapsto \langle proc \rightsquigarrow initialproc \rangle\} \in PoolId \rightarrowtail Pool & [\text{def of } initialpool] \\
\Leftrightarrow & true & [pool1 \in PoolId \text{ and } \langle proc \rightsquigarrow initialproc \rangle \in Pool]
\end{array}
$$

This completes the proof. □

A.6 Diagram Construction

A.6.1 Preliminaries

A.6.1.1 States and Functions

$$InitialInter \mathrel{\widehat=} [Inter \mid (atom\ ele).type \in \{imessage(nomessage)\} \cup \operatorname{ran} itime]$$

$$InitialEnd \mathrel{\widehat=} [End \mid (atom\ ele).type = end]$$

$$NonEvSplit \mathrel{\widehat=} [Gate \mid (atom\ ele).type \neq exgate \land \#(atom\ ele).in = 1]$$

$$EventSplit \mathrel{\widehat=} [Gate \mid (atom\ ele).type = exgate \land \#(atom\ ele).in = 1]$$

$$NonEvJoin \mathrel{\widehat=} [Gate \mid (atom\ ele).type \neq exgate \land \#(atom\ ele).out = 1]$$

$$NoInternal \mathrel{\widehat=} [ele : Element \mid getMsgs\{ele\} = \emptyset \land (atom\ ele).exit = \emptyset]$$

$$OneInOutAtom \mathrel{\widehat=} (FullTask \land NoInternal) \lor InitialInter$$

$$OneInOutObject \mathrel{\widehat=} (Activity \land NoInternal) \lor InitialInter$$

$$alls, ends, activities, subs, tasks, eerrors, nonsends : Process \pfun (Seqflow \pfun Element)$$

$$\begin{aligned}
&\forall p : Process \bullet \\
&\quad alls\ p = \{s : Seqflow;\ g : Element \mid g \in_{\mathrm{p}} p \land s \in (atom\ g).in\} \land \\
&\quad ends\ p = (alls\ p) \rhd \{InitialEnd \bullet ele\} \land \\
&\quad activities\ p = (alls\ p) \rhd \{Activity \bullet ele\} \land \\
&\quad subs\ p = (alls\ p) \rhd \{FullSub \bullet ele\} \land \\
&\quad tasks\ p = (alls\ p) \rhd \{FullTask \bullet ele\} \land \\
&\quad eerrors\ p = (alls\ p) \rhd \{End \mid (atom\ ele).type \in \operatorname{ran} eerror \bullet ele\} \land \\
&\quad nonsends\ p = \{s : Seqflow;\ g, f : Element \mid \\
&\qquad\qquad g \in_{\mathrm{p}} p \land f = (alls\ p)\ s \land s \in (atom\ g).out \land g \notin eventgate \bullet (s, f)\}
\end{aligned}$$

$$uniqueEnds : \mathbb{P}(\mathbb{F}_1\ Element)$$

$$\forall es : \mathbb{F}_1\ Element \bullet (es \in uniqueEnds \Leftrightarrow (es \in uniqueIns \land es \subseteq \{InitialEnd \bullet ele\}))$$

The function *cont* takes a process *ps* (nonempty finite set of elements) and a finite set of elements *es*, and returns a possibly empty subset of *ps* such that each element contains the set of elements *es*.

$$cont : (Process \times \mathbb{F}\ Element) \to \mathbb{P}\ Element$$

$$\begin{aligned}
&\forall ps : Process;\ es : \mathbb{F}\ Element \bullet \\
&\quad cont(ps, es) = \{e : ps \mid es \subseteq \{f : Element \mid f \in_{\mathrm{e}} e\}\}
\end{aligned}$$

The function *rep* takes a compound element and a nonempty finite set of elements (process), and replaces the content of the compound element with that finite set.

$$rep : (Element \times Process) \pfun Element$$

$$rep = (\lambda\ e : Element;\ ps : Process \mid e \in \operatorname{ran} compound \bullet compound(atom\ e, ps))$$

The function *modify* takes a process *ps* and three finite sets of elements *ns*, *rs*, *cs* such that it performs the following operation on *ps*:

- recursively finds the subprocess $s \in ps$ that contains *rs*;
- removes *rs* from *s* and adds elements from *ns* and *cs* to *s*.

This is a partial function as it is only defined for inputs where elements in rs are contained in ps.

$$
\begin{array}{|l}
modify : (Process \times \mathbb{F}\, Element \times \mathbb{F}_1\, Element) \pfun Process \\
\hline
\forall\, ps : Process;\ ns : \mathbb{F}\, Element;\ rs : \mathbb{F}_1\, Element \bullet \\
\quad (rs \subseteq ps \Rightarrow modify(ps, ns, rs) = ((ps \setminus rs) \cup ns)) \land \\
\quad (rs \subseteq (\{e : Element \mid e \in_{\mathrm{p}} ps\} \setminus ps) \Rightarrow \\
\qquad modify(ps, ns, rs) = \\
\qquad\quad ((ps \setminus cont(ps, rs)) \cup \{s : cont(ps, rs) \bullet rep(s, modify(content(s), ns, rs))\}))
\end{array}
$$

A.6.1.2 Operation Schemas

The following axiomatic definition *cge* specifies functionally the replacement of an incoming sequence flow of a BPMN element.

$$
\begin{array}{|l}
cge : (Element \times Seqflow \times Seqflow) \pfun Element \\
\hline
\forall\, e : Element;\ f, t : Seqflow \bullet \\
\quad \textbf{let}\ c == \langle\!| in \rightsquigarrow ((atom\ e).in \setminus \{\, f \,\}) \cup \{\, t \,\}, out \rightsquigarrow (atom\ e).out, \\
\qquad\qquad range \rightsquigarrow (atom\ e).range, exit \rightsquigarrow (atom\ e).exit, receive \rightsquigarrow (atom\ e).receive, \\
\qquad\qquad send \rightsquigarrow (atom\ e).send, type \rightsquigarrow (atom\ e).type |\!\rangle \bullet \\
\quad (e \in \mathrm{ran}\, atomic \Rightarrow cge(e, f, t) = atomic(c) \land \\
\quad e \in \mathrm{ran}\, compound \Rightarrow cge(e, f, t) = compound(c, content(e)))
\end{array}
$$

The axiomatic definition *ce* specifies the following operations on a BPMN element: The addition of an exception flow, and the replacement of a set of contained elements, if the BPMN element is a compound element.

$$
\begin{array}{|l}
ce : (Element \times (Seqflow \times Type) \times \mathbb{F}\, Element \times \mathbb{F}\, Element) \pfun Element \\
\hline
\forall\, e : Element;\ st : Seqflow \times Type;\ es, fs : \mathbb{F}\, Element \bullet \\
\quad \textbf{let}\ c == \langle\!| exit \rightsquigarrow ((atom\ e).exit \cup \{st\}), in \rightsquigarrow (atom\ e).in, out \rightsquigarrow (atom\ e).out, \\
\qquad\qquad range \rightsquigarrow (atom\ e).range, type \rightsquigarrow (atom\ e).type, receive \rightsquigarrow (atom\ e).receive, \\
\qquad\qquad send \rightsquigarrow (atom\ e).send |\!\rangle \bullet \\
\quad (e \in \mathrm{ran}\, atomic \Rightarrow ce(e, st, es, fs) = atomic(c) \land \\
\quad e \in \mathrm{ran}\, compound \Rightarrow ce(e, st, es, fs) = compound(c, ((content(e) \setminus es) \cup fs)))
\end{array}
$$

The definition *ct* specifies the replacement of *type* value of a BPMN element.

$$
\begin{array}{|l}
ct : (Element \times Type) \pfun Element \\
\hline
\forall\, e : Element;\ t : Type \bullet \\
\quad \textbf{let}\ c == \langle\!| type \rightsquigarrow t, in \rightsquigarrow (atom\ e).in, out \rightsquigarrow (atom\ e).out, range \rightsquigarrow (atom\ e).range, \\
\qquad\qquad exit \rightsquigarrow (atom\ e).exit, receive \rightsquigarrow (atom\ e).receive, send \rightsquigarrow (atom\ e).send |\!\rangle \bullet \\
\quad (e \in \mathrm{ran}\, atomic \Rightarrow ct(e, t) = atomic(c) \land \\
\quad e \in \mathrm{ran}\, compound \Rightarrow ct(e, t) = compound(c, content(e)))
\end{array}
$$

The definition *cm* specifies the addition of one or more incoming and outgoing message flows of a BPMN elment.

$$
\begin{array}{|l}
cm : (Element \times \mathbb{F}\, Mgeflow \times \mathbb{F}\, Mgeflow \times (Seqflow \pfun Type)) \pfun Element \\
\hline
\forall\, e : Element;\ sm, rm : \mathbb{F}\, Mgeflow;\ ef : Seqflow \pfun Type \bullet \\
\quad \textbf{let}\ c == \langle\!| send \rightsquigarrow (atom\ e).send \cup sm, receive \rightsquigarrow (atom\ e).receive \cup rm, \\
\qquad\qquad exit \rightsquigarrow ef, in \rightsquigarrow (atom\ e).in, out \rightsquigarrow (atom\ e).out, range \rightsquigarrow (atom\ e).range, \\
\qquad\qquad type \rightsquigarrow (atom\ e).type |\!\rangle \bullet \\
\quad e \in \mathrm{ran}\, atomic \Rightarrow cm(e, sm, rm, ef) = atomic(c)
\end{array}
$$

We provide a generic function that takes a relation r, and two values f and t, such that it overrides each pair of the form (s, f) for some s with the value (s, t).

$$[X, Y]$$

$$rrange : ((X \leftrightarrow Y) \times Y \times Y) \to (X \leftrightarrow Y)$$

$$\forall r : X \leftrightarrow Y;\ f, t : Y \bullet rrange(r, f, t) = r \oplus \{d : (\mathrm{dom}(r \rhd \{f\})) \bullet d \mapsto t\}$$

A.6.2 Collaboration

$$msgrecs, msgsends : Process \to (Seqflow \twoheadrightarrow Element)$$

$$\forall p : Process \bullet$$
$$msgsends\ p = \{s : Seqflow;\ g : Element \mid (alls\ p)\ s = g \land g \in \{ele : Element \mid EMgeEvent\}\} \land$$
$$msgrecs\ p = \{s : Seqflow;\ g : Element \mid$$
$$g \in p \land g \in \{ele : Element \mid SMgeEvent \lor IMgeEvent\} \land s \in (atom\ g).out\}$$

Appendix B

Preconditions

B.1 Preliminaries

Lemma B.1. $GenProc \Rightarrow [proc : Process \mid \forall\, e : \{g : Element \mid g \in_p proc\} \bullet e \in \{FlowObject \bullet ele\}]$

Proof.

$GenProc$
$\Leftrightarrow [WFProcess \mid proc \in onlyFlowObject]$ [def of $GenProc$]
$\Leftrightarrow [WFProcess \mid (\forall\, e : \{g : Element \mid g \in_{\mathrm{p}} proc\} \bullet e \in \{FlowObject \bullet ele\})]$ [def of $onlyFlowObject$]
$\Rightarrow [proc : Process \mid$ [def of $WFProcess$ and property of $\Rightarrow$]
$(\forall\, e : \{g : Element \mid g \in_{\mathrm{p}} proc\} \bullet e \in \{FlowObject \bullet ele\})]$

□

Lemma B.2 (4.12)**.**

$\forall\, ps, qs : Process \bullet$
$((ps \in \{proc : Process \mid Pool\} \land qs \in \{proc : Process \mid Pool\}) \Rightarrow$
$(\forall\, s : ps \bullet$
$s \in \operatorname{ran} compound \land$
$getSeqflows(qs) \cap (getSeqflows(ps \setminus \{s\}) \cup (atom\ s).in \cup outs(s)) = \emptyset \land$
$getMsgs(qs) \cap (getMsgs(ps \setminus \{s\}) \cup getMsg\ (atom\ s)) = \emptyset \land$
$\{e : qs \mid (atom\ e).type \in \operatorname{ran} eerror\} = \{e : content(s) \mid (atom\ e).type \in \operatorname{ran} eerror\} \Rightarrow$
$((ps \setminus \{s\}) \cup \{rep(s, qs)\}) \in \{proc : Process \mid Pool\}))$

Proof.

$\forall\, ps, qs : Process \bullet$
$((ps \in \{proc : Process \mid Pool\} \land$
$qs \in \{proc : Process \mid Pool\}) \Rightarrow$
$(\forall\, s : ps \bullet$
$s \in \operatorname{ran} compound \land$
$getSeqflows(qs) \cap (getSeqflows(ps \setminus \{s\}) \cup (atom\ s).in \cup outs(s)) = \emptyset \land$
$getMsgs(qs) \cap (getMsgs(ps \setminus \{s\}) \cup getMsg\ (atom\ s)) = \emptyset \land$
$\{e : qs \mid (atom\ e).type \in \operatorname{ran} eerror\} = \{e : content(s) \mid (atom\ e).type \in \operatorname{ran} eerror\} \Rightarrow$
$((ps \setminus \{s\}) \cup \{rep(s, qs)\}) \in \{proc : Process \mid Pool\}))$
$\Leftrightarrow \forall\, ps, qs : Process \bullet$ [def of $Pool$]
$((ps \in \{proc : Process \mid GenProc \land hasExgates\} \land$
$qs \in \{proc : Process \mid GenProc \land hasExgates\}) \Rightarrow$
$(\forall\, s : ps \bullet$
$s \in \operatorname{ran} compound \land$
$getSeqflows(qs) \cap (getSeqflows(ps \setminus \{s\}) \cup (atom\ s).in \cup outs(s)) = \emptyset \land$
$getMsgs(qs) \cap (getMsgs(ps \setminus \{s\}) \cup getMsg\ (atom\ s)) = \emptyset \land$

$$\{e : qs \mid (atom\ e).type \in \operatorname{ran} eerror\} = \{e : content(s) \mid (atom\ e).type \in \operatorname{ran} eerror\} \Rightarrow$$
$$((ps \setminus \{s\}) \cup \{rep(s, qs)\}) \in \{proc : Process \mid Pool\}))$$

$\Leftrightarrow \forall\, ps, qs : Process \bullet (\forall\, s : ps \bullet$ [def of *rep*, ∀-intro and property of ⇒]

$(ps \in \{proc : Process \mid GenProc \wedge hasExgates\} \wedge$
$qs \in \{proc : Process \mid GenProc \wedge hasExgates\} \wedge$
$s \in \operatorname{ran} compound \wedge$
$getSeqflows(qs) \cap (getSeqflows(ps \setminus \{s\}) \cup (atom\ s).in \cup outs(s)) = \emptyset \wedge$
$getMsgs(qs) \cap (getMsgs(ps \setminus \{s\}) \cup getMsg\ (atom\ s)) = \emptyset \wedge$
$\{e : qs \mid (atom\ e).type \in \operatorname{ran} eerror\} = \{e : content(s) \mid (atom\ e).type \in \operatorname{ran} eerror\}) \Rightarrow$
$((ps \setminus \{s\}) \cup \{compound(atom\ s, qs)\}) \in \{proc : Process \mid GenProc \wedge hasExgates\})$

$\Leftrightarrow \forall\, ps, qs : Process \bullet (\forall\, s : ps \bullet$ [set-compre]

$(ps \in \{proc : Process \mid GenProc \wedge hasExgates\} \wedge$
$qs \in \{proc : Process \mid GenProc \wedge hasExgates\} \wedge$
$s \in \operatorname{ran} compound \wedge$
$getSeqflows(qs) \cap (getSeqflows(ps \setminus \{s\}) \cup (atom\ s).in \cup outs(s)) = \emptyset \wedge$
$getMsgs(qs) \cap (getMsgs(ps \setminus \{s\}) \cup getMsg\ (atom\ s)) = \emptyset \wedge$
$\{e : qs \mid (atom\ e).type \in \operatorname{ran} eerror\} = \{e : content(s) \mid (atom\ e).type \in \operatorname{ran} eerror\}) \Rightarrow$
$((((ps \setminus \{s\}) \cup \{compound(atom\ s, qs)\}) \in \{proc : Process \mid GenProc\}) \wedge$
$(((ps \setminus \{s\}) \cup \{compound(atom\ s, qs)\}) \in \{proc : Process \mid hasExgates\})))$

We consider the first conjunct of the consequent.

$ps \in \{proc : Process \mid GenProc \wedge hasExgates\} \wedge$
$qs \in \{proc : Process \mid GenProc \wedge hasExgates\} \wedge$
$s \in \operatorname{ran} compound \wedge$
$getSeqflows(qs) \cap (getSeqflows(ps \setminus \{s\}) \cup (atom\ s).in \cup outs(s)) = \emptyset \wedge$
$getMsgs(qs) \cap (getMsgs(ps \setminus \{s\}) \cup getMsg\ (atom\ s)) = \emptyset \wedge$
$\{e : qs \mid (atom\ e).type \in \operatorname{ran} eerror\} = \{e : content(s) \mid (atom\ e).type \in \operatorname{ran} eerror\}$
$\Rightarrow ps \in \{proc : Process \mid GenProc\} \wedge$ [properties of ⇒]
$qs \in \{proc : Process \mid GenProc\} \wedge$
$s \in \operatorname{ran} compound \wedge$
$getSeqflows(qs) \cap (getSeqflows(ps \setminus \{s\}) \cup (atom\ s).in \cup outs(s)) = \emptyset \wedge$
$getMsgs(qs) \cap (getMsgs(ps \setminus \{s\}) \cup getMsg\ (atom\ s)) = \emptyset \wedge$
$\{e : qs \mid (atom\ e).type \in \operatorname{ran} eerror\} = \{e : content(s) \mid (atom\ e).type \in \operatorname{ran} eerror\}$
$\Leftrightarrow ps \in \{proc : Process \mid WFProcess\} \wedge$ [def of *GenProc*]
$ps \in onlyFlowObject \wedge$
$qs \in \{proc : Process \mid GenProc\} \wedge$
$s \in \operatorname{ran} compound \wedge$
$getSeqflows(qs) \cap (getSeqflows(ps \setminus \{s\}) \cup (atom\ s).in \cup outs(s)) = \emptyset \wedge$
$getMsgs(qs) \cap (getMsgs(ps \setminus \{s\}) \cup getMsg\ (atom\ s)) = \emptyset \wedge$
$\{e : qs \mid (atom\ e).type \in \operatorname{ran} eerror\} = \{e : content(s) \mid (atom\ e).type \in \operatorname{ran} eerror\}$
$\Leftrightarrow ps \in \{proc : Process \mid WFProcess\} \wedge$ [def of *onlyFlowObject*]
$(\forall\, s : \{g : Element \mid g \in_{\mathrm{p}} ps\} \bullet s \in \{FlowObject \bullet ele\}) \wedge$
$qs \in \{proc : Process \mid GenProc\} \wedge$
$s \in \operatorname{ran} compound \wedge$
$getSeqflows(qs) \cap (getSeqflows(ps \setminus \{s\}) \cup (atom\ s).in \cup outs(s)) = \emptyset \wedge$

$$
\begin{array}{ll}
& getMsgs(qs) \cap (getMsgs(ps \setminus \{s\}) \cup getMsg\,(atom\ s)) = \emptyset \land \\
& \{e : qs \mid (atom\ e).type \in \operatorname{ran} eerror\} = \{e : content(s) \mid (atom\ e).type \in \operatorname{ran} eerror\} \\
\Leftrightarrow & ps \in \{proc : Process \mid WFProcess\} \land \qquad [\text{def of } \in_{\mathrm{p}},\ FlowObject \text{ and properties of } \forall] \\
& (\forall s : \{g : Element \mid g \in_{\mathrm{p}} ps\} \bullet s \in \{FlowObject \bullet ele\}) \land \\
& qs \in \{proc : Process \mid GenProc\} \land \\
& s \in \operatorname{ran} compound \land \\
& getSeqflows(qs) \cap (getSeqflows(ps \setminus \{s\}) \cup (atom\ s).in \cup outs(s)) = \emptyset \land \\
& getMsgs(qs) \cap (getMsgs(ps \setminus \{s\}) \cup getMsg\,(atom\ s)) = \emptyset \land \\
& \{e : qs \mid (atom\ e).type \in \operatorname{ran} eerror\} = \{e : content(s) \mid (atom\ e).type \in \operatorname{ran} eerror\} \land \\
& s \in \{FullSub \bullet ele\} \\
\Leftrightarrow & ps \in \{proc : Process \mid WFProcess\} \land \qquad [\text{def of } GenProc] \\
& (\forall s : \{g : Element \mid g \in_{\mathrm{p}} ps\} \bullet s \in \{FlowObject \bullet ele\}) \land \\
& qs \in \{proc : Process \mid WFProcess\} \land \\
& (\forall e : \{g : Element \mid g \in_{\mathrm{p}} qs\} \bullet e \in \{FlowObject \bullet ele\}) \land \\
& s \in \operatorname{ran} compound \land \\
& getSeqflows(qs) \cap (getSeqflows(ps \setminus \{s\}) \cup (atom\ s).in \cup outs(s)) = \emptyset \land \\
& getMsgs(qs) \cap (getMsgs(ps \setminus \{s\}) \cup getMsg\,(atom\ s)) = \emptyset \land \\
& \{e : qs \mid (atom\ e).type \in \operatorname{ran} eerror\} = \{e : content(s) \mid (atom\ e).type \in \operatorname{ran} eerror\} \land \\
& s \in \{FullSub \bullet ele\} \\
\Rightarrow & \qquad [\{e : qs \mid (atom\ e).type \in \operatorname{ran} eerror\} = \{e : content(s) \mid (atom\ e).type \in \operatorname{ran} eerror\}] \\
& ps \in \{proc : Process \mid WFProcess\} \land \\
& (\forall s : \{g : Element \mid g \in_{\mathrm{p}} ps\} \bullet s \in \{FlowObject \bullet ele\}) \land \\
& qs \in \{proc : Process \mid WFProcess\} \land \\
& (\forall e : \{g : Element \mid g \in_{\mathrm{p}} qs\} \bullet e \in \{FlowObject \bullet ele\}) \land \\
& s \in \operatorname{ran} compound \land \\
& getSeqflows(qs) \cap (getSeqflows(ps \setminus \{s\}) \cup (atom\ s).in \cup outs(s)) = \emptyset \land \\
& getMsgs(qs) \cap (getMsgs(ps \setminus \{s\}) \cup getMsg\,(atom\ s)) = \emptyset \land \\
& \{e : qs \mid (atom\ e).type \in \operatorname{ran} eerror\} = \{e : content(s) \mid (atom\ e).type \in \operatorname{ran} eerror\} \land \\
& rep(s, qs) \in \{FullSub \bullet ele\} \\
\Rightarrow & ps \in \{proc : Process \mid WFProcess\} \land \qquad [\text{def of } FlowObject] \\
& qs \in \{proc : Process \mid WFProcess\} \land \\
& s \in \operatorname{ran} compound \land \\
& getSeqflows(qs) \cap (getSeqflows(ps \setminus \{s\}) \cup (atom\ s).in \cup outs(s)) = \emptyset \land \\
& getMsgs(qs) \cap (getMsgs(ps \setminus \{s\}) \cup getMsg\,(atom\ s)) = \emptyset \land \\
& \{e : qs \mid (atom\ e).type \in \operatorname{ran} eerror\} = \{e : content(s) \mid (atom\ e).type \in \operatorname{ran} eerror\} \land \\
& (\forall t : \{g : Element \mid g \in_{\mathrm{p}} ((ps \setminus \{s\}) \cup \{rep(s, qs)\})\} \bullet t \in \{FlowObject \bullet ele\}) \\
\Leftrightarrow & ps \in noOverLap \land \qquad [\text{def of } WFProcess \text{ and property of } \cap] \\
& ps \in processSet \land \\
& (\exists e, f : ps \bullet e \in \{Start \bullet ele\} \land f \in \{End \bullet ele\}) \land \\
& (\neg\, (\exists e : ps \bullet e \in \{Inter \mid (atom\ ele).type \in \operatorname{ran} ierror \bullet ele\})) \land \\
& qs \in noOverLap \land \\
& qs \in processSet \land \\
& (\exists e, f : qs \bullet e \in \{Start \bullet ele\} \land f \in \{End \bullet ele\}) \land \\
& (\neg\, (\exists e : qs \bullet e \in \{Inter \mid (atom\ ele).type \in \operatorname{ran} ierror \bullet ele\})) \land
\end{array}
$$

$s \in \operatorname{ran} compound \wedge$
$getSeqflows(qs) \cap (getSeqflows(ps \setminus \{s\}) \cup (atom\, s).in \cup outs(s)) = \emptyset \wedge$
$getMsgs(qs) \cap (getMsgs(ps \setminus \{s\}) \cup getMsg\,(atom\, s)) = \emptyset \wedge$
$\{e : qs \mid (atom\, e).type \in \operatorname{ran} eerror\} = \{e : content(s) \mid (atom\, e).type \in \operatorname{ran} eerror\} \wedge$
$(\forall t : \{g : Element \mid g \in_{\mathrm{p}} ((ps \setminus \{s\}) \cup \{rep(s, qs)\})\} \bullet t \in \{FlowObject \bullet ele\})$
$\Leftrightarrow (\forall e : \{g : Element \mid g \in_{\mathrm{p}} ps\}$ [def of *noOverLap*]
$\bigcup\{k : ps \bullet (atom\, k).in \cup outs(k)\} \cap \bigcup\{k : content\, e \bullet (atom\, k).in \cup outs(k)\} = \emptyset \wedge$
$(\forall f : \{g : Element \mid g \in_{\mathrm{p}} ps\} \bullet e \neq f \Rightarrow$
$(atom\, e).in \cap (atom\, f).in = \emptyset \wedge$
$outs(e) \cap outs(f) = \emptyset \wedge$
$\bigcup\{k : content\, e \bullet (atom\, k).in \cup outs(k)\} \cap \bigcup\{k : content\, f \bullet (atom\, k).in \cup outs(k)\} = \emptyset \wedge$
$getMsg\,(atom\, e) \cap getMsg\,(atom\, f) = \emptyset)) \wedge$
$ps \in processSet \wedge$
$(\exists e, f : ps \bullet e \in \{Start \bullet ele\} \wedge f \in \{End \bullet ele\}) \wedge$
$(\neg (\exists e : ps \bullet e \in \{Inter \mid (atom\, ele).type \in \operatorname{ran} ierror \bullet ele\})) \wedge$
$(\forall e : \{g : Element \mid g \in_{\mathrm{p}} qs\} \bullet$
$\bigcup\{k : qs \bullet (atom\, k).in \cup outs(k)\} \cap \bigcup\{k : content\, e \bullet (atom\, k).in \cup outs(k)\} = \emptyset \wedge$
$(\forall f : \{g : Element \mid g \in_{\mathrm{p}} qs\} \bullet e \neq f \Rightarrow$
$(atom\, e).in \cap (atom\, f).in = \emptyset \wedge$
$outs(e) \cap outs(f) = \emptyset \wedge$
$\bigcup\{k : content\, e \bullet (atom\, k).in \cup outs(k)\} \cap \bigcup\{k : content\, f \bullet (atom\, k).in \cup outs(k)\} = \emptyset \wedge$
$getMsg\,(atom\, e) \cap getMsg\,(atom\, f) = \emptyset)) \wedge$
$qs \in processSet \wedge$
$(\exists e, f : qs \bullet e \in \{Start \bullet ele\} \wedge f \in \{End \bullet ele\}) \wedge$
$(\neg (\exists e : qs \bullet e \in \{Inter \mid (atom\, ele).type \in \operatorname{ran} ierror \bullet ele\})) \wedge$
$s \in \operatorname{ran} compound \wedge$
$getSeqflows(qs) \cap (getSeqflows(ps \setminus \{s\}) \cup (atom\, s).in \cup outs(s)) = \emptyset \wedge$
$getMsgs(qs) \cap (getMsgs(ps \setminus \{s\}) \cup getMsg\,(atom\, s)) = \emptyset \wedge$
$\{e : qs \mid (atom\, e).type \in \operatorname{ran} eerror\} = \{e : content(s) \mid (atom\, e).type \in \operatorname{ran} eerror\} \wedge$
$(\forall t : \{g : Element \mid g \in_{\mathrm{p}} ((ps \setminus \{s\}) \cup \{rep(s, qs)\})\} \bullet t \in \{FlowObject \bullet ele\})$
$\Rightarrow ps \in processSet \wedge$ [Sequence flows and message flows of qs and $ps \setminus \{s\}$ do not intersect]
$(\exists e, f : ps \bullet e \in \{Start \bullet ele\} \wedge f \in \{End \bullet ele\}) \wedge$
$(\neg (\exists e : ps \bullet e \in \{Inter \mid (atom\, ele).type \in \operatorname{ran} ierror \bullet ele\})) \wedge$
$qs \in processSet \wedge$
$(\exists e, f : qs \bullet e \in \{Start \bullet ele\} \wedge f \in \{End \bullet ele\}) \wedge$
$(\neg (\exists e : qs \bullet e \in \{Inter \mid (atom\, ele).type \in \operatorname{ran} ierror \bullet ele\})) \wedge$
$s \in \operatorname{ran} compound \wedge$
$getSeqflows(qs) \cap (getSeqflows(ps \setminus \{s\}) \cup (atom\, s).in \cup outs(s)) = \emptyset \wedge$
$getMsgs(qs) \cap (getMsgs(ps \setminus \{s\}) \cup getMsg\,(atom\, s)) = \emptyset \wedge$
$\{e : qs \mid (atom\, e).type \in \operatorname{ran} eerror\} = \{e : content(s) \mid (atom\, e).type \in \operatorname{ran} eerror\} \wedge$
$(\forall t : \{g : Element \mid g \in_{\mathrm{p}} ((ps \setminus \{s\}) \cup \{rep(s, qs)\})\} \bullet t \in \{FlowObject \bullet ele\}) \wedge$
$((ps \setminus \{s\}) \cup \{rep(s, qs)\}) \in noOverLap$
$\Rightarrow ps \in processSet \wedge$ [def of ps and qs]
$qs \in processSet \wedge$

$s \in \operatorname{ran} compound \land$
$getSeqflows(qs) \cap (getSeqflows(ps \setminus \{s\}) \cup (atom\, s).in \cup outs(s)) = \emptyset \land$
$getMsgs(qs) \cap (getMsgs(ps \setminus \{s\}) \cup getMsg\,(atom\, s)) = \emptyset \land$
$\{e : qs \mid (atom\, e).type \in \operatorname{ran} eerror\} = \{e : content(s) \mid (atom\, e).type \in \operatorname{ran} eerror\} \land$
$(\forall t : \{g : Element \mid g \in_{\mathrm{p}} ((ps \setminus \{s\}) \cup \{rep(s, qs)\})\} \bullet t \in \{FlowObject \bullet ele\}) \land$
$((ps \setminus \{s\}) \cup \{rep(s, qs)\}) \in noOverLap \land$
$(\exists e, f : ((ps \setminus \{s\}) \cup \{rep(s, qs)\}) \bullet e \in \{Start \bullet ele\} \land f \in \{End \bullet ele\}) \land$
$(\neg (\exists e : ((ps \setminus \{s\}) \cup \{rep(s, qs)\}) \bullet e \in \{Inter \mid (atom\, ele).type \in \operatorname{ran} ierror \bullet ele\}))$
$\Leftrightarrow (\forall e : ps \bullet$ [def of *processSet* and property of $\cap$]
$(e \notin \{End \bullet ele\} \Rightarrow (\exists f : ps \bullet f \in \{End \bullet ele\} \land (e, f) \in (edge\, ps)^+)) \land$
$(e \notin \{Start \bullet ele\} \Rightarrow (\exists f : ps \bullet f \in \{Start \bullet ele\} \land (f, e) \in (edge\, ps)^+))) \land$
$(\forall e : ps \bullet$
$(\forall s : outs(e) \bullet \exists f : ps \bullet s \in (atom\, f).in) \land$
$(\forall s : (atom\, e).in \bullet \exists f : ps \bullet s \in outs(f))) \land$
$qs \in processSet \land$
$s \in \operatorname{ran} compound \land$
$getSeqflows(qs) \cap (getSeqflows(ps \setminus \{s\}) \cup (atom\, s).in \cup outs(s)) = \emptyset \land$
$getMsgs(qs) \cap (getMsgs(ps \setminus \{s\}) \cup getMsg\,(atom\, s)) = \emptyset \land$
$\{e : qs \mid (atom\, e).type \in \operatorname{ran} eerror\} = \{e : content(s) \mid (atom\, e).type \in \operatorname{ran} eerror\} \land$
$(\forall t : \{g : Element \mid g \in_{\mathrm{p}} ((ps \setminus \{s\}) \cup \{rep(s, qs)\})\} \bullet t \in \{FlowObject \bullet ele\}) \land$
$((ps \setminus \{s\}) \cup \{rep(s, qs)\}) \in noOverLap \land$
$(\exists e, f : ((ps \setminus \{s\}) \cup \{rep(s, qs)\}) \bullet e \in \{Start \bullet ele\} \land f \in \{End \bullet ele\}) \land$
$(\neg (\exists e : ((ps \setminus \{s\}) \cup \{rep(s, qs)\}) \bullet e \in \{Inter \mid (atom\, ele).type \in \operatorname{ran} ierror \bullet ele\}))$
$\Rightarrow$ [$qs \cap content(ps) = \emptyset$ and def of *GenProc*]
$((ps \setminus \{s\}) \cup \{rep(s, qs)\}) \in \{proc : Process \mid GenProc\}$

We consider the second conjunct of the consequent.

$ps \in \{proc : Process \mid GenProc \land hasExgates\} \land$
$qs \in \{proc : Process \mid GenProc \land hasExgates\}$
$s \in \operatorname{ran} compound \land$
$getSeqflows(qs) \cap (getSeqflows(ps \setminus \{s\}) \cup (atom\, s).in \cup outs(s)) = \emptyset \land$
$getMsgs(qs) \cap (getMsgs(ps \setminus \{s\}) \cup getMsg\,(atom\, s)) = \emptyset \land$
$\{e : qs \mid (atom\, e).type \in \operatorname{ran} eerror\} = \{e : content(s) \mid (atom\, e).type \in \operatorname{ran} eerror\}$
$\Rightarrow ps \in \{proc : Process \mid hasExgates\} \land$ [properties of $\Rightarrow$]
$qs \in \{proc : Process \mid hasExgates\}$
$s \in \operatorname{ran} compound \land$
$getSeqflows(qs) \cap (getSeqflows(ps \setminus \{s\}) \cup (atom\, s).in \cup outs(s)) = \emptyset \land$
$getMsgs(qs) \cap (getMsgs(ps \setminus \{s\}) \cup getMsg\,(atom\, s)) = \emptyset \land$
$\{e : qs \mid (atom\, e).type \in \operatorname{ran} eerror\} = \{e : content(s) \mid (atom\, e).type \in \operatorname{ran} eerror\}$
$\Leftrightarrow (\forall f : \{g : Element \mid g \in_{\mathrm{p}} ps\} \bullet f \in eventgate \Rightarrow$ [def of *hasExgates*]
$(\forall e : Element \bullet (f, e) \in edge(direct(ps, e)) \Rightarrow e \in sendelement)) \land$
$(\forall f : \{g : Element \mid g \in_{\mathrm{p}} qs\} \bullet f \in eventgate \Rightarrow$
$(\forall e : Element \bullet (f, e) \in edge(direct(qs, e)) \Rightarrow e \in sendelement)) \land$
$s \in \operatorname{ran} compound \land$
$getSeqflows(qs) \cap (getSeqflows(ps \setminus \{s\}) \cup (atom\, s).in \cup outs(s)) = \emptyset \land$

$$
\begin{array}{ll}
& getMsgs(qs) \cap (getMsgs(ps \setminus \{s\}) \cup getMsg\,(atom\ s)) = \emptyset \land \\
& \{e : qs \mid (atom\ e).type \in \operatorname{ran} eerror\} = \{e : content(s) \mid (atom\ e).type \in \operatorname{ran} eerror\} \\
\Rightarrow & \qquad [\text{def of } direct, \text{ Sequence flows of } qs \text{ and } ps \setminus \{s\} \text{ do not intersect}] \\
& (\forall f : \{g : Element \mid g \in_{\mathrm{p}} ((ps \setminus \{s\}) \cup \{rep(s, qs)\})\} \bullet f \in eventgate \Rightarrow \\
& \quad (\forall e : Element \bullet (f, e) \in edge(direct(((ps \setminus \{s\}) \cup \{rep(s, qs)\}), e)) \Rightarrow e \in sendelement)) \\
\Leftrightarrow & ((ps \setminus \{s\}) \cup \{rep(s, qs)\}) \in \{proc : Process \mid hasExgates\} \qquad [\text{def of } hasExgates]
\end{array}
$$

We have shown all constraints specified in the consequent follow from constraints in the antecedent. This completes the proof. □

B.2 Precondition of *ChangeFlow*

B.2.1 Preliminaries

Lemma B.3.

$$
\begin{array}{l}
\forall\, c : Element;\ f, t : Seqflow \bullet \\
\quad t \notin (atom\ c).in \land f \in (atom\ c).in \Rightarrow \#(atom(cge(c, f, t))).in = \#(atom\ c).in
\end{array}
$$

Proof.

$$
\begin{array}{lr}
\#(atom\ cge(c, f, t)).in & \\
= \#(((atom\ c).in \setminus \{f\}) \cup \{t\}) & [\text{def of } cge] \\
= \#((atom\ c).in \setminus \{f\}) + 1 & [t \notin (atom\ c).in] \\
= \#(atom\ c).in - 1 + 1 & [f \in (atom\ c).in] \\
= \#(atom\ c).in & [\text{arith}]
\end{array}
$$

□

B.2.2 Simplification

$$
\begin{array}{ll}
& \text{pre } ChangeFlow \\
\Leftrightarrow & [FlowObject;\ from?, to? : Seqflow \qquad [\text{def of } ChangeFlow, \text{ schema quantification, one-point}] \\
& \quad ele \notin \{Start \bullet ele\} \land \\
& \quad to? \notin getSeqflows\{ele\} \land \\
& \quad from? \in (atom\ ele).in \land \\
& \quad cge(ele, from?, to?) \in \{ele : Element \mid FlowObject\}]
\end{array}
$$

We now show that $cge(ele, from?, to?) \in \{ele : Element \mid FlowObject\}$ follows from both the declaration and constraints on the input components of the precondition schema. We first expand the constraint on the before state component $ele \in \{ele : Element \mid FlowObject\}$.

$$
\begin{array}{ll}
& ele \in \{ele : Element \mid FlowObject\} \\
\Leftrightarrow & ele \in \{ele : Element \mid Event \lor FullTask \lor FullSub \lor Gate\} \qquad [\text{def of } FlowObject] \\
\Leftrightarrow & (ele \in \{ele : Element \mid Event\} \lor \qquad [\text{set-compre}] \\
& \quad (ele \in \{ele : Element \mid FullTask\} \lor \\
& \quad (ele \in \{ele : Element \mid FullSub\} \lor \\
& \quad (ele \in \{ele : Element \mid Gate\}
\end{array}
$$

We now consider the first disjunct $ele \in \{ele : Element \mid Event\}$.

$$
ele \in \{ele : Element \mid Event\} \land
$$

$ele \notin \{Start \bullet ele\} \land$
$to? \notin getSeqflows\{ele\} \land$
$from? \in (atom\ ele).in$
$\Leftrightarrow ((ele \in \operatorname{ran} atomic \land$ [def of *Event*, $ele \notin \{Start \bullet ele\}$ and properties of $\lor$]
$\#(atom\ ele).exit+$
$\#(atom\ ele).send + \#(atom\ ele).receive = 0 \land$
$first(atom\ ele).range = 0 \land$
$second(atom\ ele).range = 0) \land$
$(((atomic^{\sim} ele).type \in \operatorname{ran} itime \cup \operatorname{ran} imessage \cup \operatorname{ran} ierror \cup \operatorname{ran} irule \land$
$\#(atom\ ele).in = 1 \land \#(atom\ ele).out = 1) \lor$
$((atom\ ele).type \in \{ end, abort \} \cup \operatorname{ran} emessage \cup \operatorname{ran} eerror \land$
$\#(atom\ ele).in = 1 \land \#(atom\ ele).out = 0))) \land$
$ele \notin \{Start \bullet ele\} \land$
$to? \notin getSeqflows\{ele\} \land$
$from? \in (atom\ ele).in \land$
$ele \in Element$
$ele \notin \{Start \bullet ele\} \land$
$to? \notin getSeqflows\{ele\} \land$
$from? \in (atom\ ele).in$
$\Rightarrow ((cge(ele, from?, to?) \in \operatorname{ran} atomic \land$ [def of *cge*]
$\#(atom\ cge(ele, from?, to?)).exit+$
$\#(atom\ cge(ele, from?, to?)).send + \#(atom\ cge(ele, from?, to?)).receive = 0 \land$
$first(atom\ cge(ele, from?, to?)).range = 0 \land$
$second(atom\ cge(ele, from?, to?)).range = 0) \land$
$(((atomic^{\sim} cge(ele, from?, to?)).type \in \operatorname{ran} itime \cup \operatorname{ran} imessage \cup \operatorname{ran} ierror \cup \operatorname{ran} irule \land$
$\#(atom\ cge(ele, from?, to?)).in = 1 \land \#(atom\ cge(ele, from?, to?)).out = 1) \lor$
$((atom\ cge(ele, from?, to?)).type \in \{ end, abort \} \cup \operatorname{ran} emessage \cup \operatorname{ran} eerror \land$
$\#(atom\ cge(ele, from?, to?)).in = 1 \land \#(atom\ cge(ele, from?, to?)).out = 0)))$
$\Rightarrow cge(ele, from?, to?) \in \{ele : Element \mid Event\}$ [def of *Event* and set-compre]
$\Rightarrow cge(ele, from?, to?) \in \{ele : Element \mid FlowObject\}$ [def of *FlowObject*]

We now consider the second disjunct $ele \in \{ele : Element \mid FullTask\}$.

$ele \in \{ele : Element \mid FullTask\} \land$
$ele \notin \{Start \bullet ele\} \land$
$to? \notin getSeqflows\{ele\} \land$
$from? \in (atom\ ele).in$
$\Leftrightarrow ele \in \operatorname{ran} atomic \land$ [def of *FullTask*]
$(atom\ ele).type \in \operatorname{ran} task \cup \operatorname{ran} miseq \cup \operatorname{ran} mipar \land$
$first((atom\ ele).range) \leq second((atom\ ele).range) \land$
$\#(atom\ ele).in = 1 \land \#(atom\ ele).out = 1 \land$
$(\forall t : Type \bullet t \in (\operatorname{ran}((atom\ ele).exit) \cap \operatorname{ran} ierror) \Rightarrow (ierror^{\sim} t) = anyexception) \land$
$(\forall t : Type \bullet t \in \operatorname{ran}((atom\ ele).exit) \Rightarrow t \in inters) \land$
$ele \notin \{Start \bullet ele\} \land$
$to? \notin getSeqflows\{ele\} \land$
$from? \in (atom\ ele).in$

$\Rightarrow ((cge(ele, from?, to?) \in \operatorname{ran} atomic \wedge$ [def of cge, properties of $\cup$ and Lemma B.3]
$(atom\ cge(ele, from?, to?)).type \in \operatorname{ran} task \cup \operatorname{ran} miseq \cup \operatorname{ran} mipar \wedge$
$first((atom\ cge(ele, from?, to?)).range) \leq second((atom\ cge(ele, from?, to?)).range) \wedge$
$\#(atom\ cge(ele, from?, to?)).in = 1 \wedge \#(atom\ cge(ele, from?, to?)).out = 1 \wedge$
$(\forall t : Type \bullet t \in (\operatorname{ran}((atom\ cge(ele, from?, to?)).exit) \cap \operatorname{ran} ierror) \Rightarrow (ierror^{\sim} t) = anyexception)) \wedge$
$(\forall t : Type \bullet t \in \operatorname{ran}((atom\ cge(ele, from?, to?)).exit) \Rightarrow t \in inters)) \wedge$
$(to? \notin (atom\ cge(ele, from?, to?)).in \wedge$
$to? \notin (atom\ cge(ele, from?, to?)).out \wedge$
$to? \notin \operatorname{dom}(atom\ cge(ele, from?, to?)).exit) \wedge$
$from? \in (atom\ cge(ele, from?, to?)).in$
$\Leftrightarrow cge(ele, from?, to?) \in \{ele : Element \mid FullTask\}$ [def of $FullTask$ and set-compre]
$\Rightarrow cge(ele, from?, to?) \in \{ele : Element \mid FlowObject\}$ [def of $FlowObject$]

We now consider the third disjunct $ele \in \{ele : Element \mid FullSub\}$.

$ele \in \{ele : Element \mid FullSub\} \wedge$
$ele \notin \{Start \bullet ele\} \wedge$
$to? \notin getSeqflows\{ele\} \wedge$
$from? \in (atom\ ele).in$
$\Leftrightarrow ((ele \in \operatorname{ran} compound \wedge$ [def of $FullSub$]
$\#(atom\ ele).send + \#(atom\ ele).receive = 0 \wedge$
$(atom\ ele).type \in \operatorname{ran} subprocess \cup \operatorname{ran} mipars \cup \operatorname{ran} miseqs \wedge$
$content(ele) \neq \emptyset \wedge$
$content(ele) \in \{WFProcess \bullet proc\} \wedge$
$\#(atom\ ele).in = 1 \wedge \#(atom\ ele).out = 0) \wedge$
$((\forall e, f : content(ele) \bullet$
$\{e, f\} \subseteq \{End \mid (atom\ ele).type \in (\operatorname{ran} eerror \cap errorCodeTypes) \bullet ele\} \wedge$
$errorCode((atom\ e).type) = errorCode((atom\ f).type) \Rightarrow f = e) \wedge$
$(\forall e : content(ele) \bullet$
$(atom\ e).type \in (\operatorname{ran} eerror \cap errorCodeTypes) \Rightarrow$
$(\exists t : \operatorname{ran}((atom\ ele).exit) \bullet$
$t \in (\operatorname{ran} ierror \cap errorCodeTypes) \wedge$
$errorCode((atom\ e).type) = errorCode(t))) \wedge$
$(\forall t : \operatorname{ran}((atom\ ele).exit) \bullet$
$(t \in (\operatorname{ran} ierror \cap errorCodeTypes) \Rightarrow$
$(\exists e : content(ele) \bullet$
$(atom\ e).type \in (\operatorname{ran} eerror \cap errorCodeTypes) \wedge$
$errorCode((atom\ e).type) = errorCode(t)))) \wedge$
$(\forall t : Type \bullet t \in \operatorname{ran}((atom\ ele).exit) \Rightarrow t \in inters))) \wedge$
$ele \notin \{Start \bullet ele\} \wedge$
$to? \notin getSeqflows\{ele\} \wedge$
$from? \in (atom\ ele).in$
$\Rightarrow ((cge(ele, from?, to?) \in \operatorname{ran} compound \wedge$ [def of cge and Lemma B.3]
$\#(atom\ cge(ele, from?, to?)).send + \#(atom\ cge(ele, from?, to?)).receive = 0 \wedge$
$(atom\ cge(ele, from?, to?)).type \in \operatorname{ran} subprocess \cup \operatorname{ran} mipars \cup \operatorname{ran} miseqs \wedge$
$content(cge(ele, from?, to?)) \neq \emptyset \wedge$

$$
\begin{array}{l}
content(cge(ele, from?, to?)) \in \{ WFProcess \bullet proc\} \land \\
\#(atom\ cge(ele, from?, to?)).in = 1 \land \#(atom\ cge(ele, from?, to?)).out = 1) \land \\
((\forall\, e, f : content(cge(ele, from?, to?)) \bullet \\
\quad \{e, f\} \subseteq \{End \mid (atom\ cge(ele, from?, to?)).type \in (\mathrm{ran}\ eerror \cap errorCodeTypes) \bullet ele\} \land \\
\quad errorCode((atom\ e).type) = errorCode((atom\ f).type) \Rightarrow f = e) \land \\
(\forall\, e : content(cge(ele, from?, to?)) \bullet \\
\quad (atom\ e).type \in (\mathrm{ran}\ eerror \cap errorCodeTypes) \Rightarrow \\
\qquad (\exists\, t : Type \bullet \\
\qquad\quad t \in \mathrm{ran}((atom\ cge(ele, from?, to?)).exit) \land \\
\qquad\quad t \in (\mathrm{ran}\ ierror \cap errorCodeTypes) \land \\
\qquad\quad errorCode((atom\ e).type) = errorCode(t))) \land \\
(\forall\, t : \mathrm{ran}((atom\ cge(ele, from?, to?)).exit) \bullet \\
\quad (t \in (\mathrm{ran}\ ierror \cap errorCodeTypes) \Rightarrow \\
\qquad (\exists\, e : content(cge(ele, from?, to?)) \bullet \\
\qquad\quad (atom\ e).type \in (\mathrm{ran}\ eerror \cap errorCodeTypes) \land \\
(\forall\, t : Type \bullet t \in \mathrm{ran}((atom\ cge(ele, from?, to?)).exit) \Rightarrow t \in inters))) \\
\Leftrightarrow cge(ele, from?, to?) \in \{ele : Element \mid FullSub\}\, defof\, \mathrm{FullSub}\, andset - compre \\
\Rightarrow cge(ele, from?, to?) \in \{ele : Element \mid FlowObject\} \qquad \text{[def of } FlowObject\text{]}
\end{array}
$$

We now consider the fourth disjunct $ele \in \{ele : Element \mid Gate\}$.

$$
\begin{array}{l}
ele \in \{ele : Element \mid Gate\} \land \\
ele \notin \{Start \bullet ele\} \land \\
to? \notin getSeqflows\{ele\} \land \\
from? \in (atom\ ele).in \\
\Leftrightarrow ((atom\ ele).type \in \{agate, xgate, exgate\} \land \\
\#(atom\ ele).in = 1 \lor \#(atom\ ele).out = 1 \land \\
\#(atom\ ele).in = 1 \Rightarrow \#(atom\ ele).out > 1) \land \\
ele \notin \{Start \bullet ele\} \land \\
to? \notin getSeqflows\{ele\} \land \\
from? \in (atom\ ele).in \\
\Rightarrow ((atom\ cge(ele, from?, to?)).type \in \{agate, xgate, exgate\} \land \qquad \text{[def of } cge\text{]} \\
\#(atom\ cge(ele, from?, to?)).in = 1 \lor \#(atom\ cge(ele, from?, to?)).out = 1 \land \\
\#(atom\ cge(ele, from?, to?)).in = 1 \Rightarrow \#(atom\ cge(ele, from?, to?)).out > 1) \\
\Leftrightarrow cge(ele, from?, to?) \in \{ele : Element \mid Gate\} \qquad \text{[def of } Gate \text{ and set-compre]} \\
\Rightarrow cge(ele, from?, to?) \in \{ele : Element \mid FlowObject\} \qquad \text{[def of } FlowObject\text{]}
\end{array}
$$

The above deductions result in the following simplified precondition schema, labelled *PreChangeFlow*.

PreChangeFlow

$$
\begin{array}{l}
FlowObject \\
from?, to? : Seqflow \\
\hline
ele \notin \{Start \bullet ele\} \\
to? \notin getSeqflows\{ele\} \\
from? \in (atom\ ele).in
\end{array}
$$

□

B.3 Precondition of *AddNoRelatedErrorExceptionSub*

$$
\begin{array}{l}
\text{pre } AddNoRelatedErrorExceptionSub \\
\Leftrightarrow \exists\, Activity' \bullet AddNoRelatedErrorExceptionSub \\
\Leftrightarrow \exists\, Activity' \bullet \\
\quad [\Delta Activity;\ etype? : Type;\ eflow? : Seqflow \mid \\
\qquad etype? \in nomsgserrors \land \\
\qquad eflow? \notin getSeqflows\{ele\} \land \\
\qquad ele' = ce(ele, (eflow?, etype?), \emptyset, \emptyset)] \\
\Leftrightarrow [Activity;\ etype? : Type;\ eflow? : Seqflow \mid & \text{[schema quantification and one-point]} \\
\quad \textbf{let } ch == ce(ele, (eflow?, etype?), \emptyset, \emptyset) \bullet \\
\quad etype? \in nomsgserrors \land \\
\quad eflow? \notin getSeqflows\{ele\} \land \\
\quad ch \in \{ele : Element \mid Activity\}]
\end{array}
$$

Now to show $ch \in \{ele : Element \mid Activity\}$ follows from constraints on the before state *Pool* and the input components, we first consider if $ed \in \{ele : Element \mid FullTask\}$.

$$
\begin{array}{l}
ed \in \{ele : Element \mid FullTask\} \land \\
etype? \in nomsgserrors \land \\
eflow? \notin getSeqflows\, proc \\
\Leftrightarrow ed \in \operatorname{ran} atomic \land & \text{[def of } FullTask\text{]} \\
\quad (atom\ ed).type \in \operatorname{ran} task \cup \operatorname{ran} sloop \cup \operatorname{ran} miseq \cup \operatorname{ran} mipar \land \\
\quad first((atom\ ed).range) \leq second((atom\ ed).range) \land \\
\quad \#(atom\ ed).in = 1 \land \#(atom\ ed).out = 1 \land \\
\quad (\forall\, t : Type \bullet t \in (\operatorname{ran}((atom\ ed).exit) \cap \operatorname{ran} ierror) \Rightarrow (ierror^{\sim} t) = anyexception) \land \\
\quad (\forall\, t : Type \bullet t \in \operatorname{ran}((atom\ ed).exit) \Rightarrow t \in inters) \land \\
\quad etype? \in nomsgserrors \land \\
\quad etype? \notin \operatorname{ran}((atom\ ele).exit) \land \\
\quad eflow? \notin getSeqflows\, proc \\
\Rightarrow ch \in \operatorname{ran} atomic \land & \text{[def of } ce \text{ and } etype?\text{]} \\
\quad (atom\ ch).type \in \operatorname{ran} task \cup \operatorname{ran} sloop \cup \operatorname{ran} miseq \cup \operatorname{ran} mipar \land \\
\quad first((atom\ ch).range) \leq second((atom\ ch).range) \land \\
\quad \#(atom\ ch).in = 1 \land \#(atom\ ch).out = 1 \land \\
\quad (\forall\, t : Type \bullet t \in (\operatorname{ran}((atom\ ch).exit) \cap \operatorname{ran} ierror) \Rightarrow (ierror^{\sim} t) = anyexception) \land \\
\quad (\forall\, t : Type \bullet t \in \operatorname{ran}((atom\ ch).exit) \Rightarrow t \in inters) \\
\Leftrightarrow ch \in \{ele : Element \mid FullTask\} & \text{[def of } FullTask\text{]} \\
\Rightarrow ch \in \{ele : Element \mid Activity\} & \text{[def of } Activity\text{]}
\end{array}
$$

We now consider if $ed \in \{ele : Element \mid FullSub\}$.

$$
\begin{array}{l}
ed \in \{ele : Element \mid FullSub\} \land \\
etype? \in nomsgserrors \land \\
eflow? \notin getSeqflows\, proc \\
\Leftrightarrow ed \in \operatorname{ran} compound \land & \text{[def of } FullSub\text{]} \\
\quad \#(atom\ ed).send + \#(atom\ ed).receive = 0 \land \\
\quad (atom\ ed).type \in \operatorname{ran} subprocess \cup \operatorname{ran} mipars \cup \operatorname{ran} miseqs \land
\end{array}
$$

$$
\begin{array}{l}
\quad content(ed) \in \{\, WFProcess \bullet proc \} \land \\
\quad \#(atom\ ed).in = 1 \land \#(atom\ ed).out = 1 \land \\
\quad (\forall\, e, f : content(ed) \bullet \\
\qquad \{e, f\} \subseteq \{End \mid (atom\ ele).type \in (\mathrm{ran}\ eerror \cap errorCodeTypes) \bullet ele\} \land \\
\qquad errorCode((atom\ e).type) = errorCode((atom\ f).type) \Rightarrow f = e) \land \\
\quad (\forall\, e : content(ed) \bullet \\
\qquad (atom\ e).type \in (\mathrm{ran}\ eerror \cap errorCodeTypes) \Rightarrow \\
\qquad\quad (\exists\, t : \mathrm{ran}((atom\ ed).exit) \bullet \\
\qquad\qquad t \in (\mathrm{ran}\ ierror \cap errorCodeTypes) \land \\
\qquad\qquad errorCode((atom\ e).type) = errorCode(t))) \land \\
\quad (\forall\, t : \mathrm{ran}((atom\ ed).exit) \bullet \\
\qquad (t \in (\mathrm{ran}\ ierror \cap errorCodeTypes) \Rightarrow \\
\qquad\quad (\exists\, e : content(ed) \bullet \\
\qquad\qquad (atom\ e).type \in (\mathrm{ran}\ eerror \cap errorCodeTypes) \land \\
\qquad\qquad errorCode((atom\ e).type) = errorCode(t)))) \land \\
\quad (\forall\, t : Type \bullet t \in \mathrm{ran}((atom\ ed).exit) \Rightarrow t \in inters) \land \\
\quad etype? \in nomsgserrors \land \\
\quad etype? \notin \mathrm{ran}((atom\ ele).exit) \land \\
\quad eflow? \notin getSeqflows\ proc \\
\Rightarrow ch \in \mathrm{ran}\ compound \land \quad [\text{def of } ce,\ etype? \notin \mathrm{ran}((atom\ ele).exit) \cup (\mathrm{ran}\ ierror \cap errorCodeTypes)] \\
\quad \#(atom\ ch).send + \#(atom\ ch).receive = 0 \land \\
\quad (atom\ ch).type \in \mathrm{ran}\ subprocess \cup \mathrm{ran}\ mipars \cup \mathrm{ran}\ miseqs \land \\
\quad content(ch) \in \{\, WFProcess \bullet proc \} \land \\
\quad \#(atom\ ch).in = 1 \land \#(atom\ ch).out = 1 \land \\
\quad (\forall\, e, f : content(ch) \bullet \\
\qquad \{e, f\} \subseteq \{End \mid (atom\ ele).type \in (\mathrm{ran}\ eerror \cap errorCodeTypes) \bullet ele\} \land \\
\qquad errorCode((atom\ e).type) = errorCode((atom\ f).type) \Rightarrow f = e) \land \\
\quad (\forall\, e : content(ch) \bullet \\
\qquad (atom\ e).type \in (\mathrm{ran}\ eerror \cap errorCodeTypes) \Rightarrow \\
\qquad\quad (\exists\, t : \mathrm{ran}((atom\ ch).exit) \bullet \\
\qquad\qquad t \in (\mathrm{ran}\ ierror \cap errorCodeTypes) \land \\
\qquad\qquad errorCode((atom\ e).type) = errorCode(t))) \land \\
\quad (\forall\, t : \mathrm{ran}((atom\ ch).exit) \bullet \\
\qquad (t \in (\mathrm{ran}\ ierror \cap errorCodeTypes) \Rightarrow \\
\qquad\quad (\exists\, e : content(ch) \bullet \\
\qquad\qquad (atom\ e).type \in (\mathrm{ran}\ eerror \cap errorCodeTypes) \land \\
\qquad\qquad errorCode((atom\ e).type) = errorCode(t)))) \land \\
\quad (\forall\, t : Type \bullet t \in \mathrm{ran}((atom\ ch).exit) \Rightarrow t \in \{\, inters \bullet (atom\ ele).type \,\}) \\
\Leftrightarrow ch \in \{ele : Element \mid FullSub\} \qquad [\text{def of } FullSub] \\
\Rightarrow ch \in \{ele : Element \mid Activity\} \qquad [\text{def of } Activity]
\end{array}
$$

We therefore obtain the following simplified precondition schema, labelled *PreAddNoRelatedErrorExceptionSub*.

PreAddNoRelatedErrorExceptionSub

$Activity$
$etype? : Type$
$eflow? : Seqflow$

$etype? \in nomsgserrors$
$eflow? \notin getSeqflows\{ele\}$

□

B.4 Precondition of *ChangeEndType*

pre *ChangeEndType*
$\Leftrightarrow [End;\ type? : Type \mid$
$\quad type? \in \text{ran}\ eerror \cap errorCodeTypes \land$
$\quad (atom\ ele).type = end \land$
$\quad ct(ele, type?) \in \{ele : Element \mid End\}]$

We now consider the membership $ct(ele, type?) \in \{ele : Element \mid End\}$.

$ele \in \{ele : Element \mid End\} \land$
$(atom\ ele).type = end \land$
$type? \in \text{ran}\ eerror \cap errorCodeTypes$
$\Leftrightarrow ele \in \text{ran}\ atomic \land$ [def of *End* and subset inclusion]
$\quad \#(atom\ ele).exit + \#(atom\ ele).send + \#(atom\ ele).receive = 0 \land$
$\quad first(atom\ ele).range = 0 \land$
$\quad second(atom\ ele).range = 0 \land$
$\quad (atom\ ele).type = end \land$
$\quad \#(atom\ ele).in = 1 \land \#(atom\ ele).out = 0 \land$
$\quad type? \in \text{ran}\ eerror \cap errorCodeTypes$
$\Rightarrow ct(ele, type?) \in \text{ran}\ atomic \land$ [def of ct]
$\quad \#(atom\ ct(ele, type?)).exit + \#(atom\ ct(ele, type?)).send + \#(atom\ ct(ele, type?)).receive = 0 \land$
$\quad first(atom\ ct(ele, type?)).range = 0 \land$
$\quad second(atom\ ct(ele, type?)).range = 0 \land$
$\quad (atom\ ct(ele, type?)).type = type? \land$
$\quad \#(atom\ ct(ele, type?)).in = 1 \land \#(atom\ ct(ele, type?)).out = 0 \land$
$\quad type? \in \text{ran}\ eerror \cap errorCodeTypes$
$\Rightarrow ct(ele, type?) \in \{ele : Element \mid End\}$ [$type? \in \text{ran}\ eerror$]

We therefore obtain the following simplified precondition schema, labelled *PreChangeEndType*.

PreChangeEndType

End
$type? : Type$

$type? \in \text{ran}\ eerror \cap errorCodeTypes$
$(atom\ ele).type = end$

□

B.5 Precondition of *AddMgeEvent*

$$
\begin{array}{l}
\text{pre } \mathit{AddMgeEvent} \\
\Leftrightarrow [\mathit{Start};\ \mathit{msg}? : \mathit{Mgeflow} \mid \quad [\text{def of } \mathit{AddMgeEvent}, \text{ schema quantification and one-point rule x 3}] \\
\quad \textbf{let } \mathit{ce} == \mathit{ct}(\mathit{ele}, \mathit{smessage}(\mathit{message}(\mathit{msg}?))) \bullet \\
\quad (\mathit{atom}\ \mathit{ele}).\mathit{type} = \mathit{smessage}(\mathit{nomessage}) \land \\
\quad \mathit{ce} \in \{\mathit{ele} : \mathit{Element} \mid \mathit{Start}\}] \lor \\
\quad [\mathit{Inter};\ \mathit{msg}? : \mathit{Mgeflow} \mid \\
\quad \textbf{let } \mathit{ce} == \mathit{ct}(\mathit{ele}, \mathit{imessage}(\mathit{message}(\mathit{msg}?))) \bullet \\
\quad (\mathit{atom}\ \mathit{ele}).\mathit{type} = \mathit{imessage}(\mathit{nomessage}) \land \\
\quad \mathit{ce} \in \{\mathit{ele} : \mathit{Element} \mid \mathit{Inter}\}] \lor \\
\quad [\mathit{End};\ \mathit{msg}? : \mathit{Mgeflow} \mid \\
\quad \textbf{let } \mathit{ce} == \mathit{ct}(\mathit{ele}, \mathit{emessage}(\mathit{message}(\mathit{msg}?))) \bullet \\
\quad (\mathit{atom}\ \mathit{ele}).\mathit{type} = \mathit{emessage}(\mathit{nomessage}) \land \\
\quad \mathit{ce} \in \{\mathit{ele} : \mathit{Element} \mid \mathit{End}\}]
\end{array}
$$

We consider the first schema disjunct and the constraint $\mathit{ce} \in \{\mathit{ele} : \mathit{Element} \mid \mathit{Start}\}$ by first expanding the before state *Start*.

$$
\begin{array}{l}
\mathit{ele} \in \operatorname{ran} \mathit{atomic} \\
\#(\mathit{atom}\ \mathit{ele}).\mathit{exit} + \#(\mathit{atom}\ \mathit{ele}).\mathit{send} + \#(\mathit{atom}\ \mathit{ele}).\mathit{receive} = 0 \land \\
\mathit{first}(\mathit{atom}\ \mathit{ele}).\mathit{range} = 0 \land \\
\mathit{second}(\mathit{atom}\ \mathit{ele}).\mathit{range} = 0 \land \\
(\mathit{atom}\ \mathit{ele}).\mathit{type} \in \{\mathit{start}\} \cup \operatorname{ran} \mathit{stime} \cup \operatorname{ran} \mathit{smessage} \cup \operatorname{ran} \mathit{slink} \land \\
\#(\mathit{atom}\ \mathit{ele}).\mathit{in} = 0 \land \#(\mathit{atom}\ \mathit{ele}).\mathit{out} = 1 \land \\
(\mathit{atom}\ \mathit{ele}).\mathit{type} = \mathit{smessage}(\mathit{nomessage}) \\
\Rightarrow \qquad\qquad [\mathit{ct} \text{ only modifies } \mathit{type} \text{ component and } \mathit{smessage}(\mathit{message}(\mathit{msg}?)) \in \operatorname{ran} \mathit{smessage}] \\
\quad \mathit{ce} \in \operatorname{ran} \mathit{atomic} \land \\
\quad \#(\mathit{atom}\ \mathit{ce}).\mathit{exit} + \#(\mathit{atom}\ \mathit{ce}).\mathit{send} + \#(\mathit{atom}\ \mathit{ce}).\mathit{receive} = 0 \land \\
\quad \mathit{first}(\mathit{atom}\ \mathit{ce}).\mathit{range} = 0 \land \\
\quad \mathit{second}(\mathit{atom}\ \mathit{ce}).\mathit{range} = 0 \land \\
\quad (\mathit{atom}\ \mathit{ce}).\mathit{type} \in \{\mathit{start}\} \cup \operatorname{ran} \mathit{stime} \cup \operatorname{ran} \mathit{smessage} \cup \operatorname{ran} \mathit{slink} \land \\
\quad \#(\mathit{atom}\ \mathit{ce}).\mathit{in} = 0 \land \#(\mathit{atom}\ \mathit{ele}).\mathit{out} = 1 \\
\Leftrightarrow \mathit{ce} \in \{\mathit{ele} : \mathit{Element} \mid \mathit{Start}\} \qquad\qquad [\text{def of } \mathit{Start}]
\end{array}
$$

We have shown $\mathit{ce} \in \{\mathit{ele} : \mathit{Element} \mid \mathit{Start}\}$ follows from the constraints on the before state of the first schema disjunct. We therefore obtain a simplified precondition schema pre *AddSMgeEvent*, labelled *PreAddSMgeEvent*.

$$\mathit{PreAddSMgeEvent} \mathrel{\widehat{=}} [\mathit{Start};\ \mathit{msg}? : \mathit{Mgeflow} \mid (\mathit{atom}\ \mathit{ele}).\mathit{type} = \mathit{smessage}(\mathit{nomessage})]$$

We now consider the second schema disjunct and the constraint $\mathit{ce} \in \{\mathit{ele} : \mathit{Element} \mid \mathit{Inter}\}$ by first expanding the before state *Inter*.

$$
\begin{array}{l}
\mathit{ele} \in \operatorname{ran} \mathit{atomic} \\
\#(\mathit{atom}\ \mathit{ele}).\mathit{exit} + \#(\mathit{atom}\ \mathit{ele}).\mathit{send} + \#(\mathit{atom}\ \mathit{ele}).\mathit{receive} = 0 \land \\
\mathit{first}(\mathit{atom}\ \mathit{ele}).\mathit{range} = 0 \land \\
\mathit{second}(\mathit{atom}\ \mathit{ele}).\mathit{range} = 0 \land \\
(\mathit{atom}\ \mathit{ele}).\mathit{type} \in \operatorname{ran} \mathit{itime} \cup \operatorname{ran} \mathit{imessage} \cup \operatorname{ran} \mathit{ierror} \cup \operatorname{ran} \mathit{irule} \land
\end{array}
$$

$$
\begin{array}{l}
\#(atom\ ele).in = 1 \land \#(atom\ ele).out = 1 \land \\
(atom\ ele).type = imessage(nomessage) \\
\Rightarrow \qquad [ct \text{ only modifies } type \text{ component and } imessage(message(msg?)) \in \operatorname{ran} imessage] \\
\quad ce \in \operatorname{ran} atomic \land \\
\quad \#(atom\ ce).exit + \#(atom\ ce).send + \#(atom\ ce).receive = 0 \land \\
\quad first(atom\ ce).range = 0 \land \\
\quad second(atom\ ce).range = 0 \land \\
\quad (atom\ ele).type \in \operatorname{ran} itime \cup \operatorname{ran} imessage \cup \operatorname{ran} ierror \cup \operatorname{ran} irule \land \\
\quad \#(atom\ ele).in = 1 \land \#(atom\ ele).out = 1 \\
\Leftrightarrow ce \in \{ele : Element \mid Inter\} \qquad [\text{def of } Inter]
\end{array}
$$

We have shown $ce \in \{ele : Element \mid Inter\}$ follows from the constraints on the before state of the second schema disjunct. We therefore obtain a simplified precondition schema pre *AddIMgeEvent*, labelled *PreAddIMgeEvent*.

$$PreAddIMgeEvent \mathrel{\widehat=} [Inter;\ msg? : Mgeflow \mid (atom\ ele).type = imessage(nomessage)]$$

We now consider the third schema disjunct and the constraint $ce \in \{ele : Element \mid End\}$ by first expanding the before state *End*.

$$
\begin{array}{l}
ele \in \operatorname{ran} atomic \\
\#(atom\ ele).exit + \#(atom\ ele).send + \#(atom\ ele).receive = 0 \land \\
first(atom\ ele).range = 0 \land \\
second(atom\ ele).range = 0 \land \\
(atom\ ele).type \in \{\ end, abort\ \} \cup \operatorname{ran} emessage \cup \operatorname{ran} eerror \land \\
\#(atom\ ele).in = 1 \land \#(atom\ ele).out = 0 \land \\
(atom\ ele).type = emessage(nomessage) \\
\Rightarrow \qquad [ct \text{ only modifies } type \text{ component and } emessage(message(msg?)) \in \operatorname{ran} emessage] \\
\quad ce \in \operatorname{ran} atomic \land \\
\quad \#(atom\ ce).exit + \#(atom\ ce).send + \#(atom\ ce).receive = 0 \land \\
\quad first(atom\ ce).range = 0 \land \\
\quad second(atom\ ce).range = 0 \land \\
\quad (atom\ ele).type \in \{\ end, abort\ \} \cup \operatorname{ran} emessage \cup \operatorname{ran} eerror \land \\
\quad \#(atom\ ele).in = 1 \land \#(atom\ ele).out = 0 \\
\Leftrightarrow ce \in \{ele : Element \mid End\} \qquad [\text{def of } End]
\end{array}
$$

We have shown $ce \in \{ele : Element \mid End\}$ follows from the constraints on the before state of the third schema disjunct. We therefore obtain a simplified precondition schema pre *AddEMgeEvent*, labelled *PreAddEMgeEvent*.

$$PreAddEMgeEvent \mathrel{\widehat=} [End;\ msg? : Mgeflow \mid (atom\ ele).type = emessage(nomessage)]$$

We therefore obtain the final simplified precondition schema pre *AddMgeEvent*, labelled *PreAddMgeEvent*.

$$PreAddMgeEvent \mathrel{\widehat=} PreAddSMgeEvent \lor PreAddIMgeEvent \lor PreAddEMgeEvent$$

□

B.6 Precondition of *AddSendMgeFlowTask*

pre *AddSendMgeFlowTask*

$\Leftrightarrow \exists\, FullTask' \bullet$ [def of pre and *AddSendMgeFlowTask*]
$[\Delta FullTask;\ msg? : Mgeflow \mid$
$msg? \notin getMsg(atom\ ele) \land$
$ele' = cm(ele, \{msg?\}, \emptyset, (atom\ ele).exit)]$
$\Leftrightarrow [FullTask;\ msg? : Mgeflow \mid$ [schema quantification and one-point rule]
$\textbf{let}\ ce == cm(ele, \{msg?\}, \emptyset, (atom\ ele).exit) \bullet$
$msg? \notin getMsg(atom\ ele) \land$
$ce \in \{ele : Element \mid FullTask\}]$

We consider the constraint $ce \in \{ele : Element \mid FullTask\}$ by first expanding the constraint of the before state *FullTask*

$ele \in \operatorname{ran} atomic \land$
$(atom\ ele).type \in \operatorname{ran} task \cup \operatorname{ran} miseq \cup \operatorname{ran} mipar \land$
$first((atom\ ele).range) \leq second((atom\ ele).range) \land$
$(\forall\, t : Type \bullet t \in (\operatorname{ran}((atom\ ele).exit) \cap \operatorname{ran} ierror) \Rightarrow (ierror^{\sim} t) = anyexception) \land$
$(\forall\, t : Type \bullet t \in \operatorname{ran}((atom\ ele).exit) \Rightarrow t \in inters) \land$
$ce = cm(ele, \{msg?\}, \emptyset, (atom\ ele).exit)$
$\Rightarrow$ [only component *send* of *ele* is modified, and $(atom\ ce).exit = (atom\ ele).exit$]
$ce \in \operatorname{ran} atomic \land$
$(atom\ ce).type \in \operatorname{ran} task \cup \operatorname{ran} miseq \cup \operatorname{ran} mipar \land$
$first((atom\ ce).range) \leq second((atom\ ce).range) \land$
$(\forall\, t : Type \bullet t \in (\operatorname{ran}((atom\ ce).exit) \cap \operatorname{ran} ierror) \Rightarrow (ierror^{\sim} t) = anyexception) \land$
$(\forall\, t : Type \bullet t \in \operatorname{ran}((atom\ ce).exit) \Rightarrow t \in inters)$
$\Leftrightarrow ce \in \{ele : Element \mid FullTask\}$ [def of *FullTask*]

We have shown $ce \in \{ele : Element \mid FullTask\}$ follows from the constraints on the before state of the precondition schema. We therefore obtain the final simplified precondition schema pre *AddSendMgeFlowTask*, labelled *PreAddSendMgeFlowTask*.

$PreAddSendMgeFlowTask \mathrel{\widehat=} [FullTask;\ msg? : Mgeflow \mid msg? \notin getMsg(atom\ ele)]$

□

B.7 Precondition of *AddReceiveMgeFlowTask*

pre *AddReceiveMgeFlowTask*
$\Leftrightarrow \exists\, FullTask' \bullet$ [def of pre and *AddReceiveMgeFlowTask*]
$[\Delta FullTask;\ msg? : Mgeflow \mid$
$msg? \notin getMsg(atom\ ele) \land$
$ele' = cm(ele, \emptyset, \{msg?\}, (atom\ ele).exit)]$
$\Leftrightarrow [FullTask;\ msg? : Mgeflow \mid$ [schema quantification and one-point rule]
$\textbf{let}\ ce == cm(ele, \emptyset, \{msg?\}, (atom\ ele).exit) \bullet$
$msg? \notin getMsg(atom\ ele) \land$
$ce \in \{ele : Element \mid FullTask\}]$

We consider the constraint $ce \in \{ele : Element \mid FullTask\}$ by expanding the constraint of the before state *FullTask*

$ele \in \operatorname{ran} atomic \land$

$(atom\ ele).type \in \operatorname{ran} task \cup \operatorname{ran} miseq \cup \operatorname{ran} mipar \land$
$first((atom\ ele).range) \leq second((atom\ ele).range) \land$
$(\forall t : Type \bullet t \in (\operatorname{ran}((atom\ ele).exit) \cap \operatorname{ran} ierror) \Rightarrow (ierror^{\sim} t) = anyexception) \land$
$(\forall t : Type \bullet t \in \operatorname{ran}((atom\ ele).exit) \Rightarrow t \in inters) \land$
$ce = cm(ele, \{msg?\}, \emptyset, (atom\ ele).exit)$
$\Rightarrow$ [only component *receive* of *ele* is modified and $(atom\ ce).exit = (atom\ ele).exit$]
$ce \in \operatorname{ran} atomic \land$
$(atom\ ce).type \in \operatorname{ran} task \cup \operatorname{ran} miseq \cup \operatorname{ran} mipar \land$
$first((atom\ ce).range) \leq second((atom\ ce).range) \land$
$(\forall t : Type \bullet t \in (\operatorname{ran}((atom\ ce).exit) \cap \operatorname{ran} ierror) \Rightarrow (ierror^{\sim} t) = anyexception) \land$
$(\forall t : Type \bullet t \in \operatorname{ran}((atom\ ce).exit) \Rightarrow t \in inters)$
$\Leftrightarrow ce \in \{ele : Element \mid FullTask\}$ [def of *FullTask*]

We have shown $ce \in \{ele : Element \mid FullTask\}$ follows from the constraints on the before state of the precondition schema. We therefore obtain the final simplified precondition schema pre *AddReceiveMgeFlowTask*, labelled *PreAddReceiveMgeFlowTask*.

$PreAddReceiveMgeFlowTask \mathrel{\widehat=} [FullTask;\ msg? : Mgeflow \mid msg? \notin getMsg(atom\ ele)]$

□

B.8 Precondition of *AddExceptionMgeFlow*

B.8.1 Preliminaries

We first show that replacing a unique maplet in a finite partial function guarantees a finite partial function.

Lemma B.4. $\forall m, n : Y;\ f : X \nrightarrow Y \bullet \#(f \rhd \{m\}) = 1 \Rightarrow rrange(f, m, n) \in X \nrightarrow Y$

Proof.

$\#(f \rhd \{m\}) = 1 \land$
$\Rightarrow \#(\operatorname{dom}(f \rhd \{m\})) = 1 \land$ [property of $\rhd$]
$\Leftrightarrow \#\{d : (\operatorname{dom}(f \rhd \{m\})) \bullet d \mapsto n\} = 1 \land$ [set-compre and def of $\mapsto$]
$\Rightarrow \{d : (\operatorname{dom}(f \rhd \{m\}) \bullet d \mapsto n\} \in X \nrightarrow Y$ [def of $\nrightarrow$]
$\Rightarrow f \oplus \{d : (\operatorname{dom}(f \rhd \{m\})) \bullet d \mapsto n\} \in X \nrightarrow Y$ [$f \in X \nrightarrow Y$ and property of $\oplus$]
$\Leftrightarrow rrange(f, m, n) \in X \nrightarrow Y$ [def of *rrange*]

□

B.8.2 Simplification

pre *AddExceptionMgeFlow*
$\Leftrightarrow \exists Activity' \bullet$ [def of pre and *AddReceiveMgeFlowTask*]
$[\Delta Activity;\ msg? : Mgeflow \mid$
$msg? \notin getMsg(atom\ ele) \land$
$\#(((atom\ ele).exit) \rhd \{imessage(nomessage)\}) = 1 \land$
$ele' = cm(ele, \emptyset, \emptyset, rrange((atom\ ele).exit, imessage(nomessage), imessage(message(msg?))))]$
$\Leftrightarrow [Activity;\ msg? : Mgeflow \mid$ [schema quantification and one-point rule]
let $ce == cm(ele, \emptyset, \emptyset, rrange((atom\ ele).exit, imessage(nomessage), imessage(message(msg?)))) \bullet$

$$
\begin{array}{l}
\quad msg? \notin getMsg(atom\ ele) \land \\
\quad \#(((atom\ ele).exit) \rhd \{imessage(nomessage)\}) = 1 \land \\
\quad ce \in \{ele : Element \mid Activity\}]
\end{array}
$$

We consider the constraint $ce \in \{ele : Element \mid Activity\}$. Since $Activity \mathrel{\widehat=} FullTask \lor FullSub$ we first consider if the before state satisfies *FullTask*.

$$
\begin{array}{l}
\quad ele \in \operatorname{ran} atomic \land \\
\quad (atom\ ele).type \in \operatorname{ran} task \cup \operatorname{ran} miseq \cup \operatorname{ran} mipar \land \\
\quad first((atom\ ele).range) \leq second((atom\ ele).range) \land \\
\quad \#(atom\ ele).in = 1 \land \#(atom\ ele).out = 1 \land \\
\quad (\forall t : Type \bullet t \in (\operatorname{ran}((atom\ ele).exit) \cap \operatorname{ran} ierror) \Rightarrow (ierror^{\sim} t) = anyexception) \land \\
\quad (\forall t : Type \bullet t \in \operatorname{ran}((atom\ ele).exit) \Rightarrow t \in inters) \land msg? \notin getMsg(atom\ ele) \land \\
\quad \#(((atom\ ele).exit) \rhd \{imessage(nomessage)\}) = 1 \land \\
\Rightarrow \qquad \text{[def of } cm\text{, } imessage(message(msg?) \in inters \text{ and Lemma B.4]} \\
\quad ce \in \operatorname{ran} atomic \land \\
\quad (atom\ ce).type \in \operatorname{ran} task \cup \operatorname{ran} miseq \cup \operatorname{ran} mipar \land \\
\quad first((atom\ ce).range) \leq second((atom\ ce).range) \land \\
\quad \#(atom\ ce).in = 1 \land \#(atom\ ce).out = 1 \land \\
\quad (\forall t : Type \bullet t \in (\operatorname{ran}((atom\ ce).exit) \cap \operatorname{ran} ierror) \Rightarrow (ierror^{\sim} t) = anyexception) \land \\
\quad (\forall t : Type \bullet t \in \operatorname{ran}((atom\ ce).exit) \Rightarrow t \in inters) \\
\Leftrightarrow ce \in \{ele : Element \mid FullTask\} \qquad \text{[def of } FullTask\text{]} \\
\Rightarrow ce \in \{ele : Element \mid Activity\} \qquad \text{[def of } Activity\text{]}
\end{array}
$$

We now consider if the before state satisfies *FullSub*.

$$
\begin{array}{l}
(ele \in \operatorname{ran} compound \land \qquad \text{[def of } FullSub\text{]} \\
\#(atom\ ele).send = \#(atom\ change).receive = 0 \land \\
(atom\ ele).type \in \operatorname{ran} subprocess \cup \operatorname{ran} mipars \cup \operatorname{ran} miseqs \land \\
content(ele) \neq \emptyset \land \\
content(ele) \in \{WFProcess \bullet proc\} \land \\
\#(atom\ ele).in = 1 \land \#(atom\ ele).out = 0) \land \\
((\forall e, f : content(ele) \bullet \\
\quad \{e, f\} \subseteq \{End \mid (atom\ ele).type \in (\operatorname{ran} eerror \cap errorCodeTypes) \bullet ele\} \land \\
\quad errorCode((atom\ e).type) = errorCode((atom\ f).type) \Rightarrow f = e) \land \\
(\forall e : content(ele) \bullet \\
\quad (atom\ e).type \in (\operatorname{ran} eerror \cap errorCodeTypes) \Rightarrow \\
\qquad (\exists_1 t : Type \bullet \\
\qquad\quad t \in \operatorname{ran}((atom\ ele).exit) \land \\
\qquad\quad t \in (\operatorname{ran} ierror \cap errorCodeTypes) \land \\
\qquad\quad errorCode((atom\ e).type) = errorCode(t))) \land \\
(\forall t : Type \bullet t \in \operatorname{ran}((atom\ ele).exit) \Rightarrow t \in inters)) \land msg? \notin getMsg(atom\ ele) \land \\
\#(((atom\ ele).exit) \rhd \{imessage(nomessage)\}) = 1 \land \\
\Rightarrow \qquad \text{[def of } cm\text{, } imessage(message(msg?) \in inters \text{ and Lemma B.4]} \\
\quad (ce \in \operatorname{ran} compound \land \\
\quad \#(atom\ ce).send = \#(atom\ ce).receive = 0 \land \\
\quad (atom\ ce).type \in \operatorname{ran} subprocess \cup \operatorname{ran} mipars \cup \operatorname{ran} miseqs \land
\end{array}
$$

$$
\begin{array}{l}
\quad content(ce) \neq \emptyset \land \\
\quad content(ce) \in \{ WFProcess \bullet proc\} \land \\
\quad \#(atom\ ce).in = 1 \land \#(atom\ ce).out = 0) \land \\
\quad ((\forall e, f : content(ce) \bullet \\
\quad\quad \{e, f\} \subseteq \{End \mid (atom\ ce).type \in (\mathrm{ran}\ eerror \cap errorCodeTypes) \bullet ele\} \land \\
\quad\quad errorCode((atom\ e).type) = errorCode((atom\ f).type) \Rightarrow f = e) \land \\
\quad (\forall e : content(ce) \bullet \\
\quad\quad (atom\ e).type \in (\mathrm{ran}\ eerror \cap errorCodeTypes) \Rightarrow \\
\quad\quad\quad (\exists_1 t : Type \bullet \\
\quad\quad\quad\quad t \in \mathrm{ran}((atom\ ele).exit) \land \\
\quad\quad\quad\quad t \in (\mathrm{ran}\ ierror \cap errorCodeTypes) \land \\
\quad\quad\quad\quad errorCode((atom\ e).type) = errorCode(t))) \land \\
\quad (\forall t : Type \bullet t \in \mathrm{ran}((atom\ ce).exit) \Rightarrow t \in inters)) \\
\Leftrightarrow ce \in \{ele : Element \mid FullSub\} \qquad [\text{def of } FullSub] \\
\Rightarrow ce \in \{ele : Element \mid Activity\} \qquad [\text{def of } Activity]
\end{array}
$$

We have shown $ce \in \{ele : Element \mid Activity\}$ follows from the constraints on the before state of the precondition schema. We therefore obtain the final simplified precondition schema pre *AddExceptionMgeFlow*, labelled *PreAddExceptionMgeFlow*.

PreAddExceptionMgeFlow

$$
\begin{array}{l}
Activity;\ msg? : Mgeflow \\
\hline
msg? \notin getMsg(atom\ ele) \\
\#(((atom\ ele).exit) \rhd \{imessage(nomessage)\}) = 1
\end{array}
$$

□

B.9 Precondition of *SeqComp*

We expand pre *SeqComp* and apply the one-point rule to arrive at the following schema.

$$
\begin{array}{l}
[Pool;\ new?, end? : Element;\ from? : Seqflow \mid \\
\quad \textbf{let}\ ed == (ends\ proc)\ from? \bullet \\
\quad\quad \textbf{let}\ md == modify(proc, \{new?, end?\}, \{ed\}) \bullet \\
\quad (outs\ new? \cup (getSeqflows\ (content\ new?))) \cap getSeqflows\ proc = \emptyset \land \\
\quad from? \in (atom\ new?).in \land \\
\quad (atom\ new?).in \subseteq \mathrm{dom}(ends\ proc) \land \\
\quad new? \in \{ele : Element \mid OneInOutObject\} \land \\
\quad end? \in \{ele : Element \mid InitialEnd\} \land \\
\quad (atom\ end?).in = (atom\ new?).out \land \\
\quad md \in \{proc : Process \mid Pool\}]
\end{array}
$$

We consider the conjunct $md \in \{proc : Process \mid Pool\}$. We show this follows directly from constraints on the before state *Pool* and the input components by induction on *proc*. We first expand the definition of $md = modify(proc, \{new?, end?\}, \{ed\})$:

$$
md = \begin{cases}
((proc \cup \{new?, end?\}) \setminus \{ed\}) & \text{if } ed \in proc \\
((proc \setminus cont(proc, \{ed\})) \cup & \text{otherwise} \\
\quad \{s : cont(proc, \{ed\}) \bullet rep(s, modify(content(s), \{new?, end?\}, \{ed\}))\})
\end{cases}
$$

For the case $ed \in proc$, we show $md \in \{proc : Process \mid Pool\}$ follows directly from constraints on the before state *Pool* and the input components, where $md == ((proc \cup \{new?, end?\}) \setminus \{ed\})$. We expand

this membership.

$$
\begin{array}{l}
(\exists\, e, f : md \bullet e \in \{Start \bullet ele\} \land f \in \{End \bullet ele\}) \land \\
(\neg\, (\exists\, e : md \bullet e \in \{Inter \mid (atom\ ele).type \in \operatorname{ran} ierror \bullet ele\})) \land \\
(\forall\, e : \{g : Element \mid g \in_{\mathrm{p}} md\} \bullet e \in \{FlowObject \bullet ele\}) \land \\
md \in processSet \land \\
md \in noOverLap \land \\
(\forall f : \{g : Element \mid g \in_{\mathrm{p}} md\} \bullet f \in eventgate \Rightarrow \\
\quad (\forall\, e : Element \bullet (f, e) \in edge(direct(md, e)) \Rightarrow e \in sendelement))
\end{array}
$$

We consider the first and second conjuncts.

$$
\begin{array}{ll}
new? \notin \{Inter \mid (atom\ ele).type \in \operatorname{ran} ierror \bullet ele\} \land & [\text{def of } new?,\ end? \text{ and } ed] \\
\{ed, end?\} \subseteq \{End \bullet ele\} & \\
\Rightarrow (\exists\, e, f : md \bullet e \in \{Start \bullet ele\} \land f \in \{End \bullet ele\}) \land & [proc \in \{GenProc \bullet proc\}] \\
\quad (\neg\, (\exists\, e : md \bullet e \in \{Inter \mid (atom\ ele).type \in \operatorname{ran} ierror \bullet ele\})) &
\end{array}
$$

We now consider the third conjunct.

$$
\begin{array}{ll}
end? \in \{ele : Element \mid FlowObject\} \land & [\text{def of } new? \text{ and } end?] \\
new? \in \{ele : Element \mid FlowObject\} & \\
\Rightarrow & [\text{def of } GenPool, \text{ Lemma B.1 and def of } OneInOutObject \Rightarrow FlowObject] \\
\quad (\forall\, e : \{g : Element \mid g \in_{\mathrm{p}} md\} \bullet e \in \{FlowObject \bullet ele\}) &
\end{array}
$$

We now consider the fourth conjunct.

$$
\begin{array}{ll}
(atom\ new?).in = \{from\} \land & [\text{def of } new?,\ end?,\ ed \text{ and } \#(atom\ new?).in = \#(atom\ new?).out = 1] \\
(atom\ new?).out = (atom\ end?).in \land & \\
(atom\ ed).exit = (atom\ new?).exit = \emptyset \land & \\
(outs(new?) \cup (getSeqflows(content(new?)))) \cap getSeqflows(proc) = \emptyset \land & \\
getMsgs\{new?, end?\} = \emptyset \land & \\
\Rightarrow (\forall\, e : \{g : Element \mid g \in_{\mathrm{p}} md\} \bullet & [\text{def of } md,\ proc \in noOverLap] \\
\quad \bigcup\{k : md \bullet (atom\ k).in \cup outs(k)\} \cap \bigcup\{k : content\ e \bullet (atom\ k).in \cup outs(k)\} = \emptyset \land & \\
\quad (\forall f : \{g : Element \mid g \in_{\mathrm{p}} md\} \bullet e \neq f \Rightarrow & \\
\qquad (atom\ e).in \cap (atom\ f).in = \emptyset \land & \\
\qquad outs(e) \cap outs(f) = \emptyset \land & \\
\qquad \bigcup\{k : content\ e \bullet (atom\ k).in \cup outs(k)\} \cap & \\
\qquad \bigcup\{k : content\ f \bullet (atom\ k).in \cup outs(k)\} = \emptyset \land & \\
\qquad getMsg\,(atom\ e) \cap getMsg\,(atom\ f) = \emptyset)) & \\
\Leftrightarrow md \in noOverLap & [\text{def of } noOverLap]
\end{array}
$$

We now consider the fifth conjunct.

$$
\begin{array}{ll}
 & [\text{def of } new?,\ end? \text{ and } \#(atom\ new?).in = 1,\ \#(atom\ new?).out = 1] \\
(atom\ new?).in = \{from\} \land & \\
(atom\ new?).out = (atom\ end?).in \land & \\
(atom\ end?).exit = (atom\ new?).exit = \emptyset \land & \\
(new?, end?) \in (edge\ md) \land & \\
ed \in \{End \bullet ele\} \land &
\end{array}
$$

$$
\begin{array}{ll}
\mathrm{dom}((edge\ proc) \rhd \{new?\}) = \mathrm{dom}((edge\ md) \rhd \{ed\}) & \\
\Rightarrow (\forall\, e : md \bullet & [proc \in processSet] \\
\quad (e \notin \{\, End \bullet ele \,\} \Rightarrow (\exists f : md \bullet f \in \{\, End \bullet ele \,\} \land (e, f) \in (edge\ md)^+)) \land & \\
\quad (e \notin \{\, Start \bullet ele \,\} \Rightarrow (\exists f : md \bullet f \in \{\, Start \bullet ele \,\} \land (f, e) \in (edge\ md)^+))) \land & \\
\quad (\forall\, e : md \bullet & \\
\quad (\forall\, s : outs(e) \bullet \exists f : md \bullet s \in (atom\ f).in) \land & \\
\quad (\forall\, s : (atom\ e).in \bullet \exists f : md \bullet s \in outs(f))) \land & \\
\Leftrightarrow md \in processSet & [\text{def of } processSet]
\end{array}
$$

We now consider the sixth conjunct.

$$
\begin{array}{ll}
(edge\ md)(\!|\ \{new?\}\ |\!) = \{end?\} \land & [\text{def of } new? \text{ and } end?] \\
(\mathrm{dom}((edge\ proc) \rhd \{ed\}) \cup \{new?\}) \cap eventgate = \emptyset \land & \\
\Rightarrow (\forall f : \{g : Element \mid g \in_{\mathrm{p}} md\} \bullet f \in eventgate \Rightarrow & [proc \in \{proc : Process \mid hasExgates\}] \\
\quad (\forall\, e : Element \bullet (f, e) \in edge(direct(md, e)) \Rightarrow e \in sendelement)) & \\
\Rightarrow md \in \{proc : Process \mid hasExgates\} & [\text{def of } hasExgates]
\end{array}
$$

We therefore conclude that $md \in \{proc : Process \mid Pool\}$ follows directly from constraints on the before state *Pool* and the input components.
For case $ed \notin proc$, we consider the following inductive hypothesis

$$
\begin{array}{l}
\{s : cont(proc, \{ed\}) \bullet modify(content(s), \{new?, end\}, \{ed\})\} \subseteq \{proc : Process \mid Pool\}) \Rightarrow \\
((proc \setminus cont(proc, \{ed\})) \cup \\
\quad \{s : cont(proc, \{ed\}) \bullet rep(s, modify(content(s), \{new?, end?\}, \{ed\}))\}) \in \{proc : Process \mid Pool\})
\end{array}
$$

According to Lemma B.2, it is sufficient to deduce the following set of facts (conjuncts) about the precondition schema.

$$
\begin{array}{ll}
cont(proc, \{ed\}) \subseteq \mathrm{ran}\ compound & \text{(B.1)} \\
\#cont(proc, \{ed\}) = 1 & \text{(B.2)} \\
\bigcup \{s : cont(proc, \{ed\}) \bullet getSeqflows(modify(content(s), \{new?, end?\}, \{ed\}))\} \cap & \text{(B.3)} \\
\quad getSeqflows(proc \setminus \{s, e : Element \mid s \in cont(proc, \{ed\}) \land e \in_{\mathrm{e}} s \bullet e\}) = \emptyset & \\
\bigcup \{s : cont(proc, \{ed\}) \bullet getMsgs(modify(content(s), \{new?, end?\}, \{ed\}))\} \cap & \text{(B.4)} \\
\quad getMsgs(proc \setminus \{s, e : Element \mid s \in cont(proc, \{ed\}) \land e \in_{\mathrm{e}} s \bullet e\}) = \emptyset & \\
\{s, e : Element \mid s \in cont(proc, \{ed\}) \land & \text{(B.5)} \\
\quad e \in modify(content(s), \{new?, end?\}, \{ed\}) \land (atom\ e).type \in \mathrm{ran}\ eerror \bullet e\} = & \\
\quad \{s, e : Element \mid s \in cont(proc, \{ed\}) \land e \in content(s) \land (atom\ e).type \in \mathrm{ran}\ eerror \bullet e\} &
\end{array}
$$

We consider each of the conjuncts individually: Conjunct B.1 follows from the definition of *cont* and schema definition *FullSub*. Conjunct B.2 follows from the membership $proc \in noOverLap$. Conjunct B.3 follows from the following constraints:

$$
\begin{array}{l}
getIns\{ed\} = getIns\{new?\} \\
getOuts\{new?\} = getIns\{end?\} \\
getSeqflows\{end?\} \cap getSeqflows\ proc = \emptyset \\
(outs\ new? \cup (getSeqflows\ (content\ new?))) \cap getSeqflows\ proc = \emptyset \\
proc \in noOverLap
\end{array}
$$

Conjunct B.4 follows from $getMsgs\{new?, end?\} = \emptyset$ and $proc \in noOverLap$, and Conjunct B.5 follows from $\{(atom\ ed).type, (atom\ end?).type, (atom\ new?).type\} \cap \mathrm{ran}\ eerror = \emptyset$. We conclude that

$md \in \{proc : Process \mid Pool\}$ follows directly from constraints on the before state *Pool* and the input components. We obtain the final simplified precondition schema, labelled *PreSeqComp*.

PreSeqComp

$Pool$
$CommonConstraints$
$end? : Element$

$new? \in \{ele : Element \mid OneInOutObject\}$
$end? \in \{ele : Element \mid InitialEnd\}$
$(atom\ end?).in = (atom\ new?).out$

□

B.10 Precondition of *Split*

We expand pre *Split* and apply the one-point rule to arrive at the following schema.

$$
\begin{array}{l}
[Pool;\ new? : Element;\ from? : Seqflow;\ outs? : \mathbb{F}_1\ Element \mid \\
\quad \textbf{let}\ ed == (ends\ proc)\ from? \bullet \\
\quad\quad \textbf{let}\ md == modify(proc, \{new?\} \cup outs?, \{ed\}) \bullet \\
\quad new? \in \{NonEvSplit \bullet ele\} \land \\
\quad outs? \in uniqueEnds \land \\
\quad getIns(outs?) \cap getSeqflows\ proc = \emptyset \land \\
\quad getIns(outs?) = (atom\ new?).out \land \\
\quad md \in \{proc : Process \mid Pool\}]
\end{array}
$$

We consider the conjunct $md \in \{proc : Process \mid Pool\}$. We first expand the definition of md:

$$
md = \begin{cases}
(proc \cup \{new?\} \cup outs?) \setminus \{ed\} & \text{if } ed \in proc \\
((proc \setminus cont(proc, \{ed\})) \cup & \text{otherwise} \\
\quad \{s : cont(proc, \{ed\}) \bullet rep(s, modify(content(s), \{new?\} \cup outs?, \{ed\}))\}) &
\end{cases}
$$

For the case $ed \in proc$, we show $md \in \{proc : Process \mid Pool\}$ follows directly from constraints on the before state *Pool* and the input components, where $md == (proc \cup \{new?\} \cup outs?) \setminus \{ed\}$. We expand this membership.

$$
\begin{array}{l}
(\exists\, e, f : md \bullet e \in \{Start \bullet ele\} \land f \in \{End \bullet ele\}) \land \\
(\neg\, (\exists\, e : md \bullet e \in \{Inter \mid (atom\ ele).type \in \operatorname{ran} ierror \bullet ele\})) \land \\
(\forall\, e : \{g : Element \mid g \in_{\mathrm{p}} md\} \bullet e \in \{FlowObject \bullet ele\}) \land \\
md \in processSet \land \\
md \in noOverLap \land \\
(\forall\, f : \{g : Element \mid g \in_{\mathrm{p}} md\} \bullet f \in eventgate \Rightarrow \\
\quad (\forall\, e : Element \bullet (f, e) \in edge(direct(md, e)) \Rightarrow e \in sendelement))
\end{array}
$$

We consider the first two conjuncts.

$$
\begin{array}{ll}
new? \in \{Gate \bullet ele\} \land & \text{[def of } ed, new? \text{ and } outs?] \\
\{ed\} \cup outs? \subseteq \{End \bullet ele\} & \\
\Rightarrow & [(atom\ new?).type \in \{xgate, agate\} \text{ and } \{o : outs? \bullet (atom\ o).type\} = \{end\}] \\
\quad (\exists\, e, f : md \bullet e \in \{Start \bullet ele\} \land f \in \{End \bullet ele\}) \land & \\
\quad (\neg\, (\exists\, e : md \bullet e \in \{Inter \mid (atom\ ele).type \in \operatorname{ran} ierror \bullet ele\})) &
\end{array}
$$

We now consider the third conjunct.

$$(\{new?\} \cup outs?) \subseteq \{ele : Element \mid FlowObject\}$$

$\Rightarrow$ [def of *GenProc*, Lemma B.1 and def of $NonEvSplit \Rightarrow FlowObject$]
$(\forall\, e : \{g : Element \mid g \in_{\mathrm{p}} md\} \bullet e \in \{FlowObject \bullet ele\})$

We now consider the fourth conjunct.

$(atom\ new?).in = \{from\} \wedge$ [def of *new?*, *outs?* and *uniqueEnds*]
$(atom\ new?).out = \bigcup\{o : outs? \bullet (atom\ o).in\} \wedge$
$\bigcup\{o : outs? \bullet (atom\ o).out\} = \emptyset \wedge$
$\bigcup\{o : outs? \cup \{new?\} \bullet (atom\ o).exit\} = \emptyset \wedge$
$getMsgs(\{new?\} \cup outs?) = \emptyset \wedge$
$getIns(outs?) \cap getSeqflows\ proc = \emptyset$
$\Rightarrow (\forall\, e : \{g : Element \mid g \in_{\mathrm{p}} md\} \bullet$ [def of md, $proc \in noOverLap$]
$\bigcup\{k : md \bullet (atom\ k).in \cup outs(k)\} \cap \bigcup\{k : content\ e \bullet (atom\ k).in \cup outs(k)\} = \emptyset \wedge$
$(\forall\, f : \{g : Element \mid g \in_{\mathrm{p}} md\} \bullet e \neq f \Rightarrow$
$(atom\ e).in \cap (atom\ f).in = \emptyset \wedge$
$outs(e) \cap outs(f) = \emptyset \wedge$
$\bigcup\{k : content\ e \bullet (atom\ k).in \cup outs(k)\} \cap$
$\bigcup\{k : content\ f \bullet (atom\ k).in \cup outs(k)\} = \emptyset \wedge$
$getMsg\,(atom\ e) \cap getMsg\,(atom\ f) = \emptyset)) \Leftrightarrow md \in noOverLap$ [def of *noOverLap*]

We now consider the fifth conjunct.

$\{ed\} \cup outs? \subseteq \{ele : Element \mid End\} \wedge$ [def of *ed*, *new?* and *outs?*]
$\{o : outs? \bullet (new?, o)\} \subseteq (edge\ md) \wedge$
$\mathrm{dom}((edge\ proc) \rhd \{ed\}) = \mathrm{dom}((edge\ md) \rhd \{new?\})$
$\Rightarrow (\forall\, e : md \bullet$ [$proc \in processSet$]
$(e \notin \{\, End \bullet ele \,\} \Rightarrow (\exists f : md \bullet f \in \{\, End \bullet ele \,\} \wedge (e, f) \in (edge\ md)^+)) \wedge$
$(e \notin \{\, Start \bullet ele \,\} \Rightarrow (\exists f : md \bullet f \in \{\, Start \bullet ele \,\} \wedge (f, e) \in (edge\ md)^+))) \wedge$
$(\forall\, e : md \bullet$
$(\forall\, s : outs(e) \bullet \exists f : md \bullet s \in (atom\ f).in) \wedge$
$(\forall\, s : (atom\ e).in \bullet \exists f : md \bullet s \in outs(f))) \wedge$
$\Leftrightarrow md \in processSet$ [def of *processSet*]

We now consider the sixth conjunct.

$(\mathrm{dom}((edge\ proc) \rhd \{ed\}) \cup outs? \cup \{new?\}) \cap eventgate = \emptyset$ [def of *ed*, *outs?* and *new?*]
$\Rightarrow (\forall f : \{g : Element \mid g \in_{\mathrm{p}} md\} \bullet f \in eventgate \Rightarrow$ [$proc \in \{proc : Process \mid hasExgates\}$]
$(\forall\, e : Element \bullet (f, e) \in edge(direct(md, e)) \Rightarrow e \in sendelement))$
$\Rightarrow md \in \{proc : Process \mid hasExgates\}$ [def of *hasExgates*]

We therefore conclude that $md \in \{proc : Process \mid GenProc \wedge hasExgates\}$ follows directly from constraints on the before state *Pool* and the input components.
For case $ed \notin proc$, we consider the following inductive hypothesis

$\{s : cont(proc, \{ed\}) \bullet modify(content(s), \{new?\} \cup outs?, \{ed\})\} \subseteq \{proc : Process \mid Pool\}) \Rightarrow$
$((proc \setminus cont(proc, \{ed\})) \cup$
$\{s : cont(proc, \{ed\}) \bullet rep(s, modify(content(s), \{new?\} \cup outs?, \{ed\}))\}) \in \{proc : Process \mid Pool\})$

According to Lemma B.2, it is sufficient to deduce the following set of facts (conjuncts) about the

precondition schema.

$$cont(proc, \{ed\}) \subseteq \text{ran } compound \tag{B.6}$$

$$\#cont(proc, \{ed\}) = 1 \tag{B.7}$$

$$\bigcup \{s : cont(proc, \{ed\}) \bullet getSeqflows(modify(content(s), \{new?\} \cup outs?, \{ed\}))\} \cap \tag{B.8}$$
$$getSeqflows(proc \setminus \{s, e : Element \mid s \in cont(proc, \{ed\}) \land e \in_{\mathrm{e}} s \bullet e\}) = \emptyset$$

$$\bigcup \{s : cont(proc, \{ed\}) \bullet getMsgs(modify(content(s), \{new?\} \cup outs?, \{ed\}))\} \cap \tag{B.9}$$
$$getMsgs(proc \setminus \{s, e : Element \mid s \in cont(proc, \{ed\}) \land e \in_{\mathrm{e}} s \bullet e\}) = \emptyset$$

$$\{s, e : Element \mid s \in cont(proc, \{ed\}) \land \tag{B.10}$$
$$e \in modify(content(s), \{new?\} \cup outs?, \{ed\}) \land (atom\ e).type \in \text{ran } eerror \bullet e\} =$$
$$\{s, e : Element \mid s \in cont(proc, \{ed\}) \land e \in content(s) \land (atom\ e).type \in \text{ran } eerror \bullet e\}$$

We consider each of the conjuncts individually: Conjunct B.6 follows from the definition of *cont* and schema definition *FullSub*. Conjunct B.7 follows from the membership $proc \in noOverLap$. Conjunct B.8 follows from the following constraints:

$$getIns\{ed\} = getIns\{new?\}$$
$$getOuts\{new?\} = getIns(outs?)$$
$$getSeqflows(outs?) \cap getSeqflows\ proc = \emptyset$$
$$(outs\ new? \cup (getSeqflows\ (content\ new?))) \cap getSeqflows\ proc = \emptyset$$
$$proc \in noOverLap$$

Conjunct B.9 follows from $getMsgs(\{new?\} \cup outs?) = \emptyset$ and $proc \in noOverLap$. Conjunct B.10 follows from $(\{o : outs? \bullet (atom\ o).type\} \cup \{(atom\ ed).type, (atom\ new?).type\}) \cap \text{ran } eerror = \emptyset$. We conclude that $md \in \{proc : Process \mid Pool\}$ follows directly from constraints on the before state *Pool* and the input components. We therefore obtain the final simplified precondition schema, labelled *PreSplit*.

PreSplit

$Pool$
$CommonConstraints$
$outs? : \mathbb{F}_1\ Element$

$new? \in \{NonEvSplit \bullet ele\}$
$outs? \in uniqueEnds$
$getIns(outs?) \cap getSeqflows\ proc = \emptyset$
$getIns(outs?) = (atom\ new?).out$

□

B.11 Precondition of *EventSplitOp*

We expand pre *EventSplitOp* and apply the one-point rule to arrive at the following schema.

$$
\begin{array}{l}
[Pool;\ new? : Element;\ from? : Seqflow;\ events?, ends? : \mathbb{F}_1\ Element \mid \\
\quad \textbf{let}\ ed == (ends\ proc)\ from? \bullet \\
\quad\quad \textbf{let}\ md == modify(proc, \{new?\} \cup events? \cup ends?, \{ed\}) \bullet \\
\quad (outs\ new? \cup (getSeqflows\ (content\ new?))) \cap getSeqflows\ proc = \emptyset \land \\
\quad from? \in (atom\ new?).in \land \\
\quad (atom\ new?).in \subseteq \mathrm{dom}(ends\ proc) \land \\
\quad events? \subseteq \{ele : Element \mid OneInOutAtom\} \land \\
\quad ends? \subseteq \{ele : Element \mid InitialEnd\} \land \\
\quad events? \cup ends? \in uniqueIns \land \\
\quad \#events? = \#ends? \land \\
\quad new? \in \{EventSplit \bullet ele\} \land \\
\quad getIns(events?) = (atom\ new?).out \land \\
\quad getSeqflows(events? \cup ends?) \cap getSeqflows\ proc = \emptyset \land \\
\quad getOuts(events?) = getIns(ends?) \land \\
\quad md \in \{proc : Process \mid Pool\}]
\end{array}
$$

We consider the conjunct $md \in \{proc : Process \mid Pool\}$. We first expand the definition of md:

$$
md = \begin{cases}
(proc \cup \{new?\} \cup events? \cup ends?) \setminus \{ed\} & \text{if } ed \in proc \\
((proc \setminus cont(proc, \{ed\})) \cup & \text{otherwise} \\
\quad \{s : cont(proc, \{ed\}) \bullet \\
\quad\quad rep(s, modify(content(s), \{new?\} \cup events? \cup ends?, \{ed\}))\}) &
\end{cases}
$$

For the case $ed \in proc$, we show $md \in \{proc : Process \mid Pool\}$ follows directly from constraints on the before state *Pool* and the input components, where $md == (proc \cup \{new?\} \cup events? \cup ends?) \setminus \{ed\}$. We expand this membership.

$$
\begin{array}{l}
(\exists\, e, f : md \bullet e \in \{Start \bullet ele\} \land f \in \{End \bullet ele\}) \land \\
(\neg\, (\exists\, e : md \bullet e \in \{Inter \mid (atom\ ele).type \in \mathrm{ran}\ ierror \bullet ele\})) \land \\
(\forall\, e : \{g : Element \mid g \in_{\mathrm{p}} md\} \bullet e \in \{FlowObject \bullet ele\}) \land \\
md \in processSet \land \\
md \in noOverLap \land \\
(\forall\, f : \{g : Element \mid g \in_{\mathrm{p}} md\} \bullet f \in eventgate \Rightarrow \\
\quad (\forall\, e : Element \bullet (f, e) \in edge(direct(md, e)) \Rightarrow e \in sendelement))
\end{array}
$$

We consider the first and second conjuncts.

$$
\begin{array}{ll}
new? \in \{Gate \bullet ele\} \land & \text{[def of } ed, new?, events \text{ and } ends?] \\
\{ed\} \cup ends? \subseteq \{ele : Element \mid InitialEnd\} \land & \\
events? \subseteq \{ele : Element \mid OneInOutAtom\} & \\
\Rightarrow (\exists\, e, f : md \bullet e \in \{Start \bullet ele\} \land f \in \{End \bullet ele\}) \land & [proc \in \{GenProc \bullet proc\}] \\
\quad (\neg\, (\exists\, e : md \bullet e \in \{Inter \mid (atom\ ele).type \in \mathrm{ran}\ ierror \bullet ele\})) &
\end{array}
$$

We now consider the third conjunct.

$$
\begin{array}{ll}
(\{new?\} \cup events? \cup ends?) \subseteq \{ele : Element \mid FlowObject\} & \text{[def of } new?, ends? \text{ and } events?] \\
\Rightarrow (\forall\, e : \{g : Element \mid g \in_{\mathrm{p}} md\} \bullet e \in \{FlowObject \bullet ele\}) & \text{[def of } GenProc, \text{ Lemma B.1]}
\end{array}
$$

We now consider the fourth conjunct.

$$
(atom\ new?).in = \{from\} \land \qquad \text{[def of } new?, ends?, events?, uniqueIns]
$$

$(atom\ new?).out = \bigcup\{e : events? \bullet (atom\ e).in\} \wedge$
$\bigcup\{e : ends? \bullet (atom\ e).out\} = \emptyset \wedge$
$\bigcup\{e : ends? \cup events? \cup \{new?\} \bullet (atom\ e).exit\} = \emptyset \wedge$
$getMsgs(\{new?\} \cup ends? \cup events?) = \emptyset \wedge$
$getSeqflows(ends? \cup events?) \cap getSeqflows\ proc = \emptyset$
$\Rightarrow (\forall e : \{g : Element \mid g \in_{\mathrm{p}} md\} \bullet$ [$proc \in noOverLap$]
$\bigcup\{k : md \bullet (atom\ k).in \cup outs(k)\} \cap \bigcup\{k : content\ e \bullet (atom\ k).in \cup outs(k)\} = \emptyset \wedge$
$(\forall f : \{g : Element \mid g \in_{\mathrm{p}} md\} \bullet e \neq f \Rightarrow$
$(atom\ e).in \cap (atom\ f).in = \emptyset \wedge$
$outs(e) \cap outs(f) = \emptyset \wedge$
$\bigcup\{k : content\ e \bullet (atom\ k).in \cup outs(k)\} \cap$
$\bigcup\{k : content\ f \bullet (atom\ k).in \cup outs(k)\} = \emptyset \wedge$
$getMsg\ (atom\ e) \cap getMsg\ (atom\ f) = \emptyset)) \Leftrightarrow md \in noOverLap$ [def of $noOverLap$]

We now consider the fifth conjunct.

$\{ed\} \cup ends? \subseteq \{ele : Element \mid End\} \wedge$ [def of $new?$, $ends?$, $events?$, md]
$events? \subseteq \{ele : Element \mid Inter\} \wedge$
$\{e : events? \bullet (new?, e)\} \subseteq (edge\ md) \wedge$
$(edge\ md)(\!|\ events?\ |\!) = ends? \wedge$
$\mathrm{dom}((edge\ proc) \rhd \{ed\}) = \mathrm{dom}((edge\ md) \rhd \{new?\})$
$\Rightarrow (\forall e : md \bullet$ [$proc \in processSet$]
$(e \notin \{\ End \bullet ele\ \} \Rightarrow (\exists f : md \bullet f \in \{\ End \bullet ele\ \} \wedge (e, f) \in (edge\ md)^+)) \wedge$
$(e \notin \{\ Start \bullet ele\ \} \Rightarrow (\exists f : md \bullet f \in \{\ Start \bullet ele\ \} \wedge (f, e) \in (edge\ md)^+)))$
$(\forall e : md \bullet$
$(\forall s : outs(e) \bullet \exists f : md \bullet s \in (atom\ f).in) \wedge$
$(\forall s : (atom\ e).in \bullet \exists f : md \bullet s \in outs(f)))$
$\Leftrightarrow md \in processSet$ [def of $processSet$]

We now consider the sixth conjunct.

[def of ed, $ends?$, $events?$ and $new?$]
$((\mathrm{dom}((edge\ proc) \rhd \{ed\})) \cup ends? \cup events?) \cap eventgate = \emptyset \wedge$
$\mathrm{dom}((edge\ proc) \rhd \{ed\}) = \mathrm{dom}((edge\ md) \rhd \{new?\} \wedge$
$(edge\ md)(\!|\ \{new?\}\ |\!) = events? \wedge$
$new? \in eventgate \wedge$
$events? \subseteq sendelement$
$\Rightarrow (\forall f : \{g : Element \mid g \in_{\mathrm{p}} md\} \bullet f \in eventgate \Rightarrow$ [$proc \in \{proc : Process \mid hasExgates\}$]
$(\forall e : Element \bullet (f, e) \in edge(direct(md, e)) \Rightarrow e \in sendelement))$
$\Rightarrow md \in \{proc : Process \mid hasExgates\}$ [def of $hasExgates$]

We therefore conclude that $md \in \{proc : Process \mid Pool\}$ follows directly from constraints on the before state $Pool$ and the input components.
For case $ed \notin proc$, we consider the following inductive hypothesis.

$\{s : cont(proc, \{ed\}) \bullet$
$modify(content(s), \{new?\} \cup events? \cup ends?, \{ed\})\} \subseteq \{proc : Process \mid Pool\}) \Rightarrow$
$((proc \setminus cont(proc, \{ed\})) \cup \{s : cont(proc, \{ed\}) \bullet$

$$rep(s, modify(content(s), \{new?\} \cup events? \cup ends?, \{ed\}))\}) \in \{proc : Process \mid Pool\})$$

According to Lemma B.2, it is sufficient to deduce the following set of facts (conjuncts) about the precondition schema.

$$cont(proc, \{ed\}) \subseteq \text{ran } compound \tag{B.11}$$

$$\#cont(proc, \{ed\}) = 1 \tag{B.12}$$

$$\bigcup\{s : cont(proc, \{ed\}) \bullet getSeqflows(modify(content(s), \{new?\} \cup outs?, \{ed\}))\} \cap \tag{B.13}$$
$$getSeqflows(proc \setminus \{s, e : Element \mid s \in cont(proc, \{ed\}) \land e \in_e s \bullet e\}) = \emptyset$$

$$\bigcup\{s : cont(proc, \{ed\}) \bullet getMsgs(modify(content(s), \{new?\} \cup outs?, \{ed\}))\} \cap \tag{B.14}$$
$$getMsgs(proc \setminus \{s, e : Element \mid s \in cont(proc, \{ed\}) \land e \in_e s \bullet e\}) = \emptyset$$

$$\{s, e : Element \mid s \in cont(proc, \{ed\}) \land \tag{B.15}$$
$$e \in modify(content(s), \{new?\} \cup outs?, \{ed\}) \land (atom\ e).type \in \text{ran } eerror \bullet e\} =$$
$$\{s, e : Element \mid s \in cont(proc, \{ed\}) \land e \in content(s) \land (atom\ e).type \in \text{ran } eerror \bullet e\}$$

We consider each of the conjuncts individually: Conjunct B.11 follows from the definition of *cont* and schema definition *FullSub*. Conjunct B.12 follows from the membership $proc \in noOverLap$. Conjunct B.13 follows from the following constraints:

$$getIns\{ed\} = getIns\{new?\}$$
$$getOuts\{new?\} = getIns(events?)$$
$$getOuts(events?) = getIns(ends?)$$
$$getSeqflows(events? \cup ends?) \cap getSeqflows\ proc = \emptyset$$
$$(outs\ new? \cup (getSeqflows\ (content\ new?))) \cap getSeqflows\ proc = \emptyset$$
$$proc \in noOverLap$$

Conjunct B.14 follows from $getMsgs(\{new?\} \cup events? \cup ends?) = \emptyset$ and $proc \in noOverLap$. Conjunct B.15 follows from $(\{o : events? \cup ends? \bullet (atom\ o).type\} \cup \{(atom\ ed).type, (atom\ new?).type\}) \cap \text{ran } eerror = \emptyset$.

We conclude that $md \in \{proc : Process \mid Pool\}$ follows directly from constraints on the before state *Pool* and the input components. We therefore obtain the final simplified precondition schema, labelled *PreEventSplitOp*.

PreEventSplitOp

$$Pool$$
$$CommonConstraints$$
$$EventSplitAux$$

$$new? \in \{EventSplit \bullet ele\}$$
$$getIns(events?) = (atom\ new?).out$$
$$getSeqflows(events? \cup ends?) \cap getSeqflows\ proc = \emptyset$$
$$getOuts(events?) = getIns(ends?)$$

□

B.12 Precondition of *JoinOp*

We expand the definitions of pre *JoinOp*, apply schema quantification, the one point rule and properties of $\subseteq$ to arrive at the following precondition schema.

$$
\begin{array}{l}
[Pool;\ gate?, end? : Element \mid \\
\quad \textbf{let}\ es == (ends\ proc)(\!|\ (atom\ gate?).in\ |\!) \bullet \\
\quad\quad \textbf{let}\ md == modify(proc, \{gate?, end?\}, es) \bullet \\
\quad gate? \in \{NonEvJoin \bullet ele\} \land \\
\quad end? \in \{InitialEnd \bullet ele\} \land \\
\quad (atom\ gate?).in \subseteq \mathrm{dom}(ends\ proc) \land \\
\quad (proc, es) \in together \land \\
\quad (atom\ gate?).out = (atom\ end?).in \land \\
\quad (atom\ end?).in \cap getSeqflows\ proc = \emptyset \land \\
\quad md \in \{proc : Process \mid Pool\}]
\end{array}
$$

We consider conjunct $md \in \{proc : Process \mid Pool\}$. We first expand the definition of md:

$$
md = \begin{cases}
((proc \cup \{gate?, end?\}) \setminus es) & \text{if } es \subseteq proc \\
((proc \setminus cont(proc, \{ed\})) \cup & \text{otherwise} \\
\quad \{s : cont(proc, \{ed\}) \bullet rep(s, modify(content(s), \{gate?, end?\}, es))\}) &
\end{cases}
$$

For the case $es \subseteq proc$, we show $md \in \{proc : Process \mid Pool\}$ follows directly from constraints on the before state *Pool* and the input components, where $md == ((proc \cup \{gate?, end?\}) \setminus es)$. We expand this membership.

$$
\begin{array}{l}
(\exists\, e, f : md \bullet e \in \{Start \bullet ele\} \land f \in \{End \bullet ele\}) \land \\
(\neg\, (\exists\, e : md \bullet e \in \{Inter \mid (atom\ ele).type \in \mathrm{ran}\, ierror \bullet ele\})) \land \\
(\forall\, e : \{g : Element \mid g \in_{\mathrm{p}} md\} \bullet e \in \{FlowObject \bullet ele\}) \land \\
md \in processSet \land \\
md \in noOverLap \land \\
(\forall\, f : \{g : Element \mid g \in_{\mathrm{p}} md\} \bullet f \in eventgate \Rightarrow \\
\quad (\forall\, e : Element \bullet (f, e) \in edge(direct(md, e)) \Rightarrow e \in sendelement))
\end{array}
$$

We consider the first and second conjuncts.

$$
\begin{array}{ll}
(atom\ gate?).type \in \{xgate, agate\} \land & [\text{def of } end?,\ gate? \text{ and } ends] \\
\{o : es \cup \{end?\} \bullet (atom\ o).type\} = \{end\} & \\
\Rightarrow (\exists\, e, f : md \bullet e \in \{Start \bullet ele\} \land f \in \{End \bullet ele\}) \land & [proc \in \{GenProc \bullet proc\}] \\
\quad (\neg\, (\exists\, e : md \bullet e \in \{Inter \mid (atom\ ele).type \in \mathrm{ran}\, ierror \bullet ele\})) &
\end{array}
$$

We now consider the third conjunct.

$$
\begin{array}{ll}
gate? \in \{ele : Element \mid FlowObject\} \land & [\text{def of } NonEvJoin \Rightarrow (Gate \Rightarrow FlowObject)] \\
end? \in \{ele : Element \mid FlowObject\} & \\
\Rightarrow & [\text{def of } GenPool, \text{ Lemma B.1}] \\
\quad (\forall\, e : \{g : Element \mid g \in_{\mathrm{p}} md\} \bullet e \in \{FlowObject \bullet ele\}) &
\end{array}
$$

We now consider the fourth conjunct.

$$
\begin{array}{ll}
(atom\ gate?).in = \bigcup\{o : es \bullet (atom\ o).in\} \land & [\text{def of } es,\ gate? \text{ and } end?] \\
\{o : es \bullet (atom\ o).type\} \subseteq \{ele : Element \mid End\} \land & \\
(atom\ gate?).out = (atom\ end?).in \land & \\
getMsgs\{gate?, end?\} = \emptyset &
\end{array}
$$

$\Rightarrow (\forall\, e : \{g : Element \mid g \in_{\mathrm{p}} md\} \bullet$ [$proc \in noOverLap$]
$\quad \bigcup\{k : md \bullet (atom\, k).in \cup outs(k)\} \cap \bigcup\{k : content\, e \bullet (atom\, k).in \cup outs(k)\} = \emptyset \wedge$
$\quad (\forall\, f : \{g : Element \mid g \in_{\mathrm{p}} md\} \bullet e \neq f \Rightarrow$
$\quad\quad (atom\, e).in \cap (atom\, f).in = \emptyset \wedge$
$\quad\quad outs(e) \cap outs(f) = \emptyset \wedge$
$\quad\quad \bigcup\{k : content\, e \bullet (atom\, k).in \cup outs(k)\} \cap$
$\quad\quad \bigcup\{k : content\, f \bullet (atom\, k).in \cup outs(k)\} = \emptyset \wedge$
$\quad\quad getMsg\,(atom\, e) \cap getMsg\,(atom\, f) = \emptyset))$
$\Leftrightarrow md \in noOverLap$ [def of $noOverLap$]

We now consider the fifth conjunct.

$(atom\, gate?).in = \bigcup\{o : es \bullet (atom\, o).in\} \wedge$ [def of es, $gate?$ and $end?$]
$\{o : es \bullet (atom\, o).type\} \subseteq \{ele : Element \mid End\} \wedge$
$(atom\, gate?).out = (atom\, end?).in \wedge$
$(atom\, gate?).exit \cup (atom\, end?).exit = \emptyset$
$\Rightarrow (\forall\, e : md \bullet$ [$proc \in processSet$ and $md \cap (ends\, proc)(\!|\, (atom\, gate?).in \,|\!) = \emptyset$]
$\quad (e \notin \{\, End \bullet ele \,\} \Rightarrow (\exists f : md \bullet f \in \{\, End \bullet ele \,\} \wedge (e, f) \in (edge\, md)^+)) \wedge$
$\quad (e \notin \{\, Start \bullet ele \,\} \Rightarrow (\exists f : md \bullet f \in \{\, Start \bullet ele \,\} \wedge (f, e) \in (edge\, md)^+))) \wedge$
$\quad (\forall\, e : md \bullet$
$\quad\quad (\forall\, s : outs(e) \bullet \exists f : md \bullet s \in (atom\, f).in) \wedge$
$\quad\quad (\forall\, s : (atom\, e).in \bullet \exists f : md \bullet s \in outs(f)))$
$\Leftrightarrow md \in processSet$ [def of $processSet$]

We now consider the sixth conjunct.

$(atom\, gate?).in = \bigcup\{o : es \bullet (atom\, o).in\} \wedge$ [def of es, $gate?$ and $end?$]
$\{o : es \bullet (atom\, o).type\} \subseteq \{ele : Element \mid End\} \wedge$
$\{gate?, end?\} \notin eventgate$
$\Rightarrow (\forall f : \{g : Element \mid g \in_{\mathrm{p}} md\} \bullet f \in eventgate \Rightarrow$ [$proc \in \{proc : Process \mid hasExgates\}$]
$\quad (\forall\, e : Element \bullet (f, e) \in edge(direct(md, e)) \Rightarrow e \in sendelement))$
$\Rightarrow md \in \{proc : Process \mid hasExgates\}$ [def of $hasExgates$]

We therefore conclude that $md \in \{proc : Process \mid Pool\}$ follows directly from constraints on the before state $Pool$ and the input components.

For case $es \cap proc = \emptyset$, we consider the following inductive hypothesis

$\{s : cont(proc, es) \bullet modify(content(s), \{gate?, end?\}, es)\} \subseteq \{proc : Process \mid Pool\}) \Rightarrow$
$((proc \setminus cont(proc, es)) \cup$
$\quad \{s : cont(proc, es) \bullet rep(s, modify(content(s), \{gate?, end?\}, es))\}) \in \{proc : Process \mid Pool\})$

According to Lemma B.2, it is sufficient to deduce the following set of facts (conjuncts) about the

precondition schema.

$$cont(proc, es) \subseteq \text{ran } compound \tag{B.16}$$

$$\#cont(proc, es) = 1 \tag{B.17}$$

$$\bigcup\{s : cont(proc, es) \bullet getSeqflows(modify(content(s), \{gate?, end?\}, es))\} \cap \tag{B.18}$$
$$getSeqflows(proc \setminus \{s, e : Element \mid s \in cont(proc, es) \land e \in_{\mathrm{e}} s \bullet e\}) = \emptyset$$

$$\bigcup\{s : cont(proc, es) \bullet getMsgs(modify(content(s), \{gate?, end?\}, es))\} \cap \tag{B.19}$$
$$getMsgs(proc \setminus \{s, e : Element \mid s \in cont(proc, es) \land e \in_{\mathrm{e}} s \bullet e\}) = \emptyset$$

$$\{s, e : Element \mid s \in cont(proc, es) \land \tag{B.20}$$
$$e \in modify(content(s), \{gate?, end?\}, es) \land (atom\ e).type \in \text{ran } eerror \bullet e\} =$$
$$\{s, e : Element \mid s \in cont(proc, \{ed\}) \land e \in content(s) \land (atom\ e).type \in \text{ran } eerror \bullet e\}$$

We consider each of the conjuncts individually: Conjunct B.16 follows from the definition of *cont* and schema definition *FullSub*. Conjunct B.17 follows from the membership $proc \in noOverLap$ and the property $(proc, es) \in together$. Conjunct B.18 follows from the following constraints:

$$getIns(es) = getIns\{gate?\}$$
$$getOuts\{gate?\} = getIns\{end?\}$$
$$getSeqflows\{gate?, end?\} \cap getSeqflows\ proc = \emptyset$$
$$proc \in noOverLap$$

Conjunct B.19 follows from $getMsgs\{gate?, end?\} = \emptyset$ and $proc \in noOverLap$, and Conjunct B.20 follows from $(\{o : es \bullet (atom\ o).type\} \cup \{(atom\ gate?).type, (atom\ end?).type\}) \cap \text{ran } eerror = \emptyset$. We conclude that $md \in \{proc : Process \mid Pool\}$ follows directly from constraints on the before state *Pool* and the input components. We therefore obtain the final simplified precondition schema, labelled *PreJoinOp*.

PreJoinOp

$$Pool$$
$$gate?, end? : Element$$

$$gate? \in \{NonEvJoin \bullet ele\}$$
$$end? \in \{InitialEnd \bullet ele\}$$
$$(atom\ gate?).in \subseteq \text{dom}(ends\ proc)$$
$$\#(cont(proc, (ends\ proc)(\!|\ (atom\ gate?).in\ |\!))) = 1$$
$$(atom\ gate?).out = (atom\ end?).in$$
$$(atom\ end?).in \cap getSeqflows\ proc = \emptyset$$

□

B.13 Precondition of *Loop*

We expand pre *Loop* and apply the one-point rule to arrive at the following schema.

$$
\begin{array}{l}
[Pool;\ from?, connect?, f2?, t2? : Seqflow;\ split?, join?, end? : Element \mid \\
\quad \textbf{let}\ ed1 == (ends\ proc)\ from? ; \\
\quad\quad ed2 == (nonsends\ proc)\ f2? \bullet \\
\quad\quad\quad \textbf{let}\ ch2 == cge(ed2, f2?, t2?) \bullet \\
\quad\quad\quad\quad \textbf{let}\ md == modify(proc, \{split?, join?, end?, ch2\}, \{ed1, ed2\}) \bullet \\
(outs\ split? \cup (getSeqflows\ (content\ split?))) \cap getSeqflows\ proc = \emptyset \land \\
from? \in (atom\ split?).in \land \\
(atom\ split?).in \subseteq \mathrm{dom}(ends\ proc) \land \\
split? \in \{NonEvSplit \bullet ele\} \land \\
end? \in \{ele : Element \mid InitialEnd\} \land \\
connect? \notin (atom\ end?).in \land \\
(atom\ split?).out = \{connect?\} \cup (atom\ end?).in \land \\
t2? \notin getSeqflows\{ed2\} \land \\
f2? \in (atom\ ed2).in \land \\
\{ed2, ch2\} \subseteq \{ele : Element \mid FlowObject\} \land \\
join? \in \{NonEvJoin \bullet ele\} \land \\
connect? \neq t2? \land \\
(atom\ join?).in = \{f2?, connect?\} \land \\
outs\ join? = \{t2?\} \land \\
from? \neq f2? \land \\
t2? \notin getSeqflows\ proc \cup (atom\ end?).in \land \\
(proc, \{ed1, ed2\}) \in together \land \\
(ed2, ed1) \in (edge(direct(proc, ed2)))^+ \land \\
md \in \{proc : Process \mid Pool\}]
\end{array}
$$

Following pre *ChangeFlow*, we can see that the conjunct $ch2 \in \{ele : Element \mid FlowObject\}$ follows directly from constraints on the before state *Pool* and the input components. We now consider the conjunct $md \in \{proc : Process \mid Pool\}$. We first expand the definition of md:

$$
md = \begin{cases}
(proc \cup \{split?, join?, end?, ch2\}) \setminus \{ed1, ed2\} & \text{if } \{ed1, ed2\} \subseteq proc \\
((proc \setminus cont(proc, \{ed1\})) \cup \{s : cont(proc, \{ed1\}) \bullet & \text{otherwise} \\
\qquad rep(s, modify(content(s), \{split?, join?, end?, ch2\}, \{ed1, ed2\}))\}) &
\end{cases}
$$

For the case $\{ed1, ed2\} \subseteq proc$, we show $md \in \{proc : Process \mid Pool\}$ follows directly from constraints on the before state *Pool* and the input components, where $md == (proc \cup \{split?, join?, end?, ch2\}) \setminus \{ed1, ed2\}$. We expand this membership.

$$
\begin{array}{l}
(\exists\, e, f : md \bullet e \in \{Start \bullet ele\} \land f \in \{End \bullet ele\}) \land \\
(\neg\, (\exists\, e : md \bullet e \in \{Inter \mid (atom\ ele).type \in \mathrm{ran}\ ierror \bullet ele\})) \land \\
(\forall\, e : \{g : Element \mid g \in_{\mathrm{p}} md\} \bullet e \in \{FlowObject \bullet ele\}) \land \\
md \in processSet \land \\
md \in noOverLap \land \\
(\forall\, f : \{g : Element \mid g \in_{\mathrm{p}} md\} \bullet f \in eventgate \Rightarrow \\
\quad (\forall\, e : Element \bullet (f, e) \in edge(direct(md, e)) \Rightarrow e \in sendelement))
\end{array}
$$

We consider the first and second conjuncts.

$$
\begin{array}{ll}
\{ed1, end?\} \subseteq \{End \bullet ele\} \land & \text{[def of } cge, ed, end?, split? \text{ and } join?] \\
(atom\ ch2).type = (atom\ ed2).type \land & \\
\{split?, join?\} \subseteq \{Gate \bullet ele\} &
\end{array}
$$

$\Rightarrow (\exists\, e, f : md \bullet e \in \{Start \bullet ele\} \land f \in \{End \bullet ele\}) \land$ [$proc \in \{GenProc \bullet proc\}$]
$\quad (\neg\, (\exists\, e : md \bullet e \in \{Inter \mid (atom\ ele).type \in \operatorname{ran} ierror \bullet ele\}))$

We now consider the third conjunct.

$\{split?, join?, end?, ch2\} \subseteq \{ele : Element \mid FlowObject\}$ [def $ch2$, $end?$, $split?$ and $join?$]
$\Rightarrow$ [def of $GenPool$, Lemma B.1 and $(NonEvJoin \lor NonEvSplit) \Rightarrow (Gate \Rightarrow FlowObject)$]
$\quad (\forall\, e : \{g : Element \mid g \in_{\mathrm{p}} md\} \bullet e \in \{FlowObject \bullet ele\})$

We now consider the fourth conjunct.

$f2? \neq from? \land$
$((atom\ end?).in \cup \{t2?, connect?\}) \cap getSeqflows\ proc = \emptyset \land$
$(atom\ split?).in = \{from?\} \land$
$(atom\ split?).out = \{connect?\} \cup (atom\ end?).in \land$
$(atom\ join?).in = \{f2?, connect?\} \land$
$(atom\ join?).out = \{t2?\} \land$
$getMsgs\{split?, join?, end?\} = \emptyset \land$
$getMsgs\{ch2\} = getMsgs\{ch1\} \land$
$(atom\ ch2).in = ((atom\ ed2).in \setminus \{f2?\}) \cup \{t2?\} \land$
$(outs\ ch2) = (outs\ ed2)$
$\Rightarrow (\forall\, e : \{g : Element \mid g \in_{\mathrm{p}} md\} \bullet$ [$proc \in noOverLap$]
$\quad \bigcup\{k : md \bullet (atom\ k).in \cup outs(k)\} \cap \bigcup\{k : content\ e \bullet (atom\ k).in \cup outs(k)\} = \emptyset \land$
$\quad (\forall f : \{g : Element \mid g \in_{\mathrm{p}} md\} \bullet e \neq f \Rightarrow$
$\qquad (atom\ e).in \cap (atom\ f).in = \emptyset \land$
$\qquad outs(e) \cap outs(f) = \emptyset \land$
$\qquad \bigcup\{k : content\ e \bullet (atom\ k).in \cup outs(k)\} \cap$
$\qquad \bigcup\{k : content\ f \bullet (atom\ k).in \cup outs(k)\} = \emptyset \land$
$\qquad getMsg\,(atom\ e) \cap getMsg\,(atom\ f) = \emptyset))$
$\Leftrightarrow md \in noOverLap$ [def of $noOverLap$]

We now consider the fifth conjunct.

$f2? \neq from? \land$
$((atom\ end?).in \cup \{t2?, connect?\}) \cap getSeqflows\ proc = \emptyset \land$
$(atom\ split?).in = \{from?\} \land$
$(atom\ split?).out = \{connect?\} \cup (atom\ end?).in \land$
$(atom\ join?).in = \{f2?, connect?\} \land$
$(atom\ join?).out = \{t2?\} \land$
$(atom\ ch2).in = ((atom\ ed2).in \setminus \{f2?\}) \cup \{t2?\} \land$
$(outs\ ch2) = (outs\ ed2) \land$
$\operatorname{dom}((edge\ proc) \rhd \{ed1\}) = \operatorname{dom}((edge\ md) \rhd \{split?\}) \land$
$\operatorname{dom}((edge\ proc) \rhd \{ed2\}) = (\operatorname{dom}((edge\ md) \rhd \{join?, ch2\})) \setminus \{split?\} \land$
$((edge\ proc)(\!|\ \{ed2\}\ |\!)) = ((edge\ md)(\!|\ \{ch2\}\ |\!))$
$\Rightarrow (\forall\, e : md \bullet$ [$proc \in processSet$]
$\quad (e \notin \{\, End \bullet ele\,\} \Rightarrow (\exists f : md \bullet f \in \{\, End \bullet ele\,\} \land (e, f) \in (edge\ md)^+)) \land$
$\quad (e \notin \{\, Start \bullet ele\,\} \Rightarrow (\exists f : md \bullet f \in \{\, Start \bullet ele\,\} \land (f, e) \in (edge\ md)^+))) \land$
$\quad (\forall\, e : md \bullet$

$$
\begin{aligned}
&(\forall\, s : outs(e) \bullet \exists f : md \bullet s \in (atom\, f).in) \land \\
&(\forall\, s : (atom\, e).in \bullet \exists f : md \bullet s \in outs(f))) \\
\Leftrightarrow\ & md \in processSet && \text{[def of } processSet\text{]}
\end{aligned}
$$

We now consider the sixth conjunct.

$$
\begin{aligned}
& ed1 \in \operatorname{ran} ends\, proc \land && \text{[def of } ed1,\ ed2,\ split?\text{ and } join?\text{]} \\
& ed2 \in \operatorname{ran} nonsends\, proc \land \\
& (atom\, end?).type = (atom\, ed1).type \land \\
& (atom\, ch2).type = (atom\, ed2).type \land \\
& \{split?, join?\} \cap eventgate = \emptyset \\
\Rightarrow\ & (\forall f : \{g : Element \mid g \in_{\mathrm{p}} md\} \bullet f \in eventgate \Rightarrow && [proc \in \{proc : Process \mid hasExgates\}] \\
& \quad (\forall\, e : Element \bullet (f, e) \in edge(direct(md, e)) \Rightarrow e \in sendelement)) \\
\Rightarrow\ & md \in \{proc : Process \mid hasExgates\} && \text{[def of } hasExgates\text{]}
\end{aligned}
$$

We therefore conclude that $md \in \{proc : Process \mid Pool\}$ follows directly from constraints on the before state $Pool$ and the input components.
For case $\{ed1, ed2\} \cap proc = \emptyset$, we consider the following inductive hypothesis

$$
\begin{aligned}
& \{s : cont(proc, \{ed1, ed2\}) \bullet \\
& \quad modify(content(s), \{split?, join?, end?, ch2\}, \{ed1, ed2\})\} \subseteq \{proc : Process \mid Pool\}) \Rightarrow \\
& ((proc \setminus cont(proc, \{ed1, ed2\})) \cup \{s : cont(proc, \{ed1, ed2\}) \bullet \\
& \quad rep(s, modify(content(s), \{split?, join?, end?, ch2\}, \{ed1, ed2\}))\}) \in \{proc : Process \mid Pool\})
\end{aligned}
$$

According to Lemma B.2, it is sufficient to deduce the following set of facts (conjuncts) about the precondition schema.

$$cont(proc, \{ed1, ed2\}) \subseteq \operatorname{ran} compound \tag{B.21}$$

$$\#cont(proc, \{ed1, ed2\}) = 1 \tag{B.22}$$

$$
\begin{aligned}
& \bigcup \{s : cont(proc, \{ed1, ed2\}) \bullet \\
& \qquad getSeqflows(modify(content(s), \{split?, join?, end?, ch2\}, \{ed1, ed2\}))\} \cap \\
& \qquad getSeqflows(proc \setminus \{s, e : Element \mid s \in cont(proc, \{ed1, ed2\}) \land e \in_{\mathrm{e}} s \bullet e\}) = \emptyset
\end{aligned} \tag{B.23}
$$

$$
\begin{aligned}
& \bigcup \{s : cont(proc, \{ed1, ed2\}) \bullet \\
& \qquad getMsgs(modify(content(s), \{split?, join?, end?, ch2\}, \{ed1, ed2\}))\} \cap \\
& \qquad getMsgs(proc \setminus \{s, e : Element \mid s \in cont(proc, \{ed1, ed2\}) \land e \in_{\mathrm{e}} s \bullet e\}) = \emptyset
\end{aligned} \tag{B.24}
$$

$$
\begin{aligned}
& \{s, e : Element \mid s \in cont(proc, \{ed1, ed2\}) \land \\
& \quad e \in modify(content(s), \{split?, join?, end?, ch2\}, \{ed1, ed2\}) \land (atom\, e).type \in \operatorname{ran} eerror \bullet e\} = \\
& \quad \{s, e : Element \mid s \in cont(proc, \{ed1, ed2\}) \land e \in content(s) \land (atom\, e).type \in \operatorname{ran} eerror \bullet e\}
\end{aligned} \tag{B.25}
$$

We consider each of the conjuncts individually: Conjunct B.21 follows from the definition of $cont$ and schema definition $FullSub$. Conjunct B.22 follows from the memberships $proc \in noOverLap$ and $(proc, \{ed1, ed2\}) \in together$. Conjunct B.23 follows from the following constraints:

$$
\begin{aligned}
& getIns\{ed1\} = getIns\{split?\} \\
& getOuts\{split?\} = (getIns\{end?\}) \cup \{connect?\} \\
& getIns\{join?\} = \{connect?\} \cup getIns\{ed2\} \\
& getOuts\{join?\} = getIns\{ch2\} \\
& getOuts\{ch2\} = getOuts\{ed2\} \\
& getSeqflows\{end?\} \cap getSeqflows\, proc = \emptyset \\
& ((outs\, split?) \cup \{t2?\}) \cap getSeqflows\, proc = \emptyset \\
& proc \in noOverLap
\end{aligned}
$$

Conjunct B.24 follows from $getMsgs\{split?, join?, end?\} = \emptyset$, $getMsgs\{ed2\} = getMsgs\{ch2\}$ and $proc \in noOverLap$. Conjunct B.25 follows from $\{e : \{ed1, ed2, end?, split?, join?, ch2\} \bullet (atom\ e).type\} \cap \mathrm{ran}\ eerror = \emptyset$.

We conclude that $md \in \{proc : Process \mid Pool\}$ follows directly from constraints on the before state $Pool$ and the input components. We apply set theory to obtain the final simplified precondition schema, labelled $PreLoop$.

PreLoop

$CommonConstraints[split?/new?]$
$ConnectJoin0$
$end? : Element$

$split? \in \{NonEvSplit \bullet ele\}$
$end? \in \{ele : Element \mid InitialEnd\}$
$connect? \notin (atom\ end?).in$
$(atom\ split?).out = \{connect?\} \cup (atom\ end?).in$
$from? \neq f2?$
$f2? \in \mathrm{dom}(nonsends\ proc)$
$t2? \notin getSeqflows\ proc \cup (atom\ end?).in$
$(proc, \{(ends\ proc)\ from?, (nonsends\ proc)\ f2?\}) \in together$
$((nonsends\ proc)\ f2?, (ends\ proc)\ from?) \in (edge(direct(proc, (nonsends\ proc)\ f2?)))^+$

□

B.14 Precondition of *EventLoop*

We expand pre *EventLoop* and apply the one-point rule to arrive at the following schema.

$$
\begin{array}{l}
[Pool;\ split?, join?, end? : Element;\ events? : \mathbb{F}_1\ Element;\ from?, connect?, f2?, t2? : Seqflow \mid \\
\quad \textbf{let}\ ed1 == (ends\ proc)\ from?;\ ed2 == (nonsends\ proc)\ f2? \bullet \\
\qquad \textbf{let}\ ch2 == cge(ed2, f2?, t2?) \bullet \\
\qquad\quad \textbf{let}\ md == modify(proc, \{split?, join?, ch2, end?\} \cup events?, \{ed1, ed2\}) \bullet \\
\qquad (outs\ split? \cup (getSeqflows\ (content\ split?))) \cap getSeqflows\ proc = \emptyset \land \\
\qquad from? \in (atom\ split?).in \land \\
\qquad (atom\ split?).in \subseteq \mathrm{dom}(ends\ proc) \land \\
\qquad events? \subseteq \{ele : Element \mid OneInOutAtom\} \land \\
\qquad events? \in uniqueIns \land \\
\qquad \#events? = 2 \land \\
\qquad split? \in \{EventSplit \bullet ele\} \land \\
\qquad end? \in \{ele : Element \mid InitialEnd\} \land \\
\qquad getIns(events?) = (atom\ split?).out \land \\
\qquad getSeqflows\ events? \cap getSeqflows\ proc = \emptyset \land \\
\qquad (\{connect?\} \cup (atom\ end?).in) \cap getIns\ events? = \emptyset \land \\
\qquad connect? \notin (atom\ end?).in \land \\
\qquad getOuts\ events? = \{connect?\} \cup (atom\ end?).in \land \\
\qquad ed2 \notin \{Start \bullet ele\} \land \\
\qquad t2? \notin getSeqflows\{ed2\} \land \\
\qquad f2? \in (atom\ ed2).in \land \\
\qquad \{ed2, ch2\} \subseteq \{ele : Element \mid FlowObject\} \land \\
\qquad join? \in \{NonEvJoin \bullet ele\} \land \\
\qquad connect? \neq t2? \land \\
\qquad (atom\ join?).in = \{f2?, connect?\} \land \\
\qquad outs\ join? = \{t2?\} \land \\
\qquad t2? \notin getSeqflows\ proc \land \\
\qquad from? \neq f2? \land \\
\qquad t2? \notin getSeqflows(events?) \land \\
\qquad (proc, \{ed1, ed2\}) \in together \land \\
\qquad (ed2, ed1) \in (edge(direct(proc, ed2)))^+ \land \\
\qquad md \in \{proc : Process \mid Pool\}]
\end{array}
$$

Following pre *ChangeFlow*, we see that the conjunct $ch2 \in \{ele : Element \mid FlowObject\}$ follows directly from constraints on the before state *Pool* and the input components. We now consider the conjunct $md \in \{proc : Process \mid Pool\}$. We first expand the definition of md:

$$
md = \begin{cases}
(proc \cup \{split?, join?, end?, ch2\} \cup events?) \setminus \{ed1, ed2\} & \text{if } \{ed1, ed2\} \subseteq proc \\
((proc \setminus cont(proc, \{ed1\})) \cup \{s : cont(proc, \{ed1\}) \bullet & \text{otherwise} \\
\quad rep(s, modify(content(s), & \\
\qquad \{split?, join?, end?, ch2\} \cup events?, \{ed1, ed2\}))\}) &
\end{cases}
$$

For the case $\{ed1, ed2\} \subseteq proc$, we show $md \in \{proc : Process \mid Pool\}$ follows directly from constraints on the before state *Pool* and the input components, where $md == (proc \cup \{split?, join?, end?, ch2\} \cup events?) \setminus \{ed1, ed2\}$. We expand this membership.

$$
\begin{array}{l}
(\exists\ e, f : md \bullet e \in \{Start \bullet ele\} \land f \in \{End \bullet ele\}) \land \\
(\neg\ (\exists\ e : md \bullet e \in \{Inter \mid (atom\ ele).type \in \mathrm{ran}\ ierror \bullet ele\})) \land \\
(\forall\ e : \{g : Element \mid g \in_{\mathrm{p}} md\} \bullet e \in \{FlowObject \bullet ele\}) \land \\
md \in processSet \land \\
md \in noOverLap \land \\
(\forall f : \{g : Element \mid g \in_{\mathrm{p}} md\} \bullet f \in eventgate \Rightarrow
\end{array}
$$

$$(\forall\, e : Element \bullet (f, e) \in edge(direct(md, e)) \Rightarrow e \in sendelement))$$

We consider the first and second conjuncts.

$$\{end?, ed1\} \subseteq \{End \bullet ele\} \wedge \qquad \text{[def of } cge, split?, join?, end? \text{ and } events?\text{]}$$
$$(\{ch2, ed2\} \cup events?) \cap \{Inter \mid (atom\ ele).type \in \operatorname{ran} ierror \bullet ele\} = \emptyset \wedge$$
$$\{split?, join?\} \subseteq \{Gate \bullet ele\} \wedge$$
$$\Rightarrow (\exists\, e, f : md \bullet e \in \{Start \bullet ele\} \wedge f \in \{End \bullet ele\}) \wedge \qquad [proc \in \{GenProc \bullet proc\}]$$
$$(\neg\, (\exists\, e : md \bullet e \in \{Inter \mid (atom\ ele).type \in \operatorname{ran} ierror \bullet ele\}))$$

We now consider the third conjunct.

$$(events? \cup \{split?, join?, end?, ch2\}) \subseteq \{ele : Element \mid FlowObject\}$$
$$\Rightarrow defof \text{GenPool,FlowObject} and Lemma\ B.1$$
$$(\forall\, e : \{g : Element \mid g \in_{\mathrm{p}} md\} \bullet e \in \{FlowObject \bullet ele\})$$

We now consider the fourth conjunct.

$$(\{t2?, connect?\} \cup getIns(\{end?\} \cup events?)) \cap getSeqflows\ proc = \emptyset \wedge$$
$$from? \neq f2? \wedge$$
$$\{from?\} = (atom\ ed1).in \wedge$$
$$f2? \in (atom\ ed2).in \wedge$$
$$(atom\ split?).in = \{from?\} \wedge$$
$$(atom\ split?).out = getIns(events?) \wedge$$
$$getOuts\ events? = \{connect?\} \cup (atom\ end?).in \wedge$$
$$(atom\ join?).in = \{f2?, connect?\} \wedge$$
$$(atom\ join?).out = \{t2?\} \wedge$$
$$getMsgs(\{split?, join?, end?\} \cup events?) = \emptyset$$
$$(atom\ ch2).in = ((atom\ ed2).in \setminus \{f2?\}) \cup \{t2?\} \wedge$$
$$(outs\ ch2) = (outs\ ed2) \wedge$$
$$getMsgs\{ch2\} = getMsgs\{ed2\}$$
$$\Rightarrow (\forall\, e : \{g : Element \mid g \in_{\mathrm{p}} md\} \bullet \qquad [proc \in noOverLap]$$
$$\bigcup\{k : md \bullet (atom\ k).in \cup outs(k)\} \cap \bigcup\{k : content\ e \bullet (atom\ k).in \cup outs(k)\} = \emptyset \wedge$$
$$(\forall f : \{g : Element \mid g \in_{\mathrm{p}} md\} \bullet e \neq f \Rightarrow$$
$$(atom\ e).in \cap (atom\ f).in = \emptyset \wedge$$
$$outs(e) \cap outs(f) = \emptyset \wedge$$
$$\bigcup\{k : content\ e \bullet (atom\ k).in \cup outs(k)\} \cap$$
$$\bigcup\{k : content\ f \bullet (atom\ k).in \cup outs(k)\} = \emptyset \wedge$$
$$getMsg\ (atom\ e) \cap getMsg\ (atom\ f) = \emptyset))$$
$$\Leftrightarrow md \in noOverLap \qquad \text{[def of } noOverLap\text{]}$$

We now consider the fifth conjunct.

$$(\{t2?, connect?\} \cup getIns(\{end?\} \cup events?)) \cap getSeqflows\ proc = \emptyset \wedge$$
$$\{from?\} = (atom\ ed1).in \wedge$$
$$f2? \in (atom\ ed2).in \wedge$$
$$(atom\ split?).in = \{from?\} \wedge$$
$$(atom\ split?).out = getIns(events?) \wedge$$
$$getOuts\ events? = \{connect?\} \cup (atom\ end?).in \wedge$$

$(atom\ join?).in = \{f2?, connect?\} \land$
$(atom\ join?).out = \{t2?\} \land$
$(atom\ ch2).in = ((atom\ ed2).in \setminus \{f2?\}) \cup \{t2?\} \land$
$(outs\ ch2) = (outs\ ed2) \land$
$\mathrm{dom}((edge\ proc) \rhd \{ed1\}) = \mathrm{dom}((edge\ md) \rhd \{split?\}) \land$
$\mathrm{dom}((edge\ proc) \rhd \{ed2\}) = (\mathrm{dom}((edge\ md) \rhd \{join?, ch2\})) \setminus events? \land$
$((edge\ proc)(\!|\ \{ed2\}\ |\!)) = ((edge\ md)(\!|\ \{ch2\}\ |\!))$
$\Rightarrow (\forall\, e : md \bullet$ [$proc \in processSet$ and transitivity]
$\quad (e \notin \{\, End \bullet ele \,\} \Rightarrow (\exists f : md \bullet f \in \{\, End \bullet ele \,\} \land (e, f) \in (edge\ md)^+)) \land$
$\quad (e \notin \{\, Start \bullet ele \,\} \Rightarrow (\exists f : md \bullet f \in \{\, Start \bullet ele \,\} \land (f, e) \in (edge\ md)^+))) \land$
$\quad (\forall\, e : md \bullet$
$\quad\quad (\forall\, s : outs(e) \bullet \exists f : md \bullet s \in (atom\ f).in) \land$
$\quad\quad (\forall\, s : (atom\ e).in \bullet \exists f : md \bullet s \in outs(f)))$
$\Leftrightarrow md \in processSet$ [def of $processSet$]

We now consider the sixth conjunct.

$\{end?, ed1\} \subseteq \{End \bullet ele\} \land$ [def of $end?$, $ed2$, $events?$, $split?$ and $join?$]
$ed2 \in \mathrm{ran}\ nonsends\ proc \land$
$\{join?\} \cap eventgate = \emptyset$
$split? \in eventgate \land$
$events? \subseteq sendelement \land$
$(atomic\ splits?).out = getIns(events?)$
$\Rightarrow (\forall f : \{g : Element \mid g \in_{\mathrm{p}} md\} \bullet f \in eventgate \Rightarrow$ [$proc \in \{proc : Process \mid hasExgates\}$]
$\quad (\forall\, e : Element \bullet (f, e) \in edge(direct(md, e)) \Rightarrow e \in sendelement))$
$\Rightarrow md \in \{proc : Process \mid hasExgates\}$ [def of $hasExgates$]

We therefore conclude that $md \in \{proc : Process \mid Pool\}$ follows directly from constraints on the before state $Pool$ and the input components.
For case $\{ed1, ed2\} \cap proc = \emptyset$, we consider the following inductive hypothesis

$\{s : cont(proc, \{ed1, ed2\}) \bullet$
$\quad modify(content(s), \{split?, join?, end?, ch2\} \cup events?, \{ed1, ed2\})\} \subseteq \{proc : Process \mid Pool\}) \Rightarrow$
$((proc \setminus cont(proc, \{ed1, ed2\})) \cup \{s : cont(proc, \{ed1, ed2\}) \bullet$
$\quad rep(s, modify(content(s), \{split?, join?, end?, ch2\} \cup events?, \{ed1, ed2\}))\}) \in \{proc : Process \mid Pool\})$

According to Lemma B.2, it is sufficient to deduce the following set of facts (conjuncts) about the

precondition schema.

$$cont(proc, \{ed1, ed2\}) \subseteq \text{ran } compound \tag{B.26}$$

$$\#cont(proc, \{ed1, ed2\}) = 1 \tag{B.27}$$

$$\bigcup \{s : cont(proc, \{ed1, ed2\}) \bullet \tag{B.28}$$
$$getSeqflows(modify(content(s), \{split?, join?, end?, ch2\} \cup events?, \{ed1, ed2\}))\} \cap$$
$$getSeqflows(proc \setminus \{s, e : Element \mid s \in cont(proc, \{ed1, ed2\}) \land e \in_{\mathrm{e}} s \bullet e\}) = \emptyset$$

$$\bigcup \{s : cont(proc, \{ed1, ed2\}) \bullet \tag{B.29}$$
$$getMsgs(modify(content(s), \{split?, join?, end?, ch2\} \cup events?, \{ed1, ed2\}))\} \cap$$
$$getMsgs(proc \setminus \{s, e : Element \mid s \in cont(proc, \{ed1, ed2\}) \land e \in_{\mathrm{e}} s \bullet e\}) = \emptyset$$

$$\{s, e : Element \mid s \in cont(proc, \{ed1, ed2\}) \land \tag{B.30}$$
$$e \in modify(content(s), \{split?, join?, end?, ch2\} \cup events?, \{ed1, ed2\}) \land (atom\ e).type \in \text{ran } eerror \bullet e\} =$$
$$\{s, e : Element \mid s \in cont(proc, \{ed1, ed2\}) \land e \in content(s) \land (atom\ e).type \in \text{ran } eerror \bullet e\}$$

We consider each of the conjuncts individually: Conjunct B.26 follows from the definition of *cont* and schema definition *FullSub*. Conjunct B.27 follows from the memberships $proc \in noOverLap$ and $(proc, \{ed1, ed2\}) \in together$. Conjunct B.28 follows from the following constraints:

$$getIns\{ed1\} = getIns\{split?\}$$
$$getOuts\{split?\} = getIns(events?)$$
$$getOuts(events?) = getIns\{end?\}) \cup \{connect?\}$$
$$getIns\{join?\} = \{connect?\} \cup getIns\{ed2\}$$
$$getOuts\{join?\} = getIns\{ch2\}$$
$$getOuts\{ch2\} = getOuts\{ed2\}$$
$$getSeqflows\{end?\} \cap getSeqflows\ proc = \emptyset$$
$$((outs\ split?) \cup getOuts(events?) \cup \{t2?\}) \cap getSeqflows\ proc = \emptyset$$
$$proc \in noOverLap$$

Conjunct B.29 follows from $getMsgs(\{split?, join?, end?\} \cup events?) = \emptyset$, $(getMsgs\{ed2\} = getMsgs\{ch2\}$ and $proc \in noOverLap$. Conjunct B.30 follows from $\{e : (\{ed1, ed2, end?, split?, join?, ch2\} \cup events?) \bullet (atom\ e).type\} \cap \text{ran } eerror = \emptyset$ We therefore conclude that $md \in \{proc : Process \mid Pool\}$ follows directly from constraints on the before state *Pool* and the input components. We apply set theory to obtain the final simplified precondition schema, labelled *PreEventLoop*.

PreEventLoop

$CommonConstraints[split?/new?]$
$EventLoopEvents$
$ConnectJoin0$
$end? : Element$

$f2? \in \mathrm{dom}(nonsends\ proc)$
$split? \in \{EventSplit \bullet ele\}$
$end? \in \{ele : Element \mid InitialEnd\}$
$getIns(events?) = (atom\ split?).out$
$getOuts\ events? = \{connect?\} \cup (atom\ end?).in$
$(\{t2?\} \cup getSeqflows\ events?) \cap getSeqflows\ proc = \emptyset$
$(\{connect?\} \cup (atom\ end?).in) \cap getIns\ events? = \emptyset$
$from? \neq f2?$
$t2? \notin getSeqflows(events?)$
$connect? \notin (atom\ end?).in$
$((nonsends\ proc)\ f2?) \notin \{Start \bullet ele\}$
$(proc, \{((ends\ proc)\ from?), ((nonsends\ proc)\ f2?)\}) \in together$
$((nonsends\ proc)\ f2?, (ends\ proc)\ from?) \in (edge(direct(proc, (nonsends\ proc)\ f2?)))^+$

□

B.15 Precondition of *AddException*

pre *AddException*

$\Leftrightarrow$ [def of *AddException*, properties of hiding and pre]

pre $(AddNoRelatedErrorException \setminus (change, change'))$ $\vee$

pre $(AddRelatedErrorException \setminus (change, change'))$

We consider the simplification of each precondition individually.

B.15.1 Precondition of *AddNoRelatedErrorException*

We first consider the precondition schema pre $(AddNoRelatedErrorException \setminus (change, change'))$. We expand the definitions, apply schema quantification, and one-point rule to arrive at the following precondition schema.

$[Pool;\ etype? : Type;\ eflow?, loc? : Seqflow;\ end? : Element \mid$
let $ed == (activities\ proc)\ loc? \bullet$
let $ch == ce(ed, (eflow?, etype?), \emptyset, \emptyset) \bullet$
let $md == modify(proc, \{end?, ch\}, \{ed\}) \bullet$
$\{ed, ch\} \subseteq \{ele : Element \mid Activity\} \wedge$
$etype? \in nomsgserrors \wedge$
$eflow? \notin getSeqflows\ proc \wedge$
$end? \in \{InitialEnd \bullet ele\} \wedge$
$(atom\ end?).in = \{eflow?\} \wedge$
$ed \in Element \wedge$
$md \in \{proc : Process \mid Pool\}]$

By pre *AddNoRelatedErrorExceptionSub* calculated in Section B.3 we can see $ch \in \{ele : Element \mid Activity\}$ follows from constraints on the before state *Pool* and the input components. We now consider the conjunct $md \in \{proc : Process \mid Pool\}$. We first expand the definition of md:

$$md = \begin{cases} (proc \cup \{end?, ch\}) \setminus \{ed\} & \text{if } ed \in proc \\ ((proc \setminus cont(proc, \{ed\})) \cup & \text{otherwise} \\ \quad \{s : cont(proc, \{ed\}) \bullet rep(s, modify(content(s), \{end?, ch\}, \{ed\}))\}) & \end{cases}$$

For the case $ed \in proc$, we show $md \in \{proc : Process \mid Pool\}$ follows directly from constraints on the before state $Pool$ and the input components, where $md == (proc \cup \{end?, ch\}) \setminus \{ed\}$. We expand this membership.

$$
\begin{array}{l}
(\exists\, e, f : md \bullet e \in \{Start \bullet ele\} \land f \in \{End \bullet ele\}) \land \\
(\neg\, (\exists\, e : md \bullet e \in \{Inter \mid (atom\ ele).type \in \operatorname{ran} ierror \bullet ele\})) \land \\
(\forall\, e : \{g : Element \mid g \in_{\mathrm{p}} md\} \bullet e \in \{FlowObject \bullet ele\}) \land \\
md \in processSet \land \\
md \in noOverLap \land \\
(\forall\, f : \{g : Element \mid g \in_{\mathrm{p}} md\} \bullet f \in eventgate \Rightarrow \\
\quad (\forall\, e : Element \bullet (f, e) \in edge(direct(md, e)) \Rightarrow e \in sendelement))
\end{array}
$$

We consider the first two conjuncts.

$$
\begin{array}{ll}
(atom\ ch).type = (atom\ ed).type \land & [\text{def of } ed,\ ch,\ end?] \\
ed \notin \{Activity \bullet ele\} \land & \\
end? \in \{End \bullet ele\} & \\
\Rightarrow (\exists\, e, f : md \bullet e \in \{Start \bullet ele\} \land f \in \{End \bullet ele\}) \land & [proc \in \{GenProc \bullet proc\}] \\
\quad (\neg\, (\exists\, e : md \bullet e \in \{Inter \mid (atom\ ele).type \in \operatorname{ran} ierror \bullet ele\})) &
\end{array}
$$

We now consider the third conjunct.

$$
\begin{array}{ll}
end? \in \{ele : Element \mid End\} \land & \\
ch \in \{ele : Element \mid Activity\} & \\
\Rightarrow & [\text{def of } GenProc,\ ch \in \{ele : Element \mid Activity\} \text{ and } InitialEnd \Rightarrow FlowObject] \\
\quad (\forall\, e : \{g : Element \mid g \in_{\mathrm{p}} md\} \bullet e \in \{FlowObject \bullet ele\}) &
\end{array}
$$

We now consider the fourth conjunct.

$$
\begin{array}{ll}
(atom\ end?).in = \{eflow?\} \land & [\text{def of } end?,\ ch \text{ and } ed] \\
outs(ch) = \{eflow?\} \cup outs(ed) \land & \\
(atom\ ch).out = (atom\ ed).out \land & \\
(atom\ ch).in = (atom\ ed).in \land & \\
getMsgs\{end?\} = \emptyset \land & \\
getMsgs\{ch\} = getMsgs\{ed\} \land & \\
getSeqflows(content(ch)) = getSeqflows(content(ed)) \land & \\
eflow? \notin getSeqflows\ proc \land & \\
\Rightarrow (\forall\, e : \{g : Element \mid g \in_{\mathrm{p}} md\} \bullet & [proc \in noOverLap] \\
\quad \bigcup\{k : md \bullet (atom\ k).in \cup outs(k)\} \cap \bigcup\{k : content\ e \bullet (atom\ k).in \cup outs(k)\} = \emptyset \land & \\
\quad (\forall\, f : \{g : Element \mid g \in_{\mathrm{p}} md\} \bullet e \neq f \Rightarrow & \\
\qquad (atom\ e).in \cap (atom\ f).in = \emptyset \land & \\
\qquad outs(e) \cap outs(f) = \emptyset \land & \\
\qquad \bigcup\{k : content\ e \bullet (atom\ k).in \cup outs(k)\} \cap & \\
\qquad \bigcup\{k : content\ f \bullet (atom\ k).in \cup outs(k)\} = \emptyset \land & \\
\qquad getMsg\ (atom\ e) \cap getMsg\ (atom\ f) = \emptyset)) & \\
\Leftrightarrow md \in noOverLap & [\text{def of } noOverLap]
\end{array}
$$

We now consider the fifth conjunct.

$$
end? \in \{End \bullet ele\} \land \qquad [\text{def of } end?,\ ch \text{ and } ed]
$$

$$
\begin{array}{l}
(((edge\ proc)^+)^\sim)(\!|\ \{ed\}\ |\!) = (((edge\ md)^+)^\sim)(\!|\ \{ch\}\ |\!) \wedge \\
(((edge\ proc)^+)(\!|\ \{ed\}\ |\!)) \cup \{end?\} = ((edge\ md)^+)(\!|\ \{ch\}\ |\!) \wedge \\
\Rightarrow (\forall\, e : md \bullet \qquad\qquad [proc \in processSet] \\
\quad (e \in \{\, Start \bullet ele \,\} \Rightarrow (\exists f : md \bullet f \in \{\, End \bullet ele \,\} \wedge (e, f) \in (edge\ md)^+)) \wedge \\
\quad (e \in \{\, End \bullet ele \,\} \Rightarrow (\exists f : md \bullet f \in \{\, Start \bullet ele \,\} \wedge (f, e) \in (edge\ md)^+))) \\
\quad (\forall\, e : p \bullet \\
\quad (\forall\, s : outs(e) \bullet \exists f : p \bullet s \in (atom\ f).in) \wedge \\
\quad (\forall\, s : (atom\ e).in \bullet \exists f : p \bullet s \in outs(f))) \\
\Leftrightarrow md \in processSet \qquad\qquad [\text{def of } processSet]
\end{array}
$$

We now consider the sixth conjunct.

$$
\begin{array}{l}
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad [\text{def of } ed \text{ and } end?] \\
(((edge\ proc)^\sim)(\!|\ \{ed\}\ |\!) \cup end?) \cap eventgate = \emptyset \wedge \\
\Rightarrow (\forall f : \{g : Element \mid g \in_{\mathrm{p}} md\} \bullet f \in eventgate \Rightarrow \qquad [proc \in \{proc : Process \mid hasExgates\}] \\
\quad (\forall\, e : Element \bullet (f, e) \in edge(direct(md, e)) \Rightarrow e \in sendelement)) \\
\Rightarrow md \in \{proc : Process \mid hasExgates\} \qquad\qquad [\text{def of } hasExgates]
\end{array}
$$

We therefore conclude that $md \in \{proc : Process \mid Pool\}$ follows directly from constraints on the before state *Pool* and the input components.
For case $ed \notin proc$, we consider the following inductive hypothesis

$$
\begin{array}{l}
\{s : cont(proc, \{ed\}) \bullet modify(content(s), \{end?, ch\}, \{ed\})\} \subseteq \{proc : Process \mid Pool\}) \Rightarrow \\
((proc \setminus cont(proc, \{ed\})) \cup \\
\{s : cont(proc, \{ed\}) \bullet rep(s, modify(content(s), \{end?, ch\}, \{ed\}))\}) \in \{proc : Process \mid Pool\})
\end{array}
$$

According to Lemma B.2, it is sufficient to deduce the following set of facts (conjuncts) about the precondition schema.

$$cont(proc, \{ed\}) \subseteq \operatorname{ran} compound \tag{B.31}$$

$$\#cont(proc, \{ed\}) = 1 \tag{B.32}$$

$$
\begin{array}{l}
\bigcup \{s : cont(proc, \{ed\}) \bullet getSeqflows(modify(content(s), \{end?, ch\}, \{ed\}))\} \cap \\
\quad getSeqflows(proc \setminus \{s, e : Element \mid s \in cont(proc, \{ed\}) \wedge e \in_{\mathrm{e}} s \bullet e\}) = \emptyset
\end{array} \tag{B.33}
$$

$$
\begin{array}{l}
\bigcup \{s : cont(proc, \{ed\}) \bullet getMsgs(modify(content(s), \{end?, ch\}, \{ed\}))\} \cap \\
\quad getMsgs(proc \setminus \{s, e : Element \mid s \in cont(proc, \{ed\}) \wedge e \in_{\mathrm{e}} s \bullet e\}) = \emptyset
\end{array} \tag{B.34}
$$

$$
\begin{array}{l}
\{s, e : Element \mid s \in cont(proc, \{ed\}) \wedge \\
\quad e \in modify(content(s), \{end?, ch\}, \{ed\}) \wedge (atom\ e).type \in \operatorname{ran} eerror \bullet e\} = \\
\quad \{s, e : Element \mid s \in cont(proc, \{ed\}) \wedge e \in content(s) \wedge (atom\ e).type \in \operatorname{ran} eerror \bullet e\}
\end{array} \tag{B.35}
$$

Conjunct B.31 follows from the definition of *cont* and schema definition *FullSub*. Conjunct B.32 follows from the membership $proc \in noOverLap$. Conjunct B.33 follows from the following constraints.

$$
\begin{array}{l}
getIns\{ed\} = getIns\{ch\} \\
outs(ch) = outs(ed) \cup \{eflow?\} \\
getIns\{end?\} = \{eflow?\} \\
eflows? \notin getSeqflows\ proc = \emptyset \\
proc \in noOverLap
\end{array}
$$

Conjunct B.34 follows from $getMsgs\{ed\} = getMsgs\{ch\}$ and $proc \in noOverLap$, and Conjunct B.35 follows from $\{(atom\ end?).type, (atom\ ch).type\} \cap \operatorname{ran} eerror = \emptyset$. We conclude that $md \in \{proc : Process \mid$

Pool} We therefore obtain the final simplified precondition schema, labelled *PreAddNoRelatedErrorException*.

PreAddNoRelatedErrorException

$$
\begin{array}{l}
Pool \\
etype? : Type \\
eflow?, loc? : Seqflow \\
end? : Element \\
\hline
loc? \in \mathrm{dom}(activities\ proc) \\
etype? \in nomsgserrors \\
eflow? \notin getSeqflows\ proc \\
end? \in \{InitialEnd \bullet ele\} \\
(atom\ end?).in = \{eflow?\}
\end{array}
$$

B.15.2 Precondition of *AddRelatedErrorException*

We now consider the precondition schema pre $(AddRelatedErrorException \setminus (change, change'))$. We expand the definitions, apply schema quantification, and one-point rule to arrive at the following precondition schema.

$$
\begin{array}{l}
[Pool;\ type?, etype? : Type;\ sflow?, eflow?, loc? : Seqflow;\ end? : Element \mid \\
\quad \textbf{let}\ ed == (subs\ proc)\ loc? \bullet \\
\quad\quad \textbf{let}\ en0 == (ends\ (content\ ed))\ sflow? \bullet \\
\quad\quad\quad \textbf{let}\ en1 == ct(en0, type?) \bullet \\
\quad\quad\quad\quad \textbf{let}\ ch == ce(ed, (eflow?, etype?), \{en0\}, \{en1\}) \bullet \\
\quad\quad\quad\quad\quad \textbf{let}\ md == modify(proc, \{end?, ch\}, \{ed\}) \bullet \\
\quad \{ch, ed\} \subseteq \{ele : Element \mid FullSub\} \land \\
\quad \{en0, en1\} \subseteq \{ele : Element \mid End\} \land \\
\quad md \in \{proc : Process \mid Pool\} \land \\
\quad eflow? \notin getSeqflows\ proc \land \\
\quad end? \in \{InitialEnd \bullet ele\} \land \\
\quad (atom\ end?).in = \{eflow?\} \land \\
\quad \{en0, en1, ed, ch\} \subseteq Element \land \\
\quad type? \in \mathrm{ran}\ eerror \cap errorCodeTypes \land \\
\quad (atom\ en0).type = end \land \\
\quad etype? \in ((\mathrm{ran}\ ierror \cap errorCodeTypes) \setminus (\mathrm{ran}((atom\ ed).exit))) \land \\
\quad errorCode\ etype? = errorCode\ type? \land \\
\quad en0 \in (content\ ed)]
\end{array}
$$

By Lemma B.1 we can see that the membership $ed \in \{ele : Element \mid FullSub\}$ follows directly from constraints on the before state *Pool* and the input components. Similarly using Lemma B.1, the constraint $ed \in \{ele : Element \mid FullSub\}$ and definition of *ends*, we can also see that the membership $en0 \in \{ele : Element \mid End\}$ follows directly from constraints on the before state *Pool* and the input components. By pre *ChangeEndType* calculated in Section B.4 we conclude that the membership $en1 \in \{ele : Element \mid End\}$ follows directly from constraints on the before state *Pool* and the input components.

We now consider the membership $ch \in \{ele : Element \mid FullSub\}$. We first expand the constraint on the membership.

$$
\begin{array}{l}
ch \in \mathrm{ran}\ compound \land \\
\#(atom\ ch).send + \#(atom\ ch).receive = 0 \land \\
(atom\ ch).type \in \mathrm{ran}\ subprocess \cup \mathrm{ran}\ mipars \cup \mathrm{ran}\ miseqs \land \\
(\forall\ e, f : \{g : Element \mid g \in_{\mathrm{p}} content(ch)\} \bullet e \neq f \Rightarrow \\
\quad (atom\ e).in \cap (atom\ f).in = \emptyset \land \\
\quad outs(e) \cap outs(f) = \emptyset \land
\end{array}
$$

$$
\begin{array}{l}
\quad \bigcup\{k : content\ e \bullet (atom\ k).in \cup outs(k)\} \cap \bigcup\{k : content\ f \bullet (atom\ k).in \cup outs(k)\} = \emptyset \wedge \\
\quad getMsg\ (atom\ e) \cap getMsg\ (atom\ f) = \emptyset) \wedge \\
(\forall\, e : content(ch) \bullet \\
\quad (e \notin \{\, End \bullet ele \,\} \Rightarrow (\exists f : content(ch) \bullet f \in \{\, End \bullet ele \,\} \wedge (e, f) \in (edge\ content(ch))^+)) \wedge \\
\quad (e \notin \{\, Start \bullet ele \,\} \Rightarrow (\exists f : content(ch) \bullet f \in \{\, Start \bullet ele \,\} \wedge (f, e) \in (edge\ content(ch))^+))) \wedge \\
(\forall\, e : content(ch) \bullet \\
\quad (\forall\, s : outs(e) \bullet \exists f : content(ch) \bullet s \in (atom\ f).in) \wedge \\
\quad (\forall\, s : (atom\ e).in \bullet \exists f : content(ch) \bullet s \in outs(f))) \wedge \\
(\exists\, e, f : content(ch) \bullet e \in \{Start \bullet ele\} \wedge f \in \{End \bullet ele\}) \wedge \\
(\neg\, (\exists\, e : content(ch) \bullet e \in \{Inter \mid (atom\ ele).type \in \mathrm{ran}\ ierror \bullet ele\})) \wedge \\
\#(atom\ ch).in = 1 \wedge \#(atom\ ch).out = 1 \wedge \\
(\forall\, e, f : content(ch) \bullet \\
\quad \{e, f\} \subseteq \{End \mid (atom\ ele).type \in (\mathrm{ran}\ eerror \cap errorCodeTypes) \bullet ele\} \wedge \\
\quad errorCode((atom\ e).type) = errorCode((atom\ f).type) \Rightarrow f = e) \wedge \\
(\forall\, e : content(ch) \bullet \\
\quad (atom\ e).type \in (\mathrm{ran}\ eerror \cap errorCodeTypes) \Rightarrow \\
\qquad (\exists\, t : \mathrm{ran}((atom\ ch).exit) \bullet \\
\qquad\quad t \in (\mathrm{ran}\ ierror \cap errorCodeTypes) \wedge \\
\qquad\quad errorCode((atom\ e).type) = errorCode(t))) \wedge \\
(\forall\, t : \mathrm{ran}((atom\ ch).exit) \bullet \\
\quad (t \in (\mathrm{ran}\ ierror \cap errorCodeTypes) \Rightarrow \\
\qquad (\exists\, e : content(ch) \bullet \\
\qquad\quad (atom\ e).type \in (\mathrm{ran}\ eerror \cap errorCodeTypes) \wedge \\
\qquad\quad errorCode((atom\ e).type) = errorCode(t)))) \wedge \\
(\forall\, t : Type \bullet t \in \mathrm{ran}((atom\ ch).exit) \Rightarrow t \in inters)
\end{array}
$$

We first consider the first three conjuncts.

$$
\begin{array}{ll}
ed \in \{ele : Element \mid FullSub\} & \\
\Rightarrow ed \in \mathrm{ran}\ compound \wedge & [\text{def of } FullSub] \\
\quad \#(atom\ ed).send + \#(atom\ ed).receive = 0 \wedge & \\
\quad (atom\ ed).type \in \mathrm{ran}\ subprocess \cup \mathrm{ran}\ mipars \cup \mathrm{ran}\ miseqs \cup \mathrm{ran}\ sloops & \\
\Rightarrow ch \in \mathrm{ran}\ compound \wedge & [\text{def of } ce \text{ (only } (atom\ ed).exit \text{ and } content(ed) \text{ are modified)}] \\
\quad \#(atom\ ch).send + \#(atom\ ch).receive = 0 \wedge & \\
\quad (atom\ ch).type \in \mathrm{ran}\ subprocess \cup \mathrm{ran}\ mipars \cup \mathrm{ran}\ miseqs \cup \mathrm{ran}\ sloops &
\end{array}
$$

We now consider the fourth conjunct.

$$
\begin{array}{ll}
ed \in \{ele : Element \mid FullSub\} & \\
\Rightarrow (\forall\, e, f : \{g : Element \mid g \in_{\mathrm{p}} content(ed)\} \bullet e \neq f \Rightarrow & [\text{def of } FullSub] \\
\quad (atom\ e).in \cap (atom\ f).in = \emptyset \wedge & \\
\quad outs(e) \cap outs(f) = \emptyset \wedge & \\
\quad \bigcup\{k : content\ e \bullet (atom\ k).in \cup outs(k)\} \cap \bigcup\{k : content\ f \bullet (atom\ k).in \cup outs(k)\} = \emptyset \wedge & \\
\quad getMsg\ (atom\ e) \cap getMsg\ (atom\ f) = \emptyset) & \\
\Rightarrow & [\text{def of } ct \text{ (only } type \text{ component of } en0 \text{ is changed)}] \\
\quad (\forall\, e, f : \{g : Element \mid g \in_{\mathrm{p}} content(ch)\} \bullet e \neq f \Rightarrow & \\
\quad (atom\ e).in \cap (atom\ f).in = \emptyset \wedge &
\end{array}
$$

$$
\begin{array}{l}
\quad \mathit{outs}(e) \cap \mathit{outs}(f) = \emptyset \land \\
\quad \bigcup\{k : \mathit{content}\, e \bullet (\mathit{atom}\, k).\mathit{in} \cup \mathit{outs}(k)\} \cap \bigcup\{k : \mathit{content}\, f \bullet (\mathit{atom}\, k).\mathit{in} \cup \mathit{outs}(k)\} = \emptyset \land \\
\quad \mathit{getMsg}\,(\mathit{atom}\, e) \cap \mathit{getMsg}\,(\mathit{atom}\, f) = \emptyset)
\end{array}
$$

We now consider the fifth conjunct.

$$
\begin{array}{l}
\quad \mathit{ed} \in \{\mathit{ele} : \mathit{Element} \mid \mathit{FullSub}\} \\
\Rightarrow (\forall\, e : \mathit{content}(\mathit{ed}) \bullet \qquad \text{[def of } \mathit{FullSub}\text{]} \\
\quad\quad (e \notin \{\, \mathit{End} \bullet \mathit{ele}\,\} \Rightarrow (\exists f : \mathit{content}(\mathit{ed}) \bullet f \in \{\, \mathit{End} \bullet \mathit{ele}\,\} \land (e, f) \in (\mathit{edge}\ \mathit{content}(\mathit{ed}))^+)) \land \\
\quad\quad (e \notin \{\, \mathit{Start} \bullet \mathit{ele}\,\} \Rightarrow (\exists f : \mathit{content}(\mathit{ed}) \bullet f \in \{\, \mathit{Start} \bullet \mathit{ele}\,\} \land (f, e) \in (\mathit{edge}\ \mathit{content}(\mathit{ed}))^+))) \\
\Rightarrow (\forall\, e : \mathit{content}(\mathit{ch}) \bullet \qquad \text{[def of } \mathit{ct} \text{ (only } \mathit{type} \text{ component of } \mathit{en}0 \text{ is changed)]} \\
\quad\quad (e \notin \{\, \mathit{End} \bullet \mathit{ele}\,\} \Rightarrow (\exists f : \mathit{content}(\mathit{ch}) \bullet f \in \{\, \mathit{End} \bullet \mathit{ele}\,\} \land (e, f) \in (\mathit{edge}\ \mathit{content}(\mathit{ch}))^+)) \land \\
\quad\quad (e \notin \{\, \mathit{Start} \bullet \mathit{ele}\,\} \Rightarrow (\exists f : \mathit{content}(\mathit{ch}) \bullet f \in \{\, \mathit{Start} \bullet \mathit{ele}\,\} \land (f, e) \in (\mathit{edge}\ \mathit{content}(\mathit{ch}))^+)))
\end{array}
$$

We now consider the sixth conjunct.

$$
\begin{array}{l}
\quad \mathit{ed} \in \{\mathit{ele} : \mathit{Element} \mid \mathit{FullSub}\} \\
\Rightarrow (\forall\, e : \mathit{content}(\mathit{ed}) \bullet \\
\quad\quad (\forall\, s : \mathit{outs}(e) \bullet \exists f : \mathit{content}(\mathit{ed}) \bullet s \in (\mathit{atom}\, f).\mathit{in}) \land \\
\quad\quad (\forall\, s : (\mathit{atom}\, e).\mathit{in} \bullet \exists f : \mathit{content}(\mathit{ed}) \bullet s \in \mathit{outs}(f))) \\
\Rightarrow \qquad \text{[def of } \mathit{ct} \text{ (only } \mathit{type} \text{ component of } \mathit{en}0 \text{ is changed and } (\mathit{atom}\, \mathit{en}1).\mathit{type} \in \operatorname{ran} \mathit{eerror}\text{]} \\
\quad (\forall\, e : \mathit{content}(\mathit{ch}) \bullet \\
\quad\quad (\forall\, s : \mathit{outs}(e) \bullet \exists f : \mathit{content}(\mathit{ch}) \bullet s \in (\mathit{atom}\, f).\mathit{in}) \land \\
\quad\quad (\forall\, s : (\mathit{atom}\, e).\mathit{in} \bullet \exists f : \mathit{content}(\mathit{ch}) \bullet s \in \mathit{outs}(f)))
\end{array}
$$

We now consider seventh, eighth and ninth conjuncts.

$$
\begin{array}{l}
\quad \mathit{ed} \in \{\mathit{ele} : \mathit{Element} \mid \mathit{FullSub}\} \\
\Rightarrow (\exists\, e, f : \mathit{content}(\mathit{ed}) \bullet e \in \{\mathit{Start} \bullet \mathit{ele}\} \land f \in \{\mathit{End} \bullet \mathit{ele}\}) \land \qquad \text{[def of } \mathit{FullSub}\text{]} \\
\quad (\neg\, (\exists\, e : \mathit{content}(\mathit{ed}) \bullet e \in \{\mathit{Inter} \mid (\mathit{atom}\ \mathit{ele}).\mathit{type} \in \operatorname{ran} \mathit{ierror} \bullet \mathit{ele}\,\})) \\
\Rightarrow \qquad \text{[def of } \mathit{ct} \text{ } ((\mathit{atom}\ \mathit{en}0).\mathit{type} = \mathit{end},\ (\mathit{atom}\ \mathit{en}1).\mathit{type} \in \operatorname{ran} \mathit{eerror} \text{ and } \mathit{en}1 \in \{\mathit{End} \bullet \mathit{ele}\})\text{]} \\
\quad (\exists\, e, f : \mathit{content}(\mathit{ch}) \bullet e \in \{\mathit{Start} \bullet \mathit{ele}\} \land f \in \{\mathit{End} \bullet \mathit{ele}\}) \land \\
\quad (\neg\, (\exists\, e : \mathit{content}(\mathit{ch}) \bullet e \in \{\mathit{Inter} \mid (\mathit{atom}\ \mathit{ele}).\mathit{type} \in \operatorname{ran} \mathit{ierror} \bullet \mathit{ele}\,\}))
\end{array}
$$

We now consider the tenth conjunct.

$$
\begin{array}{l}
\quad \mathit{ed} \in \{\mathit{ele} : \mathit{Element} \mid \mathit{FullSub}\} \\
\Rightarrow \#(\mathit{atom}\ \mathit{ed}).\mathit{in} = 1 \land \#(\mathit{atom}\ \mathit{ed}).\mathit{out} = 1 \qquad \text{[def of } \mathit{FullSub}\text{]} \\
\Rightarrow \#(\mathit{atom}\ \mathit{ch}).\mathit{in} = 1 \land \#(\mathit{atom}\ \mathit{ch}).\mathit{out} = 1 \quad \text{[def of } \mathit{ce} \text{ (components } \mathit{in} \text{ and } \mathit{out} \text{ are unchanged)]}
\end{array}
$$

We now consider eleventh and twelveth conjuncts

$$
\begin{array}{l}
\quad \mathit{ed} \in \{\mathit{ele} : \mathit{Element} \mid \mathit{FullSub}\} \\
\Rightarrow (\forall\, e, f : \mathit{content}(\mathit{ed}) \bullet \\
\quad\quad \{e, f\} \subseteq \{\mathit{End} \mid (\mathit{atom}\ \mathit{ele}).\mathit{type} \in (\operatorname{ran} \mathit{eerror} \cap \mathit{errorCodeTypes}) \bullet \mathit{ele}\} \land \\
\quad\quad \mathit{errorCode}((\mathit{atom}\ e).\mathit{type}) = \mathit{errorCode}((\mathit{atom}\, f).\mathit{type}) \Rightarrow f = e) \\
\Rightarrow (\forall\, e, f : \mathit{content}(\mathit{ch}) \bullet \qquad \text{[def of } \mathit{ce} \text{ } (\mathit{errorCode}((\mathit{atom}\ \mathit{en}1).\mathit{type}) = \mathit{errorCode}(\mathit{etype}?)\text{]} \\
\quad\quad \{e, f\} \subseteq \{\mathit{End} \mid (\mathit{atom}\ \mathit{ele}).\mathit{type} \in (\operatorname{ran} \mathit{eerror} \cap \mathit{errorCodeTypes}) \bullet \mathit{ele}\} \land \\
\quad\quad \mathit{errorCode}((\mathit{atom}\ e).\mathit{type}) = \mathit{errorCode}((\mathit{atom}\, f).\mathit{type}) \Rightarrow f = e)
\end{array}
$$

We now consider thirteenth and fourteenth conjuncts.

$$
\begin{array}{l}
ed \in \{ele : Element \mid FullSub\} \\
\Rightarrow (\forall\, e : content(ed) \bullet \\
\quad (atom\ e).type \in (\mathrm{ran}\ eerror \cap errorCodeTypes) \Rightarrow \\
\quad\quad (\exists\, t : \mathrm{ran}((atom\ ed).exit) \bullet \\
\quad\quad\quad t \in (\mathrm{ran}\ ierror \cap errorCodeTypes) \wedge \\
\quad\quad\quad errorCode((atom\ e).type) = errorCode(t))) \wedge \\
\quad (\forall\, t : \mathrm{ran}((atom\ ed).exit) \bullet \\
\quad\quad (t \in (\mathrm{ran}\ ierror \cap errorCodeTypes) \Rightarrow \\
\quad\quad\quad (\exists\, e : content(ed) \bullet \\
\quad\quad\quad\quad (atom\ e).type \in (\mathrm{ran}\ eerror \cap errorCodeTypes) \wedge \\
\quad\quad\quad\quad errorCode((atom\ e).type) = errorCode(t)))) \\
\Rightarrow \qquad\qquad\qquad\qquad\qquad\qquad [\text{def of } ce\ (errorCode((atom\ en1).type) = errorCode(etype?)\ \text{and}] \\
\qquad\qquad\qquad\qquad\qquad\qquad [etype? \in ((\mathrm{ran}\ ierror \cap errorCodeTypes) \setminus (\mathrm{ran}((atom\ ed).exit)))] \\
\quad (\forall\, e : content(ch) \bullet \\
\quad\quad (atom\ e).type \in (\mathrm{ran}\ eerror \cap errorCodeTypes) \Rightarrow \\
\quad\quad\quad (\exists\, t : \mathrm{ran}((atom\ ch).exit) \bullet \\
\quad\quad\quad\quad t \in (\mathrm{ran}\ ierror \cap errorCodeTypes) \wedge \\
\quad\quad\quad\quad errorCode((atom\ e).type) = errorCode(t))) \wedge \\
\quad (\forall\, t : \mathrm{ran}((atom\ ch).exit) \bullet \\
\quad\quad (t \in (\mathrm{ran}\ ierror \cap errorCodeTypes) \Rightarrow \\
\quad\quad\quad (\exists\, e : content(ch) \bullet \\
\quad\quad\quad\quad (atom\ e).type \in (\mathrm{ran}\ eerror \cap errorCodeTypes) \wedge \\
\quad\quad\quad\quad errorCode((atom\ e).type) = errorCode(t))))
\end{array}
$$

We now consider fifteenth conjunct.

$$
\begin{array}{l}
ed \in \{ele : Element \mid FullSub\} \\
\Rightarrow (\forall\, t : Type \bullet t \in \mathrm{ran}((atom\ ed).exit) \Rightarrow t \in inters) \\
\Rightarrow (\forall\, t : Type \bullet t \in \mathrm{ran}((atom\ ch).exit) \Rightarrow t \in inters) \qquad [etype? \in (\mathrm{ran}\ ierror \cap errorCodeTypes)]
\end{array}
$$

We therefore conclude that the membership $ch \in \{ele : Element \mid FullSub\}$ follows directly from constraints on the before state $Pool$ and the input components. We now consider the conjunct $md \in \{proc : Process \mid Pool\}$. We first expand the definition of md:

$$
md = \begin{cases}
(proc \cup \{end?, ch\}) \setminus \{ed\} & \text{if } ed \in proc \\
((proc \setminus cont(proc, \{ed\})) \cup & \text{otherwise} \\
\quad \{s : cont(proc, \{ed\}) \bullet rep(s, modify(content(s), \{end?, ch\}, \{ed\}))\}) &
\end{cases}
$$

For the case $ed \in proc$, we show $md \in \{proc : Process \mid Pool\}$ follows directly from constraints on the before state $Pool$ and the input components, where $md == (proc \cup \{end?, ch\}) \setminus \{ed\}$. We expand this membership.

$$
\begin{array}{l}
(\exists\, e, f : md \bullet e \in \{Start \bullet ele\} \wedge f \in \{End \bullet ele\}) \wedge \\
(\neg\, (\exists\, e : md \bullet e \in \{Inter \mid (atom\ ele).type \in \mathrm{ran}\ ierror \bullet ele\})) \wedge \\
(\forall\, e : \{g : Element \mid g \in_{\mathrm{p}} md\} \bullet e \in \{FlowObject \bullet ele\}) \wedge \\
md \in processSet \wedge \\
md \in noOverLap \wedge
\end{array}
$$

$$(\forall f : \{g : Element \mid g \in_{\mathrm{p}} md\} \bullet f \in eventgate \Rightarrow$$
$$(\forall e : Element \bullet (f, e) \in edge(direct(md, e)) \Rightarrow e \in sendelement))$$

We consider the first two conjuncts.

$$(atom\ ch).type = (atom\ ed).type \wedge \qquad \text{[def of } ed, ch \text{ and } end?\text{]}$$
$$(atom\ ed).type \notin \operatorname{ran} ierror \wedge$$
$$end? \in \{End \bullet ele\}$$
$$\Rightarrow (\exists e, f : md \bullet e \in \{Start \bullet ele\} \wedge f \in \{End \bullet ele\}) \wedge \qquad [proc \in \{GenProc \bullet proc\}]$$
$$(\neg (\exists e : md \bullet e \in \{Inter \mid (atom\ ele).type \in \operatorname{ran} ierror \bullet ele\}))$$

We now consider the third conjunct.

$$end? \in \{ele : Element \mid End\} \wedge$$
$$ch \in \{ele : Element \mid Activity\}$$
$$\Rightarrow \qquad \text{[def of } GenProc, ch \in \{ele : Element \mid Activity\} \text{ and } InitialEnd \Rightarrow FlowObject\text{]}$$
$$(\forall e : \{g : Element \mid g \in_{\mathrm{p}} md\} \bullet e \in \{FlowObject \bullet ele\})$$

We now consider the fourth conjunct.

$$(atom\ end?).in = \{eflow?\} \wedge \qquad \text{[def of } end?, ch, ed\text{]}$$
$$outs(ch) = \{eflow?\} \cup outs(ed) \wedge$$
$$(atom\ ch).out = (atom\ ed).out \wedge$$
$$(atom\ ch).in = (atom\ ed).in \wedge$$
$$getMsgs\{end?\} = \emptyset \wedge$$
$$getMsgs\{ch\} = getMsgs\{ed\} \wedge$$
$$getSeqflows(content(ch)) = getSeqflows(content(ed)) \wedge$$
$$eflow? \notin getSeqflows\ proc \wedge$$
$$\Rightarrow (\forall e : \{g : Element \mid g \in_{\mathrm{p}} md\} \bullet \qquad [proc \in noOverLap]$$
$$\bigcup\{k : md \bullet (atom\ k).in \cup outs(k)\} \cap \bigcup\{k : content\ e \bullet (atom\ k).in \cup outs(k)\} = \emptyset \wedge$$
$$(\forall f : \{g : Element \mid g \in_{\mathrm{p}} md\} \bullet e \neq f \Rightarrow$$
$$(atom\ e).in \cap (atom\ f).in = \emptyset \wedge$$
$$outs(e) \cap outs(f) = \emptyset \wedge$$
$$\bigcup\{k : content\ e \bullet (atom\ k).in \cup outs(k)\} \cap$$
$$\bigcup\{k : content\ f \bullet (atom\ k).in \cup outs(k)\} = \emptyset \wedge$$
$$getMsg\ (atom\ e) \cap getMsg\ (atom\ f) = \emptyset))$$
$$\Leftrightarrow md \in noOverLap \qquad \text{[def of } noOverLap\text{]}$$

We now consider the fifth conjunct.

$$end? \in \{End \bullet ele\} \wedge \qquad \text{[def of } ed, end? \text{ and } ch\text{]}$$
$$(((edge\ proc)^+)^\sim)(\!|\ \{ed\}\ |\!) = (((edge\ md)^+)^\sim)(\!|\ \{ch\}\ |\!) \wedge$$
$$(((edge\ proc)^+)(\!|\ \{ed\}\ |\!)) \cup \{end?\} = ((edge\ md)^+)(\!|\ \{ch\}\ |\!) \wedge$$
$$\Rightarrow (\forall e : md \bullet \qquad [proc \in processSet]$$
$$(e \notin \{\ End \bullet ele\ \} \Rightarrow (\exists f : md \bullet f \in \{\ End \bullet ele\ \} \wedge (e, f) \in (edge\ md)^+)) \wedge$$
$$(e \notin \{\ Start \bullet ele\ \} \Rightarrow (\exists f : md \bullet f \in \{\ Start \bullet ele\ \} \wedge (f, e) \in (edge\ md)^+))) \wedge$$
$$(\forall e : md \bullet$$
$$(\forall s : outs(e) \bullet \exists f : md \bullet s \in (atom\ f).in) \wedge$$
$$(\forall s : (atom\ e).in \bullet \exists f : md \bullet s \in outs(f)))$$

$\Leftrightarrow md \in processSet$ [def of *processSet*]

We now consider the sixth conjunct.

[def of *ed* and *end*?]

$(((edge\ proc)^{\sim})(\!|\ \{ed\}\ |\!) \cup end?) \cap eventgate = \emptyset \wedge$

$\Rightarrow (\forall f : \{g : Element \mid g \in_{\mathrm{p}} md\} \bullet f \in eventgate \Rightarrow$ [$proc \in \{proc : Process \mid hasExgates\}$]

$(\forall e : Element \bullet (f, e) \in edge(direct(md, e)) \Rightarrow e \in sendelement))$

$\Rightarrow md \in \{proc : Process \mid hasExgates\}$ [def of *hasExgates*]

We therefore conclude that $md \in \{proc : Process \mid Pool\}$ follows directly from constraints on the before state *Pool* and the input components.
For case $ed \notin proc$, we consider the following inductive hypothesis

$\{s : cont(proc, \{ed\}) \bullet modify(content(s), \{end?, ch\}, \{ed\})\} \subseteq \{proc : Process \mid Pool\}) \Rightarrow$
$((proc \setminus cont(proc, \{ed\})) \cup$
$\{s : cont(proc, \{ed\}) \bullet rep(s, modify(content(s), \{end?, ch\}, \{ed\}))\}) \in \{proc : Process \mid Pool\})$

According to Lemma B.2, it is sufficient to deduce the following set of facts (conjuncts) about the precondition schema.

$$cont(proc, \{ed\}) \subseteq \operatorname{ran} compound \tag{B.36}$$

$$\#cont(proc, \{ed\}) = 1 \tag{B.37}$$

$$\bigcup \{s : cont(proc, \{ed\}) \bullet getSeqflows(modify(content(s), \{end?, ch\}, \{ed\}))\} \cap$$
$$getSeqflows(proc \setminus \{s, e : Element \mid s \in cont(proc, \{ed\}) \wedge e \in_{\mathrm{e}} s \bullet e\}) = \emptyset \tag{B.38}$$

$$\bigcup \{s : cont(proc, \{ed\}) \bullet getMsgs(modify(content(s), \{end?, ch\}, \{ed\}))\} \cap$$
$$getMsgs(proc \setminus \{s, e : Element \mid s \in cont(proc, \{ed\}) \wedge e \in_{\mathrm{e}} s \bullet e\}) = \emptyset \tag{B.39}$$

$$\{s, e : Element \mid s \in cont(proc, \{ed\}) \wedge$$
$$e \in modify(content(s), \{end?, ch\}, \{ed\}) \wedge (atom\ e).type \in \operatorname{ran} eerror \bullet e\} =$$
$$\{s, e : Element \mid s \in cont(proc, \{ed\}) \wedge e \in content(s) \wedge (atom\ e).type \in \operatorname{ran} eerror \bullet e\} \tag{B.40}$$

We consider each of the conjuncts individually: Conjunct B.36 follows from the definition of *cont* and schema definition *FullSub*. Conjunct B.37 follows from the membership $proc \in noOverLap$. Conjunct B.38 follows from the following constraints.

$$getIns\{ed\} = getIns\{ch\}$$
$$outs(ch) = outs(ed) \cup \{eflow?\}$$
$$getIns\{end?\} = \{eflow?\}$$
$$eflows? \notin getSeqflows\ proc = \emptyset$$
$$proc \in noOverLap$$

Conjunct B.39 follows from $getMsgs\{ed\} = getMsgs\{ch\}$ and $proc \in noOverLap$, and Conjunct B.40 follows from $\{(atom\ end?).type, (atom\ ch).type\} \cap \operatorname{ran} eerror = \emptyset$. We conclude that $md \in \{proc : Process \mid Pool\}$ follows directly from constraints on the before state *Pool* and the input components. We therefore obtain the final simplified precondition schema, labelled *PreAddRelatedErrorException*.

PreAddRelatedErrorException

$$
\begin{array}{l}
Pool \\
type?, etype? : Type \\
sflow?, eflow?, loc? : Seqflow \\
end? : Element \\
\hline
loc? \in \mathrm{dom}(subs\ proc) \\
sflow? \in \mathrm{dom}(ends\ (content\ ((subs\ proc)\ loc?))) \\
eflow? \notin getSeqflows\ proc \\
end? \in \{InitialEnd \bullet ele\} \\
(atom\ end?).in = \{eflow?\} \\
etype? \in ((\mathrm{ran}\ ierror \cap errorCodeTypes) \setminus (\mathrm{ran}((atom\ ((subs\ proc)\ loc?)).exit))) \\
type? = eerror(exception(errorCode\ etype?))
\end{array}
$$

The precondition of *AddException*, labelled *PreAddException*, is defined as follows.

$$PreAddException \mathrel{\widehat=} PreAddNoRelatedErrorException \lor PreAddRelatedErrorException$$

□

B.16 Precondition of *ConnectMgeFlowDiagram*

We expand the definitions of pre *ConnectMgeFlowDiagram* and apply the one-point rule to arrive at the following schema.

$$
\begin{array}{l}
\exists\, Diagram' \bullet [\Delta Diagram;\ id1?, id2? : PoolId;\ msg? : Mgeflow;\ tos?, tor? : Seqflow \mid \\
\quad (\textbf{let}\ pc1 == (pool\ id1?).proc \bullet \\
\qquad pool\ id1? = \langle\!|proc \rightsquigarrow pc1|\!\rangle \land \\
\qquad pc1 \in \{proc : Process \mid Pool\} \land \\
\qquad (pool'\ id1? = \langle\!|proc \rightsquigarrow pc2Def(pc1, tos?, msg?)|\!\rangle \lor \\
\qquad pool'\ id1? = \langle\!|proc \rightsquigarrow pc3Def(pc1, tos?, msg?)|\!\rangle)) \land \\
\quad (\textbf{let}\ pc5 == (pool\ id2?).proc \bullet \\
\qquad pool\ id2? = \langle\!|proc \rightsquigarrow pc5|\!\rangle \land \\
\qquad pc5 \in \{proc : Process \mid Pool\} \land \\
\qquad (pool'\ id2? = \langle\!|proc \rightsquigarrow pc6Def(pc5, tor?, msg?)|\!\rangle \lor \\
\qquad pool'\ id2? = \langle\!|proc \rightsquigarrow pc7Def(pc5, tor?, msg?)|\!\rangle \lor \\
\qquad pool'\ id2? = \langle\!|proc \rightsquigarrow pc8Def(pc5, tor?, msg?)|\!\rangle \lor \\
\qquad pool'\ id2? = \langle\!|proc \rightsquigarrow pc9Def(pc5, tor?, msg?)|\!\rangle)) \land \\
\quad id1? \neq id2? \land \\
\quad msg? \notin getMsgs(\bigcup\{p : \mathrm{ran}\ pool \bullet p.proc\}) \land \\
\quad \{id1?, id2?\} \subseteq \mathrm{dom}\ pool \land \\
\quad \{id1?, id2?\} \lhd pool = \{id1?, id2?\} \lhd pool']
\end{array}
$$

For the purpose of exposition we compartmentalize the definition axiomatically. We relegate the elimination of the existential quantification over $Diagram'$ to later in the simplification process. Specifically each of the axiomatic definitions considers a possible after state of one of the two BPMN pools identified by $id1$ and $id2$. Functions $pc2Def$ and $pc3Def$ define the value of component *proc* of the after state $pool'\ id1$. Function $pc2Def$ adds an outgoing message flow to an end event contained in $(pool\ id1).proc$. Function $pc3Def$ adds an outgoing message flow to a task contained in $(pool\ id1).proc$.

$$pc2Def : (Process \times Seqflow \times Mgeflow) \nrightarrow Process$$

$$
\begin{array}{l}
\forall\, pc1 : Process;\ tos : Seqflow;\ msg : Mgeflow \bullet \\
\quad (\textbf{let}\ chs1 == (msgsends\ pc1)\ tos \bullet \\
\qquad \textbf{let}\ chs2 == ct(chs1, emessage(message(msg))) \bullet \\
\qquad\quad \textbf{let}\ pc2 == modify(pc1, \{chs2\}, \{chs1\}) \bullet \\
\quad (\{chs1, chs2\} \subseteq \{ele : Element \mid End\} \land \\
\quad (atom\ chs1).type = emessage(nomessage) \land \\
\quad pc2 \in \{proc : Process \mid Pool\}) \Rightarrow pc2Def(pc1, tos, msg) = pc2)
\end{array}
$$

$$
\begin{array}{|l}
pc3Def : (Process \times Seqflow \times Mgeflow) \nrightarrow Process \\
\hline
\forall\, pc1 : Process;\ tos : Seqflow;\ msg : Mgeflow \bullet \\
\quad (\textbf{let}\ chs3 == (tasks\, pc1)\, tos \bullet \\
\qquad \textbf{let}\ chs4 == cm(chs3, \{msg\}, \emptyset, (atom\ chs3).exit) \bullet \\
\qquad\quad \textbf{let}\ pc3 == modify(pc1, \{chs4\}, \{chs3\}) \bullet \\
\quad (\{chs3, chs4\} \subseteq \{ele : Element \mid FullTask\} \land \\
\quad pc3 \in \{proc : Process \mid Pool\}) \Rightarrow pc3Def(pc1, tos, msg) = pc3)
\end{array}
$$

Functions $pc6Def$, $pc7Def$, $pc8Def$ and $pc9Def$ define the value of component $proc$ of the after state $pool'\ id2$. Function $pc6Def$ adds an incoming message flow to a start event contained in $(pool\ id2).proc$. Function $pc7Def$ adds an incoming message flow to an intermediate event contained in $(pool\ id2).proc$. Function $pc8Def$ adds an incoming message flow to an intermediate message event attached to either a task or a subprocess contained in $(pool\ id2).proc$. Function $pc9Def$ adds an incoming message flow to a task contained in $(pool\ id2).proc$.

$$
\begin{array}{|l}
pc6Def : (Process \times Seqflow \times Mgeflow) \nrightarrow Process \\
\hline
\forall\, pc5 : Process;\ tor : Seqflow;\ msg : Mgeflow \bullet \\
\quad (\textbf{let}\ chr1 == (msgrecs\, pc5)\, tor \bullet \\
\qquad \textbf{let}\ chr2 == ct(chr1, smessage(message(msg))) \bullet \\
\qquad\quad \textbf{let}\ pc6 == modify(pc5, \{chr2\}, \{chr1\}) \bullet \\
\quad (\{chr1, chr2\} \subseteq \{ele : Element \mid Start\} \land \\
\quad (atom\ chr1).type = smessage(nomessage) \land \\
\quad pc6 \in \{proc : Process \mid Pool\}) \Rightarrow pc6Def(pc5, tor, msg) = pc6)
\end{array}
$$

$$
\begin{array}{|l}
pc7Def : (Process \times Seqflow \times Mgeflow) \nrightarrow Process \\
\hline
\forall\, pc5 : Process;\ tor : Seqflow;\ msg : Mgeflow \bullet \\
\quad (\textbf{let}\ chr3 == (msgrecs\, pc5)\, tor \bullet \\
\qquad \textbf{let}\ chr4 == ct(chr3, imessage(message(msg))) \bullet \\
\qquad\quad \textbf{let}\ pc7 == modify(pc5, \{chr4\}, \{chr3\}) \bullet \\
\quad (\{chr3, chr4\} \subseteq \{ele : Element \mid Inter\} \land \\
\quad (atom\ chr3).type = imessage(nomessage) \land \\
\quad pc7 \in \{proc : Process \mid Pool\}) \Rightarrow pc7Def(pc5, tor, msg) = pc7)
\end{array}
$$

$$
\begin{array}{|l}
pc8Def : (Process \times Seqflow \times Mgeflow) \nrightarrow Process \\
\hline
\forall\, pc5 : Process;\ tor : Seqflow;\ msg : Mgeflow \bullet \\
\quad (\textbf{let}\ chr5 == (activities\, pc5)\, tor \bullet \\
\qquad \textbf{let}\ rr1 == rrange((atom\ chr5).exit, imessage(nomessage), imessage(message(msg))) \bullet \\
\qquad\quad \textbf{let}\ chr6 == cm(chr5, \emptyset, \emptyset, rr1) \bullet \\
\qquad\qquad \textbf{let}\ pc8 == modify(pc5, \{chr6\}, \{chr5\}) \bullet \\
\quad (\{chr5, chr6\} \subseteq \{ele : Element \mid Activity\} \land \\
\quad \#(((atom\ chr5).exit) \rhd \{imessage(nomessage)\}) = 1 \land \\
\quad pc8 \in \{proc : Process \mid Pool\}) \Rightarrow pc8Def(pc5, tor, msg) = pc8)
\end{array}
$$

$$
\begin{array}{|l}
pc9Def : (Process \times Seqflow \times Mgeflow) \nrightarrow Process \\
\hline
\forall\, pc5 : Process;\ tor : Seqflow;\ msg : Mgeflow \bullet \\
\quad (\textbf{let}\ chr7 == (tasks\, pc5)\, tor \bullet \\
\qquad \textbf{let}\ chr8 == cm(chr7, \emptyset, \{msg\}, (atom\ chr7).exit) \bullet \\
\qquad\quad \textbf{let}\ pc9 == modify(pc5, \{chr8\}, \{chr7\}) \bullet \\
\quad (\{chr7, chr8\} \subseteq \{ele : Element \mid FullTask\} \land \\
\quad pc9 \in \{proc : Process \mid Pool\}) \Rightarrow pc9Def(pc5, tor, msg) = pc9)
\end{array}
$$

We first consider the constraint $\{chs1, chs2\} \subseteq \{ele : Element \mid End\}$ specified in the definition $pc2Def$.

$$
\begin{array}{ll}
& chs1 = (msgsends\ pc1)\ tos? \land \\
& (atom\ chs1).type = emessage(nomessage) \\
\Rightarrow & chs1 \in \{ele : Element \mid EMgeEvent\} \land & [\text{def of } msgsends] \\
& (atom\ chs1).type = emessage(nomessage) \\
\Rightarrow & chs1 \in \{ele : Element \mid End\} \land & [\text{def of } EMgeEvent] \\
& (atom\ chs1).type = emessage(nomessage) \\
\Rightarrow & \{chs1, chs2\} \subseteq \{ele : Element \mid End\} & [\text{pre } AddMgeEvent]
\end{array}
$$

We can see $\{chs1, chs2\} \subseteq \{ele : Element \mid End\}$ follows directly from constraints on $(pool\ id1?).proc$ and $chs1$.

We now consider the constraint $\{chs3, chs4\} \subseteq \{ele : Element \mid FullTask\}$ specified in the definition $pc3Def$. We notice the following.

$$
\begin{array}{ll}
& chs3 = (tasks\ pc1)\ tos? \land \\
& msg? \notin getMsgs(\bigcup\{p : \mathrm{ran}\ pool \bullet p.proc\}) \\
\Rightarrow & chs3 \in \{ele : Element \mid FullTask\} \land & [\text{def of } tasks \text{ and } getMsgs] \\
& msg? \notin getMsg(atom\ chs3) \\
\Rightarrow & \{chs3, chs4\} \subseteq \{ele : Element \mid FullTask\} & [\text{pre } AddSendMgeFlowTask]
\end{array}
$$

We can see $\{chs3, chs4\} \subseteq \{ele : Element \mid FullTask\}$ follows directly from constraints on $(pool\ id1?).proc$ and $chs3$.

We now consider the constraint $\{chr1, chr2\} \subseteq \{ele : Element \mid Start\}$ specified in the definition $pc6Def$.

$$
\begin{array}{ll}
& chr1 = (msgrecs\ pc5)\ tor? \land \\
& (atom\ chr1).type = smessage(nomessage) \\
\Rightarrow & chr1 \in \{ele : Element \mid SMgeEvent\} & [\text{def of } msgrecs \text{ and } SMgeEvent] \\
& (atom\ chr1).type = smessage(nomessage) \\
\Rightarrow & chr1 \in \{ele : Element \mid Start\} & [\text{def of } SMgeEvent] \\
& (atom\ chr1).type = smessage(nomessage) \\
\Rightarrow & \{chr1, chr2\} \subseteq \{ele : Element \mid Start\} & [\text{pre } AddMgeEvent]
\end{array}
$$

We can see $\{chr1, chr2\} \subseteq \{ele : Element \mid Start\}$ follows directly from constraints on $(pool\ id2?).proc$ and $chr1$.

We now consider the constraint $\{chr3, chr4\} \subseteq \{ele : Element \mid Inter\}$ specified in the definition $pc7Def$.

$$
\begin{array}{ll}
& chr3 = (msgrecs\ pc5)\ tor? \land \\
& (atom\ chr3).type = imessage(nomessage) \\
\Rightarrow & chr3 \in \{ele : Element \mid IMgeEvent\} \land & [\text{def of } msgrecs \text{ and } IMgeEvent] \\
& (atom\ chr3).type = imessage(nomessage) \\
\Rightarrow & chr3 \in \{ele : Element \mid Inter\} \land & [\text{def of } IMgeEvent] \\
& (atom\ chr3).type = imessage(nomessage) \\
\Rightarrow & \{chr3, chr4\} \subseteq \{ele : Element \mid Inter\} & [\text{pre } AddMgeEvent]
\end{array}
$$

We can see $\{chr3, chr4\} \subseteq \{ele : Element \mid Inter\}$ follows directly from constraints on $(pool\ id2?).proc$ and $chr3$.

We now consider the constraint $\{chr5, chr6\} \subseteq \{ele : Element \mid Activity\}$ specified in the definition $pc8Def$.

$$\begin{aligned}
&chr5 = (activities\ pc5)\ tor? \land \\
&\#(((atom\ chr5).exit) \rhd \{imessage(nomessage)\}) = 1 \land \\
&msg? \notin getMsgs(\bigcup\{p : \operatorname{ran} pool \bullet p.proc\}) \\
&\Rightarrow chr5 \in \{ele : Element \mid Activity\} \land && [\text{def of } activities] \\
&\quad \#(((atom\ chr5).exit) \rhd \{imessage(nomessage)\}) = 1 \land \\
&\quad msg? \notin getMsgs(\bigcup\{p : \operatorname{ran} pool \bullet p.proc\}) \\
&\Rightarrow chr5 \in \{ele : Element \mid Activity\} \land && [\text{def of } getMsgs] \\
&\quad \#(((atom\ chr5).exit) \rhd \{imessage(nomessage)\}) = 1 \land \\
&\quad msg? \notin getMsg(atom\ chr5)) \\
&\Rightarrow \{chr5, chr6\} \subseteq \{ele : Element \mid Activity\} && [\text{pre } AddExceptionMgeFlow]
\end{aligned}$$

We can see $\{chr5, chr6\} \subseteq \{ele : Element \mid Activity\}$ follows directly from constraints on $(pool\ id2?).proc$ and $chr5$.

We now consider the constraint $\{chr7, chr8\} \subseteq \{ele : Element \mid FullTask\}$ specified in the definition $pc9Def$.

$$\begin{aligned}
&chr7 = (tasks\ pc5)\ tor? \land \\
&msg? \notin getMsgs(\bigcup\{p : \operatorname{ran} pool \bullet p.proc\}) \\
&\Rightarrow chr7 \in \{ele : Element \mid FullTask\} \land && [\text{def of } tasks \text{ and } \rhd] \\
&\quad msg? \notin getMsgs(\bigcup\{p : \operatorname{ran} pool \bullet p.proc\}) \\
&\Rightarrow chr7 \in \{ele : Element \mid FullTask\} \land && [\text{def of } getMsgs] \\
&\quad msg? \notin getMsg(atom\ chr7) \\
&\Rightarrow \{chr7, chr8\} \subseteq \{ele : Element \mid FullTask\} && [\text{pre } AddReceiveMgeFlowTask]
\end{aligned}$$

We can see $\{chr7, chr8\} \subseteq \{ele : Element \mid FullTask\}$ follows directly from constraints on $(pool\ id2?).proc$ and $chr7$.

We now consider the constraint $pc2 \in \{proc : Process \mid Pool\}$ specified in the definition $pc2Def$. Note that by definition of the before state schema $Diagram$, we have the following implication.

$$pc1 = (pool\ id1?).proc \Rightarrow pc1 \in \{proc : Process \mid Pool\} \qquad [pool \in PoolId \rightarrowtail\!\!\!\!\rightarrow Pool]$$

We now expand the definition of $pc2$:

$$pc2 = \begin{cases} (pc1 \cup \{chs2\}) \setminus \{chs1\} & \text{if } chs1 \in pc1 \\ ((pc1 \setminus cont(proc, \{chs1\})) \cup & \text{otherwise} \\ \quad \{s : cont(pc1, \{chs1\}) \bullet rep(s, modify(content(s), \{chs2\}, \{chs1\}))\}) & \end{cases}$$

For the case $chs1 \in pc1$, we show $pc2 \in \{proc : Process \mid Pool\}$ follows directly from constraints on the before state $Pool$ and the input components, where $pc2 == (pc1 \cup \{chs2\}) \setminus \{chs1\}$.

$$\begin{aligned}
&pc1 \in \{proc : Process \mid Pool\} \land && [\text{def of } pc1,\ chs1 \text{ and } chs2] \\
&\{chs1, chs2\} \subseteq \{End \bullet (atom\ ele).type \in \operatorname{ran} emessage\} \\
&\Leftrightarrow && [\text{set-compre, def of } GenProc,\ hasExgates \text{ and rearrange}] \\
&\quad (\exists\, e, f : pc1 \bullet e \in \{Start \bullet ele\} \land f \in \{End \bullet ele\}) \land \\
&\quad (\neg\ (\exists\, e : pc1 \bullet e \in \{Inter \mid (atom\ ele).type \in \operatorname{ran} ierror \bullet ele\})) \land \\
&\quad (\forall\, e : \{g : Element \mid g \in_{\mathrm{p}} pc1\} \bullet e \in \{FlowObject \bullet ele\}) \land \\
&\quad pc1 \in processSet \land \\
&\quad pc1 \in noOverLap \land
\end{aligned}$$

$$
\begin{array}{l}
(\forall f : \{g : Element \mid g \in_{\mathrm{p}} pc1\} \bullet f \in eventgate \Rightarrow \\
\quad (\forall e : Element \bullet (f, e) \in edge(direct(pc1, e)) \Rightarrow e \in sendelement)) \\
\Rightarrow \qquad\qquad [\text{only } type \text{ component } chs2 \text{ is different from } chs1, \text{ and } msg? \notin getMsgs(pc1)] \\
(\exists e, f : pc2 \bullet e \in \{Start \bullet ele\} \land f \in \{End \bullet ele\}) \land \\
(\neg (\exists e : pc2 \bullet e \in \{Inter \mid (atom\ ele).type \in \operatorname{ran} ierror \bullet ele\})) \land \\
(\forall e : \{g : Element \mid g \in_{\mathrm{p}} pc2\} \bullet e \in \{FlowObject \bullet ele\}) \land \\
pc2 \in processSet \land \\
pc2 \in noOverLap \land \\
(\forall f : \{g : Element \mid g \in_{\mathrm{p}} pc2\} \bullet f \in eventgate \Rightarrow \\
\quad (\forall e : Element \bullet (f, e) \in edge(direct(pc2, e)) \Rightarrow e \in sendelement))
\end{array}
$$

We therefore conclude that $pc2 \in \{proc : Process \mid Pool\}$ follows directly from constraints on $pc1$ and the input components
For case $chs1 \notin pc1$, we consider the following inductive hypothesis

$$
\begin{array}{l}
\{s : cont(pc1, \{chs1\}) \bullet modify(content(s), \{chs2\}, \{chs1\})\} \subseteq \{proc : Process \mid Pool\}) \Rightarrow \\
((proc \setminus cont(proc, \{chs1\})) \cup \\
\{s : cont(pc1, \{chs1\}) \bullet rep(s, modify(content(s), \{chs2\}, \{chs1\}))\}) \in \{proc : Process \mid Pool\})
\end{array}
$$

According to Lemma B.2, it is sufficient to deduce the following set of facts (conjuncts) about the precondition schema.

$$cont(pc1, \{chs1\}) \subseteq \operatorname{ran} compound \tag{B.41}$$

$$\#cont(pc1, \{chs1\}) = 1 \tag{B.42}$$

$$
\begin{array}{l}
\bigcup \{s : cont(pc1, \{chs1\}) \bullet getSeqflows(modify(content(s), \{chs2\}, \{chs1\}))\} \cap \\
\quad getSeqflows(pc1 \setminus \{s, e : Element \mid s \in cont(pc1, \{chs1\}) \land e \in_{\mathrm{e}} s \bullet e\}) = \emptyset
\end{array} \tag{B.43}
$$

$$
\begin{array}{l}
\bigcup \{s : cont(pc1, \{chs1\}) \bullet getMsgs(modify(content(s), \{chs2\}, \{chs1\}))\} \cap \\
\quad getMsgs(pc1 \setminus \{s, e : Element \mid s \in cont(pc1, \{chs1\}) \land e \in_{\mathrm{e}} s \bullet e\}) = \emptyset
\end{array} \tag{B.44}
$$

$$
\begin{array}{l}
\{s, e : Element \mid s \in cont(pc1, \{chs1\}) \land \\
\quad e \in modify(content(s), \{chs2\}, \{chs1\}) \land (atom\ e).type \in \operatorname{ran} eerror \bullet e\} = \\
\quad \{s, e : Element \mid s \in cont(pc1, \{chs1\}) \land e \in content(s) \land (atom\ e).type \in \operatorname{ran} eerror \bullet e\}
\end{array} \tag{B.45}
$$

We consider each of the conjuncts individually: Conjunct B.41 follows from the definition of *cont* and schema definition *FullSub*. Conjunct B.42 follows from the membership $pc1 \in noOverLap$. Conjunct B.43 follows from the following constraints $getIns\{chs1\} = getIns\{chs2\}$, $outs(chs1) = outs(chs2)$ and $pc1 \in noOverLap$. Conjunct B.44 follows from $getMsgs\{chs2\} = \{msg?\}$, $msg? \notin getMsgs(pc1)$ and $pc1 \in noOverLap$. Conjunct B.45 follows from $\{(atom\ chs1?).type, (atom\ chs2?).type\} \cap \operatorname{ran} eerror = \emptyset$.

We can conclude that $pc2 \in \{proc : Process \mid Pool\}$ follows directly from constraints on the before state and the input components. Note that since we have the following implication,

$$pc5 = (pool\ id2?).proc \Rightarrow pc5 \in \{proc : Process \mid Pool\} \qquad [pool \in PoolId \rightarrowtail Pool]$$

we may apply to same reasoning for the following constraints.

$$
\begin{array}{l}
pc7 \in \{proc : Process \mid Pool\} \\
pc6 \in \{proc : Process \mid Pool\}
\end{array}
$$

where $pc6 = modify(pc5, \{chr2\}, \{chr1\})$ specified in the definition $pc6Def$ and $pc7 = modify(pc5, \{chr4\}, \{chr3\})$ specified in the definition $pc7Def$. Here the *type* component is the only modification between $chr1$ and $chr2$, and $chr3$ and $chr4$. Moreover, their *type* components are members of $\operatorname{ran} smessage \cup \operatorname{ran} imessage$.

We now consider the constraint $pc3 \in \{proc : Process \mid Pool\}$ specified in the definition $pc3Def$. We show this follows directly from constraints on $pc1$ and the input components by induction over the containment in $pc3$. We first expand the definition of $pc3$:

$$pc3 = \begin{cases} (pc1 \cup \{chs4\}) \setminus \{chs3\} & \text{if } chs3 \in pc1 \\ ((pc1 \setminus cont(proc, \{chs3\})) \cup & \text{otherwise} \\ \quad \{s : cont(pc1, \{chs3\}) \bullet rep(s, modify(content(s), \{chs4\}, \{chs3\}))\}) & \end{cases}$$

For the case $chs3 \in pc1$, we show $pc3 \in \{proc : Process \mid Pool\}$ follows directly from constraints on the before state $Pool$ and the input components, where $pc3 == ((pc1 \cup \emptyset) \setminus \{chs3\}) \cup \{chs4\}$.

$pc1 \in \{proc : Process \mid Pool\} \land$ [def of $pc1$, $chs3$ and $chs4$]
$\{chs3, chs4\} \subseteq \{ele : Element \mid FullTask\}$
$\Leftrightarrow$ [set-compre, def of $GenProc$, $hasExgates$ and rearrange]
$(\exists\, e, f : pc1 \bullet e \in \{Start \bullet ele\} \land f \in \{End \bullet ele\}) \land$
$(\neg\, (\exists\, e : pc1 \bullet e \in \{Inter \mid (atom\ ele).type \in \operatorname{ran} ierror \bullet ele\})) \land$
$(\forall\, e : \{g : Element \mid g \in_{\mathrm{p}} pc1\} \bullet e \in \{FlowObject \bullet ele\}) \land$
$pc1 \in processSet \land$
$pc1 \in noOverLap \land$
$(\forall\, f : \{g : Element \mid g \in_{\mathrm{p}} pc1\} \bullet f \in eventgate \Rightarrow$
$\quad (\forall\, e : Element \bullet (f, e) \in edge(direct(pc1, e)) \Rightarrow e \in sendelement)) \land$
$\{chs3, chs4\} \subseteq \{ele : Element \mid FullTask\}$
$\Rightarrow$ [only $send$ component $chs4$ is different from $chs3$, and $msg? \notin getMsgs(pc1)$]
$(\exists\, e, f : pc3 \bullet e \in \{Start \bullet ele\} \land f \in \{End \bullet ele\}) \land$
$(\neg\, (\exists\, e : pc3 \bullet e \in \{Inter \mid (atom\ ele).type \in \operatorname{ran} ierror \bullet ele\})) \land$
$(\forall\, e : \{g : Element \mid g \in_{\mathrm{p}} pc3\} \bullet e \in \{FlowObject \bullet ele\}) \land$
$pc3 \in processSet \land$
$pc3 \in noOverLap \land$
$(\forall\, f : \{g : Element \mid g \in_{\mathrm{p}} pc3\} \bullet f \in eventgate \Rightarrow$
$\quad (\forall\, e : Element \bullet (f, e) \in edge(direct(pc3, e)) \Rightarrow e \in sendelement))$

We therefore conclude that $pc3 \in \{proc : Process \mid Pool\}$ follows directly from constraints on the before state $Pool$ and the input components
For case $chs3 \notin pc1$, we consider the following inductive hypothesis

$\{s : cont(pc1, \{chs3\}) \bullet modify(content(s), \{chs4\}, \{chs3\})\} \subseteq \{proc : Process \mid Pool\}) \Rightarrow$
$((proc \setminus cont(proc, \{chs3\})) \cup$
$\{s : cont(pc1, \{chs3\}) \bullet rep(s, modify(content(s), \{chs4\}, \{chs3\}))\}) \in \{proc : Process \mid Pool\})$

According to Lemma B.2, it is sufficient to deduce the following set of facts (conjuncts) about the precondition schema.

$$cont(pc1, \{chs3\}) \subseteq \operatorname{ran} compound \tag{B.46}$$
$$\#cont(pc1, \{chs3\}) = 1 \tag{B.47}$$
$$\bigcup \{s : cont(pc1, \{chs3\}) \bullet getSeqflows(modify(content(s), \{chs4\}, \{chs3\}))\} \cap \tag{B.48}$$
$$\quad getSeqflows(pc1 \setminus \{s, e : Element \mid s \in cont(pc1, \{chs3\}) \land e \in_{\mathrm{e}} s \bullet e\}) = \emptyset$$
$$\bigcup \{s : cont(pc1, \{chs3\}) \bullet getMsgs(modify(content(s), \{chs4\}, \{chs3\}))\} \cap \tag{B.49}$$
$$\quad getMsgs(pc1 \setminus \{s, e : Element \mid s \in cont(pc1, \{chs3\}) \land e \in_{\mathrm{e}} s \bullet e\}) = \emptyset$$
$$\{s, e : Element \mid s \in cont(pc1, \{chs3\}) \land \tag{B.50}$$
$$\quad e \in modify(content(s), \{chs4\}, \{chs3\}) \land (atom\ e).type \in \operatorname{ran} eerror \bullet e\} =$$
$$\quad \{s, e : Element \mid s \in cont(pc1, \{chs3\}) \land e \in content(s) \land (atom\ e).type \in \operatorname{ran} eerror \bullet e\}$$

We consider each of the conjuncts individually: Conjunct B.46 follows from the definition of *cont* and schema definition *FullSub*. Conjunct B.47 follows from the membership $pc1 \in noOverLap$. Conjunct B.48 follows from the following constraints $getIns\{chs3\} = getIns\{chs4\}$, $outs(chs3) = outs(chs4)$ and $pc1 \in noOverLap$. Conjunct B.49 follows from $getMsgs\{chs4\} = getMsgs\{chs3\} \cup \{msg?\}$, $msg? \notin getMsgs(pc1)$ and $pc1 \in noOverLap$, and Conjunct B.50 follows from $\{(atom\ chs3).type, (atom\ chs4).type\} \cap \operatorname{ran} eerror = \emptyset$.

We therefore conclude that $pc3 \in \{proc : Process \mid Pool\}$ follows directly from constraints on the before state and $pc1$. Note that we may again apply to same reasoning for the following constraints.

$$pc9 \in \{proc : Process \mid Pool\}$$

where $pc9 = modify(pc5, \{chr8\}, \{chr7\})$ specified in the definition $pc9Def$. Here the *receive* component is the only modification between $chr7$ and $chr8$ such that $getMsgs\{chr7\} \cup \{msg?\} = getMsgs\{chr8\}$ and $msg? \notin getMsgs(pc1)$.

We now consider the constraint $pc8 \in \{proc : Process \mid Pool\}$ specified in the definition $pc8Def$. We show this follows directly from constraints on the before state $pc5$ and the input components by induction over the containment in $pc8$. We first expand the definition of $pc8$:

$$pc8 = \begin{cases} (pc5 \cup \{chr6\}) \setminus \{chr5\} & \text{if } chr5 \in pc5 \\ ((pc5 \setminus cont(proc, \{chr5\})) \cup & \text{otherwise} \\ \quad \{s : cont(pc5, \{chr5\}) \bullet rep(s, modify(content(s), \{chr6\}, \{chr5\}))\}) & \end{cases}$$

For the case $chr5 \in pc5$, we show $pc8 \in \{proc : Process \mid Pool\}$ follows directly from constraints on the before state $Pool$ and the input components, where $pc8 == (pc5 \cup \{chr6\}) \setminus \{chr5\}$.

$$\begin{array}{ll}
& pc5 \in \{proc : Process \mid GenProc \land hasExgates\} \land \qquad \text{[def of } pc5\text{, } chr5 \text{ and } chr6\text{]} \\
& \{chr5, chr6\} \subseteq \{ele : Element \mid FullSub\} \land \\
& getSeqflows\{chr6\} = getSeqflows\{chr5\} \\
\Leftrightarrow & \qquad \text{[set-compre, def of } GenProc\text{, } hasExgates \text{ and rearrange]} \\
& \quad (\exists\, e, f : pc5 \bullet e \in \{Start \bullet ele\} \land f \in \{End \bullet ele\}) \land \\
& \quad (\neg\, (\exists\, e : pc5 \bullet e \in \{Inter \mid (atom\ ele).type \in \operatorname{ran} ierror \bullet ele\})) \land \\
& \quad (\forall\, e : \{g : Element \mid g \in_{\mathrm{p}} pc5\} \bullet e \in \{FlowObject \bullet ele\}) \land \\
& \quad pc5 \in processSet \land \\
& \quad pc5 \in noOverLap \land \\
& \quad (\forall f : \{g : Element \mid g \in_{\mathrm{p}} pc5\} \bullet f \in eventgate \Rightarrow \\
& \qquad (\forall\, e : Element \bullet (f, e) \in edge(direct(pc5, e)) \Rightarrow e \in sendelement)) \land \\
& \quad \{chr5, chr6\} \subseteq \{ele : Element \mid FullSub\} \\
\Rightarrow & \qquad \text{[only } exit \text{ component } chr6 \text{ is different from } chr5\text{, } msg? \notin getMsg(pc5)\text{]} \\
& \quad (\exists\, e, f : pc8 \bullet e \in \{Start \bullet ele\} \land f \in \{End \bullet ele\}) \land \\
& \quad (\neg\, (\exists\, e : pc8 \bullet e \in \{Inter \mid (atom\ ele).type \in \operatorname{ran} ierror \bullet ele\})) \land \\
& \quad (\forall\, e : \{g : Element \mid g \in_{\mathrm{p}} pc8\} \bullet e \in \{FlowObject \bullet ele\}) \land \\
& \quad pc8 \in processSet \land \\
& \quad pc8 \in noOverLap \land \\
& \quad (\forall f : \{g : Element \mid g \in_{\mathrm{p}} pc8\} \bullet f \in eventgate \Rightarrow \\
& \qquad (\forall\, e : Element \bullet (f, e) \in edge(direct(pc8, e)) \Rightarrow e \in sendelement))
\end{array}$$

We therefore conclude that $pc8 \in \{proc : Process \mid Pool\}$ follows directly from constraints on $pc5$ and the input components
For case $chr5 \notin pc5$, we consider the following inductive hypothesis

$$\{s : cont(pc5, \{chr5\}) \bullet modify(content(s), \{chr6\}, \{chr5\})\} \subseteq \{proc : Process \mid Pool\}) \Rightarrow$$
$$((proc \setminus cont(proc, \{chr5\})) \cup$$

$$\{s : cont(pc5, \{chr5\}) \bullet rep(s, modify(content(s), \{chr6\}, \{chr5\}))\}) \in \{proc : Process \mid Pool\})$$

According to Lemma B.2, it is sufficient to deduce the following set of facts (conjuncts) about the precondition schema.

$$cont(pc5, \{chr5\}) \subseteq \operatorname{ran} compound \tag{B.51}$$

$$\#cont(pc5, \{chr5\}) = 1 \tag{B.52}$$

$$\bigcup\{s : cont(pc5, \{chr5\}) \bullet getSeqflows(modify(content(s), \{chr6\}, \{chr5\}))\} \cap getSeqflows(pc5 \setminus \{s, e : Element \mid s \in cont(pc5, \{chr5\}) \wedge e \in_e s \bullet e\}) = \emptyset \tag{B.53}$$

$$\bigcup\{s : cont(pc5, \{chr5\}) \bullet getMsgs(modify(content(s), \{chr6\}, \{chr5\}))\} \cap getMsgs(pc5 \setminus \{s, e : Element \mid s \in cont(pc5, \{chr5\}) \wedge e \in_e s \bullet e\}) = \emptyset \tag{B.54}$$

$$\{s, e : Element \mid s \in cont(pc5, \{chr5\}) \wedge e \in modify(content(s), \{chr6\}, \{chr5\}) \wedge (atom\ e).type \in \operatorname{ran} eerror \bullet e\} = \{s, e : Element \mid s \in cont(pc5, \{chr5\}) \wedge e \in content(s) \wedge (atom\ e).type \in \operatorname{ran} eerror \bullet e\} \tag{B.55}$$

We consider each of the conjuncts individually: Conjunct B.51 follows from the definition of *cont* and schema definition *FullSub*. Conjunct B.52 follows from the membership $pc5 \in noOverLap$. Conjunct B.53 follows from the following constraints: $getIns\{chr5\} = getIns\{chr6\}$, $outs(chr5) = outs(chr6)$ and $pc5 \in noOverLap$. Conjunct B.49 follows from $getMsgs\{chr5\} = getMsgs\{chr6\} \cup \{msg?\}$, $msg? \notin getMsgs(pc5)$ and $pc5 \in noOverLap$. Conjunct B.50 follows from $\{(atom\ chr5).type, (atom\ chr6).type\} \cap \operatorname{ran} eerror = \emptyset$.

We conclude that $pc8 \in \{proc : Process \mid Pool\}$ follows directly from constraints on the before state *Pool* and the input components.

We now consider eliminating the existential quantifier over the after state $Diagram'$. This requires investigation into all possible combinations of assigning values for $pool'\ id1?$ and $pool'\ id2?$ of the after state. By applying schema quantification and one-point rule to the $pool'$ component of the after state, we consider the constraint $pool' \in \{pool : PoolId \rightarrowtail\!\!\!\!\!\rightarrow Pool \mid Diagram\}$ where $pool'$ takes one of the following 8 values.

$$pool' = pool \oplus \{id1? \mapsto \langle\!| proc \leadsto pc2 |\!\rangle, id2? \mapsto \langle\!| proc \leadsto pc6 |\!\rangle\} \tag{B.56}$$

$$pool' = pool \oplus \{id1? \mapsto \langle\!| proc \leadsto pc2 |\!\rangle, id2? \mapsto \langle\!| proc \leadsto pc7 |\!\rangle\} \tag{B.57}$$

$$pool' = pool \oplus \{id1? \mapsto \langle\!| proc \leadsto pc2 |\!\rangle, id2? \mapsto \langle\!| proc \leadsto pc8 |\!\rangle\} \tag{B.58}$$

$$pool' = pool \oplus \{id1? \mapsto \langle\!| proc \leadsto pc2 |\!\rangle, id2? \mapsto \langle\!| proc \leadsto pc9 |\!\rangle\} \tag{B.59}$$

$$pool' = pool \oplus \{id1? \mapsto \langle\!| proc \leadsto pc3 |\!\rangle, id2? \mapsto \langle\!| proc \leadsto pc6 |\!\rangle\} \tag{B.60}$$

$$pool' = pool \oplus \{id1? \mapsto \langle\!| proc \leadsto pc3 |\!\rangle, id2? \mapsto \langle\!| proc \leadsto pc7 |\!\rangle\} \tag{B.61}$$

$$pool' = pool \oplus \{id1? \mapsto \langle\!| proc \leadsto pc3 |\!\rangle, id2? \mapsto \langle\!| proc \leadsto pc8 |\!\rangle\} \tag{B.62}$$

$$pool' = pool \oplus \{id1? \mapsto \langle\!| proc \leadsto pc3 |\!\rangle, id2? \mapsto \langle\!| proc \leadsto pc9 |\!\rangle\} \tag{B.63}$$

We select Combination B.56, expand the constraint $pool' \in \{pool : PoolId \rightarrowtail\!\!\!\!\!\rightarrow Pool \mid Diagram\}$ and consider each conjunct individually.

$$[\{\langle\!| proc \leadsto pc2 |\!\rangle, \langle\!| proc \leadsto pc6 |\!\rangle\} \cap \operatorname{ran} pool = \emptyset]$$

$$pool' \in PoolId \rightarrowtail\!\!\!\!\!\rightarrow Pool$$

$$[pool \neq \emptyset]$$

$$pool' \neq \emptyset$$

$$[getSeqflows(pc2) = getSeqflows(pc1) \text{ and } getSeqflows(pc6) = getSeqflows(pc5)]$$

$$\forall p, q : \operatorname{ran} pool' \bullet p \neq q \Rightarrow getSeqflows\ p.proc \cap getSeqflows\ q.proc = \emptyset$$

$$[getSds(pc2) = getSds(pc1) \cup \{msg?\},\ getRecs(pc2) = getRecs(pc1),]$$
$$[getSds(pc5) = getSds(pc6) \text{ and } getRecs(pc5) = getRecs(pc6) \cup \{msg?\}]$$
$$\forall p, q : \operatorname{ran} pool' \bullet (p \neq q \Rightarrow getSds\ p.proc \cap getSds\ q.proc = \emptyset \land getRecs\ p.proc \cap getRecs\ q.proc = \emptyset)$$

$$[getSds(pc2) = getSds(pc1) \cup \{msg?\},\ getRecs(pc2) = getRecs(pc1),]$$
$$[getSds(pc5) = getSds(pc6) \text{ and } getRecs(pc5) = getRecs(pc6) \cup \{msg?\}]$$
$$\forall p : \operatorname{ran} pool' \bullet$$
$$(\forall m : getSds\ p.proc \bullet (\exists q : \operatorname{ran} pool \bullet (p \neq q \land m \in getRecs\ p.proc))) \land$$
$$(\forall m : getRecs\ p.proc \bullet (\exists q : \operatorname{ran} pool \bullet (p \neq q \land m \in getSds\ p.proc)))$$

We therefore conclude that $pool' \in \{pool : PoolId \rightarrowtail Pool \mid Diagram\}$ with Combination B.56 follows directly from constraints on the before state *Diagram* and the input components. Note that we could apply the same reasoning for the other possible combinations B.57 to B.63. We therefore obtain the final simplified precondition schema, labelled *PreConnectMgeFlowDiagram*.

PreConnectMgeFlowDiagram

$$Diagram$$
$$id1?, id2? : PoolId$$
$$msg? : Mgeflow$$
$$tos?, tor? : Seqflow$$

$$id1? \neq id2? \land$$
$$\{id1?, id2?\} \subseteq \operatorname{dom} pool \land$$
$$msg? \notin getMsgs(\bigcup\{p : \operatorname{ran} pool \bullet p.proc\}) \land$$
$$(((atom\ ((msgsends\ ((pool\ id1?).proc))\ tos?)).type = emessage(nomessage) \lor$$
$$tos? \in \operatorname{dom}(tasks\ ((pool\ id1?).proc)))$$
$$\land$$
$$((atom\ ((msgrecs\ ((pool\ id2?).proc))\ tor?)).type = smessage(nomessage) \lor$$
$$(atom\ ((msgrecs\ ((pool\ id2?).proc))\ tor?)).type = imessage(nomessage) \lor$$
$$tor? \in \operatorname{dom}(tasks\ ((pool\ id2?).proc)) \lor$$
$$(tor? \in \operatorname{dom}(activities\ ((pool\ id2?).proc)) \land$$
$$\#(((atom\ ((activities\ ((pool\ id2?).proc))\ tor?)).exit) \rhd \{imessage(nomessage)\}) = 1)))$$

Appendix C

Proofs

C.1 Proofs for Section 5.8

In this section we write $D_p[\![P]\!]$ to denote the semantics of some BPMN process P, and let X and Y be two BPMN processes such that their behaviours are given by the following CSP processes,

$$\begin{aligned} D_p[\![X]\!] &= \|\, i : I \bullet \alpha P(i) \circ P(i) \\ D_p[\![Y]\!] &= \|\, j : J \bullet \alpha P(j) \circ P(j) \end{aligned}$$

where process X directly contains the set of elements I and Y directly contains the set of elements J. Similarly, we write $D[\![D]\!]$ to denote the semantics of some BPMN diagram D and let M and N be two BPMN diagrams such that their behaviours are given by the following CSP processes,

$$\begin{aligned} D[\![M]\!] &= \|\, i : \operatorname{dom} M.\mathit{pools} \bullet \alpha Pl(M, i) \circ Pl(M, i) \\ D[\![N]\!] &= \|\, j : \operatorname{dom} N.\mathit{pools} \bullet \alpha Pl(N, J) \circ Pl(N, j) \end{aligned}$$

where $Pl(M, i) = D_p[\![M.\mathit{pools}(i).\mathit{proc}]\!]$.

Lemma C.1. *Let X and Y be BPMN processes satisfying Condition a on Page 92, and that $D_p[\![X]\!] \sqsubseteq_{\mathcal{F}} D_p[\![Y]\!]$. Given some element $g \in I \cap J$, such that CSP processes XP and YP are defined as follow,*

$$\begin{aligned} XP &= \|\, i : I \setminus \{g\} \bullet \alpha P(i) \circ P(i) \\ YP &= \|\, j : J \setminus \{g\} \bullet \alpha P(j) \circ P(j) \end{aligned}$$

then $XP \sqsubseteq_{\mathcal{F}} YP$ holds.

Proof. Due to Condition a, we can partition the set of elements $I \setminus J$ into two sets $\langle A, B\rangle$ $\mathsf{partition}$ $(I \setminus J)$: Each element $e \in A$ is either an data-based XOR split gateway, a subprocess, a nondeterministic sequential multiple instance activity or a nondeterministic parallel multiple instance activity such that there exists an element $e' \in J \setminus I$ where $P(e) \sqsubseteq_{\mathcal{F}} P(e')$. For each element $f \in B$, $(e, f) \in \mathit{edge}(X)^+$ for some data-based XOR split gateway $e \in A$. Again due to Condition a, we further partition that $J \setminus I$ into two sets $\langle M, N\rangle$ $\mathsf{partition}$ $J \setminus I$: For each $e' \in M$, there exists exactly one element $e \in A$ such that $P(e) \sqsubseteq_{\mathcal{F}} P(e')$, and each element $e \in N$ is a XOR join gateway such that there exists a XOR join gateway $f \in B$ with the same outgoing sequence flow and whose set of incoming sequence flows is a superset of e's. Let $K \subseteq B$ be the set of elements such that there is no XOR join gateway $f \in B$ with the same outgoing sequence flow. We observe that YP does not offer any events performed by elements in K, therefore YP can be expressed as follows,

$$YP = \|\, j : (J \cup B) \setminus (N \cup \{g\}) \bullet \alpha P(j) \circ P(j)$$

where processes modelling behaviour of elements in K are composed in parallel with those of elements in J such that processes modelling behaviour of element in N can safely be replaced by those of element in $B \setminus K$. This is possible since CSP events, which model those incoming sequence flows of elements in $B \setminus K$ and do not belong to elements in N, have to be synchronised with elements in K, whose only events we know can be performed are those for cooperating with completion, Furthermore, by the following argument:

$$\begin{aligned} &(J \cup B) \setminus (N \cup \{g\}) \\ &\equiv (J \cup ((I \setminus J) \setminus A)) \setminus (N \cup \{g\}) && [\langle A, B\rangle \ \mathsf{partition}\ (I \setminus J)] \end{aligned}$$

$$\equiv (J \cup (I \setminus A)) \setminus (N \cup \{g\})$$
$$\equiv (M \cup (I \setminus A)) \setminus \{g\} \qquad [\langle M, N\rangle \text{ partition } J \setminus I]$$

YP can be expressed as follows:

$$YP = \|\, j : ((I \setminus A) \cup M) \setminus \{g\} \bullet \alpha P(j) \circ P(j)$$

Since $g \notin M \cup A$, let $I' = I \setminus (\{g\} \cup A)$ and both XP and YP can be expressed as follows:

$$XP = \|\, i : I' \cup A \bullet \alpha P(i) \circ P(i)$$
$$YP = \|\, i : I' \cup M \bullet \alpha P(i) \circ P(i)$$

Since $\#A = \#M$ and for all $e \in A$ there exists $e' \in M$ such that $P(e) \sqsubseteq_{\mathcal{F}} P(e')$, by monotonicity of $\|$ with respect to $\mathcal{F}$, we conclude $XP \sqsubseteq_{\mathcal{F}} YP$. □

Lemma C.2. *Let X and Y be BPMN processes satisfy Condition a on Page 92 and that $D_p[\![X]\!] \sqsubseteq_{\mathcal{F}} D_p[\![Y]\!]$. Given some set of end events $F \subset I \cap J$, such that CSP processes XP and YP are defined as follow,*

$$XP = \|\, i : I \setminus F \bullet \alpha P(i) \setminus \{f : F \bullet c.f\} \circ P(i)$$
$$YP = \|\, j : J \setminus F \bullet \alpha P(j) \setminus \{f : F \bullet c.f\} \circ P(j)$$

then $XP \sqsubseteq_{\mathcal{F}} YP$ holds.

Proof. We let XP' and YP' defined as follows:

$$XP' = \|\, i : I \setminus F \bullet \alpha P(i) \circ P(i)$$
$$YP' = \|\, j : J \setminus F \bullet \alpha P(j) \circ P(j)$$

By Lemma C.1, we have $XP' \sqsubseteq_{\mathcal{F}} YP'$. We now consider processes XP and YP, which can be equivalently expressed as $XP' \,[\![\, \{f : F \bullet c.f\} \,]\!]\, \mathit{Skip}$ and $YP' \,[\![\, \{f : F \bullet c.f\} \,]\!]\, \mathit{Skip}$ respectively. By monotonicity of $\|$ with respect to $\mathcal{F}$, we conclude $XP \sqsubseteq_{\mathcal{F}} YP$. □

Lemma C.3. *Let X and Y be BPMN processes satisfy Condition a on Page 92 and that $D_p[\![X]\!] \sqsubseteq_{\mathcal{F}} D_p[\![Y]\!]$. Given two sets of end events $E \subseteq I \cap J$ and $F \cap (I \cup J) = \emptyset$, such that CSP processes XQ and YQ are defined as follow,*

$$XQ = \|\, i : I \setminus E \bullet (\alpha P(i) \setminus ES) \cup FS \circ \mathit{addends}(P(i), F)$$
$$YQ = \|\, j : J \setminus E \bullet (\alpha P(j) \setminus ES) \cup FS \circ \mathit{addends}(P(j), F)$$

where $ES = \{e : E \bullet c.e\}$ and $FS = \{f : F \bullet c.f\}$, and $XQ \sqsubseteq_{\mathcal{F}} YQ$ holds.

Proof. We let $XP = \|\, i : I \setminus E \bullet \alpha P(i) \setminus ES \circ P(i)$ and $YP = \|\, j : J \setminus E \bullet \alpha P(j) \setminus ES \circ P(j)$. Due to Lemma C.2 we know $XP \sqsubseteq_{\mathcal{F}} YP$. We observe that XQ may perform an event $c.f$ where $f \in F$ if and only if for all $i \in I$, $\mathit{addends}(P(i), F)$ offers to perform $c.f$. This also holds for YQ. Moreover, after performing $c.f$, both XQ and YQ become *Skip* As a result XQ and YQ can be expressed as follows,

$$XQ \equiv (((XP \,[\![\, \Sigma \setminus FS \,]\!]\, dp(XQ)) \triangle k \to \mathit{Skip}) \,[\![\, \Sigma \,]\!]\, EP) \setminus \{k\} \tag{C.1}$$
$$YQ \equiv (((YP \,[\![\, \Sigma \setminus FS \,]\!]\, dp(XQ)) \triangle k \to \mathit{Skip}) \,[\![\, \Sigma \,]\!]\, EP) \setminus \{k\} \tag{C.2}$$

where for all $\oplus \in \{\Box, \|, \triangle, \fatsemi, |||\}$, $dp(P)$ is defined as follows.

$$dp(\mathit{Skip}) = \mathit{Skip} \qquad dp(\mathit{Stop}) = \mathit{Stop} \qquad dp(e \to P) = e \to dp(P)$$
$$dp(P \setminus A) = dp(P) \setminus A \qquad dp(P \sqcap Q) = dp(P) \,\Box\, dp(Q) \qquad dp(P \oplus Q) = dp(P) \oplus dp(Q)$$

Specifically, $dp(P)$ replaces all occurrences of $\sqcap$ with $\Box$, while EP is defined as follows.

$$\begin{aligned} EP = {} & (\Box\, i : \Sigma \setminus \{e : G \cup H \bullet c.i\} \bullet i \to EP) \,\Box \\ & (\Box\, e : G \cup H \bullet c.e \to \mathit{Skip}) \,\Box \\ & (\Box\, i : F \bullet c.i \to k \to \mathit{Skip}) \end{aligned}$$

The process defined by Expression C.1 behaves as XP until one of $c.f$ where $f \in F$ is performed, and in which case the process may only perform a hidden event k and terminate; the process defined by Expression C.2 behaves similarly for YP. Since all standard CSP operators are monotonic with respect to $\mathcal{F}$, we therefore conclude $XQ \sqsubseteq_{\mathcal{F}} YQ$. □

Lemma C.4. *Let X and Y be BPMN processes satisfy Condition a on Page 92 and that $D_p[\![X]\!] \sqsubseteq_{\mathcal{F}} D_p[\![Y]\!]$. Given the following:*

- *An end event $end? \notin I \cup J$;*
- *A BPMN element $m \in I \cap J$ with an incoming sequence flow x such that for all event based XOR split gateways $n \in I \cup J$ we have $(n, m) \notin edge(direct(X, m)) \cup edge(direct(Y, m))$.*
- *A XOR join gateway j with incoming sequence flows x, y and outgoing sequence flow z, where x is also an incoming sequence flow of m, and no element contained in X and Y has sequence flows y and z.*
- *A BPMN element m' such that $P(m') = P(m)[s.x \leftarrow s.z]$.*

and define CSP processes XU and YU as follow:

$$\begin{aligned} XU &= \|\, i : (I \setminus \{m\}) \cup \{m', j\} \bullet \alpha P(i) \cup \{c.end?\} \circ addend(P(i), \{end?\}) \\ YU &= \|\, j : (J \setminus \{m\}) \cup \{m', j\} \bullet \alpha P(j) \cup \{c.end?\} \circ addend(P(j), \{end?\}) \end{aligned}$$

then $XU \sqsubseteq_{\mathcal{F}} YU$ holds.

Proof. Let XP and YP be the following processes:

$$\begin{aligned} XP &= \|\, i : I \setminus \{m\} \bullet \alpha P(i) \circ P(i) \\ YP &= \|\, j : J \setminus \{m\} \bullet \alpha P(j) \circ P(j) \end{aligned}$$

By Lemma C.1 we have $XP \sqsubseteq_{\mathcal{F}} YP$. Let XP' and YP' be the following processes:

$$\begin{aligned} XP' &= \|\, i : (I \setminus \{m\}) \cup \{m', j\} \bullet \alpha P(i) \circ P(i) \\ YP' &= \|\, j : (J \setminus \{m\}) \cup \{m', j\} \bullet \alpha P(j) \circ P(j) \end{aligned}$$

Again, by monotonicity of $\|$ with respect to $\mathcal{F}$, we conclude $XP' \sqsubseteq_{\mathcal{F}} YP'$. Furthermore, by Lemma C.3 we can conclude $XU \sqsubseteq_{\mathcal{F}} YU$. □

Lemma C.5. *Let X and Y be BPMN processes satisfy Condition a on Page 92 and that $D_p[\![X]\!] \sqsubseteq_{\mathcal{F}} D_p[\![Y]\!]$. Given the following:*

- *An end event $end? \notin I \cup J$;*
- *An activity BPMN element $m \in I \cap J$;*
- *An activity BPMN element m' obtained by adding exception flow (f, t) to m, where no element contained in X and Y has sequence flow f.*

and define CSP processes XT and YT as follow:

$$\begin{aligned} XT &= \|\, i : (I \setminus \{m\}) \cup \{m'\} \bullet \alpha P(i) \cup \{c.end?\} \circ addend(P(i), \{end?\}) \\ YT &= \|\, j : (J \setminus \{m\}) \cup \{m'\} \bullet \alpha P(j) \cup \{c.end?\} \circ addend(P(j), \{end?\}) \end{aligned}$$

then $XT \sqsubseteq_{\mathcal{F}} YT$ holds.

Proof. Let XP and YP be the following processes:

$$XP = \|\, i : I \setminus \{m\} \bullet \alpha P(i) \circ P(i)$$
$$YP = \|\, j : J \setminus \{m\} \bullet \alpha P(j) \circ P(j)$$

By Lemma C.1 we have $XP \sqsubseteq_{\mathcal{F}} YP$. Let XP' and YP' be the following processes:

$$XP' = \|\, i : (I \setminus \{m\}) \cup \{m'\} \bullet \alpha P(i) \circ P(i)$$
$$YP' = \|\, j : (J \setminus \{m\}) \cup \{m'\} \bullet \alpha P(j) \circ P(j)$$

Again, by monotonicity of $\|$ with respect to $\mathcal{F}$, we conclude $XP' \sqsubseteq_{\mathcal{F}} YP'$. Furthermore, by Lemma C.3 we can conclude $XT \sqsubseteq_{\mathcal{F}} YT$. □

Lemma C.6. *Given M and N be BPMN diagrams satisfy Condition b on Page 92, and that $D[\![M]\!] \sqsubseteq_{\mathcal{F}} D[\![N]\!]$, let CSP processes M' and N' defined as follow,*

$$M' = \|\, i : \mathrm{dom}\, M.pools \setminus \{id1?, id2?\} \bullet \alpha Pl(M, i) \circ Pl(M, i)$$
$$N' = \|\, i : \mathrm{dom}\, N.pools \setminus \{id1?, id2?\} \bullet \alpha Pl(N, i) \circ Pl(N, i)$$

where $\{id1?, id2?\} \subseteq \mathrm{dom}\, M.pools \cap \mathrm{dom}\, N.pools$, then $M' \sqsubseteq_{\mathcal{F}} N'$ holds.

Proof. Due to Condition b, we know $\mathrm{dom}\, M.pools \setminus \{id1?, id2?\} = \mathrm{dom}\, N.pools \setminus \{id1?, id2?\}$ and that $\forall\, i \in \mathrm{dom}\, M.pools \bullet Pl(M, i) \sqsubseteq_{\mathcal{F}} Pl(N, i)$, by monotonicity of $\|$ with respect to $\mathcal{F}$, we conclude $M' \sqsubseteq_{\mathcal{F}} N'$. □

Lemma C.7. *Given Condition a on Page 92, the operations SeqComp, Split, EventSplitOp, JoinOp are monotonic with respect to failures refinement.*

Proof. We consider operation $SeqComp(proc, new?, from?, end?)$ defined in Section 5.7.1 on Page 79, where *proc* is one of two BPMN processes X and Y. Specifically, we show that if $D_p[\![X]\!] \sqsubseteq_{\mathcal{F}} D_p[\![Y]\!]$ then $D_p[\![SeqComp(X, new?, from?, end?)]\!] \sqsubseteq_{\mathcal{F}} D_p[\![SeqComp(Y, new?, from?, end?)]\!]$, where e, m, $end?$ and $new?$ are as defined in Section 5.7.1.

$D_p[\![X]\!] \sqsubseteq_{\mathcal{F}} D_p[\![Y]\!]$
$\Leftrightarrow \|\, i : I \bullet \alpha P(i) \circ P(i) \sqsubseteq_{\mathcal{F}} \|\, j : J \bullet \alpha P(j) \circ P(j)$ [Def of X and Y]
$\Rightarrow \|\, i : I \setminus \{e\} \bullet \alpha P(i) \setminus \{c.e\} \circ P(i) \sqsubseteq_{\mathcal{F}} \|\, j : J \setminus \{e\} \bullet \alpha P(j) \setminus \{c.e\} \circ P(j)$ [Lemma C.2]
$\Rightarrow \|\, i : I \setminus \{e\} \bullet (\alpha P(i) \setminus \{c.e\}) \cup \{c.end?\} \circ addend(P(i), \{end?\}) \sqsubseteq_{\mathcal{F}}$ [Lemma C.3]
$\quad \|\, j : J \setminus \{e\} \bullet (\alpha P(j) \setminus \{c.e\}) \cup \{c.end?\} \circ addend(P(j), \{end?\})$
$\Rightarrow (\, \|\, i : I \setminus \{e\} \bullet (\alpha P(i) \setminus \{c.e\}) \cup \{c.end?\} \circ addend(P(i), \{end?\}))$ [Property of $\|$]
$\qquad |[alphas(I) \setminus \{c.e\}) \cup \{c.end?\} \mid alphas(\{new?, end?\})]|$
$\qquad (P(new?) \,|[\, \alpha P(new?) \mid \alpha P(end?) \,]|\, P(end?)) \sqsubseteq_{\mathcal{F}}$
$\quad (\, \|\, j : J \setminus \{e\} \bullet (\alpha P(j) \setminus \{c.e\}) \cup \{c.end?\} \circ addend(P(j), \{end?\}))$
$\qquad |[alphas(J) \setminus \{c.e\} \cup \{c.end?\} \mid alphas(\{new?, end?\})]|$
$\qquad (P(new?) \,|[\, \alpha P(new?) \mid \alpha P(end?) \,]|\, P(end?))$
$\Leftrightarrow D_p[\![SeqComp(X, new?, from?, end?)]\!] \sqsubseteq_{\mathcal{F}}$ [Def of *SeqComp*]
$\qquad D_p[\![SeqComp(Y, new?, from?, end?)]\!]$

Operations *Split*, *EventSplitOp* and *JoinOp* can be shown to be monotonic similarly. □

Lemma C.8. *Given Condition a on Page 92, the operation AddException is monotonic with respect to failures refinement.*

Proof. We consider operation $AddNoRelatedErrorException(proc, end?, etype?, eflow?, loc?)$ defined in Section 5.7.5 on Page 85, where *proc* is one of two BPMN processes X and Y, and *loc?* is the incoming sequence flow of some activity element A to which the exception flow $(eflow?, etype?)$ is added. Here we specifically show that if $D_p[\![X]\!] \sqsubseteq_{\mathcal{F}} D_p[\![Y]\!]$ then $D_p[\![AddNoRelatedErrorException(X, end?, etype?, eflow?, loc?)]\!] \sqsubseteq_{\mathcal{F}} D_p[\![AddNoRelatedErrorException(Y, end?, etype?, eflow?, loc?)]\!]$, where m, m' and $end?$ are as defined in Section 5.7.5.

$$
\begin{array}{ll}
& D_p[\![X]\!] \sqsubseteq_{\mathcal{F}} D_p[\![Y]\!] \\
\Leftrightarrow & \|\, i : I \bullet \alpha P(i) \circ P(i) \sqsubseteq_{\mathcal{F}} \|\, j : J \bullet \alpha P(j) \circ P(j) \qquad \text{[Def of } X \text{ and } Y\text{]} \\
\Rightarrow & (\, \|\, i : I \setminus \{m\} \bullet (\alpha P(i) \cup \{c.end?\}) \circ addend(P(i), \{end?\})) \qquad \text{[Lemma C.5]} \\
& \qquad [\![alphas(I \setminus \{m\}) \cup \{c.end?\} \mid \alpha P(m')\})\,]\!] \; P(m') \sqsubseteq_{\mathcal{F}} \\
& \quad (\, \|\, j : J \setminus \{m\} \bullet (\alpha P(j) \cup \{c.end?\}) \circ addend(P(j), \{end?\})) \\
& \qquad [\![alphas(J \setminus \{m\}) \cup \{c.end?\} \mid \alpha P(m')\})\,]\!] \; P(m') \\
\Rightarrow & (\, \|\, i : I \setminus \{m\} \bullet (\alpha P(i) \cup \{c.end?\}) \circ addend(P(i), \{end?\})) \qquad \text{[Property of } \|\text{]} \\
& \qquad [\![alphas(I \setminus \{m\}) \cup \{c.end?\} \mid \alpha P(m')\})\,]\!] \; P(m') \\
& \qquad\quad (P(m') \; [\![\, \alpha P(m') \mid \alpha P(end?)\,]\!] \; P(end?)) \sqsubseteq_{\mathcal{F}} \\
& \quad (\, \|\, j : J \setminus \{m\} \bullet (\alpha P(j) \cup \{c.end?\}) \circ addend(P(j), \{end?\})) \\
& \qquad [\![alphas(J \setminus \{m\}) \cup \{c.end?\} \mid \alpha P(m', end?)\})]\!] \\
& \qquad\quad (P(m') \; [\![\, \alpha P(m') \mid \alpha P(end?)\,]\!] \; P(end?)) \\
\Leftrightarrow & \qquad \text{[Def of } AddNoRelatedErrorException\text{]} \\
& D_p[\![AddNoRelatedErrorException(X, end?, etype?, eflow?, loc?)]\!] \sqsubseteq_{\mathcal{F}} \\
& \qquad D_p[\![AddNoRelatedErrorException(Y, end?, etype?, eflow?, loc?)]\!]
\end{array}
$$

Operation *AddRelatedErrorException* can be shown to be monotonic similarly. □

Lemma C.9. *Given Condition a on Page 92, the operations Loop and EventLoop are monotonic with respect to failures refinement.*

Proof. We consider the semantics of operation $Loop(proc, split?, join?, end?, connect?, from?, f2?, t2?)$ defined in Section 5.7.4 on Page 83, where *proc* is one of two BPMN processes X and Y. Specifically we show that if $D_p[\![X]\!] \sqsubseteq_{\mathcal{F}} D_p[\![Y]\!]$ then $D_p[\![Loop(X, split?, join?, end?, connect?, from?, f2?, t2?)]\!] \sqsubseteq_{\mathcal{F}} D_p[\![Loop(Y, split?, join?, end?, connect?, from?, f2?, t2?)]\!]$, where e, m, m', $split?$, $join?$, $end?$ are as defined in Section 5.7.4.

$$
\begin{array}{ll}
& D_p[\![X]\!] \sqsubseteq_{\mathcal{F}} D_p[\![Y]\!] \\
\Leftrightarrow & \|\, i : I \bullet \alpha P(i) \circ P(i) \sqsubseteq_{\mathcal{F}} \|\, j : J \bullet \alpha P(j) \circ P(j) \qquad \text{[Def of } X \text{ and } Y\text{]} \\
\Rightarrow & (\, \|\, i : I \setminus \{e, m\} \bullet ((\alpha P(i) \setminus \{c.e\}) \cup \{c.end?\}) \circ addend(P(i), \{end?\})) \qquad \text{[Lemmas C.2, C.4]} \\
& \qquad [\![alphas(I \setminus \{e, m\}) \cup \{c.end?\} \mid \alpha P(m') \cup \alpha P(join?)]\!] \\
& \qquad\quad (P(m') \; [\![\, \alpha P(m') \mid \alpha P(join?)\,]\!] \; P(join?)) \sqsubseteq_{\mathcal{F}} \\
& \quad (\, \|\, j : J \setminus \{e, m\} \bullet ((\alpha P(j) \setminus \{c.e\}) \cup \{c.end?\}) \circ addend(P(j), \{end?\})) \\
& \qquad [\![alphas(J \setminus \{e, m\}) \cup \{c.end?\} \mid \alpha P(m') \cup \alpha P(join?)]\!] \\
& \qquad\quad (P(m') \; [\![\, \alpha P(m') \mid \alpha P(join?)\,]\!] \; P(join?)) \\
\Rightarrow & (\, \|\, i : I \setminus \{e, m\} \bullet ((\alpha P(i) \setminus \{c.e\}) \cup \{c.end?\}) \circ addend(P(i), \{end?\})) \qquad \text{[Property of } \|\text{]} \\
& \qquad [\![alphas(I \setminus \{e, m\}) \cup \{c.end?\} \mid alphas(\{split?, join?, end?, m'\})]\!] \\
& \qquad (P(m') \; [\![\, \alpha P(m') \mid alphas(\{split?, join?, end?\})]\!] \\
& \qquad\quad (P(split?) \; [\![\, \alpha P(split?) \mid alphas(\{join?, end?\})]\!] \\
& \qquad\qquad (P(join?) \; [\![\, \alpha P(join?) \mid \alpha P(end?)\,]\!] \; P(end?)))) \sqsubseteq_{\mathcal{F}} \\
& \quad (\, \|\, j : J \setminus \{e, m\} \bullet ((\alpha P(j) \setminus \{c.e\}) \cup \{c.end?\}) \circ addend(P(i), \{end?\}))
\end{array}
$$

$$
\begin{array}{l}
\quad [\![alphas(J \setminus \{e, m\}) \cup \{c.end?\} \mid alphas(\{split?, join?, end?, m'\})]\!] \\
\quad (P(m') \; [\![\alpha P(m') \mid alphas(\{split?, join?, end?\})]\!] \\
\qquad (P(split?) \; [\![\alpha P(split?) \mid alphas(\{join?, end?\})]\!] \\
\qquad\quad (P(join?) \; [\![\alpha P(join?) \mid \alpha P(end?)]\!] \; P(end?)))) \\
\Leftrightarrow D_p[\![Loop(X, split?, join?, end?, connect?, from?, f2?, t2?)]\!] \sqsubseteq_{\mathcal{F}} \quad \text{[Def of } Loop\text{]} \\
\quad D_p[\![Loop(Y, split?, join?, end?, connect?, from?, f2?, t2?)]\!]
\end{array}
$$

Operation *EventLoop* can be shown to be monotonic similarly. □

Lemma C.10. *Given Conditions a and b on Page 92, the operation ConnectMgeFlowDiagram is monotonic with respect to failures refinement.*

Proof. We consider the semantics of operation $ConnectMgeFlowDiagram(pools, id1?, id2?, msg?, tos?, tor?)$ defined in Section 5.7.6 on Page 88. Let $\{X, Y\} \subseteq Diagram$ such that if $D[\![X]\!] \sqsubseteq_{\mathcal{F}} D[\![Y]\!]$ then $D[\![ConnectMgeFlowDiagram(X.pools, id1?, id2?, msg?, tos?, tor?)]\!] \sqsubseteq_{\mathcal{F}} D[\![ConnectMgeFlowDiagram(Y.pools, id1?, id2?, msg?, tos?, tor?)]\!]$ and processes AS and AR are defined in Section 5.7.6 on Page 88.

$$
\begin{array}{l}
D[\![X]\!] \sqsubseteq_{\mathcal{F}} D[\![Y]\!] \\
\Leftrightarrow \; \|\!|\, i : \mathrm{dom}\, X.pools \bullet \alpha Pl(X, i) \circ Pl(X, i) \sqsubseteq_{\mathcal{F}} \quad \text{[Def of } X \text{ and } Y\text{]} \\
\qquad \|\!|\, j : \mathrm{dom}\, Y.pools \bullet \alpha Pl(Y, j) \circ Pl(Y, j) \\
\Rightarrow (\; \|\!|\, i : \mathrm{dom}\, X.pools \setminus \{id1?, id2?\} \bullet \alpha Pl(X, i) \circ Pl(X, i)) \quad \text{[Lemma C.6, property of } \|\!|\text{]} \\
\quad [\![\bigcup \{ i : \mathrm{dom}\, X.pools \setminus \{id1?, id2?\} \bullet \alpha Pl(X, i) \} \mid \alpha Pl(X, id1?) \cup \alpha Pl(X, id2?) \cup \{m.msg?\}]\!] \\
\qquad (AS(X.pools(id1?).proc, tos?, msg?) \\
\qquad\quad [\![\alpha Pl(X, id1?) \cup \{m.msg?\} \mid \alpha Pl(X, id2?) \cup \{m.msg?\}]\!] \\
\qquad\quad AR(X.pools(id2?).proc, tor?, msg?)) \sqsubseteq_{\mathcal{F}} \\
\quad (\; \|\!|\, j : \mathrm{dom}\, Y.pools \setminus \{id1?, id2?\} \bullet \alpha Pl(Y, i) \circ Pl(Y, i)) \\
\quad [\![\bigcup \{ i : \mathrm{dom}\, Y.pools \setminus \{id1?, id2?\} \bullet \alpha Pl(Y, i) \} \mid \alpha Pl(Y, id1?) \cup \alpha Pl(Y, id2?) \cup \{m.msg?\}]\!] \\
\qquad (AS(Y.pools(id1?).proc, tos?, msg?) \\
\qquad\quad [\![\alpha Pl(Y, id1?) \cup \{m.msg?\} \mid \alpha Pl(Y, id2?) \cup \{m.msg?\}]\!] \\
\qquad\quad AR(Y.pools(id2?).proc, tor?, msg?)) \\
\Leftrightarrow \quad \text{[Def of } ConnectMgeFlowDiagram\text{]} \\
\quad D[\![ConnectMgeFlowDiagram(X, id1?, id2?, msg?, tos?, tor?)]\!] \sqsubseteq_{\mathcal{F}} \\
\qquad D[\![ConnectMgeFlowDiagram(Y, id1?, id2?, msg?, tos?, tor?)]\!]
\end{array}
$$

□

Theorem (5.4)**.** *Monotonicity. Given Conditions a and b on Page 92, the operations SeqComp, Split, EventSplitOp, JoinOp, Loop, EventLoop, AddException and ConnectMgeFlowDiagram are monotonic with respect to failures refinement.*

Proof. By Lemmas C.7, C.8, C.9 and C.10. □

Theorem (5.5)**.** *Given any subprocess s satisfying Conditions a and b, and such that its behaviour is modelled by CSP process $C[S]$, where S is the CSP process that models elements directly contained in s and $C[.]$ is a CSP process context that models s's sequence, message and exception flows. Let t be any subprocess whose behaviour is modelled by CSP process $C[T]$ and T is the CSP process that models elements directly contained in t. If we have both $S \sqsubseteq_{\mathcal{F}} T$ and $C[S] \sqsubseteq_{\mathcal{F}} C[T]$, then we have $C[S'] \sqsubseteq_{\mathcal{F}} C[T']$ where S' and T' are the results of applying any one of the operations SeqComp, Split, EventSplitOp, JoinOp, Loop, EventLoop, and AddException on S and T respectively.*

Proof. We observe that none of the operations *SeqComp*, *Split*, *EventSplitOp*, *JoinOp*, *Loop*, *EventLoop*, and *AddException* alter the exception flows S and T, as a result context $C[.]$ does not change after any one of the operations. Since $C[.]$ is defined using standard CSP operators, monotonicity is preserved. □

C.2 Proofs for Section 5.9

Theorem (5.14). *For any BPMN processes P and Q, if compatible(P, Q) then their collaboration is deadlock-free, that is, compatible$(P, Q) \Rightarrow DF \sqsubseteq_{\mathcal{F}} (D_p[\![P]\!] \,[\![\, \alpha D_p[\![P]\!] \mid \alpha D_p[\![Q]\!] \,]\!]\, D_p[\![Q]\!])$*

Proof. If $D_p[\![Q]\!]$ *RespondsTo* $D_p[\![P]\!]$ and P is deadlock-free, we can let $N = \{P\}$. By Theorem 5.12, $DF \sqsubseteq_{\mathcal{F}} (D_p[\![P]\!] \,[\![\, \alpha D_p[\![P]\!] \mid \alpha D_p[\![Q]\!] \,]\!]\, D_p[\![Q]\!])$. Conversely, if $D_p[\![P]\!]$ *RespondsTo* $D_p[\![Q]\!]$ and Q is deadlock-free, we can let $N = \{Q\}$. By Theorem 5.12, $DF \sqsubseteq_{\mathcal{F}} (D_p[\![P]\!]\,[\![\alpha D_p[\![P]\!] \mid \alpha D_p[\![Q]\!]\,]\!]\, D_p[\![Q]\!])$. □

Theorem (5.15). *For any BPMN processes P and Q, if compatible(P, Q) for all $D_p[\![P]\!] \sqsubseteq_{\mathcal{F}} D_p[\![P']\!]$ and $D_p[\![Q]\!] \sqsubseteq_{\mathcal{F}} D_p[\![Q']\!]$, compatible$(P', Q')$*

Proof. We know *compatible*(P, Q) implies both P and Q are deadlock-free and hence all of their refinements are also deadlock free. Due to Theorem 5.10, we also know *RespondsTo* is refinement-closed, therefore this shows *compatible* is also refinement-closed. □

Theorem (5.16). *Given a collaboration of business participants (BPMN processes) $B = \{i : I \bullet B(i)\}$, indexed by I, where the semantics of each business participant is denoted by process $P(i) = D_p[\![B(i)]\!]$ for $i \in I$ and the collaboration $C = \Vert\, i : I \bullet \alpha P(i) \circ P(i)$ is deadlock-free. Suppose there is a new business participant R, denoted by $Q = D_p[\![R]\!]$ such that $\alpha Q \cap (\alpha P(i) \cap \alpha P(j)) = \emptyset$ for $i, j : I$ and $i \neq j$ (an appropriated assumption as by definition each message flow connects exactly two participants). Then if*

$$\exists\, i : I \bullet \alpha Q \cap \alpha P(i) \neq \emptyset \wedge \forall\, i : I \bullet \alpha Q \cap \alpha P(i) \neq \emptyset \Rightarrow \textit{compatible}(R, B(i))$$

then $E = C \,[\![\bigcup\{i : I \bullet \alpha P(i)\} \mid \alpha Q \,]\!]\, Q$ is also deadlock-free.

Proof. Due to Theorem 5.12 this theorem holds for Q *RespondsTo* P. We now show this theorem also holds for P *RespondsTo* Q by contradiction.

Let's assume E could deadlock, that is, there exists $(s, \Sigma^{\checkmark}) \in \textit{failures}(E)$. This means there exists $(t, X) \in \textit{failures}(C)$ and $(u, Y) \in \textit{failures}(Q)$ such that both X and Y are maximal refusal sets where $t = s \upharpoonright \alpha C$, $u = s \upharpoonright \alpha Q$ and $(X \cup Y) = (\alpha C \cup \alpha Q)^{\checkmark}$ assuming (reasonably) $X \subseteq \alpha C$ and $Y \subseteq \alpha Q$. Since C is deadlock-free, $X \notin \Sigma^{\checkmark}$ by definition, therefore one of the following two cases can happen:

1. C can only terminate, that is, $X = \Sigma$ and for $(s_i, X_i) \in \textit{failures}(P(i))$ where $s_i = s \upharpoonright \alpha P(i)$, $\checkmark \notin X_i$.
2. C can only perform events in the alphabet of Q, that is, $\alpha C \setminus X \neq \emptyset$ and there exists i such that $\alpha P(i) \setminus X_i \cap \alpha Q \neq \emptyset$.

Case 1 We know there exists some i such that $P(i)$ *RespondsTo* Q (this also implies $P(i)$ *RespondsToLive* Q by Theorem 5.10) where $\alpha P(i) \cap \alpha Q \neq \emptyset$, and both $P(i)$ and Q are deadlock-free. If Y contains $\checkmark$, we have $(s_i, X_i^{\checkmark}) \in \textit{failures}(P(i) \parallel Q)$ but since Q is deadlock-free, therefore $(u, X_i^{\checkmark}) \notin \textit{failures}(Q)$, contradicting the assumption that $P(i)$ *RespondsTo* Q.

Case 2 We know $\checkmark \in X_i$. $\alpha P(i) \cap \alpha P(j) \cap \alpha Q = \emptyset$ for $i \neq j$ by definition, therefore $\alpha P(i) \setminus X_i \subseteq Y$ (Q refusing everything in the joint alphabet that $P(i)$ does not). But we know Q is deadlock-free therefore $Y \neq \Sigma^{\checkmark}$. Since $P(i)$ *RespondsTo* Q, there must exist some $e \in (\alpha P(i) \cap \alpha Q)$ such that $e \in \textit{initials}(P/s_i)$ and $e \in \alpha P(i) \setminus X_i$ and $e \notin Y$, which contradicts the assumption that $P(i)$ *RespondsTo* Q.

□

C.3 Proofs for Section 6.6

Lemma (6.15). *Given two consecutive time stable states X and Y, such that $X \Longrightarrow Y$, we have $pg(X, Y)$.*

Proof. We let Z denote the subsequent timed state of X such that $X \xrightarrow{\tau} Z$. By Rule A-E, $dur(X) = minimum(\{e : Os \cup Er \bullet min(e)\} \cup \{e : De(X) \bullet max(e)\})$, where $Er' = \{m : M_f(X) \bullet second(splitsq(m))\}$, where $Os = Pp(X) \setminus (M_n(X) \cup M_p(X) \cup M_d(X))$, $Er = ((Fr(X) \cup (\bigcup\{e : Fr(X) \bullet exp(assoc(e))\} \setminus Pp(X))) \setminus M_f(X)) \cup \{m : M_f(X) \cup M_n(X) \bullet second(splitsq(m))\}$.

If $dur(X) = minimum\{e : OS \bullet min(e)\}$, then OS is not empty and at least one postponed element $e \in Pp(X)$ becomes enactable at state Z, therefore $Pp(X) \not\subseteq Pp(Y)$, satisfying Condition 1 of time progression.

If $dur(X) = minimum\{e : De(X) \bullet max(e)\}$, then $De(X)$ is not empty, at least one delayed element $e \in D(X)$ becomes enactable and can no longer be delayed at state Z, therefore $De(X) \not\subseteq De(Y)$, satisfying Condition 2 of time progression.

If $dur(X) = minimum\{e : Er \bullet min(e)\}$, then either $e \in Fr(X)$ becomes enactable at state Z, therefore $Fr(X) \not\subseteq Pp(Y)$ satisfying Condition 3 of time progression, or $e \in \{m : M_n(X) \bullet second(splitsq(m))\}$ becomes enactable at state Z, and we have $Pp(X) \not\subseteq Pp(Y)$, satisfying Condition 1 of time progression, or $e \in \{m : M_f(X) \bullet second(splitsq(m))\}$ becomes enactable at state Z, therefore $M_f(X) \not\subseteq Pp(Y)$, satisfying Condition 3 of time progression. □

To prove Theorem 6.16, we use the following abbreviations from Section 5.7 on Page 77: We write $P(s)$ to denote the CSP process modelling BPMN element s's untimed behaviour; we define $alphas(es) = \bigcup\{i : es \bullet \alpha P(i)\}$ to be the alphabet of the process semantics of elements in es, $procs(es) = \|\, i : es \bullet \alpha P(i) \circ P(i)$ to be the parallel composition of processes, each modelling the behaviour of an element in es. We write $init(P) = \{a : \alpha P \mid \langle a \rangle \in traces(P)\}$ for the set of P's initial events, $s \downarrow u = \#(s \upharpoonright \{u\})$ for the number of occurrences of u in sequence s, $bal(s, es) \Leftrightarrow \forall\, e, f \in es \bullet s \downarrow e = s \downarrow f$ if there is an equal number of occurrences between events of es in sequence s, and define $sub(s, es)$ as follows:

$$sub(s, es) = \begin{cases} \langle\rangle & s = \langle\rangle \vee last(s) \in es \\ sub(front(s), es) \frown \langle last(s) \rangle & \text{otherwise} \end{cases}$$

Specifically, $sub(s, es)$ returns either sequence s or the shortest suffix of s not containing events in es. We provide the following abbreviations.

$$miseqs'(e) \Leftrightarrow (atom\ e).type \in \operatorname{ran} miseqs \qquad miseq'(e) \Leftrightarrow (atom\ e).type \in \operatorname{ran} miseq$$
$$itime'(e) \Leftrightarrow (atom\ e).type \in \operatorname{ran} itime \qquad task'(e) \Leftrightarrow (atom\ e).type \in \operatorname{ran} task$$
$$sf(es) = \{f : es \bullet s.f\}$$

We now provide the following characterisation relating a state in the relative timed coordination of some BPMN pool p to p's enactment process.

Definition C.11. *Let Z be the initial state, and X be a state of some BPMN pool p's relative timed coordination such that $Z \xRightarrow{s} X$ where $s \in traces(PL(p))$ is a trace of p's enactment process, if $init(PL(p)/s) \neq \emptyset$ then either for all CSP events $e \in init(PL(p)/s)$ such that $(s, \{e\}) \in failures(PL(p))$ and X satisfies one of the following Conditions 1 and 2. or there exists some CSP event $e \in init(PL(p)/s)$ such that $(s, \{e\}) \notin failures(PL(p))$ and X satisfies one of the following Conditions 1 and 2.*

(1) $X \xrightarrow{e}$, or

(2) there exists Y such that $X \xRightarrow{\tau*} Y$ and $Y \xrightarrow{e}$, where $\tau*$ denotes a finite sequence of τ events.

In addition, state X satisfies one of the following Conditions a and b:

(a) If $X = (S, M, E, W, T)$ is an untimed state, then Equation C.3 holds.

(b) If $X = (S, M, E, W, P, C, O)$ is a timed state, then Equation C.4 holds

$$\begin{aligned} init(PL(p)/s) = \bigcup(&\{t : T \mid \neg\, task'(c) \wedge \neg\, miseq'(c) \bullet out(c)\} \cup \\ &\{t : T \cup E \cup W \bullet sf(\mathrm{dom}\, assoc(t))\} \cup \\ &\{t : T \mid miseq'(t) \vee task'(t) \bullet work(t)\} \cup exit(\!|\, \mathrm{dom}\, M \,|\!) \cup \\ &\{o : E \mid miseqs'(o) \Rightarrow M(o) = 0 \bullet out(o)\} \cup \\ &\{o : E;\ t : Start \mid miseqs'(o) \wedge M(o) > 0 \wedge t.ele \in content(o) \bullet out(t.ele)\}) \end{aligned} \tag{C.3}$$

$$\begin{aligned} init(PL(p)/s) = \bigcup(&out(\!|\, E \cup O \,|\!) \cup exit(\!|\, \mathrm{dom}\, M \,|\!) \cup \{c : C \cup P \mid task'(c) \vee miseq'(c) \bullet work(c)\} \cup \\ &\{c : C \cup P \mid \neg\, task'(c) \wedge \neg\, miseq'(c) \bullet out(c)\} \cup \\ &\{o : E \cup W \bullet sf(\mathrm{dom}\, assoc(o))\}) \end{aligned} \tag{C.4}$$

Lemma C.12. *For all states X, Y of some BPMN pool p's relative timed coordination such that $X \xrightarrow{\tau} Y$ is a transition defined by Rule* A-E*. X satisfies Equation C.3 if and only if Y satisfies Equation C.4.*

Proof.

$$\begin{aligned} &init(PL(p)/s) && \text{[Equation C.4]} \\ &= \bigcup(out(\!|\, E \cup O \,|\!) \cup exit(\!|\, \mathrm{dom}\, M \,|\!) \cup \{c : C \cup P \mid task'(c) \vee miseq'(c) \bullet work(c)\} \cup \\ &\quad \{c : C \cup P \mid \neg\, task'(c) \wedge \neg\, miseq'(c) \bullet out(c)\} \cup \{o : E \cup W \bullet sf(\mathrm{dom}\, assoc(o))\}) \\ &= && \text{[Rule A-E, } E \cup O = \emptyset,\ \{e : \{M_n(X) \cup M_f(X) \bullet first(splitsq(m))\} \bullet out(e)\} = \emptyset] \\ &\quad [\{e : \{M_n(X) \cup M_f(X) \bullet first(splitsq(m))\} \bullet work(e)\} = \{e : M_n(X) \cup M_f(X) \bullet work(e)\}] \\ &\quad \bigcup(exit(\!|\, \mathrm{dom}\, M \,|\!) \cup \\ &\quad \{c : As \cup Ac \cup M_p(X) \cup M_d(X) \cup M_n(X) \cup M_f(X) \mid task'(c) \vee miseq'(c) \bullet work(c)\} \cup \\ &\quad \{c : As \cup Ac \cup M_p(X) \cup M_d(X) \mid \neg\, task'(c) \wedge \neg\, miseq'(c) \bullet out(c)\} \cup \\ &\quad \{o : W \bullet sf(\mathrm{dom}\, assoc(o))\}) \\ &= \bigcup(exit(\!|\, \mathrm{dom}\, M \,|\!) \cup \{c : As \cup M_p(X) \cup M_d(X) \cup M_n(X) \cup M_f(X) \mid && [AC \subseteq AS] \\ &\quad task'(c) \vee miseq'(c) \bullet work(c)\} \cup \\ &\quad \{c : As \cup M_p(X) \cup M_d(X) \mid \neg\, task'(c) \wedge \neg\, miseq'(c) \bullet out(c)\} \cup \\ &\quad \{o : W \bullet sf(\mathrm{dom}\, assoc(o))\}) \\ &= && \text{[Def of } AS] \\ &\quad \bigcup(exit(\!|\, \mathrm{dom}\, M \,|\!) \cup \{c : Os \cup Er \cup De(X) \cup M_p(X) \cup M_d(X) \cup M_n(X) \cup M_f(X) \mid \\ &\quad task'(c) \vee miseq'(c) \bullet work(c)\} \cup \\ &\quad \{c : Os \cup Er \cup De(X) \cup M_p(X) \cup M_d(X) \mid \neg\, task'(c) \wedge \neg\, miseq'(c) \bullet out(c)\} \cup \\ &\quad \{o : W \bullet sf(\mathrm{dom}\, assoc(o))\}) \\ &= && \text{[Def of } OS \text{ and } M_p(X) \cup M_d(X) \cup M_n(X) \subseteq Pp(X)] \\ &\quad \bigcup(exit(\!|\, \mathrm{dom}\, M \,|\!) \cup \{c : Pp(X) \cup Er \cup De(X) \cup M_f(X) \mid task'(c) \vee miseq'(c) \bullet work(c)\} \cup \\ &\quad \{c : Pp(X) \cup Er \cup De(X) \mid \neg\, task'(c) \wedge \neg\, miseq'(c) \bullet out(c)\} \cup \\ &\quad \{o : W \bullet sf(\mathrm{dom}\, assoc(o))\}) \\ &= && \text{[Def of } Er \text{ and } M_n(X) \subseteq Pp(X)] \\ &\quad \bigcup(exit(\!|\, \mathrm{dom}\, M \,|\!) \cup \{c : Pp(X) \cup (Nr \cup \{m : M_f(X) \bullet second(splitsq(m))\}) \cup De(X) \cup M_f(X) \mid \\ &\quad task'(c) \vee miseq'(c) \bullet work(c)\} \cup \\ &\quad \{c : Pp(X) \cup ((Nr \setminus M_f(X)) \cup \{m : M_f(X) \bullet second(splitsq(m))\}) \cup De(X) \mid \\ &\quad \neg\, task'(c) \wedge \neg\, miseq'(c) \bullet out(c)\} \cup \{o : W \bullet sf(\mathrm{dom}\, assoc(o))\}) \\ &= && \text{[Def of } Nr,\ \{e : \bigcup\{e : Fr(X) \bullet exp(assoc(e))\} \bullet work(e)\} = \emptyset] \\ &\quad \bigcup(exit(\!|\, \mathrm{dom}\, M \,|\!) \cup \end{aligned}$$

$$
\begin{aligned}
&\{c : Pp(X) \cup (Fr(X) \cup \{m : M_f(X) \bullet second(splitsq(m))\}) \cup De(X) \cup M_f(X) \mid \\
&\quad task'(c) \vee miseq'(c) \bullet work(c)\} \cup \\
&\{c : Pp(X) \cup (((Fr(X) \cup \bigcup\{e : Fr(X) \bullet exp(assoc(e))\}) \setminus M_f(X)) \cup \\
&\quad \{m : M_f(X) \bullet second(splitsq(m))\}) \cup De(X) \mid \neg\, task'(c) \wedge \neg\, miseq'(c) \bullet out(c)\} \cup \\
&\{o : W \bullet sf(\operatorname{dom} assoc(o))\}) \\
= \quad & [\{e : Fr(X) \cup \{m : M_f(X) \bullet second(splitsq(m))\} \bullet work(e)\} = \{e : Fr(X) \bullet work(e)\}] \\
\bigcup(exit(\!| \operatorname{dom} M |\!)) \cup \quad & [M_f(X) \subseteq Fr(X)] \\
&\{c : Pp(X) \cup Fr(X) \cup De(X) \mid task'(c) \vee miseq'(c) \bullet work(c)\} \cup \\
&\{c : Pp(X) \cup (((Fr(X) \cup \bigcup\{e : Fr(X) \bullet exp(assoc(e))\}) \setminus M_f(X)) \cup \\
&\quad \{m : M_f(X) \bullet second(splitsq(m))\}) \cup De(X) \mid \neg\, task'(c) \wedge \neg\, miseq'(c) \bullet out(c)\} \cup \\
&\{o : W \bullet sf(\operatorname{dom} assoc(o))\}) \\
= \quad & [\{e : Fr(X) \cup \{m : M_f(X) \bullet second(splitsq(m))\} \bullet out(e)\} = \{e : Fr(X) \bullet out(e)\}] \\
\bigcup(exit(\!| \operatorname{dom} M |\!)) \cup & \\
&\{c : Pp(X) \cup Fr(X) \cup De(X) \mid task'(c) \vee miseq'(c) \bullet work(c)\} \cup \\
&\{c : Pp(X) \cup ((Fr(X) \cup \bigcup\{e : Fr(X) \bullet exp(assoc(e))\}) \setminus M_f(X)) \cup De(X) \mid \\
&\quad \neg\, task'(c) \wedge \neg\, miseq'(c) \bullet out(c)\} \cup \\
&\{o : W \bullet sf(\operatorname{dom} assoc(o))\}) \\
= \quad & [\{e : Fr(X) \cup \{m : M_f(X) \bullet second(splitsq(m))\} \bullet out(e)\} = \{e : Fr(X) \bullet out(e)\}] \\
& [\{e : M_f(X) \bullet out(e)\} = \emptyset] \\
\bigcup(exit(\!| \operatorname{dom} M |\!)) \cup \{c : Pp(X) \cup Fr(X) \cup De(X) \mid task'(c) \vee miseq'(c) \bullet work(c)\} \cup & \\
&\{c : Pp(X) \cup (Fr(X) \cup \bigcup\{e : Fr(X) \bullet exp(assoc(e))\}) \cup De(X) \mid \\
&\quad \neg\, task'(c) \wedge \neg\, miseq'(c) \bullet out(c)\} \cup \\
&\{o : W \bullet sf(\operatorname{dom} assoc(o))\}) \\
= \quad & [\langle De(X), Pp(X), Fr(X)\rangle \text{ partition } T] \\
\bigcup(exit(\!| \operatorname{dom} M |\!)) \cup \{t : T \mid miseq'(t) \vee task'(t) \bullet work(t)\} \cup \{t : T \cup W \bullet sf(\operatorname{dom} assoc(t))\} \cup & \\
&\{t : T \mid \neg\, task'(c) \wedge \neg\, miseq'(c) \bullet out(c)\}) \\
= \quad & init(PL(p)/s) \qquad \text{[Equation C.3]}
\end{aligned}
$$

□

Theorem (6.16)**.** ***Deadlock Freedom*** *Given any BPMN pool p, such that if process $PL(p)$, modelling p's untimed behaviour, is deadlock free, then process $PL(p) \,[\![\, \Sigma \,]\!]\, CP(p)$ is also deadlock free.*

Proof. We prove by showing all reachable states in the relative timed coordination satisfy Definition C.11. This is achieved by induction on the sequence of states in the relative timed coordination.

Base case We consider the initial state of the relative timed coordination. According to the syntactic assumptions defined in Section 6.2.2 on Page 103, only one start event is directly contained in p. According to the untimed semantics, $init(PL(p))$ is defined as follows:

$$init(PL(p)) = \bigcup\{s : S \mid s \in \{Start \bullet ele\} \bullet out(s)\} \tag{C.5}$$

Let v be that start event, and $initials(PL(p)) = \{e\}$ is a singleton, where $e \in out(v)$. Let R be the initial state of the relative timed coordination. By definition of untimed states on Page 104, $R = (S, \emptyset, I, \emptyset, J)$ where $I = S \cap \{Start \bullet ele\}$ and $J = S \cap \{Start \mid (atom\ ele).type \in \operatorname{ran} stime \bullet ele\}$. By definition of the untimed semantics (Equation C.5) we know $I \cup J \neq \emptyset$.

1. If $I \neq \emptyset$, then by definition of transition rules shown on Figure 6.4 on Page 106, $R \xrightarrow{e}$, satisfying Condition 1. Condition a is satisfied due to Equation C.5.

2. If $I = \emptyset$, $J = \{v\}$. $R = (S, \emptyset, \emptyset, \emptyset, \{v\})$ is a time stable state. By Rule A-E, $R \xrightarrow{\tau} R'$, where $R' = (S, \emptyset, \emptyset, \emptyset, \emptyset, \{v'\}, \emptyset)$ is a timed state, and $v' = dmin(v, min(v))$. By definition of transition rules shown on Figure 6.17 on Page 117, $R' \xrightarrow{e}$, satisfying Condition 2. Again, Condition a is satisfied due to Equation C.5.

Inductive case Given initial state R, and some state X satisfying Definition C.11 and such that $R \stackrel{s}{\Longrightarrow} X$ where $s \in traces(PL(p))$, we show either if $X \xrightarrow{\tau} Y$ or $X \xrightarrow{f} Y$ where $f \in init(PL(p)/s)$, Y satisfies Definition C.11. According to the restriction of the relative timed coordination (Page 103), we consider the following cases:

(a) If $f \in out(w)$, such that w is either an AND split gateway where $bal(sub(s \frown \langle f \rangle, \mathrm{dom}\, assoc(w)), out(w))$, that is all of w's outgoing sequence flows have been triggered, or a XOR split gateway, or a join gateway, or a start event, then by definition of transition rules shown in Figure 6.4 on Page 106, $X = (S, M, E, W, T)$ is an untimed state and $w \in E$. We consider the types of element u can be such that $f \in in(u)$.

(i) If u is an untimed atomic element, such that if u is an AND join gateway, we have $bal(sub(s \frown \langle f \rangle, \mathrm{dom}\, assoc(u)), in(u))$, that is all of u's incoming sequence flows have been triggered, then by Rule U-U, $Y = (S, M, (E \setminus \{w\}) \cup \{u\}, W \setminus \{u\}, T)$ and by induction Y satisfies Conditions 1 and a.

(ii) If u is a subprocess containing a start event t, by Rule U-U, $Y = (S, M, (E \setminus \{w\}) \cup \{t\}, W \setminus \{t\}, T)$, and by induction Y satisfies Conditions 1 and a.

(iii) If u is an AND join gateway and $\neg bal(sub(s \frown \langle f \rangle, \mathrm{dom}\, assoc(u)), in(u))$, that is not all of u's incoming sequence flows have been triggered, then $out(u) \notin init(PL(p)/s \frown \langle f \rangle)$. By Rule U-J, $Y = (S, M, E \setminus \{w\}, (W \setminus \{u\}) \cup \{rm(u, f)\}, T)$. By induction Y satisfies Condition a.

i. If $E = \{w\}$ then Y is a time stable state. By Rule A-E and Lemma 6.15, at least one $t \in T$ such that $b \in out(t) \cup work(t) \cup exit(t)$ and $Y \stackrel{\tau*}{\Longrightarrow} Y' \xrightarrow{b}$, which satisfies Condition 2.

ii. If $\{w\} \subset E$ then by induction, Y satisfies Condition 1.

(iv) If u is a task or multiple instance task element, by Rule U-T, $Y = (S, M, E \setminus \{w\}, W, T \cup \{u\})$. By induction Y satisfies Condition a. Follow from a(iii)i and a(iii)ii above, Y satisfies one of Conditions 1 and 2.

(v) If u is a subprocess containing a start timer event t, by Rule U-T, $Y = (S, M, E \setminus \{w\}, W, T \cup \{t\})$. By induction Y satisfies Condition a. Follow from a(iii)i and a(iii)ii above, Y satisfies one of Conditions 1 and 2.

(vi) If u is a multiple instance subprocess directly containing some start event t and specifies l instances, then by Rule U-M, $Y = (S, M \cup \{(u, l-1)\}, (E \setminus \{w\}) \cup \{t\}, W, T)$. By induction Y satisfies Conditions 1 and a.

(vii) If u is a multiple instance subprocess directly containing some start timer event t and specifies l instances, then by Rule U-M', $Y = (S, M \cup \{(u, l-1)\}, E \setminus \{w\}, W, T \cup \{t\})$. By induction Y satisfies Condition a. Follow from a(iii)i and a(iii)ii above, Y satisfies one of Conditions 1 and 2.

(b) If $f \in out(w)$ for some AND split gateway w such that $\neg bal(sub(s \frown \langle f \rangle, \mathrm{dom}\, assoc(w)), out(w))$, then by definition of transition rules shown in Figure 6.3 on Page 105, $X = (S, M, E, W, T)$ is an untimed state and $w \in E$. We consider the types of element u can be such that $f \in in(u)$.

(i) If u is an untimed atomic element, such that if u is an AND join gateway, we have $bal(sub(s \frown \langle f \rangle, \mathrm{dom}\, assoc(u)), in(u))$, then by Rule S-U, $Y = (S, M, (E \setminus \{w\}) \cup \{rm(f, w), u\}, W \setminus \{u\}, T)$ and by induction Y satisfies Conditions 1 and a.

(ii) If u is a subprocess containing a start event t, by Rule S-U, $Y = (S, M, (E \setminus \{w\}) \cup \{rm(f, w), t\}, W \setminus \{t\}, T)$, and by induction Y satisfies Conditions 1 and a.

(iii) If u is an AND join gateway and $\neg bal(sub(s \frown \langle f \rangle, \mathrm{dom}\, assoc(u)), in(u))$, then $out(u) \notin init(PL(p)/s \frown \langle f \rangle)$. By Rule S-J, $Y = (S, M, (E \setminus \{w\}) \cup \{rm(f, w)\}, (W \setminus \{u\}) \cup \{rm(u, f)\}, T)$. By induction Y satisfies Conditions 1 and a.

(iv) If u is a task or multiple instance task element, by Rule S-T, $Y = (S, M, (E \setminus \{w\}) \cup \{rm(f, w)\}, W, T \cup \{u\})$. By induction Y satisfies Conditions 1 and a.

(v) If u is a multiple instance subprocess directly containing some start event t and specifies l instances, then by Rule S-M, $Y = (S, M \cup \{(u, l-1)\}, (E \setminus \{w\}) \cup \{rm(f, w), t\}, W, T)$. By induction Y satisfies Conditions 1 and a.

(vi) If u is a multiple instance subprocess directly containing some start timer event t and specifies l instances, then by Rule S-M', $Y = (S, M \cup \{(u, l-1)\}, (E \setminus \{w\}) \cup \{rm(f, w)\}, W, T \cup \{t\})$. By induction Y satisfies Conditions 1 and a.

(c) If $f \in out(w)$ for some compound element w, then by definition of transition rules shown in Figure 6.7 on Page 108, $X = (S, M, E, W, T)$ is an untimed state, $w \in E$. If w is a multiple instance subprocess that directly contains some start event x and specifies l instances, then by induction we have $(sub(s \frown \langle y \rangle, \mathrm{dom}\ assoc(w)) \downarrow y) \bmod l = 0$, where $y \in out(x)$ (all instances have been executed). We consider the types of element u can be such that $f \in in(u)$.

(i) If u is an untimed atomic element, such that we have $bal(sub(s \frown \langle f \rangle, \mathrm{dom}\ assoc(u)), in(u))$ if u is an AND join gateway, then by Rule U-U, $Y = (S, M \setminus \{(w, 0)\}, (E \setminus \{w\}) \cup \{u\}, W \setminus \{u\}, T)$ and by induction Y satisfies Conditions 1 and a.

(ii) If u is a subprocess containing a start event t, by Rule U-U, $Y = (S, M \setminus \{(w, 0)\}, (E \setminus \{w\}) \cup \{t\}, W \setminus \{t\}, T)$, and by induction Y satisfies Conditions 1 and a.

(iii) If u is an AND join gateway and $\neg bal(sub(s \frown \langle f \rangle, \mathrm{dom}\ assoc(u)), in(u))$, then $out(u) \notin init(PL(p)/s \frown \langle f \rangle)$. By Rule U-J, $Y = (S, M \setminus \{(w, 0)\}, E \setminus \{w\}, (W \setminus \{u\}) \cup \{rm(u, f)\}, T)$. By induction Y satisfies Condition a. Follow from a(iii)i and a(iii)ii above, Y satisfies one of Conditions 1 and 2.

(iv) If u is a task or multiple instance task element, by Rule U-T, $Y = (S, M \setminus \{(w, 0)\}, E \setminus \{w\}, W, T \cup \{u\})$. By induction Y satisfies Condition a. Follow from a(iii)i and a(iii)ii above, Y satisfies one of Conditions 1 and 2.

(v) If u is a multiple instance subprocess directly containing some start event t and specifies l instances, then by Rule U-M, $Y = (S, (M \setminus \{(w, 0)\}) \cup \{(u, l-1)\}, (E \setminus \{w\}) \cup \{t\}, W, T)$. By induction Y satisfies Conditions 1 and a.

(vi) If u is a multiple instance subprocess directly containing some start timer event t and specifies l instances, then by Rule U-M', $Y = (S, (M \setminus \{(w, 0)\}) \cup \{(u, l-1)\}, E \setminus \{w\}, W, T \cup \{t\})$. By induction Y satisfies Condition a. Follow from a(iii)i and a(iii)ii above, Y satisfies one of Conditions 1 and 2.

(d) If f is an completion of some end event w, then by definition of transition rules shown in Figure 6.5 on Page 107, $X = (S, M, E, W, T)$ is an untimed state and $w \in E$. There are two cases in which f can be performed: 1) w is directly contained in some compound element c, that is contained in p, or 2) w is directly contained in p. For case 1, by Rule E-M, $Y = (S, M, (E \setminus \{e\}) \cup \{c\}, W, T)$ and by induction Y satisfies Conditions 1 and a. For case 2, by Rule E-E, $X = (S, \emptyset, \{e\}, \emptyset, \emptyset)$ and $Y = (S, \emptyset, \emptyset, \emptyset, \emptyset)$. By induction, $init(PL(p)/s) = \{f\}$ and, by the untimed semantics of p, for all elements i in p, $P(i)$ must be willing to perform f and that after performing f, $P(i)$ becomes *Skip*. Hence Y satisfies Conditions 1 and a.

(e) If $f \in out(w)$ for some timer event w, then by definition of transition rules shown in Figure 6.17 on Page 117, $X = (S, M, E, W, P, C, O)$ is a timed state, $w \in C$. We consider the types of element u can be such that $f \in in(u)$.

(i) If u is an untimed atomic element, such that if u is an AND join gateway, then we have $bal(sub(s \frown \langle f \rangle, \mathrm{dom}\ assoc(u)), in(u))$, then by Rule V-U, $Y = (S, M, E \cup \{u\}, W \setminus \{u\}, P, C \setminus \{w\}, O)$, and by induction Y satisfies Condition b.

i. If $C = \{w\}$ and $O = \emptyset$, then by Rule T-C, the only transition is $Y \xrightarrow{\tau} Y'$, where $Y' = (S, M, E \cup \{u\}, W \setminus \{u\}, P)$ is an untimed state. By induction $Y' \xrightarrow{e}$ where $e \in out(e)$, satisfying Condition 2.

ii. If $\{w\} \subset C \cup O$ then by induction, Y satisfies Condition 1.

(ii) If u is a subprocess containing a start event t, by Rule V-U, $Y = (S, M, E \cup \{t\}, W, P, C \setminus \{w\}, O)$. By induction Y satisfies Condition b, and follow from e(i)i and e(i)ii, Y satisfies one of Conditions 1 and 2.

(iii) If u is an AND join gateway and $\neg bal(sub(s \frown \langle f \rangle, \mathrm{dom}\, assoc(u)), in(u))$, then $out(u) \notin init(PL(p)/s \frown \langle f \rangle)$. By Rule V-J, $Y = (S, M, E, (W \setminus \{u\}) \cup \{rm(u, f)\}, P, C \setminus \{w\}, O)$, and by induction Y satisfies Condition b.

i. If $E \neq \emptyset$, then follow from e(i)i and e(i)ii, Y satisfies one of Conditions 1 and 2.

ii. If $C = \{w\}$ and $O \cup E = \emptyset$ then by induction $init(PL(p)/s \frown \langle f \rangle) \neq \emptyset \Rightarrow P \neq \emptyset$ and by Rule T-C, there is only one transition $Y \xrightarrow{\tau} Y'$ such that Y' is a time stable state. By Rule A-E and Lemma 6.15, at least one $t \in P$ such that $b \in out(t) \cup work(t) \cup exit(t)$ and we have $Y' \xRightarrow{\tau*} Y'' \xrightarrow{b}$, therefore Y satisfies Condition 2.

(iv) If u is a task or a multiple instance task element, by Rule V-T, $Y = (S, M, E, W, P \cup \{u\}, C \setminus \{w\}, O)$. By induction Y satisfies Condition b, and follow from e(i)i and e(i)ii, Y satisfies one of Conditions 1 and 2.

(v) If u is a subprocess containing a start timer event t, by Rule V-T, $Y = (S, M, E, W, P \cup \{t\}, C \setminus \{w\}, O)$. By induction Y satisfies Condition b, and follow from e(i)i and e(i)ii, Y satisfies one of Conditions 1 and 2.

(vi) If u is a multiple instance subprocess directly containing some start event t and specifies l instances, then by Rule V-M, $Y = (S, M \cup \{(u, l-1)\}, E \cup \{t\}, W, P, C \setminus \{w\}, O)$. By induction Y satisfies Condition b, and follow from e(i)i and e(i)ii, Y satisfies one of Conditions 1 and 2.

(vii) If u is a multiple instance subprocess directly containing some start timer event t and specifies l instances, then by Rule V-M', $Y = (S, M \cup \{(u, l-1)\}, E, W, P \cup \{t\}, C \setminus \{w\}, O)$. By induction Y satisfies Condition b, and follow from e(i)i and e(i)ii, Y satisfies one of Conditions 1 and 2.

(f) If $f \in work(w)$, where w is either a task or a multiple instance task, then by definition of transition rules shown in Figure 6.18 on Page 118, $X = (S, M, E, W, P, C, O)$ is a timed state. 1) If w is a task, then by Rule T-E $w \in C$ and $Y = (S, M, E, W, P, C \setminus \{w\}, O \cup \{w\})$, and Y satisfies Conditions 1 and b. 2) If w is a multiple instance task, then by Rules T-M and T-M', $w \in P$ and $t \in C$, where t is a task instance of w. We let l to denote the number of instances defined by w and consider the following two cases:

(i) If $(sub(s \frown \langle f \rangle, \mathrm{dom}\, assoc(w)) \downarrow f) \bmod l = 0$, then $out(w) \in init(PL(p)/s \frown \langle f \rangle)$. By induction and Rule T-M, $Y = (S, M, E, W, P \setminus \{w\}, C \setminus \{t\}, O \cup \{w\})$, satisfying Conditions 1 and b.

(ii) If $(sub(s \frown \langle f \rangle, \mathrm{dom}\, assoc(w)) \downarrow f) \bmod l \neq 0$, then w has remaining instances and $out(w) \notin initial(PL(p)/s \frown \langle f \rangle)$. By induction and Rule T-M', $Y = (S, M, E, W, P \setminus \{w\}, C \setminus \{t\}, O)$, satisfying Conditions 1 and b.

(g) If $f \in out(w)$ where w is either a task or a multiple instance task, then by definition of transition rules shown in Figure 6.19 on Page 120, $X = (S, M, E, W, P, C, O)$ is a timed state, such that $w \in O$. We consider the type of element u can be such that $f \in in(u)$.

(i) If u is an untimed atomic element, such that we have $bal(sub(s \frown \langle f \rangle, \mathrm{dom}\, assoc(u)), in(u))$ if u is an AND join gateway, then by Rule T-U, $Y = (S, M, E \cup \{u\}, W \setminus \{u\}, P \setminus ex(w, P), C \setminus ex(w, C), O \setminus \{w\})$. By induction Y satisfies Condition b, and follow from e(i)i and e(i)ii, Y satisfies one of Conditions 1 and 2.

(ii) If u is a subprocess containing a start event t, by Rule T-U, $Y = (S, M, E \cup \{t\}, W, P \setminus ex(w, P), C \setminus ex(w, C), O \setminus \{w\})$, and by induction Y satisfies Condition b, and follow from e(i)i and e(i)ii, Y satisfies one of Conditions 1 and 2.

(iii) If u is an AND join gateway and $\neg bal(sub(s \frown \langle f \rangle, \mathrm{dom}\, assoc(u)), in(u))$, then $out(u) \notin init(PL(p)/s \frown \langle f \rangle)$. By Rule T-J, $Y = (S, M, E, (W \setminus \{u\}) \cup \{rm(u, f)\}, P \setminus ex(w, P), C \setminus ex(w, C), O \setminus \{w\})$. By induction Y satisfies Condition b, and follow from e(iii)i and e(iii)ii, Y satisfies one of Conditions 1 and 2.

(iv) If u is a task or multiple instance task element, by Rule T-T, $Y = (S, M, E, W, (P \setminus ex(w, P)) \cup \{u\}, C \setminus ex(w, C), O \setminus \{w\})$. By induction Y satisfies Condition b, and follow from e(i)i and e(i)ii, Y satisfies one of Conditions 1 and 2.

(v) If u is a subprocess containing a start timer event t, by Rule T-T, $Y = (S, M, E, W, (P \setminus ex(w, P)) \cup \{t\}, C \setminus ex(w, C), O \setminus \{w\})$. By induction Y satisfies Condition b, and follow from e(i)i and e(i)ii, Y satisfies one of Conditions 1 and 2.

(vi) If u is a multiple instance subprocess directly containing some start event t and specifies l instances, then by Rule T-M, $Y = (S, M \cup \{(u, l-1)\}, E \cup \{t\}, W, P \setminus ex(w, P), C \setminus ex(w, C), O \setminus \{w\})$. By induction Y satisfies Condition b, and follow from e(i)i and e(i)ii, Y satisfies one of Conditions 1 and 2.

(vii) If u is a multiple instance subprocess directly containing some start timer event t and specifies l instances, then by Rule T-M', $Y = (S, M \cup \{(u, l-1)\}, E, W, P \setminus ex(w, P), C \setminus ex(w, C), O \setminus \{w\})$. By induction Y satisfies Condition b, and follow from e(i)i and e(i)ii, Y satisfies one of Conditions 1 and 2.

(h) If f is a timed exception flow, then by definition of transition rules shown in Figure 6.20 on Page 121, $X = (S, M, E, W, P, C, O)$ is a timed state and there exists an intermediate timer event $w \in C$ such that $f \in out(w)$. We let A, U and V be abbreviations for sets $assocs(f, t)$, $mults(f, t)$ and $timers(f, t)$, and consider the types of element u can be such that $f \in in(u)$.

(i) If u is an untimed atomic element, such that we have $bal(sub(s \frown \langle f \rangle, \mathrm{dom}\, assoc(u)), in(u))$ if u is an AND join gateway, then by Rule X-U, $Y = (S, M \setminus U, (E \setminus A) \cup \{u\}, W \setminus (\{u\} \cup A), P \setminus (A \cup V), C \setminus (A \cup V), O \setminus A)$. By induction Y satisfies Condition b, and follow from e(i)i and e(i)ii, Y satisfies one of Conditions 1 and 2.

(ii) If u is a subprocess containing a start event t, by Rule X-U, $Y = (S, M \setminus U, (E \setminus A) \cup \{t\}, W \setminus A, P \setminus (A \cup V), C \setminus (A \cup V), O \setminus A)$, and by induction Y satisfies Condition b, and follow from e(i)i and e(i)ii, Y satisfies one of Conditions 1 and 2.

(iii) If u is an AND join gateway and $\neg bal(sub(s \frown \langle f \rangle, \mathrm{dom}\, assoc(u)), in(u))$, then $out(u) \notin init(PL(p)/s \frown \langle f \rangle)$. By Rule X-J, $Y = (S, M \setminus U, E \setminus A, (W \setminus (\{u\} \cup A)) \cup \{rm(u, f)\}, P \setminus (A \cup V), C \setminus (A \cup V), O \setminus A)$. By induction Y satisfies Condition b, and follow from e(iii)i and e(iii)ii, Y satisfies one of Conditions 1 and 2.

(iv) If u is a task or multiple instance task element, by Rule X-T, $Y = (S, M \setminus U, E \setminus A, W \setminus A, (P \setminus (A \cup V)) \cup \{u\}, C \setminus (A \cup V), O \setminus A)$. By induction Y satisfies Condition b, and follow from e(i)i and e(i)ii, Y satisfies one of Conditions 1 and 2.

(v) If u is a subprocess containing a start timer event t, by Rule X-T, $Y = (S, M \setminus U, E \setminus A, W \setminus A, (P \setminus (A \cup V)) \cup \{t\}, C \setminus (A \cup V), O \setminus A)$. By induction Y satisfies Condition b, and follow from e(i)i and e(i)ii, Y satisfies one of Conditions 1 and 2.

(vi) If u is a multiple instance subprocess directly containing some start event t and specifies l instances, then by Rule X-M, $Y = (S, (M \setminus U) \cup \{(u, l-1)\}, (E \setminus A) \cup \{t\}, W \setminus A, P \setminus (A \cup V), C \setminus (A \cup V), O \setminus A)$. By induction Y satisfies Condition b, and follow from e(i)i and e(i)ii, Y satisfies one of Conditions 1 and 2.

(vii) If u is a multiple instance subprocess directly containing some start timer event t and specifies l instances, then by Rule X-M', $Y = (S, (M \setminus U) \cup \{(u, l-1)\}, E \setminus A, W \setminus A, (P \setminus (A \cup V)) \cup \{t\}, C \setminus (A \cup V), O \setminus A)$. By induction Y satisfies Condition b, and follow from e(i)i and e(i)ii, Y satisfies one of Conditions 1 and 2.

(i) The transition $X \xrightarrow{\tau} Y$ can be triggered by one of Rules L-U, L-T, A-E, T-D and T-C. We consider them individually.

(i) By Rules L-U and L-T shown in Figure 6.8 on Page 108, $X = (S, M, E, W, T)$ is an untimed state and the rules specify τ-transitions triggered by some multiple task instance w, which directly contains some start event u, and specifies l instances, such that $(sub(s \frown \langle e \rangle, \mathrm{dom}\, assoc(w)) \downarrow e) \bmod l \neq 0$, where $e \in out(u)$. By induction $w \in E$ and $e \in init(PL(p)/s)$.

i. If u is a start event, then by Rule L-U, $Y = (S, M \oplus \{w \mapsto M(w) - 1\}, (E \setminus \{w\}) \cup \{u\}, W, T)$, satisfying Conditions 2 and a.

ii. If u is a start timer event, then by Rule L-T, $Y = (S, M \oplus \{w \mapsto M(w) - 1\}, E \setminus \{w\}, W, T \cup \{u\})$, satisfying Condition a. Moreover, follow from a(iii)i and a(iii)ii above, Y satisfies one of Conditions 1 and 2.

(ii) By Rule A-E shown in Figure 6.14 on Page 114, $X = (S, M, \emptyset, W, T)$ is a time stable state and $Y = (S, M, \emptyset, W, P, C, \emptyset)$ is a timed state. By Lemma C.12 and induction, Y satisfies Condition b. Follow from e(i)i and e(i)ii above, Y satisfies one of Conditions 1 and 2.

(iii) By Rule T-D shown in Figure 6.18 on Page 118, $X = (S, M, E, W, P, C, O)$ is timed state and the rule specifies τ-transitions triggered by $w \in C$, where w is either a multiple task instance or a task. By Rule T-D, $Y = (S, M, E, W, P \cup \{w\}, C \setminus \{w\}, O)$. By induction, Y satisfies Condition b, and follow from e(i)i and e(i)ii above, Y satisfies one of Conditions 1 and 2.

(iv) By Rule T-C shown in Definition 6.12 on Page 116, $X = (S, M, E, W, P, \emptyset, \emptyset)$ is a timed state and $Y = (S, M, E, W, T)$ is a untimed state such that $P = T$. By induction, Y satisfies Condition a and one of Conditions 1 and 2.

□

Appendix D

Process Semantics

This chapter defines function that are partially defined in Chapter 5 for modelling the untimed behaviour of BPMN.

D.1 Alphabet

This section defines functions that are partially defined in Section 5.2 on Page 58 for generating the alphabet of a CSP process of a BPMN element.

```
>istask, iserror, hasmessage, ismessage :: Type -> Bool
>istask (Task _ _) = True
>istask _ = False

>iserror (Eerror _) = True
>iserror (Ierror _) = True
>iserror _ = False

>ismessage (Smessage _) = True
>ismessage (Imessage _) = True
>ismessage (Emessage _) = True
>ismessage _ = False

>hasmessage (Smessage (Just m)) = True
>hasmessage (Imessage (Just m)) = True
>hasmessage (Emessage (Just m)) = True
>hasmessage _ = False

>taskname :: Type -> TaskName
>taskname (Miseq t y l f) = t
>taskname (Mipar t y l f) = t
>taskname (Task t y) = t

>errorcode :: Type -> ErrorCode
>errorcode (Eerror (Exception e)) = e
>errorcode (Ierror (Exception e)) = e
```

D.2 Atomic Elements

This section defines functions that are partially defined in Section 5.3 on Page 60 for modelling the behaviour of atomic elements.

```
>getvalue :: Maybe a -> [a]
>getvalue = (maybe [] single) where single m = [m]


>internal :: String -> Event
>internal s = "internal."++s

>isOne :: FlowType -> Bool
>isOne One = True
>isOne _ = False

>ndetseq :: Int -> FlowType -> [Seqflow] -> Process -> ([Local],Process)
>ndetseq lp ft os p
```

```
>  | lp == 1 = sact os p
>  | lp > 1 && isOne ft =
>      ([mp],(Hide (SeqComp (SeqComp p (seqext os))
>                 (Parinter (Interrupt (seqcomps ((replicate (lp-1) np)++[ap])) cp)
>                           ce (ProcId "Mon"))) ce))
>  | lp > 1 && (not.isOne) ft =
>      ([mp],(Hide (SeqComp p
>                 (SeqComp (Parinter (Interrupt (seqcomps ((replicate (lp-1) np)++[ap])) cp)
>                                         ce (ProcId "Mon")) (seqext os))) ce))
>  where np = Intern p ap
>        ap = Prefix (internal "a") Skip
>        cp = Prefix (internal "b") Skip
>        ce = List Set (map internal ["a","b"])
>        mp = LP ("Mon",[],(Prefix (internal "a") (Prefix (internal "b") Skip)))


>parinters :: [Process] -> Events -> Process
>parinters [p] es = p
>parinters (p:ps) es = Parinter p es (parinters ps es)

>interleaves :: [Process] -> Process
>interleaves [p] = p
>interleaves (p:ps) = Inter p (interleaves ps)

>fixpar :: Int -> FlowType -> [Seqflow] -> Process -> ([Local],Process)
>fixpar lp ft os p
>    | lp == 1 = sact os p
>    | lp > 1 && isOne ft =
>        ([],(Hide (Parinter (parinters (replicate lp np) le)
>                           se (SeqComp (seqext os) (Prefix (internal "a") Skip))) le))
>    | lp > 1 && (not.isOne) ft =
>        ([],(SeqComp (interleaves (replicate lp p)) (seqext os)))
>  where np = SeqComp p (Extern (SeqComp (seqext os) (Prefix (internal "a") Skip))
>                               (Prefix (internal "a") Skip))
>        le = List Set [(internal "a")]
>        se = List Set ((map seqflow os)++[(internal "a")])


>ndetpar :: Int -> FlowType -> [Seqflow] -> Process -> ([Local],Process)
>ndetpar lp ft os p
>    | lp == 1 = sact os p
>    | lp > 1 && isOne ft =
>        ([],(Hide (Parinter (interleaves (replicate lp np)) se sp)
>                  (List Set (map internal ["a","b"]))))
>    | lp > 1 && (not.isOne) ft =
>        ([],(SeqComp (Hide (Parinter (interleaves (replicate lp np1)) se1 sp1)
>                           (List Set (map internal ["a","b"])))
>                     (seqext os)))
>  where np = Extern (SeqComp p (Extern (SeqComp (seqext os) (Prefix (internal "a") Skip))
>                                       (Prefix (internal "a") Skip)))
>                    (Prefix (internal "b") Skip)
>        ep = Extern (Prefix (internal "a") Skip) (Prefix (internal "b") Skip)
>        se = List Set ((map seqflow os) ++ (map internal ["a","b"]))
>        sp = SeqComp (seqext os) (interleaves (replicate lp ep))
>        np1 = Extern (SeqComp p (Prefix (internal "a") Skip))
>                     (Prefix (internal "b") Skip)
>        sp1 = Prefix (internal "a") (interleaves (replicate (lp-1) ep))
>        se1 = List Set (map internal ["a","b"])


>encap :: Atom -> [Event] -> ([Local],Process) -> ([Local],Process)
>encap a es (l,p)
>    | length e == 0 = (l,p)
>    | length e > 0 && (isatom . etype) a =
>           (l,(Parinter (Interrupt p (exits e)) (List Set (outevents a))
>                        (Extern (exits e) (seqext o))))
>    | length e > 0 && (iscompound . etype) a =
>           (l++[LP (("C"++(eid a)),[],mon), LS ("A"++(eid a),(List Set es))],
>           (Parinter (Interrupt p (catches e))
```

```
>                    (SName ("A"++(eid a))) (ProcId ("C"++(eid a)))))
>  where e = exit a
>        o = outs a
>        mon = Extern (Indextern ("x",Diff (SName ("A"++(eid a))) (List Set (outevents a)))
>                           (Prefix "x" (ProcId ("C"++(eid a)))))
>                  (Extern (seqext o) (alternate e))

>actsem :: Atom -> [Event] -> Process -> ([Local],Process)
>actsem a es p
>  | issingle t = let (l,np) = (encap a es (sact (outs a) p) ) in inc (l,np)
>  | isseq t = let (l,np) = (encap a es (miseq t (outs a) p)) in inc (l,np)
>  | ispar t = let (l,np) = (encap a es (mipar t (outs a) p)) in inc (l,np)
>  where t = etype a
>        s = ins a
>        o = outs a
>        m = receive a
>        inc (lc,pr) = (lc,SeqComp (Inter (mgeext m) (seqext s)) pr)

>outevents :: Atom -> [Event]
>outevents (Atom n t i o e r im om) = (exitevents e) ++ (map seqflow o)

>exits :: [(Type,Seqflow)] -> Process
>exits es = externs (map exitp es)

>exitp :: (Type,Seqflow) -> Process
>exitp (t,s)
>  | ismessage t = mgesem t [] [s]
>  | iserror t = errsem t [] [s]
>  | otherwise = othersem t [] [s]

>issub, ismiseqs, ismipars :: Type -> Bool
>issub (SubProcess _ _) = True
>issub _ = False
>ismiseqs (Miseqs _ _ _ _) = True
>ismiseqs _ = False
>ismipars (Mipars _ _ _ _) = True
>ismipars _ = False

>isseq, ispar, isact :: Type -> Bool
>isseq t = ismiseq t || ismiseqs t
>ispar t = ismipar t || ismipars t
>isact t = isatom t || iscompound t

>iscompound, issingle :: Type -> Bool
>iscompound t = issub t || ismiseqs t || ismipars t
>issingle t = istask t || issub t

>miseq , mipar :: Type -> [Seqflow] -> Process -> ([Local],Process)
>miseq t o p =
>  case getloop t of
>    (Fix i) -> (fixseq i (getflow t) o p)
>    (Ndet i) -> (ndetseq i (getflow t) o p)

>mipar t o p =
>  case getloop t of
>    (Fix i) -> (fixpar i (getflow t) o p)
>    (Ndet i) -> (ndetpar i (getflow t) o p)
```

D.3 Subprocesses

This section defines functions that are partially defined in Section 5.4 on Page 68 for modelling the behaviour of compound elements.

```
>end :: ElementId -> Event
>end id = "c."++ id

>element :: Element -> [ProcessDef]
```

```
>element e =
>  let am = atom e
>      (l,p) = (element1 am (alpha' e))
>      term = (pterm . eid ) am
>  in case e of
>       (Atomic a) -> [(term,l,p)]
>       (Compound a es) -> [(term,l,p)] ++ (embed (eid a) es)

>complete :: [Element] -> Atom -> ProcessDef -> ProcessDef
>complete es a (n,l,p)
>  | (isstart . etype) a = (n,[], Extern (SeqComp p ep) ep)
>  | (isend . etype) a = (n,[],Extern (SeqComp p (Prefix ((end . eid) a) Skip)) ep)
>  | otherwise = (n,(ld:l), (ProcId "X"))
>  where ep = externs [ Prefix (end e) Skip | e <- ends es, not ((isend . etype) a && e == eid a) ]
>        mp = (if ((isexgate. etype) a && (length . outs) a > 1) then (msync es a) else p)
>        ld = LP ("X",[],Extern (SeqComp mp (ProcId "X")) ep)

>msync :: [Element] -> Atom -> Process
>msync es a = SeqComp ((seqext . ins) a) (externs mp)
>  where mp = [ msgalpha b | b <- (map atom es), (not. null) (intersect (ins b) (outs a))]

>aterm :: String -> SetName
>aterm a = "A("++a++")"

>pterm,spterm :: String -> ProcVar
>pterm i = "P("++i++")"
>spterm i = "SP("++i++")"

>name :: Element -> String
>name (Atomic a) = eid a
>name (Compound a es) = eid a

>par :: (a -> b) -> (c -> d) -> (a,c) -> (b,d)
>par f g (x,y) = (f x, g y)

>split :: a -> (a,a)
>split x = (x,x)

>isend, isstart :: Type -> Bool
>isend End = True
>isend (Emessage _) = True
>isend (Eerror _) = True
>isend _ = False

>isstart Start = True
>isstart (Smessage _) = True
>isstart (Stime _) = True
>isstart (Srule _) = True
>isstart _ = False

>isexgate :: Type -> Bool
>isexgate Exgate = True
>isexgate _ = False

>msgalpha :: Atom -> Process
>msgalpha a =
>  let is = ins a
>      im = receive a
>  in case etype a of
>      (Itime t) -> seqext is
>      (Imessage (Nothing)) -> seqext is
>      (Imessage (Just m)) -> Prefix (mgeflow m) (seqext is)
>      _ -> SeqComp (mgeext im) (seqext is)

>alternate :: [(Type,Seqflow)] -> Process
>alternate es = externs ((map exitp nes) ++
>                    (map (\ (t,s) -> Prefix ((except . errorcode) t) (seqext [s])) ees))
>  where ees = [ e | e <- es, (iserror.fst) e && (hasexcept.fst) e ]
>        nes = [ e | e <- es, (not.iserror.fst) e || ((iserror.fst) e && (not.hasexcept.fst) e) ]
```

```
>catches :: [(Type,Seqflow)] -> Process
>catches es = externs ((map exitp nes) ++ [(seqext . snd . unzip) ees])
>  where ees = [ e | e <- es, (iserror.fst) e && (hasexcept.fst) e ]
>        nes = [ e | e <- es, (not.iserror.fst) e || ((iserror.fst) e && (not.hasexcept.fst) e) ]
```

D.3.1 Pools and Diagrams

This section defines functions that are partially defined in Section 5.5 on Page 72 for modelling the behaviour of BPMN pools and diagrams.

```
>pool :: (PoolId,[Element]) -> Script
>pool (id,es) =
>  let term = plterm id
>      procs = (embed id es) ++ [(term,[],ProcId (spterm id))]
>      index = (List Set (map (eid.atom) es))
>      events = ((aterm id),(Comp Set [("i",index),("j",(SName (aterm "i")))] "" "j"))
>  in Script (datas es) [] procs ((alphabet es es)++[events]) []
```

Appendix E

Modelling Relative Time

This chapter defines function that are partially defined in Chapter 6 for modelling the relative timed behaviour of BPMN.

```
>pTot :: (PoolId,[Element]) -> Script
>pTot (id,es) = Script d c (p++[coord(id,es),(tpterm id,[],tproc)]) e (s++[(spec (tpterm id))])
>  where tproc = Parinter (ProcId (cterm id)) (SName "Events") (ProcId (plterm id))
>        (Script d c p e s) = pool(id,es)

>tpterm,cterm,cpterm :: PoolId -> ProcVar
>tpterm s = "TP("++s++")"
>cterm s = "CD("++s++")"
>cpterm s = "CP("++s++")"

>coord :: (PoolId,[Element]) -> ProcessDef
>coord (id,ss) = ((cterm id),[],stable state)
>  where state = (ss,[],(filter (isstart.etype.atom) ss),[],(filter (isstime.etype.atom) ss))
```

E.1 Preliminaries

E.1.1 Containment

Function `container` takes a list of elements `es` and an element `s`, and returns `a` where `a`, a member of `es`, is the subprocess that contains `s`. function `container1` takes a compound element `cp` and an element `s`, and returns either `cp` if `cp` directly contains `s` or an element in `cp`, which directly contains `s`; function `isIn` takes two elements `cp` and `u`, and checks if either `cp` has the same id as `u` or `cp` contains an element, which has the same id as `u`; function `dcontained` takes a compound element of the value `Compound a b` and returns `b`, that is, elements directly contained in `a`. and function `subset` takes two lists `a` and `b` and checks if `b` contains everything in `a`.

```
>container0 :: [Element] -> Element -> (Maybe Element)
>container0 [] _ = Nothing
>container0 es s =
>  let compounds = (filter (iscompound.etype.atom) es)
>  in if (or [ s == u | u <- (es \\ compounds) ]) || (null compounds) then Nothing
>     else maybe (container0 (tail compounds) s) Just (container1 (head compounds) s)

>container1 :: Element -> Element -> (Maybe Element)
>container1 cp u =
>  if (not.(isIn cp)) u then Nothing
>  else if or [ x == u | x <- (dcontained cp) ] then Just cp
>       else container0 (filter (iscompound.etype.atom) (dcontained cp)) u

>hascontainer :: [Element] -> Element -> Bool
>hascontainer es s = (container0 es s) /= Nothing

>isIn :: Element -> Element -> Bool
>isIn cp u = u == cp || ((iscompound.etype.atom) cp && or [ (isIn e u) | e <- (dcontained cp) ])

>dcontained :: Element -> [Element]
>dcontained (Compound e es) = es

>flatten :: [Element] -> [Element]
>flatten [] = []
```

```
>flatten (e:es) =
>  if (not.iscompound.etype.atom) e then [e]++(flatten es)
>  else [e]++((flatten . dcontained) e)++(flatten es)

>subset :: (Eq a) => [a] -> [a] -> Bool
>subset xs ys = all (`elem` ys) xs

>findoriginal :: [Element] -> Element -> Element
>findoriginal es = fromJust.(findoriginalfrom es)

>findoriginalfrom :: [Element] -> Element -> (Maybe Element)
>findoriginalfrom es e =
>  if (isitime.etype.atom) e && (null.ins.atom) e
>  then find (not.null.(intersect ((outs.atom) e)).snd.unzip.exit.atom) (flatten es)
>  else find ((subset ((ins.atom) e)).ins.atom) (flatten es)
```

E.2 Coordinating Untimed States

This section defines the functions for implementing Step 1 of the coordination procedure, described in Section 6.3.

E.2.1 Auxiliary Functions

```
>exts :: [Process] -> Process
>exts [p] = p
>exts (p:ps) = (Extern p (Styling "\n\t" (exts ps) ""))

>deciterations :: [(Element,Int)] -> Element -> [(Element,Int)]
>deciterations ms m = (removems ms m) ++ [(m,(curiteration ms m) - 1)]

>removems :: [(Element,Int)] -> Element -> [(Element,Int)]
>removems ms m = filter ((/= ((eid.atom) m)).eid.atom.fst) ms

>getStart :: Element -> Element
>getStart c = head [ e | e <- (dcontained c), (isstart.etype.atom) e ]

>rm :: Seqflow -> Element -> Element
>rm s (Atomic a) = (Atomic (rm0 a s))
>rm s (Compound a b) = (Compound (rm0 a s) b)

>rm0 :: Atom -> Seqflow -> Atom
>rm0 (Atom d t i o e r m n) s = (Atom d t (delete s i) (delete s o) re r m n)
>  where re = [ (t,f) | (t,f) <- e, f /= s ]

>istimed :: Type -> Bool
>istimed t = not (disjs [iscompound,isend,isnstart,isgate] t)

>lookupm :: Element -> [(Element,Int)] -> Maybe Int
>lookupm _ [] = Nothing
>lookupm e ((x,y):xys)
>    | ((eid.atom) e) == ((eid.atom) x) = Just y
>    | otherwise = lookup e xys

>curiteration :: [(Element,Int)] -> Element -> Int
>curiteration ms e = maybe ((loop.getloop.etype.atom) e) id (lookupm e ms)

>inms :: [(Element,Int)] -> Element -> Bool
>inms ms e = isJust (lookupm e ms)

>noinstance :: [(Element,Int)] -> Element -> Bool
>noinstance ms = ((== 0).(curiteration ms))

>hasmoreinstances :: [(Element,Int)] -> Element -> Bool
>hasmoreinstances ms = ((> 0).(curiteration ms))
```

```
>findsucceed :: [Element] -> Seqflow -> Element
>findsucceed es = fromJust.(findsucceed0 es)

>findsucceed0 :: [Element] -> Seqflow -> (Maybe Element)
>findsucceed0 es s = find ((elem s).ins.atom) (flatten es)
```

E.2.2 Main Functions

```
>type UntimedState = ([Element], [(Element,Int)], [Element], [Element], [Element])

>stable :: UntimedState -> Process
>stable (ss,[],[],[],[]) = Skip
>stable (ss,ms,[],ae,te) = timer ss ms ae te
>stable (ss,ms,ue,ae,te) = exts [ branch (ss,ms,ue,ae,te) e | e <- ue ]

>branch :: UntimedState -> Element -> Process
>branch (ss,ms,ue,ae,rn) e
>  | srules e = exts [branchs (ss,ms,ue,ae,rn) e s | s <- (outs.atom) e]
>  | urules e = exts [branchu (ss,ms,ue,ae,rn) e s | s <- (outs.atom) e]
>  | erules e = branche (ss,ms,ue,ae,rn) e
>  | mrules e = exts [branchm (ss,ms,ue,ae,rn) e s | s <- (outs.atom) e]
>  | lrules e = branchl (ss,ms,ue,ae,rn) e
>  where srules = conjs [(isagate.etype.atom),((>1).length.outs.atom)]
>        urules = conjs [disjs [(not.isagate.etype.atom),((==1).length.outs.atom)],
>                        ((>0).length.outs.atom),(not.iscompound.etype.atom)]
>        erules = isend.etype.atom
>        mrules = disjs [(issub.etype.atom), conjs [(ismiseqs.etype.atom),(noinstance ms)]]
>        lrules = conjs [(ismiseqs.etype.atom),(hasmoreinstances ms)]

>suffix :: Seqflow -> UntimedState -> UntimedState
>suffix s (ss,ms,ue,ae,rn)
> | utrans ss ae s = ustate e (ss,ms,ue,ae,rn)
> | jtrans ss ae s = jstate e s (ss,ms,ue,ae,rn)
> | mtrans ss s = mstate e (ss,ms,ue,ae,rn)
> | ntrans ss s = nstate e (ss,ms,ue,ae,rn)
> | ttrans ss s = tstate e (ss,ms,ue,ae,rn)
>  where e = findnext ss ae s
```

E.2.3 Coordinating Atomic Elements

```
>branchs :: UntimedState -> Element -> Seqflow -> Process
>branchs (ss,ms,ue,ae,rn) r s =
>  Prefix (seqflow s) ((stable.(suffix s)) (ss,ms,union (delete r ue) [(rm s r)],ae,rn))

>branchu :: UntimedState -> Element -> Seqflow -> Process
>branchu (ss,ms,ue,ae,rn) r s = Prefix (seqflow s) ((stable.(suffix s)) (ss,ms,(delete r ue),ae,rn))

>branche :: UntimedState -> Element -> Process
>branche (ss,[],ue,[],[]) e | ue == [e] && elem e ss = Prefix ((end.eid.atom) e) (stable (ss,[],[],[],[]))
>branche (ss,ms,ue,ae,rn) e = Prefix ((end.eid.atom) e) (stable (ss,ms,union (delete e ue) [c],ae,rn))
>  where c = head [ s | s <- (flatten ss), (iscompound.etype.atom) s, elem e (dcontained s) ]
```

E.2.4 Coordinating Compound Elements

`branchm` models rules whose name are has prefix M, and `branchl` models rules whose name are has prefix L.

```
>branchm :: UntimedState -> Element -> Seqflow -> Process
>branchm (ss,ms,ue,ae,rn) m s = Prefix (seqflow s) ((stable.(suffix s)) (ss,ns,(delete m ue),ae,rn))
> where ns = [ (x,y) | (x,y) <- ms, x /= m ]

>branchl :: UntimedState -> Element -> Process
>branchl (ss,ms,ue,ae,rn) m
> | (not.istimed.etype.atom) e = stable (ss,(deciterations ms m),union (delete m ue) [e],ae,rn)
```

```
> | (istimed.etype.atom) e = stable (ss,(deciterations ms m),delete m ue,ae,(e:rn))
>  where e = getStart m
```

`utrans ss ae s` returns true if the element with incoming sequence flow `s` is either an untimed, atomic element such that if it is an AND gateway that it has only one incoming sequence flow `s`, or a subprocess element.

```
>findnext :: [Element] -> [Element] -> Seqflow -> Element
>findnext ss ae s = let e = findsucceed ss s in if elem e ae then findsucceed ae s else e

>utrans :: [Element] -> [Element] -> Seqflow -> Bool
>utrans ss ae s = disjs [ atome, compe ] (findnext ss ae s)
>  where atome = conjs [(not.istimed.etype.atom), (not.iscompound.etype.atom), dtome]
>        dtome = disjs [(not.isagate.etype.atom),((==1).length.ins.atom)]
>        compe = conjs [(not.istimed.etype.atom), (issub.etype.atom) ]

>jtrans :: [Element] -> [Element] -> Seqflow -> Bool
>jtrans ss ae s = conjs [(isagate.etype.atom),((>1).length.ins.atom)] (findnext ss ae s)

>mtrans :: [Element] -> Seqflow -> Bool
>mtrans ss s = conjs [ ismiseqs.etype.atom, not.istimed.etype.atom.getStart ] c
>  where c = findsucceed ss s

>ntrans :: [Element] -> Seqflow -> Bool
>ntrans ss s = conjs [ ismiseqs.etype.atom, istimed.etype.atom.getStart ] c
>  where c = findsucceed ss s

>ttrans :: [Element] -> Seqflow -> Bool
>ttrans ss = istimed.etype.atom.(findsucceed ss)
```

Functions matching suffixes of the transition rules

```
>ustate :: Element -> UntimedState -> UntimedState
>ustate c (ss,ms,ue,ae,rn) = (ss,ms,(e:ue),delete e ae,rn)
>  where e = if (iscompound.etype.atom) c then getStart c else c

>jstate :: Element -> Seqflow -> UntimedState -> UntimedState
>jstate e s (ss,ms,ue,ae,rn) = (ss,ms,ue,union (delete e ae) [(rm s e)],rn)

>mstate :: Element -> UntimedState -> UntimedState
>mstate c (ss,ms,ue,ae,rn) = (ss,union ms [(c,l-1)],(e:ue),ae,rn)
>  where e = getStart c
>        l = curiteration ms c

>nstate :: Element -> UntimedState -> UntimedState
>nstate c (ss,ms,ue,ae,rn) = (ss,union ms [(c,l-1)],ue,ae,(e:rn))
>  where e = getStart c
>        l = curiteration ms c

>tstate :: Element -> UntimedState -> UntimedState
>tstate e (ss,ms,ue,ae,rn) = (ss,ms,ue,delete e ae,(e:rn))
```

E.3 Calculating Time Progression

This section defines the functions for implementing Step 2 of the coordination procedure, described in Section 6.4.

E.3.1 Timed Exception Association

The functions `minrange` and `maxrange` take a timed BPMN element and either return minimum and maximum durations respectively if the element is a task element, or both return its duration if it is a timer event element.

```
>minrange,maxrange :: Element -> Time
```

```
>minrange e =
>  case (etype.atom) e of
>    (Stime t) -> t
>    (Itime t) -> t
>    _ -> (fst.range.atom) e

>maxrange e =
>  case (etype.atom) e of
>    (Stime t) -> t
>    (Itime t) -> t
>    _ -> (snd.range.atom) e

>cmpmin :: Element -> Element -> Ordering
>cmpmin x y | (minrange x) < (minrange y) = LT
>          | (minrange x) > (minrange y) = GT
>          | otherwise = EQ
```

The function `getexceptions ss` is defined such that `getexceptions ss e` returns `e` and its list of exception associations with respect to the list of BPMN elements `ss`; the function `subexceptions ss e` returns a list of intermediate timer event elements such that each element's type t and outgoing sequence flow s forms a pair in $assoc(e)$, satisfying the second condition of Definition 6.11, and the function `fdexcept` calculates three lists of elements: the first two lists are the sublists of delayed and postponed elements that are currently active, and the third list contains fresh timed elements and their exception associations.

```
>separate :: Element -> (Either (Element) (Element,Element))
>separate s =
>  if (not.isact.etype.atom) s then Left s
>  else case find (isitime.fst) ((exit.atom) s) of
>    Just (t,f) -> Right (s,(Atomic (Atom (((++"except").eid.atom) s) t [] [f] [] (zero,zero) [] [])))
>    Nothing -> Left s

>unsplit (a,b) = [a,b]

>getexceptions :: [Element] -> Element -> [Element]
>getexceptions ss = (uncurry union).(par sep (subexceptions ss)).split
>  where sep = (either (\a -> [a]) unsplit).separate

>subexceptions :: [Element] -> Element -> [Element]
>subexceptions ss e = map (makeexception.snd) ps
>  where ps = filter ((== e).fst) (concatMap incompound (filter (iscompound.etype.atom) ss))

>incompound :: Element -> [(Element,Element)]
>incompound e =
>  if hastimeexcept e then [ (t,e) | t <- (contained e) ]
>  else concatMap incompound (filter (iscompound.etype.atom) (dcontained e))

>contained :: Element -> [Element]
>contained = flatten.dcontained

>hastimeexcept :: Element -> Bool
>hastimeexcept e = or [ isitime t | (t,y) <- (exit.atom) e ]

>makeexception :: Element -> Element
>makeexception s =
>  case find (isitime.fst) ((exit.atom) s) of
>    Just (t,f) -> Atomic (Atom "" t [] [f] [] (zero,zero) [] [])
>    Nothing -> error "no exception flow"

>fdexcept :: [Element] -> [Element] -> ([Element],[Element],[Element])
>fdexcept ss rn =
>  let (ps,ds) = (filter (postponed ss) rn, filter (delayed ss) rn)
>  in (ds,ps,filter (`notElem` ps) (concatMap (getexceptions ss) (rn \\ (union ps ds))))
```

E.3.2 Multiple Instance

This section defines the functions for gathering the instance of each timed active multiple instance element. This procedure is described in Section 6.4.3.

```
>instancenm :: String -> String
>instancenm = (++"instance")

>hasinstance :: Element -> Element -> Bool
>hasinstance m t =
>  (istask.etype.atom) t && (ismiseq.etype.atom) m && (eid.atom) t == (instancenm.eid.atom) m

>splitseq :: Element -> (Element,Element)
>splitseq (Atomic (Atom d (Miseq n t l f) i o e r m s)) =
>  let lp = if (isfix l) then Fix ((loop l) - 1) else Ndet ((loop l) - 1)
>  in ((Atomic (Atom d (Miseq n t lp f) i o e r m s)),
>      (Atomic (Atom (instancenm d) (Task n t) [] [] [] r [] [])))

>parpost :: [Element] -> [Element] -> ([Element],[Element])
>parpost ss = partition (conjs [ismiseq.etype.atom,\m -> not (any (hasinstance m) ss)])
```

E.3.3 Ordering the Timed Sequence

This section defines the main function `timer` that implements Step 2 of the coordination procedure, and also the function `sorting` for ordering the sequence of time active elements for enactment, described in Section 6.4.4.

```
>timer :: [Element] -> [(Element,Int)] -> [Element] -> [Element] -> Process
>timer ss ms ae rn =
>  let (de,ps,fr) = fdexcept ss rn
>      ((nm,ni),op) = par (unzip.(map splitseq)) id (parpost (union ps de) ps)
>      (fm,fi) = (unzip.(map splitseq).(filter (ismiseq.etype.atom))) fr
>      (pm,as) = sorting op de (unions [(fr \\ fm),fi,ni])
>      (en,ac) = break ((> 0).minrange) as
>  in exe (ss,ms,[],ae,(unions [ac,pm,nm,fm]),en,[])

>hasmiseq :: [Element] -> Element -> (Element,Element)
>hasmiseq xs m =
>  case find (hasinstance m) xs of
>    Just t -> (m,t)
>    Nothing -> error "cannot find instances"

>sorting :: [Element] -> [Element] -> [Element] -> ([Element],[Element])
>sorting op de fr =
>  let (ms,os) = partition (ismiseq.etype.atom) op
>      mt = minimum ((map minrange (unions [os,fr]))++(map maxrange de))
>  in (ms,sortBy cmpmin (map (subtime mt) (unions [os,de,fr])))
```

E.4 Coordinating Timed States

This section defines the functions for implementing Step 3 of the coordination procedure, described in Section 6.5.

```
>candelay = (\e -> minrange e /= maxrange e)
>work = (task.taskname.etype.atom)

>intersects :: (Eq a) => [a] -> [a] -> Bool
>intersects x y = (not.null) (intersect x y)

>attaches :: Element -> Element -> Bool
>attaches e c = intersects ((snd.unzip.exit.atom) c) ((outs.atom) e)

>findattach :: [Element] -> Element -> [Element]
```

```
>findattach ss e = filter (‘attaches‘ e) ss

>isexcept :: Element -> Bool
>isexcept = conjs [isitime.etype.atom,null.ins.atom]

>msend :: Element -> Bool
>msend = (== 0).loop.getloop.etype.atom

>isinstance :: [Element] -> Element -> Bool
>isinstance ps w = find (‘hasinstance‘ w) ps /= Nothing

>tot :: [Element] -> [Element] -> UntimedState -> TimedState
>tot cs os (ss,ms,es,ae,ps) = (ss,ms,es,ae,ps,cs,os)

>cancels :: [Element] -> [Element] -> Element -> [Element]
>cancels ss ru e = unions [cs,ds,is]
>  where (Just a) = find (attaches e) ss
>        cs = if (not.iscompound.etype.atom) a then [a]
>             else filter (‘elem‘ (contained a)) ru
>        ds = filter (\x -> any (attaches x) cs) ru
>        is = filter (\t -> any (‘hasinstance‘ t) cs) ru

>type TimedState = ([Element],[(Element,Int)],[Element],[Element],[Element],[Element],[Element])

>exe :: TimedState -> Process
>exe (ss,ms,es,ae,ps,[],[]) = stable (ss,ms,es,ae,ps)
>exe (ss,ms,es,ae,ps,cs,os) =
>  exts ([ csbranch (ss,ms,es,ae,ps,cs,os) e | e <- cs] ++
>        [ osbranch (ss,ms,es,ae,ps,cs,os) e | e <- os])

>csbranch :: TimedState -> Element -> Process
>csbranch state e
> | vrule e = exts [ branchv state e s | s <- (outs.atom) e ]
> | isexcept e = exts [ branchx state e s | s <- (outs.atom) e ]
> | drule e = Extern (csbranch1 state e) (Prefix "tinternal" ((exe.(dstate e)) state))
> | otherwise = csbranch1 state e
>   where vrule = conjs [(not.isexcept),(istimer.etype.atom)]
>         drule = conjs [(istask.etype.atom),candelay]

>dstate :: Element -> TimedState -> TimedState
>dstate e (ss,ms,es,ae,ps,cs,os) = (ss,ms,es,ae,(e:ps),(delete e cs),os)

>branchv :: TimedState -> Element -> Seqflow -> Process
>branchv (ss,ms,es,ae,ps,cs,os) r s =
>  Prefix (seqflow s) ((exe.(tot (delete r cs) os).(suffix s)) (ss,ms,es,ae,ps))

>csbranch1 :: TimedState -> Element -> Process
>csbranch1 (ss,ms,es,ae,ps,cs,os) e
> | erule e = Prefix (work e) (exe (ss,ms,es,ae,ps,(delete e cs),(e:os)))
> | mrule e = Prefix (work e) (exe (ss,ms,es,ae,(delete m ps),(delete e cs),(m:os)))
> | nrule e = Prefix (work e) (exe (ss,ms,es,ae,ps,(delete e cs),os))
>   where erule = conjs [(not.(isinstance ps)),(istask.etype.atom)]
>         mrule = conjs [(isinstance ps),(istask.etype.atom),msend]
>         nrule = conjs [(isinstance ps),(istask.etype.atom),(not.msend)]
>         m = (fromJust.(find (‘hasinstance‘ e))) ps

>osbranch :: TimedState -> Element -> Process
>osbranch state e | trule e = exts [ brancht state e s | s <- (outs.atom) e ]
> where trule = disjs [(istask.etype.atom),(ismiseq.etype.atom)]

>brancht :: TimedState -> Element -> Seqflow -> Process
>brancht (ss,ms,es,ae,ps,cs,os) r s =
>  Prefix (seqflow s) ((exe.(tot (cs \\ ex) (delete r os)).(suffix s)) (ss,ms,es,ae,ps \\ ex))
>  where ex = findattach (unions [ps,cs]) r

>branchx :: TimedState -> Element -> Seqflow -> Process
>branchx (ss,ms,es,ae,ps,cs,os) r s =
>  Prefix (seqflow s) (texe (ss,ms \\ ns,es \\ rs,ae \\ rs,ps \\ rs))
>  where e = findsucceed ss s
```

```
>       rs = cancels ss (unions [ps,cs]) e
>       ns = foldr (\c n -> removems n c) ms (filter (iscompound.etype.atom) (e:cs))
>       texe = exe.(tot (cs \\ (r:rs)) (os \\ rs)).(suffix s)
```

Appendix F

From BPMN To `Empiricol`

The function `findtermin` finds the prerequisite activities to terminate this empirical workflow; the function `checktermin` checks for terminating dependent activities; the function `trans` that takes two copies of the same BPMN diagram (single BPMN pool) and returns an `Empiricol` workflow described by the diagram, and the function `ckele` takes list of elements and returns a list of sequence rules modelled by subprocess elements in that list.

```
>bToe :: Diagram -> Empiricol
>bToe ds = ([(startseq es)]++(map (checktermin pa) (trans es es))++end)
>  where es = (snd.head) ds
>        pa = findtermin es (filter (isend.etype.atom) es)
>        end = maybe [] (\p -> [NStop p]) pa

>trans :: [Element] -> [Element] -> Empiricol
>trans es = foldr (\e w -> (ckele es e)++w) []

>findtermin :: [Element] -> [Element] -> Maybe PreAct
>findtermin sp es = listToMaybe ep
>  where ep = map (pre sp) (concatMap (ins.atom) es)

>checktermin :: Maybe PreAct -> EventSequencing -> EventSequencing
>checktermin pa es = (maybe es (\p -> cktermin p NSTOP es)) pa

>cktermin :: PreAct -> ActivityId -> EventSequencing -> EventSequencing
>cktermin pa id (Event n p de c1 c2 dc r w)
>  | isInPre n pa = (Event n p de c1 c2 (insertD (da id) dc) r w)
>  | otherwise = (Event n p de c1 c2 dc r w)

>isInPre :: ActivityId -> PreAct -> Bool
>isInPre nm (OneOf ps) = (any (isInPre nm) ps)
>isInPre nm (All ps) = (any (isInPre nm) ps)
>isInPre nm (Leaf p) = p == nm

>insertD :: DptAct -> DptAct -> DptAct
>insertD a NoFlow = (Seq [a])
>insertD a (Seq acts) = (Seq (acts++[a]))

>isterminal :: Element -> Bool
>isterminal = isend.etype.atom

>startseq :: [Element] -> EventSequencing
>startseq es = (Empiricol.Start (dpta es f))
>  where f = (head.outs.atom.fromJust.(find (isstart.etype.atom))) es

>ckele :: [Element] -> Element -> [EventSequencing]
>ckele es e =
>  if (not.issub.etype) a then []
>  else [Event (ename e) (pre es ((head.ins) a)) d (scond e) (econd e) ((norm.dpt) e) (reps e) ((norm.inv) e)]
>  where a = atom e
>        d | (null.exit.atom) e = dpta es ((head.outs) a)
>          | otherwise = OneOf ([dpta es ((head.outs) a)]++(map (dpta es) ((snd.unzip.exit) a)))

>isseqsub :: Element -> Bool
>isseqsub = (conjs [issub,(== SeqB).subtype]).etype.atom
```

The function `idsBname` takes a subprocess name and returns the id of the activity which the subprocess describes; the function `idsTname` takes a task name and returns the identifier of the activity which the task element describes; the function `tkTowk` takes a task name and returns an description of work

procedure which the task element describes; the function `filtersub` takes a `SubProcessType` value and a list of compound elements and returns the list of the subsets which have been labelled with that that `SubprocessType` value, and the function `ename` takes a subprocess element, and returns the identifier of the activity, which that element describes.

```
>idsBname :: BName -> ActivityId
>idsBname = Id.(map toUpper)

>idsTname :: TaskName -> (ActivityId,ActType)
>idsTname t = ((Id.(map toUpper)) k,ap at)
>  where (k,at) = ((par id tail).(splitAt i)) t
>        i = ((maybe (length k) id).(findIndex (== '_'))) t

>ap :: String -> ActType
>ap "Manual" = Manual
>ap "Auto" = Automatic
>ap _ = Manual

>tkTowk :: TaskName -> WorkId
>tkTowk n = n

>subTowk :: BName -> WorkId
>subTowk n = n

>filtersub :: SubProcessType -> [Element] -> [Element]
>filtersub s = filter ((conjs [iscompound,(== s).subtype]).etype.atom)

>findsub = \ s -> listToMaybe . (filtersub s)

>ename :: Element -> ActivityId
>ename = idsBname.subname.etype.atom

>subname :: Type -> BName
>subname (Miseqs b y l f) = b
>subname (Mipars b y l f) = b
>subname (SubProcess b y) = b
```

F.1 Prerequisites and Dependences

The function `dpta` takes a list of elements and an outgoing sequence flow of a subprocess and returns the dependence tree of the sequence rule modelled by that subprocess, and the function `pre` takes a list of elements and an incoming sequence flow of a subprocess and returns the prerequisite tree of the sequence rule modelled by that subprocess.

```
>dpta :: [Element] -> Seqflow -> DptEvent
>dpta = tree Dpts

>pre :: [Element] -> Seqflow -> PreAct
>pre = tree Pres

>tree :: TreeType -> [Element] -> Seqflow -> Tree
>tree t es s =
>  ((\x -> g x es s).fromJust.(find f)) es
>  where (f,g) | t == Pres = ((elem s).allouts,preap)
>              | t == Dpts = ((elem s).ins.atom,dptap)

>preap :: Element -> [Element] -> Seqflow -> Tree
>preap e es s
>  | (isstart.etype) a = Leaf START
>  | isxj e = OneOf (map (tree Pres (es++[e])) (ins a))
>  | isaj e = Empiricol.All (map (tree Pres (es++[e])) (ins a))
>  | disjs [isxs,isas] e = tree Pres (es++[e]) ((head.ins) a)
>  | (iscompound.etype) a = Leaf (ename e)
>  where a = atom e
```

```
>dptap :: Element -> [Element] -> Seqflow -> Tree
>dptap e es s
>  | (isend.etype) a = Leaf NSTOP
>  | isxs e = OneOf (map (tree Dpts (es++[e])) (outs a))
>  | isas e = Empiricol.All (map (tree Dpts (es++[e])) (outs a))
>  | disjs [isaj,isxj] e = tree Dpts (es++[e]) ((head.outs) a)
>  | (iscompound.etype) a = Leaf (ename e)
>  where a = atom e
```

Functions `isxs`, `isas`, `isxj` and `isaj` check if a given element is a XOR Split, AND Split, XOR Join and AND Join element. The function `cond` takes an element projects the condition of an intermediate rule element attached to that element, and the functions `scond` and `econd` take an element and project the start and terminate conditions of the sequence rule modelled by that element.

```
>isxs, isas, isxj, isaj :: Element -> Bool
>isxs = (conjs [isxgate.etype,(== 1).length.ins]).atom
>isas = (conjs [isagate.etype,(== 1).length.ins]).atom
>isxj = (conjs [isxgate.etype,(== 1).length.outs]).atom
>isaj = (conjs [isagate.etype,(== 1).length.outs]).atom

>cond :: Element -> Condition
>cond e | isatom t = ((maybe NoCond rule).(find isirule)) x
>       | otherwise = rule t
>       where t = (etype.atom) e
>             x = (fst.unzip.exit.atom) e

>rule :: Type -> Condition
>rule (Srule c) = c
>rule (Irule c) = c
>rule _ = NoCond

>scond,econd :: Element -> Condition
>scond = (maybe NoCond (rule.etype.atom)).(find (issrule.etype.atom)).embedding
>econd = (maybe NoCond rule).(find isirule).fst.unzip.exit.atom

>issrule,isirule :: Type -> Bool
>issrule (Srule _) = True
>issrule _ = False

>isirule (Irule _) = True
>isirule _ = False
```

F.2 Observation and Work Groups

The function `dpt` takes a subprocess element representing a sequence rule and returns the rule's observation group; the function `inv` takes a subprocess element representing a sequence rule and returns the rule's work group (procedure workflow); the function `wbs` takes a list of elements embedded by a subprocess modelling a RWP and returns the workflow of procedures of the RWP; the function `dptact` takes a task element modelling an observation and returns that observation; the function `wb` takes a task element modelling a SNP and returns that SNP, and the function `wk` takes either a multiple instance task element and returns a SRP or a subprocess element and returns a RWP.

```
>dpt :: Element -> DptAct
>dpt = swf Dpt DptB

>inv :: Element -> Works
>inv = swf Wks InvB

>wbs :: [Element] -> WorkBlock
>wbs es =
>  ((maybe NoFlow f).(find ((disjs [isstart,isstime]).etype)).(map atom)) es
>  where f = (gswf Wbs es).head.outs

>swf :: WfType -> SubProcessType -> Element -> Swf
```

```
>swf w p e = ((maybe NoFlow ds).(findsub p).embedding) e
>  where ds = \x -> swfs w x

>swfs :: WfType -> Element -> Swf
>swfs w e = maybe NoFlow ((gswf w es).head.outs.atom) (find (isstart.etype.atom) es)
>  where es = embedding e

>gswf :: WfType -> [Element] -> Seqflow -> Swf
>gswf w es s = maybe NoFlow (\_ -> Seq (fst (seqswf w s es))) (findsucceed0 es s)

>swfmult :: WfType -> [Element] -> [Seqflow] -> ([Swf],Maybe Seqflow)
>swfmult w es ss =
>  maybe (a,(head p)) (\_ -> (a,Nothing)) (find (((/=).head) p) (tail p))
>  where (a,p) = (unzip.(map (pathswf w es))) ss

>pathswf :: WfType -> [Element] -> Seqflow -> (Swf, Maybe Seqflow)
>pathswf w es s = maybe (NoFlow,Nothing) (pathswf1 w es s) (findsucceed0 es s)

>pathswf1 :: WfType -> [Element] -> Seqflow -> Element -> (Swf, Maybe Seqflow)
>pathswf1 w es s e
>  | isxs e = (Choice f,n)
>  | isas e = (Par f,n)
>  | (inb w) e = (Seq d,t)
>  | disjs [isxj,isaj] e = (NoFlow,(listToMaybe.outs.atom) e)
>  | otherwise = (NoFlow,Nothing)
>  where (d,t) = seqswf w s es
>        (f,n) = swfmult w es ((outs.atom) e)

>seqswf :: WfType -> Seqflow -> [Element] -> ([Swf],Maybe Seqflow)
>seqswf w s es =
>  let (g,d) = unzip (seqswfs w s es)
>  in case reverse g of
>       (NoFlow:h) -> (init g,last d)
>       _ -> (g,last d)

>seqswfs :: WfType -> Seqflow -> [Element] -> [(Swf,Maybe Seqflow)]
>seqswfs w s es = maybe [] (seqswfs1 w s es) (findsucceed0 es s)

>seqswfs1 :: WfType -> Seqflow -> [Element] -> Element -> [(Swf,Maybe Seqflow)]
>seqswfs1 w s es e
>  | isxs e = [(Choice f,n)]++cs
>  | isas e = [(Par f,n)]++cs
>  | disjs [isxj,isaj] e = [(NoFlow,(listToMaybe.outs.atom) e)]
>  | (inb w) e =
>      [(single w e,(listToMaybe.outs.atom) e)]++
>      (seqswfs w ((head.outs.atom) e) es)
>  | otherwise = []
>  where (f,n) = swfmult w es ((outs.atom) e)
>        cs = maybe [] (\s -> seqswfs w s es) n

>inb :: WfType -> (Element -> Bool)
>inb Dpt = isatom.etype.atom
>inb Wks = (disjs [ismiseq,ismiseqs]).etype.atom
>inb Wbs = istask.etype.atom

>single :: WfType -> Element -> Swf
>single Dpt e = Single (Dp (dptact e))
>single Wks e = Single (Wk (wk e))
>single Wbs e = Single (Wu (wb e))

>procedure :: TaskType -> Procedure
>procedure (InvT (Inv n q m)) = ProcedureA (Treatment n q m)
>procedure _ = DetailHidden

>dptact :: Element -> Observation
>dptact e =
>  if (istask.etype.atom) e then (d,m,n,1,cond e,a)
>  else (d,m,n,l,cond e,a)
>  where (m,n) = par tTod tTod ((range.atom) e)
```

```
>       l = (loop.getloop.etype.atom) e
>       (d,a) = dname e

>wb :: Element -> WorkSUnit
>wb e =
>  let t = (etype.atom) e
>      (m,n) = (range.atom) e
>  in ((tkTowk.taskname) t,(procedure.tasktype) t,tTod m,tTod n)

>wk :: Element -> Work
>wk e =
>  let t = (etype.atom) e
>      r = (loop.getloop) t
>      (m,n) = ((par tTod tTod).range.atom) e
>      i = maybe zero maxrange (find (isstime.etype.atom) (embedding e))
>  in if ismiseq t
>     then WkS (((tkTowk.taskname) t),((procedure.tasktype) t),m,n,r)
>     else WkM (((subTowk.subname) t),(wbs (embedding e)),(tTod i),r)

>dname :: Element -> (ActivityId,ActType)
>dname = idsTname.taskname.etype.atom
```

F.3 Repetition

The function `reps` takes a subprocess element modelling a sequence rule and returns a possibly empty list of repeat clauses.

```
>mint,maxt :: [Element] -> Duration
>maxt = (maybe UNBOUNDED (tTod.ttime)).(find ((any (isitime.fst)).exit.atom))
>mint = (maybe UNBOUNDED (tTod.ttime)).(find (isstime.etype.atom))

>minl,maxl :: Loops -> Repeat
>minl (Fix i) = i
>minl (Ndet i) = 0
>maxl (Fix i) = i
>maxl (Ndet i) = i

>reps :: Element -> [RepeatExp]
>reps = (map rep).(filtersub RepB).embedding

>rep :: Element -> RepeatExp
>rep e =
>  let (c,d) = (range.atom) e
>      l = (getloop.etype.atom) e
>  in (tTod c, tTod d, minl l, maxl l, cond e)
```

Appendix G

From `Empiricol` To BPMN

The function `eTob` takes an empirical study and returns a BPMN pool that models it, and the function `mksub` maps a sequence rule to a subprocess element that models it;

```
>eTob :: Empiricol -> BPMN
>eTob w = ("Diagram",dependency w f g)
>  where (f,g) = ((par id head).unzip.(foldl cst [])) w
>        cst xs (Empiricol.Start _) = xs
>        cst xs (NStop _) = xs
>        cst [] e = [mksub e "Flow1"]
>        cst (x:xs) e = ((mksub e (snd x)):x:xs)

>mksub :: EventSequencing -> Seqflow -> (Element,Seqflow)
>mksub (Event id pre dpe cc ec dpt reps wk) s =
>  (Compound (Atom ("SubProcess"++(si s))
>                (SubProcess (idToBn id) SeqB) [] [] et norange [] []) em,ns)
>  where (em,ns) = bsub cc dpt reps wk n id
>        (et,n) = case ec of
>                  NoCond -> ([],s)
>                  c -> ([(Irule c,s)],sflow s)

>bsub :: Condition -> DptAct -> [RepeatExp] -> Works -> Seqflow -> ActivityId -> ([Element],Seqflow)
>bsub cc dpt reps wk s id =
>  case wk of
>    NoFlow -> ([start,end,fst (swfsub Dpt s (Left (dpt,id)))]++sreps,es)
>    _ -> ([start,end,aps,aj,sd,sw]++sreps,es)
>  where (start,sl) = case cc of
>                       NoCond -> ele ("Start"++(si s)) BPMN.Start s []
>                       c -> ele ("Start"++(si s)) (Srule c) s []
>        (sd,ds) = swfsub Dpt (sflow s) (Left (dpt,id))
>        (sreps,rs) = extreps ds sd id reps
>        (sw,ws) = swfsub Wks (sflow rs) (Left (wk,id))
>        (end,es) = case wk of
>                     NoFlow -> ele ("End"++(si rs)) End rs []
>                     _ -> ele ("End"++((si.bk) js)) End (bk js) []
>        (aps,s2) = node (Left [s]) (Left (map (head.ins.atom) [sd,sw])) "" ("Gate"++(si s)) Agate
>        (aj,js) = node (Left (map bk [rs,ws])) (Right 1) "" ("Gate"++(si ws)) Agate
```

G.1 Acyclic Structure Workflow

The function `swfsub` takes the ASW of an observation workflow or a procedure workflow or a RWP and returns a BPMN subprocess element embedding that ASW; the function `extswf` takes an ASW, and returns a pair, where the first component is a list of BPMN elements modelling the ASW and the second component is a sequence flow fresh with respect to that list of BPMN elements; the function `extswfm` takes the type `Agate` or `Xgate`, a sequence flow and a list of ASWs and returns a pair of a list of BPMN elements and a fresh sequence flow, where the list of BPMN elements models the choice or the interleaving of the list of ASWs depending on the `Type` value; the function `mkdpt` takes a sequence flow and an observation, and returns a pair where the first component is either an atomic task element or a multiple instance task element, and the second component is a fresh sequence flow; the function `mkwk` takes either a SRP or a SNP, and a sequence flow (`Seqflow`) and returns a pair where the first component is either an atomic task element or a multiple instance task element, and the second component is a fresh sequence flow, and the function `mkwb` takes a RWP and a sequence flow and returns a subprocess element modelling the RWP and a fresh sequence flow.

```
>type Struct = Either (Swf,ActivityId) (String,Swf,Duration,Repeat)

>swfsub :: WfType -> Seqflow -> Struct -> (Element,Seqflow)
>swfsub w s d = (Compound a (start:end:sf), sflow e)
>  where (tp,sp,wf) = either f1 f2 d
>        a = Atom ("SubProcess"++(si e)) tp [s] [e] [] norange [] []
>        (start,ss) = ele ("Start"++((si.sflow) s)) sp (sflow s) []
>        (sf,o) = extswf (sflow s) wf
>        (end,e) = ele ("End"++(si o)) End (bk o) []
>        f1 (u,v) = (swftype w (idToBn v) Nothing,BPMN.Start,u)
>        f2 (m,n,o,p) = (swftype w m (Just p),Stime (dTot o),n)

>swftype :: WfType -> String -> (Maybe Repeat) -> Type
>swftype Dpt n _ = SubProcess (n++"_Obv") DptB
>swftype Wks n _ = SubProcess (n++"_Work") InvB
>swftype Wbs n (Just r) = Miseqs (wkToTk n) Embedded (Fix (rep r)) BPMN.All

>extswf :: Seqflow -> Swf -> ([Element],Seqflow)
>extswf s NoFlow = ([],s)
>extswf s (Choice ws) = extswfm Xgate s ws
>extswf s (Par ws) = extswfm Agate s ws
>extswf s (Seq ws) = ((par concat last).unzip.(swfm id s)) ws
>extswf s (Single g) = sg s g

>sg :: Seqflow -> Acts -> ([Element],Seqflow)
>sg s (Wk (WkM m)) = (par (:[]) id) (mkwb m s)
>sg s (Wk (WkS w)) = (par (:[]) id) (mkwk (Left w) s)
>sg s (Wu w) = (par (:[]) id) (mkwk (Right w) s)
>sg s (Dp (a,b,c,d,e,f)) =
>  if elem a specialIds then ([],s)
>  else (par (:[]) id) (mkdpt (a,b,c,d,e,f) s)

>extswfm :: Type -> Seqflow -> [Swf] -> ([Element],Seqflow)
>extswfm t s wf = ((g:j:(concat fe)),(head.outs.atom) j)
>  where (g,_) = node (Left [s]) (Left (concatMap (ins.atom.head) fe)) "" ("Gate"++(si s)) t
>        (fe,fs) = unzip (swfm sflow (sflow s) wf)
>        (j,_) = node (Left fs) (Right 1) "" ("Gate"++((si.head) fs)) t

>swfm :: (Seqflow -> Seqflow) -> Seqflow -> [Swf] -> [([Element],Seqflow)]
>swfm f _ [] = []
>swfm f s (d:ds) = (g,t):(swfm f (f t) ds) where (g,t) = extswf s d

>mkdpt :: Observation -> Seqflow -> (Element,Seqflow)
>mkdpt (id,min,max,rp,cond,atype) s =
>  if rp == 1 then f ("Task"++(si s)) (Task (idToTn id atype) DptT)
>  else f ("Loop"++(si s)) (Miseq (idToTn id atype) DptT (Fix rp) BPMN.All)
>  where t = (dTot min,dTot max)
>        (n,p) = (sflow s,sflow n)
>        f ni y = (Atomic (Atom ni y [s] [n] ef t [] []), q)
>        (ef,q) = case cond of
>                  NoCond -> ([],p)
>                  _ -> ([(Irule cond,p)],sflow p)

>mkwk :: Either WorkMUnit WorkSUnit -> Seqflow -> (Element,Seqflow)
>mkwk u s = if loop i == 0 then f ("Task"++(si s)) (Task (wkToTk w) (tm t))
>          else f ("Loop"++(si s)) (Miseq (wkToTk w) (tm t) i BPMN.All)
>  where (x,y) = (sflow s,sflow x)
>        r = (dTot n,dTot m)
>        f ni p = (Atomic (Atom ni p [s] [x] [] r [] []), y)
>        (w,t,m,n,i) = either (\(a,b,c,d,e) -> (a,b,c,d,reps e e))
>                             (\(a,b,c,d) -> (a,b,c,d,Fix 0)) u

>mkwb :: WorkMBlock -> Seqflow -> (Element,Seqflow)
>mkwb m s = swfsub Wbs s (Right m)

>tm :: Procedure -> TaskType
>tm DetailHidden = InvT NoDetail
>tm (ProcedureA (Treatment n q m)) = (InvT (Inv n q m))
```

```
>setflows :: Element -> [Seqflow] -> [Seqflow] -> Element
>setflows (Compound a es) i o = Compound (setflows1 a i o) es
>setflows (Atomic a) i o = Atomic (setflows1 a i o)

>setflows1 :: Atom -> [Seqflow] -> [Seqflow] -> Atom
>setflows1 (Atom i t it ot e tr im om) is os = (Atom i t ni no e tr im om)
>  where (ni,no) = (if null is then it else is,if null os then ot else os)
```

G.2 Element and Sequence Flow Construction

This section defines the auxiliary functions for constructing individual BPMN elements and sequence flows. The function `ele` creates an event or a task element; the function `node` creates either a XOR gateway or an AND gateway element; the function `sflows` takes a sequence flow s and creates a list of fresh seqflows with respect to s, and the function `sflow` takes a sequence flow s and creates a fresh seqflow with respect to input argument by incrementing the integer value that uniquely identifies s.

```
>isinter :: Type -> Bool
>isinter (Irule _) = True
>isinter (Imessage _) = True
>isinter (Itime _) = True
>isinter (Ierror _) = True
>isinter _ = False

>ele :: ElementId -> Type -> Seqflow -> [Type] -> (Element,Seqflow)
>ele i t s ts
>  | disjs [isstart,isstime] t = (Atomic (Atom i t [] [s] [] norange [] []),sflow s)
>  | isinter t = (Atomic (Atom i t [s] [n] [] norange [] []),sflow n)
>  | istask t && null ts = (Atomic (Atom i t [s] [n] [] norange [] []),sflow n)
>  | istask t = (Atomic (Atom i t [s] [n] e norange [] []),sflow g)
>  | isend t = (Atomic (Atom i t [s] [] [] norange [] []),n)
>       where n = sflow s
>             (e,g) = ((par (zip ts) newest).split) (sflows n (length ts))

>type Bflow = Either [Seqflow] Int

>node :: Bflow -> Bflow -> Seqflow -> ElementId -> Type -> (Element,Seqflow)
>node (Left a) (Right b) _ i t = (node1 0 b i t a [])
>node (Right a) (Right b) s i t = (node1 a b i t [s] [])
>node (Right a) (Left b) _ i t = (node1 a 0 i t b [])
>node (Left a) (Left b) _ i t = (node1 0 0 i t a b)

>node1 :: Int -> Int -> ElementId -> Type -> [Seqflow] -> [Seqflow] -> (Element,Seqflow)
>node1 inc out i t s u
>  | (inc == 0 && out == 0) = (Atomic (Atom i t s u [] norange [] []),(sflow.newest) u)
>  | inc == 0 = (Atomic (Atom i t s o [] norange [] []),(sflow.newest) o)
>  | out == 0 = (Atomic (Atom i t is s [] norange [] []),(sflow.newest) is)
>  | otherwise = (Atomic (Atom i t is os [] norange [] []),(sflow.newest) os)
>  where is = sflows (newest s) inc
>        os = sflows (newest is) out
>        o = sflows (newest s) out

>newest :: [Seqflow] -> Seqflow
>newest = maximumBy (\x y -> cmp ((read.si) x :: Int) ((read.si) x :: Int))
>  where cmp i j | i > j = GT
>                | i < j = LT
>                | otherwise = EQ

>sflows :: Seqflow -> Int -> [Seqflow]
>sflows s 0 = []
>sflows s n = inc:(sflows inc (n-1)) where inc = sflow s

>sflow :: Seqflow -> Seqflow
>sflow s = (fst cur)++(show ((read (snd cur) :: Int)+1))
>  where cur = (splitAt 4) s
```

```
>bk :: Seqflow -> Seqflow
>bk s = (fst cur)++(show ((read (snd cur) :: Int)-1))
>  where cur = (splitAt 4) s

>gt :: Seqflow -> Seqflow -> Bool
>gt s1 s2 = (read ((drop 4) s1) :: Int) > (read ((drop 4) s2) :: Int)

>si :: Seqflow -> String
>si = snd.(splitAt 4)
```

G.3 Repeat Clauses

This section defines the functions for constructing BPMN elements modelling the repeat clauses of a sequence rule.

```
>extreps :: Seqflow -> Element -> ActivityId -> [RepeatExp] -> ([Element],Seqflow)
>extreps s e a [] = ([],s)
>extreps s e a r = ((par id last).unzip.(extreps2 s e a)) r

>extreps2 :: Seqflow -> Element -> ActivityId -> [RepeatExp] -> [(Element,Seqflow)]
>extreps2 _ _ _ [] = []
>extreps2 s e a (r:rs) = (re,ns):(extreps2 ns e a rs) where (re,ns) = extrep s e a r

>extrep :: Seqflow -> Element -> ActivityId -> RepeatExp -> (Element,Seqflow)
>extrep s e a (m,n,ml,nl,c) =
>  (Compound (Atom ("SubProcess"++(si s)) tp [s] [es] et norange [] []) [se,e,dp],ns)
>  where tp = Miseqs ("Repeat_"++(idToBn a)) RepB (reps nl ml) BPMN.All
>        (md,ad) = (dTot n,dTot m)
>        (se,ss) = ele ("Start"++((si.sflow) s)) (Stime md) (sflow s) []
>        dp = setflows e [sflow s] [ss]
>        (e,es) = ele ("End"++(si ss)) End ss []
>        (et,ns) = case c of
>                  NoCond -> ([(Itime ad,es)],sflow es)
>                  _ -> ([(Itime ad,es),(Irule c,sflow es)],(sflow.sflow) es)
```

G.4 Connecting Sequence rules

This section defines the functions that connects the subprocess elements, each modelling a sequence rule into a empirical workflow using the prerequisite and dependence trees of the sequence rules. This is described in Section 7.3.4.1. The function `flow` takes a list of subprocess elements, each modelling a sequence rule in the workflow, and a sequence flow fresh from the list of subprocesses, and returns a list of all elements, such that these elements are to be embedded in a BPMN pool and together model the control flow of the empirical workflow; the function `dpts` takes a list of sequence rules, a list of elements and a sequence flow, and connects each BPMN subprocess with its direct successors by evaluating the dependence tree of the sequence rule which the subprocess models, and the function `pres` takes a list of sequence rules, a list of elements and a sequence flow, and connects each BPMN subprocess with their direct predecessors by evaluating the prerequisite tree of the sequence rule which the subprocess models.

```
>type Assmt = (Maybe Seqflow,Maybe Seqflow)
>type Ids = (Either EventSequencing ActivityId)

>dependency :: Empiricol -> [Element] -> Seqflow -> Diagram
>dependency w es s = [("Pool",c)]
>  where c = flow w es s

>flow :: Empiricol -> [Element] -> Seqflow -> [Element]
>flow ws es s = pres ws ne (sflow ns) where (ne,ns) = dpts ws es s

>isnstop (NStop _) = True
>isnstop _ = False

>isgn e = all (not.null) [(outs.atom) e,(ins.atom) e]
```

```
>notsq = not.(disjs [isbegin,isnstop])

>pres :: Empiricol -> [Element] -> Seqflow -> [Element]
>pres [] es s = es
>pres ((Empiricol.Start _):ws) es s = pres ws es s
>pres (w:ws) es s
>  | (length.gtree.getPr) w == 1 = pres ws es s
>  | otherwise = pres ws (update es (a:mss)) rs
>  where (mss,rs) = btree Pres es ((head.ins.atom) a) ((normtree.getPr) w)
>        a = setflows ((fromJust.(findele (Left w))) es) [s] []

>dpts :: Empiricol -> [Element] -> Seqflow -> ([Element],Seqflow)
>dpts [] es s = (es,s)
>dpts ((NStop _):ws) es s = dpts ws es s
>dpts (w:ws) es s = dpts ws (update es (a:nss)) ts
>  where (nss,ts) = btree Dpts es b ((normtree.getDe) w)
>        (a,b) = cg Dpts (Left w) es (Nothing,Just s)

>cg :: TreeType -> Ids -> [Element] -> Assmt -> (Element,Seqflow)
>cg y e es (i,o) =
>  case e of
>    (Left (Empiricol.Start _)) -> maybe (st,fromJust o) cg2 em
>    (Left (NStop _)) -> maybe (en,fromJust i) cg2 em
>    (Right START) -> maybe (st,fromJust o) cg2 em
>    (Right NSTOP) -> maybe (en,fromJust i) cg2 em
>    _ -> (cg2.fromJust) em
>  where em = findele e es
>        (st,xs) = ele ("Start"++((si.fromJust) o)) BPMN.Start (fromJust o) []
>        (en,ys) = ele ("End"++((si.fromJust) i)) End (fromJust i) []
>        cg2 x | y == Pres = (setflows x (maybeToList i) (maybeToList o),maybe (fromJust o) id i)
>              | y == Dpts = (setflows x (maybeToList i) (maybeToList o),maybe (fromJust i) id o)

>btree :: TreeType -> [Element] -> Seqflow -> Tree -> ([Element],Seqflow)
>btree y es s t =
>  if (length.gtree) t > 1 then ((g:(concat fs)),(sflow.newest) ks)
>  else (par (:[]) sflow) (cg y (Right (treeid t)) es (sf y (treeid t) s))
>  where (fs,ks) = unzip (btree2 y es (sflow s) (gtree t))
>        g | y == Pres = fst (gate y t s (concatMap (outs.atom.head) fs))
>          | y == Dpts = fst (gate y t s (concatMap (ins.atom.head) fs))

>btree2 :: TreeType -> [Element] -> Seqflow -> [Tree] -> [([Element],Seqflow)]
>btree2 _ _ _ [] = []
>btree2 y es s (t:ts) =
>  if (length.gtree) t > 1 || y == Dpts then ((n,u):(btree2 y (update es n) u ts))
>  else (([(fromJust.(findele (Right (treeid t)))) es],s):(btree2 y es s ts))
>  where (n,u) = btree y es s t

>findele :: Ids -> [Element] -> (Maybe Element)
>findele (Left (Empiricol.Start d)) = find (isstart.etype.atom)
>findele (Left (NStop d)) = find (isend.etype.atom)
>findele (Right START) = find (isstart.etype.atom)
>findele (Right NSTOP) = find (isend.etype.atom)
>findele a = find (chknme ((either vname id) a))

>sf :: TreeType -> ActivityId -> Seqflow -> Assmt
>sf y START s = (Nothing,Just s)
>sf y NSTOP s = (Just s,Nothing)
>sf Pres a s = (Just (sflow s),Just s)
>sf Dpts a s = (Just s,Just (sflow s))

>gate :: TreeType -> Tree -> Seqflow -> [Seqflow] -> (Element,Seqflow)
>gate y t s ts = node i o "" ("Gate"++(si s)) (gy t) where (i,o) = gtype y s ts

>gy (OneOf ts) = Xgate
>gy (Empiricol.All ts) = Agate

>gtype Pres s ts = (Left ts,Left [s])
>gtype Dpts s ts = (Left [s],Left ts)
```

Bibliography

[ABdBR07] F. Arbab, C. Baier, F. S. de Boer, and J. Rutten. Models and temporal logics for timed component connectors. *Software and Systems Modeling*, 6(1), 2007.

[ACKM03] G. Alonso, F. Casati, H. Kuno, and V. Machiraju. *Web Services: Concepts, Architectures and Applications*. Springer, September 2003.

[ACMM07] F. Arbab, T. Chothia, S. Meng, and Y. J. Moon. Component connectors with QoS guarantees. In *Proceedings of 9th International Conference on Coordination Languages*, volume 4467 of *LNCS*. Springer, 2007.

[ACT] ActiveBPEL. `www.activebpel.org`.

[AGH05] K. Arnold, J. Gosling, and D. Holmes. *The Java (TM) Programming Language*. Addison-Wesley Professional, 2005.

[AKS08] F. Arbab, N. Kokash, and M. Sun. Towards using reo for compliance-aware business process modelling. In *Proceedings of the International Symposium on Leveraging Applications of Formal Methods, Verification and Validation*, volume 17 of *CCIS*. Springer, October 2008.

[Arb04] F. Arbab. Reo: a channel-based coordination model for component composition. *Mathematical Structures in Computer Science*, 14(03), 2004.

[BBKK09] Christel Baier, Tobias Blechmann, Joachim Klein, and Sascha Klüppelholz. A uniform framework for modeling and verifying components and connectors. In *Proceedings of the 11th International Conference on Coordination Models and Languages*, volume 5521 of *LNCS*. Springer, 2009.

[BCD+05] J. Brenton, C. Caldas, J. Davies, S. Harris, and P. Maccallum. CancerGrid: developing open standards for clinical cancer informatics. In *Proceedings of the UK e-science All Hands Meeting 2005*, 2005.

[BCPV04] A. Brogi, C. Canal, E. Pimentel, and A. Vallecillo. Formalizing Web Services Choreographies. In *Electronic Notes in Theoretical Computer Science 105*, 2004.

[BD00] C. Bolton and J. Davies. Activity Graphs and Processes. In *Proceedings of 2nd International Conference on Integrated Formal Methods*, volume 1945 of *LNCS*. Springer, 2000.

[BFS00] J. Bury, J. Fox, and D. Sutton. The PROforma guideline specification language: progress and prospects. In *EWGLP 2000: Proceedings of the First European Workshop, Computer-based Support for Clinical Guidelines and Protocols*, November 2000.

[BHF05] M. Butler, T. Hoare, and C. Ferreira. A trace semantics for long-running transactions. In *Proceedings of 25 Years of CSP*, volume 3525 of *LNCS*. Springer, 2005.

[BIZ] Microsoft biztalk server. `www.microsoft.com/biztalk`.

[BJA+08] R.S. Barga, J. Jackson, N. Araujo, D. Guo, N. Gautam, K. Grochow, and E Lazowska. Trident: Scientific workflow workbench for oceanography. In *2008 IEEE Congress on Services*. IEEE Computer Society, 2008.

[BK85] J. A. Bergstra and J. W. Klop. Algebra of communicating processes with abstraction. *Theoretical Computer Science*, 37, 1985.

[BPE03] Business Process Execution Language for Web Services, Version 1.1., May 2003. `www.ibm.com/developerworks/library/ws-bpel`.

[BR05] M. Butler and S. Ripon. Executable Semantics for Compensating CSP. In *Proceedings of 2nd International Workshop on Web Services and Formal Methods*, volume 3670 of *LNCS*. Springer, 2005.

[BSAR06] C. Baier, M. Sirjani, F. Arbab, and J. Rutten. Modeling component connectors in reo by constraint automata. *Science of Computer Programming*, 61(2), 2006.

[BtH97] A. P. Barros and A. H. M. ter Hofstede. Formal Semantics of Coordinative Workflow Specifications. Technical Report 420, Department of Computer Science and Electrical Engineering, University of Queensland, December 1997.

[BtH99] A. P. Barros and A. H. M. ter Hofstede. Modelling Extensions for Concurrent Workflow Coordination. In *CoopIS'99: Proceedings of Fourth IFCIS International Conference on Cooperative Information Systems*. IEEE Computer Society, September 1999.

[BZ07a] M. Bravetti and G. Zavattaro. A Theory for Strong Service Compliance. In *Proceedings of the 9th International Conference on Coordination Models and Languages*, volume 4467 of *LNCS*. Springer, June 2007.

[BZ07b] M. Bravetti and G. Zavattaro. Towards a Unifying Theory for Choreography Conformance and Contract Compliance. In *Proceedings of 6th International Symposium on Software Composition*, volume 4829 of *LNCS*. Springer, March 2007.

[CAL] Clinical Trials Management Tools. University of Pittsburgh, `www.dbmi.pitt.edu/services/ctma.html`.

[Cal06] R. Calinescu. Model-based SOA generation for cancer clinical trials. In *Proceedings of the 4th OOPSLA International Workshop on SOA and Web Services*, 2006.

[CAN] CancerGrid, A consortium to develop open standards for clinical cancer informatics. `www.cancergrid.org`.

[CCQS05] P. Ciccaresea, E. Caffib, S. Quaglinia, and M. Stefanelli. Architectures and tools for innovative Health Information Systems: The Guide Project. *International Journal of Medical Informatics*, 74, 2005.

[CGH+06] D. Churches, G. Gombas, A. Harrison, J. Maassen, C. Robinson, M. Shields, I. Taylor, and I. Wang. Programming scientific and distributed workflow with triana services. *Special Issue of Concurrency and Computation: Practice and Experience*, 18(10), 2006.

[CHG+07] R. Calinescu, S. Harris, J. Gibbons, J. Davies, I. Toujilov, and S. Nagl. Model-Driven Architecture for Cancer Research. In *Software Engineering and Formal Methods*, September 2007.

[CHY07] M. Carbone, K. Honda, and N. Yoshida. Structured communication-centred programming for web services. In *Programming Languages and Systems*, volume 4421 of *LNCS*. Springer, 2007.

[CPHP87] P. Caspi, D. Pilaud, N. Halbwachs, and J. Plaice. Lustre: a declarative language for programming synchronous systems. In *14th ACM Conference on Principles of Programming Languages*, January 1987.

[CPT01] C. Canal, E. Pimentel, and J. M. Troya. Compatibility and inheritance in software architectures. *Science of Computer Programming*, 41(2), 2001.

[Cre05] S. Creese. Industrial Strength CSP: Opportunities and Challenges in Model-Checking. In *Proceedings of 25 Years of CSP*, volume 3525 of *LNCS*. Springer, 2005.

[CS96] R. Cleaveland and S. T. Sims. The NCSU Concurrency Workbench. In *Proceedings of the 8th Conference on Computer-Aided Verification*, volume 1102 of *LNCS*. Springer, 1996.

[DAC99] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. In *Proceedings of the 21st International Conference on Software engineering*, 1999.

[DBG+04] E. Deelman, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, S. Patil, M. H. Su, K. Vahi, and M. Livny. Pegasus: Mapping Scientific Workflows onto the Grid. In *Grid Computing: Second European AcrossGrids Conference, AxGrids 2004*, January 2004.

[DDH72] O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, editors. *Structured programming*. Academic Press Ltd., London, UK, UK, 1972.

[DDO08] R. M. Dijkman, M. Dumas, and C. Ouyang. Semantics and analysis of business process models in BPMN. *Information and Software Technology*, 2008.

[DGHW07] M. Dumas, A. Grosskopf, T. Hettel, and M. Wynn. Semantics of Standard Process Models with OR-Joins. In *On the Move to Meaningful Internet Systems 2007: CoopIS, DOA, ODBASE, GADA, and IS*, volume 4803 of *LNCS*. Springer, 2007.

[ECH+04] H. Earl, C. Caldas, H. Howard, J. Dunn, and C. Poole. Neo-tAnGo A neoadjuvant study of sequential epirubicin + cyclophosphamide and paclitaxel +/- gemcitabine in the treatment of high risk early breast cancer with molecular profiling, proteomics and candidate gene analysis, 2004.

[Esh02] H. Eshuis. *Semantics and Verification of UML Activity Diagrams for Workflow Modelling*. PhD thesis, University of Twente, 2002.

[FaPPY07] E. Fersman, P. Krcal amd P. Pettersson, and Wang Yi. Task automata: Schedulability, decidability and undecidability. *Information and Computation*, 2007.

[FF95] M. Feather and S. Fickas. Requirements monitoring in dynamic environments. In *Proceedings of 2nd International Symposium on Requirements Engineering*. IEEE Computer Society, 1995.

[For98] Formal Systems (Europe) Ltd. *Failures-Divergences Refinement, FDR2 User Manual*, 1998. `www.fsel.com`.

[Fos06] H. Foster. *A Rigorous Approach To Engineering Web Service Composition*. PhD thesis, Imperial College London, University of London, 2006.

[FUMK04] H. Foster, S. Uchitel, J. Magee, and J. Kramer. Compatibility verification for web service choreography. In *IEEE International Conference on Web Services*, 2004.

[FUMK06] H. Foster, S. Uchitel, J. Magee, and J. Kramer. Model-Based Analysis of Obligations in Web Service Choreography. In *IEEE International Conference on Internet and Web Applications and Services*, 2006.

[GLO] The Globus Alliance. `www.globus.org`.

[GTM+04] J. M. Grimshaw, R. E. Thomas, G. MacLennan, C. Fraser, C. R. Ramsay, L. Vale, P. Whitty, M. P. Eccles, L. Matowe, L. Shirran, M. Wensing, R. Dijkstra, and C Donaldson. Effectiveness and efficiency of guideline dissemination and implementation strategies. *Health technology assessment (Winchester, England)*, 8, 2004.

[Har87] D. Harel. Statecharts: a Visual Formalism for Complex Systems. *Science of Computer Programming*, 8, 1987.

[HC06] Steve Harris and Radu Calinescu. CancerGrid clinical trials model 1.0. Technical Report MRC/1.4.1.1, CancerGrid, 2006. `www.cancergrid.org/public/documents`.

[HKNP06] A. Hinton, M. Kwiatkowska, G. Norman, and D. Parker. Prism: A tool for automatic verification of probabilistic systems. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 3920 of *LNCS*. Springer, 2006.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[Hol03] G. J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, 2003.

[HS96] P. Hammond and M. J. Sergot. Computer Support for Protocol-Based Treatment of Cancer. *Journal of Logic Programming*, 26(2), 1996.

[HSW95] P. Hammand, M. J. Sergot, and J. C. Wyatt. Formalisation of Safety Reasoning in Protocols and Hazard Regulations. In *19th Annual Symposium on Computer Applications in Medical Care*, October 1995.

[Jon03] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries The Revised Report*. Cambridge University Press, 2003.

[Jos05] M. Josephs. Models for Data-Flow Sequential Processes. In *Proceedings of 25 Years of CSP*, volume 3525 of *LNCS*. Springer, 2005.

[KA09] N. Kokash and F. Arbab. Formal behavioral modeling and compliance analysis for service-oriented systems. In *Formal Methods for Components and Objects*, volume 5751 of *LNCS*. Springer, 2009.

[KBR+05] N. Kavantzas, D. Burdett, G. Ritzinger, T. Fletcher, and Y. Lafon. *Web Services Choreography Description Language 1.0*, 2005. W3C Candidate Recommendation.

[Kie02] B. Kiepuszewski. *Expressiveness and Suitability of Languages for Control Flow Modelling in Workflows*. PhD thesis, Queensland University of Technology, Brisbane, Australia, 2002.

[Kos03] M. Koshkina. Verification of business processes for web services. Master's thesis, York University, Toronto, October 2003.

[Kus06] R. Kush. Can the protocol be standardised? Technical report, Clinical Data Interchange Standards Consortium, 2006.

[LAB+06] B. Ludäscher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger-Frank, M. Jones, E. Lee, J. Tao, and Y. Zhao. Scientific Workflow Management and the Kepler System. *Special Issue of Concurrency and Computation: Practice and Experience*, 18, 2006.

[Law05] J. Lawrence. Practical Application of CSP and FDR to Software Design. In *Proceedings of 25 Years of CSP*, volume 3525 of *LNCS*. Springer, 2005.

[LJBB06] I. Linden, J.-M. Jacquet, K. De Bosschere, and A. Brogi. On the expressiveness of timed coordination models. *Sci. Comput. Program.*, 61(2), 2006.

[LOT89] LOTOS - A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. Technical Report Technical Report 8807, International Standards Organisation, 1989.

[Low08] G. Lowe. Specification of communicating processes: temporal logic versus refusals-based refinement. *Formal Aspects of Computing*, 20(3), 2008.

[LPT07] A. Lapadula, R. Pugliese, and F Tiezzi. Calculus for orchestration of web services. In *Programming Languages and Systems*, volume 4421 of *LNCS*. Springer, 2007.

[LS00] S. Ling and H. Schmidt. Time petri nets for workflow modelling and analysis. In *Proceedings of 2000 IEEE International Conference on Systems, Man, and Cybernetics*, 2000.

[MH03] S. Modgil and P. Hammond. Decision support tools for clinical trial design. *Artificial Intelligence in Medicine*, 27, 2003.

[Mil89] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[Mil99] R. Milner. *Communicating and mobile systems: the pi calculus*. Cambridge University Press, 1999.

[MK99] J. Magee and J. Kramer. *Concurrency - State Models and Java Programs*. John Wiley, 1999.

[MP92] Z. Manna and A. Pnueli. *The temporal logic of reactive and concurrent systems*. Springer, 1992.

[MP96] Z. Manna and A. Pnueli. *Logic and Software Engineering*, chapter Clocked transition systems. World Scientific, 1996.

[MSA01] D. Moher, K. F. Schultz, and D. G. Altman. The CONSORT statement: revised recommendations for improving the quality of reports of parallel group randomized trials. *The Lancet*, 357, 2001.

[MSC96] Message Sequence Charts. Technical Report Recommendation Z.120, International Telecommunications Union, 1996.

[MSJ96] S. Miksch, Y. Shahar, and P. D. Johnson. Asbru: A Task-Specific, Intention-Based, and Time-Oriented Language for Representing Skeletal Plans. Technical Report SMI-96-0650, Stanford Medical Informatics, 1996.

[NEP] Project Neptune. `www.neptune.washington.edu/`.

[OAF+04] T. Oinn, M. Addis, J. Ferris, D. Marvin, M. Senger, M. Greenwood, T. Carver, K. Glover, M. R. Pocock, A. Wipat, and P. Li. Taverna: a tool for the composition and enactment of bioinformatics workflows. *Bioinformatics*, 20(17), 2004.

[OMG] Object Management Group. `www.omg.org`.

[OMG08] OMG. *Business Process Modeling Notation, V1.1*, February 2008. Available Specification, `www.bpmn.org`.

[ORA] Oracle BPEL Process Manager. `www.oracle.com/technology/products/ias/bpel/`.

[Oua01] J. Ouaknine. *Discrete analysis of continuous behaviour in real-time concurrent systems*. D.Phil thesis, University of Oxford, 2001.

[OvdADtH06] C Ouyang, W. M. P. van der Aalst, M. Dumas, and A. H. M. ter Hofstede. Translating BPMN to BPEL. Technical Report BPM-06-02, BPM Center, 2006.

[Pet62] C. A. Petri. *Kommunikation mit Automaten*. PhD thesis, Institut fur instrumentelle Mathematik, Bonn, 1962.

[PQ07] D. Prandi and P. Quaglia. Stochastic cows. In *Proceedings 5th International Conference on Service-oriented Computing*, volume 4749 of *LNCS*. Springer, 2007.

[PQZ08] D. Prandi, P. Quaglia, and N. Zannone. Formal analysis of bpmn via a translation into cows. In *Coordination Models and Languages*, volume 5052 of *LNCS*. Springer, 2008.

[PRO] Protocure - Integrating formal methods in the development process of medical guidelines and protocols. `www.protocure.org`.

[PTB+03] M. Peleg, S. Tu, J. Bury, P. Ciccarese, J. Fox, R. A. Greenes, R. Hall, P. D. Johnson, N. Jones, A. Kumar, S. Miksch, S. Quaglini, A. Seyfang, E. H. Shortliffe, and M. Stefanelli. Comparing computer-interpretable guideline models: a case-study approach. *Journal of the American Medical Informatics Association*, 10(1), Jan-Feb 2003.

[QSSF97] S Quaglini, R Saracco, M Stefanelli, and C Fassino. Supporting tools for guideline development and dissemination. In *Proceedings of the Sixth Conference on AIME*, 1997.

[Rei95] W. Reif. The KIV-approach to Software Verification. In *KORSO - Methods, Languages, and Tools for the Construction of Correct Software*, 1995.

[RM06] J. Recker and J. Mendling. On the Translation between BPMN and BPEL: Conceptual Mismatch between Process Modeling Languages. In *Proceedings 18th International Conference on Advanced Information Systems Engineering*, 2006.

[Ros98] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall, 1998.

[RRS05] A. W. Roscoe, J. N Reed, and J. E Sinclair. Machine-verifiable responsiveness. In *Proceedings of AVOCS 2005*, 2005.

[RSR04] J. N. Reed, J. E. Sinclair, and A. W. Roscoe. Responsiveness of interoperating components. *Form. Asp. Comput.*, 16(4), 2004.

[SB09] G. Salaün and T. Bultan. Realizability of Choreographies Using Process Algebra Encodings. In *Proceedings of 7th International Conference on Integrated Formal Methods*, volume 5423 of *LNCS*. Springer, February 2009.

[Sch00] S. A. Schneider. *Concurrent and Real Time Systems: the CSP approach*. John Wiley, 2000.

[SF03] D. R. Sutton and J. Fox. The Syntax and Semantics of PROforma Guideline Modelling Language. *Journal of the American Medical Informatics Association*, 10(5), Sep-Oct 2003.

[SLD08] J. Sun, Y. Liu, and J. S. Dong. Model checking csp revisited: Introducing a process analysis toolkit. In *Proceedings of 3rd International Symposium on Leveraging Applications of Formal Methods, Verification and Validation*, volume 17 of *CCIS*. Springer, 2008.

[SM06] G. Spanoudakis and K. Mahbub. Non intrusive monitoring of service based systems. *International Journal of Cooperative Information Systems*, 15(3), 2006.

[Spi92] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall International Series in Computer Science, 1992.

[tHN93] A. H. M. ter Hofstede and E. R. Nieuwland. Task structure semantics through process algebra. *Software Engineering Journal*, 8(1), 1993.

[UML04] Unified Modelling Language: Superstructure. Technical report, Object Management Group, 2004. availabe at `www.omg.org`.

[vdA97] W. M. P. van der Aalst. Verification of Workflow Nets. In *ICATPN '97: Proceedings of the 18th International Conference on Application and Theory of Petri Nets*, 1997.

[vdAtH05] W. M. P. van der Aalst and A. H. M. ter Hofstede. Yawl: yet another workflow language. *Information Systems*, 30(4), 2005.

[vdAtHKB03] W. M. P. van der Aalst, A. H. M. ter Hofstede, B. Kiepuszewski, and A. P. Barros. Workflow Patterns. *Distributed and Parallel Databases*, 14(3), July 2003.

[vdAvHtH+10] W. M. P. van der Aalst, K. van Hee, A. H. M. ter Hofstede, N. Sidorova, H. Verbeek, M. Voorhoeve, and M. Wynn. Soundness of workflow nets: classification, decidability, and analysis. *Formal Aspects of Computing*, 2010.

[vDdMV+05] B. van Dongen, A. de Medeiros, H. Verbeek, A. Weijters, and W. M. P. van der Aalst. The ProM Framework: A New Era in Process Mining Tool Support. In *Applications and Theory of Petri Nets 2005*, volume 3536 of *LNCS*. Springer, 2005.

[W3C] World Wide Web Consortium. `www.w3c.org`.

[W3C02] W3C. *Web Service Choreography Interface (WSCI) 1.0*, November 2002. `www.w3.org/TR/wsci`.

[WFM] Workflow Management Coalition. `www.wfmc.org`.

[WG07] P. Y. H. Wong and J. Gibbons. A Process-Algebraic Approach to Workflow Specification and Refinement. In *Proceedings of the 6th International Symposium on Software Composition*, volume 4829 of *LNCS*. Springer, March 2007.

[WG08a] P. Y. H. Wong and J. Gibbons. A Process Semantics for BPMN. In *Proceedings of 10th International Conference on Formal Engineering Methods*, volume 5256 of *LNCS*. Springer, October 2008.

[WG08b] P. Y. H. Wong and J. Gibbons. On Specifying and Visualising Long-Running Empirical Studies. In *Proceedings of International Conference on Model Transformation*, volume 5063 of *LNCS*. Springer, July 2008.

[WG08c] P. Y. H. Wong and J. Gibbons. Verifying Business Process Compatibility. In *Proceedings of 8th International Conference on Quality Software*. IEEE Computer Society, August 2008.

[WG09a] P. Y. H. Wong and J. Gibbons. A Relative-Timed Semantics for BPMN. In *Proceedings of 7th International Workshop on the Foundations of Coordination Languages and Software Architectures*, volume 229 of *ENTCS*, July 2009. Invited for special issue in Science of Computer Programming.

[WG09b] P. Y. H. Wong and J. Gibbons. Property Specifications for Workflow Modelling. In *Proceedings of 7th International Conference on Integrated Formal Methods*, volume 5423 of *LNCS*. Springer, February 2009. Invited for special issue in Science of Computer Programming.

[WG11a] P. Y. H. Wong and J. Gibbons. Formalisations and Applications of BPMN. *Science of Computer Programming*, 76(8), August 2011. Special issue of 7th International Workshop on the Foundations of Coordination Languages and Software Architectures.

[WG11b] P. Y. H. Wong and J. Gibbons. Property Specifications for Workflow Modelling. *Science of Computer Programming*, 76(10), October 2011. Special issue of 7th International Conference on Integrated Formal Methods.

[Wir71] N. Wirth. Program Development by Stepwise Refinement. *Communications of the ACM*, 14(4), 1971.

[Won06] P. Y. H. Wong. Towards A Unified Model for Workflow Processes, June 2006. Presented at 1st Service-Oriented Software Research Network workshop.

[WWF] Windows workflow foundation (winfx). `msdn.microsoft.com/workflow/`.

[XS004] XML Schema Part 2: Datatypes Second Edition, October 2004. `www.w3.org/TR/2004/REC-xmlschema-2-20041028/`.

[Yan96] J. T. Yantchev. Arc a tool for efficient refinement and equivalence checking for csp. In *Proceedings of 2nd International Conference on Algorithms and Architectures for Parallel Processing*. IEEE Computer Society, 1996.

[YSSW08] J. H. Ye, S. X. Sun, W. Song, and L. J. Wen. Formal semantics of bpmn process models using yawl. In *Proceedings of the 2008 Second International Symposium on Intelligent Information Technology Application - Volume 02*. IEEE Computer Society, 2008.

[YZQ+06] H. Yang, X. Zhao, Z. Qiu, G. Pu, and S. Wang. A formal model for web service choreography description language (ws-cdl). In *In proceedings of International Conference on Web Services 2006*. IEEE Computer Society, 2006.

[ZMS07] A. Zisman, K. Mahbub, and G. Spanoudakis. A service discovery framework based on linear composition. In *IEEE International Conference on Services Computing 2007*, 2007.

[ZYQ06] X. Zhao, H. Yang, and Z. Qiu. Towards the formal model and verification of web service choreography description language. In *In proceedings of 3rd International Workshop on Web Services and Formal Methods*, volume 4184 of *LNCS*. Springer, 2006.