A Formal Refinement Framework for the Systems Modeling Language

by

Petrus Jacobus Jacobs

Thesis presented for the degree of Doctor of Philosophy in Computer Science at the University of Oxford

Department of Computer Science
University of Oxford
Wolfson Building
Parks Road
Oxford OX1 3QD
United Kingdom

Exeter College
University of Oxford
Turl Street
Oxford OX1 3DP
United Kingdom

Supervisor: Dr. A.C. Simpson

Abstract

A Formal Refinement Framework for the Systems Modeling Language

Doctor of Philosophy in Computer Science Thesis

Petrus Jacobus Jacobs

Department of Computer Science
University of Oxford
Wolfson Building
Parks Road
Oxford OX1 3QD
United Kingdom

Exeter College
University of Oxford
Turl Street
Oxford OX1 3DP
United Kingdom

The Systems Modeling Language (SysML), an extension of a subset of the Unified Modeling Language (UML), is a visual modelling language for systems engineering applications. At present, the semi-formal SysML, which is widely utilised for the design of complex heterogeneous systems, lacks integration with other more formal approaches. In this thesis, we describe how Communicating Sequential Processes (CSP) and its associated refinement checker, Failures Divergences Refinement (FDR), may be used to underpin an approach that facilitates the refinement checking of the behavioural consistency of SysML diagrams. We do so by utilising CSP as a semantic domain for reasoning about SysML behavioural aspects: activities, state machines and interactions are given a formal process-algebraic semantics. These behaviours execute within the context of the structural diagrams to which they relate, and this is reflected in the CSP descriptions that depict their characteristic patterns of interaction. The resulting abstraction gives rise to a framework that enables the formal treatment of integrated behaviours via refinement checking. In SysML, requirement diagrams allow for the allocation of behavioural features in order to present a more detailed description of a captured requirement. Moreover, we demonstrate that, by providing a common basis for behaviours and requirements, the approach supports requirements traceability: SysML requirements are amenable to formal verification using FDR. In addition, the proposed framework is able to detect inconsistencies that arise due to the multi-view nature of SysML. We illustrate and validate the contribution by applying our methodology to a safety critical system of moderate size and complexity.

With Thanks

I am greatly indebted to my supervisor, Andrew Simpson, 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 support for guiding, reviewing and commenting on my work have made this thesis possible.

I would like to thank Jim Davies and Alessandra Cavarra for being my confirmation examiners, and for steering my research in the right direction. I would also like to extend my gratitude and thanks towards my examiners Alessandra Cavarra and Clara Benac Earle for their insightful comments and help in making this a better thesis.

I would like to thank all my colleagues and all my friends in Oxford for making this the best city possible to learn and live in.

Adam, thanks for always being there and for making every day a good one. Your unwavering support during the final stages of this DPhil is so appreciated. Love you.

I would like to thank my sister Anje for her love and support. To my parents, Karin and Casper, who have selflessly supported me in every aspect of my life — love you and thanks for everything you have done for me.

DEDICATIONS

for my grandparents

DISSEMINATION

We have submitted and presented the following papers addressing topics on the specification of safety critical case studies, requirements traceability, and on formalising and verifying SysML models using CSP. All publications are available via the following link: `http://www.cs.ox.ac.uk/people/jaco.jacobs/`.

1. J. Jacobs and A. C. Simpson. A process algebraic approach to decomposition of communicating SysML blocks. In *Proceedings of the 2nd International Conference on System Engineering and Modeling (ICSEM 2013)*, pages 153–157. IACSIT, 2013.

2. J. Jacobs and A. C. Simpson. Towards a process algebra framework for supporting behavioural consistency and requirements traceability in SysML. In *Proceedings of the 15th International Conference on Formal Engineering Methods (ICFEM 2013)*, volume 8144 of *Lecture Notes in Computer Science*, pages 266–281. Springer, 2013.

3. J. Jacobs and A. C. Simpson. On a process algebraic representation of sequence diagrams. In *Proceedings of the 1st International Workshop on Safety and Formal Methods (SaFoMe 2014)*, volume 8938 of *Lecture Notes in Computer Science*, pages 71–85. Springer, 2015.

4. J. Jacobs and A. C. Simpson. On the formal interpretation of SysML blocks using a safety critical case study. In *Proceedings of the 8th Brazilian Symposium on Software Components, Architectures and Reuse (SBCARS 2014)*. IEEE, 2015.

5. J. Jacobs and A. C. Simpson. A formal model of SysML blocks using CSP for assured systems engineering. In *Proceedings of the 3rd International Workshop on Formal Techniques for Safety-Critical Systems (FTSCS 2014)*, *Communications in Computer and Information Science*. Springer, 2015.

6. J. Jacobs and A.C. Simpson. On the Formal Interpretation and Behavioural Consistency of SysML Blocks. In *International Journal on Software and Systems Modeling*. Springer, 2015.

Contents

Abstract ii

With Thanks iii

Dedications iv

Dissemination v

Contents vi

List of Figures vii

List of Tables viii

I Introduction, Background and Literature Review 1

1 Introduction 2
- 1.1 Motivation . . . 2
- 1.2 Justification . . . 3
- 1.3 Contribution . . . 5
- 1.4 Publications . . . 7
- 1.5 Outline . . . 8

2 Background 10
- 2.1 Running Case Study . . . 10
  - 2.1.1 Actuators and Sensors . . . 11
  - 2.1.2 Travelling Crane . . . 11
- 2.2 Systems Modeling Language . . . 12
  - 2.2.1 Systems Engineering . . . 12
  - 2.2.2 Diagram Taxonomy . . . 13
  - 2.2.3 Structural Diagrams . . . 14
  - 2.2.4 Behavioural Diagrams . . . 18
  - 2.2.5 Requirement Diagram . . . 23
  - 2.2.6 Tool Support . . . 24
- 2.3 Communicating Sequential Processes . . . 24
  - 2.3.1 Patterns of Behaviour . . . 25
  - 2.3.2 Specification and Refinement . . . 31

2.3.3 Tool Support 34
2.4 Conclusion 35

3 Literature Review 36
3.1 Related Surveys 36
3.2 Methodology 37
3.3 Detailed Research Questions 39
3.3.1 Terminology 39
3.3.2 Research Questions 40
3.4 Results 42
3.4.1 Process Algebraic Model Checkers 42
3.4.2 Other Model Checking Tools 46
3.5 Other Approaches 50
3.6 Alternative Semantics 53
3.7 Discussion 53
3.8 Conclusion 55

II A Formal Refinement Framework 56

4 State Machines 57
4.1 Assumptions 57
4.2 Abstract Syntax 59
4.3 Modelling Constructs 62
4.3.1 Pivot State 62
4.3.2 Behaviours 63
4.3.3 Termination Condition 65
4.3.4 Transitions 66
4.3.5 Mapping Function 68
4.4 Behavioural Semantics 68
4.4.1 Initial State 68
4.4.2 Final and Terminate States 69
4.4.3 Junction and Choice States 71
4.4.4 Simple State 73
4.4.5 Simple Composite State 74
4.4.6 Event Queue 75
4.4.7 State Machine 75
4.5 Example 77
4.6 Discussion 80
4.6.1 Alternative Formalisations 81
4.6.2 Reflections 82
4.6.3 Possible Extensions 84
4.7 Conclusion 85

5 Activities 86
5.1 Assumptions 86
5.2 Abstract Syntax 87

5.3 Modelling Constructs . . . 90
5.3.1 Outgoing Edges . . . 90
5.3.2 Execution Environment . . . 91
5.3.3 Mapping Function . . . 91
5.4 Behavioural Semantics . . . 91
5.4.1 Control Flow . . . 92
5.4.2 Object Flow . . . 93
5.4.3 Initial Node . . . 93
5.4.4 Final Node . . . 94
5.4.5 Send Signal Event Action . . . 94
5.4.6 Receive Signal Event Action . . . 96
5.4.7 Value Specification Action . . . 97
5.4.8 Opaque Action . . . 97
5.4.9 Call Behaviour Action . . . 99
5.4.10 Fork Node . . . 100
5.4.11 Join Node . . . 101
5.4.12 Decision Node . . . 101
5.4.13 Merge Node . . . 102
5.4.14 Activity . . . 103
5.5 Example . . . 105
5.6 Discussion . . . 108
5.6.1 Alternative Formalisations . . . 108
5.6.2 Reflections . . . 109
5.6.3 Possible Extensions . . . 111
5.7 Conclusion . . . 112

6 Interactions **113**
6.1 Assumptions . . . 113
6.2 Abstract Syntax . . . 113
6.3 Modelling Constructs . . . 115
6.3.1 Occurence Observations . . . 115
6.3.2 Operators . . . 115
6.3.3 Template Approach . . . 115
6.3.4 Process Definitions . . . 116
6.4 Behavioural Semantics . . . 118
6.4.1 Sequencing Operators . . . 118
6.4.2 Interaction Operators . . . 120
6.4.3 Interaction Interpretation . . . 128
6.5 Example . . . 131
6.6 Discussion . . . 132
6.6.1 Alternative Formalisations . . . 132
6.6.2 Reflections . . . 133
6.6.3 Possible Extensions . . . 134
6.7 Conclusion . . . 134

7 Refinement Checking SysML **136**
7.1 Introduction . . . 136

7.2 Structural Counterparts . . . 138
7.2.1 Enumerations and Signals . . . 138
7.2.2 Blocks, Parts, and Connectors . . . 138
7.3 Behavioural Consistency . . . 141
7.3.1 A SysML Model . . . 141
7.3.2 Integrative Example . . . 142
7.4 Compositional Approach to Specification and Design . . . 147
7.4.1 Interpretations . . . 148
7.4.2 Compositional Design . . . 150
7.5 Requirements Traceability . . . 151
7.6 Discussion . . . 153
7.6.1 Contribution . . . 153
7.6.2 Related Work . . . 154
7.6.3 Future Work . . . 154
7.7 Conclusion . . . 155

III Case Study, Discussion and Conclusion 156

8 Case Study 157
8.1 A Production Cell . . . 157
8.1.1 Components . . . 158
8.1.2 Operation . . . 161
8.2 Requirements . . . 164
8.2.1 Safety Requirements . . . 164
8.2.2 Other Requirements . . . 165
8.3 Model . . . 165
8.3.1 Low Level Components . . . 167
8.3.2 High Level Components . . . 170
8.4 Results . . . 174
8.5 Conclusion . . . 176

9 Discussion and Conclusion 178
9.1 Contributions . . . 178
9.2 Shortcomings and Limitations . . . 180
9.3 Future Work . . . 180
9.4 Concluding Remarks . . . 181

List of References 182

Appendices 192

A SysML Model 193
A.1 Low Level Components . . . 193
A.2 Intermediate Components . . . 196
A.3 High Level Components . . . 197

B CSP Model 204

B.1 Low Level Components . . . 204
B.1.1 Bidirectional Motor . . . 204
B.1.2 Unidirectional Motor . . . 205
B.1.3 Magnet . . . 206
B.1.4 Potentiometer . . . 207
B.1.5 Switch . . . 208
B.1.6 Photoelectric Sensor . . . 208
B.2 Intermediate Components . . . 209
B.2.1 Arm Controller . . . 209
B.2.2 Arm . . . 211
B.3 High Level Components . . . 211
B.3.1 Feed Belt . . . 211
B.3.2 Rotary Table . . . 212
B.3.3 Rotary Robot . . . 214
B.3.4 Metal Press . . . 216
B.3.5 Deposit Belt . . . 217
B.3.6 Travelling Crane . . . 219

List of Figures

1.1 Research contributions in relation to thesis structure and research questions. . 9

2.1 Side view of the crane. 11
2.2 Classification of the diagrams found in SysML. 14
2.3 An example block definition diagram. 16
2.4 An example internal block diagram. 17
2.5 State machine notation relevant to our formalisation. 20
2.6 An example state machine diagram. 20
2.7 Activity notation relevant to our formalisation. 21
2.8 An example activity diagram. 22
2.9 Interaction notation relevant to our formalisation. 23
2.10 An example sequence diagram. 23
2.11 An example requirement diagram. 24
2.12 State transition graphs of *DElectromagnet* and *NDElectromagnet*. 34

4.1 State machine M_1. 62
4.2 State machine M_2. 69
4.3 State machine M_3. 70
4.4 State machine M_4. 72
4.5 The state machine diagram of the BDMotor block. 77
4.6 The state machine diagram of the Magnet block. 78
4.7 The state machine diagram of the PDMeter block. 79
4.8 The state machine diagram of the Controller block. 79
4.9 State machine M_5. 82

5.1 Activity A_1. 89
5.2 Activity A_2. 92
5.3 Activity A_3. 93
5.4 Activity A_4. 95
5.5 Activity A_5. 97
5.6 Activity A_6. 98
5.7 Activity A_7. 99
5.8 Activity A_8. 100
5.9 Activity A_9. 101
5.10 Activity A_{10}. 102
5.11 Activity A_{11}. 104
5.12 The activities TurnOnBDMotor and TurnOffBDMotor. 105

5.13 The activities Extend and Retract. 106
5.14 The activities Magnetise and Demagnetise. 107
5.15 The activities ActivateMagnet and DeactivateMagnet. 108
5.16 The activity OnSense. 108
5.17 Activity A_{12}. 110

6.1 Interaction I_1. 114
6.2 Interaction I_2. 119
6.3 Interaction I_3. 120
6.4 Interaction I_4. 121
6.5 Interaction I_5. 122
6.6 Interaction I_6. 123
6.7 Interaction I_7. 124
6.8 Interaction I_8. 125
6.9 Interaction I_9. 126
6.10 Interaction I_{10}. 128
6.11 Interaction I_{11}. 128
6.12 Interaction I_{12}. 130
6.13 Interaction Grasp. 131

7.1 Enumeration and signal definitions. 138
7.2 Provided and required receptions of blocks B_1 and B_2. 139
7.3 Block definition and internal block diagrams for B_3. 140
7.4 State machine M_6. 142
7.5 State machine M_7. 142
7.6 Activites A_{12}, A_{13} and A_{14}. 143
7.7 Sequence diagram showing interaction I_{13} between b_1 and b_2. 146
7.8 State machine M_8. 149
7.9 The compositional approach afforded by CSP. 150
7.10 Requirement R. 152
7.11 Internal block definition diagram for B_4. 154

8.1 Top-down view of the production cell. 158
8.2 Top-down and side views of the feed belt. 158
8.3 Top-down and side views of the elevating rotary table. 159
8.4 Top-down and side views of the rotary robot. 159
8.5 Top-down and side views of the press. 160
8.6 Top-down and side views of the deposit belt. 160
8.7 Side view of the crane. 161
8.8 Plate conveyed onto the table. 162
8.9 Robot in the initial position. 162
8.10 Robot in the position where the upper arm picks a blank plate from the table. . 163
8.11 Robot in the position where the lower arm picks a pressed plate from the metal press. 163
8.12 Robot in the position where the lower arm drops a pressed plate onto the deposit belt. 163

8.13 Robot in the position where the upper arm drops a blank plate in the metal press. 164
8.14 Side view of the crane's arm in the extended and retracted positions. 165
8.15 Components of the case study separated by level. 166
8.16 Composition of the components. 166
8.17 Unidirectional motor state machine. 167
8.18 TurnOnUDMotor activity. 168
8.19 Switch state machine. 169
8.20 OnDetect activity. 169
8.21 Metal press state machine. 171
8.22 MPInLoad activity. 172
8.23 Travelling crane state machine. 173
8.24 MPInLoad activity. 174

A.1 Bidirectional motor diagrams. 193
A.2 Unidirectional motor diagrams. 194
A.3 Magnet diagrams. 194
A.4 Potentiometer diagrams. 195
A.5 Switch diagrams. 195
A.6 Photoelectric sensor diagrams. 195
A.7 Arm controller diagrams. 196
A.8 Arm diagrams. 197
A.9 Block definition diagram of the complete system. 197
A.10 Feed belt diagrams. 198
A.11 Rotary table diagrams. 199
A.12 Rotary robot diagrams. 200
A.13 Metal press diagrams. 201
A.14 Deposit belt diagrams. 202
A.15 Travelling Crane diagrams. 203

List of Tables

3.1 Classification of process algebraic approaches by variant, diagram and purpose. 47
3.2 Classification of process algebraic approaches by tool, feedback, automation, integration and evaluation. 48
3.3 Classification of other model checking approaches by variant, diagram and purpose. 51
3.4 Classification of other model checking approaches by tool, feedback, automation, integration and evaluation. 52
3.5 Classification of other formal approaches by variant, diagram and purpose. . . 53

8.1 Signals used by the high level components of the production cell. 170

PART I

INTRODUCTION, BACKGROUND AND LITERATURE REVIEW

INTRODUCTION

1.1 Motivation

State-of-the-art systems are typically organic, multi-disciplinary compositions of interconnecting components or systems, functioning as a whole in order to achieve a shared goal. This makes it increasingly difficult to orchestrate combined behaviour, and to ensure that the intended behavioural characteristics of the complete system are adhered to, whilst maintaining operational independence of the components or systems that make up the whole. The need to establish solutions to technologically challenging problems, often in a short time frame, further complicates the matter. Moreover, the environment external to a system is constantly evolving and ever more demanding, resulting in external interactions of increased complexity. These interactions are more taxing either because we are using the system in an unfamiliar or previously unforeseen context, or the systems themselves are more complex. Whatever the case may be, it is apparent that we require a systematic approach to deal with this inherent complexity and entanglement.

The OMG's[1] *Systems Modeling Language* (SysML) [**1**] is a graphical modelling notation that can be used to describe complex, heterogeneous, multi-disciplinary systems comprised of various components. These components might be simple structural elements, or might themselves be viewed as systems comprised of various components working together. This recursive decomposition allows for considerable expressive power when it comes to modelling system behaviour. In addition, domain-specific languages can be used to further model components, for example the *Unified Modeling Language* (UML) [**2**] for software, or the *VHSIC*[2] *Hardware Description Language* (VHDL) [**3**] for hardware.

Communicating Sequential Processes (CSP) [**4**] is a process algebra used to describe complex patterns of interaction between processes, with each process having its own characteristic behaviour. In contrast to SysML, which can at best be classified as a semi-formal notation, CSP has a rigorous mathematical underpinning and is a formal approach to specifying complex behaviour. In addition, a well-established refinement theory with various semantic models is defined for CSP processes. Moreover, the existence of *Failures Di-*

[1] `http://www.omg.org`

[2] This, in turn, stands for *Very High Speed Integrated Circuit*.

vergences Refinement (FDR) [**5**; **6**; **7**; **8**], a model checker[3] for refinement checking CSP, readily allows for the reasoning about correctness of designs and verification that asserted properties hold.

The two notations described above clearly differ with regards to their users and application domains. The CSP user base is quite niche and largely academic, however the range of application is impressive: cryptanalysis of security protocols [**9**]; detecting errors in telecommunications protocols [**10**]; safety critical applications, for example railways signalling [**11**]; and mission critical applications, for example the modelling of spacecraft behaviour [**12**]. In contrast, the SysML user base lies mainly in the systems engineering community. It would be beneficial, however, to develop a framework that integrates these approaches and offers the advantages of both. Specifically, by translating SysML into CSP we will precisely define the intended behaviour of a given SysML model, by making use of the underlying formal semantics of CSP. Consequently, this allows refinement checking of the SysML model.

The intention here is not to replace existing SysML modelling tools, but rather to develop a formal framework that can be used in conjunction with such tools in order to complement the modelling activity undertaken.

SysML allows for the specification of a system using different diagram kinds depending on the level of detail required, the context of use, and the nature of what is being specified. For example, block diagrams are suited for describing the structural composition of entities in a design, whereas state machine diagrams are geared towards the specification of state-based behaviour. Considered in isolation, these diagrams allow the modeller to describe the system from various perspectives, as appropriate. However, when viewed in unison their amalgamation must form a complete and coherent description of the system. Checking consistency between the various diagrams is an important step towards ensuring well-formedness of designs. The removal of specification-level anomalies ensures that the design is realisable and that it can subsequently be implemented. Ultimately, having a formal framework in place would not only be beneficial in detecting inconsistencies in SysML models, but would at the same time lay the foundations of a formal refinement framework that can assist, for example, in the model-based testing of the resulting formal model. Our goal, then, is to integrate semi-formal and formal approaches to specification in order to provide the modeller with the benefits of both approaches.

1.2 Justification

In this section we provide the rationale for respectively selecting SysML and CSP as our source and target notations in a framework that integrates graphical notations and formal methods.

The justification for selecting SysML as our semi-formal notation can be summarised thus.

- Industrial focus is shifting towards a standardised approach in the design and integration of large scale systems. SysML is the dominant standard emerging for this purpose and has the backing of the OMG. Moreover, some of the most complex systems that require integration are safety or mission critical: collision avoidance,

[3] Strictly speaking, FDR is not a model checker, but a refinement checker.

navigational, communications and black box systems within the avionics industry; mission or tactical avionics, military communications, radar, sonar and weapons systems within the defence industry.

- SysML is a standard, rather than a profile of the UML like UML-RT [**13**]. The uptake of a formal semantics is therefore more likely than in the case of a profile.

- SysML is less software-centric than UML and geared towards the specification of a system comprised of various subsystems. As such, it has far fewer diagrams and modelling constructs than UML. We argue that it is therefore much easier to provide a formal semantics: the fact that we have fewer diagrams and fewer interactions between different diagram kinds simplifies the task of defining a formal semantics.

- There is typically only one kind of diagram available to the modeller in order to specify a certain type of behaviour. For example, SysML only provides sequence diagrams to specify interactions, whereas in UML the modeller has a choice between sequence diagrams or interaction overview diagrams. The more restrictive approach adopted by SysML makes it easier to define formal behavioural semantics: the choices available to the modeller are limited. It follows that SysML models are generally more consistent than their UML counterparts. The fact that most organisations adopt a particular systems engineering methodology further solidifies this fact [**14**].

- The behavioural semantics defined as per the OMG SysML standard [**1**] uses natural language. We simply wish to formalise this in order to give a precise description of intended behaviour.

- SysML places significant emphasis on requirements traceability. It allows us to adorn requirements with a more detailed description of intent and ultimately link this back to the model of the system. The requirement diagram is introduced in SysML as a core part of the language; UML has no equivalent counterpart.

The justification for selecting CSP as our semantic base can be characterised as follows.

- Tool support is readily available in the form of a refinement checker, FDR. Other useful tools include process animators like Probe[4].

- CSP has a well-defined, formal underpinning: denotational, operational and algebraic semantics exist. In addition, congruence theorems link the different semantic models. The interested reader can refer to [**15**] for a detailed account of the underlying theory of CSP.

- CSP's approach to process composition, combined with the fact that refinement is preserved within context, would allow us to decompose a complex design of a system (or system of systems) in such a way that the refinement analysis is computationally feasible.

[4]FDR version 3 [**8**] has integrated support for Probe via the user interface.

- The behavioural aspects of SysML can be modelled naturally by a process-algebraic formalism such as CSP.
- The compositional nature of SysML complements that of CSP.
- Mobility, as supported by the π-calculus [**16**], is not required due to the fact that dynamically evolving systems are not within our scope. The structural aspects of a SysML model are static components that inherently determine the composition of process descriptions within our CSP model. SysML connectors dictate the way in which the classifier behaviour of the respective blocks communicate. However, once defined, the blocks and the connectors remain static within the context they are placed and cannot be dynamically reconfigured during system execution. The classifier behaviours, modelled by appropriate CSP processes, are the only dynamic components here. We thus have no use for mobility owing to the fact that the composition of the CSP process network remains static during computation. The interested reader may refer to [**17**] for an account of how to model mobility in CSP.

When mapping between two notations it is important that the goals of the transformation be stated clearly in order to evaluate the success of the transformation. Our goals can be summarised thus.

- We aim to provide a mathematically sound behavioural framework that allows reasoning about SysML behaviours in an integrated fashion.
- We aim to include constructs that are sufficiently powerful to allow for the design and analysis of systems of systems of reasonable size and sizeable complexity.
- We aim to include constructs in the source notation that would result in a clean and elegant representation in the target notation. SysML constructs that would noticeably compromise the clarity of the resulting CSP formulation are avoided as far possible.
- We aim to exclude SysML constructs that would significantly contribute to the CSP state space, but would not result in a comparative increase in terms of SysML modelling capability.

It is evident that some of the goals above are clearly conflicting. Ultimately, we will have to compromise when selecting a suitable subset of SysML for formalisation. Throughout we will document our rationale for either including or excluding a particular construct. We refer back to the aforementioned when we discuss the results of this thesis in Chapter 9.

1.3 Contribution

This section presents the research questions that we seek to answer and subsequently states the contributions of this thesis.

The main focus is on giving a behavioural semantics to SysML behavioural diagrams using the process algebra CSP. The behavioural formalisms of interest to us are state machines, activities and interactions. However, when attempting to formalise behaviour, we still need to consider structural aspects: if we are to define a sensible behavioural

semantics, we cannot be agnostic with respect to the static composition of the SysML model. Structural diagrams provide the context in which behaviours execute. It follows that the resulting CSP process network and its associated communication configuration need to reflect the structural specification of the original SysML model. The intention is then to exploit this overarching semantic framework, encompassing behavioural as well as the relevant structural formalisms, in order to ensure a sound and consistent design. We broadly characterise the different types of design consistency conceivable in SysML thus.

- Consistency among the same structural formalisms, i.e. *intra-structural consistency*.
- Consistency between different structural formalisms, i.e. *inter-structural consistency*.
- Consistency among the same behavioural formalisms, i.e. *intra-behavioural consistency*.
- Consistency between different behavioural formalisms, i.e. *inter-behavioural consistency*.
- Consistency between structural and behavioural formalisms.

Reasoning about inter-behavioural as well as intra-behavioural consistency is a rather cumbersome activity for the human mind: our cognitive ability to cope with multiple descriptions of behaviour, and ultimately fuse these into a unified interpretation, is rather limited. We need to augment our mental capacities with appropriate notations in order to reason effectively about such behaviours. Moreover, if we are going to utilise these notations in a meaningful fashion we require mechanised tool support. CSP is one such notation, backed up by FDR in the form of a refinement checker.

Requirements traceability plays an important role as part of any model-based systems engineering methodology. In SysML, requirements can be related to other requirements, as well as to model elements via one or more relationships: a behavioural construct can be allocated to a particular requirement, and we can subsequently use FDR to ensure that the model satisfies it. We achieve this by exploiting an overarching semantic framework for a carefully selected subset of SysML model elements. This framework enables us to:

- provide a unified interpretation for those model elements that collectively make up our design in order to form a complete description of our system;
- assign a detailed interpretation to those model elements that represent system requirements; and
- associate CSP refinement relations with appropriate SysML relationships in order to check whether requirements are satisfied in the system design.

Requirements traceability is undoubtedly a good thing; a formal representation of requirement diagrams in CSP gives rise to a formal means of requirements traceability. This provides a formal foundation for this otherwise informal technique that serves as justification for the development of a formal framework.

The research questions driving the work described in this thesis can therefore be stated thus.

RQ1 Can a formal refinement framework, defined over SysML diagrams, be helpful in pointing out inconsistencies in a SysML model, and ultimately lead to a more cohesive and well-formed SysML model?

RQ2 Is it possible to define a framework that supports requirements traceability through formal refinement checking?

Any approach that converts between the two paradigms needs to be precisely defined. This is especially applicable as we are using a formal methodology, CSP, as our target model; it is of utmost importance that the CSP model *conforms* to the SysML model it is based upon.

The research contributions can be summarised as follows.

RC1 The definition of a formal behavioural semantics for SysML. By modelling SysML diagrams in CSP, we provide the former with a precisely defined behaviour, and, consequently, a mathematical framework that can be used to reason and compare different behaviours.

RC2 The use of a refinement framework to detect inconsistencies in SysML models. Refinement is used as a basis for determining whether models are well-formed. By well-formed we mean whether the different diagrams are generally consistent with each other, and whether the collective behavioural model prescribed is cohesive.

RC3 The use of a refinement framework to automate requirements traceability in SysML models. Requirements are formalised via SysML behavioural diagrams. Refinement is then used as a basis for determining whether the stated requirements imposed on the model hold.

Secondary research contributions can be stated thus.

SC1 A detailed account of past and current research on the topic of integrating process-algebraic approaches with behavioural formalisms is presented in the form of a systematic literature survey. In particular we look at the behavioural formalisms relevant to SysML: state machines, activities and interactions.

SC2 Our methodology is evaluated within the context of a safety critical system of moderate size and complexity. To the best of our knowledge this is the first time an integrated formal approach is validated in this context: other approaches are typically demonstrated only on pedagogic examples.

1.4 Publications

We have submitted and presented the following papers addressing topics on the specification of safety critical case studies, requirements traceability, and on formalising and verifying SysML.

- J. Jacobs and A. C. Simpson. A process algebraic approach to decomposition of communicating SysML blocks. In *Proceedings of the 2nd International Conference on System Engineering and Modeling (ICSEM 2013)*, pages 153–157. IACSIT, 2013.

- J. Jacobs and A. C. Simpson. Towards a process algebra framework for supporting behavioural consistency and requirements traceability in SysML. In *Proceedings of the 15th International Conference on Formal Engineering Methods (ICFEM 2013)*, volume 8144 of *Lecture Notes in Computer Science*, pages 266–281. Springer, 2013.
- J. Jacobs and A. C. Simpson. On a process algebraic representation of sequence diagrams. In *Proceedings of the 1st International Workshop on Safety and Formal Methods (SaFoMe 2014)*, volume 8938 of *Lecture Notes in Computer Science*, pages 71–85. Springer, 2015.
- J. Jacobs and A. C. Simpson. On the formal interpretation of SysML blocks using a safety critical case study. In *Proceedings of the 8th Brazilian Symposium on Software Components, Architectures and Reuse (SBCARS 2014)*. IEEE, 2015. *(To appear)*.
- J. Jacobs and A. C. Simpson. A formal model of SysML blocks using CSP for assured systems engineering. In *Proceedings of the 3rd International Workshop on Formal Techniques for Safety-Critical Systems (FTSCS 2014)*, *Communications in Computer and Information Science*. Springer, 2015. *(To appear)*.

1.5 Outline

The thesis is divided into three main parts, namely: *Introduction, Background and Literature Review*; *A Formal Refinement Framework*; and *Case Study, Discussion and Conclusion*. It is structured as such to provide a logical separation between the introductory material, the core contribution, and an application of the contribution to a moderately sized case study in order to serve as a proof of concept and validate the research contribution.

Part I (Chapters 1–3) gives an overview of the proposed research and introduces the necessary background material required to support the contribution. In addition, we present a detailed literature review to put the contribution into context. The primary aim of these chapters is to provide general background information pertinent to the work presented in this thesis, rather than provide exhaustive coverage of the respective topics. Chapter 2 briefly introduces: the running case study; the OMG's SysML; and the language of CSP, along with the necessary semantic models of refinement required in later chapters. Chapter 3 is the result of a systematic literature review.

Part II (Chapters 4–7) embodies the core of the thesis. Each of the contributions listed in Section 1.3 map to one or several chapters in this part. Chapters 4–6 provide a starting point for modelling the different types of SysML diagrams using the process algebra CSP. The mapping from SysML to CSP is done in a fashion that is completely agnostic to any automated transformation approach, with the focus on deriving clear and concise counterparts in CSP. We define transformation templates that serve as exemplars of our methodology. In the latter part of each of these chapters we apply these transformations to a running case study in order to illustrate our approach. Chapter 7 is concerned with refinement-checking SysML diagrams, and considers how a formal framework can

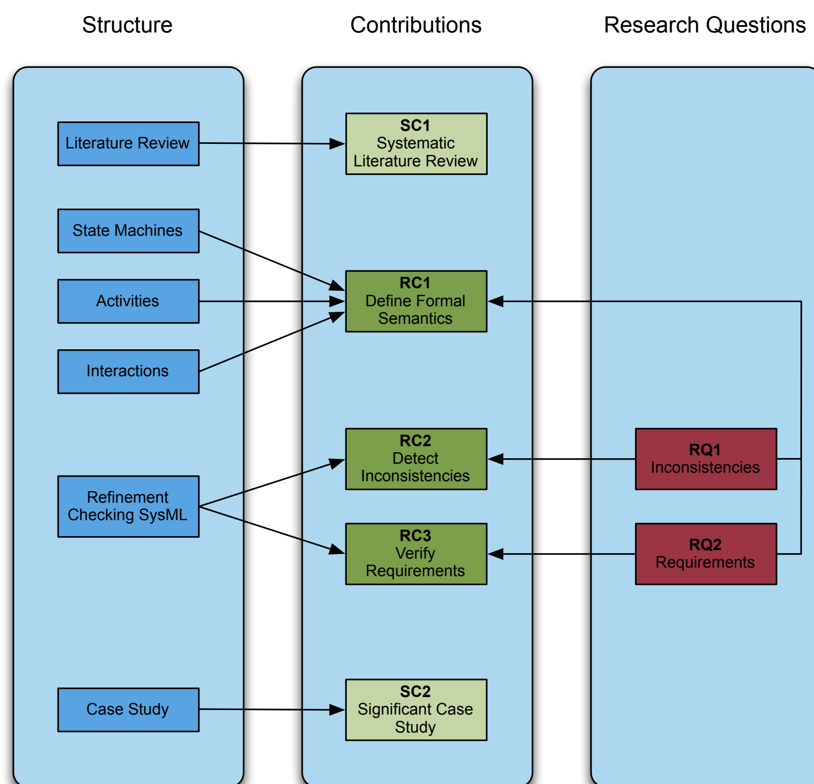


Figure 1.1: Research contributions in relation to the thesis structure and the research questions posed. On the left we have the core chapters, in the centre the research contributions, and on the right the research questions we seek to answer.

be used to detect inconsistencies. Furthermore, we investigate how behavioural requirements, suitably mapped to the same semantic domain, can be verified against the resulting CSP model. We demonstrate how the resulting framework underpins an approach that facilitates the refinement checking of behavioural consistency of SysML diagrams and supports requirements traceability in an automated fashion. Figure 1.1 presents an overview of the structure of the thesis in relation to the contributions and research questions posed in Section 1.3.

Part III (Chapters 8–9) contains an example case study and concludes with a discussion on the outcome of the thesis. Chapter 8 studies a system of moderate size and complexity that is used as a means of illustrating and validating the work presented in the preceding chapters. The case study is that of a safety critical system: a production cell that forms part of a larger installation, located in an actual factory in Germany. Chapter 9 summarises the contributions of the thesis, comments on the weaknesses and limitations of the various techniques employed, and outlines possible avenues of further work.

We relegate further details on the case study of Chapter 8 to the appendices. Appendix A lays out the SysML model; Appendix B presents the corresponding CSP formalisation.

2

BACKGROUND

This chapter serves as an introduction to both the OMG's SysML and the process algebra CSP. The primary objective of the presentation is to provide general background information pertinent to the work presented in this thesis, rather than provide exhaustive coverage of the respective topics. More information on SysML can be found either in the specification [**1**], or in one of the textbooks covering the topic [**14**; **18**]. The reader interested in obtaining more information on CSP can refer to one of several relevant textbooks [**4**; **15**; **17**; **19**]. However, before we introduce SysML and CSP, we provide a brief narrative surrounding our running case study.

2.1 Running Case Study

In this section we provide a running case study that will be used to drive the development of our formal framework. We consider a single component — the travelling crane — of the fully fledged case study introduced in Chapter 8. The complete case study is modelled after an industrial installation of a metal processing plant located in Karlsruhe, Germany. The interested reader should refer to the original case study for additional information [**20**].

The travelling crane can be considered a system in its own right. Both case studies were selected on the basis that they fit in neatly with the underlying principles of systems engineering: they enable us to demonstrate, in a pedagogic setting, the integration and design of a system composed of complex subsystems.

In addition to the travelling crane, we also introduce actuator and sensor elements that are, in turn, the constituents of the components of our complete system. Some of the actuators and sensors are not components of the travelling crane. However, we introduce them here as they are instructive in demonstrating some of the concepts relevant to our methodology. As such, we employ them as illustrative examples throughout the remaining chapters.

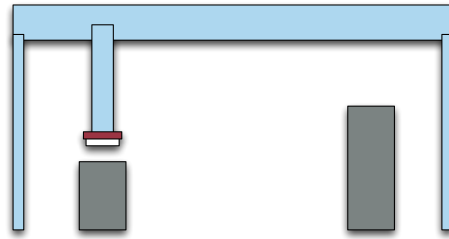


Figure 2.1: Side view of the crane.

2.1.1 Actuators and Sensors

Before we are able to describe our system of interest, we need to understand the function of the actuators and sensors that are, in turn, constituents of the components that make up our system.

Actuators and sensors communicate with the system controller. *Actuators* receive outputs from the controller in order to coordinate the operation of several components. Examples of actuators include the unidirectional and bidirectional motors, as well as the electromagnets. A *unidirectional motor* is an electric motor that can only provide power in a single direction; a *bidirectional motor* can operate in two opposing directions. An *electromagnet* can activate or deactivate a magnetic field using an electric current.

Sensors send inputs to the controller; examples of sensors are the switches, potentiometers and photoelectric sensors. A *switch* is activated when a certain position is reached. A *potentiometer* provides a value within a certain range such as the angle of rotation, or gives an indication as to the range of extension. A *photoelectric sensor* is used to detect the presence of an object. A photoelectric sensor consists of a transmitter located within the line of sight of a receiver. A ray of light is beamed from the transmitter to the receiver. In this arrangement, an object is detected when it intercepts the ray of light; conversely, no object is detected if the beam reaches the receiver.

2.1.2 Travelling Crane

A *travelling crane* transfers objects between its two ends using a central arm. The objects being transported could conceivably be picked up or dropped from different heights. For example, the crane might be used to pick up metal objects from one crate and place them in another, where the crates have different heights.

The crane has a bidirectional motor responsible for the horizontal movement of the central arm between the crane ends. Sensors are placed above the two ends so as to indicate the position of the arm. The arm is equipped with a bidirectional motor responsible for vertical extension. An electromagnet is placed at the front of the arm for handling objects; a potentiometer is present to indicate the range of extension of the arm. Refer to Figure 2.1 for the side view of the crane.

The travelling crane in this section is used in a slightly different context than the version in Chapter 8: it transports metal objects between two crates rather than a crate and a conveyor belt. The actual design of the crane, however, remains the same.

2.2 Systems Modeling Language

The *Systems Modeling Language* (SysML) [**1**] is a standardised, general purpose, graphical modelling notation utilised for the design, specification and visualisation of a diverse range of complex systems, forged from the composition of interconnecting components or subsystems. It is used extensively in the field of systems engineering as part of a structured approach to systems design. The intention is to specify the *system of interest* within a particular context of use. Within this context alternative models can then be evaluated and compared against each other, and the most appropriate design selected.

The start of this section provides a high level introduction to systems engineering, insofar as to place SysML within the context of which it is used. Next, we provide a brief overview of the different types of diagrams present in the OMG SysML specification. All diagrams are introduced, but greater emphasis is placed on the structural and behavioural diagrams relevant to our formalisation. We conclude the section with a brief discussion on current modelling tools.

2.2.1 Systems Engineering

Systems engineers are tasked with the specification and integration of complex, large-scale systems. A keystone of this activity is ensuring that requirements, as imposed by the various stakeholders[1], are adequately captured and subsequently addressed when specifying a potential solution.

A *model* is an appropriate abstraction of reality that aids our understanding of a particular problem space. It identifies complex aspects of the problem and promotes communication amongst the stakeholders. Holt and Perry [**18**] list a number of features essential to an effective systems modelling language:

- it should allow for independent views of the same system;
- it should, within reason, be meaningful to any observer lacking specialist knowledge;
- it should be connected to reality;
- it should provide an adequate level of abstraction;
- it should permit a trade-off analysis between competing solutions; and
- it should allow requirements traceability linking stakeholder requirements back to the design.

Moreover, to create quality system models a disciplined *systems engineering methodology* needs to be followed. A flexible modelling language would exhibit the aforementioned features without imposing any specific methodology.

Several systems engineering methodologies exist. A particular methodology has a precisely specified workflow and set of artefacts associated with it. The modelling activity is carried out in a particular order and each stage stipulates the modelling notations to

[1] We use the term stakeholders to denote those who have a vested interest in the system (in any phase of the life cycle).

be used. For example, a *functional decomposition approach* would break down high-level functionality into smaller functional units and assign these to appropriate components. In contrast, a *scenario-based approach* is based on an analysis of the interaction between structural components. If SysML is used, the first approach would stipulate activity diagrams to model functional behaviour, whereas the second would insist upon sequence diagrams for modelling interactions. A detailed discussion on the different systems engineering methodologies is beyond the scope of this thesis; the interested reader is instead referred to a literature survey on the topic [**21**].

The three standard texts applicable to systems engineering are the *International Standards Organization* (ISO) 15288 standard [**22**], the *Institute of Electrical and Electronics Engineers* (IEEE) 1220 standard [**23**], and the *International Council on Systems Engineering* (INCOSE) best-practice handbook [**24**]. The OMG, the INCOSE and the ISO, along with several industrial partners (Lockheed Martin Corporation, The Boeing Company, BAE Systems, etc.), tool vendors (Sparx Systems, IBM, etc.) and universities contributed to the definition of SysML in its current form. We now introduce SysML and expand upon the modelling constructs applicable to us. The introduction that follows is based on [**14**].

2.2.2 Diagram Taxonomy

SysML is a more compact language than UML. This fact is reflected by the total number of diagrams and constructs present in the specification: SysML has a total of nine diagrams, whereas UML has fourteen. SysML is an extension of a subset of UML: some of the UML diagrams are reused (state machine diagram, sequence diagram, and package diagram); some are extended (activity diagram); some are modified (block definition diagram and internal block diagram modify the class diagram and composite structure diagram, respectively); and others are newly introduced (requirement diagram and parametric diagram).

The focus of this thesis is to provide SysML with a behavioural semantics. However, in order to do so we need to consider the context in which the behavioural diagrams execute. As such, our formalisation also needs to take into account some of the structural diagrams. In addition, requirement diagrams allow for the allocation of a behavioural diagram to further refine and formalise the behaviour attributed to a specific requirement. Therefore, the diagrams we deem relevant to our formalisation are as follows.

- **Structural diagrams**: block definition diagram and internal block diagram.
- **Behavioural diagrams**: state machine diagram, activity diagram, and sequence diagram.
- Requirement diagram.

Figure 2.2 shows the nine diagrams of SysML grouped according to the diagram type: the behavioural diagrams, the requirement diagram, the parametric diagram, and the structural diagrams. The diagrams we consider in this thesis are shaded in green.

We provide brief introductions to each of the considered diagrams in Sections 2.2.3–2.2.5. An overview of the remaining diagrams follows. We do not consider these in our treatment, and, as such, omit a detailed discussion here.

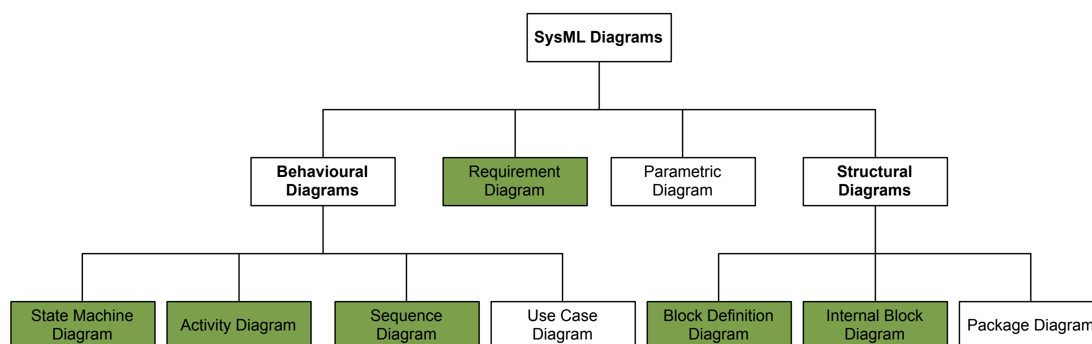


Figure 2.2: Classification of the diagrams found in SysML. Diagrams shaded in green will be considered in our treatment.

Package Diagram

Package diagrams provide a means to organise and partition a model into logical and cohesive groupings, called *packages*, based on some principle of organisation. For example, it is possible to partition a model based on the diagram type: via packages containing structural, behavioural, requirement and parametric diagrams, respectively. A package can contain another package, thereby enabling the modelling of containment hierarchies.

Use Case Diagram

Use case diagrams enable the modelling of high level behaviour, from the perspective of external entities, with the intent to convey the functionality that the system ought to support. These diagrams are behavioural in the sense that they allow us to capture function-based behaviour in terms of how the system is used by external entities.

Parametric Diagram

Parametric diagrams depict relationships and constraints amongst model elements. These diagrams allow for the specification of constraints in a rule-based fashion that relate properties of different model elements in a parametric fashion using mathematical equations. The constraints are normally formulated in terms of structural features, and, as such, we exclude parametric diagrams from our treatment.

2.2.3 Structural Diagrams

Structural diagrams enable the modelling of static aspects of a design, based on the principles of organisation, composition, classification and interconnection [**14**]. In addition, they provide the context in which behavioural diagrams execute. As such, we introduce them briefly below.

Block Definition Diagram

A *block definition diagram* depicts units of structure — in terms of *blocks* — and allows us to express relationships amongst these blocks by means of their composition and classification.

A block can be thought of as akin to a UML class. As such, it is used to conceptualise any abstract or concrete entities of a system into structural units. This logical abstraction defines the types of a SysML specification: a block is a *classifier* that describes the structural features of its instances. In addition, an optional classifier behaviour stipulates the behaviour of all instances of the block.

Block definition diagrams show the structural composition of a block. A block can be composed of other blocks in a *whole-part* relationship, termed a *composition*. Diagrammatically, the whole end of a composite association is adorned by a filled diamond. Alternatively, a block can reference another block in a *has-a* relationship, termed an *aggregation*. This is indicated with a hollow diamond adornment at the referring block's end.

The *structural features* of blocks can be classified thus:

- *part properties*, described by the composite association between a composite block and its parts;
- *reference properties*, described by the aggregate association when a block references other blocks; and
- *value properties*, like primitive types or enumerations, that describe unique quantitative features.

Blocks provide the context in which behaviours execute. The *classifier behaviour* is the main behaviour of a block, and executes from the instant the instance is created until the point of destruction. The behaviour of a block can be defined using any of the appropriate behavioural formalisms provided by SysML.

- The behaviour of event-driven, state-based blocks is most adequately specified using state machines.
- The behaviour of blocks where the emphasis is placed on control flow is best specified using activities.
- Message-based behaviour between parts or references of a block is specified using interactions.

The choice of which behavioural formalism to use is largely dependent on the purpose that the block serves, but to a lesser extent also depends on the aesthetic tastes of the modeller. In addition, it is possible to specify behaviour using a combination of formalisms; one approach is to use a state machine to describe the state-based behaviour of a block, and then to use activities that execute within a given state or on a given transition [**14**].

A block is often composed of other blocks, each of which has its own associated behaviour. In this case, there are two alternative interpretations with regards to the combined behaviour [**14**].

- The classifier behaviour of the block acts as a controller in order to actively orchestrate the behaviours of its parts. In this case, the behaviour of the block is a combination of its behaviour and that of its parts.

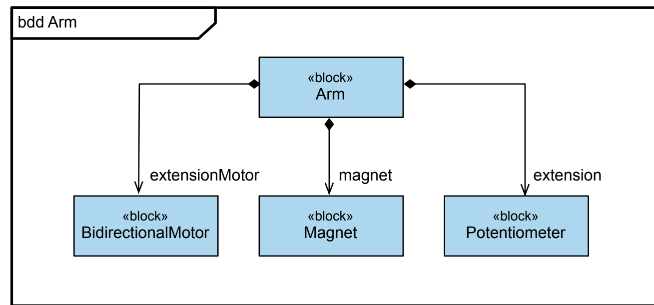


Figure 2.3: An example block definition diagram.

- Alternatively, the classifier behaviour of the block can serve as an abstraction of the behaviours of its parts. The abstraction serves as a specification that the parts must realise: the parts must interact in such a way that their combined behaviour conforms to the abstraction.

The *behavioural features* of blocks are operations and receptions. Operations can be triggered synchronously or asynchronously, whereas receptions can only be triggered asynchronously. Additionally, SysML defines the semantics of operations so that a request for an operation always triggers a response from the block; receptions, on the other hand, are only handled if the block explicitly accepts the request [**14**]. We exclude operations from our treatment. *Receptions* stipulate which stimuli a block responds to. Although these are behavioural features, they require no additionally defined behaviour[2]: the receipt of the stimuli alters the classifier behaviour of the owning block. Specialised constructs, such as an accept event action (for activities) or trigger event (for state machines), make explicit how the receipt of the stimuli alters the classifier behaviour.

A *signal* is a classifier that types the asynchronous messages that are communicated between blocks. Every reception is associated with a signal. Each signal optionally has an associated set of attributes which correspond to the parameters that make up the content of the message. A *signal event* is an instance of a signal.

A block indicates whether it provides a behavioural feature, or, alternatively, whether it requires a behavioural feature. *Provided behavioural features* are those behaviours provided by the block itself and made available as services to the external environment. Conversely, *required behavioural features* are behaviours that the block expects its external environment to provide or make available as services.

Blocks model classification hierarchies using the concept of *generalisation*. This relationship is not considered in our formalisation and will not be dealt with in any detail here.

Example 2.1. The arm of a crane consists of a bidirectional motor, a magnet and a potentiometer. We can use a block definition diagram, as per Figure 2.3 to depict this. The diagram shows that the Arm block consists of other blocks, namely BidirectionalMotor,

[2]Alternatively, in the case of operations, if a separate behavioural definition is required, the behavioural specification is termed a *method*. Methods only execute in response to the stimuli associated with the operation, and then terminates. In contrast to classifier behaviour, the lifetime of a method is limited. Methods are usually specified using activities [**14**].

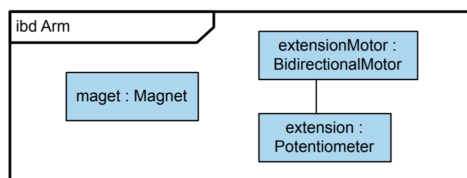


Figure 2.4: An example internal block diagram.

Magnet and Potentiometer.

□

Internal Block Definition Diagram

An *internal block definition diagram* models the internal structure of a block, and shows the interconnection and interfaces between constituent parts or references of a block. In contrast to the block definition diagram that conveys a sense of the composition of a block, the internal block diagram instead places the emphasis on the interconnection. Consequently, it provides an alternative visualisation of the composition of a block.

The diagram allows for the specification of interfaces amongst constituent blocks by means of connectors and ports.

A *connector* connects two or more parts or references. The connection formally allows the connected components to interact, although the connector does not characterise the nature of the interaction. Instead, the interaction is stipulated by the behaviours of the connected blocks. It is, however, possible to specify the behavioural characteristics of a connection between parts — using an *association block*. The association block can be assigned a classifier behaviour that would serve as a protocol specification for the connection; the behaviour of connected blocks should then conform to the interaction protocol designated by the connector.

The ends of a connector allow for the specification of multiplicities that describe the number of instances connected by the connector; the connector makes use of *links* to more precisely describe connections between instances. We exclude connectors with multiple links from our treatment. As such, we insist that there is a single instance present on both connector ends.

SysML allows us to model the flow of *items* on the connectors between parts or references of blocks. Item flows range from the abstract (for example, the flow of information or control) to the concrete (for example, electronic signals or physical items in a system). These items are typed either by blocks or signals. Typically, blocks are used when the items have significant internal structure, and signals are used to specify the flow of information or control. Diagrammatically, item flows are indicated on the relevant connector, with an arrow indicating the appropriate direction. We restrict flow items to signals for the purposes of this thesis.

Example 2.2. The internal structure of the Arm block and the interconnections between its constituent components is communicated using an internal block definition diagram, as per Figure 2.4.

□

2.2.4 Behavioural Diagrams

This section presents a brief overview of the behavioural diagrams relevant to our treatment. SysML defines several behavioural formalisms: opaque behaviours, function behaviours, use cases, state machines, activities, and interactions. *Opaque behaviours* are defined as expressions of a language which is external to SysML. *Function behaviours* are similar, but must be side-effect free: they are not permitted to alter the state of their owning block. We mentioned use cases in Section 2.2.2. We now expand upon activities, interactions and state machines.

State Machine Diagram

State machine diagrams allow for the specification of state-based behaviour. A *state machine* graphically depicts state-dependent behaviour in terms of nodes and labelled edges: nodes represent either simple or composite states, whereas the edges correspond to transitions between states. State machines exhibit the behaviour prescribed by their current state and respond accordingly to events — both those generated internally and externally — that fire the transitions between states.

In SysML, a state machine prescribes the state-based behaviour of an owning block and executes within the context of an instance of that block; thus, a *state* is an abstraction of the mode that the owning block finds itself in. A change of state is thus effected by the arrival of a triggering event, causing an appropriate transition to fire. A *transition* consists of a trigger, a guard and an effect component. The *trigger* denotes the event that serves as stimulus for the transition to fire; the *guard* is a conditional expression used to decide whether the transition is to fire at all; and the *effect* is a supplementary behaviour that executes on the transition.

We distinguish between pseudo states and non-pseudo states. A *pseudo state* is a transient point that exists to determine the next active state; as such, a state machine cannot stay indefinitely in a pseudo state. Examples of pseudo states are initial, final, junction and choice states. A *complex transition* is a compound transition composed of two separate transitional legs via a junction or choice state, where the entire compound transition can only be taken in response to a single triggering event.

A *region* defines its own set of states and these are exclusive in the sense that, when the region is active, only one state is active. Thus, a region characterises the state-related behaviour associated with a particular state. State hierarchies occur when a state contains its own regions. A *simple state* has no region, whereas a *composite state* consists of one or more regions. A state with just one region is the most common case and is termed a *simple composite state*. A state with more than one region is termed an *orthogonal composite state*.

The behaviour of any state machine is described by a single composite state at the top-level. *Non-hierarchical state machines* have a simple composite state at the top of the state hierarchy; likewise the top-level state of a *hierarchical state machine* is a composite state, but every state may contain its own region(s).

An *initial state* indicates the point at which execution starts when a region first becomes active. Conversely, a *final state* signifies the termination of a region. Initial and final states are shown as a filled circle and encircled filled circle, respectively. A *terminate state* forces the termination of the state machine as a whole. It is distinct from a final state

in that it brings the execution of the entire state machine to an end, whereas a final state is specifically linked to the termination of a particular region. *Junction* and *choice* states support the construction of complex transitions between states, with multiple guards and effects. Diagrammatically, junction and choice states are indicated using a solid circle and diamond shaped node. To model orthogonal composite states effectively, additional constructs are needed. *Fork* and *join* states are used to specify transitions into and out of orthogonal composite states, respectively. Fork and join nodes are graphically depicted as shaded bars.

State machines allow the definition of supplementary behaviours that are executed either on entry, while the state is active, or on exit. These behaviours are optional. An *entry behaviour* is executed upon entry; conversely, an *exit behaviour* is executed on exit, before the transition is taken. The *do behaviour* executes while the state is active and represents an interruptible behaviour. It starts executing after the entry behaviour has finished; it continues to execute either until completion, or until it is interrupted by an event[3], whichever comes first. All state-based behaviours are typically described via activities.

Another useful construct is that of a *submachine state*, a state that references another state machine. The behaviour of this state when active is that of the referenced state machine.

The execution semantics for SysML state machines are as follows. A *run to completion semantics* is defined: a state machine is only permitted to consume a single triggering event, namely the *current event*, at any one instant and must do so until processing of the said event is complete. For this purpose each state machine has an associated *event queue*.

- An event is *received* when it is accepted and waiting for processing. The event is placed at the back of the event queue.

- The event at the front of the queue is removed and presented to the state machine. At this instant it becomes the current event. We say the event is *dispatched*.

- The state machine finishes processing of the current event, uninterrupted, and until completion. Once this has happened, the event is *consumed*. A consumed event is no longer available for processing.

In the above we assume a FIFO queue, although the standard [**1**; **2**] does not define the order of dequeuing.

Figure 2.5 presents a diagrammatic overview of the structural constructs relevant to our formalisation.

In most systems engineering methodologies, state machines are used as the classifier behaviour of blocks. Two block instances communicate using signal events. The initiating block sends a signal event to a target block. This signal event is defined as part of the supplementary behaviours associated with the initiating state machine: the entry, exit, or do behaviours of the active state; or the effect component of the enabled transition. The receipt of the signal event in the target block may subsequently trigger a transition in its state machine. The approach described above is popular when modelling event-based systems.

[3]The interrupting event is an explicit or completion event that emanate from an enclosing state higher up in the state hierarchy.

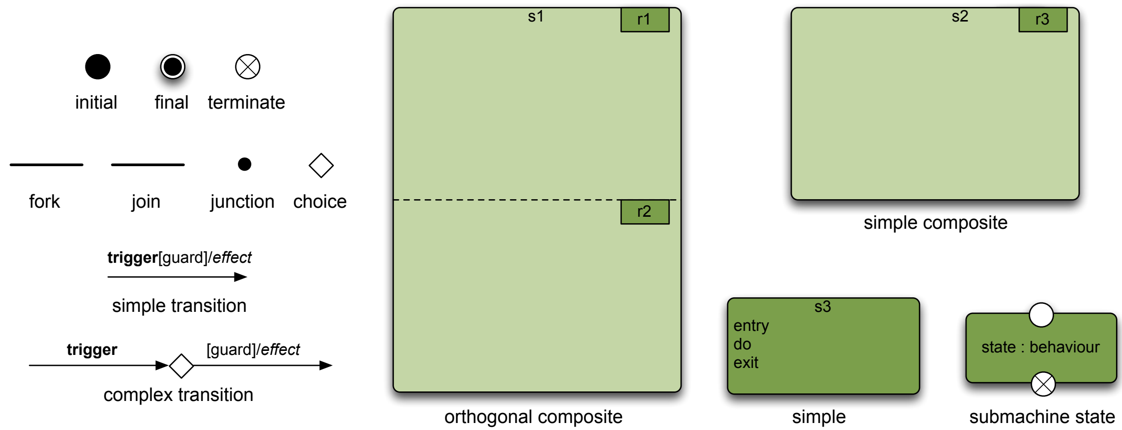


Figure 2.5: State machine notation relevant to our formalisation.

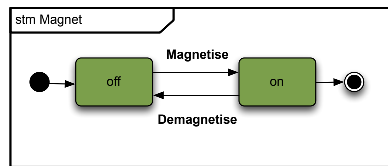


Figure 2.6: An example state machine diagram.

Example 2.3. The classifier behaviour of the block Magnet is modelled using the state machine Magnet, shown in Figure 2.6.

□

Activity Diagram

Activity diagrams provide constructs that allow the modeller to describe complex routes along which actions execute. These routes are termed *flows*. In SysML activities there are two types of flows: control flows and object flows.

Actions are the fundamental building blocks of *activities* and always execute within the context of an activity. Diagrammatically, actions are represented as rounded rectangles. An action accepts inputs and produces outputs. The flow of input and output items between actions are described using *object flows*. *Control flows*, on the other hand, impose additional constraints on the execution of actions. When a control flow connects one action to another, the target action cannot start until the source action has completed. *Control nodes* are used in the specification of control flow: they are used to impose control logic on the execution of actions. The control nodes are the fork, join, decision, merge, initial and final nodes.

A *fork node* is used to split a thread of behaviour into two independent threads of behaviour. Semantically, the control flow is interleaved into concurrently executing flows. Conversely, a *join node* synchronises concurrent behaviour into a single thread of execution. A *decision node* models conditional branching; a *merge node* is used to merge different branches into a single route. *Initial* and *activity final nodes* exhibit the activity starting and termination points, respectively. A *flow final node* models the termination

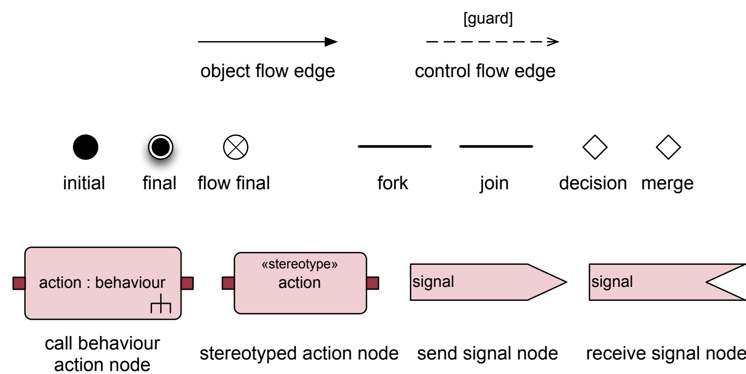


Figure 2.7: Activity notation relevant to our formalisation.

of a particular flow, without terminating the activity. Fork and join nodes are graphically depicted as shaded bars; decision and merge nodes are both represented with a diamond shaped node. An initial node is shown as a filled black circle; the activity final node symbol is shown as an encircled filled circle, and the flow final node as an encircled cross.

Several types of actions exist. The *send signal event action* sends a signal event; conversely, the *receive signal event action* waits on the receipt of a particular signal event. These are known as *primitive actions*. Other primitive actions are: the *value specification action*, which allows the specification of a particular value to an input of an action; and the *opaque action*, which executes an opaque behaviour. A *call behaviour action* invokes another behaviour when it executes; this behaviour can be any of the behavioural formalisms afforded by SysML. Typically the behaviour will be another activity.

Activity parameter nodes are placed on the boundaries of activities in order to convey information to the activity. Each parameter has a name and an associated type. For example, an activity that is initiated via a call behaviour action might be passed additional information at the point of invocation.

Figure 2.7 presents a diagrammatic overview of the structural constructs relevant to our formalisation.

In most systems engineering methodologies, activities are typically used as a complementary modelling notation to state machines: it is the behavioural formalism normally associated with the effect component of a transition; alternatively, it is used to model entry, exit and do behaviours related to a particular state. However, activities are also utilised in the description of operations associated with a block. Another use includes the specification of workflows.

Example 2.4. A state machine can use an activity to describe the behaviour of the effect component of a transition. The state machine of the Arm block uses the activity ActivateMagnet to turn the magnet on. The activity is shown in Figure 2.8.

□

Sequence Diagram

Message-based modelling is made possible via *sequence diagrams*. Behaviour is modelled as a temporal sequence of *occurrence observations* between structural constructs. The occurrence observations for a message exchange correspond to the sending and re-

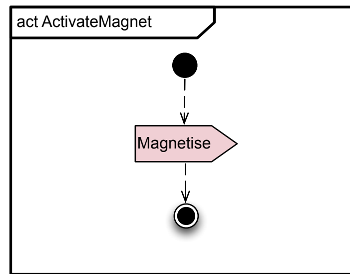


Figure 2.8: An example activity diagram.

ceiving of the message, respectively. Messages are exchanged either synchronously or asynchronously. The sender blocks until the arrival of a response if the communication is *synchronous*. Conversely, during an *asynchronous* exchange, the sender does not block, but, instead, continues execution after sending the message. We exclude synchronous message exchanges from our treatment.

In SysML, an *interaction* executes within the context of its owning block, and specifies the interaction between parts or references. A sequence diagram depicts this interaction graphically. On the diagram, *lifelines* correspond to the parts or references. Each lifeline is represented as a dashed line with the name of the reference or part enclosed in a rectangle. An *asynchronous message* is represented using a solid line from the sending lifeline to the receiving lifeline; there is no associated return message as the interaction does not block[4].

When an interaction executes, it produces a sequence of interaction occurrences, termed a *trace*.

Several *interaction operators* exist. An operator either restricts the behaviour of the prescribed sequence, or alters our interpretation of the trace. Operators that limit the behaviour are: the weak and strict sequencing operators; the parallel, break and critical operators; the alternative and optional operators; and the loop operator. Operators that alter our interpretation are the negative and the assert operators, and the consider and ignore operators.

Figure 2.9 presents a diagrammatic overview of the structural constructs relevant to our formalisation.

Interactions are used to depict scenarios of communication. A *scenario* is a description of an exchange taking place between instances of communicating blocks. Thus, interactions are not typically employed to record every conceivable exchange that might occur between communicating block instances, but rather to describe a particular exchange. However, the scenario depicted should still be a valid interaction: that is, legal behaviour in terms of the classifier behaviours of the individual instances.

Example 2.5. The interaction between the Arm and Magnet is modelled using a sequence diagram, as per Figure 2.10. The arm turns the magnet on by sending a signal. Note that the diagram depicts an instance of the Arm block called central. Similarly, magnet is an instance of block Magnet.

□

[4]A *synchronous message* is indicated using a solid line with a filled arrowhead from the sending lifeline to the receiving lifeline; the return message, unblocking the sender, is a dashed line with opposite direction.

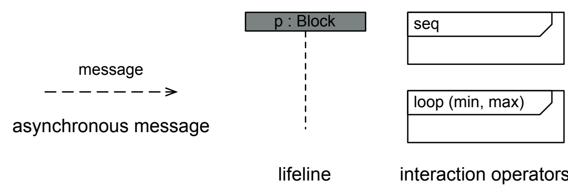


Figure 2.9: Interaction notation relevant to our formalisation.

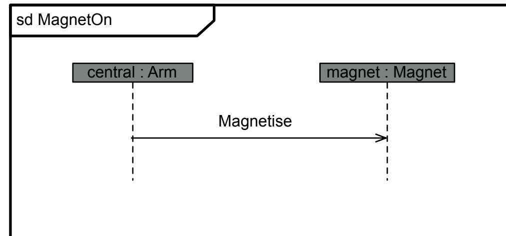


Figure 2.10: An example sequence diagram.

2.2.5 Requirement Diagram

The requirement diagram makes possible the modelling of requirements. In particular, it allows for linking requirements to design elements and test cases in order to support the traceability of requirements, one of the key objectives of SysML.

A *requirement* is a SysML-specific modelling construct represented explicitly in the syntax via the *requirement diagram*. Requirements, in their most basic form, are typically text-based, and allow for the description of conditions that must be satisfied by a particular system. Requirements can be related to other requirements and to modelling constructs via several relationships. We concern ourselves with the satisfy and refine relationships, which are defined as follows.

> *"The satisfy relationship describes how a design or implementation model satisfies one or more requirements"* [**1**]

The *satisfy* relationship is used to state that a particular model element meets the associated requirement. This is merely an assertion and not a proof of fact.

> *"The refine requirement relationship can be used to describe how a model element or set of elements can be used to further refine a requirement"* [**1**]

A text-based requirement can be captured in a SysML model, with the danger being that a textual description can often be ambiguous. Furthermore, such a description is in general and for obvious reasons not well-suited to automated reasoning. A more precise definition is possible in SysML, which allows any behavioural formalism — for example, a state machine — to be assigned to a requirement using the *refine* relationship. This results in a more formal representation of the corresponding desired behaviour.

Example 2.6. The requirement diagram is a fundamental part of the SysML language and enables linking requirements directly to the design. Figure 2.11 models the requirement

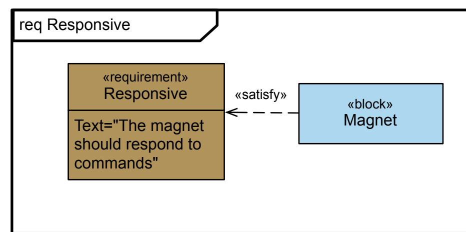


Figure 2.11: An example requirement diagram.

Responsive, which states that a magnet should respond to commands. The diagram additionally asserts that the block Magnet satisfies the said requirement.

□

2.2.6 Tool Support

Traditionally, modelling notations were used as part of a *document-based systems engineering approach*, where the requirements and system design are produced as documents and exchanged between the stakeholders. In contrast, a *model-based systems engineering approach* relies on computer-aided design tools to support the systems engineer in leveraging the model throughout the different life cycles of the system. For SysML, several such tools exist — hardly surprising given the fact that many of the tool vendors were the driving force behind the development and adaptation of SysML as a standard. Another factor attributed to the myriad of tools available is the prevalence of a more structured approach to systems engineering in industry.

In a model-based systems engineering approach, a model is more than a mere collection of diagrams. The set of diagrams do collectively make up the model, but there are dependencies and inherent relationships that interconnect the diagrams. These interconnections make for a more coherent and tightly bound description of the problem domain. For example, a state machine diagram should only permit the use of signal events that are defined elsewhere in the model. Thus, current tools are invaluable to help ensure more coherent designs. However, at present, these tools are incapable of detecting behavioural inconsistencies.

The modelling tool we use is MagicDraw[5] with the SysML plugin[6]. Examples of other tools are Visual Paradigm for UML[7] and PolarSys[8]. MagicDraw was selected due to its high level of conformance to the standard [**1**; **2**].

2.3 Communicating Sequential Processes

Process algebras provide a formal, mathematical framework for modelling and reasoning about concurrent systems. *Communicating Sequential Processes* (CSP) [**4**] is one such

[5] http://www.nomagic.com/products/magicdraw.html
[6] http://www.nomagic.com/products/magicdraw-addons/sysml-plugin.html
[7] http://www.visual-paradigm.com
[8] http://polarsys.org

notation for capturing patterns of behaviour and reasoning about concurrently interacting processes.

This section provides an introduction to the basic operators of the language, which can be used to define precisely the interactions of a process. Next, we introduce the concepts of specification and refinement. We conclude the section with an introduction to CSP's refinement checker, *Failures Divergences Refinement* (FDR) [**5**; **6**; **7**; **8**].

2.3.1 Patterns of Behaviour

This section is concerned with the formalisation of patterns of behaviour using the process algebra CSP. In the following, P and Q represent processes, e, f and g denote events, and X and Y represent sets of events.

Events and Channels

An *event* is an indivisible communication or interaction, fundamental to the synchronisation mechanism employed in CSP. We denote by Σ the set of all possible events in a particular CSP model.

A communication takes place when two or more processes agree on an event. The communication can either be a primitive event, or can take a more structured, message passing approach, utilising *channels*. The message passing mechanism is fundamentally based on the principle of a rendezvous between a sending and a receiving process: if the communication takes place on channel c, and a sending process wants to output a value e, the receiving process has to allow for this by inputting on c. Once this has happened, the event is abstracted as $c.e$. As such, we consider $c.e \in \Sigma$. A process indicates that it intends to output a value on a channel using the syntax $c!e$; similarly, the willingness to receive an input on a channel is expressed $c?x$. The aforementioned approach generalises for more than two processes.

In the untimed version of CSP, we focus on the order in which events occur: the intervals between successive events are abstracted away.

Alphabet

The *alphabet* of a process is the set of events that it is allowed to communicate. Thus, it is impossible for a process to communicate outside a given alphabet. The alphabet, on the other hand, may contain events not present in the process description. In this thesis we follow the approach of Roscoe [**15**; **17**] regarding alphabets, in that the alphabet of a process must be given explicitly, if required.

Example 2.7. A possible alphabet of a process modelling an electromagnet is the set of events $\{magnetise, demagnetise\}$. The process and its environment uses the event *magnetise* to agree on the act of turning on the electromagnet; conversely, the event *demagnetise* denotes the act of the electric current being turned off.

□

Basic Processes

A *process* is a basic pattern of communication that potentially interacts with other processes in order to form more complex expositions of interaction.

The simplest primitive process is *Stop*, the *deadlocked process*: it will refuse all events and never communicate.

Skip, another primitive process defined in terms of *Stop*, indicates successful termination by communicating the special event *tick*.

$$Skip = \checkmark \rightarrow Stop$$

The process $Run(X)$ is always willing to communicate any member $X \subseteq \Sigma$.

$$Run(X) = \Box\, e : X \bullet e \rightarrow Run(X)$$

$Chaos(X)$ is the process that can communicate or reject any member of $X \subseteq \Sigma$.

$$Chaos(X) = (\Box\, e : X \bullet e \rightarrow Chaos(X)) \sqcap Stop$$

Example 2.8. A process that forever offers the choice between the events *magnetise* and *demagnetise*, and is always willing to communicate either, follows.

$$RunMagnet = Run(\{magnetise, demagnetise\})$$

□

Prefixing

The *prefixing operator* allows us to model the behaviour of a process as a temporal sequence of events. For example, the process P below communicates the events e, f and g in that order before terminating successfully.

$$P = e \rightarrow f \rightarrow g \rightarrow Skip$$

Example 2.9. A process that turns the electromagnet on, followed by off, and then refuses to communicate forever more can be modelled thus.

$$OnThenOff = magnetise \rightarrow demagnetise \rightarrow Stop$$

□

Recursion

Recursion allows for the succinct definition of a process in terms of itself — we can capture infinite behaviour using a finite description. In order for recursions to be useful, they need to be guarded. In a guarded recursion, any recursive call is always proceeded by an event. Thus, the recursion is constructive in the sense that we are able to make progress as the recursion unfolds. As an example, we define, using recursion, a process P

that communicates the event e indefinitely.

$$P = e \rightarrow P$$

Example 2.10. Two or more processes that are *mutually recursive* are defined in terms of one another. We can define, using mutual recursion, a pair of processes that alternate between turning the electromagnet on and off.

$$\begin{aligned}&\mathit{ForeverOnOff} = \mathit{On}\\&\mathit{On} = \mathit{demagnetise} \rightarrow \mathit{Off}\\&\mathit{Off} = \mathit{magnetise} \rightarrow \mathit{On}\end{aligned}$$

ForeverOnOff differs from *RunMagnet* in that it will insist on alternating between events *demagnetise* and *magnetise*, whereas *RunMagnet* is happy to communicate either at any time.

□

Choice Operators

CSP provides two choice operators: the *external* or *deterministic choice operator* offers the environment the choice between the initial events of its argument processes; conversely, the *internal* or *nondeterministic choice operator* offers no such choice and the observed behaviour may be that of either process. We write $P \Box Q$ to denote the deterministic choice between P and Q; similarly, $P \sqcap Q$ denotes the nondeterministic choice between the initial events of P, and those of Q.

Both forms of choice have an associated replicated form.

- $\Box\, i : I \bullet P(i)$ is an external choice between processes $P(i)$, where i serves as an index for the parameterised process P.
- $\sqcap\, i : I \bullet P(i)$ is the corresponding internal choice.

Input along a channel can also be expressed using external choice: for example, we can write $\Box\, x : X \bullet c.x \rightarrow P$ to model input along channel c.

Example 2.11. The process *NDElectromagnet* models an electromagnet that internalises the choice as to whether it activates the magnetic field. Such a process might be used to model a broken magnet[9].

$$\begin{aligned}&\mathit{NDElectromagnet} =\\&\quad \mathit{magnetise} \rightarrow \mathit{NDElectromagnet}\\&\quad \sqcap\\&\quad \mathit{demagnetise} \rightarrow \mathit{NDElectromagnet}\end{aligned}$$

□

Example 2.12. We can however model a magnet that responds to all commands issued to it. The process *DElectromagnet* models an electromagnet that offers the environment the

[9]Given the fact that a magnet is an actuator that ought to respond to the commands issued to it.

choice as to whether it activates the magnetic field: the process is happy to communicate either event.

$$
\begin{array}{l}
DElectromagnet = \\
\quad magnetise \rightarrow DElectromagnet \\
\quad \Box \\
\quad demagnetise \rightarrow DElectromagnet
\end{array}
$$

$\Box$

Parallel Operators

CSP is concerned with describing patterns of behaviour. We are particularly interested in how multiple processes interact and combine to form new process descriptions. The constructs we have seen so far enable us to accurately define the independent behaviour of a process; the operators introduced next allow for the parallel composition of different threads of behaviour. Several parallel operators exist, each imposing its own synchronisation semantics on its operands.

The process $P \,[|\, X \,|]\, Q$ uses the *generalised parallel operator* to define an interface, X, on which P and Q must synchronise. Events outside X may occur independently in either process.

$P\,[X \,\|\, Y]\,Q$ denotes *alphabetised parallel*, where synchronisation takes place on events in the set $X \cap Y$. P can only communicate events in its alphabet, X; similarly, Q cannot communicate outside Y.

The *interleaving operator* expresses the unsynchronised concurrent interleaving of the events of its argument processes. We write $P \;|||\; Q$. The behaviour of P and Q are completely independent, in that any event communicated by the combination must stem from precisely one of P or Q. In the eventuality where both are able to communicate an event, the choice is resolved nondeterministically.

Replicated forms exist for all parallel operators.

- $[|\, X \,|]\; i : I \bullet P(i)$ denotes the generalised parallel composition of processes $P(i)$. In the composition, X corresponds to the synchronisation interface of the constituent processes.

- $\|\; i : I \bullet [X(i)]P(i)$ is the alphabetised parallel composition of processes $P(i)$ with corresponding alphabets $X(i)$.

- $|||\; i : I \bullet P(i)$ models the interleaving of processes $P(i)$.

Example 2.13. We can describe the behaviour of a broken magnet that interacts with an environmental process that indefinitely turns the magnet on and off.

$$
\begin{array}{l}
MagnetEnvironment = \\
\quad NDElectromagnet \;[|\; \{magnetise, demagnetise\} \;|]\; ForeverOnOff
\end{array}
$$

This process is bound to deadlock. *NDElectromagnet* can nondeterministically select an event and refuse to communicate any other event. *ForeverOnOff* insists on communicating the synchronisation events in a particular order. Deadlock occurs when both insist on

communicating different events.

□

Example 2.14. The designers of the crane's arm could conceivably fit it with more than one magnet. The intention is for the magnets to behave independently. This can be modelled with the interleaving operator.

$$TwoMagnets = DElectromagnet \mathbin{|||} DElectromagnet$$

This configuration would not allow the designers to determine which magnet responded to a particular command.

□

Example 2.15. An alternative design comprises two magnets with the intention that they provide a more dependable approach: the magnets will only be activated (or deactivated) if both are able to respond to the command issued. This can be modelled with the alphabetised parallel operator.

$$\begin{array}{l} TwoMagnets' = \\ \quad DElectromagnet \\ \quad\quad [\{magnetise, demagnetise\} \parallel \{magnetise, demagnetise\}] \\ \quad DElectromagnet \end{array}$$

In this configuration both magnets have to agree on every command issued to the combination. Therefore, immediately after a *magnetise* (or *demagnetise*) command was issued we can be sure that both magnets will be activated (or deactivated).

□

Example 2.16. A competitor manufactures an arm that requires both magnets to cooperate in order to be magnetised, but allows the magnets to be independently demagnetised.

$$TwoMagnets'' = DElectromagnet \; [| \; \{magnetise\} \; |] \; DElectromagnet$$

It remains impossible to determine which magnet was turned off.

□

Sequential Composition

The process $P \fatsemi Q$ represents the *sequential composition* of P and Q. This process behaves as P until it terminates, after which it behaves as Q.

An associated replicated form exists: $\fatsemi\ s : S \bullet P(s)$ denotes the sequential composition of parameterised processes $P(s)$. The order of the sequential composition is that imposed by the sequence S.

Example 2.17. A magnet that performs an initialisation sequence and subsequently behaves as either a broken or a functioning magnet is modelled thus.

$$\begin{array}{l} Startup = init.1 \rightarrow init.2 \rightarrow Skip \\ Magnet = Startup \fatsemi (DElectromagnet \sqcap NDElectromagnet) \end{array}$$

□

Hiding

The *hiding operator* conceals the events of X from the view of the external environment of P. Hiding enables a process to have events that are neither controlled by nor visible to the environment. This is a powerful abstraction mechanism. Consider a process, composed of several parallel component processes, synchronising on a certain event e. If this process is itself a component of another parallel composition, then, without hiding, other processes would be able to prevent e from happening. This might be undesirable. By making use of the hiding operator we can conceal the event and avoid this situation. We write $P \setminus X$ to hide the set of events X. All hidden events are replaced by τ actions — invisible events not seen or controlled by the environment. A process cannot remain indefinitely in a given state when τ actions are present: either the τ action or a visible event must eventually happen. However, because τ actions are neither controlled by nor visible to the environment, they are perfect for modelling nondeterminism. It is well known that hiding introduces nondeterminism [**15**; **17**].

Example 2.18. The hiding operator can be used to conceal the initialisation sequence of the magnet.

$$ConcealedMagnet = Magnet \setminus \{init.1, init.2\}$$

□

Event Renaming

The *renaming operator* applies a mapping function to the alphabet of a process P. This transformation changes the events observed by the external environment of P to that dictated by the mapping.

The construction $P[e \leftarrow f]$ denotes the process that behaves exactly as P, but with all instances of the event e renamed to f.

Example 2.19. A magnet that performs the initialisation sequence in reverse order is described by the following exposition.

$$ReverseMagnet = Magnet[init.1 \leftarrow init.2, init.2 \leftarrow init.1]$$

□

Interrupt

The *interrupt operator* allows an interrupting event to discontinue the behaviour of a process P in favour of another process Q. Consider the construct $P \triangle (e \rightarrow Q)$. The combination behaves as P until the occurrence of event e. Thereafter, it behaves as Q.

Example 2.20. Due to the fact that a magnet can potentially malfunction we might want to add a failsafe as follows.

$$FailsafeMagnet = Magnet \triangle (reset \rightarrow FailsafeMagnet)$$

□

Timeout

The *timeout operator* models the *asymmetric choice* between the events of its operands. Given $P \triangleright Q$, we may be able to communicate the initial events of P, but the initial events of Q must eventually be offered. We have

$$P \triangleright Q \Leftrightarrow (P \sqcap Stop) \,\square\, Q$$

Example 2.21. A magnet that offers an asymmetric choice between the modes of operation is described thus.

$$AsymmetricMagnet = (failsafe.1 \rightarrow FailsafeMagnet) \triangleright (failsafe.0 \rightarrow Magnet)$$

We can view the above composition in the following manner. If the environment is quick enough it may be able to activate the failsafe as the event $failsafe.1$ is offered for a short time. The event $failsafe.0$, however, must be offered if we wait long enough.

□

2.3.2 Specification and Refinement

In this section we focus on how we can compare the behaviour of one process to that of another. We start by considering the various ways in which we can capture process behaviour and then introduce the concept of refinement.

Traces

Consider a set of all the possible finite sequences of events which a process P can communicate, written $traces\,[\![\, P \,]\!]$. An element of this set is termed a *trace*. Traces are invaluable when we set out to formalise behavioural conduct: they prescribe precisely all the possible communications of a process.

Below we present the traces of some basic processes to demonstrate the concept. We take $\Sigma = \{e, f, g\}$.

$$\begin{aligned}
&traces\,[\![\, Stop \,]\!] = \{\langle\rangle\} \\
&traces\,[\![\, e \rightarrow f \rightarrow Stop \,]\!] = \{\langle\rangle, \langle e\rangle, \langle e, f\rangle\} \\
&traces\,[\![\, P = e \rightarrow P \,]\!] = \{\langle\rangle\} \cup \{n : \mathbb{N}_1 \bullet \langle e\rangle^n\} \\
&traces\,[\![\, e \rightarrow Stop \,\square\, f \rightarrow Stop \,]\!] = \{\langle\rangle, \langle e\rangle, \langle f\rangle\} \\
&traces\,[\![\, e \rightarrow Stop \sqcap f \rightarrow Stop \,]\!] = \{\langle\rangle, \langle e\rangle, \langle f\rangle\} \\
&traces\,[\![e \rightarrow f \rightarrow Stop \,[|\, \{f\} \,|]\, f \rightarrow g \rightarrow Stop]\!] = \{\langle\rangle, \langle e\rangle, \langle e, f\rangle, \langle e, f, g\rangle\}
\end{aligned}$$

Example 2.22. The traces of the processes *NDElectroMagnet* and *DElectroMagnet*, defined in Examples 2.11 and 2.12, follow.

$$\begin{aligned}
&traces\,[\![\, NDElectroMagnet \,]\!] = \{magnetise, demagnetise\}^* \\
&traces\,[\![\, DElectroMagnet \,]\!] = \{magnetise, demagnetise\}^*
\end{aligned}$$

The expression X^* denotes the set containing all finite subsequences (including the empty sequence) consisting of elements drawn from the set X. The aforementioned processes have the same set of traces, even though they clearly have different behaviours. It follows that traces on their own are not sufficient to fully describe process behaviour.

□

Failures

Traces depict only the actions that a process is able to perform, but say nothing about the actions it can refuse. Therefore, in order to get a broader description of process behaviour, we also need to encompass the refusals of a process, written $\mathit{refusals}\,[\![P]\!]$. In the following, P/t represents the process P after the trace t. A refusal set is a set of events that a process can fail to accept anything from however long it is offered. We denote by $\mathit{refusals}\,[\![\,P\,]\!]$ the set of P's initial refusal sets. The set $\mathit{failures}\,[\![\,P\,]\!]$ consists of pairs of the form (t, X) such that, for a trace $t \in \mathit{traces}\,[\![\,P\,]\!]$, $X \in \mathit{refusals}\,[\![\,P/t\,]\!]$. An element of this set is termed a *failure*.

Below we present the failures of some basic processes to demonstrate the concept, assuming $\Sigma = \{e, f, g\}$.

$$
\begin{array}{l}
\mathit{failures}\,[\![\,\mathit{Stop}\,]\!] = \\
\quad \{X : \mathbb{P}\,\Sigma \bullet (\langle\rangle, X)\} \\
\mathit{failures}\,[\![\,e \rightarrow f \rightarrow \mathit{Stop}\,]\!] = \\
\quad \{X : \mathbb{P}(\Sigma \setminus \{e\}) \bullet (\langle\rangle, X)\} \\
\quad \cup \\
\quad \{X : \mathbb{P}(\Sigma \setminus \{f\}) \bullet (\langle e\rangle, X)\} \\
\quad \cup \\
\quad \{X : \mathbb{P}\,\Sigma \bullet (\langle e, f\rangle, X)\} \\
\mathit{failures}\,[\![\,P = e \rightarrow P\,]\!] = \\
\quad \{X : \mathbb{P}(\Sigma \setminus \{e\}) \bullet (\langle\rangle, X)\} \\
\quad \cup \\
\quad \{X : \mathbb{P}(\Sigma \setminus \{e\});\ n : \mathbb{N}_1 \bullet (\langle e\rangle^n, X)\} \\
\mathit{failures}\,[\![\,e \rightarrow \mathit{Stop} \,\Box\, f \rightarrow \mathit{Stop}\,]\!] = \\
\quad \{X : \mathbb{P}(\Sigma \setminus \{e, f\}) \bullet (\langle\rangle, X)\} \\
\quad \cup \\
\quad \{X : \mathbb{P}\,\Sigma \bullet (\langle e\rangle, X)\} \\
\quad \cup \\
\quad \{X : \mathbb{P}\,\Sigma \bullet (\langle f\rangle, X)\} \\
\mathit{failures}\,[\![\,e \rightarrow \mathit{Stop} \sqcap f \rightarrow \mathit{Stop}\,]\!] = \\
\quad \{X : \mathbb{P}(\Sigma \setminus \{e\}) \bullet (\langle\rangle, X)\} \\
\quad \cup \\
\quad \{X : \mathbb{P}(\Sigma \setminus \{f\}) \bullet (\langle\rangle, X)\} \\
\quad \cup \\
\quad \{X : \mathbb{P}\,\Sigma \bullet (\langle e\rangle, X)\} \\
\quad \cup \\
\quad \{X : \mathbb{P}\,\Sigma \bullet (\langle f\rangle, X)\}
\end{array}
$$

$$
\begin{array}{l}
failures\,[\![e \rightarrow f \rightarrow Stop \,[|\, \{f\} \,|]\, f \rightarrow g \rightarrow Stop]\!] = \\
\quad \{X : \mathbb{P}(\Sigma \setminus \{e\}) \bullet (\langle\rangle, X)\} \\
\quad \cup \\
\quad \{X : \mathbb{P}(\Sigma \setminus \{f\}) \bullet (\langle e\rangle, X)\} \\
\quad \cup \\
\quad \{X : \mathbb{P}(\Sigma \setminus \{g\}) \bullet (\langle e, f\rangle, X)\} \\
\quad \cup \\
\quad \{X : \mathbb{P}\,\Sigma \bullet (\langle e, f, g\rangle, X)\}
\end{array}
$$

Example 2.23. The failures of the process *OnThenOff*, defined in Example 2.9, follow.

$$
\begin{array}{l}
failures\,[\![\, OnThenOff \,]\!] = \\
\quad \{(\langle\rangle, \{demagnetise, reset\})\} \\
\quad \cup \\
\quad \{(\langle magnetise\rangle, \{magnetise, reset\})\} \\
\quad \cup \\
\quad \{(\langle magnetise, demagnetise\rangle, \{magnetise, demagnetise, reset\})\}
\end{array}
$$

In the above we assume $\Sigma = \{magnetise, demagnetise, reset\}$.

□

Refinement

CSP and FDR allow us to compare the behaviour of one process against that of another. Informally, a process Q is said to be a refinement of another process P if and only if Q cannot behave in a manner forbidden by P. Therefore, the possible behaviours of Q must be strictly contained within those of P. We write $P \sqsubseteq_M Q$, where $M \in \{T, F\}$ represents the notion of refinement in terms of one of the denotational semantic models. These notions of refinement are expressed in terms of the observable behaviours of the processes we want to compare: that is, in terms of traces and failures.

We define *traces refinement*, using reverse containment, as

$$P \sqsubseteq_T Q \Leftrightarrow traces\,[\![\, Q \,]\!] \subseteq traces\,[\![\, P \,]\!]$$

Similarly, we define *failures refinement* as

$$
\begin{array}{l}
P \sqsubseteq_F Q \Leftrightarrow \\
\quad traces\,[\![\, Q \,]\!] \subseteq traces\,[\![\, P \,]\!] \wedge failures\,[\![\, Q \,]\!] \subseteq failures\,[\![\, P \,]\!]
\end{array}
$$

It should be clear that the aforementioned models are structured hierarchically:

$$P \sqsubseteq_F Q \Rightarrow P \sqsubseteq_T Q$$

Example 2.24. Based on the above semantic models of refinement we can assert the following.

$$
\begin{array}{l}
NDElectromagnet \sqsubseteq_T DElectromagnet \\
DElectromagnet \sqsubseteq_T NDElectromagnet \\
NDElectromagnet \sqsubseteq_F DElectromagnet
\end{array}
$$

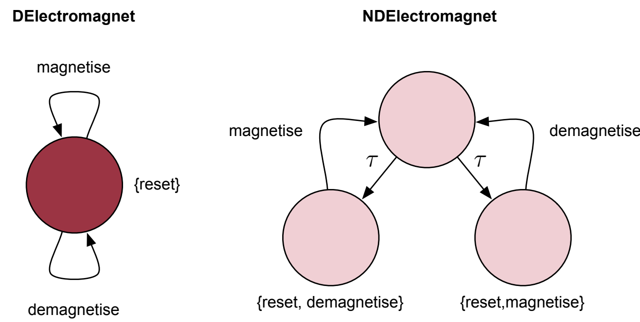


Figure 2.12: State transition graphs of *DElectromagnet* and *NDElectromagnet*. Refusal sets are indicated next to each state.

$$DElectromagnet \not\sqsubseteq_F NDElectromagnet$$

□

2.3.3 Tool Support

One of the prerequisites for the widespread acceptance and uptake of any formal method is the availability of reliable, mechanised tool support that facilitates the automated analysis of the formalism. FDR, one such tool, is a refinement checker that allows us to compare finite state CSP processes in terms of the denotational semantic models introduced above. In a wider context this is known as model checking [**25**].

The central idea behind model checking is to develop an adequate abstraction or model of a system. This abstraction describes the behaviour of the system at a level of detail sufficient to express and then verify certain properties. A *model checker* is a program that automatically decides whether a property holds for a given model, by enumerating all its relevant states. If the property is not satisfied in a particular state, a *counterexample* provides details of the execution path that led to the violation. For most model checkers, properties are expressed in a *temporal logic*, a language entirely different to that used for describing system behaviour. FDR, is a *refinement checker*, where the model, termed the *implementation*, and the property, termed the *specification*, are both written in the same language. The language, for the purposes of automated verification, is the machine dialect of CSP, CSP_M [**15**; **17**]. Thus, both the specification and implementation are expressed as CSP processes. The process modelling the behaviour of the property we intend to verify — that is, the specification — is termed the *characteristic process*.

An example of another tool is the *Process Analysis Toolkit* (PAT) [**26**], which supports *linear temporal logic* (LTL) [**25**] model checking, as well as refinement checking a CSP-like language.

Example 2.25. In order to perform the refinement check

$$DElectromagnet \sqsubseteq_F NDElectromagnet$$

tools like FDR require an effective mechanism to ensure that the behaviour of the implementation process, *NDElectromagnet*, is contained within that of the specification process, *DElectromagnet*. FDR constructs *state transition graphs* which mimic the opera-

tional semantics of each process in order to carry out the refinement check. Figure 2.12 shows the state transition graphs of *DElectromagnet* and *NDElectromagnet*. Refusal sets are indicated next to each state. These state transition graphs are then compared against each other in order to explore the entire state space. FDR checks the refinement by identifying for any given trace that *NDElectromagnet* is able to perform the equivalent alternate state of *DElectromagnet*. Furthermore, let q_i denote the state reached by *NDElectromagnet*, and q_s the equivalent state of *DElectromagnet*. The states are compared according to the semantic model selected. For example, in the failures model:

- every transition emanating from q_i should also be possible for q_s (corresponding to the condition on traces); and
- provided q_i has no τ actions, the refusal sets of q_i should be possible refusal sets of q_s (corresponding to the condition on failures).

By comparing the graphs in Figure 2.12 we can see that the trace refinement holds:

$$DElectromagnet \sqsubseteq_T NDElectromagnet$$

However, the failures refinement in the same direction does not hold: on performing the trace $\langle\rangle$, *NDElectromagnet* refuses the set of events $\{reset, magnetise\}$, which is not a refusal set of *DElectromagnet*. FDR will produce a counterexample and conclude that

$$DElectromagnet \not\sqsubseteq_F NDElectromagnet$$

□

We acknowledge that the above example is an oversimplification of the refinement checking process. For example, in a practical setting the state transition graph of the specification process is transformed into a deterministic graph[10] prior to performing the refinement check. In our illustrative example the specification is already deterministic. However, it suffices for our purposes.

2.4 Conclusion

In this chapter we introduced our running case study and provided the background to the modelling notations required in the remainder of this thesis.

Having given consideration to both SysML and CSP, we present in the next chapter the results of a literature survey pertaining to research contributions that combine the aforementioned approaches.

[10]A procedure that preserves the operational semantics of the original graph.

3

LITERATURE REVIEW

This chapter is the result of a systematic literature review of related work. There is a wealth of literature on the formalisation of UML diagrams. We provide a comprehensive and thorough analysis of literature that applies mechanised verification techniques, in particular model checking[1], to behavioural UML[2].

We start with a brief introduction of related surveys. The methodology undertaken in order to perform the systematic review is described next. We then pose the detailed research questions we intend to answer through this investigation. The results of the systematic literature review are presented in the next section. The chapter concludes with a detailed discussion of those approaches most closely related to our own.

3.1 Related Surveys

The literature review was carried out in accordance with guidelines provided by Kitchenham *et al.* [**27**]. In [**27**] a systematic literature review is defined thus:

> *"A form of secondary study that uses a well-defined methodology to identify, analyse and interpret all available evidence related to a specific research question in a way that is unbiased and (to a degree) repeatable"*

Several literature reviews have been written on the subject of UML. Related references of note include the works of Genero *et al.* [**28**], Usman *et al.* [**29**] and Lucas *et al.* [**30**]. Genero *et al.* published a review on the quality of UML models. A total of 266 papers were analysed and classified according to criteria such as the research goal and the types of diagrams considered. Usman *et al.* analysed 20 papers to present a survey on techniques to establish consistency in UML. Papers are categorised, amongst others, according to the diagrams considered, the type of consistency according to the classification scheme defined by the authors, and the methodology used to support the approach. Lucas *et al.* analysed 42 papers as part of another survey on consistency management techniques for

[1]We include in this refinement checking.

[2]We include in this behavioural SysML.

UML. However, the research questions investigated are slightly different than those posed in [**29**]. The authors present a detailed rationale as to why the chosen papers were selected. Approaches are classified according to the diagrams analysed, the type of consistency, tool support, whether the approach is automated, etc. Lucas *et al.* conclude that inter-model consistency is less frequently studied. Our review is different from the aforementioned surveys in that we are not exclusively concerned with consistency management as in [**29**] and [**30**], but we are wholly interested in model checking approaches. Moreover, we are only concerned with behavioural UML.

Throughout this chapter we interpret the term *"UML behavioural formalisms"* to encompass the relevant behavioural formalisms of UML, UML-related dialects or profiles, and SysML. We restrict our focus to state machines, activities, and interactions. However, we include in our analyses works where the aforementioned formalisms are analysed in combination with other diagrams. The review is carried out with a view to identify research areas not previously investigated, and will subsequently enable us to justify the novelty of our research proposition within the appropriate context.

3.2 Methodology

This section provides a detailed account of the approach undertaken during this review. We list the sources consulted, the search terms employed, and inclusion criteria used that resulted in the final papers selected for analysis.

Sources

The sources we consulted during our investigation are the following.

- IEEE Explore[3].
- ACM Digital Library[4].
- Google Scholar[5].
- ScienceDirect[6].
- SpringerLink[7].

Kitchenham *et al.* [**27**] list a number of sources that would result in a relatively thorough and complete search within the area of Software Engineering. The list above is based partially on those recommendations.

[3] `http://ieeexplore.ieee.org`
[4] `http://dl.acm.org`
[5] `http://scholar.google.com`
[6] `http://www.sciencedirect.com`
[7] `http://link.springer.com`

Search Terms

The search terms used to identify potential candidate papers are best described by considering the production rules of the following grammar.

search:	**[variant] [diagram] [tool] [general]**
variant:	"UML" | "UML-RT" | "SysML"
diagram:	"state machine diagrams" | "state charts" | "state machines" | "activity diagrams" | "flow charts" | "activities" | "sequence diagrams" | "message sequence charts" | "interactions"
tool:	"FDR" | "PAT" | "ProB" | "CWB" | "MWB" | "SMV" | "NuSMV" | "SPIN" | "UPPAAL"
general:	"process algebra" | "behavioural consistency" | "formal methods" | "refinement checking" | "model checking" | "CSP" | "CTL" | "LTL"

Corpus of Papers

Executing the searches above yielded roughly 155 candidate papers. In order to obtain a transitive closure of sorts, the citations of the initial papers were analysed and included in the initial corpus of papers if we deemed it to be relevant. After the removal of duplicate papers we were left with roughly 195 potential papers.

Inclusion Criteria

The formalisation of UML[8] diagrams encompasses a great body of work. In order to draw a meaningful conclusion, we need to limit the scope of the review. As such, we define an inclusion criteria: candidate papers need to conform to the criteria in order to be included in the review. The abstracts of the initial 195 papers were considered in order to determine their relevance. The inclusion criteria can be stated thus.

- Approaches need to be related to the behavioural formalisms of interest. Particularly, we are interested in state machines, activities and interactions of UML, UML-related dialects, and SysML.
- Variations on the considered behavioural formalisms, such as state charts with Harel semantics [**31**], are not included in the review.
- We include works that consider the behavioural formalisms in combination with other diagrams, such as class diagrams.
- We exclude approaches not related to the application of model checking. We further restrict our focus to general purpose mechanised support: we do not consider domain-specific or specialised model checkers and only allow those model checkers as per the grammar introduced above.
- Finally, the appropriateness of the candidate paper in terms of the research questions posed is also taken under consideration.

[8] This includes diagrams found in UML-related dialects and SysML, as per our broad interpretation of UML formalisms.

Selected Papers

We are aware of the fact that it is impossible to achieve complete coverage of all literature during any review. This is even more true in the case of UML by virtue of the fact that this is a particularly active area of research. However, our focus here is to present the most relevant and most authoritative papers on the topic of model checking the behavioural formalisms of interest: state machines, activities, and interactions. As such, we have identified 42 papers to consider in our detailed review.

3.3 Detailed Research Questions

This section poses the detailed research questions we intend to answer in this chapter. In order to evaluate and interpret the available research in a consistent manner, we need to establish an exact terminology. Therefore, we provide the relevant definitions first before introducing the research questions.

3.3.1 Terminology

Terms employed to classify the different notions of consistency are not always applied in a consistent manner throughout the literature: individual authors propose different classifications, each with their own definitions of consistency. As such, we provide a set of definitions as interpreted for the purposes of this review.

The following notions of consistency, in terms of *comparative views*, are used to classify the selected approaches.

- *Intra-model consistency*: inconsistencies that arise in a model at the same level of abstraction. These primarily stem from contradictory modelling artefacts, introduced as a result of the multi-view nature of UML.

- *Inter-model consistency*: inconsistencies that arise between models at different levels of abstraction. On the one hand, there is the initial, abstract design; on the other, there is the detailed refinement that is closer to the implementation. In between these we find successive evolutions that add more detail at each level. Inter-model consistency is concerned with maintaining consistency between the different levels of abstraction.

The following notions of consistency, in terms of *type*, can be defined.

- *Semantic consistency*: consistency where the emphasis is placed on the behavioural or dynamic aspects.

- *Syntactic consistency*: consistency where emphasis is placed on structural or static aspects.

Semantic consistency can be broken down further.

- *Intra-behavioural consistency*: consistency among the same behavioural formalisms.

- *Inter-behavioural consistency*: consistency between different behavioural formalisms.

We use the term *structural-behavioural consistency* to denote the checking of consistency between structural and behavioural formalisms.

3.3.2 Research Questions

We attempt to answer the following research questions with this review. For each question, we also state the necessary assumptions made; these should be taken into consideration when interpreting the results.

Variant

> *"Which version of UML, UML-related dialect or SysML is considered?"*

The version number reported is based on the reference section of the paper under review. If no version number is provided, a best-guess is made by considering the time of publication.

Diagrams

> *"Which diagrams are considered?"*

We only report on the behavioural diagrams considered as part of the review. If additional diagrams are treated as part of the approach, we mention this as a comment.

Purpose

> *"What is the purpose of the approach?"*

Here, we distinguish between *checking consistency* and *verification of desirable properties*. Typically, we would expect that, if a formal model is created for the purpose of checking consistency, it should be possible to test for desirable properties. The assumption here is that the model is at a suitable level of abstraction. We therefore adopt the following approach: if the main focus is the checking of model consistency, it is classified thus; otherwise, we classify it as the verification of desirable properties.

If the purpose is the verification of desirable properties, we additionally classify these as follows:

- *liveness assertions*, such as freedom from deadlock or livelock; and
- *safety assertions*, such as the verification of a desirable property (expressed in a temporal logic or as a characteristic process).

We further divide consistency checking in terms of the type of consistency checking addressed:

- intra-model or inter-model consistency;
- intra-behavioural or inter-behavioural consistency; and
- structural-behavioural consistency.

Tool

"Which model checking tool is used as part of the approach?"

We need to limit the scope here, as a myriad of such tools exist. As such, we only consider a limited number of model checkers. These are selected based on popularity and relevance to our own work. In addition, we attempt to include a broad range of such tools: we consider, amongst others, explicit *linear temporal logic* (LTL) [**25**] model checkers, symbolic LTL and *computation tree logic* (CTL) [**25**] model checkers, and a process algebraic refinement checker. In addition, we focus and include all approaches related to refinement checking CSP, whereas we include only a representative subset of approaches for the remaining tools.

The tools considered are the following.

- *Failures Divergences Refinement* (FDR) [**5**; **6**; **7**; **8**] is a refinement checker for the process algebra Communicating Sequential Processes (CSP) [**4**].
- The *Process Analysis Toolkit* (PAT) [**26**] is a simulator, model checker and refinement checker for real time and concurrent systems. It supports, amongst others, refinement checking and LTL model checking for a CSP-like language: classic CSP is augmented with programming constructs such as conditionals, arrays, and loops.
- ProB [**32**] is a model checker for the B-method [**33**] that supports CSP refinement checking.
- The *Edinburgh Concurrency Workbench* (CWB) [**34**] is a model checker for the process algebra CCS [**35**].
- The *Edinburgh Mobility Workbench* (MWB) [**36**] is a model checker for the π-calculus [**16**].
- *Symbolic Model Verifier* (SMV) [**37**] is a symbolic model checker that supports CTL model checking.
- *New Symbolic Model Verifier* (NuSMV) [**38**] is a reimplementation of SMV that supports CTL and LTL model checking.
- SPIN [**39**] is an explicit state model checker that allows LTL model checking. The input language of the model checker is PROMELA [**39**].
- UPPAAL [**40**] is a model checker for real time systems modelled as networks of timed automata [**41**].

Feedback

"Is the feedback provided by the approach in a practitioner friendly form?"

Model checking tools generally provide feedback by means of a counterexample. In order to make sense of this counterexample some level of familiarity is assumed with the formal method. If the counterexample is transformed into a user friendly form, it would enable the UML practitioner to reap the benefits of the approach without having to necessarily understand the underlying methodology.

Automation

> *"Is the approach automated or manual?"*

Whether the approach is automated or not does not detract from the quality or relevance of the research. However, we include this in order to gauge the extent of automation in current research. We only consider an approach automated if there is explicit reference to this in the paper.

Integration

> *"Does the approach support integration with case tools?"*

UML is a graphical language, and, as such, it is beneficial to the UML practitioner if there is some form of integration with graphical tools. Moreover, if a well-known case tool is supported, rather than a proprietary application, it would provide even more benefit. We only report integration with case tools if there is explicit mention of this in the paper. Again, whether an approach supports integration with case tools or not does not detract from the quality or relevance of the research.

Evaluation

> *"How is the proposed approach evaluated?"*

The proposed research contribution can either be evaluated within the context of a substantive case study, or alternatively demonstrated with small pedagogic examples.

Scalability

> *"How well does the approach scale?"*

Model checking approaches frequently encounter the state space explosion problem. A suitable abstraction can alleviate the problem to an extent. The above question is non-trivial. However, we report on techniques employed to alleviate the state space explosion problem, or any special measures taken to address complexity in order to aid scalability. We address this question during the general discussion of the selected papers.

3.4 Results

This section summarises the findings of the literature review. Our work is concerned with providing a process algebraic semantics to behavioural SysML. As such, this section is divided accordingly: process algebraic model checkers and other model checking approaches.

3.4.1 Process Algebraic Model Checkers

We present the related work by model checking tool.

FDR

Abdelhalim *et al.* [**43**; **42**] presented an approach based on checking behavioural consistency between a UML state machine diagram and a corresponding fUML [**44**] activity diagram. The authors introduce a framework based on formalising these diagrams in the process algebra CSP; behavioural conformance is affirmed by means of a trace refinement. In the case of a failed refinement, the FDR counterexample is related back to the fUML model in an interactive fashion: a model debugger component allows the modeller to interactively debug the consistency problem on the fUML activity diagram. The modeller is isolated from the CSP domain as feedback is provided in the form of a sequence diagram, rather than a CSP counterexample. Model transformations are used to translate between the respective domains. The framework presented in [**43**; **42**] only supports the checking of liveness properties; verifying safety properties is omitted. Furthermore, the framework only permits a single instance per class.

Dan [**45**] translated sequence diagrams into CSP using a model-driven engineering methodology. In [**45**], the emphasis is placed on the translation process, which is insightful in terms of a mechanised implementation approach. However, the work is in contrast to that of Ng and Butler [**46**]. In [**46**], the authors directed their efforts towards defining a formal behavioural semantics in an implementation-independent manner. Dan [**45**] does not consider all complex interaction operators, and the formalisation does not allow for the modelling of message overtaking. In particular, the author only considers the optional, alternative, break and loop interaction operators.

Davies and colleagues [**47**; **48**] provided a behavioural semantics to combinations of class, object, and state machine diagrams using CSP. In addition they consider simple activity and sequence diagrams. A formal description of a simple activity diagram is provided in order to verify that the class description satisfies the requirements captured by the diagram. In addition, a sequence diagram is formalised and checked for consistency against the class description. Behavioural conformance is formalised within the context of traces and failures refinement of CSP. In [**47**], the authors explore different notions of concurrency. The possibility of using the refinement orderings to compare alternative designs for the same component, for example as part of the refactoring process, is also considered. The authors consider a limited subset of diagrammatic constructs; only basic forms of the aforementioned diagrams are considered in their treatment.

Küster and colleagues [**49**; **50**; **51**] developed a prototypical tool that consists of: a catalogue of consistency problems; transformation rules that allow the modeller to map UML into the semantic domain of CSP; and definitions of consistency checks for each problem in the catalogue. The consistency checks are subsequently executed against the model. In [**49**], the author proposed that sequence diagrams be used to demonstrate consistency violations rather than forwarding the trace of the refinement checker FDR. In related work, Küster *et al.* [**52**; **53**; **54**; **55**] investigated the notion of consistency within the context of UML-RT [**13**]. UML-RT is an extension of UML for modelling embedded, reactive and real time systems. Capsules are the fundamental modelling construct in a UML-RT model; and state machines model the behaviour of a capsule. As such, the main focus in these works is on state machine and sequence diagrams. Behavioural consistency is expressed within the semantic framework of CSP. In [**53**], the focus is on intra-model consistency, whereas in [**54**; **55**] they emphasise inter-model consistency. In the latter, they consider the effect of model transformations, performed within the context of UML-

RT, on the corresponding traces and failures refinement relations in the semantic domain of CSP.

Heckel and Küster [**56**] translated state machine diagrams into CSP using a model-driven engineering methodology. The resulting translation is used to analyse consistency concepts within the behavioural framework provided by CSP. The authors investigate consistency concepts within the context of an inheritance hierarchy, where one class is a generalisation of another. The behaviours of both classes are modelled with state machines. The state machine associated with a class is used to specify the order in which operations can legally be invoked on an instance of the class. The authors investigate two interpretations of behavioural conformity that can be applied to an inheritance hierarchy, namely invocable consistency and observable consistency. The former is based on the substitution principle, where any instance of a subclass can be used in the place of an instance of a superclass, which would imply that any sequence of operations invocable on the superclass can also be invoked on the subclass. The latter is based on the notion that a state machine is interpreted as a description of an upper bound to the observable sequences of method calls. This would imply that each sequence observable with respect to a subclass must result — under the projection of known methods — in an observable sequence of its superclass.

Ng and Butler proposed the formalisation of UML state machine diagrams using CSP as the semantic domain [**46**]. They define the translation in terms of a mapping function from structural diagrammatic constructs to their CSP counterparts. The translation starts from an initial state, and then proceeds to deduce the behaviour of the entire state machine in terms of CSP descriptions. Broadly speaking, each state is mapped to a process and each UML event is mapped to a CSP event. Their work considers simple states, composite states (both single region and orthogonal composite states with at least two regions) and choice pseudo states. Although the presented approach is arguably the most extensive formalisation utilising CSP as a semantic domain, they still omit key features, such as complex transitions involving junction pseudo states, join and fork pseudo states, and submachine states. In related work, Ng and Butler [**57**] present an approach for translating UML to CSP with the aim of utilising both UML and CSP in a system design. The authors partition the visualisation task based on three aspects of CSP: the dynamic behaviour; the static architecture; and refinement assertions. The approach allows for the visualisation of CSP using UML — with the focus on state machine and class diagrams.

Rasch and Wehrheim [**58**] used an object-oriented variant of Z [**59**] in combination with CSP. The authors used CSP-OZ [**60**] to check for consistency between a class and its state machine. The authors demonstrate how consistency checks between static and dynamic diagrams can be carried out using FDR. The approach is limited to simple state machine descriptions. In particular, their formalisation does not allow for complex transitions, or guards and effects on transitions. Several notions of consistency are defined within the context of their framework, and the effect of transformations on these notions of consistency are investigated. In [**61**] the authors showed how the requirements, in the form of sequence diagrams, are validated for a model consisting of class, state machine and composite structure diagrams. Once again, CSP-OZ is used as the formalism, and FDR is used as the refinement checker.

In [**62**], Varró *et al.* reflect on different graph transformation approaches to transform simple activity diagrams into CSP. The approaches considered are taken from the entrants of a tool contest. In all, eleven different approaches from different teams are compared.

The objective here was to evaluate different graph transformation tools, rather than on defining a behavioural semantics with the goal of formal verification.

Xu *et al.* [**63**; **64**] formalised activity diagrams in CSP. A transformation function is defined that maps the mathematical representation of an activity to the semantic domain of CSP. The goal in [**63**; **64**] is on providing a formal semantics for activities in terms of CSP, rather than checking behavioural conformance. Only a limited number of diagrammatic constructs are considered and only control flows are considered.

The work of Yeung *et al.* [**65**] built on that of Ng and Butler [**46**] by generalising inter-level transitions and prioritising transitions at different levels of the state hierarchy. The authors therefore provide an alternative semantics for state machines. However, their approach only takes into account those constructs formalised by Ng and Butler [**46**]. A simple example is provided to demonstrate the translation process.

PAT

Zhang and Liu [**66**] mapped state machine diagrams to CSP using the model checker PAT [**26**]. A state machine is modelled by a single CSP process; translation rules are presented that map state machine constructs to CSP# [**26**], the input language of PAT. Refinement checking as well as LTL [**25**] model checking is possible: both are natively supported by the model checker. The transformation methodology is presented via a set of rules — the presentation of which is slightly less formal than previous attempts such as those found in [**46**] and [**65**].

ProB

Turner *et al.* [**67**] used CSP ‖ B [**68**] as the formalism in their work on translating a subset of xUML [**69**] for the purpose of verification. The authors introduce a tool that utilises model transformations in order to generate the resulting CSP ‖ B specification. The focus is on class and state machine diagrams: associations, but not generalisations, are considered for classes; the behaviour of a class is described using a state machine. The framework provides support for reasoning about platform independent models: verification that a model preserves its multiplicities at certain execution points, and checks to ensure freedom from deadlock, is possible. The verification of safety properties is omitted.

Yeung [**70**] presented an approach where both CSP and B [**33**] are used to ensure consistency between a static class diagram and a dynamic state machine diagram. The formal methods CSP and B are applied separately in order to provide a clear separation of concerns with regards to behaviour and structure. The approach relies on that of Schneider and Treharne [**68**] for linking CSP and B descriptions. The author notes that further work is required to fully generalise the approach and handle more complex diagrammatic constructs.

CWB

Küster-Filipe [**71**] formalised sequence diagrams using CCS [**35**]. Conventional sequence diagrams are augmented with OCL [**72**] and provided with a formal behavioural semantics in terms of CCS. The resulting formalisation provides a means to reason about the

message-based behaviour of different objects. A concurrent communication logic is introduced that allows for the specification of constraints: it is possible to specify that certain behaviours are forbidden or mandatory.

MWB

Lam and Padget [**73**] utilised the π-calculus to provide a formal semantics for sequence and state machine diagrams. Both diagrams are encoded in the π-calculus; the resulting encodings are then used to check that the scenario described by the sequence diagram is permitted by the state machine diagram. Only a limited number of constructs are considered for both diagram types. The proposed methodology is validated within the context of a simple example.

We summarise the results for process algebraic approaches by variant, diagram type and purpose in Table 3.1.

3.4.2 Other Model Checking Tools

Checking the formal correctness or consistency of UML models using model checking received a considerable amount of attention in the past. We attempt to provide a representative overview of approaches that are not process algebraic in this section. In order to do so, we severely limit the scope and only include a few examples of each tool here.

SMV

Cunha *et al.* [**74**] defined a formal semantics for sequence diagrams by utilising Petri nets [**75**] as an intermediate format. The Petri net representation is subsequently transformed into a form suitable for input to the SMV model checker [**37**]. Safety and liveness properties are formulated in terms of CTL [**25**]. A properties specification interface is used for the specification of CTL properties to aid the modeller by providing a high level of abstraction. This eliminates, to an extent, the requirement to be familiar with CTL. The interaction constructs considered, however, are fairly basic: complex interaction operators are excluded from the treatment. No satisfactory justification is provided as to why Petri nets are utilised as the intermediate format, given that the destination formalism is the input language of SMV.

Yang *et al.* [**76**] formalised sequence diagrams by utilising timed coloured Petri nets as intermediate format. SMV is used for the purposes of verification. Both safety and liveness properties, expressed using CTL, are verified.

NuSMV

Eshuis [**77**] provided two alternative formulations for activity diagrams using the symbolic model checker NuSMV [**38**]. The formulations are based on the similarities between activity diagrams and state machines; one formulation is based on semantics proposed by Harel [**31**], whereas the other is based on the run-to-completion semantics of the OMG. The first formulation can be verified more efficiently than the second, as the former assumes perfect synchronisation. An activity diagram is first transformed into an intermediate representation, termed an activity hypergraph. The intermediate representation is subsequently utilised to transform the activity into an encoding suitable for the NuSMV

Paper	Variant	Diagrams	Purpose
Abdelhalim *et al.* [**43**; **42**]	UML 2.3 & fUML 1.0	ACT, STM	Inter-model & Inter-behavioural
Bolton and Davies [**48**]	UML 1.4	ACT, STM, SD	Intra-model & Inter-behavioural
Dan [**45**]	UML 2.2	SD	Verification
Davies and Crichton [**47**]	UML 1.4	ACT, STM, SD	Intra-model & Inter-behavioural
Engels *et al.* [**53**]	UML 1.3 & UML-RT	STM	Intra-model & Intra-behavioural
Engels *et al.* [**54**; **55**]	UML 1.3 & UML-RT	STM	Inter-model & Intra-behavioural
Heckel and Küster [**56**]	UML 1.4	STM	Intra-model & Intra-behavioural
Küster and Stroop [**52**]	UML 1.5 & UML-RT	SD, STM	Inter-model & Inter-behavioural
Küster [**49**; **50**]	UML 1.5 & UML-RT	STM	Intra-model & Intra-behavioural
Küster-Filipe [**71**]	UML 2.0 & OCL 2.0	SD	Verification
Lam and Padget [**73**]	UML 2.0	SD, STM	Intra-model & Inter-behavioural
Ng and Butler [**57**; **46**]	UML 1.3	STM	Verification
Rasch and Wehrheim [**58**]	UML 1.5	STM	Intra-model & Structural-behavioural
Rasch and Wehrheim [**61**]	UML 1.5	SD, STM	Intra-model & Inter-behavioural
Turner *et al.* [**67**]	UML 2.0 & xUML 1.0	STM	Intra-model & Structural-behavioural
Varró *et al.* [**62**]	UML 2.1	ACT	Verification
Xu *et al.* [**63**; **64**]	UML 2.0	ACT	Verification
Yeung [**70**]	UML 1.4	STM	Intra-model & Structural-behavioural
Yeung *et al.* [**65**]	UML 1.5	STM	Verification
Zhang and Liu [**66**]	UML 2.2	STM	Verification

Table 3.1: Classification of process algebraic approaches by variant, diagram and purpose. In the above ACT, SD and STM denote activity, sequence and state machine diagrams, respectively.

model checker. The approach is limited in the number of diagrammatic constructs considered.

Lam [**78**] introduced a formalism for reasoning about the correctness of activity diagrams. The main contribution of [**78**] is that the author considers the token-based semantics of activity diagrams. A manual translation process is presented to encode an activity into a form suitable for input to NuSMV. The formalisation supports the verification of LTL [**25**] properties on a single activity diagram; the interaction between multiple activity

Paper	Tool	Feedback	Automation	Integration	Evaluation
Abdelhalim *et al.* [**43**; **42**]	FDR	Y	Y	Y	Example
Bolton and Davies [**48**]	FDR	N	N	N	Example
Dan [**45**]	FDR	N	Y	Y	Example
Davies and Crichton [**47**]	FDR	N	Y	Y	Example
Engels *et al.* [**53**]	FDR	N	N	N	Example
Engels *et al.* [**54**; **55**]	FDR	N	N	N	Example
Heckel and Küster [**56**]	FDR	N	N	N	Example
Küster and Stroop [**52**]	FDR	N	Y	Y	Example
Küster [**49**; **50**]	FDR	Y/N	Y	Y	Example
Küster-Filipe [**71**]	CWB	N	N	N	None
Lam and Padget [**73**]	MWB	N	N	N	Example
Ng and Butler [**57**; **46**]	FDR	N	Y	N	Example
Rasch and Wehrheim [**58**]	FDR	N	N	N	Example
Rasch and Wehrheim [**61**]	FDR	N	N	N	Example
Turner *et al.* [**67**]	ProB	N	Y	Y	Example
Varró *et al.* [**62**]	FDR	N	Y	Y/N	Example
Xu *et al.* [**63**; **64**]	FDR	N	N	N	Example
Yeung [**70**]	ProB	N	N	N	Example
Yeung *et al.* [**65**]	FDR	N	N	N	Example
Zhang and Liu [**66**]	PAT	N	Y	N	Example

Table 3.2: Classification of process algebraic approaches by tool, feedback, automation, integration and evaluation. Y denotes yes; N denotes no.

diagrams are not considered. The author only considers control flows; object flows are excluded from the treatment.

SPIN

Del Mar Gallardo *et al.* [**79**] provided an overview of how model checking can be utilised to improve the quality of UML models. They consider only simple state machine and sequence diagrams, and many useful constructs are omitted. No explicit translation rules are provided as such; instead, the authors give general rules and guidelines that exploit the benefits of both graphical and model checking approaches. A small case study is used to illustrate concepts.

Guelfi and Mammar [**80**] included the notion of time in their formulation of activity diagrams. A clocked transition system is used to define the formal semantics, with the subsequent mapping into PROMELA [**39**] specified in terms of this structure. The authors mention that future work would include a strategy to reduce the state space, which would imply that the state space explosion problem is a limitation of the approach as it appears in [**80**]. User feedback is provided by highlighting the relevant flow of the activity diagram should an LTL property be violated.

Inverardi *et al.* [**81**] proposed the use of state and sequence diagrams for describing a system architecture. The approach is formalised within the context of the model checker SPIN [**39**]: architectural state diagrams are used to generate a PROMELA specification that reflects the separation among components at the architectural level. Scenarios, described by sequence diagrams, define a temporally ordered sequence of events; these are subsequently translated into LTL formulae. The authors validate the conformance between architectural state diagrams and scenarios by checking whether the PROMELA model conforms to the LTL specification. The treatment is limited to simple activity and state machine diagrams.

Jing *et al.* [**82**] described the verification of activity diagrams using the model checker SPIN. The authors formalise activity diagrams by making use of extended hierarchical automata [**83**]; the subsequent mapping into PROMELA is then defined in terms of this formalism. Their approach is based on that of Latella *et al.* [**84**]. In [**84**], the authors used hierarchical extended automata to formalise state machine diagrams. This approach, in turn, is similar to that proposed by Mikk *et al.* [**85**].

Latella *et al.* [**84**] described a translation from state machine diagrams into PROMELA. They make use of an operational semantics, defined in terms of extended hierarchical automata, in order the formalise state machine diagrams. The translation into the target language, PROMELA, is described in terms of the automata. Limitations of this approach are that it does not permit model checking more than one state machine diagram, and it only considers a subset of state machine constructs. Preliminary optimisations are discussed that alleviate the state space explosion problem to an extent.

Lilius and Paltor [**86**] investigated the analysis of state machine diagrams using the model checker SPIN, which allows linear temporal logic model checking. They introduce a tool which takes a UML model and converts it to an equivalent PROMELA specification, which is then fed into SPIN for analysis. The tool does not require any knowledge of the underlying formal method, because the properties to be verified are automatically translated into a suitable form; counterexamples, in the case of a violation, are represented using sequence diagrams. In order to keep the model checking process transparent, the tool does not support the model checking of LTL formulae. Collaboration, class and state machine diagrams are used to construct the specification. The authors define the execution semantics in terms of a hypothetical machine that simulate the run-to-completion semantics of UML state machines. The state space explosion problem appears to be a significant impediment on the complexity and size of the input models that the prototype tool can cope with.

Lima *et al.* [**87**] introduced an approach to formalise sequence diagrams in terms of the send and receive occurrences of messages. They formalise the semantics directly in terms of PROMELA. As part of the formalisation they consider combined fragments, allowing for the specification of complex interactions. LTL is used to write specifications that incorporate both the source and destination as well as the send and receive occurrence specifications of a particular message.

Liu *et al.* [**88**] used SPIN to provide verification support for state machine diagrams. The model checking of both safety and liveness properties is supported. In addition, the prototypical tool allows simulation based on animating the state machine. The approach permits the modelling of multiple communicating state machines.

Schäfer *et al.* [**89**] developed a prototypical tool to automatically verify whether the interactions expressed by a collaboration diagram can be realised by a group of state

machines. A PROMELA model is constructed from the state machines; collaboration diagrams are appropriately encoded as never claims. SPIN is used to verify the model against the automata and a counterexample is produced if the never claim is met. A small example is used to demonstrate the approach.

Zhao *et al.* [**90**] proposed a formalism called split automata to bridge the gap between state machine diagrams and SPIN. The authors argue that the resulting PROMELA specification is much more concise than an equivalent one obtained from flattened automata. They further point out that split automata do not solve the state space explosion problem — due to the fact that SPIN explicates the states during verification. However, the various state space reduction strategies of the model checker can alleviate the problem to an extent. Their focus is on state machine and sequence diagrams. The approach is validated within the context of an illustrative example.

UPPAAL

Huang *et al.* [**91**] extend state machines with temporal and probabilistic constructs. A model-driven engineering methodology is proposed to transform the augmented state machines into timed automata [**41**], as required by the model checker UPPAAL. Safety and liveness properties are expressed using LTL.

Diethers and Huhn [**92**] introduced a prototypical tool that verifies whether a group of state machines, modelling the system, satisfies a set of communication and timing constraints given as UML sequence diagrams. The tool employs the model checker UPPAAL [**40**] as underlying formalism. The verification results are presented in sequence diagram format.

Knapp *et al.* [**93**] described a prototypical tool designed to verify whether scenarios specified by collaborations are realised by a set of state machines, where both the collaborations and state machines contain timing information. The timed state machines are compiled into timed automata, as used by the model checker UPPAAL. A time-annotated collaboration is translated into an observer UPPAAL timed automaton. UPPAAL is then called upon to verify the timed automata against the observer timed automaton. The authors identify several shortcomings: the tool only handles a subset of state machine constructs; and there can only be one instance of a given class.

We summarise the results for approaches that are not process algebraic and classified according to variant, diagram type and purpose in Table 3.3.

3.5 Other Approaches

In the previous section we explored mechanised verification approaches previously applied to UML. In this section we present notable examples of other formalisations of behavioural UML, either to provide a semantics, or with a view to ensure and reason about consistency.

Model Finders

Alloy [**94**] is a bounded model finder that takes constraints of a model and finds structures that satisfy or violate them. Garis *et al.* [**95**] presented an approach that takes a UML model and converts it into an Alloy [**94**] specification. The intended behaviour of a class

Paper	Variant	Diagrams	Purpose
Cunha *et al.* [**74**]	UML 2.0	SD	Verification
Del Mar Gallardo *et al.* [**79**]	UML 1.4	SD, STM	Intra-model & Inter-behavioural
Diethers and Huhn [**92**]	UML 1.4	ACT, SD	Intra-model & Inter-behavioural
Eshuis [**77**]	UML 1.5	ACT	Verification
Guelfi and Mammar [**80**]	UML 1.5	ACT	Verification
Huang *et al.* [**91**]	SysML 1.2	STM	Verification
Inverardi *et al.* [**81**]	UML 1.3	SD, STM	Intra-model & Inter-behavioural
Jing *et al.* [**82**]	UML 2.0	ACT	Verification
Knapp *et al.* [**93**]	UML 1.4	SD, STM	Intra-model & Inter-behavioural
Lam [**78**]	UML 2.0	ACT	Verification
Latella *et al.* [**84**]	UML 1.3	STM	Verification
Lilius and Paltor [**86**]	UML 1.3	STM	Verification
Lima *et al.* [**87**]	UML 2.0	SD	Verification
Liu *et al.* [**88**]	UML 2.4	STM	IAM & IAB
Schäfer *et al.* [**89**]	UML 1.4	STM	Verification
Yang *et al.* [**76**]	UML 2.1	SD	Verification
Zhao *et al.* [**90**]	UML 2.0	SD, STM	Intra-model & Inter-behavioural

Table 3.3: Classification of other model checking approaches by variant, diagram and purpose. In the above ACT, SD and STM denote activity, sequence and state machine diagrams, respectively.

is specified using a protocol state machine; the class diagram is annotated with the *Object Constraint Language* (OCL) [**72**] to impose additional constraints. Alloy modules are created for both the protocol state machine and the OCL-annotated class diagram, and subsequently checked for consistency. If a violation is found a counterexample is returned. It should be noted that verification is only performed within a bounded scope; a single verification run does not equate to an exhaustive exploration of the state space — other counterexamples may still exist for different bounds.

Description Logics

Van der Straeten *et al.* [**96**; **97**] explored inter-model consistency using description logics [**98**]. Model transformations, either as a result of refactoring or due to model refinement, are investigated with the aim of preserving consistency. The authors formally define and explore the notion of consistency between a refined model and the original. The preservation of behavioural semantics of a refactored model with respect to the original is considered. A prototypical tool is developed as part of the approach.

Paper	Tool	Feedback	Automation	Integration	Evaluation
Cunha *et al.* [**74**]	SMV	N	Y	N	Example
Del Mar Gallardo *et al.* [**79**]	SPIN	N	N	N	Example
Diethers and Huhn [**92**]	UPPAAL	Y	Y	Y	Example
Eshuis [**77**]	NuSMV	Y	Y	Y	Example
Guelfi and Mammar [**80**]	SPIN	Y	Y	Y	Example
Huang *et al.* [**91**]	UPPAAL	N	Y	Y	Example
Inverardi *et al.* [**81**]	SPIN	N	Y	N	Case study
Jing *et al.* [**82**]	SPIN	N	N	N	Example
Knapp *et al.* [**93**]	UPPAAL	N	Y	N	Example
Lam [**78**]	NuSMV	N	N	N	Example
Latella *et al.* [**84**]	SPIN	N	Y	N	Case study
Lilius and Paltor [**86**]	SPIN	Y	Y	N	Example
Lima *et al.* [**87**]	SPIN	N	Y	Y	Example
Liu *et al.* [**88**]	SPIN	N	Y	N	Example
Schäfer *et al.* [**89**]	SPIN	N	Y	N	Example
Yang *et al.* [**76**]	SMV	N	Y	N	Example
Zhao *et al.* [**90**]	SPIN	N	N	N	Example

Table 3.4: Classification of other model checking approaches by tool, feedback, automation, integration and evaluation. Y denotes yes; N denotes no.

Abstract State Machines

Several behavioural UML diagrams have been given a formal semantics by making use of Abstract State Machines [**99**]. Notable examples include those of Börger *et al.* for activity [**100**] and state machine [**101**] diagrams.

Z-related Approaches

Kim and Carrington [**102**] used Object-Z [**103**] to define a formal behavioural semantics for state machines. Each model construct is formalised using an Object-Z class and given a denotational semantics. Based on this semantics, a detailed operational semantics is subsequently defined in terms of Object-Z operations and invariants.

Petri Nets

Several diagrams have been formalised using Petri nets [**75**]. For example, Carneiro *et al.* [**104**] and Andrade *et al.* [**105**] translated state machines and activities into Petri nets for analysis, respectively.

Paper	Variant	Diagrams	Purpose
Andrade *et al.* [**105**]	SysML 1.1	ACT	Formalisation
Börger *et al.* [**100**]	UML 1.3	ACT	Formalisation
Börger *et al.* [**101**]	UML 1.3	STM	Formalisation
Carneiro *et al.* [**104**]	SysML 1.1	STM	Formalisation
Kim and Carrington [**102**]	UML 1.3	STM	Formalisation
Van der Straeten *et al.* [**96**; **97**]	UML 2.0	STM, SD	Inter-model

Table 3.5: Classification of other formal approaches by variant, diagram and purpose.

Contract-based Approaches

Cimatti and Tonetta [**106**] explored the use of a paradigm where each component is associated with a contract: a clear description of the expected interaction of the component with its environment. The contract consists of the assumptions that must be satisfied by the environment, and, subsequently, the guarantees satisfied by the component in response to a request. Ultimately, the goal is to allow for a compositional approach to specification and design, using the concept of refinement and proof. The decomposition of a component into its constituent parts is complemented with the corresponding refinement of its contracts. The interested reader might also refer to the work of Sljivo *et al.* [**107**].

Table 3.5 provides a summary of the approaches described above.

3.6 Alternative Semantics

Many of the UML diagrams are closely related and inspired by other diagrams. In this section we briefly mention a few approaches where automated verification techniques were applied to these diagrams.

Harel State Charts

Mikk *et al.* [**85**] translated state machines with a view to utilise the model checker SPIN, but made use of the semantics proposed by Harel [**31**]. Similarly, Roscoe and Wu [**108**] used CSP and FDR to formalise state charts using the semantics proposed by Harel.

Message Sequence Charts

Sequence diagrams are based on, and closely related to message sequence charts. Alur *et al.* [**109**] used the model checker SPIN to reason about the latter.

3.7 Discussion

In this section we consider the results from the systematic literature review.

Kitchenham *et al.* [**27**] list the most common reasons for performing a review, which closely mirror our own. As such, they are listed below. Literature surveys are performed in order:

- to summarise the extent of current literature concerning a particular treatment;
- to identify areas where current research are lacking; and
- to provide the context within which to position new research activities.

Bearing the above, and the fact that we are using CSP as semantic domain in mind, we reflect on the results of the literature survey.

With regard to each of the behavioural diagrams considered, we make the following observations.

- The formalisation of state machines in CSP by Ng and Butler [**46**] is arguably the most comprehensive in terms of number of constructs considered. There is however still scope to further expand this work due to the fact that complex constructs such as join and fork pseudo states are not considered. In addition, the modelling of *do behaviours* can be extended. In particular, it is possible to model these with any UML behavioural feature.
- The work on formalising activities in CSP by Xu *et al.* [**63**; **64**] leaves much room for extension. Particularly, it is possible to consider constructs such as the token based semantics of activity diagrams in more detail. Call behaviour actions and structured activity nodes are not considered in [**63**; **64**].
- To the best of our knowledge, there has been no effort to provide a formal behavioural semantics for sequence diagrams using CSP. Dan [**45**] mapped sequence diagrams to the semantic domain of CSP, but with an emphasis on the translation approach. Moreover, only three of the twelve complex interaction operators are considered.

In general, we make the following additional observations.

- To the best of our knowledge, there has been no single treatment that considered all of the aforementioned behavioural diagrams in an integrated fashion. Moreover, no approach has been developed with a view to reason about complex behaviours composed of basic behavioural formalisms. A typical example is a state machine where the entry behaviour of a particular state is described via an activity. The integration of the behavioural semantics of different behaviours is a non-trivial task, and needs to be treated with caution. A common semantic framework is required to support the automated verification of such integrated behaviours.
- The work of Ng and Butler [**46**] and Xu *et al.* [**63**; **64**] provide extensive formalisations for state machines and activities, respectively. However, these diagrams are only considered in isolation. For example, in [**46**] the authors exclude the composition of two communicating state machines from their treatment.
- Approaches that consider more than one behavioural diagram all do so with the intent to check consistency. In most of the reviewed literature this involves a state machine and an interaction, but these are normally not formalised in great detail.

- Many of the formalisations above are based on behavioural diagrams that stem from old UML specifications. The standard is constantly evolving: many diagrams have a completely different semantics in the latest version of the standard than their counterparts in previous iterations. It would be beneficial to provide an updated semantics.

- To the best of our knowledge, none of the integrated formal approaches are validated within the context of a case study of sizeable complexity.

In particular, within the context of SysML, the following observations can be made.

- SysML introduces the notion of allocation via the satisfy and refine relationships. In particular, a behavioural construct can be assigned to a requirement in order to better describe the intention of that requirement. Requirements are indicated on a requirement diagram and are then linked to the model or model elements. There has been no approach to formalise this procedure in order to allow formal requirement traceability and verification via refinement checking.

- In general, there has been no approach that considered — within the context of SysML — the formalisation of the behavioural diagrams reviewed.

- SysML has concepts like abstract blocks, which allows for a top-down approach to specification. The abstract block describes the combined behaviour of a set of concrete realisations at the lower level. A formal framework is required to detect inconsistencies that can arise at different levels of specification due to the multi-view nature of SysML. Consistency problems that are unique to SysML need to be considered.

- SysML is compact with a limited number of diagrams. UML, on the other hand, offers the modeller more choice with regards to the behavioural formalisms supported. In UML, interactions can be described using sequence diagrams or interaction overview diagrams, amongst others. We argue that a more restrictive graphical modelling approach sits better with formal methods: the modelling activity is carried out more consistently. For example, a state machine will always be used to specify state based behaviour and we will always utilise sequence diagrams to model interactions between blocks. A more consistent modelling activity would be more amenable to automated formal methods. It is more beneficial to focus on a smaller subset, but provide a complete formal semantics.

- Significant effort has gone into the formalisation of UML profiles like UML-RT. SysML, however, is a standard — we argue that any formal approach is more likely to be adopted in industry than a similar approach introduced for a custom profile.

3.8 Conclusion

The amount of active research is a clear indication as to the importance of integrating behavioural formalisms and formal methods. The lack of a precise behavioural semantics for SysML renders reasoning about integrated behaviours impractical. In addition,

in contrast to UML, SysML supports the modelling of requirements explicitly in the language. The verification of requirements by means of refinement checking would thus be beneficial. With this in mind, we now set out to define a formal behavioural semantics as part of our efforts to develop a formal refinement framework.

PART II

A FORMAL REFINEMENT FRAMEWORK

4

State Machines

In this chapter we provide a formal behavioural semantics for SysML state machines using CSP. We start by placing certain restrictions on the state machines we consider in our formal framework. Additionally, we impose well-formedness constraints to ensure the behavioural semantics defined are sensible. Next, we define a structural mathematical model and additional modelling constructs that serve to clarify the presentation of the CSP descriptions. We then apply the formalisation to our running case study to illuminate the approach and to serve as a proof of concept. Next, we critically evaluate the theory proposed and list some of the advantages and potential shortcomings of the results. Some of the work presented in this chapter previously appeared in [**110**].

4.1 Assumptions

State machines are a rudimentary tool for any systems engineer: the different phases of a block or system and the intentions of its expected behaviour can be expressed in a clear, and (to an extent) exact manner. Our aim is to precisely formalise this intended behaviour by exploiting CSP. When mapping between two seemingly disparate notations, it is crucial that the source notation be adequately constrained in order to ensure that the resulting formulation in the target notation is sensible.

To this end, we place certain restrictions on the state machines we consider.

1. Every top-level state machine[1] in our formalisation is a simple composite state: the region of this state houses the entire state machine. This state has no associated state-based behaviours.

2. Orthogonal composite states are not allowed. As such, we do not consider fork and join states.

3. We do not support do behaviours.

[1]The state machine at the root of the state hierarchy.

4. Transitions between regions are limited to source and target states that are simple or simple composite states themselves. Thus transitions that cross regional boundaries are not permitted to start or terminate on pseudo states.

5. Submachine states are not supported.

6. We do not consider history pseudo states in our formalisation.

The restrictions introduced above have to do with the fact that our ultimate goal is refinement checking: we need to limit the potential impact of the state space explosion problem in order to make the mechanised verification of communicating state machines computationally tractable. We justify these restrictions in Section 4.6.

We additionally impose the following well-formedness constraints, in line with the standard [**1**; **2**].

1. Every region contains a valid state machine. In particular, this state machine is only allowed one compulsory initial state. Multiple final or terminate states are allowed, as appropriate.

2. Each initial state has a single outgoing transition: by insisting on a single outgoing transition we ensure that the first active starting state is unambiguously defined.

3. Incoming transitions are disallowed for initial states.

4. The outgoing transition of an initial state may not contain explicit triggers or guards.

5. Outgoing transitions are disallowed for final states.

We assume the above to always hold.

State machines and activities are two closely related formalisms in SysML. Typically, the latter can be viewed as a supplementary behaviour to the former: activities enrich the behaviour of state machines by allowing for the specification of additional state-based or transition-related behaviour. When integrating these behaviours, there are additional constraints that must hold in order for the formalisation of the combined behaviour to be valid[2]. We discuss these and the integration of the two formalisms in Chapter 7. For the purposes of this chapter, however, we assume that all behaviours are the empty behaviour. This would allow us to focus on state machines, the main focal point of this chapter.

The work in this chapter builds on that of Ng and Butler [**46**] in that we adopt a similar approach with regards to defining our behavioural semantics. In particular, we also make use of an abstract syntax and a mapping function to lay out our behavioural descriptions. The differences between the two proposals and our contribution in terms of extending the prior work are described in Section 4.6. The work of Davies, Crichton and Bolton [**47**; **48**] also significantly influenced our formulation.

[2]However, that said, it would still be possible to detect ill-defined behaviour using the process algebraic tools at our disposal: FDR can detect that a given refinement does not hold; alternatively, an animator can illuminate unwanted behaviour. We will see examples of this in Chapter 7.

4.2 Abstract Syntax

In order to define a formal semantics for state machines, we need a precise description of their syntax. To this end, we define simple mathematical constructs that are closely related to the syntactical structure of their corresponding SysML counterparts. Our purpose is not to formulate a complete syntactical specification of the considered constructs, but rather to employ these expositions to assist us in defining the formal behavioural semantics in a comprehensible and sympathetic fashion.

State Machines

Let $\mathcal{M}$ denote the set containing all state machines. In the following, we consider the formalisation as it relates to a single state machine $M \in \mathcal{M}$. A state machine defines state-based, event-driven behaviour in terms of a finite collection of states, and transitions between those states. A state machine M is thus an ordered pair (S_M, T_M), where:

- S_M represents the set of states; and
- T_M represents the set of transitions.

States

We further require that S_M is partitioned such that:

- S_M^I denotes the set of initial states;
- S_M^F denotes the set of final states;
- S_M^T denotes the set of terminate states;
- S_M^J denotes the set of junction states;
- S_M^C denotes the set of choice states;
- S_M^S denotes the set of simple states;
- S_M^{SC} denotes the set of simple composite states; and
- S_M^R denotes the set of regions.

The aforementioned sets are pairwise disjoint and fully partition S_M, that is:

$$(\forall s_i, s_j : \{S_M^I, S_M^F, S_M^T, S_M^J, S_M^C, S_M^S, S_M^{SC}, S_M^R\} \bullet s_i \neq s_j \Rightarrow s_i \cap s_j = \emptyset)$$
$$\wedge$$
$$\#S_M = \#S_M^I + \#S_M^F + \#S_M^T + \#S_M^J + \#S_M^C + \#S_M^S + \#S_M^{SC} + \#S_M^R$$

Every state in our formalisation has a unique name; in addition, every simple or simple composite state has an optional entry and exit behaviour. All state-based behaviours are modelled via activities. We define, for a state $s \in S_M^S \cup S_M^{SC}$, the following functions:

- the name, given by $name : S_M \rightarrowtail N_M$;

- the entry behaviour, given by *entry* : $(S^S_M \cup S^{SC}_M) \rightarrow \mathcal{A}$; and
- the exit behaviour, given by *exit* : $(S^S_M \cup S^{SC}_M) \rightarrow \mathcal{A}$.

In the above, N_M denotes the set of state names of state machine M. The function *name* is a total injective function: we are explicit in that no two states in a state machine M are allowed to have the same name. $\mathcal{A}$ represents the set containing all activities. The rest of the functions are all total: *entry* and *exit* return the empty behaviour[3] in the event that no behaviour is defined for s.

The set of outgoing transitions of a state $s \in S_M$ is given by

$$outgoing : S_M \rightarrow \mathbb{P}\, T_M$$

Similarly, the set of incoming transitions of a state $s \in S_M$ is given by

$$incoming : S_M \rightarrow \mathbb{P}\, T_M$$

These functions are defined to be total: when a given state has no outgoing (or incoming) transitions, the function *outgoing* (or *incoming*) returns the empty set.

Transitions

A transition consists of a trigger, a guard, and an effect; it exists between a source and a target state. Diagrammatically, we annotate a transition thus: **trigger**[guard]/*effect*. We define the following functions, to return for a transition t:

- the source state, given by *source* : $T_M \rightarrow S_M$;
- the target state, given by *target* : $T_M \rightarrow S_M$;
- the trigger, given by *trigger* : $T_M \nrightarrow \mathcal{S}$;
- the guard, given by *guard* : $T_M \nrightarrow Bool$; and
- the effect, given by *effect* : $T_M \nrightarrow \mathcal{A}$.

In the above, $\mathcal{S}$ denotes the set of all signals. We denote by *Bool* the Boolean result obtained from evaluating the guard of a transition t.

Every valid transition must have a source and target state. We distinguish between legal and illegal components of transitions. A *legal* component is a component that is permitted by the standard for a particular transition; similarly, an illegal component is forbidden. For example, the outgoing transition of an initial state is not allowed a guard: a guard component is illegal. However, a trigger component is legal[4]. The functions *trigger*, *guard* and *effect* are partial by virtue of the fact that not all components are legal for a transition t. However, to enhance the clarity of the presentation we make the following assumptions about the functions above.

- For a transition t, if the trigger component is legal then the trigger is either a completion event or an explicit trigger.

[3]Defined in Section 4.3.2.

[4]The trigger component is legal, but may only be a completion event.

- For a transition t, if the guard component is legal but the guard is omitted in the transition notation[5], then $guard(t)$ evaluates to true.
- For a transition t, if the effect component is legal but the behaviour is omitted in the transition notation[6], then $effect(t)$ is the empty behaviour.

Enclosing States

A state machine has an enclosing top state $s \in S_M$ at the root of the state hierarchy. This state is a simple composite state with a single region. Assume the existence of a function *top* that returns the enclosing top state:

$$\forall M : \mathcal{M} \bullet top(M) \in S_M^{SC}$$

To simplify the presentation we refer to this state as top_M when working with state machine M.

$$top_M == top(M)$$

We define a helper function *surround* that associates a state with its immediately enclosing state.

$$surround : S_M \nrightarrow S_M$$

The function above is partial by virtue of the fact that no state encloses the top state. Furthermore, we require *surround* to be anti-reflexive: the hierarchy imposed on states may not include circular definitions where a state is allowed to be its own enclosing state.

$$\forall s : \mathrm{dom}\, surround \bullet s \mapsto s \notin surround$$

We may also wish to consider all states that enclose a particular state. A second helper function

$$enclose : S_M \nrightarrow \mathbb{P}\, S_M$$

returns for a particular state $s \in S_M$ the set containing those states that immediately or transitively enclose s. Thus

$$\forall s : \mathrm{dom}\, enclose \bullet enclose(s) = surround^{+} (\!|\ \{s\}\ |\!)$$

Regions

Every simple composite state has exactly one region.

$$region : S_M^{SC} \rightarrowtail S_M^{R}$$

Note that a simple state has no region.

---

[5] In other words, no guard is specified.

[6] In other words, no effect is specified.

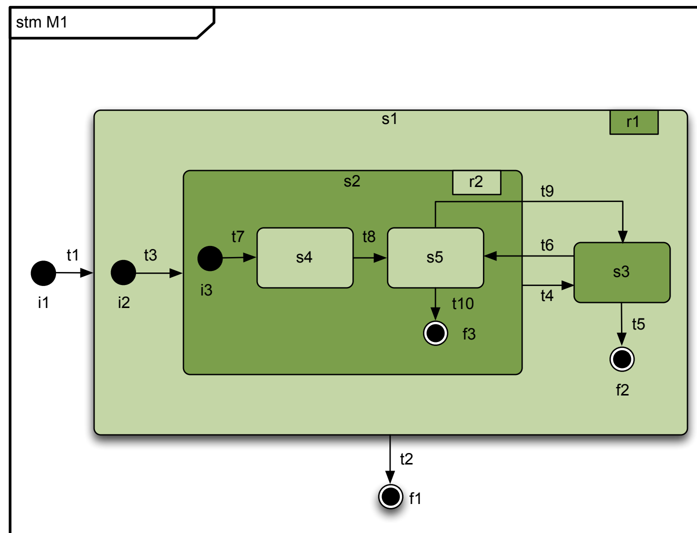


Figure 4.1: State machine M_1. M_1 is a hierarchical state machine. State s_1 is a simple composite state with a single region, r_1. We have merely labelled the transitions, omitting triggers, guards and effects. Similarly, entry, exit and do behaviours of the respective states are elided.

Example 4.1. Consider Figure 4.1: state machine M_1 with top state top_{M_1}. We have

$$
\begin{aligned}
&source(t_7) = i_3 \\
&target(t_7) = s_4 \\
&outgoing(s_4) = \{t_8\} \\
&incoming(s_4) = \{t_7\} \\
&surround(s_1) = top_{M_1} \\
&enclose(s_5) = \{s_2, s_1, top_{M_1}\} \\
®ion(s_1) = r_1
\end{aligned}
$$

□

4.3 Modelling Constructs

We introduce additional modelling constructs in this section in order to enhance the clarity of our presentation.

4.3.1 Pivot State

Before we consider entry and exit behaviour, we need to introduce the concept of a pivot state. A *pivot state* is the state, for a given transition, where the nested exit behaviour of the source state has finished execution, and the behaviour of the effect component is executed, before the nested entry behaviour of the target state begins.

We assume the existence of a function

$$pivot : T_M \to S_M$$

For a transition $t \in T_M$, we define a pivot state to be the innermost jointly transitive enclosing state with regards to the source and target states of t. Thus

$$\begin{array}{l} \forall\, t : \mathrm{dom}\, pivot \bullet \\ \quad pivot(t) = (\mu\, p : S_M \mid p \in innermost(enclose(source(t)) \cap enclose(target(t)))) \end{array}$$

In the above, the function

$$innermost : \mathbb{P}\, S_M \rightarrow \mathbb{P}\, S_M$$

returns, given a set of states, the set of states at the innermost level of the state hierarchy. More specifically, *innermost* returns from the input set of states only those that do not enclose any of the other states.

$$\begin{array}{l} \forall\, S : \mathrm{dom}\, innermost \bullet \\ \quad S = \{top_M\} \Rightarrow innermost(S) = \{top_M\} \\ \quad \wedge \\ \quad S \neq \{top_M\} \Rightarrow innermost(S) = \{s_i : S \mid (\forall\, s_j : S \mid s_j \neq top_M \bullet s_i \notin enclose(s_j))\} \end{array}$$

The function *pivot* returns a single pivot state by virtue of the fact that the source and target states must be enclosed by a single state somewhere in the state hierarchy. This state, in turn, can be enclosed by other states higher up in the hierarchy, hence our use of the function *innermost*. Thus, the common enclosing[7] state at the innermost level of the state hierarchy is known as the pivot state of t. The region of this state encloses both the source and target states of t. The effect component of t is executed at this point.

Example 4.2. Consider transition t_9 of Figure 4.1. The source and target states are s_5 and s_3, respectively. The sets of states that enclose s_5 and s_3 are the following.

$$\begin{array}{l} enclose(s_5) = \{s_2, s_1, top_{M_1}\} \\ enclose(s_3) = \{s_1, top_{M_1}\} \end{array}$$

The function pivot returns the pivot state s_1 of transition t_9.

$$\begin{aligned} pivot(t_9) &= (\mu\, p : S_M \mid p \in innermost(\{s_2, s_1, top_{M_1}\} \cap \{s_1, top_{M_1}\})) \\ &= (\mu\, p : S_M \mid p \in innermost(\{s_1, top_{M_1}\})) \\ &= s_1 \end{aligned}$$

□

4.3.2 Behaviours

In our formalisation all transition effects, entry and exit behaviours are modelled using activities. In general, an activity can be as complex as required, or may, instead, be reduced to a single, simple event.

[7] Enclosing both the source and target states of t.

Empty Behaviour

The aforementioned behaviours are all optional: the effect component of a transition can be omitted if not needed; similarly, entry and exit behaviours are not compulsory. For the case where the SysML behaviour is left unspecified, we make use of the empty activity. This approach is chosen in order to make the CSP formalisation easier and more concise. The empty activity, process A_ϵ, is defined thus.

$$A_\epsilon = Skip$$

Where a behaviour is not allowed (for example some transitions do not permit effects) we will explicitly flag this when setting out the formalisation.

Exit Behaviour

The exit behaviour is executed upon exiting a state. In particular, this happens after the triggering event, but before the behaviour specified by the effect component of the selected transition. Exit behaviours cannot be interrupted.

SysML allows the outgoing transitions of an enclosing state to be triggered from within one of the nested states. In this case, the exit behaviours of the transitively enclosing states will be triggered starting with the innermost active state until we reach the pivot state in the state hierarchy. These transitions that emanate from a composite state are termed *high level transitions*.

Let s be the current active state with outgoing transition t. Furthermore, restrict the source and target states of t to simple or simple composite states. Thus

$$s \in S_M^S \cup S_M^{SC} \land t \in outgoing(s) \land target(t) \in S_M^S \cup S_M^{SC}$$

We can formalise exit behaviour as follows.

$$\begin{aligned} Exit(s,t) &= exit(s) \fatsemi exit(s_0) \fatsemi \cdots \fatsemi exit(s_n) \\ \text{where} & \\ \{s_0 \mathinner{..} s_n\} &= \{s_i : enclose(s) \mid s_i \notin enclose(target(t))\} \\ s_0 &= surround(s) \\ s_n &= surround(s_{n-1}) \\ pivot(t) &= surround(s_n) \end{aligned}$$

It is important to note that the above is only defined between states that are not pseudo states. By imposing this restriction we can be confident that the function *exit* is defined for every state in the construction above. Moreover, integrating the semantics of high level transitions with that of complex transitions would add little in terms of expressive power, but would unnecessarily complicate the formalisation in terms of CSP.

Example 4.3. Consider Figure 4.1. When s_5 is active, the high level transition t_4 may fire. In this eventuality the exit behaviour of s_5 will be executed, followed by that of s_2, until the pivot state s_1 is reached. At this point in time, the behaviour specified for the effect of t_4 will execute.

□

Entry Behaviour

The entry behaviour is executed upon entering a state. Again, this behaviour is not susceptible to interruption. After the behaviour of the effect component has finished execution, the nested entry behaviour is executed starting with the outermost state inwards towards the designated target state.

We assume a current active state s with outgoing transition t. Therefore

$$s \in S_M^S \cup S_M^{SC} \wedge t \in outgoing(s) \wedge target(t) \in S_M^S \cup S_M^{SC}$$

The entry behaviour can be formalised as follows.

$$\begin{aligned}
&Entry(s,t) = entry(s_0) \fatsemi \cdots \fatsemi entry(s_n)\\
&\text{where}\\
&\quad \{s_0 \mathrel{..} s_n\} = \{s_i : enclose(target(t)) \mid s_i \notin enclose(s)\}\\
&\quad s_n = target(t)\\
&\quad s_{n-1} = surround(s_n)\\
&\quad pivot(t) = surround(s_0)
\end{aligned}$$

Again, we insist that the above is only defined for non-pseudo states to ensure the construction is valid.

Example 4.4. Consider Figure 4.1 again. Assume that s_3 is active and that transition t_6 has just fired. The exit behaviour of s_3, followed by the effect behaviour of t_6 will execute[8] next. Finally, the entry behaviour of s_2, followed by the entry behaviour of s_5, will execute.

□

4.3.3 Termination Condition

The *termination condition* relates to the generation of a completion event. A completion event is generated only once the relevant completion criteria, defined in terms of the termination condition of the currently active state, have been met. The completion criteria can be summarised thus.

- For a simple state $s \in S_M^S$ the termination condition is, according to the standard [**1**; **2**], the completion of the do behaviour. However, since we do not model do behaviours in our semantics, this condition is trivially satisfied upon entry into a state. We argue that the completion event should therefore be offered along with permitted explicit transitions.

- For a simple composite state $s \in S_M^{SC}$ the termination condition is satisfied, according to the standard [**1**; **2**], if either:
  - the final state of the state machine residing in the region of the composite state is reached; or
  - the do behaviour of s terminates.

[8] State s_1 is the pivot state.

As we do not model do behaviours, it follows that the termination condition is satisfied if and only if the state machine residing in the region of the composite state s has reached a final state.

4.3.4 Transitions

Transitions are key to our formalisation. We elaborate on the different components that constitute a transition and present basic concepts that aid in their formalisation.

Triggers

The trigger is the stimulus that enables a transition to fire. In this thesis we consider triggers to be either signal events or completion events.

- A signal event corresponds to the arrival of an asynchronous message typed by a signal. A signal event is an explicit trigger. As such, we must annotate the trigger component of the transition with the signal that typed the triggering event.
- A completion event is an implicit event that signifies the exit from a state. Diagrammatically, it is annotated on a transition by omitting the trigger component.

The definitions for implicit and explicit transitions can be stated thus.

$$\forall s : S_M^S \cup S_M^{SC} \bullet explicit(s) = \{t : outgoing(s) \mid trigger(t) \in \mathcal{S} \setminus \{\pi\}\}$$
$$\forall s : S_M \bullet implicit(s) = \{t : outgoing(s) \mid trigger(t) = \pi\}$$

Consider a state $s \in S_M$ with a single outgoing transition t. Thus

$$t \in T_M \wedge t \in outgoing(s) \wedge \#outgoing(s) = 1$$

We model the CSP process describing s as follows[9].

$$name(s) = trigger(t) \rightarrow name(target(t))$$

This approach is chosen in order to provide a uniform treatment for all triggering events. Thus for a transition $t \in T_M$ we use $trigger(t)$ to denote the trigger, whether that transition is explicitly or implicitly triggered.

In our formalisation, completion events are modelled using CSP events of the form π. Explicit events correspond to CSP events that reflect the name of the classifying signal, along with any parameters communicated as part of the event; a signal with an argument associated with it, will input on the CSP channel with the corresponding name. For example, if the signal is named $sig1$ with argument a, we use the CSP event $sig1?a$ as the trigger.

Guards

The guard of a transition must evaluate to true in order for the transition to occur; alternatively, if the guard is false the triggering event is consumed without effect.

[9]We ignore the guard and effect components for a moment, as well as all state-based behaviours.

Consider a state $s \in S_M$ with a single outgoing transition t. Thus

$$t \in T_M \wedge t \in outgoing(s) \wedge \#outgoing(s) = 1$$

We model the CSP process describing s as follows[10].

$$\begin{array}{l} name(s) = \\ \quad trigger(t) \rightarrow \\ \quad\quad \text{if } guard(t) \text{ then} \\ \quad\quad\quad name(target(t)) \\ \quad\quad \text{else} \\ \quad\quad\quad name(s) \end{array}$$

In the above, the evaluation of the guard following the triggering event determines the next active state.

A guard is evaluated based on the arguments that are served up along with the triggering event. For example, if we had the event $Sig?a$ as trigger, the a can be used in the evaluation of the guard. To preserve the clarity of the presentation, we simply write $guard(t)$ to denote the evaluation of the guard.

Effects

The effect is a behaviour, executed upon the transition between states; in this thesis such behaviours are described via activities. In particular, the behaviour of the effect component is executed after the nested exit behaviours of the source state, but before the nested entry behaviours of the target state. In the state hierarchy, this corresponds to the pivot state. The effect component of a transition t is given by $effect(t)$.

Transition Behaviour

The combined behaviour of a transition can now be modelled. Consider a state with a single high level transition t that contains a trigger, guard and effect; the high level transition is between non-pseudo states.

$$\begin{array}{l} name(s) = \\ \quad trigger(t) \rightarrow \\ \quad\quad \text{if } guard(t) \text{ then} \\ \quad\quad\quad Exit(s,t) \fatsemi effect(t) \fatsemi Entry(s,t) \fatsemi name(target(t)) \\ \quad\quad \text{else} \\ \quad\quad\quad name(s) \end{array}$$

Alternatively, if either the source or target state is a pseudo state, we know from the restrictions imposed in Section 4.1 that no transitions crossing regional boundaries are permitted. It follows that entry and exit behaviours, where applicable, are simplified and reduce to $entry(target(t))$ or $exit(s)$, respectively.

[10] We ignore the effect component for a moment, as well as all state-based behaviours.

4.3.5 Mapping Function

We make use of a mapping function $\mathcal{F}$ that maps the structural constructs to their CSP counterparts. Broadly speaking: each state in SysML corresponds to a process in CSP; each SysML event corresponds to a CSP event. Thus, the idea is that the mapping rules take us from a given SysML state to the next: in CSP this corresponds to initially behaving like one process, and then behaving like another. In each state the CSP process behaves like the state machine would in the corresponding SysML state. Every rule, $\mathcal{F}(M,s)$, is defined such that it describes the behaviour of state machine M at state s. These rules define process definitions, where each state is represented by a CSP process. Recall that a CSP process definition is of the format $\mathit{processname} = \mathit{behaviour}$. Because our formalisation maps states to processes, each state will result in a CSP process. In CSP, the name of the process describing state s is given by $\mathit{name}(s)$; the behaviour of this process will be given by $\mathcal{F}(M,s)$. Therefore

$$\mathit{name}(s) = \mathcal{F}(M,s)$$

The mapping rules for state machine M start from the initial state contained in the region of the top-level state, top_M. This approach is similar to that taken by Ng and Butler [**46**].

4.4 Behavioural Semantics

This section outlines an approach to integrate SysML state machines and CSP by providing a behavioural semantics for the former in terms of the latter. Throughout, we make use of the syntactical structures and modelling constructs defined in Sections 4.2 and 4.3, respectively.

The behavioural semantics of state machines, as defined in this thesis, are centred around the type of the currently active state. As such, the outline of our presentation mirrors this by providing a formalisation for each of the different state types in the remainder of this section.

4.4.1 Initial State

An initial state designates the first active state of state machine M. Consider an initial state $s \in S^I_M$ with a lone outgoing transition $t \in \mathit{outgoing}(s)$. A completion event, that serves as an implicit trigger, is generated upon entry to the pseudo state. Recall that the function *effect* returns the empty behaviour if no effect component is defined for t. Because no explicit triggers are allowed, only an implicit trigger is generated and $\mathit{trigger}(t) = \pi$. The CSP process modelling s follows.

$$\mathcal{F}(M,s) = \pi \rightarrow \mathit{effect}(t) \,\raise{0.5ex}{\text{\tiny ⨾}}\, \mathit{entry}(\mathit{target}(t)) \,⨾\, \mathcal{F}(M,\mathit{target}(t))$$

Note that a guard is not allowed on the outgoing transition.

An explicit transition is not allowed due to the fact that a state machine is not allowed to linger in a pseudo state. If we allowed explicit triggers emanating from state s, the state machine would be allowed to remain in s until the explicit signal event arrives. This is clearly not desired. By generating a completion event as soon as the initial state is

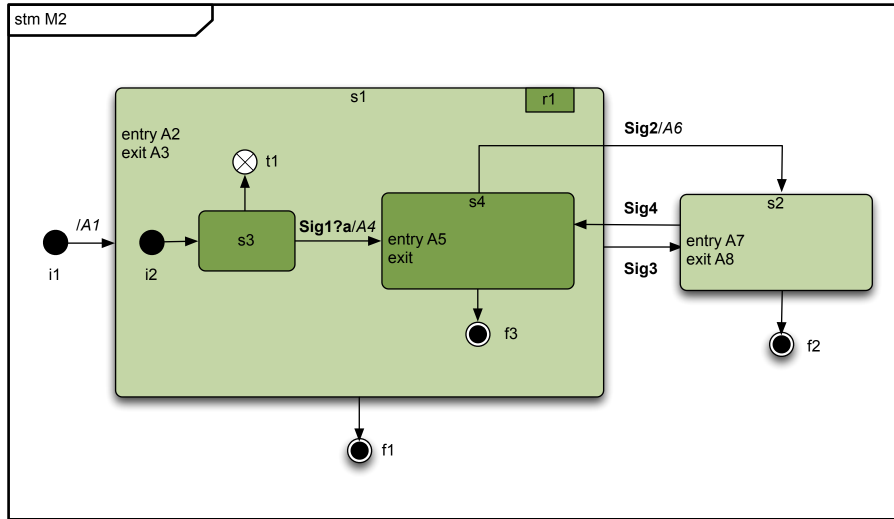


Figure 4.2: State machine M_2.

entered, we are guaranteed to progress to the next active state. Similarly, a state machine is not allowed to be hindered from transitioning out of the initial state by means of a guard that evaluates to false. Again, this would imply that the state machine would linger in the initial state: the completion event would be consumed without effect and the state machine would remain in the initial state.

We insist on the presence of an initial state in every region. This rule ensures that the state machine residing in the said region is well defined in the scenario where it is entered by default: an incoming transition ends on the state hosting the region.

$$\forall s_r : S_M^R \bullet (\exists_1 s_i : S_M^I \bullet surround(s_i) = s_r)$$

Example 4.5. Consider Figure 4.2. The initial state i_1 of M_2 can be formalised as follows.

$$\mathcal{F}(M_2, i_1) = \pi \rightarrow A_1 \fatsemi A_2 \fatsemi \mathcal{F}(M_2, s_1)$$

Similarly, we can formalise the initial state i_2 of region r_1 thus.

$$\mathcal{F}(M_2, i_2) = \pi \rightarrow A_\epsilon \fatsemi A_\epsilon \fatsemi \mathcal{F}(M_2, s_3)$$

In the above, the expression $\mathcal{F}(M, s)$ denotes the unique CSP process name of the state s in state machine M, that is $name(s)$. In the remainder of this chapter, we shall use the format containing the mapping function for clarity.

□

4.4.2 Final and Terminate States

A final state indicates the termination of a region. Subsequently, it has no outgoing transitions. Once a final state is reached, a completion event is generated to indicate the completion of all behaviour of the containing region, and, thus, the termination of the state machine. There is, however, no requirement for a region to contain a final state. For

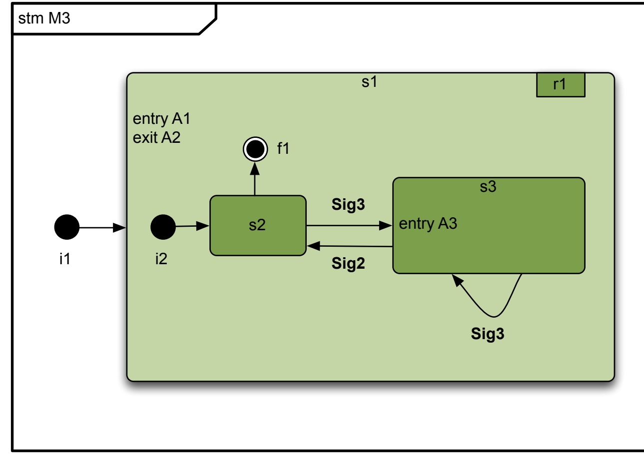


Figure 4.3: State machine M_3.

example, a state machine that never terminates would not have a final state.

Let $s \in S_M^F$ be a final state. If the final state resides within a simple composite state and there is at least one outgoing transition emanating from any of the transitively enclosing states, that is

$$surround(s) \in S_M^{SC} \wedge \exists\, s' : enclose(s) \bullet outgoing(s') \neq \emptyset$$

then reaching the final state represents the termination of the state machine in the region of the state $surround(s)$. The completion transition emanating from $surround(s)$, if present, or any of the explicit transitions emanating from transitively enclosing states, may be taken. Completion transitions emanating from transitively enclosing states other than $surround(s)$ may only be taken once their respective final states are reached.

$$\begin{array}{l}
\mathcal{F}(M,s) = \\
\quad \Box\, s' : enclose(s) \bullet \Box\, t : explicit(s') \bullet trigger(t) \rightarrow \\
\quad\quad \text{if } guard(t) \text{ then} \\
\quad\quad\quad Exit(s,t) \fatsemi effect(t) \fatsemi Entry(s,t) \fatsemi \mathcal{F}(M, target(t)) \\
\quad\quad \text{else} \\
\quad\quad\quad \mathcal{F}(M,s) \\
\quad \Box \\
\quad \sqcap\, t : implicit(surround(s)) \bullet trigger(t) \rightarrow \\
\quad\quad \text{if } guard(t) \text{ then} \\
\quad\quad\quad Exit(s,t) \fatsemi effect(t) \fatsemi Entry(s,t) \fatsemi \mathcal{F}(M, target(t)) \\
\quad\quad \text{else} \\
\quad\quad\quad \mathcal{F}(M,s)
\end{array}$$

Alternatively, if the final state resides within a simple composite state and there are no outgoing transitions emanating from any of the transitively enclosing states, that is

$$surround(s) \in S_M^{SC} \wedge \forall\, s' : enclose(s) \bullet outgoing(s') = \emptyset$$

then reaching the final state denotes the termination of M:

$$\mathcal{F}(M, s) = Skip$$

Reaching a final state in a non-hierarchical state machine is a special case of the last scenario.

The formalisation of a terminate state is trivial: the state machine as a whole terminates successfully, regardless of any outgoing transitions.

$$\mathcal{F}(M, s) = Skip$$

Example 4.6. Consider Figure 4.2 and state machine M_2. The final state f_3 residing in r_1 indicates the termination of the state machine residing in that particular region. There is at least one outgoing transition emanating from the states that transitively enclose f_3. The implicit as well as explicit outgoing transitions of s_1 will be available. Thus

$$\begin{aligned}
&\mathcal{F}(M_2, f_3) = \\
&\quad \pi \rightarrow A_3 \fatsemi A_\epsilon \fatsemi \mathcal{F}(M_2, f_1) \\
&\quad \Box \\
&\quad Sig_3 \rightarrow A_3 \fatsemi A_\epsilon \fatsemi A_7 \fatsemi \mathcal{F}(M_2, s_2)
\end{aligned}$$

□

Example 4.7. Consider Figure 4.3 and state machine M_3. The final state f_1 residing in r_1 indicates the termination of the state machine residing in that particular region. Since there are no outgoing transitions in any of the enclosing states, M_3 as a whole terminates.

$$\mathcal{F}(M_3, f_1) = Skip$$

□

Example 4.8. Consider Figure 4.2 and state machine M_2. The terminate state t_1 can be formalised as follows.

$$\mathcal{F}(M_2, t_1) = Skip$$

□

4.4.3 Junction and Choice States

A junction state is a transient point along a compound transition. The first leg of the compound transition contains the trigger and optional guard; note that an effect component is not allowed. Similarly, on the second leg, a trigger component is not allowed due to the fact that a complex transition must be selected in response to a single event. As the triggering event is assumed to have occurred on the first leg of the transition, it is disallowed for the second. Guard and effect components are allowed for the second leg of the compound transition.

A choice state differs from a junction state with regards to the first leg of the compound transition in that it additionally allows for an effect component. The consequence of

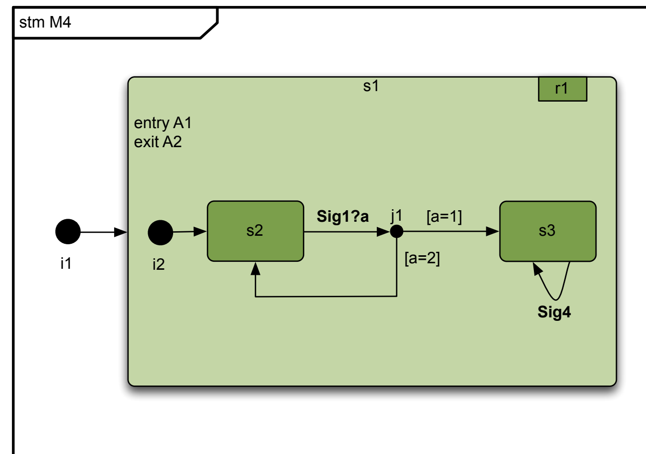


Figure 4.4: State machine M_4.

this is that the effect component can influence the outcome of the guards on the second leg. In contrast, a junction state does not permit an effect component on the incoming transition, and, as such, the guards on the second leg are in principle evaluated the instant the triggering event is served up for processing. However, as our formalisation is centred around the outgoing transitions, there is no difference between the CSP formalisation for a junction state, and that of a choice state. The distinction, however, would be made during the formalisation of the source state of the first leg of the compound transition.

Because junction and choice states are transient points along a complex transition, we do not allow explicit events to be triggered from the transitively enclosing states. Explicit events from the states that transitively enclose the junction or choice state can only be triggered once both legs of the compound transition have been taken. This semantics is in line with the notion that these are merely pseudo states along a compound transition, and that the transition must be completed before other events can be triggered.

The process modelling the junction or choice state is instantiated with the arguments passed on the trigger of the first leg of the complex transition. These arguments are used to evaluate the guards of the transitions emanating from the state. The parameters are indicated between the square brackets following the mapping function.

$$
\begin{aligned}
&\mathcal{F}(M,s)[params(trigger('t))] = \\
&\quad \Box\, t : outgoing(s) \bullet \\
&\qquad (\text{if } eval(guard(t), params(trigger('t))) \text{ then} \\
&\qquad\quad effect(t) \mathbin{;} entry(target(t)) \mathbin{;} \mathcal{F}(M, target(t)) \\
&\qquad \text{else} \\
&\qquad\quad Stop)
\end{aligned}
$$

Note that in our formulation the transitions that terminated and emanated from junction or choice states are not allowed to cross regional boundaries. In the above, the composition $params(trigger('t))$ correspond to the formal parameters of the signal that typed the send signal event that served as the trigger. The transition $'t$ is that of the first leg of the compound transition.

Example 4.9. Consider Figure 4.4 and state machine M_4. The junction state j_1 can be

modelled as a parameterised CSP process that takes as its parameter the argument passed on channel Sig_1. The transition emanating from s_2 is a complex transition: the trigger occurred on the first leg of the compound transition terminating on j_1; the second leg of the compound transition will then either be the transition terminating on s_2 or s_3, depending on the respective guards.

$$
\begin{array}{l}
\mathcal{F}(M_4, j_1)[a] = \\
\quad (\text{if } (a = 1) \text{ then} \\
\quad\quad A_\epsilon \fatsemi A_\epsilon \fatsemi \mathcal{F}(M_4, s_3) \\
\quad \text{else} \\
\quad\quad \mathit{Stop}) \\
\quad \Box \\
\quad (\text{if } (a = 2) \text{ then} \\
\quad\quad A_\epsilon \fatsemi A_\epsilon \fatsemi \mathcal{F}(M_4, s_2) \\
\quad \text{else} \\
\quad\quad \mathit{Stop})
\end{array}
$$

□

4.4.4 Simple State

A simple state $s \in S_M^S$ does not contain a region, and, as such, cannot host nested states. The behaviour of a simple state is formalised in terms of its own outgoing transitions, and the explicit transitions of the states that transitively enclose it.

$$
\begin{array}{l}
\mathcal{F}(M, s) = \\
\quad \sqcap\, t : \mathit{implicit}(s) \bullet \mathit{trigger}(t) \rightarrow \\
\quad\quad \text{if } \mathit{guard}(t) \text{ then} \\
\quad\quad\quad \mathit{Exit}(s,t) \fatsemi \mathit{effect}(t) \fatsemi \mathit{Entry}(s,t) \fatsemi \mathcal{F}(M, \mathit{target}(t)) \\
\quad\quad \text{else} \\
\quad\quad\quad \mathcal{F}(M, s) \\
\quad \Box \\
\quad \Box\, t : \mathit{explicit}(s) \bullet \mathit{trigger}(t) \rightarrow \\
\quad\quad \text{if } \mathit{guard}(t) \text{ then} \\
\quad\quad\quad \mathit{Exit}(s,t) \fatsemi \mathit{effect}(t) \fatsemi \mathit{Entry}(s,t) \fatsemi \mathcal{F}(M, \mathit{target}(t)) \\
\quad\quad \text{else} \\
\quad\quad\quad \mathcal{F}(M, s) \\
\quad \Box \\
\quad \Box\, s' : \mathit{enclose}(s) \bullet \Box\, t : \mathit{explicit}(s') \bullet \mathit{trigger}(t) \rightarrow \\
\quad\quad \text{if } \mathit{guard}(t) \text{ then} \\
\quad\quad\quad \mathit{Exit}(s,t) \fatsemi \mathit{effect}(t) \fatsemi \mathit{Entry}(s,t) \fatsemi \mathcal{F}(M, \mathit{target}(t)) \\
\quad\quad \text{else} \\
\quad\quad\quad \mathcal{F}(M, s)
\end{array}
$$

The above composition can be deconstructed as follows.

- The process offers the non-deterministic choice between all completion or implicit

transitions. The indexed form of the choice operator is used as it is possible for a state to have more that one completion transition.

- The process offers the deterministic choice between all explicit transitions emanating from s. Deterministic choice is selected as the transition taken is dependent upon the triggering signal; the decision as to which transition to take is not internalised as is the case for completion transitions. Thus, all possible transitions ought to be offered.

- For transitively enclosing states of s we only permit explicit transitions. Transitions emanating from the states that transitively enclose s are allowed to trigger whilst s is active. When a higher level transition fires while s is active, the state s is exited and the transition followed.

Each of the above corresponds to a different scenario under which s might be exited. However, the behaviour once an exit is triggered remains the same, as described by the expositions following the guard.

Example 4.10. Consider Figure 4.2 and state machine M_2. The simple state s_4 has an explicit transition that will fire on signal Sig_2. Explicit transitions that emanate from transitively enclosing states are permitted while s_4 is active: Sig_3 with source state s_1. However, implicit transitions that emanate from transitively enclosing states are not allowed to trigger an exit while s_4 is active: the implicit transition terminating on f_1 is disallowed and may only fire once the state machine residing in r_1 terminates. State s_4 can be formalised as follows.

$$
\begin{aligned}
&\mathcal{F}(M_2, s_4) = \\
&\quad \pi \rightarrow A_\epsilon \mathbin{⨾} \mathcal{F}(M_2, f_3) \\
&\quad \Box \\
&\quad Sig_2 \rightarrow A_\epsilon \mathbin{⨾} A_6 \mathbin{⨾} A_7 \mathbin{⨾} \mathcal{F}(M_2, s_2) \\
&\quad \Box \\
&\quad Sig_3 \rightarrow A_\epsilon \mathbin{⨾} A_\epsilon \mathbin{⨾} A_7 \mathbin{⨾} \mathcal{F}(M_2, s_2)
\end{aligned}
$$

$\Box$

4.4.5 Simple Composite State

A simple composite state $s \in S_M^{SC}$ has a single region which contains nested states. For that region, only one of the nested substates is allowed to be active. Recall that each region has a unique initial state. The behaviour of the composite state is described by the behaviours of the states that it contains. It follows that a transition to a composite state is equivalent to a transition to the initial state of that composite state. The composite state thus behaves like the initial state that it contains. The subsequent behaviour is then completely described by the target state of the initial state, and so forth. This is by virtue of the fact that our formalisation for non-composite states, like initial, final and simple states incorporate the transitions of the states that transitively enclose them.

Given $i \in (surround^{-1}(\!|\ \{s\}\ |\!)) \cap S^I_M$, the unique initial state of s, the formalisation is as follows.

$$\mathcal{F}(M,s) = \mathcal{F}(M,i)$$

Example 4.11. Consider Figure 4.2 and state machine M_2. The simple composite state s_1 initially behaves like its unique initial state i_2. Subsequent behaviour is contained within the behaviours of s_3, s_4, f_3 and t_1.

$$\mathcal{F}(M_2,s_1) = \mathcal{F}(M_2,i_2)$$

□

4.4.6 Event Queue

We model the event queue of a state machine using a CSP process called *EQ* that communicates on a channel *queue*.

$$EQ = queue?e \rightarrow local?p!e \rightarrow EQ$$

The above formulation is in essence a buffer process that sits between the state machine and its external environment. A triggering event *a* is communicated along channel *queue* by the external environment. The event is then consumed, and therefore removed from the buffer, by the state machine on channel *local*. An event can be consumed in one of two ways.

- The event can be processed: the current active state has an outgoing transition with a triggering event that correspond to the event served up for processing. In this case the event takes the form *local.proc.e*.
- The event can be discarded: the current active state has no outgoing transition corresponding to the event served up for processing. In this case the event takes the form *local.disc.e*.

The event queue is willing to communicate either event by inputting on the channel; the events offered by the current active state will determine whether the event is processed or discarded.

Here, we assume a queue with a maximum capacity of 1; the queue blocks when full. Non-blocking semantics, where events are discarded when the queue becomes full, is conceivable; so are event queues with different capacities. A semantics with an unbounded queue is also conceivable, although this is not finite state, and therefore not amenable to verification with FDR. However, animators can be used to explore the behaviour in the case of unbounded event queues.

4.4.7 State Machine

The state machine as a whole is modelled with a single process that contains, as appropriate, localised process descriptions corresponding to its syntactic structure. The overall structure is similar to that given by Davies and Crichton [**47**]. The environment external

to the state machine communicates with the event queue. The state machine receives all communications through its associated event queue, modelled as a CSP buffer of size 1. It communicates with this buffer on a CSP channel, *local*. Each of the localised processes has access to this channel in order to receive communications from the event queue. The overall process $M(queue, local)$ initially behaves as the process associated with the initial state $\mathcal{F}(M,i)$, and as each of the subsequent processes, until it terminates in state $\mathcal{F}(M,f)$. The local process EQ models the event queue.

$$
\begin{array}{l}
M(queue, local) = \\
\quad \text{let} \\
\quad\quad \mathcal{F}(M,i) = \ldots \\
\quad\quad \vdots \\
\quad\quad \mathcal{F}(M,f) = Skip \\
\quad\quad EQ = queue?e \rightarrow local?p!e \rightarrow EQ \\
\quad \text{within} \\
\quad\quad \mathcal{F}(M,i) \; [| \; \{| \; local \; |\} \; |] \; EQ
\end{array}
$$

The process definitions, as they stand in Sections 4.4.1–4.4.5, are not entirely complete: we need to adapt these for use within the context of the process M. In particular, triggering events need to be extended to include communication on the channel *local*.

The communication semantics for multiple state machines, as well as the integration with other constructs such as activities and interactions, are discussed in Chapter 7.

Example 4.12. Refer to Figure 4.4. State machine M_4 can be formalised as follows.

$$
\begin{array}{l}
M_4(queue, local) = \\
\quad \text{let} \\
\quad\quad I_1 = A_1 \mathbin{;} S_1 \\
\quad\quad S_1 = I_2 \\
\quad\quad I_2 = S_2 \\
\quad\quad S_2 = \\
\quad\quad\quad local.proc.Sig_1?a \rightarrow J_1(a) \\
\quad\quad\quad \Box \\
\quad\quad\quad local.disc?e : \{| \; Sig_4 \; |\} \rightarrow S_2 \\
\quad\quad J_1 = \\
\quad\quad\quad (\text{if } (a = 1) \text{ then } S_3 \text{ else } Stop) \\
\quad\quad\quad \Box \\
\quad\quad\quad (\text{if } (a = 2) \text{ then } S_2 \text{ else } Stop) \\
\quad\quad F_1 = Skip \\
\quad\quad S_3 = \\
\quad\quad\quad local.proc.Sig_4 \rightarrow S_3 \\
\quad\quad\quad \Box \\
\quad\quad\quad local.disc?e : \{| \; Sig_1 \; |\} \rightarrow S_3 \\
\quad\quad EQ = queue?e \rightarrow local?p!e \rightarrow EQ \\
\quad \text{within} \\
\quad\quad I_1 \; [| \; \{| \; local \; |\} \; |] \; EQ
\end{array}
$$

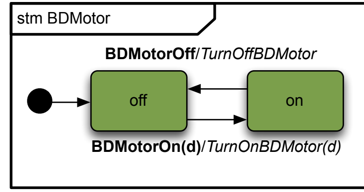


Figure 4.5: The state machine diagram of the BDMotor block.

In the above, we assume that the *provided* receptions[11] are Sig_1 and Sig_4.

□

4.5 Example

In this section we apply the concepts central to our methodology to an illustrative case study, with the focus on state machine constructs. The component studied here is the robotic arm, composed of a bidirectional motor, a magnet, a potentiometer and a controller.

Activities that execute on the transitions as the effect component are italicised after the trigger, set in bold typeface. For now, it is sufficient to assume that the activities are modelled using CSP processes: we provide formal semantics for activities in Chapter 5. Activities that model state-based behaviours are shown in the relevant states.

Figure 4.5 shows the classifier state machine of the bidirectional motor. The signals *BDMotorOff* and *BDMotorOn* serve as triggers between the different states. The *BDMotorOn* signal takes a parameter indicating the direction of the bidirectional motor.

$$
\begin{array}{l}
BDMotor(queue, local) = \\
\quad \text{let} \\
\qquad I_0 = OFF \\
\qquad OFF = \\
\qquad\quad local.proc.BDMotorOn?d \rightarrow \\
\qquad\qquad TurnOnBDMotor(d) \fatsemi ON \\
\qquad\quad \Box \\
\qquad\quad local.disc?e : \{\| BDMotorOff \|\} \rightarrow OFF \\
\qquad ON = \\
\qquad\quad local.proc.BDMotorOff?d \rightarrow \\
\qquad\qquad TurnOffBDMotor(d) \fatsemi OFF \\
\qquad\quad \Box \\
\qquad\quad local.disc?e : \{\| BDMotorOn \|\} \rightarrow ON \\
\qquad EQ = queue?e \rightarrow local?p!e \rightarrow EQ \\
\quad \text{within} \\
\qquad I_0 \; [| \; \{\| local \|\} \; |] \; EQ
\end{array}
$$

$$BDMOTOR = BDMotor(bdmotor, bdmotorlocal)$$

[11]The set of events, according to the block definition diagram, that the state machine makes available as services.

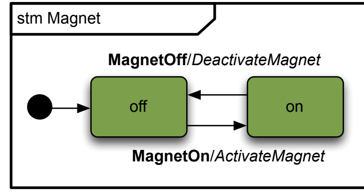


Figure 4.6: The state machine diagram of the Magnet block.

$$
\begin{array}{l}
\alpha BDMOTOR = \\
\quad Union(\{\{\mid bdmotor, bdmotorlocal \mid\}, \alpha TurnOnBDMotor, \alpha TurnOffBDMotor\})
\end{array}
$$

Note that in every state the dispatched, unexpected events are discarded and thus removed from the event queue without effect: this corresponds to communications of the form $local.disc.e$, where e is a signal event. Events that are served up for processing and successfully processed by the state machine correspond to communications of the form $local.proc.e$.

Figure 4.6 shows the state machine of the magnet. The *MagnetOn* triggering event causes the *ActivateMagnet* behaviour to execute on the transition; similarly, the *MagnetOff* event deactivates the magnet via the *DeactivateMagnet* transition behaviour.

$$
\begin{array}{l}
Magnet(queue, local) = \\
\quad \text{let} \\
\quad\quad I_0 = OFF \\
\quad\quad OFF = \\
\quad\quad\quad local.proc.MagnetOn \rightarrow \\
\quad\quad\quad\quad ActivateMagnet \,⨟\, ON \\
\quad\quad\quad \Box \\
\quad\quad\quad local.disc?e : \{\mid MagnetOff \mid\} \rightarrow OFF \\
\quad\quad ON = \\
\quad\quad\quad local.proc.MagnetOff \rightarrow \\
\quad\quad\quad\quad DeactivateMagnet \,⨟\, OFF \\
\quad\quad\quad \Box \\
\quad\quad\quad local.disc?e : \{\mid MagnetOn \mid\} \rightarrow ON \\
\quad\quad EQ = queue?e \rightarrow local?p!e \rightarrow EQ \\
\quad \text{within} \\
\quad\quad I_0 \;[\mid \{\mid local \mid\} \mid]\; EQ
\end{array}
$$

$$
MAGNET = Magnet(magnet, magnetlocal)
$$

$$
\begin{array}{l}
\alpha MAGNET = \\
\quad Union(\{\{\mid magnet, magnetlocal \mid\}, \alpha ActivateMagnet, \alpha DeactivateMagnet\})
\end{array}
$$

The state machine of the potentiometer is shown in Figure 4.7. The channel *pdmeter* is used for communication with the state machine of the potentiometer; the channel *pdmeterlocal* is used for internal communication between the event queue and state ma-

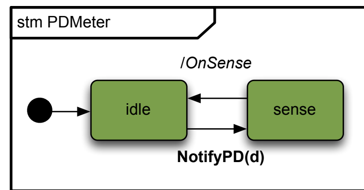


Figure 4.7: The state machine diagram of the PDMeter block.

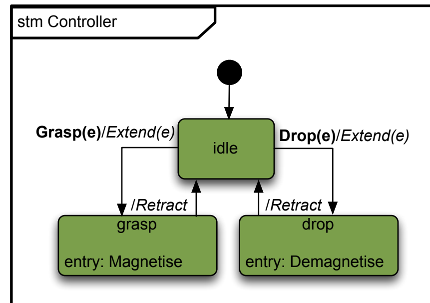


Figure 4.8: The state machine diagram of the Controller block.

chine.

$$
\begin{array}{l}
PDMeter(queue, local) = \\
\quad \text{let} \\
\quad\quad I_0 = IDLE \\
\quad\quad IDLE = \\
\quad\quad\quad local.proc.NotifyPD?pd \rightarrow SENSE \\
\quad\quad SENSE = \\
\quad\quad\quad OnSense \fatsemi IDLE \\
\quad\quad\quad \Box \\
\quad\quad\quad local.disc?e : \{| NotifyPD |\} \rightarrow SENSE \\
\quad\quad EQ = queue?e \rightarrow local?p!e \rightarrow EQ \\
\quad \text{within} \\
\quad\quad I_0 \; [| \{| local |\} |] \; EQ
\end{array}
$$

$$PDMETER = PDMeter(pdmeter, pdmeterlocal)$$

$$
\begin{array}{l}
\alpha PDMETER = \\
\quad Union(\{\{| pdmeter, pdmeterlocal |\}, \alpha OnSense\})
\end{array}
$$

The purpose of the controller is to coordinate the behaviour of the rest of the components of the arm, as per Figure 4.8.

$$
\begin{array}{l}
Controller(queue, local) = \\
\quad \text{let} \\
\quad\quad I_0 = IDLE \\
\quad\quad IDLE =
\end{array}
$$

$$
\begin{array}{l}
\quad\quad local.proc.Grasp?e \rightarrow \\
\quad\quad\quad Extend(local, e) \operatorname{;} Magnetise \operatorname{;} GRASP \\
\quad\quad \Box \\
\quad\quad local.proc.Drop?e \rightarrow \\
\quad\quad\quad Extend(local, e) \operatorname{;} Demagnetise \operatorname{;} DROP \\
\quad\quad \Box \\
\quad\quad local.disc?e : \{| \; OnPD \; |\} \rightarrow IDLE \\
\quad GRASP = \\
\quad\quad Retract(local) \operatorname{;} IDLE \\
\quad\quad \Box \\
\quad\quad local.disc?e : \{| \; Grasp, Drop, OnPD \; |\} \rightarrow GRASP \\
\quad DROP = \\
\quad\quad Retract(local) \operatorname{;} IDLE \\
\quad\quad \Box \\
\quad\quad local.disc?e : \{| \; Grasp, Drop, OnPD \; |\} \rightarrow DROP \\
\quad EQ = queue?e \rightarrow local?p!e \rightarrow EQ \\
\text{within} \\
\quad I_0 \; [| \; \{| \; local \; |\} \; |] \; EQ
\end{array}
$$

$$CONTROLLER = Controller(controller, controllerlocal)$$

$$
\begin{array}{l}
\alpha CONTROLLER = \\
\quad Union(\{\{| \; controller, controllerlocal \; |\}, \\
\quad\quad \alpha Magnetise, \alpha Demagnetise, \alpha Extend, \alpha Retract\})
\end{array}
$$

Alphabets of the individual processes are defined below each process definition. The alphabet of a state machine is the set of events that it can communicate, as well as the alphabets of its associated activities.

We can model the behaviour of the robotic arm as the combined behaviour of the bidirectional motor, the magnet, the potentiometer and the controller. Chapter 7 will discuss different interpretations of SysML blocks in more detail, depending on the context of use. For now, we can assume that the concrete behaviour of the arm can be described as the parallel composition of the component parts. Therefore, if

$$P = \{CONTROLLER, MAGNET, BDMOTOR, PDMETER\}$$

the arm can be modelled thus

$$CONCRETE = \parallel p : P \bullet [\alpha p] p$$

4.6 Discussion

In this section we discuss alternative CSP representations of state machines that are closest to our own. We reflect on our formalisation by highlighting key advantages of our approach, and conclude the section with a critical analysis of the work presented in this chapter.

4.6.1 Alternative Formalisations

Our approach can be considered a hybrid of the semantics defined by Ng and Butler [**46**], and those defined by Davies, Crichton and Bolton [**47**; **48**].

We extend or altered the semantics of Ng and Butler in numerous ways.

- In [**46**], the semantics allow multiple initial states per region, in contravention of the standard [**1**; **2**]. In contrast, our semantics only allow a single initial state per region.
- In [**46**], entry behaviours only consist of a single action. We allow activities as the entry behaviour.
- In [**46**], exit behaviours only consist of a single action. We allow activities as the exit behaviour.
- In [**46**], do behaviours are modelled as three discrete CSP events: an event indicating the start of the activity; an event indicating the end of the activity; and an event indicating some intermediate point during the execution of the activity. This approach is quite restrictive and allows very little scope for actually modelling do behaviours. In order to keep our semantics as elegant as possible we do not model do behaviours. Instead, we focus on providing a computationally feasible semantics for communicating state machines.
- In [**46**], transitions that emanate from source states located higher in the state hierarchy to target states located lower in the hierarchy are disallowed. For example in Figure 4.1, transition t_6 of state machine M_1 is not possible using the semantics of [**46**].
- Our interpretation of guards is fundamentally different. The standard [**1**; **2**] makes clear that if a guard evaluates to false, the triggering event must be discarded without effect. Ng and Butler simply do not offer the triggering event in the case where the guard is false, which would be fine if we were to only consider state machines in isolation. However, it causes a problem when two state machines communicate via the said event: the CSP process modelling the sending state machine would forever block, waiting on an event that is never offered by the CSP process modelling the receiving state machine.
- In [**46**], the authors do not model the event queue responsible for storing events to be processed. Instead, the generation and consumption of events are abstracted into a single event: the moment the event is processed. The focus in [**46**] is on defining the behaviour of a single state machine: less emphasis is placed on the external environment, and the semantics relevant to formalise the communication with other state machines. Instead, the authors assume that the environment external to a state machine is always willing to engage with the state machine by communicating all possible events. This interpretation, however, is unrealistic when we come to consider the behaviour of multiple state machines.

The semantics of Davies, Crichton and Bolton [**47**; **48**] address the concerns related to the formalisation of the execution mode discussed above. We extend the semantics of Davies, Crichton and Bolton in several ways.

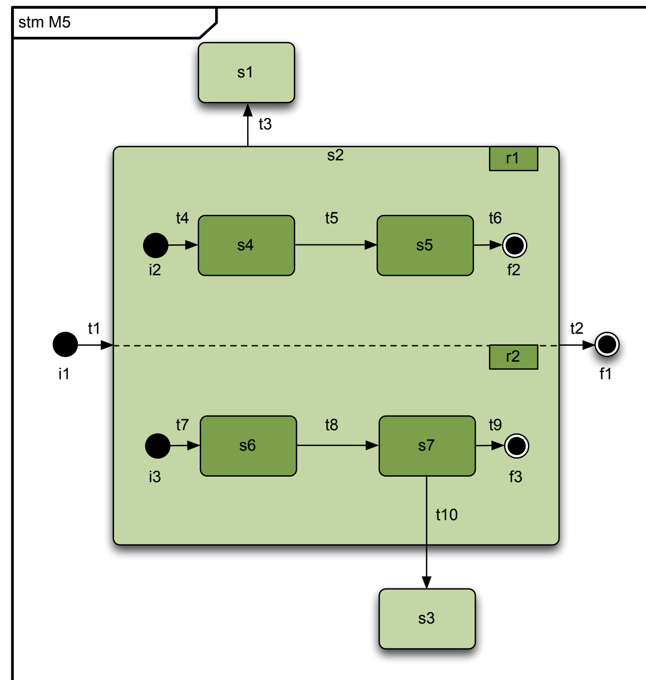


Figure 4.9: State machine M_5.

- In [**47**; **48**] the formalisation is based on non-hierarchical state machines. The main emphasis is placed on communication between the different state machines via the event passing mechanism of UML. We extend the work by providing a semantics for communicating, hierarchical state machines.
- We integrate the behaviours of state machines with that of activities.
- We formalise several constructs not addressed in [**47**; **48**]: hierarchical states; complex transitions via junction states; and complex transitions via choice states.

By considering a hybrid approach, encompassing the best of both formalisations, we are able to provide a realistic semantics for communicating state machines. This semantics forms the keystone for reasoning about the behaviour of a system comprised of systems.

We discussed the two semantics closest to our own above. For a detailed account of other approaches, the interested reader is referred to Chapter 3.

4.6.2 Reflections

We briefly outline the advantages and disadvantages of our approach and contend that, from the stance of formal verification, the approach's biggest weakness is also its biggest strength. We argue the case by presenting a narrative of how inclusion of certain constructs would either convolute our semantics, or would make the refinement checking of the resulting formalisms computationally intractable.

Advantages

The biggest advantage of our approach is that it presents a clear, comprehensible mapping between state machines and CSP. The resulting CSP representation is computationally tractable and offers a reasonable level of scalability. However, we readily acknowledge that the state space explosion problem will always eventually be an ultimate curb on the size and complexity of systems we can reasonably expect to model. In addition, we provided a semantics that is inherently integrated with activities — a key construct frequently used to supplement state-based or transition-related behaviours.

Disadvantages

The biggest criticisms of our approach is perhaps the exclusion of some of the key features of SysML state machines: the fact that we do not support do behaviours, or orthogonal composite states. The argument for supporting these is that they are popular modelling constructs used frequently within the SysML community.

Fork and join states are not formalised as these are merely syntactical sugar [**111**]; the constructs are furthermore only applicable to fork or merge the behaviour of different regions when entering and exiting an orthogonal composite state. Similarly, submachine states can instead be modelled using simple composite states. We acknowledge that submachine states do have the added advantage that they allow us to reuse existing state machines. However, their inclusion would not provide additional expressive power and we foresee little problems integrating them into our existing framework. History pseudo states are excluded: the construct would affect scalability and are less frequently used.

Critical Discussion

We contend that by omitting do behaviours from our semantics we are able to provide a more scalable CSP representation. To make the argument, consider if we had allowed do behaviours. We would have had to adapt the semantics as follows.

- For every state we would have had to define an additional process in parallel that models the respective do behaviour. Moreover, if we had a hierarchical state machine with multiple layers of nesting, the do behaviour of each simple composite state would be placed in parallel with the do behaviours of every state up to the innermost active state.

- We would have to alter the termination criteria for simple states. If any of the transitively enclosing states terminate (due to their do behaviours terminating), so should the do behaviour of the simple state. It follows that the do behaviour of every state needs to take account of the termination criteria of all its transitively enclosing states.

- We would have to alter the termination criteria for simple composite states. If the do behaviour of a simple composite state terminates, so should the do behaviour of every active state contained within the region of that composite state. The termination criteria of a composite state would also need to encompass the currently active state of all the states contained within its region: if a final state is reached within the region, the composite state's do behaviour needs to terminate.

Formalising the above in CSP would ultimately result in more convoluted process descriptions and negatively affect the state space. We can also argue, as Simons [**111**] does, that do behaviours simply don't belong in a canonical representation of state machines: the do behaviour of a simple state can instead be modelled by turning the simple state into a simple composite state.

We do not consider orthogonal composite states in our formulation. The idea of orthogonal composite states is to have concurrently active regions in a single state machine. We argue that since we place significant emphasis on the communication of a state machine with other state machines, that if concurrently active regions are required, they be modelled with separate state machines that execute within the context of different blocks. Furthermore, orthogonal composite states would only serve to further exacerbate the state space explosion problem: the do behaviours in different regions of the composite state are active simultaneously. We argue that formalising orthogonal composite states would be in contention with our goal of a clear CSP representation: due to the fact that subregions are essentially running in parallel, any exit from a composite state would terminate the behaviour of every active substate in that region; regions of a composite state would further need to cooperate on any common events.

As it stands, our semantics are also compositional, in the sense that we can define the behaviour of a substate completely by the transitions that transitively enclose it. This mirrors the informal behaviour of state machines where the transitions of transitively enclosing states are inherently present on the enclosed states. If we had included orthogonal composite states, we would break this sense of compositionality: we would be unable to define, for any substate of an orthogonal composite state, the semantics purely in terms of its transitively enclosing states. This observation follows from that fact that the behaviour of any region of an orthogonal composite state is dependent upon the behaviour of all other regions of the same orthogonal composite state: we simply cannot specify these independently.

Example 4.13. Refer to Figure 4.9. We are unable to formalise region r_1 independently of its context of use: transition t_{10} that exits region r_2 would also cause all behaviour in region r_1 to terminate. Thus, if we added more regions we would have to alter the definitions for all states contained in each of the regions belonging to a particular orthogonal composite state. This would therefore break our compositionality: a state is defined in terms of all its transitively emanating transitions. □

4.6.3 Possible Extensions

The model presented in this chapter has no notion of time, even though state machines can be modelled using transitions that fire based on events that denote the passage of time, called *timed events*. Tock CSP [**15**], Timed CSP [**112**] or the work of Schneider [**19**] can be used as the basis for formalising such semantics.

As it stands, our semantics non-deterministically select a transition in the eventuality where there are conflicts between transitions at different levels of the state hierarchy. In the case of a conflict, SysML state machines gives preference to the transition emanating from the innermost active state. A possible solution to this problem would be to alter the guards so that if the guard at the lower level evaluates to false, only then would the guard at the higher level be evaluated.

Clearly it is possible to derive a semantics for the features we deliberately ignored, but the benefits of doing this would have to be weighed against more convoluted CSP descriptions that might not sufficiently scale to structures that are computationally tractable.

4.7 Conclusion

In this chapter we provided a practical CSP formalisation for SysML state machines. The novelty of the approach is that we considered state machines within their context of execution: the SysML block. Our formalisation took into account the external execution environment and considered the system as a whole. In addition, we modelled the event queuing mechanism of SysML. In the next chapter we formalise activities, a behavioural construct typically used to augment the behaviour of state machines.

5

ACTIVITIES

In this chapter we provide a formal behavioural semantics for SysML activities using CSP. We start by placing certain restrictions on the activities we consider in our formal framework. Additionally, we impose well-formedness constraints to ensure that the behavioural semantics defined are sensible. Next, we define a structural mathematical model and additional modelling constructs that serve to clarify the presentation of the CSP descriptions. We then apply the formalisation to our running case study to illuminate the approach and to serve as a proof of concept. Next, we critically evaluate the theory proposed and list some of the advantages and potential shortcomings of the results. The work presented in this chapter previously appeared in [**113**].

5.1 Assumptions

Activities in our formalisation are typically assumed to execute within the context of a state machine. Additionally, the argument for imposing restrictions on the source notation to ensure sensible semantics in the target notation still applies.

We restrict the constructs permitted in activities as follows.

- Every activity has a unique starting point designated either by a single initial node, or if no initial node is present, the activity must have a single parameter node from which control starts.
- Flow final nodes are not permitted.
- We do not allow for activity partitions in our formalisation.
- An action can either have outgoing object flows, or outgoing control flows, but not both.
- The behaviour of a call behaviour action in our formalisation is assumed to be an activity.
- We exclude activities with activity parameter output nodes from consideration.

- We only consider simple fork and join nodes: if a fork node splits into k separate flows, then those k flows will eventually be joined via a join node.

We assume the following well-formedness constraints.

- Incoming flows are not allowed for initial nodes.
- Initial nodes are allowed a single outgoing control flow.
- Outgoing flows are not allowed for final nodes.
- Final nodes are only allowed incoming control flows.

5.2 Abstract Syntax

In order to define a formal behavioural semantics for activities, we need a precise description of their syntax. To this end, we define simple mathematical constructs that are closely correlated to their syntactical structure. Our purpose is not to formulate a complete syntactical specification, but rather to employ these expositions to assist us in defining the formal behavioural semantics in a comprehensible and sympathetic fashion.

Activities

Let $\mathcal{A}$ denote the set containing all activities. We consider the formalisation as it relates to a single activity $A \in \mathcal{A}$. An activity A is a quintuple $(A_A, R_A, P_A, CF_A, OF_A)$, where:

- A_A denotes the set of action nodes;
- R_A denotes the set of routing nodes;
- P_A denotes the set of parameter nodes;
- CF_A denotes the set of control flows; and
- OF_A denotes the set of object flows.

An activity defines flow-based behaviour in terms of nodes and edges: the nodes are represented by the sets A_A, R_A and P_A; the edges are denoted by CF_A and OF_A.

Action Nodes

Actions are the building blocks of activities. The set of action nodes A_A are further partitioned such that:

- A_A^{SS} denotes the set of send signal event actions;
- A_A^{RS} denotes the set of receive signal event actions;
- A_A^{VS} denotes the set of value specification actions;
- A_A^{O} denotes the set of opaque actions; and

- A_A^{CB} denotes the set of call behaviour actions.

The aforementioned sets are pairwise disjoint and fully partition A_A, that is

$$(\forall a_i, a_j : \{A_A^{SS}, A_A^{RS}, A_A^{VS}, A_A^{O}, A_A^{CB}\} \bullet a_i \neq a_j \Rightarrow a_i \cap a_j = \emptyset)$$
$$\wedge$$
$$\#A_A = \#A_A^{SS} + \#A_A^{RS} + \#A_A^{VS} + \#A_A^{O} + \#A_A^{CB}$$

Routing Nodes

Routing nodes are used to model complex flow-based logic. The set of control nodes R_A are partitioned thus:

- R_A^I denotes the set of initial nodes;
- R_A^F denotes the set of final nodes;
- R_A^{FK} denotes the set of fork nodes;
- R_A^{JN} denotes the set of join nodes;
- R_A^D denotes the set of decision nodes; and
- R_A^M denotes the set of merge nodes.

The aforementioned sets are pairwise disjoint and fully partition R_A, that is

$$(\forall a_i, a_j : \{R_A^I, R_A^F, R_A^{FK}, R_A^{JN}, R_A^D, R_A^M\} \bullet a_i \neq a_j \Rightarrow a_i \cap a_j = \emptyset)$$
$$\wedge$$
$$\#R_A = \#R_A^I + \#R_A^F + \#R_A^{FK} + \#R_A^{JN} + \#R_A^D + \#R_A^M$$

Parameter Nodes

Activity parameter nodes, denoted by P_A, are used to model the input and output arguments of activities. For our purposes however, we only consider activity parameter nodes that serve as inputs to activities; this is partly due to a desire to keep the semantics elegant and partly due to the context in which we use activities. We assume that a single argument is passed per node.

Object Flows

An object flow is used to model the passage of a parameter between two nodes: the parameter flows from the source or outputting node to the target or inputting node. We define the following functions for an object flow $of \in OF_A$.

- The function $source_{of} : OF_A \rightarrow (A_A^{RS} \cup A_A^{VS} \cup R_A^{FK} \cup R_A^{JN} \cup R_A^D \cup R_A^M \cup P_A)$ yields the outputting object node.
- The function $target_{of} : OF_A \rightarrow (A_A^{SS} \cup A_A^{CB} \cup R_A^{FK} \cup R_A^{JN} \cup R_A^D \cup R_A^M)$ yields the inputting object node.

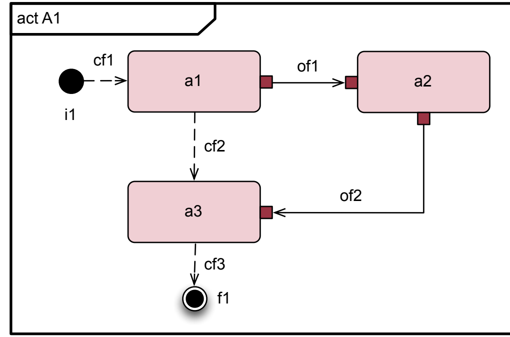


Figure 5.1: Activity A_1. A_1 is not allowed in our formalisation due to the fact that action a_1 has an outgoing control flow cf_2 as well as an outgoing object flow of_1. However, A_1 would have exactly the same behaviour if cf_2 was removed.

Note that the aforementioned are total functions: every object flow must have an associated source (or target) node.

The outgoing edges of a decision node have mutually exclusive guards that determine the next flow to be taken. The function

$$guard : OF_A \nrightarrow Bool$$

denotes the evaluation of the guard of an object flow emanating from a decision node. We denote by *Bool* the set $\{true, false\}$. The guard is evaluated based on the value of the object passed along the incoming flow of the decision node.

Control Flows

Control flows model the passage of control between the constituent nodes of an activity. Control flows are permitted between action and routing nodes. We define the following functions for a control flow $cf \in CF_A$:

- $source_{cf} : CF_A \rightarrow ((A_A \setminus A_A^{VS}) \cup R_A^I \cup R_A^{FK} \cup R_A^{JN} \cup R_A^M)$ yields the source node; and
- $target_{cf} : CF_A \rightarrow ((A_A \setminus A_A^{VS}) \cup R_A^F \cup R_A^{FK} \cup R_A^{JN} \cup R_A^M)$ yields the target node.

Guards on control flows are not modelled. Our formalisation of state machines evaluated guards based on the value of a parameter passed with the triggering event. In SysML, guards on control flows are used in a more general sense and typically expressed in natural language; interpreting guards using natural language is unsuitable in a formal framework, where an exact meaning is required. Moreover, we seek to provide a semantics where guards are interpreted in a uniform manner. Activities typically execute within the context of state machines and we will therefore interpret the guards in an analogous manner. The reason why we do not allow control flows via decision nodes stems from the fact that we need to assign a precise meaning to guards, but cannot do so if these are expressed using natural language. Thus we do not incorporate guards and exclude from consideration all constructs that require a semantic interpretation of guards on control flows.

Example 5.1. Refer to Figure 5.1 and activity A_1. We have

$$A_{A_1} = \{a_1, a_2, a_3\}$$
$$R_{A_1} = \{i_1, f_1\}$$
$$OF_{A_1} = \{of_1, of_2\}$$
$$CF_{A_1} = \{cf_1, cf_2, cf_3\}$$

□

5.3 Modelling Constructs

We introduce additional modelling constructs in this section in order to enhance the clarity of our presentation.

5.3.1 Outgoing Edges

The function

$$outgoing_{of} : (A_A^{RS} \cup A_A^{VS} \cup A_A^{CB} \cup R_A^{FK} \cup R_A^{JN} \cup R_A^{D} \cup R_A^{M} \cup P_A) \rightarrow \mathbb{P}\, OF_A$$

returns the set of outgoing object flows for certain types of nodes. We define a similar auxiliary function that returns, for certain types of nodes, the set of outgoing control flow edges:

$$outgoing_{cf} : ((A_A \setminus A_A^{VS}) \cup R_A^{I} \cup R_A^{FK} \cup R_A^{JN} \cup R_A^{M}) \rightarrow \mathbb{P}\, CF_A$$

The functions above are total: in the case where no outgoing edges are present, the empty set is returned.

Recall our assumption from Section 5.1 that an action can either have an outgoing object flow or an outgoing control flow, but not both:

$$\begin{aligned} &\forall\, a : \operatorname{dom} outgoing_{of} \cap \operatorname{dom} outgoing_{cf} \bullet \\ &\quad outgoing_{of}(a) \neq \emptyset \Rightarrow outgoing_{cf}(a) = \emptyset \\ &\quad \wedge \\ &\quad outgoing_{cf}(a) \neq \emptyset \Rightarrow outgoing_{of}(a) = \emptyset \end{aligned}$$

Example 5.2. Refer to Figure 5.1. A_1 is not allowed in our formalisation due to the fact that action a_1 has an outgoing control flow cf_2 as well as an outgoing object flow of_1. However, A_1 would have exactly the same behaviour if cf_2 was removed: action a_3 cannot begin execution until the object on of_2 is output by a_2. The order of execution is therefore a_1, followed by a_2, followed by a_3. Control flow cf_2, even though it is technically allowed by the standard [**1**; **2**], is redundant in this instance. Control flow cf_2 has no effect on the behaviour of A_1.

□

We readily acknowledge that there are cases where the restriction imposed above would limit the expressiveness of activities allowed in our formalisation. However, we

will justify this restriction and the consequences of its inclusion on the state space in Section 5.6.

5.3.2 Execution Environment

The activities in our formalisation execute within the context of a state machine. Each state machine has an event queue that interacts with the external environment under a run to completion assumption. The activities here thus execute under that same assumption using the event queue of the associated, owning state machine.

$$
\begin{array}{l}
M(queue, local) = \\
\quad \text{let} \\
\quad\quad A_1 = \ldots \\
\quad\quad A_2 = \ldots \\
\quad\quad \vdots \\
\quad\quad \mathcal{F}(M,f) = Skip \\
\quad\quad EQ = queue?e \rightarrow local?p!e \rightarrow EQ \\
\quad \text{within} \\
\quad\quad \mathcal{F}(M,i) \; [| \; \{| \; local \; |\} \; |] \; EQ
\end{array}
$$

In the above, activities A_1 and A_2 execute within the context of state machine M with event queue EQ.

5.3.3 Mapping Function

We make use of a mapping function $\mathcal{F}$ that maps the structural constructs to their CSP counterparts. Broadly speaking: every node in SysML corresponds to a process in CSP; similarly, every edge has an associated CSP process. The idea is that the mapping rules take us from the SysML source node to the target node: in CSP this entails initially behaving like the source process, then behaving like the edge connecting the source and target nodes, and then behaving like the target node. The behaviour of the edge is usually the behaviour of its target node; an object flow edge has an additional process parameter to model the flow of objects. Every rule, $\mathcal{F}(A,c)$, is defined such that it describes the behaviour of activity A at a particular construct c. These constructs are the nodes and edges of A. The mapping rules for activity A start from the initial node.

5.4 Behavioural Semantics

This section outlines an approach to integrate SysML activities and CSP by providing a behavioural semantics for the former in terms of the latter. Throughout, we make use of the syntactical structures and modelling constructs defined in Sections 5.2 and 5.3, respectively.

Our treatment closely reflects the structure of the activity as it appears on an activity diagram. The behavioural semantics of activities, as defined in this thesis, are centred

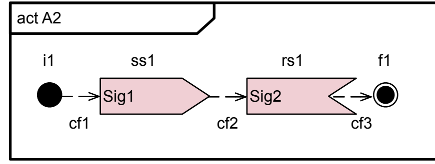


Figure 5.2: Activity A_2.

around the type of the currently active node. As such, the outline of our presentation mirrors this by providing a formalisation for each of the different node types in the remainder of this section. The semantics further distinguish and provide alternative interpretations based on the type of the connecting edges, activity parameter nodes and value specification actions.

5.4.1 Control Flow

Control flows are used to route the flow of control between different nodes of an activity. In our formalisation, a control flow $cf \in CF_A$ can be thought of as a CSP process. The behaviour of this process is dependent on the target node of the control flow, given by $target_{cf}(cf)$. If the target is not a join node, i.e. $target_{cf}(cf) \notin R_A^{JN}$, the process simply designates its behaviour to be that of the target node.

$$
\begin{aligned}
&\mathcal{F}(A, cf) = \\
&\quad \mathcal{F}(A, target_{cf}(cf)) && \text{if } target_{cf}(cf) \notin R_A^{JN} \\
&\quad Join(cf) && \text{otherwise}
\end{aligned}
$$

In the case where $target_{cf}(cf) \in R_A^{JN}$, there will be, based on our assumptions in Section 5.1, $k-1$ other control flows which terminate in the same join node. Let the control flows be $cf_0..cf_{k-1}$. Exactly one of the control flows, cf_0, will exhibit the behaviour of the join node.

$$
\begin{aligned}
&Join(e) = \\
&\quad join \rightarrow Skip && \text{if } e \neq cf_0 \\
&\quad join \rightarrow \mathcal{F}(A, target_{cf}(e)) && \text{otherwise}
\end{aligned}
$$

The above construction ensures that exactly one of the previously forked flows continues after the join node.

Example 5.3. Consider Figure 5.2. The control flow edge cf_1 of A_2 can be formalised as follows.

$$\mathcal{F}(A_2, cf_1) = \mathcal{F}(A_2, ss_1)$$

□

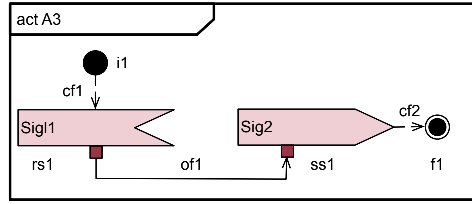


Figure 5.3: Activity A_3.

5.4.2 Object Flow

Object flows model the flow of objects between the different nodes in an activity. In our formalisation, an object flow $of \in OF_A$ behaves similarly to a control flow edge. However, the value of the object being passed along the flow is passed as a process argument between the processes representing the different constructs. We do not permit object flows to terminate on or emanate from join nodes, as discussed in Section 5.4.11. Thus, the behaviour of an object flow is given by the behaviour of the node that it terminates on, $target_{of}(of)$.

The process modelling the object flow simply designates its behaviour to be that of the target node, passing the object o as a process parameter.

$$\mathcal{F}(A, of)[o] = \mathcal{F}(A, target_{of}(of))[o]$$

Example 5.4. Consider Figure 5.3. The object flow edge of_1 of A_3 can be formalised as follows.

$$\mathcal{F}(A_3, of_1)[o] = \mathcal{F}(A_3, ss_1)[o]$$

The output of the receive signal event node is used to instantiate the process parameter o; the instantiation occurs in the process modelling the receive signal event rs_1.

□

5.4.3 Initial Node

An initial node $i \in R_A^I$ is a routing node that designates the starting point of an activity A. Subsequently, only a single control flow is permitted to emanate from i; object flows are disallowed and can neither emanate from, nor terminate on i. Let the unique control flow be

$$cf = (\mu f : CF_A \mid f \in outgoing_{cf}(i))$$

The formalisation of the initial node follows.

$$\mathcal{F}(A, i) = \mathcal{F}(A, cf)$$

Example 5.5. Consider Figure 5.2. The initial node i_1 of A_2 can be formalised as follows.

$$\mathcal{F}(A_2, i_1) = \mathcal{F}(A_2, cf_1)$$

The initial node behaves as the control flow emanating from it, but since the target node of this control flow is not a join node, the control flow edge simply behaves like the process describing its target.

□

5.4.4 Final Node

A final node $f \in R_A^F$ has no outgoing edges and a single incoming control flow edge. It is trivially modelled as the CSP process *Skip*.

$$\mathcal{F}(A,f) = Skip$$

Example 5.6. Consider Figure 5.2. The final node f_1 of A_2 can be formalised as follows.

$$\mathcal{F}(A_2,f_1) = Skip$$

The final node denotes the termination of the control flow of the activity and is modelled as the process *Skip*.

□

5.4.5 Send Signal Event Action

A send signal event action is used as a means of communication between different activities that execute within the context of state machines; the action corresponds to the sending of a signal event. A send signal event action node is entered either via a control flow or an object flow edge; in the case where the node is entered via a control flow an incoming object flow is possible, provided the object flow emanates from a parameter node. Note that a send signal event action is always exited via a control flow edge.

A send signal event action has an input pin corresponding to the attribute of the signal to be sent[1] and one input pin to specify the target for the signal.

Entry via Control Flow

A send signal event action $ss \in A_A^{SS}$, entered via a control flow edge, with a single outgoing control flow,

$$cf = (\mu f : CF_A \mid f \in outgoing_{cf}(ss))$$

can be formalised thus:

$$\mathcal{F}(A,ss) = target(ss).signal(ss) \rightarrow \mathcal{F}(A,cf)$$

A send signal event action has an input pin that names the target of the send signal action. In CSP, this corresponds to the channel name used to communicate with the target state machine, denoted by $target(ss)$ in the above. The name of the signal event is given by $signal(ss)$.

[1] No input pin is present if the signal does not have an associated attribute.

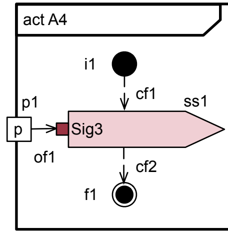


Figure 5.4: Activity A_4.

Optionally, an incoming object flow of is possible, which serves as input to the send signal event action, and models the parameters sent as part of the send signal event. In our semantics, the object flow of, if present, emanates from an activity parameter node $p \in P_A$ and terminates on the send signal event node ss; ss is therefore not entered via of, but of is instead used to pass a value to be used as part of the action. Note that an incoming control flow is still present and also terminates on ss. The construction $par(p)$ is the parameter available within the context of the owning activity, as defined per the arguments of the process modelling the activity.

$$\mathcal{F}(A, ss) = target(ss).signal(ss).par(p) \rightarrow \mathcal{F}(A, cf)$$

Example 5.7. Consider Figure 5.2. The send signal event action node ss_1 of A_2 is entered via a control flow and has no incoming object flow. The formalisation follows.

$$\mathcal{F}(A_2, ss_1) = tar_1.Sig_1 \rightarrow \mathcal{F}(A_2, cf_2)$$

We assume that tar_1 is the channel used to communicate with the target in the above.

□

Example 5.8. Consider Figure 5.4. The send signal event action node ss_1 of A_4 is entered via control flow cf_1; object flow of_1 connects ss_1 with parameter node p_1. Target tar_2 is specified for sig_3. The formalisation follows.

$$\mathcal{F}(A_4, ss_1) = tar_2.Sig_3.p \rightarrow \mathcal{F}(A_4, cf_2)$$

In the above, p is assumed to be available within the context of the process modelling A_4.

□

Entry via Object Flow

Consider a send signal event action $ss \in A_A^{SS}$, entered via an object flow edge, with a single outgoing control flow

$$cf = (\mu f : CF_A \mid f \in outgoing_{cf}(ss))$$

In this case, the process modelling the send signal event action would have an input argument, p, passed from the process modelling the incoming object flow.

$$\mathcal{F}(A, ss)[p] = target(ss).signal(ss).p \rightarrow \mathcal{F}(A, cf)$$

Example 5.9. Consider Figure 5.3. The send signal event action node ss_1 of A_3 is entered via object flow of_1. The formalisation follows.

$$\mathcal{F}(A_3, ss_1)[o] = tar_1.Sig_2.o \rightarrow \mathcal{F}(A_3, cf_2)$$

□

5.4.6 Receive Signal Event Action

A receive signal event node represents the action of receiving a signal event. A receive signal event action node can only be entered via a control flow, but may be exited either via a control or object flow.

A receive signal event action may output the received signal attribute on an output pin. An input pin is used to specify the source of the event.

Exit via Control Flow

A receive signal event node is typically exited via a control flow edge if the receive signal event has no associated attributes. Assume a receive signal event action $rs \in A_A^{RS}$ with a single outgoing control flow

$$cf = (\mu f : CF_A \mid f \in outgoing_{cf}(rs))$$

and no signal attributes associated with the signal. In the following, $source(rs)$ denotes the source specified for the receive signal event.

$$\mathcal{F}(A, rs) = source(rs).signal(rs) \rightarrow \mathcal{F}(A, cf)$$

If a signal attribute is present, we have an input on the CSP channel corresponding to the attribute a.

$$\mathcal{F}(A, rs) = source(rs).signal(rs)?a \rightarrow \mathcal{F}(A, cf)$$

Example 5.10. Consider Figure 5.2. The receive signal event action node rs_1 of A_2 is exited via a control flow cf_3. The formalisation follows.

$$\mathcal{F}(A_2, rs_1) = src_1.Sig_2 \rightarrow \mathcal{F}(A_2, cf_3)$$

□

Exit via Object Flow

A receive signal event action node can only be exited via an object flow if the signal has attributes that are output along the object flow. Assume a receive signal event $rs \in A_A^{RS}$ with a single outgoing object flow

$$of = (\mu f : OF_A \mid f \in outgoing_{of}(rs))$$

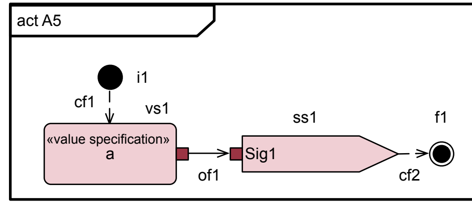


Figure 5.5: Activity A_5.

and an associated attribute a.

$$\mathcal{F}(A, rs) = source(rs).signal(rs)?a \rightarrow \mathcal{F}(A, of)[a]$$

The attribute a is passed along the object flow by instantiating the process modelling the flow correspondingly.

Example 5.11. Consider Figure 5.3. The receive signal event action node rs_1 of A_3 is exited via object flow of_1. The formalisation follows.

$$\mathcal{F}(A_3, rs_1) = src_2.Sig_1?a \rightarrow \mathcal{F}(A_3, of_1)[a]$$

□

5.4.7 Value Specification Action

A value specification action is a primitive action that outputs a constant value on its output pin. A value specification node is always entered via a control flow edge. Let $vs \in A_A^{VS}$ be a value specification action with outgoing object flow

$$of = (\mu f : OF_A \mid f \in outgoing_{of}(vs))$$

We have

$$\mathcal{F}(A, vs) = \mathcal{F}(A, of)[value(vs)]$$

In the above, $value(vs)$ denotes the value output by the action.

Example 5.12. Consider Figure 5.5. The value specification node vs_1 of A_5 outputs the value a via object flow of_1. The formalisation follows.

$$\mathcal{F}(A_5, vs_1) = \mathcal{F}(A_5, of_1)[a]$$

□

5.4.8 Opaque Action

An opaque action is an action executed in a language external to SysML. An opaque action may optionally take an input and, after executing the action, produce an output. In CSP, these opaque actions are modelled as CSP events. There are three possible formalisations for opaque actions permitted in our formalisation.

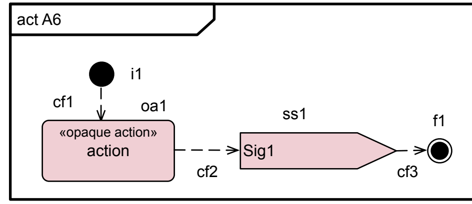


Figure 5.6: Activity A_6.

Entry and Exit via Control Flows

Assume an opaque action $oa \in A_A^O$ node entered via an incoming control flow, and an outgoing control flow

$$cf = (\mu f : CF_A \mid f \in outgoing_{cf}(oa))$$

In the following, $action(o)$ denotes the event corresponding to the opaque action of node oa.

$$\mathcal{F}(A, oa) = action(oa) \rightarrow \mathcal{F}(A, cf)$$

Entry and Exit via Object Flows

Assume an opaque action $oa \in A_A^O$ node with an incoming object flow and an outgoing object flow

$$of = (\mu f : OF_A \mid f \in outgoing_{of}(oa))$$

In the composition below, o denotes the object that serves as input to the opaque action, passed via the object flow.

$$\mathcal{F}(A, oa)[o] = action(oa) \rightarrow \mathcal{F}(A, of)[g(o)]$$

It is possible to view opaque actions as functional: the action takes an input and produces an output. In CSP, this can be modelled as a function g.

Entry via Object Flow; Exit via Conrol Flow

Assume an opaque action $oa \in A_A^O$ node with an incoming object flow passing object o and an outgoing control flow

$$cf = (\mu f : CF_A \mid f \in outgoing_{cf}(oa))$$

We have

$$\mathcal{F}(A, oa)[o] = action(oa) \rightarrow \mathcal{F}(A, cf)$$

Example 5.13. Consider Figure 5.6. The opaque action oa_1 of A_6 is entered via control

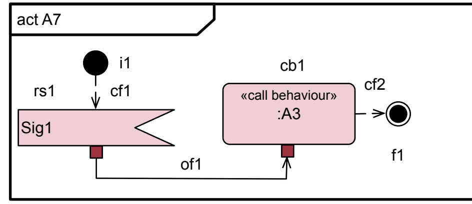


Figure 5.7: Activity A_7.

flow cf_1 and exited via control flow cf_2. The formalisation follows.

$$\mathcal{F}(A_6, oa_1) = action \rightarrow \mathcal{F}(A_6, cf_2)$$

□

5.4.9 Call Behaviour Action

A call behaviour action allows an activity to call another behaviour as one of its actions. Technically, this is any behavioural formalism of SysML; we only consider activities in this thesis. We restrict the activities used as call behaviour actions to only contain input pins; output pins are not permitted. Alternatively, a call behaviour action node may be entered via a control flow. In our formalisation, call behaviour nodes are always exited using control flows.

Entry via Control Flow

Consider a call behaviour action $cb \in A_A^{CB}$ with an incoming control flow and a single outgoing control flow

$$cf = (\mu f : CF_A \mid f \in outgoing_{cf}(cb))$$

The construction $behaviour(cb)$ represents the CSP process modelling the activity specified as part of the call behaviour action cb.

$$\mathcal{F}(A, cb) = behaviour(cb) \mathbin{;} \mathcal{F}(A, cf)$$

Entry via Object Flow

Consider a call behaviour action $cb \in A_A^{CB}$ with an incoming object flow and a single outgoing control flow

$$cf = (\mu f : CF_A \mid f \in outgoing_{cf}(cb))$$

The object o is passed to the CSP process representing $behaviour(cb)$.

$$\mathcal{F}(A, cb)[o] = behaviour(cb)[o] \mathbin{;} \mathcal{F}(A, cf)$$

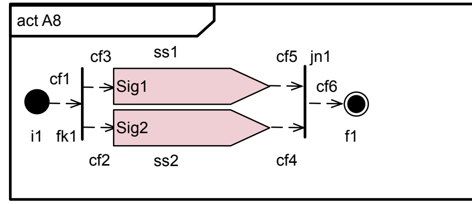


Figure 5.8: Activity A_8.

Example 5.14. Consider Figure 5.7. The call behaviour action cb_1 of A_7 is entered via an object flow of_1. The behaviour specified is that of activity A_3. The formalisation follows.

$$\mathcal{F}(A_7, cb_1)[o] = A_3(o) \fatsemi \mathcal{F}(A_7, cf_2)$$

□

5.4.10 Fork Node

A fork node splits a single flow into multiple separate flows. The flows can either be control flows or object flows. However, if the flow terminating on a fork node is a control flow, then control flows must exit the node; similarly, if an object flow terminates on the fork node, then multiple object flows must leave the fork node.

Control Flows

A fork node $fk \in R_A^{FK}$ operating on control flows splits the incoming control flow in k separate outgoing flows $cf_0 \ldots cf_{k-1}$. The alphabetised indexed parallel construction ensures that all the different threads of control only synchronise on the *join* event; all other events are interleaved.

$$\mathcal{F}(A, fk) = [|join|]\ cf : outgoing_{cf}(fk) \bullet \mathcal{F}(A, cf)$$

Object Flows

The formalisation for object flows is similar. The difference is that the object is passed to each of the k separate forked flows. A fork node $fk \in R_A^{FK}$ operating on object flows splits the incoming object flow in k separate outgoing object flows $cf_0 \ldots cf_k$.

$$\mathcal{F}(A, fk)[o] = [|join|]\ of : outgoing_{of}(fk) \bullet \mathcal{F}(A, of)[o]$$

Example 5.15. Consider Figure 5.9. The fork node fk_1 of A_9 splits object flow of_1 into object flows of_2 and of_3. The formalisation follows.

$$\mathcal{F}(A_9, fk_1)[o] = \mathcal{F}(A_9, of_2)[o]\ [|\ join\ |]\ \mathcal{F}(A_9, of_3)[o]$$

□

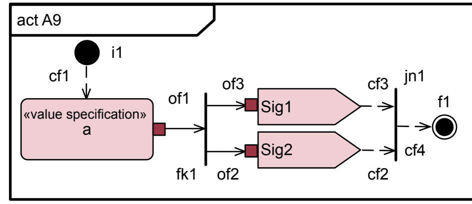


Figure 5.9: Activity A_9.

5.4.11 Join Node

A join node synchronises previously forked flows: it has k incoming flows and a single outgoing flow. We only formalise join nodes that operate on control flows. The reason for this is that the semantics of join nodes that operate on object flows do not sit well with our formalisation. The object flows in our semantics essentially deal with process arguments, i.e. concrete values. However, token flow semantics on object flows via join nodes require that each token on every flow is passed to the outgoing flow. This would be inconsistent with our treatment of object flows and are thus excluded.

A join node $jn \in R_A^{JN}$ synchronises k parallel control flows and has a single outgoing control flow, cf:

$$cf = (\mu f : CF_A \mid f \in outgoing_{cf}(jn))$$

Recall that only one of the previously forked flows will behave as the join node.

$$\mathcal{F}(A, jn) = \mathcal{F}(A, cf)$$

Example 5.16. Consider Figure 5.9. The join node jn_1 of A_9 unifies control flows cf_2 and cf_3 into a single flow cf_4. One of the incoming control flows will insist on the *join* event and then behave as the process modelling the join node; the other will insist on the *join* event and then behave as *Skip*. It follows that only one of the previously forked flows will continue.

$$\mathcal{F}(A_9, jn_1) = \mathcal{F}(A_9, cf_4)$$

□

5.4.12 Decision Node

A decision node offers the choice between possible alternative flows, based on the evaluation of guards. A decision node has an incoming flow edge and several outgoing flow edges, with mutually exclusive guards placed on the outgoing edges. However, only one of the outgoing edges — the edge where the guard evaluates to true — is taken. Our formalisation only incorporates object flows via decision nodes. This is intrinsically linked with the evaluation of the guards of the outgoing edges, as explained in Section 5.2.

A decision node $d \in R_A^D$ over object flows passes an object along one of several possible object flows. The k alternative outgoing object flows $of_0 \ldots of_{k-1}$ are evaluated, where a particular object flow $of_i \in outgoing_{of}(d)$. The following assumes mutually exclusive

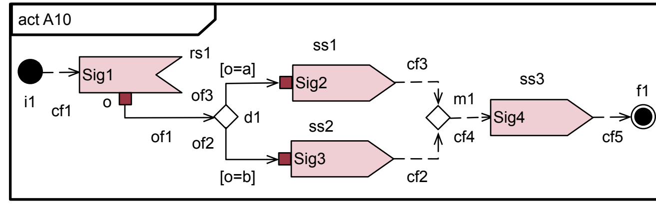


Figure 5.10: Activity A_{10}.

guards. A guard may contain an else clause, in which case it is trivially mapped to a CSP else construct. Machine-readable CSP only has an *if then else* conditional construct. We are therefore forced to adapt the formalisation to include the process *Stop*, as presented below.

$$
\begin{array}{l}
\mathcal{F}(A,d)[o] = \\
\quad \text{if } \mathit{guard}(\mathit{of}_0) \text{ then} \\
\quad\quad \mathcal{F}(A,\mathit{of}_0)[o] \\
\quad \vdots \\
\quad \text{else if } \mathit{guard}(\mathit{of}_{k-1}) \text{ then} \\
\quad\quad \mathcal{F}(A,\mathit{of}_{k-1})[o] \\
\quad \text{else} \\
\quad\quad \mathit{Stop}
\end{array}
$$

Example 5.17. Consider Figure 5.10. The decision node d_1 of A_{10} offers the choice between two alternative object flows of_2 and of_3, based on the evaluation of the guards of the two outgoing flow edges.

$$
\begin{array}{l}
\mathcal{F}(A_{10},d_1)[o] = \\
\quad \text{if } o = a \text{ then} \\
\quad\quad \mathcal{F}(A_{10},\mathit{of}_3)[o] \\
\quad \text{else if } o = b \text{ then} \\
\quad\quad \mathcal{F}(A_{10},\mathit{of}_2)[o] \\
\quad \text{else} \\
\quad\quad \mathit{Stop}
\end{array}
$$

□

5.4.13 Merge Node

A merge node has several incoming flows, but only one outgoing flow edge. The merge node behaves like the process modelling its outgoing flow.

Control Flows

A merge node $m \in R_A^M$ has a single outgoing control flow, cf:

$$\mathit{cf} = (\mu f : CF_A \mid f \in \mathit{outgoing}_{\mathit{cf}}(m))$$

The behaviour can be modelled thus.

$$\mathcal{F}(A,m) = \mathcal{F}(A,cf)$$

Object Flows

A merge node $m \in R_A^M$ has a single outgoing object flow

$$of = (\mu f : OF_A \mid f \in outgoing_{of}(m))$$

along which the object originating from one of the incoming flows is passed.

$$\mathcal{F}(A,m)[o] = \mathcal{F}(A,of)[o]$$

Example 5.18. Consider Figure 5.10. The merge node m_1 of A_{10} brings together alternative control flows cf_2 and cf_3 into a single control flow cf_4. The merge node behaves as the process modelling its outgoing control flow edge.

$$\mathcal{F}(A_{10},m_1) = \mathcal{F}(A_{10},cf_4)$$

□

5.4.14 Activity

The activity as a whole is modelled with a single process that contains, as appropriate, localised process descriptions corresponding to its syntactic structure. We provide two formalisations, based on whether the activity has an initial node, or whether execution starts from the activity parameter node.

Activity with Initial Node

An activity with an initial node always starts execution from the initial node, whether an activity parameter node is present or not. Initially, the overall process behaves like the initial node $i \in R_A^I$. In the following, the process argument a corresponds to the argument of the activity parameter node; if no parameter node is present, the process argument is elided. The argument a is a global parameter available for use anywhere in the activity; diagrammatically, the use of this parameter would be indicated with an object flow connecting the parameter node and the node using a as input.

$$\begin{array}{l} A(a) = \\ \quad \text{let} \\ \quad\quad \mathcal{F}(A,i) = \ldots \\ \quad\quad \vdots \\ \quad\quad \mathcal{F}(A,f) = Skip \\ \quad \text{within} \\ \quad\quad \mathcal{F}(A,i) \end{array}$$

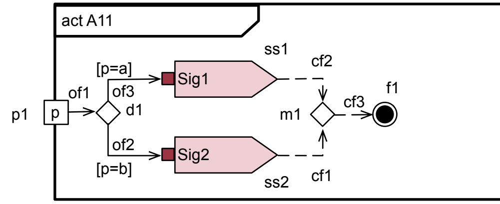


Figure 5.11: Activity A_{11}.

Activity without Initial Node

It is possible for an activity to initially behave as an activity parameter node. In this case, we assume there is no initial node, and that a single object flow connects another node with the activity parameter node. Thus, the activity initially behaves as the activity parameter node. In this case we assume a single activity parameter node with a single outgoing object flow. Let $p \in P_A$ be the lone activity parameter node, with argument a.

$$
\begin{array}{l}
A(a) = \\
\quad \text{let} \\
\quad\quad \mathcal{F}(A,p)[o] = \ldots \\
\quad\quad \vdots \\
\quad \text{within} \\
\quad\quad \mathcal{F}(A,p)[a]
\end{array}
$$

The activity parameter node behaves like the outgoing object flow:

$$of = (\mu f : OF_A \mid f \in outgoing_{of}(p))$$

$$\mathcal{F}(A,p)[o] = \mathcal{F}(A,of)[o]$$

Example 5.19. Consider Figure 5.11. The CSP process modelling activity A_{11} has a single process argument corresponding to node p_1. Execution starts from node d_1: it is connected via an object flow to p_1 and A_{11} has no initial node.

$$
\begin{array}{l}
A_{11}(p) = \\
\quad \text{let} \\
\quad\quad \mathcal{F}(A_{11},p_1)[o] = \mathcal{F}(A_{11},of_1)[o] \\
\quad\quad \mathcal{F}(A_{11},of_1)[o] = \mathcal{F}(A_{11},d_1)[o] \\
\quad\quad \mathcal{F}(A_{11},d_1)[o] = \\
\quad\quad\quad \text{if } p = a \text{ then} \\
\quad\quad\quad\quad \mathcal{F}(A_{11},of_3)[o] \\
\quad\quad\quad \text{else if } p = b \text{ then} \\
\quad\quad\quad\quad \mathcal{F}(A_{11},of_2)[o] \\
\quad\quad\quad \text{else} \\
\quad\quad\quad\quad \mathit{Stop} \\
\\
\quad\quad \vdots
\end{array}
$$

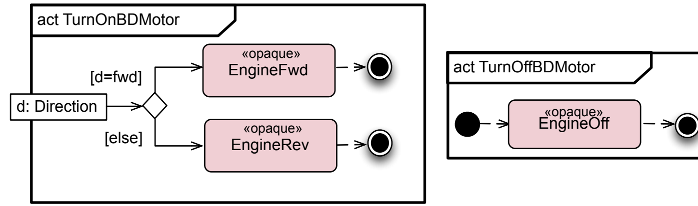


Figure 5.12: The activities TurnOnBDMotor and TurnOffBDMotor.

within
$\quad\mathcal{F}(A_{11}, p_1)[p]$

□

5.5 Example

In this section we continue our running case study. In particular, we introduce CSP process definitions for the activities that correspond to the state-based and transition-related behaviours of the robotic arm.

The activities of the bidirectional motor — TurnOnBDMotor and TurnOffBDMotor — can be modelled thus.

$$
\begin{array}{l}
\mathit{TurnOnBDMotor}(d) = \\
\quad \text{let} \\
\quad\quad \mathit{DEC}_0 = \\
\quad\quad\quad \text{if } d == \mathit{fwd} \text{ then} \\
\quad\quad\quad\quad \mathit{OA}_0 \\
\quad\quad\quad \text{else} \\
\quad\quad\quad\quad \mathit{OA}_1 \\
\quad\quad \mathit{OA}_0 = \mathit{opaque}.\mathit{enginefwd} \to F_0 \\
\quad\quad \mathit{OA}_1 = \mathit{opaque}.\mathit{enginerev} \to F_0 \\
\quad\quad F_0 = \mathit{Skip} \\
\quad \text{within} \\
\quad\quad \mathit{DEC}_0
\end{array}
$$

$$
\begin{array}{l}
\alpha \mathit{TurnOnBDMotor} = \\
\quad \{|\ \mathit{opaque}.\mathit{enginefwd}, \mathit{opaque}.\mathit{enginerev}\ |\}
\end{array}
$$

$$
\begin{array}{l}
\mathit{TurnOffBDMotor} = \\
\quad \text{let} \\
\quad\quad I_0 = \mathit{OA}_0 \\
\quad\quad \mathit{OA}_0 = \mathit{opaque}.\mathit{engineoff} \to F_0 \\
\quad\quad F_0 = \mathit{Skip} \\
\quad \text{within} \\
\quad\quad I_0
\end{array}
$$

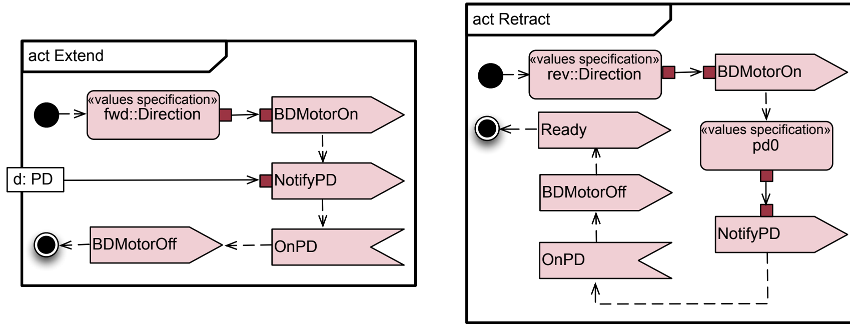


Figure 5.13: The activities Extend and Retract.

$$\alpha TurnOffBDMotor = \{\mid opaque.engineoff \mid\}$$

The processes — *Extend*, *Retract*, *Magnetise* and *Demagnetise* — modelling the activities used in the *CONTROLLER* process follow. An activity can take parameters, passed from the arguments of the triggering event of the owning state machine as input.

$$
\begin{array}{l}
Extend(local, pd) = \\
\quad \text{let} \\
\quad\quad I_0 = VS_0 \\
\quad\quad VS_0 = SS_0(fwd) \\
\quad\quad SS_0(o) = bdmotor.BDMotorOn.o \rightarrow SS_1 \\
\quad\quad SS_1 = pdmeter.NotifyPD.pd \rightarrow RS_0 \\
\quad\quad RS_0 = \\
\quad\quad\quad local.proc.OnPD \rightarrow SS_2 \\
\quad\quad\quad \Box \\
\quad\quad\quad local.disc?ev : \{\mid Grasp, Drop \mid\} \rightarrow RS_0 \\
\quad\quad SS_2 = bdmotor.BDMotorOff \rightarrow F_0 \\
F_0 = Skip \\
\quad \text{within} \\
\quad\quad I_0
\end{array}
$$

$$
\begin{array}{l}
\alpha Extend = \\
\quad \{\mid bdmotor.BDMotorOn.fwd, bdmotor.BDMotorOff, \\
\quad\quad pdmeter.NotifyPD \mid\}
\end{array}
$$

$$
\begin{array}{l}
Retract(local) = \\
\quad \text{let} \\
\quad\quad I_0 = VS_0 \\
\quad\quad VS_0 = SS_0(rev) \\
\quad\quad SS_0(o) = bdmotor.BDMotorOn.o \rightarrow VS_1 \\
\quad\quad VS_1 = SS_1(pd_0) \\
\quad\quad SS_1(o) = pdmeter.NotifyPD.0 \rightarrow RS_0 \\
\quad\quad RS_0 = \\
\quad\quad\quad local.proc.OnPD \rightarrow SS_2 \\
\quad\quad\quad \Box
\end{array}
$$

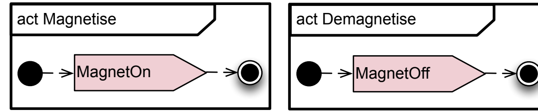


Figure 5.14: The activities Magnetise and Demagnetise.

$local.disc?ev : \{| Grasp, Drop |\} \rightarrow RS_0$
$SS_2 = bdmotor.BDMotorOff \rightarrow SS_3$
$SS_3 = client.Ready \rightarrow F_0$
$F_0 = Skip$
within
I_0

$\alpha Retract =$
$\{| bdmotor.BDMotorOn.rev, bdmotor.BDMotorOff,$
$pdmeter.NotifyPD.pd_0, client.Ready |\}$

Extend and Retract have receive signal events as actions; these receive signal events need to be passed via the event queue mechanism of the state machine. It follows that the channel used for local communication with the state machine ought to be passed in as an argument to the activity.

$Magnetise =$
let
$I_0 = SS_0$
$SS_0 = magnet.MagnetOn \rightarrow F_0$
$F_0 = Skip$
within
I_0

$\alpha Magnetise = \{| magnet.MagnetOn |\}$

$Demagnetise =$
let
$I_0 = SS_0$
$SS_0 = magnet.MagnetOff \rightarrow F_0$
$F_0 = Skip$
within
I_0
$\alpha Demagnetise = \{| magnet.MagnetOff |\}$

The rest of the activities — ActivateMagnet, DeactivateMagnet, and OnSense — can be similarly defined. We omit the CSP process definitions here in the interest of brevity. See Figure 5.15 for ActivateMagnet and DeactivateMagnet, and Figure 5.16 for OnSense.

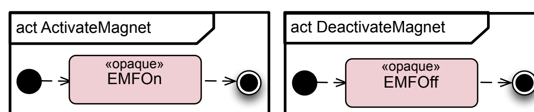


Figure 5.15: The activities ActivateMagnet and DeactivateMagnet.

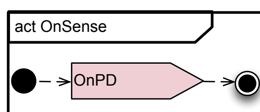


Figure 5.16: The activity OnSense.

5.6 Discussion

In this section we discuss alternative CSP representations of activities. We reflect on our formalisation by highlighting both advantages and disadvantages of our approach, and conclude the section with a critical analysis of the work presented in this chapter.

5.6.1 Alternative Formalisations

The activity diagram is the SysML behavioural formalism least often considered for formalisation. To the best of our knowledge, our approach is the first CSP formalisation to encompass both control and object flows. In the past, most formalisations either emphasised modelling simple object flows [**43**; **42**], or instead focussed on control flows [**63**; **64**].

Abdelhalim *et al.* [**43**; **42**] considers object flows within the context of fUML [**44**] activities. We extend or alter the semantics in numerous ways.

- In [**43**; **42**], no clear distinction is made between control and object flows, at least not at the level of presentation: the same graphical stereotype is used for both control and object flows, and the accompanying narrative does not place much emphasis on the distinction. In contrast, we provide a clear distinction between the different types of flows as well as which nodes they are permitted to emanate from or terminate on.

- In [**43**; **42**], send and receive signal event actions, value specification actions and activity parameter nodes are formalised. However, the authors exclude constructs such as fork and join nodes, which are considered in our work.

- Nondeterministic choice is used to model the behaviour of a decision node in [**43**; **42**]. The drawback of the approach is that CSP events need to be introduced that correspond to the respective guards. In our formalisation, we do not allow guards on control flows; guards of object flows are evaluated based on the value of the object passed.

- The work of Abdelhalim *et al.* emphasises a model-driven engineering approach: the mappings between activities and CSP are given in terms of model-to-model and

model-to-text transformation rules. The authors develop a prototypical tool that automates the transformation process between the two paradigms. A small subset of activity constructs, specified via 11 transformation rules, and an even smaller subset of state machine constructs, specified via four transformation rules, are given. The aim is then to use FDR to check for consistency between the state machine diagram and an activity diagram. Our work, in contrast, is not automated, but we formalise a greater number of constructs. In addition, the emphasis is placed on checking consistency between communicating state machines, with their respective behaviours augmented via activities.

Davies and colleagues [**47**; **48**] translate simple activities into CSP.

- The work in [**48**] is similar to ours in that the transformation process is not automated, but no formal transformation rules are given. However, their work is fundamentally different to ours in that the action nodes in their activities correspond to method calls; hence the emphasis is on the control flow between individual method calls. Constructs such as signal events, value specification actions, call behaviour actions and any notion of object flows are not considered.

- In [**47**], the concept of activities are only informally discussed and no formal model is presented. The authors mention that activities can be interpreted in one of two ways: to either specify the flow of control, or to model the flow of data. However, in both cases the authors only mention that it is possible to map such an activity to a communicating CSP process, with no CSP process descriptions given.

The work of Xu *et al.* [**63**; **64**] focus solely on mapping constructs related to control flow in activities.

- The work in [**63**] is similar to ours in the sense that the transformation rules are given in terms of syntactic structures and a mapping function in which each node is mapped to a corresponding CSP process. Emphasis is placed solely on modelling control flow; there is no mention of data flow or any formalisation of action nodes related to it. Only simple actions are considered: signal events, call behaviour actions, opaque actions and value specification actions are excluded from consideration.

- In [**64**], the work emphasises an automated approach to model checking activity diagrams. Again, only constructs related to control flow are considered.

- In both [**63**; **64**] the resulting CSP model consists of a behavioural description of a single activity. The contributions do not demonstrate any verification or refinement analytics on the translated CSP model.

5.6.2 Reflections

We briefly outline the advantages and biggest criticisms of our approach and contend that, from the stance of formal verification, our biggest weaknesses are also our biggest strengths. We will argue the case by presenting a narrative of how inclusion of certain constructs would either convolute our semantics, or would make the refinement checking of the resulting formalisms computationally intractable.

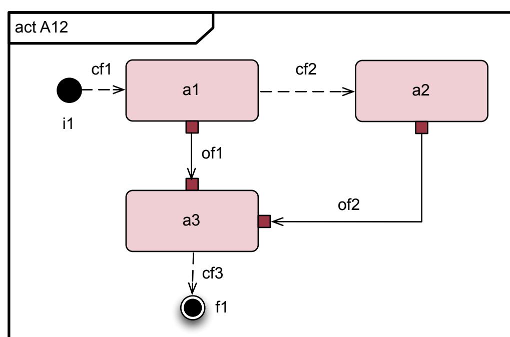


Figure 5.17: Activity A_{12}.

Advantages

The biggest advantage of our approach is that it provides a clear, formal mapping between activities and CSP. The resulting CSP representation is computationally tractable and offers a reasonable level of scalability. However, we readily acknowledge that the state space explosion problem will always eventually be an ultimate curb on the size and complexity of systems we can reasonably expect to model. In addition we provided a semantics that is inherently integrated with state machines — a key construct frequently used to supplement state-based or transition-related behaviours.

As far as we are aware, this is the first formalisation that considers data as well as control flow within activities. For example, the formalisations of Xu *et al.* [**63**; **64**] focussed exclusively on control flows. Furthermore, the contributions [**63**; **64**] only consider a limited subset of constructs.

Disadvantages

The biggest drawback of our approach is perhaps the fact that we place a restriction on the outgoing control and object flows: we allow actions to only have either outgoing object flows, or outgoing control flows, but not both. The argument for supporting a less restrictive form of control flow can be made; however, we have opted to define semantics that are computationally feasible, yet sufficiently powerful. Moreover, the activities in this thesis, and our modelling methodology, are typically used within the context of state machines. We argue that the control and object flow semantics defined here are sufficiently powerful to model complex state or transition-based behaviours. Constructs like activity partitions and data store nodes are excluded based on the observation that these are unlikely to be used in activities within this context.

Critical Discussion

The aforementioned approaches [**43**; **42**; **47**; **48**; **63**; **64**] do not consider in their formalisation the environment external to the activity, or define the execution semantics of multiple communicating activities. In our formalisation, the context of execution of an activity is its parent state machine. It follows that our activities execute under the run-to-completion semantics of SysML state machines. In [**77**] Eshuis explores two translations of activity diagrams with regards to their external execution semantics, based on alternative seman-

tics for state machines: the semantics as proposed by Harel [**31**] and the semantics of the OMG [**1**; **2**]. Activities are formalised in terms of the input language of the model checker NuSMV [**38**].

We provided a comprehensive treatment of control and data flow modelling in SysML activities. However, some of the omitted or partially treated constructs can be justified as follows.

- Activity partitions are not considered as the inclusion of these would provide no additional benefit. Our activities execute within the context of its associated state machine, and thus all actions specified per activity executes within the context of the owning part. Send and receive signal event actions have target and source pins that identify the destination and origin of the signal event.

- More complex fork and join structures are possible, but we opted for a simpler and more elegant semantics.

- More complex data flow semantics could have been included, but again these would have made the model checking computationally intractable. For example, if we had lifted the restriction that prohibits a node from having both a control and data flow path exiting the action node, it would be possible to model behaviour as depicted in Figure 5.17. However, this would involve defining a blocking semantics where an action node blocks until the minimum number of tokens on every input pin is reached, in addition to receiving a control token if an incoming control flow is present. In Figure 5.17, a possible solution would be to utilise the parallel composition of the processes modelling actions a_1, a_2 and a_3. However, this would again have an adverse effect on the overall state space, considering that the activities execute in the context of communicating state machines, that are themselves composed in parallel.

- Our formalisation, as it stands, only considers inputs via activity parameters nodes. Activities can produce outputs as well, and in SysML these are placed on activity parameter output nodes, after the activity has finished execution. CSP process arguments are typically used to record process state, and the language of machine-readable CSP is declarative. It follows that the notion of activity output parameters does not sit well with the parameter passing mechanism of CSP. Thus, if we wanted to model the passage of an object from one activity parameter node to the next, we would have to do so utilising CSP channels[2]. We have chosen not to do this, opting for a simpler semantics: considering only activity parameter nodes that serve as inputs allows us to model the parameter node as the process argument. Moreover, considering the execution context of our activities, it is unlikely that outputs are required: it would make little sense for state-based or transition-related behaviours to have outputs.

[2]For example, a call behaviour action that passes the output of the called activity to the input of the next called behaviour via an object flow.

5.6.3 Possible Extensions

The SysML specification [**1**] extends UML activities [**2**] with probabilistic constructs. Lowe [**114**; **115**] extended conventional CSP with probabilistic features. In [**115**] a probabilistic choice construct is introduced and a prototypical version of FDR presented that utilises the model checker PRISM [**116**]. Other possibilities include using the alternative CSP model checker PAT [**26**], for which a probabilistic toolset exists [**117**].

Clearly it is possible to derive a semantics for the features we deliberately ignored, but the benefits of doing this would have to be weighed against more convoluted CSP descriptions that might not sufficiently scale to structures that are computationally tractable.

5.7 Conclusion

In this chapter we provided a practical CSP formalisation for SysML activities. The novelty of the approach is that we considered activities within their context of execution: the SysML state machine. There exists a close coupling between these two behavioural formalisms, and the integration of such behaviours is a complex task. We provided a powerful yet elegant formalisation for activities within this context. In the next chapter we formalise interactions: a behavioural construct used to specify typical interaction scenarios between blocks.

6

INTERACTIONS

In this chapter we provide a formal behavioural semantics for SysML interactions using CSP. The structure of this chapter mirrors that of Chapters 4 and 5. First, we introduce the restrictions and constraints imposed in order to ensure a sensible behavioural semantics. A structural mathematical model and modelling constructs are introduced, followed by a template-based semantics. We then apply the formalisation to our running case study to illuminate the approach and to serve as a proof of concept. Next, we critically evaluate the theory proposed and list some of the advantages and potential shortcomings of the results. The work presented in this chapter previously appeared in [**118**].

6.1 Assumptions

We restrict the constructs permitted in our interactions as follows.

- We only consider asynchronous messages in our formalisation; synchronous messages are excluded. This is due to the fact that we focus solely on the sending and receiving of signals, which are asynchronous.
- We do not consider activation bars on lifelines.

We assume the following well-formedness constraint.

- The temporal order in which message exchanges appear between all participating lifelines in an interaction is unambiguous: message exchanges are positioned such that the order between any two observations (the sending or receiving of a message) occurring on any two lifelines is clear.

6.2 Abstract Syntax

In this section we provide formal definitions in order to conveniently describe the behavioural aspects relating to sequence diagrams.

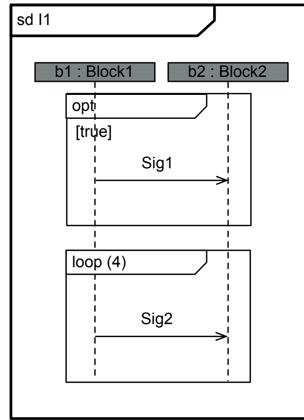


Figure 6.1: Interaction I_1.

In SysML an interaction executes within the context of its owning block, and specifies the interaction between parts [**14**]. On the diagram, lifelines correspond to the parts. A sequence diagram depicts this interaction graphically. An asynchronous message is represented using a solid line from the sending lifeline to the receiving lifeline. Note that during an asynchronous exchange the sending lifeline does not block. When an interaction executes, it produces a sequence of interaction occurrences, termed a *trace*.

We consider an interaction, I, to be a quadtuple of the form (L_I, S_I, M_I, O_I), where:

- L_I denotes the set of lifelines participating in the interaction, I;
- S_I denotes the set of signal events;
- $M_I : ID \nrightarrow S_I$ is a function that uniquely identifies the asynchronous messages of an interaction and associates a message with the signal that typed it; and
- $O_I \subseteq L_I \times \text{seq}\,(ID \times \{snd, rcv\})$ describes all interaction occurrences as a set of pairs, with the first element being the lifeline, and the second a sequence of occurrence observations in the order that they appear on the particular lifeline.

We define the following auxiliary functions:

- $sd : ID \nrightarrow L_I$ returns, for a message identifier, the sending lifeline;
- $rv : ID \nrightarrow L_I$ returns, for a message identifier, the receiving lifeline; and
- $occurrences : L_I \nrightarrow \text{seq}\,(ID \times \{snd, rcv\})$ denotes the sequence of event occurrences on the argument lifeline in temporal order.

Example 6.1. Consider Figure 6.1. We have the following mathematical structures describing the optional fragment.

$$L_{opt} = \{b_1, b_2\}$$
$$S_{opt} = \{Sig_1\}$$
$$M_{opt} = \{(1, Sig_1)\}$$
$$O_{opt} = \{(b_1, \langle(1, snd)\rangle), (b_2, \langle(1, rcv)\rangle)\}$$

□

6.3 Modelling Constructs

We introduce additional modelling constructs in this section in order to enhance the clarity of our presentation.

6.3.1 Occurence Observations

Sequence diagrams facilitate the modelling of interactions between structural constructs as sequences of temporal occurrences, termed *occurrence observations*. As mentioned in Section 6.1, we only consider asynchronous message exchanges. Every occurrence observation considered in this chapter is therefore either:

- an asynchronous message being sent; or
- an asynchronous message being received.

6.3.2 Operators

Several operators exist for SysML interactions.

- A *sequencing operator* prescribes whether the sequencing of participating lifelines are weak or strict (partial or total order).
- An *interaction operator* alters the behaviour of the prescribed sequence (enclosed within the corresponding combined fragment). *Combined fragments* allow for the description of complex patterns of interaction in a concise and compact manner. SysML defines different interaction operators, each enabling the specification of different rules with regards to the ordering of messages (and their associated occurrence observations). A combined fragment is an *interaction* operator with associated *operands*.
- An *interpretation operator* alters our interpretation of the trace.

6.3.3 Template Approach

Our approach for translating sequence diagrams to CSP is based on mirroring the structure of the corresponding diagram. Broadly speaking, each lifeline is mapped to a process, and each occurrence observation is mapped to a CSP event. The process then enforces weak sequencing by insisting that the occurrence observations appear in the temporal order specified on the corresponding lifeline. The acts of sending and receiving a message are completely detached in our model. We therefore require an additional constraint process to enforce the fact that a message cannot be received before it was sent.

We treat the various interaction operators of sequence diagrams using template processes that describe their respective patterns of behaviour. These are defined formally in Section 6.4.

Consider an interaction, I, with a corresponding sequence diagram. Our approach can be outlined as follows.

- With each lifeline, $l \in L_I$, we associate a sequence of events of the same temporal order. The sequence of events is given by the function $occurrences\,(l)$. An element of this sequence is a pair of the form (id, obs), where $id \in ID$ and $obs \in \{snd, rcv\}$.
- Model each occurrence observation with a corresponding CSP event. Thus, for asynchronous messages, we have $asynch.obs.sd(id).rv(id).M_I(id)$.
- Each lifeline has a corresponding CSP process that communicates the events in the required order (defined in the template process).
- For each message in an interaction, we associate a triple of the form $(from, to, name)$, where $\{from, to\} \subseteq L_I$ and $name \in S_I$.
- Each message has an associated process with send and receive occurrence events that synchronise with the appropriate sending and receiving lifelines (defined in the template process).
- Messages where arguments are passed as part of the exchange are generalised by means of CSP channels.
- Depending on the interaction, instantiate the correct template process (as defined in the next section) to describe the behaviour.
- A sequence diagram that consists of more than one interaction operator is subsequently defined as the sequential composition of the CSP template processes that describe the respective interaction operators.

Example 6.2. Consider Figure 6.1. In order to describe the behaviour of interaction I_1, we require the sequential composition of the processes modelling the optional and loop fragments. Assuming that *OPT* is the CSP processes describing the optional fragment, and *LOOP* is the process that behaves like the loop fragment, we have:

$$I_1 = OPT \fatsemi LOOP$$

□

6.3.4 Process Definitions

CSP process definitions that are reused in the remainder of this chapter are defined in this section.

PrefixComposition

The order of the occurrence observations along any one lifeline is significant. The process *PrefixComposition*, if supplied a sequence as input, is the process that communicates the events in order and then behaves as *Skip*. Given a temporal sequence of interaction occurrences for a lifeline, we define *PrefixComposition* as follows:

$PrefixComposition(s) =$
  if $null(s)$ then

$$
\begin{array}{l}
\quad \mathit{Skip} \\
\text{else} \\
\quad \mathit{head}(s) \rightarrow \mathit{PrefixComposition}(\mathit{tail}(s))
\end{array}
$$

Message

The process *Message* asserts the behaviour that the sending of a message necessarily occurs before its reception. In the following: the parameter *id* corresponds to the type *ID*; *from* and *to* model the sending and receiving lifelines; and *name* corresponds the signal (an instance of an event type drawn from the set $\mathcal{S}$).

$$
\begin{array}{l}
\mathit{Message}(\mathit{id}, \mathit{from}, \mathit{to}, \mathit{name}) = \\
\quad \mathit{msg.id.snd.from.to.name} \rightarrow \mathit{msg.id.rcv.from.to.name} \rightarrow \mathit{Skip}
\end{array}
$$

The function that generates the alphabet for a message can be defined as follows:

$$
\begin{array}{l}
\alpha\, \mathit{Message}(\mathit{id}, \mathit{from}, \mathit{to}, \mathit{name}) = \\
\quad \{\mid \mathit{msg.id.snd.from.to.name}, \mathit{msg.id.rcv.from.to.name} \mid\}
\end{array}
$$

Messages

The process *Messages* is the parallel composition of all the *Message* processes. Each *Message* process takes a quintuple of the form $(\mathit{id}, \mathit{from}, \mathit{to}, \mathit{name})$ as input.

$$
\begin{array}{l}
\mathit{Messages}(\mathit{messages}) = \\
\quad \| (\mathit{id}, \mathit{from}, \mathit{to}, \mathit{name}) : \mathit{messages} \bullet \\
\quad\quad [\alpha\, \mathit{Message}(\mathit{id}, \mathit{from}, \mathit{to}, \mathit{name})]\, \mathit{Message}(\mathit{id}, \mathit{from}, \mathit{to}, \mathit{name})
\end{array}
$$

The definition above makes use of the definitions of *Message* and $\alpha\, \mathit{Message}$.

The alphabet of process *Messages* is defined thus:

$$
\begin{array}{l}
\alpha\, \mathit{Messages}(\mathit{messages}) = \\
\quad \bigcup \{ (\mathit{id}, \mathit{from}, \mathit{to}, \mathit{name}) : \mathit{messages} \bullet \alpha\, \mathit{Message}(\mathit{id}, \mathit{from}, \mathit{to}, \mathit{name}) \}
\end{array}
$$

Lifelines

We define the process *Lifelines* to model the parallel composition of a set of lifelines. The process takes as input a set of sequences, where each sequence describes occurrence specifications for a lifeline in temporal order. Each lifeline in the composition synchronises on its entire alphabet. In the following, the function *set* converts a sequence to a set.

$$
\begin{array}{l}
\mathit{Lifelines}(\mathit{lifelines}) = \\
\quad \| \ \mathit{line} : \mathit{lifelines} \bullet [\mathit{set}(\mathit{line})] \mathit{PrefixComposition}(\mathit{line})
\end{array}
$$

The alphabet for the *Lifelines* process is given below. Again, the input set contains the sequences describing occurrence specifications per lifeline:

$$
\alpha\, \mathit{Lifelines}(\mathit{lifelines}) = \bigcup \{ \mathit{line} : \mathit{lifelines} \bullet \mathit{set}(\mathit{line}) \}
$$

6.4 Behavioural Semantics

We are now able to model the sequencing and interaction operators of SysML interactions.

6.4.1 Sequencing Operators

SysML defines two types of sequencing: weak and strict. In this section, we provide CSP definitions that restrict occurrence observations according to one of the aforementioned semantics.

Weak Sequencing Operator

Interaction occurrences appear in temporal order on a lifeline, with time progressing downwards. An interaction implicitly imposes an order on the messages sent between lifelines, termed *weak sequencing*. Weak sequencing is the SysML default and implies the following [**1**; **2**].

1. The ordering of occurrence specifications within each of the messages are maintained. Thus, a message needs to be sent before it can be received.

2. Occurrence specifications on different lifelines from different messages may come in any order. Thus, occurrence specifications between different lifelines (also between different messages) impose no additional ordering constraints upon each other.

3. Occurrence specifications on the same lifeline from different messages are ordered such that an occurrence specifications of the first message comes before that of the second message. Thus, the temporal order of the occurrence specifications on each lifeline must be honoured.

We can model weak sequencing as follows. By placing *Messages* and *Lines* in parallel, we restrict the traces to adhere to the behaviours imposed by the first and last condition. Condition 2 places no further restrictions on the behaviour, and as the interaction occurrences between different lifelines do not have any shared events in common, we require no process to model this behaviour.

The CSP process *Seq* models weak sequencing and is defined thus.

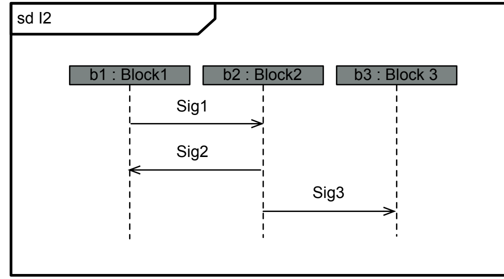


Figure 6.2: Interaction I_2.

$$
\begin{aligned}
& Seq(lifelines, messages) = \\
& \quad let \\
& \quad\quad Lines = Lifelines(lifelines) \\
& \quad\quad Msgs = Messages(messages) \\
& \quad\quad \alpha\, Lines = \alpha\, Lifelines(lifelines) \\
& \quad\quad \alpha\, Msgs = \alpha\, Messages(messages) \\
& \quad within \\
& \quad\quad Lines\ [\ \alpha\, Lines \parallel \alpha\, Msgs\ ]\ Msgs
\end{aligned}
$$

Weak sequencing semantics are explicitly denoted by *seq*.

Example 6.3. Consider Figure 6.2 and interaction I_2. No sequencing operator is specified, so weak sequencing semantics applies: after the send observation of Sig_2 is observed, Sig_2 can either be received by b_1, or the send observation of Sig_3 can be observed. This is exactly the behaviour of the *SEQ* process. Note that we make use of *Seq*, the template process for weak sequencing.

$$
\begin{aligned}
L = \{ & \langle msg.1.snd.b_1.b_2.Sig_1, msg.2.rcv.b_2.b_1.Sig_2 \rangle, \\
& \langle msg.1.rcv.b_1.b_2.Sig_1, msg.2.snd.b_2.b_1.Sig_2, msg.3.snd.b_2.b_3.Sig_3 \rangle, \\
& \langle msg.3.rcv.b_2.b_3.Sig_3 \rangle \} \\
M = \{ & (1, b_1, b_2, Sig_1), (2, b_2, b_1, Sig_2), (3, b_2, b_3, Sig_3) \} \\
SEQ = & \ Seq(L, M)
\end{aligned}
$$

□

Strict Sequencing Operator

Strict sequencing semantics, denoted by *strict*, impose an additional constraint upon weak sequencing, in that the messages must be sequenced across all participating lifelines [**14**]. We can subsequently define strict sequencing by placing another process, *EnforceStrict*, in parallel to constrain the behaviour of weak sequencing.

The process *EnforceStrict* can be defined thus. The parameter *obs* is a sequence of occurrence observations across all participating lifelines, arranged according to the semantics of strict sequencing.

$$EnforceStrict(obs) = PrefixComposition(obs)$$

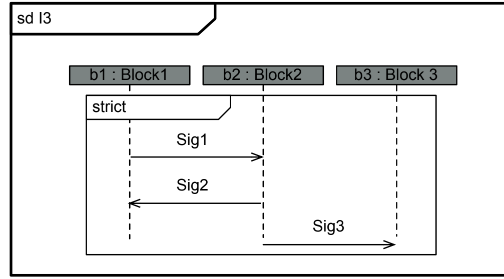


Figure 6.3: Interaction I_3.

The process *Strict* is then defined as follows.

$$
\begin{aligned}
& Strict(lifelines, messages, obs) = \\
& \quad let \\
& \quad\quad Lines = Lifelines(lifelines) \\
& \quad\quad Msgs = Messages(messages) \\
& \quad\quad \alpha\, Lines = \alpha\, Lifelines(lifelines) \\
& \quad\quad \alpha\, Msgs = \alpha\, Messages(messages) \\
& \quad within \\
& \quad\quad (Lines \,[\, \alpha\, Lines \parallel \alpha\, Msgs \,]\, Msgs) \\
& \quad\quad\quad [|\alpha\, Msgs|] \\
& \quad\quad EnforceStrict(obs)
\end{aligned}
$$

Example 6.4. Consider Figure 6.3 and interaction I_3. The messages, lifelines and order of exchanges are exactly the same as that of interaction I_2. This time, however, strict sequencing semantics applies. The process *STRICT* models the only possible observation sequence allowed: the send and receive observations of Sig_1; the send and receive observations of Sig_2; and the send and receive observations of Sig_3.

$$
\begin{aligned}
L = \{ & \langle msg.1.snd.b_1.b_2.Sig_1, msg.2.rcv.b_2.b_1.Sig_2 \rangle, \\
& \langle msg.1.rcv.b_1.b_2.Sig_1, msg.2.snd.b_2.b_1.Sig_2, msg.3.snd.b_2.b_3.Sig_3 \rangle, \\
& \langle msg.3.rcv.b_2.b_3.Sig_3 \rangle \} \\
M = \{ & (1, b_1, b_2, Sig_1), (2, b_2, b_1, Sig_2), (3, b_2, b_3, Sig_3) \} \\
O = \langle & msg.1.snd.b_1.b_2.Sig_1, msg.1.rcv.b_1.b_2.Sig_1, msg.2.snd.b_2.b_1.Sig_2, \\
& msg.2.rcv.b_2.b_1.Sig_2, msg.3.snd.b_2.b_3.Sig_3, msg.3.rcv.b_2.b_3.Sig_3 \rangle \\
STRICT & = Strict(L, M, O)
\end{aligned}
$$

□

6.4.2 Interaction Operators

In this section we define the interaction operators of SysML in terms of CSP. Note that the operands of an interaction operator are dependent upon the type of the operator: the alternative and parallel operator each have multiple horizontal partitions, separated by dashed lines that correspond to their operands; the loop operator, on the other hand, has

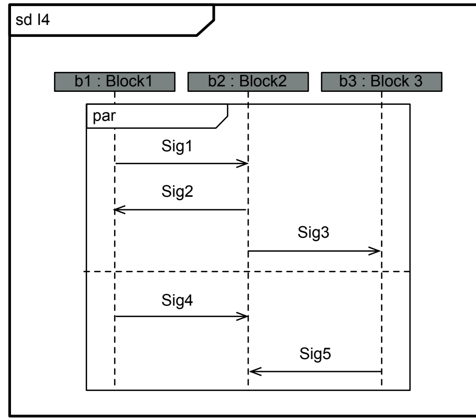


Figure 6.4: Interaction I_4.

just a single partition [**14**]. For single partition operators, their operands correspond to the messages enclosed in the combined fragment.

Parallel Interaction Operator

The parallel operator, *par*, designates an interleaving between its operands. The horizontal partitions (within the combined fragment) correspond to the operands. The operator designates that the combined fragment represents a parallel merge between the behaviours of the operands. The occurrence specifications of the different operands can be interleaved in any way as long as the ordering imposed by each operand as such is preserved [**1**; **2**]. In the following definition, we assume a weak sequencing between the messages of each operand.

The interleaving operator of CSP models this pattern of behaviour perfectly. We therefore define the parallel interaction operator as the interleaved behaviour of two sequentially interleaved processes. The definition below assumes that there are only two partitions (for reasons of readability) within the combined fragment. We can, however, easily extend this to cover more partitions, or even generalise the definition to cover an arbitrary number of horizontal partitions.

$$
\begin{array}{l}
Par(lifelines_1, messages_1, lifelines_2, messages_2) = \\
\quad let \\
\quad\quad Seq_1 = Seq(lifelines_1, messages_1) \\
\quad\quad Seq_2 = Seq(lifelines_2, messages_2) \\
\quad within \\
\quad\quad Seq_1 \mathbin{|||} Seq_2
\end{array}
$$

Example 6.5. Figure 6.4 depicts interaction I_4: the interleaving of Sig_1, Sig_2 and Sig_3, with Sig_4 and Sig_5. The template process *Par* can be used to model this behaviour.

$$
\begin{array}{rl}
L_1 = \{ & \langle msg.1.snd.b_1.b_2.Sig_1, msg.2.rcv.b_2.b_1.Sig_2\rangle, \\
& \langle msg.1.rcv.b_1.b_2.Sig_1, msg.2.snd.b_2.b_1.Sig_2, msg.3.snd.b_2.b_3.Sig_3\rangle, \\
& \langle msg.3.rcv.b_2.b_3.Sig_3\rangle\}
\end{array}
$$

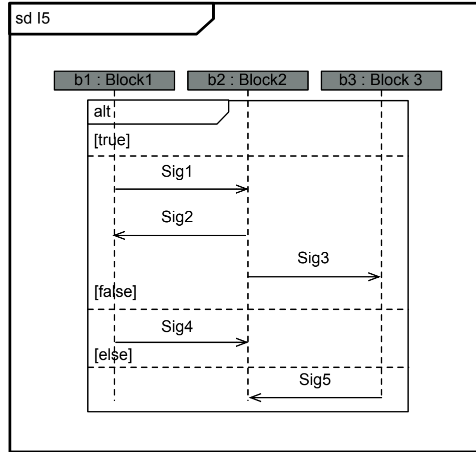


Figure 6.5: Interaction I_5.

$$\begin{aligned}
M_1 &= \{(1, b_1, b_2, Sig_1), (2, b_2, b_1, Sig_2), (3, b_2, b_3, Sig_3)\} \\
L_2 &= \{\langle msg.4.snd.b_1.b_2.Sig_4\rangle, \\
&\quad \langle msg.4.rcv.b_1.b_2.Sig_4, msg.5.rcv.b_3.b_2.Sig_5\rangle, \\
&\quad \langle msg.5.snd.b_3.b_2.Sig_5\rangle\} \\
M_2 &= \{(4, b_1, b_2, Sig_4), (5, b_3, b_2, Sig_5)\} \\
PAR &= Par(L_1, M_1, L_2, M_2)
\end{aligned}$$

□

Alternative Interaction Operator

The alternative operator, *alt*, offers the choice between the behaviours of its operands, based on the guard condition associated with each partition. Recall that the horizontal partitions (within the combined fragment) correspond to the operands. In the scenario where more than one guard evaluates to true, the choice is nondeterministic. If none of the guards evaluate to true, an optional else partition is selected [**14**].

We can use the nondeterministic and conditional choice constructs of CSP to model this behaviour. Below we provide a definition for a combined fragment consisting of two conditionally guarded partitions and one else clause. This definition can be generalised to handle an arbitrary number of conditional clauses, but a simplified version is presented here to illustrate the concepts.

$$\begin{aligned}
& Alt(lifelines_1, messages_1, guard_1, lifelines_2, messages_2, guard_2, \\
& \quad lifelines_3, messages_3) = \\
& \quad let \\
& \quad\quad Seq_1 = Seq(lifelines_1, messages_1) \\
& \quad\quad Seq_2 = Seq(lifelines_2, messages_2) \\
& \quad\quad Seq_3 = Seq(lifelines_3, messages_3) \\
& \quad within \\
& \quad\quad \text{if } (guard_1 \wedge guard_2) \text{ then} \\
& \quad\quad\quad Seq_1 \sqcap Seq_2
\end{aligned}$$

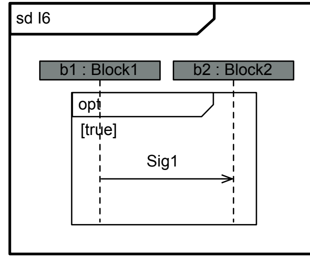


Figure 6.6: Interaction I_6.

elseif $guard_1$ then
  Seq_1
elseif $guard_2$ then
  Seq_2
else
  Seq_3

Example 6.6. Figure 6.5 depicts interaction I_5, in which we have simplified the Boolean guards to simplify the demonstration. The template process *Alt* models this behaviour.

$$
\begin{aligned}
L_1 &= \{\langle msg.1.snd.b_1.b_2.Sig_1, msg.2.rcv.b_2.b_1.Sig_2\rangle,\\
&\quad \langle msg.1.rcv.b_1.b_2.Sig_1, msg.2.snd.b_2.b_1.Sig_2, msg.3.snd.b_2.b_3.Sig_3\rangle,\\
&\quad \langle msg.3.rcv.b_2.b_3.Sig_3\rangle\}\\
M_1 &= \{(1, b_1, b_2, Sig_1), (2, b_2, b_1, Sig_2), (3, b_2, b_3, Sig_3)\}\\
L_2 &= \{\langle msg.4.snd.b_1.b_2.Sig_4\rangle,\\
&\quad \langle msg.4.rcv.b_1.b_2.Sig_4\rangle\}\\
M_2 &= \{(4, b_1, b_2, Sig_4)\}\\
L_3 &= \{\langle msg.5.rcv.b_3.b_2.Sig_5\rangle,\\
&\quad \langle msg.5.snd.b_3.b_2.Sig_5\rangle\}\\
M_3 &= \{(5, b_3, b_2, Sig_5)\}\\
ALT &= Alt(L_1, M_1, true, L_2, M_2, false, L_3, M_3)
\end{aligned}
$$

□

Optional Interaction Operator

The optional operator, *opt*, is used to express optional behaviour. The operand (messages contained within the combined fragment) is only executed if the guard condition is true. This behaviour is exactly that of an alternative operator with a single operand.

$Opt(lifelines, messages, guard) =$
  let
    $Seq = Seq(lifelines, messages)$
  within
    if $(guard)$ then
      Seq

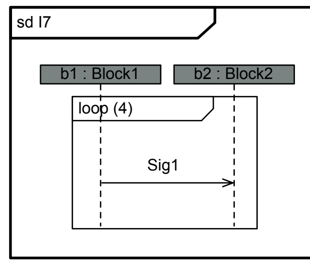


Figure 6.7: Interaction I_7.

else
  Skip

Example 6.7. Figure 6.6 depicts the optional operator. The template process *Opt* models optional behaviour.

$$\begin{aligned} L = \{&\langle msg.1.snd.b_1.b_2.Sig_1\rangle, \\ &\langle msg.1.rcv.b_1.b_2.Sig_1\rangle\} \end{aligned}$$
$$M = \{(1, b_1, b_2, Sig_1)\}$$
$$OPT = Opt(L, M, true)$$

□

Loop Interaction Operator

The *loop* operator repeats its operand (the messages contained within the combined fragment) until the termination condition imposed upon it is satisfied.

The semantics of the loop operator allows for the termination condition to be expressed as either: an iteration bound (of the form $(lower, upper)$ or $(exact)$); a Boolean condition; or a combination of both. In practice, however, one would use one or the other, rather than a combination.

In addition, the specification [**1**; **2**] is ambiguous with regards to the semantics when the termination condition is expressed as a combination of an iteration bound and Boolean guard: it is unclear what happens if the Boolean condition evaluates to false before the minimum number of iterations have executed. This ambiguity arises as a result of the following two quotes from the UML specification.

> *"After the minimum number of iterations have executed and the Boolean expression is false the loop will terminate"* [**2**].

> *"The loop will only continue if that specification evaluates to true during execution regardless of the minimum number of iterations specified in the loop"* [**2**].

Given the above, we consider in our treatment only the cases where either an iteration bound or Boolean guard is specified.

The sequencing operator of CSP is used to express behaviour as a sequence of process executions. We can convey the desired behaviour of the loop operator through successive

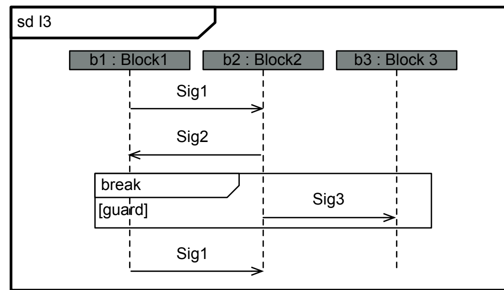


Figure 6.8: Interaction I_8.

applications of the sequencing operator (to the CSP process modelling the behaviour of the operand) in accordance with the stated termination condition.

Consider, for example, the case where a single integer bound is specified as the termination condition. The process *Loop* models this:

$$
\begin{array}{l}
\mathit{Loop}(\mathit{lifelines}, \mathit{messages}, \mathit{exact}) = \\
\quad \mathit{let} \\
\qquad \mathit{Seq}_1 = \mathit{Seq}(\mathit{lifelines}, \mathit{messages}) \\
\quad \mathit{within} \\
\qquad \text{if } (\mathit{exact} \geq 1) \text{ then} \\
\qquad\quad \mathit{Seq}_1 \,\text{;}\, \mathit{Loop}(\mathit{lifelines}, \mathit{messages}, \mathit{exact} - 1) \\
\qquad \text{else} \\
\qquad\quad \mathit{Skip}
\end{array}
$$

Example 6.8. Figure 6.7 depicts the loop operator, with the expectation that b_1 will send Sig_1 to b_2 four times. The template process *Loop* models the looping behaviour.

$$
\begin{array}{l}
L = \{\langle msg.1.snd.b_1.b_2.Sig_1\rangle, \\
\qquad \langle msg.1.rcv.b_1.b_2.Sig_1\rangle\} \\
M = \{(1, b_1, b_2, Sig_1)\} \\
LOOP = Loop(L, M, 4)
\end{array}
$$

□

Break Interaction Operator

The break operator, *break*, is used to model a breaking scenario from another enclosing fragment. The behavioural semantics is such that if the guard associated with the break evaluates to true, then its operand is executed (rather than the remainder of the enclosing fragment). This is often used to represent the handling of exceptional scenarios.

For example, consider a break nested within an enclosing *seq* fragment, which we model in terms of the process *Break*. The first two parameters ($lifelines_{pre}$ and $messages_{pre}$) describe the lifelines and messages of the enclosing fragment preceding the break; the final two parameters ($lifelines_{pst}$ and $messages_{pst}$) model the remainder of the enclosing behaviour. The *lifelines*, *messages* and *grd* parameters correspond to the operands of the break fragment.

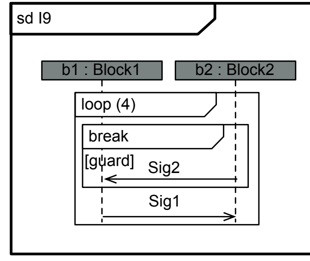


Figure 6.9: Interaction I_9.

$$
\begin{aligned}
&Break(lifelines_{pre}, messages_{pre}, lifelines_{break}, messages_{break}, guard,\\
&\quad lifelines_{post}, messages_{post}) =\\
&\; let\\
&\quad Seq_{pre} = Seq(lifelines_{pre}, messages_{pre})\\
&\quad Seq_{break} = Seq(lifelines_{break}, messages_{break})\\
&\quad Seq_{post} = Seq(lifelines_{post}, messages_{post})\\
&\; within\\
&\quad Seq_{pre} \fatsemi (\text{if } guard \text{ then } Seq_{break} \text{ else } Seq_{post})
\end{aligned}
$$

Example 6.9. Figure 6.8 depicts interaction I_8 that models a breaking scenario. The instantiated CSP process *BREAK* models the behaviour shown.

$$
\begin{aligned}
&L_{pre} = \{\langle msg.1.snd.b_1.b_2.Sig_1, msg.2.rcv.b_2.b_1.Sig_2\rangle,\\
&\qquad \langle msg.1.rcv.b_1.b_2.Sig_1, msg.2.snd.b_2.b_1.Sig_2\rangle\}\\
&M_{pre} = \{(1, b_1, b_2, Sig_1), (2, b_2, b_1, Sig_2)\}\\
&L = \{\langle msg.3.snd.b_2.b_3.Sig_3\rangle,\\
&\qquad \langle msg.3.rcv.b_2.b_3.Sig_3\rangle\}\\
&M = \{(3, b_2, b_3, Sig_3)\}\\
&L_{pst} = \{\langle msg.4.snd.b_1.b_2.Sig_1\rangle,\\
&\qquad \langle msg.4.rcv.b_1.b_2.Sig_1\rangle\}\\
&M_{pst} = \{(4, b_1, b_2, Sig_1)\}\\
&BREAK = Break(L_{pre}, M_{pre}, L, M, guard, L_{pst}, M_{pst})
\end{aligned}
$$

□

The scenario where the break operator is used to break from an enclosing loop fragment, as per Figure 6.9, can be formalised thus. In the following, i denotes the number of iterations.

$$
\begin{aligned}
&Break(lifelines_{loop}, messages_{loop}, i, lifelines_{break}, messages_{break}, guard) =\\
&\; let\\
&\quad Seq_{loop} = Seq(lifelines_{loop}, messages_{loop})\\
&\quad Seq_{break} = Seq(lifelines_{break}, messages_{break}) \quad within\\
&\quad \text{if } guard \text{ then}\\
&\qquad Seq_{break}\\
&\quad \text{elseif } (i \geq 1) \text{ then}
\end{aligned}
$$

$$
\begin{array}{l}
\quad Seq_{loop} \\
\quad\quad ; \\
\quad Break(lifelines_{loop}, messages_{loop}, i-1, lifelines_{break}, messages_{break}, guard) \\
\text{else} \\
\quad Skip
\end{array}
$$

In the above we model the break operator, as well as the enclosing loop fragment. As soon as the guard evaluates to true the breaking sequence executes.

Critical Interaction Operator

The critical region operator, *critical*, designates that the operands operate in a critical region, where the traces within the region cannot be interleaved with any other occurrence specifications that may occur on the lifelines that are contained within the critical region [**2**]. This operator is frequently nested within an enclosing *par* fragment; the critical fragment then restricts the behaviour of the enclosing fragment by insisting that its operands are treated atomically. In the following, cs_1 and cs_2 refer to the observations, as a temporally ordered sequence, on the lifeline enclosed by the critical fragments. The local process *EnforceCritical* restricts the interleaved behaviour according to the observations of the critical regions.

$$
\begin{array}{l}
Critical(lifelines_1, messages_1, lifelines_2, messages_2, cs_1, cs_2) = \\
\quad let \\
\quad\quad Seq_1 = Seq(lifelines_1, messages_1) \\
\quad\quad Seq_2 = Seq(lifelines_2, messages_2) \\
\quad\quad EnforceCritical_1 = PrefixComposition(cs_1) \\
\quad\quad EnforceCritical_2 = PrefixComposition(cs_2) \\
\quad\quad EnforceCritical = EnforceCritical'_1 \;\Box\; EnforceCritical'_2 \\
\quad\quad EnforceCritical'_1 = EnforceCritical_1 \,;\, EnforceCritical_2 \\
\quad\quad EnforceCritical'_2 = EnforceCritical_2 \,;\, EnforceCritical_1 \\
\quad\quad \alpha\, Critical = \alpha\, Lifelines(cs) \\
\quad within \\
\quad\quad (Seq_1 \;|||\; Seq_2) \\
\quad\quad\quad [|\alpha\, Critical|] \\
\quad\quad EnforceCritical
\end{array}
$$

Example 6.10. Figure 6.10 depicts interaction I_{10} that showcases the critical operator. We have two critical regions: the receive observation of Sig_1 and the send observation of Sig_2 needs to be executed atomically; the same holds for the receive observation of Sig_3 and send observation of Sig_4.

$$
\begin{array}{l}
L_1 = \{\langle msg.1.snd.b_1.b_2.Sig_1\rangle, \\
\quad\quad \langle msg.1.rcv.b_1.b_2.Sig_1, msg.2.snd.b_2.b_3.Sig_2\rangle, \\
\quad\quad \langle msg.2.rcv.b_2.b_3.Sig_2\rangle\} \\
M_1 = \{(1, b_1, b_2, Sig_1), (2, b_2, b_3, Sig_2)\}
\end{array}
$$

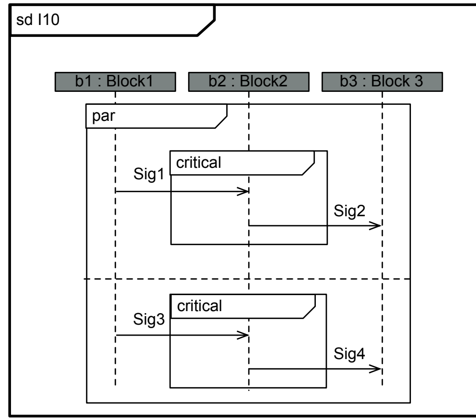


Figure 6.10: Interaction I_{10}.

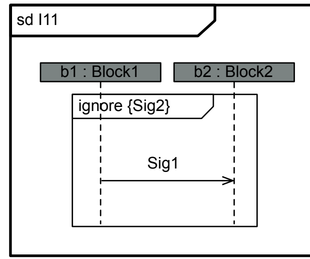


Figure 6.11: Interaction I_{11}.

$$
\begin{aligned}
L_2 &= \{\langle msg.3.snd.b_1.b_2.Sig_3\rangle,\\
&\quad \langle msg.3.rcv.b_1.b_2.Sig_3, msg.4.snd.b_2.b_3.Sig_4\rangle,\\
&\quad \langle msg.4.rcv.b_2.b_3.Sig_4\rangle\}\\
M_2 &= \{(3, b_1, b_2, Sig_3), (4, b_2, b_3, Sig_4)\}\\
CS_1 &= \langle msg.1.rcv.b_1.b_2.Sig_1, msg.2.snd.b_2.b_3.Sig_2\rangle\\
CS_2 &= \langle msg.3.rcv.b_1.b_2.Sig_3, msg.4.snd.b_2.b_3.Sig_4\rangle\\
CRITICAL &= Critical(L_1, M_1, L_2, M_2, CS_1, CS_2)
\end{aligned}
$$

□

6.4.3 Interaction Interpretation

The interaction operators described in Sections 6.4.1 and 6.4.2 allowed us to either impose a particular sequencing semantics on the interaction observations, or to model different patterns of control flow. In this section, we introduce the four operators that change our interpretation of a particular interaction sequence. We discuss these in the context of how they might possibly be used in a refinement check. In addition, we motivate why it is inappropriate to define process definitions in the spirit of the preceding section.

Ignore Operator

The *ignore* interaction operator provides, as part of the combined fragment, a set of messages that are to be ignored. Consequently, the messages are not allowed within the interaction fragment. The interpretation is that the messages are insignificant and irrelevant and are to be ignored if they appear in the interaction. An alternative interpretation is that the ignored messages can appear anywhere in a trace and still be considered valid. It is possible to model this as a template process, where the ignored traces are interleaved with those of the interaction (assuming we followed the second interpretation, and *ignore* contained all the valid observations of the ignored events between participating lifelines). In the following, we enforce the fact that a message must be sent before it can be received.

$$
\begin{array}{l}
\mathit{Ignore}(\mathit{lifelines}, \mathit{messages}, \mathit{messages}_{\mathit{ignore}}) = \\
\quad \mathit{let} \\
\qquad \mathit{Seq} = \mathit{Seq}(\mathit{lifelines}, \mathit{messages}) \\
\qquad \mathit{Ignored} = \Box(\mathit{id}, \mathit{from}, \mathit{to}, \mathit{name}) : \mathit{messages}_{\mathit{ignore}} \bullet \\
\qquad\quad \mathit{msg.snd.id.from.to.name} \rightarrow \mathit{msg.rcv.id.from.to.name} \rightarrow \mathit{Ignored} \\
\quad \mathit{within} \\
\qquad \mathit{Seq} \mathbin{|||} \mathit{Ignored}
\end{array}
$$

A more elegant solution can be achieved via the hiding operator and the first interpretation: in a refinement, we simply hide the ignored events from any behaviour we are comparing against. For example, the following would check, via a refinement, that an interaction is valid for a pair of communicating state machines, *StateMachines*:

$$\mathit{StateMachines} \setminus \mathit{ignore} \sqsubseteq \mathit{Seq}(\mathit{lifelines}, \mathit{messages})$$

Example 6.11. Consider Figure 6.11. Using the second interpretation, and the template process *Ignore*, we have:

$$
\begin{array}{l}
L_1 = \{\langle \mathit{msg}.1.\mathit{snd}.b_1.b_2.\mathit{Sig}_1\rangle, \langle \mathit{msg}.1.\mathit{rcv}.b_1.b_2.\mathit{Sig}_1\rangle\} \\
M_1 = \{(1, b_1, b_2, \mathit{Sig}_1)\} \\
N = \{(2, b_1, b_2, \mathit{Sig}_2), (3, b_2, b_1, \mathit{Sig}_2)\} \\
\mathit{IGNORE} = \mathit{Ignore}(L_1, M_1, N)
\end{array}
$$

The behaviour is Sig_1 with the observations of Sig_2 interleaved (these messages are assumed to be unimportant and ought to be ignored).

□

Consider Operator

The *consider* interaction operator specifies a set of messages that are to be considered as part of this combined fragment; all other messages are ignored. Consequently, the combined fragment can only contain the considered messages. The semantics are interpreted to mean that other messages might occur as part of the interaction, but that these are irrelevant and ought to be ignored. The *consider* operator can be defined in terms of *ignore*: ignore all other messages not considered. As was the case for *ignore*, there exists an al-

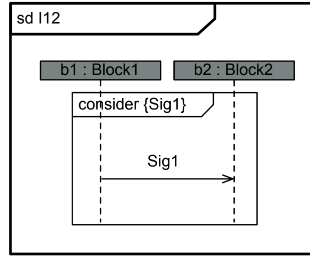


Figure 6.12: Interaction I_{12}.

ternative interpretation, where all messages that are not considered may appear anywhere in the traces.

The consider operator can be stated in terms of ignore thus.

$$\begin{aligned}&Consider(lifelines, messages, messages_{consider}) = \\ &\quad Ignore(lifelines, messages, \Omega \setminus messages_{consider})\end{aligned}$$

In the above Ω is the set of all possible events for a particular interaction.

Example 6.12. Consider Figure 6.12. We model the behaviour of I_{12} as follows:

$$\begin{aligned}&L_1 = \{\langle msg.1.snd.b_1.b_2.Sig_1\rangle, \langle msg.1.rcv.b_1.b_2.Sig_1\rangle\} \\ &M_1 = \{(1, b_1, b_2, Sig_1)\} \\ &\Omega = \{(1, b_1, b_2, Sig_1), (2, b_1, b_2, Sig_2)\} \\ &N = \{(1, b_1, b_2, Sig_1)\} \\ &CONSIDER = Ignore(L_1, M_1, \Omega \setminus N)\end{aligned}$$

□

Negative Operator

A sequence diagram, by default, describes *positive* traces — in the sense that it describes a valid interaction. We can invert this by making use of the negative operator, *neg*. A negative fragment therefore depicts invalid behaviour, i.e. behaviour that, if it occurs, is considered a failure. For example, if we have a pair of communicating state machines (*StateMachines*), and we want to assert that an interaction (assuming weak semantics) is not possible for them, we can use the following:

$$\begin{aligned}&Neg = Seq(lifelines, messages) \\ &StateMachines \setminus hiddenevents \sqsubseteq Neg\end{aligned}$$

The set *hiddenevents* represents CSP events that are not communicable by the process *Neg*. If the above refinement succeeds, we know that the negative (illegal) trace is possible — which is disallowed.

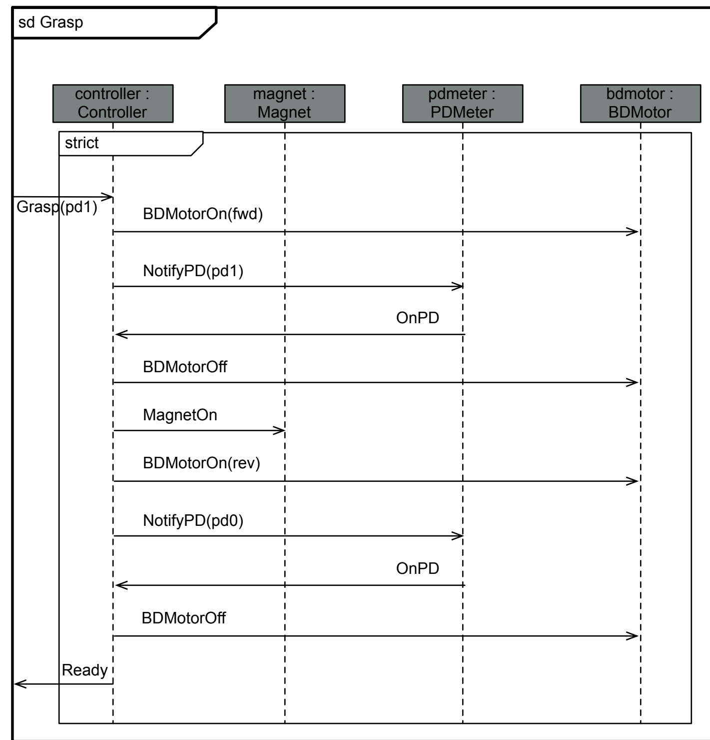


Figure 6.13: Interaction Grasp. Shows one possible interaction between the parts of the Arm block: the potentiometer, magnet, bidirectional motor, and controller.

Assert Operator

The assertion operator, *assert*, declares that the interaction fragment models the only valid continuations; any other eventuality is considered invalid. In this case, we need the refinement relation to hold in both directions.

6.5 Example

Sequence diagrams are used to depict the interactions between the parts of a block. In this section, we apply the behavioural semantics defined earlier in this chapter to our running case study. In particular, we formalise the interactions of the bidirectional motor, potentiometer, magnet and controller parts that make up the robotic arm.

Figure 6.13 formalises this behaviour between parts using a sequence diagram with strict sequencing. The signals *Grasp* and *Ready* are communications with the environment external to our system of interest, the arm. When the arm is commanded to grasp an object at extension pd_1: the bidirectional motor is turned on; the potentiometer is ordered to inform us when the specified potential difference is reached; the controller is notified once the potential difference is reached; the bidirectional motor is turned off and the magnet activated in order to grab the metal object. Another sequence of signals is subsequently executed in order to move the arm to extension pd_0 (while holding on to the object).

The CSP process modelling this behaviour follows.

$$
\begin{aligned}
L = \{&\langle msg.1.snd.controller.bdmotor.BDMotorOn.fwd,\\
&\quad msg.2.snd.controller.pdmeter.NotifyPD.pd_1, \ldots\\
&\quad msg.9.snd.controller.bdmotor.BDMotorOff\rangle,\\
&\quad \vdots\\
&\langle \ldots msg.9.rcv.controller.bdmotor.BDMotorOff\rangle\}\\
M = \{&(1, controller, bdmotor, BDMotorOn.fwd), \ldots\\
&\quad (9, controller, bdmotor, BDMotorOff)\}\\
O = \langle &msg.1.snd.controller.bdmotor.BDMotorOn.fwd,\\
&msg.2.snd.controller.pdmeter.NotifyPD.pd_1,\\
&\vdots\\
&msg.9.rcv.controller.bdmotor.BDMotorOff\rangle\\
Grasp &= Strict(L, M, O)
\end{aligned}
$$

Using appropriate renaming, we can assert that the above interaction is a possible execution sequence for the components of the arm. This and other concepts of consistency checking using CSP refinement will be formalised in Chapter 7.

6.6 Discussion

In this section we discuss alternative CSP representations of interactions that are closest to our own. We reflect on our formalisation by highlighting key advantages of our approach, and conclude the section with a critical analysis of the work presented in this chapter.

6.6.1 Alternative Formalisations

To the best of our knowledge there has been no CSP approach based on template processes, as presented in this chapter. We compare our work with that of Dan [**45**] and Davies and colleagues [**48**; **47**].

The work of Dan [**45**] considered the automatic translation of sequence diagrams to CSP using a model-driven approach.

- The main difference between our approach and that of Dan is that we define our semantics in terms of template processes that describe the patterns of behaviour for the various interaction or sequencing operators. In contrast, their semantics are defined in terms of *Query/View/Transformation* (QVT) [**119**] model transformations.

- The number of interaction or sequencing operators formalised by Dan is limited. In contrast, we consider the *strict*, *critical*, *ignore*, *consider*, *neg* and *assert* operators.

- The semantics formalised by Dan are based on message exchanges; our semantics are based on occurrence observations. Thus, in our semantics, every message has a send observation occurrence as well as a receive observation occurrence; in

contrast, their semantics have a single CSP event corresponding to the message exchange (the act of sending as well as receiving the particular message).

The other reference of note, where sequence diagrams were considered as part of a refinement framework is the work of Davies and colleagues [**48**; **47**].

- The main difference between our approach and the above is that in [**48**; **47**] basic sequence diagrams are used to specify the order of method invocations of a particular scenario. This is then checked against a behavioural class specification.
- Again, as is the case with Dan, the semantics defined in [**48**; **47**] are based purely on message exchanges, rather than on occurrence observations.
- No explicit reference is made to any of the sequencing or interaction operators treated in this chapter.

6.6.2 Reflections

We briefly outline the advantages and disadvantages of our approach and conclude with a critical discussion of the contributions of this chapter.

Advantages

One advantage of our approach is that any implementation of an automated translation mechanism would only have to instantiate the proposed CSP processes in order to describe the behaviour of the desired interaction operator.

The approach presented here is novel as we give a detailed account of interaction operators. Moreover, due to the nature of a process algebraic formalism like CSP, where the focus is on describing intricate patterns of behaviour, we are able to deal with interaction operators that alter our interpretation of a interaction sequence more naturally that in approaches that rely on traditional model checking using temporal logics [**87**]. The process algebraic approach suggested enables us to compare the behaviour of a sequence of interactions against another interaction in a natural fashion. In addition, the refinement checker, FDR, which allows the behaviour of one process to be compared against that of another in terms of a refinement hierarchy, provides a practical means of comparing behaviour of one sequence diagram against that of another (incorporating the operators that alter interaction interpretation, for example).

The presented approach does not rely on fixed size buffers in order to model asynchronous exchanges. Furthermore, because we consider separately the actions of sending and receiving a message, we are able to deal with lost and found messages, as well as message overtaking.

Disadvantages

The biggest disadvantage of our approach is the assumption that suitable, generalised process definitions exists that describe all desired patterns of behaviour. For example, we have only modelled the *loop* operator in terms of an iteration bound. If we required a semantics where a Boolean guard was used in the evaluation of the termination criteria, we would need to provide an alternative definition.

Our approach is less flexible when it comes to modelling patterns of interactions involving several nested operators. Specialised template processes would need to be defined in order to deal with more complex cases. We assume that interactions that consist of several consecutive interaction operators can be formalised using the CSP sequential composition of the appropriate template processes.

Critical Discussion

The message-passing mechanism in CSP is fundamentally based on the principle of a rendezvous between a sending and receiving process. Dan [**45**] models the sending of a message between lifelines L_1 and L_2 using the channel construct: the lifelines synchronise on the message being exchanged. The problem here, from our perspective, is that we require the sending and receiving of a message to be modelled as two, separate, detached events (the sending and receiving occurrence specifications related to the message exchange). However, the suggested approach abstracts them into a single event. This might have been appropriate, for example, if we were only concerned with the act of exchanging a message. However, this is not our desire here. Instead, we wish to decompose the exchange into two separate events. In doing so we will be able to operate our CSP models at a finger granularity.

Consider making use of sequence diagrams to check the validity of communicating state machines. Using our model for sequence diagrams, we would be able to make use of events (like a state machine sending an asynchronous message) that correspond to interaction occurrences on the sequence diagram. We should bear in mind that the sending of an asynchronous message by one state machine does not imply that the message is instantaneously received by another. Even if it is received immediately, it might still be placed in an event queue, so the receiving state machine might only process it later. If we operated at a coarser granularity, we would have to be content with only modelling the exchange of the message, making it impossible to distinguish between when it was sent and when it was processed.

6.6.3 Possible Extensions

The work presented in this chapter can be extended to incorporate the modelling of synchronous messages, corresponding to synchronous operations. In [**118**], we provided a uniform treatment of synchronous and asynchronous messages by incorporating the return message associated with a synchronous message exchange. The presentation here, however, focussed on asynchronous messages due to the fact that the formalisation of state machines in Chapter 4, and activities in Chapter 5, exclusively made use of signals.

6.7 Conclusion

In this chapter we provided a practical CSP formalisation for SysML interactions. The novelty of the approach is our use of CSP template processes to formalise the behaviour of complex interaction operators not previously considered within the formal methods community. In the next chapter we utilise the formal models derived in Chapters 4–6 to

present an overarching semantic framework — within the context of SysML — encompassing behavioural as well as the relevant structural formalisms.

7

REFINEMENT CHECKING SYSML

This chapter uses the notion of refinement over SysML diagrams in order to ensure that the resulting models are well-formed. We investigate the use of refinement in order to verify the model against behavioural requirements. The chapter amalgamates the formalisations of the preceding three chapters — state machines, activities, and interactions — and combines these into a unifying framework. We introduce the remaining SysML constructs and their CSP counterparts: blocks, parts, signals, enumerations and connectors, along with their associated diagrams. A compositional approach to specification and design is established and linked to the different interpretations of blocks in SysML. Next, the requirement diagram and the concept of allocation is discussed. We present an approach that allows for the formalisation of requirements and enables their subsequent traceability back to the model. Formal verification techniques are employed to verify that the stated requirements are satisfied by the proposed design. The work presented in this chapter previously appeared in [**120**; **110**; **121**; **113**].

7.1 Introduction

The work in this chapter stems from several previous publications that relate to the refinement checking of behavioural diagrams within the context of SysML. We briefly discuss these contributions here in order to set the context and provide an outline for the rest of this chapter.

Behavioural Consistency

The key justification here stems from the following:

> *"The integration of the semantics of different types of behaviours is sometimes complex and should be used with care"* [**14**]

Activities and state machines are the core behavioural constructs used to ascribe behaviour to SysML blocks. The aforementioned constructs are frequently used in combination: activities are used to assign behavioural features that ought to execute in a particular

state, or on a given transition. In this chapter, we provide a behavioural semantics for the conjoined behaviour of state machines and activities. In the past, there have been several contributions where the sole focus lied either with the formalisation of state machines, or activities. To the best of our knowledge, this chapter is the first contribution where the intention is on the provision of a behavioural semantics that encompasses both these formalisms. We use refinement to ensure consistency between communicating state machines and the specification of their interactions via sequence diagrams.

Reasoning about behaviour — in particular, the myriad of interactions between components — is a rather cumbersome activity for the human mind. In addition, our cognitive ability to cope with multiple, separate descriptions of behaviour, and ultimately fuse these into a unified interpretation, is rather limited. We need to augment our faculties with appropriate notations in order to effectively reason about such behaviours. Moreover, if we are going to utilise these notations in a meaningful fashion, we require mechanised tool support.

Component Interactions

Accidents associated with complex systems are frequently the result of unforeseen interactions amongst components that all satisfy their individual requirements [**122**]. These *component interaction accidents* are increasingly common: state of the art systems are more interdependent on other technologically advanced systems and interact in ways not foreseen or intended by the original designer. The *Mars Polar Lander* accident is one example of such a failure: both the landing legs and the control software of the descent engines functioned as specified by their respective behavioural specifications. The systems engineers, however, did not consider all the potential interactions between the landing legs and the control software of the descent engines [**122**].

In order to effectively reason about the safe integration of different components or systems we utilise the formalisations defined in the preceding chapters.

Compositional Approach to Specification and Design

At the structural level, SysML takes a compositional stance with regards to systems specification: a block can be comprised of other blocks, which, in turn, might themselves consist of blocks. However, for the approach to be effective and useful, the behavioural conduct of these blocks need to be specified in a consistent manner. Moreover, the approach needs to enable the modeller to sufficiently abstract away details irrelevant to a particular level of abstraction.

Requirements Tracibility

Requirements traceability plays an important role as part of any model-based systems engineering methodology. In SysML, requirements can be related to other requirements, as well as to other model elements via one or more relationships: a behavioural construct can be allocated to a particular requirement, and we can subsequently use FDR to ensure that the model satisfies it.

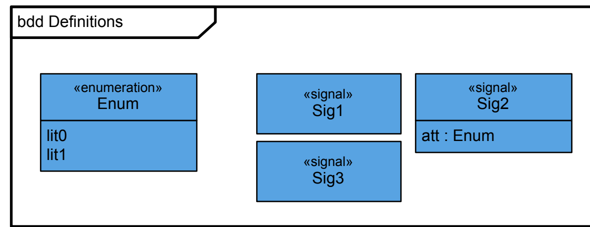


Figure 7.1: Enumeration and signal definitions.

7.2 Structural Counterparts

In this section we briefly provide formalisations for the structural counterparts of our framework. The behavioural formalisms we have introduced so far — state machines, activities, and sequence diagrams — all execute within the context of a structural counterpart. The structural constructs of interest to us are enumerations, signals, blocks, parts, and connectors.

7.2.1 Enumerations and Signals

Signal and enumeration definitions introduce the messages and associated parameters communicated between state machines and activities.

Let $\mathcal{E}$ denote the set of all enumerations. An enumeration is a user-defined type where the enumeration literals represent distinct constants in the model. The CSP counterpart of a SysML enumeration is a CSP datatype. Here, there is a one-to-one mapping between the constants of the CSP datatype, and the enumeration literals defined for the SysML enumeration.

Signals are communicated along the connectors connecting parts. Let $\mathcal{S}$ denote the set of all signals. The signals used by the communicating state machines correspond to constants of a CSP datatype definition. For each block, the signals corresponding to the provided receptions of the particular block are used. Where a signal has associated parameters, these are included in the datatype definition.

Example 7.1. Consider Figure 7.1. We can model the SysML enumeration *Enum* as a CSP datatype with constants corresponding to the enumeration literals.

$$\text{datatype } Enum = lit_0 \mid lit_1$$

□

7.2.2 Blocks, Parts, and Connectors

The fundamental modelling construct present in SysML is the block. Each block is assigned an associated main behaviour, called its *classifier* behaviour. Depending on the purpose of the block, the classifier behaviour can either be a state machine or an activity.

Let $\mathcal{B}$ denote the set of all blocks. A block $B \in \mathcal{B}$ is a classifier that describes common behavioural and structural features of its instances, and can be considered akin to a UML

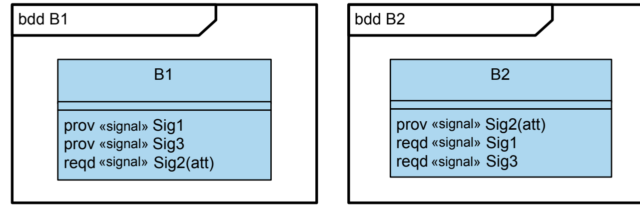


Figure 7.2: Provided and required receptions of blocks B_1 and B_2.

class. We assume that the classifier behaviour is specified using state machines, and given by the function

$$classifier : \mathcal{B} \rightarrow \mathcal{M}$$

These blocks communicate via events (instances of signals) that act as stimuli for the respective state machines.

A block makes known the names of the *receptions*, each corresponding to a signal, that: it responds to (i.e. the block provides the behaviour); or, alternatively, expects its SysML environment to respond to (i.e. the environment provides the behaviour). These behavioural features are designated as *provided* and *required behavioural features*. We define the functions

$$prov : \mathcal{B} \rightarrow \mathbb{P}\,\mathcal{S}$$
$$reqd : \mathcal{B} \rightarrow \mathbb{P}\,\mathcal{S}$$

to return provided and required receptions.

The *internal block diagram* graphically sets out the internal structure of a block from its parts. In contrast, the *block definition diagram* depicts the composition of a block, but abstracts away from the internal structure. A *part* is connected to another part via a connector; it is an instance of a block. As such, it represents a particular usage of its classifying block within the context of its owning block. Each part $P \in \mathcal{P}$ is typed by a block $B \in \mathcal{B}$; the function

$$type : \mathcal{P} \rightarrow \mathcal{B}$$

reflects this.

The *connector* serves as a bidirectional link between the block instances and is used to convey signals send between communicating block instances. The state machine of a block B only receives (through its event queue) the provided receptions, $prov(B)$. The required features, $reqd(B)$, are communicated across the connectors linking parts. SysML block instances are connected using connectors; connectors are modelled using CSP channels. For simplicity we use the name of the association end for the purposes of communication, and assume this to be the name of the associated block instance. For example, if another part P_i provides a feature S_j that a part P_k requires, the state machine of P_k will use the name of part P_i as the CSP channel to send the required event on.

The structure, and subsequent overall behaviour, of a block $B \in \mathcal{B}$, composed from N constituent block instances, the parts $\{P_0 \mathinner{\ldotp\ldotp} P_{N-1}\} \subseteq \mathcal{P}$, is expressed in CSP via parallel

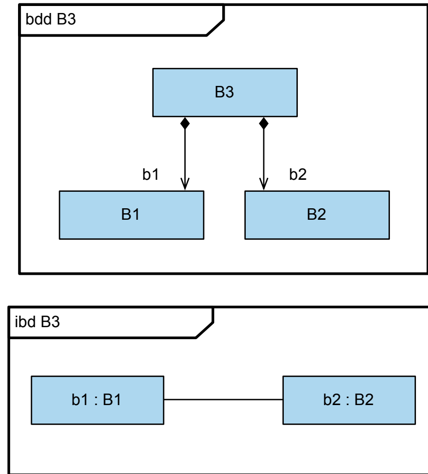


Figure 7.3: Block definition and internal block diagrams for B_3.

composition. The classifier behaviour of each part P_j is modelled via a state machine M_j, given by $\mathit{classifier}(\mathit{type}(P_j))$.

The complete system, B_i, can be modelled by placing the processes corresponding to each of the state machines M_j, where $0 \leq j \leq N-1$, in parallel:

$$B_i = \|\, j : \{0\,..\,N-1\} \bullet [\alpha M_j] M_j$$

In Section 7.4 we show how this formalisation naturally results in a compositional approach to specifying and designing systems composed from other systems or components.

Example 7.2. Consider Figures 7.1 and 7.2. We have the following CSP datatype definitions corresponding to the provided receptions of B_1 and B_2, respectively.

datatype $\mathit{ReceptionsB}_1 = \mathit{Sig}_1 \mid \mathit{Sig}_3$
datatype $\mathit{ReceptionsB}_2 = \mathit{Sig}_2.\mathit{Enum}$

□

Example 7.3. B_3 is composed of B_1 and B_2, as shown in the block definition diagram in Figure 7.3. The interconnection of B_3's parts is shown on the internal block diagram. Part b_1 requires $\mathit{Sig}_2(\mathit{att})$, which is provided by Part b_2. In CSP, our convention is that the communication is done on a channel corresponding to the name of the part that provides the required reception. For example, if b_1 wanted to send the signal event $\mathit{Sig}_2(\mathit{lit}_0)$ to b_2, the communication would take the form $b_2.\mathit{Sig}_2.\mathit{lit}_0$. It follows that we would expect CSP channel definitions on which the datatypes corresponding to the provided receptions can be communicated.

channel $b_1 : \mathit{ReceptionsB}_1$
channel $b_2 : \mathit{ReceptionsB}_2$

□

7.3 Behavioural Consistency

We have now introduced all the behavioural and structural features that we need to reason about the consistency of our design. In this section we formalise the aforementioned constructs at a high level with the view to bring together the different behavioural and structural diagrams of interest.

7.3.1 A SysML Model

A SysML model is a 9-tuple $(\mathcal{B}, \mathcal{P}, \mathcal{C}, \mathcal{S}, \mathcal{E}, \mathcal{M}, \mathcal{A}, \mathcal{I}, \mathcal{R})$, where

- $\mathcal{B}$ represents the set of all blocks;
- $\mathcal{P}$ represents the set of all parts;
- $\mathcal{C}$ represents the set of all connectors;
- $\mathcal{S}$ represents the set of all signals;
- $\mathcal{E}$ represents the set of all enumerations;
- $\mathcal{M}$ represents the set of all state machines;
- $\mathcal{A}$ represents the set of all activities;
- $\mathcal{I}$ represents the set of all interactions; and
- $\mathcal{R}$ represents the set of all requirements.

The mathematical compositions, and their respective mappings to CSP, defined in the present and the previous chapters still apply. Recall that in Chapters 4–6 we introduced state machines, activities and interactions. We said that state machines execute within the context of a block, and that activities are supplementary constructs used to describe state-based or transition-related behaviours. This chapter introduced all the structural components necessary to describe the types of SysML models we consider. We are yet to formalise requirements, a fundamental part of SysML; this is the topic of Section 7.5.

The mathematical constructs above were then mapped to corresponding CSP processes. These processes collectively form a formal behavioural description of the associated SysML model. We can gain a deeper understanding of the specified system behaviour using tools such as animators. We can use the power of refinement, supported by automated tools like FDR, to:

- ensure that all the behavioural constructs interact in such a way that the overall system is free from deadlock;
- ensure that concrete blocks satisfy the specification of their abstract composite block (in other words the parts satisfy the behaviour of the whole); and
- ensure that requirements, formalised using behavioural constructs such as state machines, are satisfied by the model.

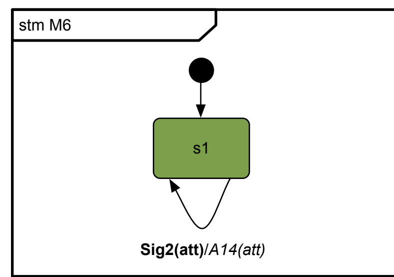


Figure 7.4: State machine M_6.

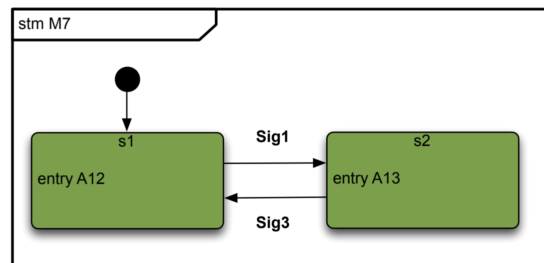


Figure 7.5: State machine M_7.

Moreover, the above approach of abstract and concrete blocks naturally lead to a compositional approach to specification and design.

However, before we can explore the ideas above, we need a clear understanding of how the behavioural and structural constructs of interest are integrated in CSP.

7.3.2 Integrative Example

We now provide a pedagogic example, based on two communicating state machines, with simple activities associated with each of these in order to demonstrate how the structural CSP definitions fit in with the behavioural formalisms we saw in earlier chapters. The goal here is not to introduce all the diagrammatic constructs of the respective formalisms, but rather to show how they all integrate.

The SysML model we wish to use as exemplar consists of the blocks B_1, B_2 and B_3 and parts b_1 and b_2. See Figures 7.1 – 7.3.

Assume that the classifier behaviour of block B_1 is state machine M_7, and the classifier behaviour of block B_2 is M_6, that is

$$classifier(B_1) = M_7$$
$$classifier(B_2) = M_6$$

The state machines, along with their associated activities, are shown in Figures 7.4 – 7.6. Block B_1 provides receptions Sig_1 and Sig_3. Thus we would expect the classifying state machine to contain those signals as triggers on the transitions between different states. Similarly, B_2 provides $Sig_2(att)$ and we would expect corresponding triggers in the classifying state machine.

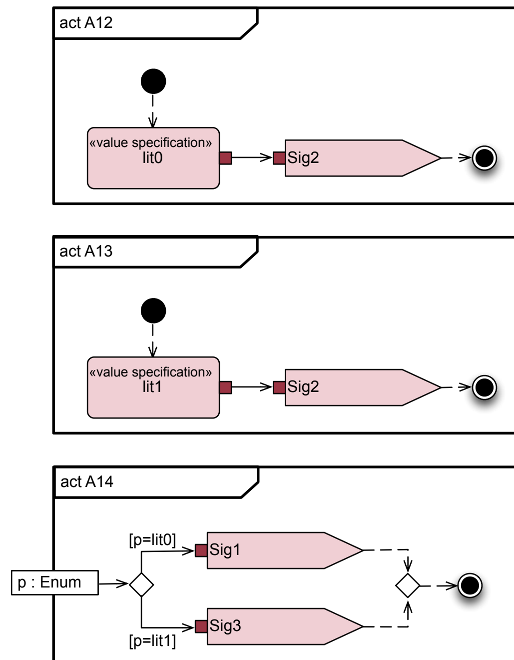


Figure 7.6: Activites A_{12}, A_{13} and A_{14}.

Structure

SysML blocks are connected using connectors; connectors are modelled using CSP channels. For every block we require two CSP channels: the first models the event queue that the block uses to communicate with the external environment; the second is used for internal communication between the block and its associated event queue. CSP datatype definitions model the signals (and associated enumerations) communicated between blocks.

datatype $Enum = lit_0 \mid lit_1$
datatype $ReceptionsB_1 = Sig_1 \mid Sig_3$
datatype $ReceptionsB_2 = Sig_2.Enum$
datatype $Dispatched = proc \mid disc$
channel $b_1 : ReceptionsB_1$
channel $b_2 : ReceptionsB_2$
channel $b_1local : Dispatched.ReceptionsB_1$
channel $b_2local : Dispatched.ReceptionsB_2$

In the above, the channel b_1 is used to communicate with the state machine of b_1 via its associated event queue; the channel b_1local is used by the event queue of b_1's state machine to dispatch events for processing. Thus b_1local can be thought of as an internal channel between the classifier state machine of b_1 and its event queue. The datatype *Dispatched* models this: an event can either be processed, or, if the state machine is in a state where the dispatched event is not expected, discarded.

Behaviour

The CSP processes of state machines M_6 and M_7 follow. Activity A_{14} is a transition-related behaviour of M_6; A_{12} and A_{13} are state-based (entry) behaviours of M_7.

State machine M_6 has a single trigger: Sig_2. After the triggering event is received, the effect component of the transition executes: activity A_{14}. The state machine process is instantiated with the appropriate channels in line with the connectors. The alphabet of M_6 is calculated as the events that the state machine can engage in, as well as those of its associated activities.

$$
\begin{array}{l}
M_6'(queue, local) = \\
\quad \text{let} \\
\quad\quad I = S_1 \\
\quad\quad S_1 = local.proc.Sig_2?att \rightarrow A_{14}(att) \mathbin{;} S_1 \\
\quad\quad EQ = queue?e \rightarrow local?p!e \rightarrow EQ \\
\quad \text{within} \\
\quad\quad I \; [| \; \{| \; local \; |\} \; |] \; EQ
\end{array}
$$

$$
\begin{array}{l}
M_6 = M_6'(b_2, b_2local) \\
\alpha M_6 = Union(\{\{| \; b_2, b_2local \; |\}, \alpha A_{14}\})
\end{array}
$$

Activity A_{14} has a single activity parameter node, which takes as input an *Enum*; depending on the value of the input, the process either sends Sig_1 or Sig_3 to b_1. Note that because we are using process composition between the processes of activity A_{14} and the local process definitions of M_6, we are able to provide an integration between the semantics of these constructs.

$$
\begin{array}{l}
A_{14}(p) = \\
\quad \text{let} \\
\quad\quad D(o) = \\
\quad\quad\quad \text{if } o = lit_0 \text{ then} \\
\quad\quad\quad\quad SS_1 \\
\quad\quad\quad \text{else if } o = lit_1 \text{ then} \\
\quad\quad\quad\quad SS_2 \\
\quad\quad\quad \text{else} \\
\quad\quad\quad\quad Stop \\
\quad\quad SS_1 = b_1.Sig_1 \rightarrow F \\
\quad\quad SS_2 = b_1.Sig_3 \rightarrow F \\
\quad\quad F = Skip \\
\quad \text{within} \\
\quad\quad D(p)
\end{array}
$$

$$
\alpha A_{14} = \{| \; b_1.Sig_1, b_1.Sig_3 \; |\}
$$

The state machine of block b_1, M_7 is represented in CSP as follows. Note that the entry behaviour is executed before the triggers of the target state are available (recall that the entry behaviour is a non-interruptible behaviour). Also note that events that are not expected in a particular state are discarded without effect: local process S_1 discards Sig_3

by removing it from the front of the event queue. EQ models the event queue.

$$
\begin{array}{l}
M'_7(queue, local) = \\
\quad \text{let} \\
\quad\quad I = A_{12} \fatsemi S_1 \\
\quad\quad S_1 = \\
\quad\quad\quad local.proc.Sig_1 \rightarrow A_{13} \fatsemi S_2 \\
\quad\quad\quad \Box \\
\quad\quad\quad local.disc?e : \{| Sig_3 |\} \rightarrow S_1 \\
\quad\quad S_2 = \\
\quad\quad\quad local.proc.Sig_3 \rightarrow A_{12} \fatsemi S_1 \\
\quad\quad\quad \Box \\
\quad\quad\quad local.disc?e : \{| Sig_1 |\} \rightarrow S_2 \\
\quad\quad EQ = queue?e \rightarrow local?p!e \rightarrow EQ \\
\quad \text{within} \\
\quad\quad I \; [| \; \{| local |\} \; |] \; EQ
\end{array}
$$

$$
\begin{array}{l}
M_7 = M'_7(b_1, b_1local) \\
\alpha M_7 = Union(\{\{| b_1, b_1local |\}, \alpha A_{12}, \alpha A_{13}\})
\end{array}
$$

The entry behaviour of state s_2 of M_7 is represented in CSP as follows.

$$
\begin{array}{l}
A_{12} = \\
\quad \text{let} \\
\quad\quad I = VS \\
\quad\quad VS = SS(lit_0) \\
\quad\quad SS(o) = b_2.Sig_2.o \rightarrow F \\
\quad\quad F = Skip \\
\quad \text{within} \\
\quad\quad I
\end{array}
$$

$$\alpha A_{12} = \{| b_2.Sig_2.lit_0 |\}$$

Activity A_{13} uses a value specification action and an object flow to send a signal to b_2.

$$
\begin{array}{l}
A_{13} = \\
\quad \text{let} \\
\quad\quad I = VS \\
\quad\quad VS = SS(lit_1) \\
\quad\quad SS(o) = b_2.Sig_2.o \rightarrow F \\
\quad\quad F = Skip \\
\quad \text{within} \\
\quad\quad I
\end{array}
$$

$$\alpha A_{13} = \{| b_2.Sig_2.lit_1 |\}$$

The overall behaviour of the above state machines and activities is as expected. The communicating state machine (M_7) of block b_1, enters state s_1. The entry behaviour of s_1

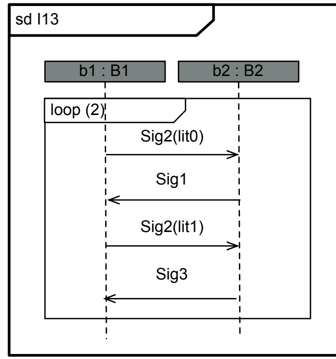


Figure 7.7: Sequence diagram showing interaction I_{13} between b_1 and b_2.

executes, which sends a signal to b_2's state machine, M_6. In M_6, this triggers a transition-to-self, but as part of this A_{14} is executed. This, in turn, results in the transition with Sig_1 being triggered in M_7. M_7 alternates between s_1 and s_2 forevermore.

Assuming that

$$P = \{M_6, M_7\}$$

we then have

$$CONCRETE = \parallel p : P \bullet [\alpha p]p$$

The process CONCRETE represents the combined behaviour of B_1 and B_2.

We can show that our composition, CONCRETE, at the very least, is free from deadlock using the following check:

$$CONCRETE : [\text{deadlock free}]$$

The interactions between the parts of B_3, here b_1 and b_2, can be modelled via interaction I_{13}, as per Figure 7.7. The template process *Loop* can be used to model this behaviour: a scenario containing two iterations between the communicating state machines.

$$\begin{aligned}
L = \{&\langle msg.1.snd.b_1.b_2.Sig_2.lit_0, msg.2.rcv.b_2.b_1.Sig_1,\\
&\quad msg.3.snd.b_1.b_2.Sig_2.lit_1, msg.4.rcv.b_2.b_1.Sig_3\rangle,\\
&\langle msg.1.rcv.b_1.b_2.Sig_2.lit_0, msg.2.snd.b_2.b_1.Sig_1,\\
&\quad msg.3.rcv.b_1.b_2.Sig_2.lit_1, msg.4.snd.b_2.b_1.Sig_3\rangle\}\\
M = \{&(1, b_1, b_2, Sig_2.lit_0), (2, b_2, b_1, Sig_1), (3, b_1, b_2, Sig_2.lit_1), (4, b_2, b_1, Sig_3)\}\\
LOOP &= Loop(L, M, 2)
\end{aligned}$$

In order to compare this behaviour against the classifier state machines, we need to rename the events of the process *LOOP*.

$CONCRETE^R$ is the process with events suitably renamed to ensure compatible alpha-

bets.

$$\begin{array}{l} LOOP^R = \\ \quad LOOP[\\ \qquad msg.1.snd.b_1.b_2.Sig_2.lit_0 \leftarrow b_2.Sig_2.lit_0, \\ \qquad msg.2.rcv.b_2.b_1.Sig_1 \leftarrow b_1local.proc.Sig_1, \\ \qquad msg.3.snd.b_1.b_2.Sig_2.lit_1 \leftarrow b_2.Sig_2.lit_1, \\ \qquad msg.4.rcv.b_2.b_1.Sig_3 \leftarrow b_1local.proc.Sig_3, \\ \qquad \vdots \\ \qquad msg.4.snd.b_2.b_1.Sig_3 \leftarrow b_1.Sig_3] \end{array}$$

We can then assert that this is a valid scenario for the communicating state machines as follows:

$$CONCRETE \sqsubseteq_T LOOP^R$$

Error Injection

We have illustrated how to verify that a correct behavioural description involving state machines, activities and interactions can be combined to form an overall description of our system. Furthermore, we have shown how to carry out checks, such as freedom from deadlock, or to assert that a given scenario, specified by an interaction, is a valid, possible realisation of the interactions of the communicating state machines and activities belonging to the blocks represented as lifelines.

What we have not shown, is how FDR can detect anomalies in our design. Consider, for example, M_7 of Figure 7.5: if the entry behaviours of the two states had been erroneously reversed, the resulting system would deadlock. M_7 would start by sending $Sig_2(lit_1)$ to M_6; M_6 would respond with Sig_3, which M_7 would discard. M_7 would then forever wait on Sig_1, and the resulting system would be deadlocked. FDR will detect this. Similarly, FDR will detect (and supply a counterexample) if a given interaction is not a possible scenario of the communicating state machines and activities.

7.4 Compositional Approach to Specification and Design

Modelling a system with SysML relies on the concept of blocks — each with an associated set of states — that communicate via events, possibly resulting in a change of state for one or more of the communicating blocks. The architecture of these systems allows a top-down design, starting from an abstract level with high level concepts, down to levels with increasingly more detail. These successive transformations allow replacing an abstract block with a composition of parts, but the big drawback of this decomposition is that it is at best semi-formal and cannot guarantee consistency between a block and its parts. However, there are two alternative interpretations with regards to the behaviour of the resulting composition [**14**].

1. The classifier behaviour of the block can serve as an *abstraction* of the behaviours of its parts. The abstraction serves as a specification that the parts must realise: the parts must interact in such a way that their combined behaviour conforms to the abstraction.

2. Alternatively, the classifier behaviour of the block acts as a *controller* in order to actively orchestrate the behaviours of its parts. In this case, the behaviour of the block is a combination of its behaviour and that of its parts.

By making use of a process-algebraic formalism we are able to explore these interpretations more rigorously, and present a compositional approach to specification and design of systems.

The interested reader is referred to [**121**] for a case study that explores the concepts introduced in this section. The case study is explored from two slightly different perspectives, related to the different interpretations which can be attributed to the composition of a block from its parts.

7.4.1 Interpretations

We now discuss and formalise the different interpretations attributed to the resulting composition when a block is composed of other blocks.

Abstraction

The first of the interpretations above assumes that the behaviour of the composite block serves as an abstraction of the behaviour of its parts. Assume a block $B_i \in \mathcal{B}$ composed of K constituent blocks $B_0 \,..\, B_{K-1}$, where $i \geq K$. We know that the aggregate behaviour exhibited by blocks $B_0 \,..\, B_{K-1}$ must adhere to that of the composite block B_i. B_i is an abstract specification block that the more concrete implementation blocks $B_0 \,..\, B_{K-1}$ must implement. Stated in terms of CSP: the characteristic process of B_i serves as the specification process and $B_0 \,..\, B_{K-1}$, suitably combined using parallel composition, form the implementation process.

Assume that $\mathit{classifier}(B)$ represents the classifier behaviour of a SysML block. Using CSP the conformance of the implementation process to that of the specification can be stated thus.

$$\begin{aligned}&\mathit{CONCRETE} = \parallel P : \{B_0 \,..\, B_{K-1}\} \bullet \mathit{classifier}(P)\\&\mathit{classifier}(B_i) \sqsubseteq \mathit{CONCRETE}\end{aligned}$$

Events introduced at the lower level of implementation are excluded from the above observation; the hiding operator of CSP can be used to conceal such events.

Using this approach, and assuming the refinement holds, B_i can be safely substituted for the concrete composition $B_0 \,..\, B_{K-1}$. This stepwise, compositional approach to systems specification and design sits well with CSP's approach to refinement. This statement is not necessarily true for conventional model checkers that rely on temporal logics to assert safety or liveness properties. In a *system of systems*, B_i, previously our *system of interest*, is now just a component block representing one of the subsystems.

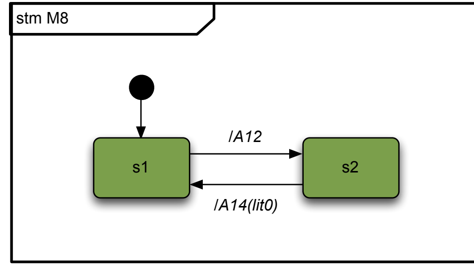


Figure 7.8: State machine M_8.

Example 7.4. Refer to Figure 7.3. Using the abstraction interpretation, B_3 is a specification, to be implemented by B_1 and B_2; B_3 is an abstract block, and B_1 and B_2 are concrete implementation blocks. If we assume that state machine M_8 of Figure 7.8 is the classifier behaviour of B_3, and that we follow the abstraction interpretation, we can use hiding (and event renaming, if needed) to consolidate the alphabets of the respective processes prior to the refinement check.

$M_8 =$
  let
    $I = S_1$
    $S_1 =$
      $A_{12} \mathbin{;} S_2$
    $S_2 =$
      $A_{14}(lit_0) \mathbin{;} S_1$
  within
    I

$\alpha M_8 = \{b_2.Sig_2.lit_0, b_1.Sig_1\}$
$CONCRETE = \| p : \{M_6, M_7\} \bullet [\alpha p]p$
$M_8 \sqsubseteq_T CONCRETE \setminus \{\Sigma \setminus \alpha M_8\}$

In the above, M_8 is the specification process corresponding to the classifier behaviour of B_3. We omit the event queue because M_8 consists solely of activities that are send signal event actions. The above specification states that $Sig_2(lit_0)$ and Sig_1 should alternate. Here, the other events are lower level implementation details and are hidden from view at this level of the specification, using CSP hiding.

□

Controller

The second interpretation assumes that a block acts as a controller for its parts. The behaviour of the block is a combination of its behaviour and that of its parts: the behaviour of its parts are considered as part of the behaviour of the composite.

Assume a block $B_i \in \mathcal{B}$ composed of K constituent blocks $B_0\,.\,.B_{K-1}$, where $i \geq K$. We known that the aggregate behaviour is that of blocks $B_0\,.\,.\,B_{K-1}$ as well as the composite block B_i. Here, we think of all of the blocks as implementation blocks that combine in

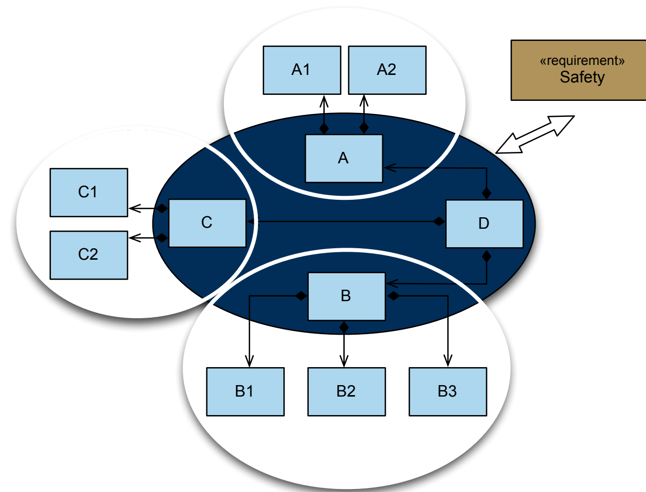


Figure 7.9: The compositional approach afforded by CSP.

order to give the desired behaviour. This can be formalised in terms of CSP thus.

$$CONCRETE = \parallel P : Union(\{\{B_0 \,..\, B_{K-1}\}, \{B_i\}\}) \bullet classifier(P)$$

Example 7.5. Refer to Figure 7.3. Using the controller interpretation, the behaviour of B_3 is its own classifier behaviour as well as those of its parts B_1 and B_2. B_1, B_2 and B_3 are all considered concrete implementation blocks. For example, it is easy to image B_3 as a block that orchestrates the behaviour of B_1 and B_2 by resetting them, or shutting them down by sending an appropriate signal. In this case, the state machines M_6 and M_7 would have to be amended to incorporate the reset and shutdown signals (to the first and final states, respectively). The resulting behaviour is the cumulative behaviour of the composite block (B_3) as well as its composing blocks (B_1 and B_2).

□

7.4.2 Compositional Design

The controller interpretation from Section 7.4.1 has the drawback that there is no specification process that can be substituted for the composition. However, this interpretation sits well where the overall system architecture is described in terms of high level blocks. These high level blocks might be specification level or abstract blocks, each obtained from previous refinements using the first interpretation. At the architectural level, however, the integrated behaviour of all the components would be of interest to the modeller. Here, techniques that would assist in assured requirements traceability would be beneficial, as discussed in Section 7.5.

Figure 7.9 depicts these concepts graphically. The white ellipses denote the behavioural interpretation of blocks using the abstraction approach. The system of interest, block D, is composed of abstract blocks and serves as a controller that orchestrates the behaviour. The second interpretation, shown inside the dark blue ellipse, applies here. The behavioural of the overall system is the combined behaviour of blocks A, B, C and D. Furthermore, block A serves as a behavioural specification that must be satisfied by its

constituted blocks A_1 and A_2. Block A can be substituted for its components in the overall system. A similar line of argument can be followed for blocks B and C, together with their component blocks. Safety requirements can be allocated behavioural constructs to further refine the intentions of the modeller, and checked for conformance using CSP.

7.5 Requirements Traceability

A *requirement* $R \in \mathcal{R}$ is a SysML-specific modelling construct represented explicitly in the syntax of the language via the *requirement diagram*. Requirements, in their most basic form, are typically text-based, and allow for the description of conditions that must be satisfied by a particular system. Requirements can be related to other requirements and to modelling constructs via several relationships. For the purposes of this thesis, we concern ourselves with the *satisfy* and *refine* relationship, which are defined as follows.

> "The satisfy relationship describes how a design or implementation model satisfies one or more requirements" [**1**]

This relationship is used to state that a particular model element meets the associated requirement. This is merely an assertion and not a proof of fact.

> "The refine requirement relationship can be used to describe how a model element or set of elements can be used to further refine a requirement" [**1**]

A text-based requirement can be captured in a SysML model, with the danger being that a textual description can often be ambiguous. Furthermore, such a description is, in general (and for obvious reasons), not well-suited to automated reasoning. A more precise definition is possible in SysML, which allows any behavioural formalism — for example, a state machine — to be assigned to a requirement using the refine relationship. This results in a more formal representation of the corresponding desired behaviour. If this behavioural requirement is subsequently mapped to CSP — as a corresponding characteristic process — we can utilise the refinement checker to assist in reasoning about whether this refinement holds for a given model. The characteristic process is therefore a CSP process that describes the patterns of behaviour of an associated textual requirement.

Consider Figure 7.10. Here, we assume that $\mathit{classifier}(B)$, M and M' all denote behavioural constructs of SysML, and R represents a text-based requirement. In the case of a static modelling construct like a block, we assume the corresponding classifier behaviour. We base our treatment of the SysML satisfy and refine relationships on that of the satisfaction and refinement relations of [**17**]. In the following, CHAR_R denotes the characteristic process of a textual requirement R, and B, M, and M' are the CSP processes that represent their SysML counterparts with the same name, respectively. For the satisfy relationship between a behavioural formalism B and a textual requirement R, we define:

$$B\ \mathit{satisfy}\ R \Leftrightarrow \mathit{CHAR}_R \sqsubseteq B$$

The definition of the refine relationship is dependent on the model elements involved in the relationship. In the case where the relationship is between two behavioural for-

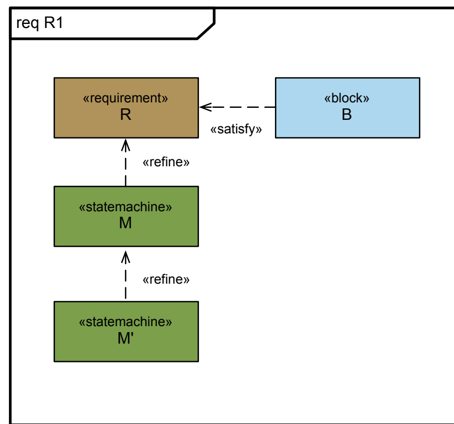


Figure 7.10: Requirement R.

malisms M and M', we define:

$$M' \mathit{refine}\, M \Leftrightarrow M \sqsubseteq M'$$

Alternatively, in the case where the refine relationship is expressed between a behavioural formalism M and a textual requirement R, we define the corresponding process M to be the characteristic process $CHAR_R$ of R:

$$M\, \mathit{refine}\, R \Leftrightarrow CHAR_R = M$$

Therefore, using the definitions above (and substitution) we can check whether R holds in B by executing a refinement check (where the set *Hidden* consists of those events not present in M):

$$M \sqsubseteq B \setminus \mathit{Hidden}$$

If the refinement does not hold, FDR generates a *counterexample* that demonstrates where the behaviour of B deviates from M, prompting the designer to correct the design. Furthermore, we can check the refinement relation between M and M':

$$M \sqsubseteq M' \setminus \mathit{Hidden}$$

These successive transformations (assuming the refinements hold) are behavioural formalisms that we can think of as getting more specific as we move along the refinement chain: M' is more refined than M, which means that, if we only consider traces, $\mathit{traces}\, [\![M']\!] \subseteq \mathit{traces}\, [\![M]\!]$ due to the reverse inclusion characteristic of refinement. The direction of the arrowhead in the requirement diagram, as per Figure 7.10, and its treatment in CSP is also of significance. In SysML, the direction of the arrow is from the dependent model element to the independent model element. The state machine B is dependent on the requirement R: any change in R could impact B; CSP refinement honours this dependency.

The requirement diagram is introduced in SysML and is a core part of the language. Requirements traceability is undoubtedly a good thing; a formal representation of require-

ment diagrams in CSP gives rise to a formal means of requirements traceability. This provides a formal foundation for this otherwise informal technique that serves as justification for the development of a formal framework. It should be noted that the above approach hinges on the characteristic process being specified correctly.

7.6 Discussion

In this section we reflect on our formalisation by highlighting key contributions of our approach. We conclude the section with a discussion on related and future work.

7.6.1 Contribution

In this chapter we demonstrated how the refinement checker FDR can be used in a practical setting to ensure that different behavioural formalisms — activities, state machines and interactions — are consistent. Moreover, we have demonstrated how the concept of refinement can be used to decompose a complex design problem to give rise to a top-down approach to designing a system comprised of subsystems. To the best of our knowledge this is the first contribution that explored the different notions of behavioural integration from a formal, process algebraic perspective. Furthermore, we are not aware of any other contribution that integrates the behavioural formalisms explored in this chapter — state machines and activities — using CSP. Finally, we showed how the resulting framework may be used to underpin an approach that not only facilitates the refinement checking of behavioural consistency of SysML diagrams, but also enables requirements traceability via formal refinement checking.

The contributions of this chapter can be summarised thus.

- We presented a formal model of SysML blocks using CSP. In particular, we demonstrated two possible interpretations of SysML blocks for modelling and integrating system behaviour in a formal setting.
- We presented an overarching behavioural semantics for state machines and activities. To the best of our knowledge, this is the first formalisation that encompasses and considers the combined behaviour of both of these constructs.
- We linked state machines and activities with interactions and showed how refinement can be used to assert consistency.
- We demonstrated how CSP can be used in conjunction with SysML in a compositional, refinement-based approach to specification. The proposed methodology was evaluated using a case study that is well suited to underline the principles of systems engineering.
- We formalised requirements, a core concept in SysML, and linked these with the refinement notions of CSP.
- We showed how refinement can be utilised to facilitate requirements traceability and bestow a sense of confidence in the validity of the design.

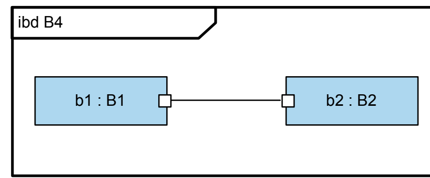


Figure 7.11: Internal block definition diagram for B_4.

The choice of CSP is due to a number of factors. The behavioural aspects of SysML can be modelled naturally by a process-algebraic formalism such as CSP, resulting in a formal framework where assertions about requirements can be proved or refuted with relative ease. CSP's approach to process composition, combined with the fact that refinement is preserved within context, would allow us to decompose a complex design of a system (or system of systems) in such a way that the automated analysis is computationally feasible. In particular, the decompositional approach to specification allows us to substitute a collection of blocks with a single block that depicts the intended behaviour of the whole. Furthermore, CSP's approach to establish refinement — by comparing the behaviour of a characteristic specification process to that of a concrete implementation process — coincides with SysML's compositional outlook to specification and the notion that a block can act as a specification of constituent blocks. In contrast, in conventional model checking approaches where there is no concept of refinement, this distinction is less clear.

7.6.2 Related Work

A formal semantics for some of the SysML diagrams have been given in terms of the *COMPASS Modelling Language* (CML) [**123**]. A set of translation rules is given that maps SysML diagrams to their counterparts in CML. The work presented here is different in that CML integrates state-based as well as process algebraic description techniques. In contrast, our work is solely concerned with defining a process algebraic approach to ensure behavioural conformance amongst the behavioural diagrams of SysML.

Graves and Bijan [**124**] integrated SysML with a higher order type theory logic in an incremental, top-down approach that aims to maintain the logical consistency of the design, with a case study from the aerospace industry. The example provides evidence that coupling formal methods with SysML can realistically be applied to solve aerospace development problems.

7.6.3 Future Work

Ports enable the modeller to specify interfaces between connected blocks. We view a port as being an intermediary between a block and a connector. SysML supports two types of ports: full ports and proxy ports. A full port can alter the item flows before presenting it to the block or an external connector, whereas a proxy port can not. A *proxy port* merely serves as an abstraction interface to the features of the block. Graphically, ports are shown as rectangles on the boundaries of blocks, as per Figure 7.11. Proxy ports are typed by interface blocks. An *interface block* is a specialised type of block that has no associated

classifier behaviour. The structural features are the names of the receptions made available via the proxy port.

In CSP, ports can be modelled as processes placed in parallel with the classifier behaviour of the respective blocks. These port processes can then allow or disallow events based on the interface specification of the port.

SysML makes provision for ports with behavioural features, called *full ports*. In terms of the CSP model, it would be no more difficult to incorporate full ports as we would simply have to place the process modelling the behaviour of the port in parallel with the state machine of the respective block. The approach would allow the modeller to verify the behavioural integrity of the model in much the same manner as we have seen in this chapter.

Similarly, connectors can be assigned a behaviour. Again, a CSP process can be used here to model the connector; it is anticipated that this process would use the same channels as the state machines that it connects.

7.7 Conclusion

In this chapter we presented — within the context of SysML — an approach to specification and design that: is compositional; accommodates the different interpretations of SysML blocks; enables formal requirements traceability; and allows formal consistency checking. The next chapter employs a realistic case study as a means of illuminating and validating the contributions of this thesis.

PART III

CASE STUDY, DISCUSSION AND CONCLUSION

8

Case Study

This chapter presents a case study of a safety critical system as a means of illustrating and validating the work presented in this thesis. Our focus is a real life production cell that has, in the past, served as a benchmark for the evaluation of different specification and design approaches. The work presented in this chapter previously appeared in [**121**].

8.1 A Production Cell

The case study is modelled after an industrial installation of a metal processing plant located in Karlsruhe, Germany. The production cell represents a system with roughly 10^{12} different states, and can therefore be classed as a system of moderate complexity [**20**]. Due to the safety critical, real-time nature of the system, it has been the subject of widespread study, particularly in the formal methods community. This is advantageous from our perspective: detailed formal descriptions of the design exist; and safety and liveness requirements have been formalised extensively.

The specification and requirements that follow summarise the original case study. The interested reader should refer to [**20**] for additional information.

A *production cell* is an automated installation that forms a part of a metal processing plant. The purpose of the cell is to transform metal *blanks*, or *unpressed* plates, into *pressed* plates. The installation consists of several components: a feed belt, an elevating rotary table, a rotary robot, a metal press, a deposit belt, and a travelling crane. An operator places a blank, unpressed plate on the feed belt. The plate then makes its way through the processing cycle until it reaches the deposit belt, at which point it is picked up and placed in a crate by the crane. In the actual production cell the crane links the deposit belt not with a crate, but with yet another cell in the metal processing plant[1]. A top-down view of the production cell is shown in Figure 8.1.

[1] The original case study proposed that the crane be stationed between the deposit and feed belt in order to make the processing cycle cyclical, thus eliminating the need for an operator and a crate.

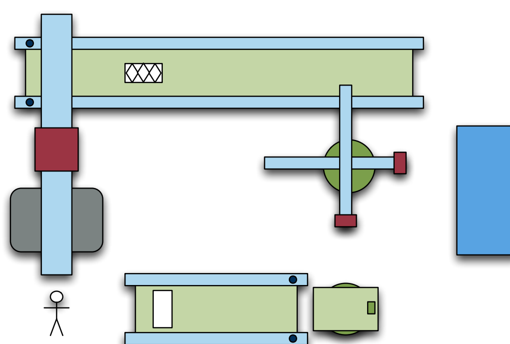


Figure 8.1: Top-down view of the production cell.

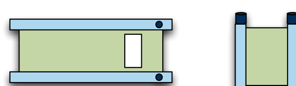


Figure 8.2: Top-down and side views of the feed belt.

8.1.1 Components

In this section we describe the individual components that make up the system of interest.

Feed Belt

The *feed belt* is responsible for conveying a blank, unpressed plate to the table.

A unidirectional electric motor is responsible for providing power to the belt. A photoelectric sensor is placed near the end of the belt in order to indicate the presence or absence of a plate. The top-down and side views of the feed belt are shown in Figure 8.2.

Elevating Rotary Table

The *elevating rotary table* is tasked with transporting blanks between the feed belt and the robot. As the name suggests, it is responsible for both a vertical and horizontal rotational displacement of a plate. In particular:

- the plate is elevated from the lower level of the feed belt to a height suitable for pick up by the upper arm of the robot; and
- the plate is rotated by 90° as the direction in which plates are placed on the feed belt is perpendicular to the direction required by the rest of the processing plant.

It follows that the table can be in one of two positions: the *load* position when a plate is lowered onto the table; or the *unload* position when a plate is picked up by the robot.

The table is powered by two bidirectional electric motors: one responsible for the rotational motion and another for the vertical translation. The top-down and side views of the table are shown in Figure 8.3.

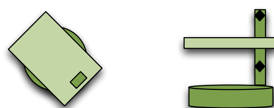


Figure 8.3: Top-down and side views of the elevating rotary table.

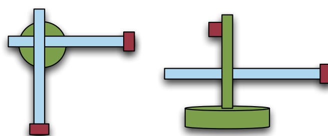


Figure 8.4: Top-down and side views of the rotary robot.

Rotary Robot

The *rotary robot* is tasked with fetching unpressed plates from the table and placing them in the press, and moving forged plates between the press and the deposit belt. It comprises two orthogonal arms placed at different levels in the vertical plane. The arms are tasked with picking up or dropping metal plates. An arm can retract or extend horizontally in order to reach the various cell components (table, press or deposit belt) serviced by that particular arm. In order to service the different components, the robot rotates. Rotational motion of the robot occurs as a unit, consequently both arms rotate jointly. Two arms are necessary in order to increase the throughput of the production cell by maximising the utilisation of the press.

We can identify the following characteristic positions.

- The *initial position*.
- The robot rotated and upper arm extended so that it is able to pick up a plate from the table, i.e. the *upper pick* position.
- The robot rotated and lower arm extended so that it is able to pick up a pressed plate from the press, i.e. the *lower pick* position.
- The robot rotated and lower arm extended so that it is able to drop a pressed plate onto the deposit belt, i.e. the *lower drop* position.
- The robot rotated and upper arm extended so that it is able to place a blank in the metal press, i.e. the *upper drop* position.

Each arm consists of a bidirectional motor and an electromagnet; each arm is additionally fitted with a potentiometer to indicate the range of extension. In addition, the robot has a bidirectional motor responsible for the rotational motion; a potentiometer indicates the angle of rotation. Figure 8.4 shows the top-down and side views of the robot.

Metal Press

The *metal press* is responsible for forging metal blanks. The forging process involves the compression of a plate between two forging surfaces. A metal blank is placed on a

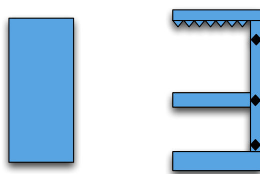


Figure 8.5: Top-down and side views of the press.

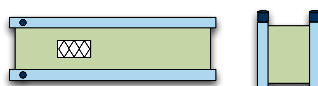


Figure 8.6: Top-down and side views of the deposit belt.

movable forging surface which is displaced along the vertical axis, forging the plate. It follows that the press can be in one of three positions:

- the movable forging surface at a level suitable for loading by the upper robot arm, i.e. the *load* position;
- the movable forging surface at a level suitable for unloading by the lower robot arm, i.e. the *unload* position; or
- the *forge* position, where the movable forging surface is pressed against a static forging surface.

The metal press has a bidirectional motor to power the press; sensors are present as to indicate the position of the forging surface. Figure 8.5 shows the top-down and side views of the press.

Deposit Belt

The *deposit belt* is tasked with conveying pressed plates, unloaded by the robot lower arm, to the travelling crane.

A photoelectric sensor is used to detect the presence of a pressed plate at the end of the belt. A unidirectional motor powers the deposit belt. The top-down and side views of the deposit belt are shown in Figure 8.6.

Travelling Crane

The *travelling crane* is responsible for picking up plates from the end of the deposit belt, and placing them into a crate[2] using a central arm. The purpose of the arm is to perform vertical translations between the deposit belt and crate as their heights are different.

The crane has a bidirectional motor responsible for the horizontal movement of a central arm between the deposit belt and the crate. Sensors are placed above the deposit

[2]For our purposes we assume that the crate never reaches maximum capacity: in the actual metal processing plant the crane transports forged plates to another cell in the manufacturing unit.

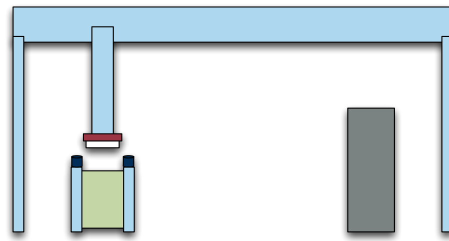


Figure 8.7: Side view of the crane.

belt and crate so as to indicate the position of the arm. The arm is equipped with a bidirectional motor responsible for vertical extension. An electromagnet is placed at the front of the arm for handling plates; a potentiometer is present to indicate the range of extension of the arm. Refer to Figure 8.7 for the side view of the crane.

8.1.2 Operation

We describe the operation cycle of an unpressed plate through the production cell. In order to adequately describe the behaviour of the robot we additionally assume that there is another plate present in the metal press. The complete operation cycle can be summarised thus.

1. The operator places a metal blank onto the feed belt such that the length of the plate is perpendicular to the direction of travel.
2. The feed belt transports the unpressed plate to the elevating rotary table. See Figure 8.8.
3. The table rotates so that the length of the plate is now perpendicular to its original direction. The table adjusts its height so that it is in a position that would allow the robot to unload the plate.
4. The robot rotates from its initial position, as per Figure 8.9, to the upper pick position, as per Figure 8.10. The robot uses its upper arm to pick the plate from the table; this is done by magnetising the magnets positioned at the front of the arm.
5. The robot uses its lower arm to pick a pressed plate from the press, as demonstrated in Figure 8.11.
6. The robot then rotates counterclockwise so that the unpressed plate faces the press and the lower arm the deposit belt.
7. The lower arm of the robot releases the pressed plate onto the deposit belt. Refer to Figure 8.12.
8. The press is assumed to be in a loading position.
9. The upper arm of the robot releases the electromagnet causing the plate to drop in the press, as per Figure 8.13.

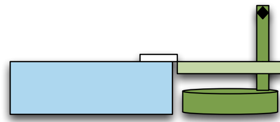
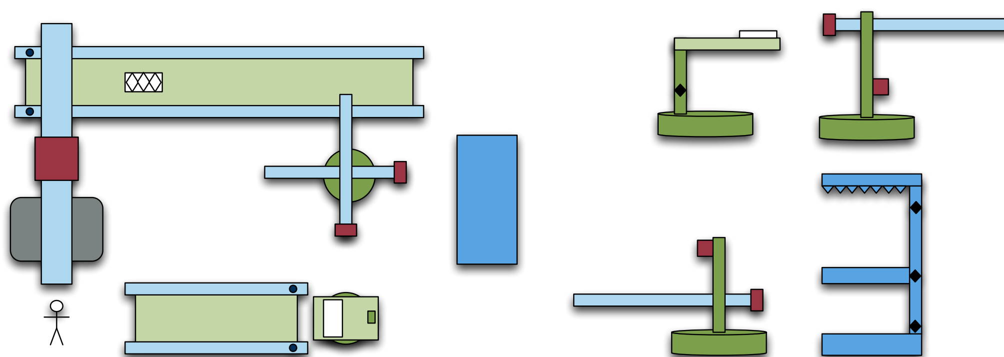


Figure 8.8: Plate conveyed onto the table. The table is shown in the load position.

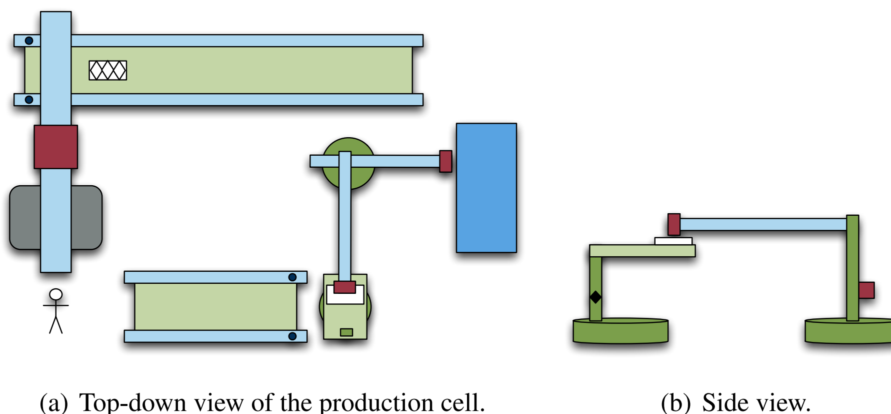
10. The press forges the metal plate.
11. The robot rotates clockwise as to return to its initial position. See Figure 8.9.
12. The upper arm picks up a blank from the table.
13. The lower arm of the robot picks up the pressed plate.
14. The robot rotates counterclockwise and places the pressed plate onto the deposit belt.
15. The deposit belt conveys the plate to the end of the belt.
16. The crane picks up the plate using a central arm. Refer to Figure 8.14.
17. The crane transports the plate to a position above the crate and then releases the electromagnet.

We assume that all arms are retracted during rotations as to allow safe movement. We assume that the rotation and extension constants are known and that the components are positioned so that these constants are adequate. We also assume that the rate at which plates are fed into the system and subsequently processed is such that the timing constraints imposed by the mechanical translations are feasible. Our presentation here (as is the case in the original case study) abstracts away any such concerns.



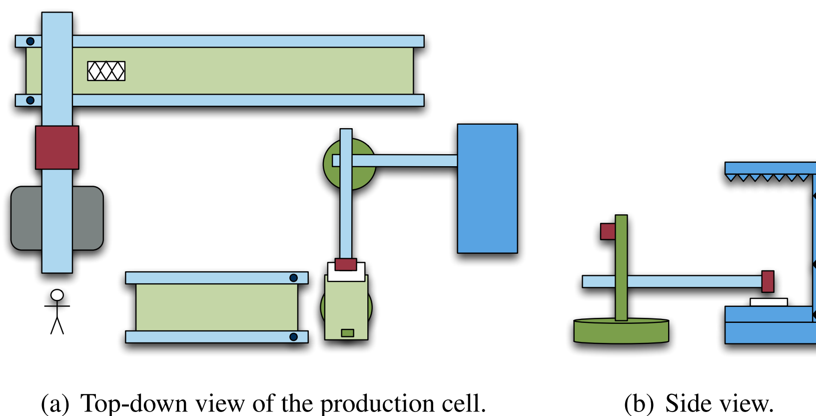
(a) Top-down view of the production cell. (b) Side view.

Figure 8.9: Robot in the initial position.



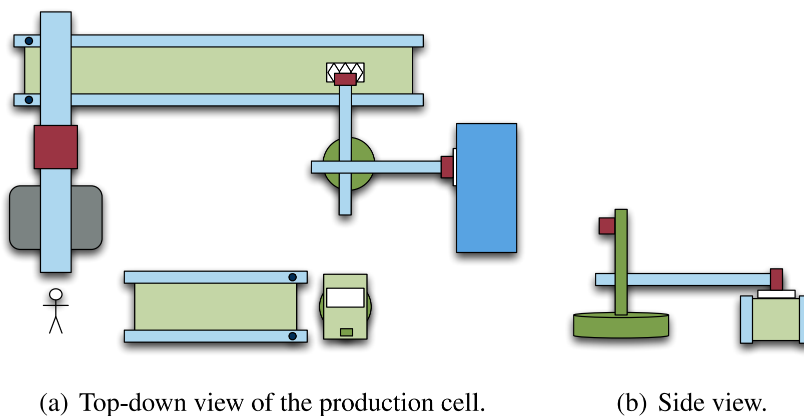
(a) Top-down view of the production cell. (b) Side view.

Figure 8.10: Robot in the position where the upper arm picks a blank plate from the table.



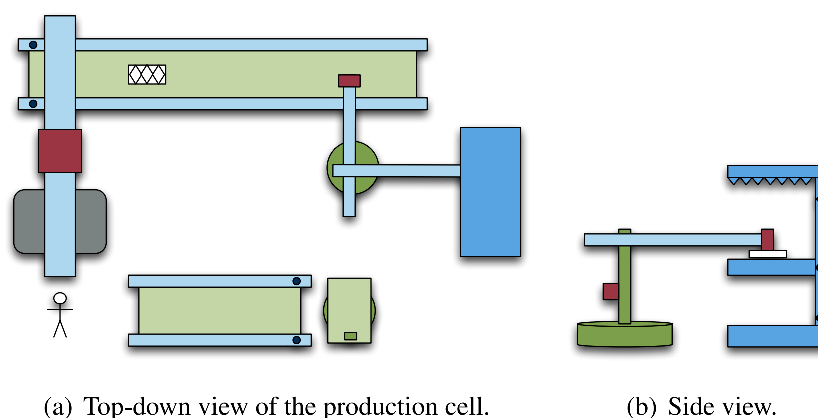
(a) Top-down view of the production cell. (b) Side view.

Figure 8.11: Robot in the position where the lower arm picks a pressed plate from the metal press.



(a) Top-down view of the production cell. (b) Side view.

Figure 8.12: Robot in the position where the lower arm drops a pressed plate onto the deposit belt.



(a) Top-down view of the production cell. (b) Side view.

Figure 8.13: Robot in the position where the upper arm drops a blank plate in the metal press.

8.2 Requirements

In this section we list some requirements to ensure the safe operation of the system. Additionally, we might want to introduce other, non-safety related requirements. For example, a requirement that states that the system does what it is intended to do: forge a plate. As such, we distinguish between safety and other requirements. The requirements listed here are based on those of the original case study.

8.2.1 Safety Requirements

We list several safety classes in this section. For each class, we list only a single requirement due to restrictions on space.

Machine Mobility

This class of safety properties is used to restrict the mobility of machines within safe limits.

> *"If the travelling crane is positioned above the crate, it may only move towards the deposit belt, and if it is positioned above the deposit belt, it may only move towards the crate"*

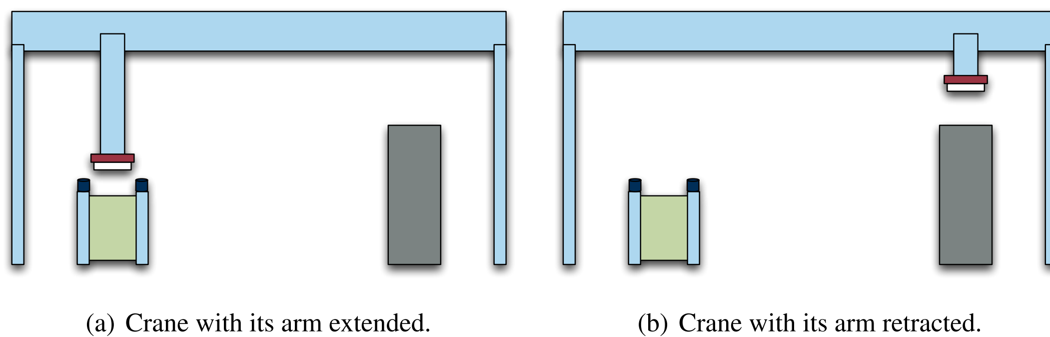
Avoid Collisions

This class of safety properties ensures that there are no collisions between machines that make up the system.

> *"The press may only close when no robot arm is positioned inside it"*

Safe Drops

This class of safety properties avoids injury by only permitting the components to drop metal plates when it is safe to do so.



(a) Crane with its arm extended. (b) Crane with its arm retracted.

Figure 8.14: Side view of the crane's arm in the extended and retracted positions.

> *"The magnet of the crane may only be deactivated, if the robotic arm is above the crate"*

Safe Distance

Erroneous behaviour occurs when plates overlap, or if a plate is placed at the start of one of the conveyor belts before the plate at the end is removed.

> *"Do not put blanks in the press if it is loaded"*

8.2.2 Other Requirements

Requirements that are not safety critical are listed here.

Liveness

We have the following liveness property.

> *"The production cell shall be free from deadlock"*

Forge Plates

We need to ensure that the design models a system that is fit for purpose.

> *"Every blank introduced to the system will eventually be forged"*

8.3 Model

The chosen case study can naturally be decomposed into low level, intermediate, and high level components.

Our approach is as follows.

- We apply the formal semantics proposed in the preceding chapters at each level. This allows us to reason about the behaviour of individual components in isolation; we view every component as a system on its own.

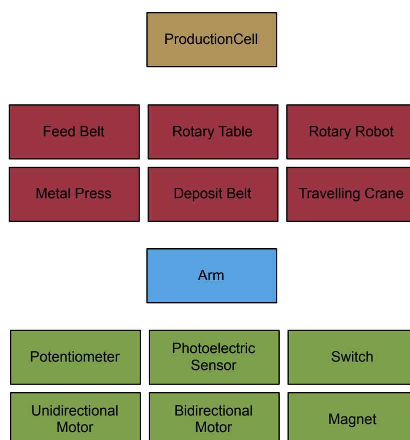


Figure 8.15: Components of the case study separated by level. Low level components are shown in green, intermediate components in blue, and high level components appear in red.

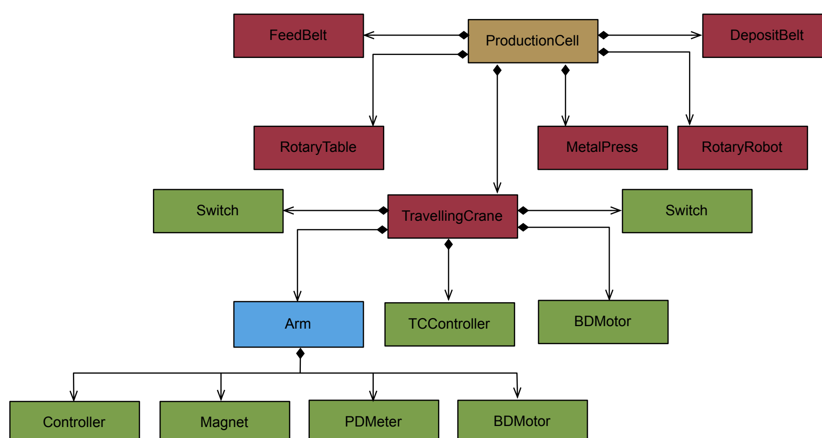


Figure 8.16: Composition of the components. High level components are shown in red, intermediate components in blue, and sensors and actuators in green.

- The concept of refinement is used to establish a link between the behaviours specified at the different levels.

Figure 8.15 shows the components of the case study partitioned: low level components are shown in green, intermediate components in blue, and high level components appear in red.

At the lower level we have basic components, described by simple CSP processes: we view these as the basic building blocks of our system. The behavioural specifications of these leaf level processes are modelled once and then reused as part of more complex processes. The complex processes, in turn, correspond to the components higher up in the hierarchy.

For example, the SysML diagrams corresponding to the magnet can be viewed in isolation, and used to derive a behavioural specification. A similar argument can be made for the potentiometer and the bidirectional motor. The aforementioned processes are then composed in order to describe the behaviour of a robotic arm. The travelling crane, in turn,

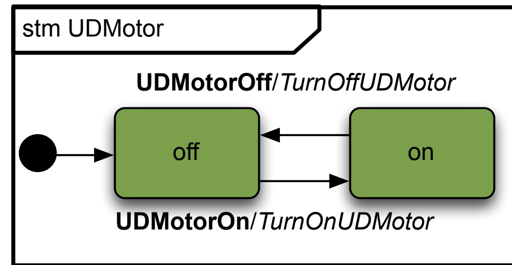


Figure 8.17: Unidirectional motor state machine.

can then be composed from a robotic arm, a bidirectional motor, and two switch processes. The rest of the high level process can be formed similarly, and all of these subsequently combine to form the overall process modelling the production cell. Between each level, however, we use the concept of refinement between an abstract block (modelling the high level behaviour of a component) and its lower level constituent blocks (the specification process made up of the processes corresponding to the concrete implementation blocks).

We have already discussed the robotic arm as part of our running case study, and as such we exclude it from discussion here. In the interest of brevity we discuss two low level and two high level components in the remainder of this section.

8.3.1 Low Level Components

In our design low level components are made up of the sensors and actuators of the case study. We have already encountered behavioural descriptions of sensors and actuators in our running case study.

For each low level component, we present the characteristic state machine, associated activities and corresponding CSP processes associated with each diagram. The alphabets of each CSP process follow the process definition.

Unidirectional motor

The unidirectional motor is an actuator, and the state machine and associated activity diagrams are strikingly similar to that of the bidirectional motor. The state machine diagram of the unidirectional motor is shown in Figure 8.17.

$$
\begin{array}{l}
UDMotor(queue, local) = \\
\quad \text{let} \\
\quad\quad I_0 = OFF \\
\quad\quad OFF = \\
\quad\quad\quad local.proc.UDMotorOn \rightarrow \\
\quad\quad\quad\quad TurnOnUDMotor \fatsemi ON \\
\quad\quad\quad \Box \\
\quad\quad\quad local.disc?e : \{\mid UDMotorOff \mid\} \rightarrow OFF \\
\quad\quad ON = \\
\quad\quad\quad local.proc.UDMotorOff \rightarrow \\
\quad\quad\quad\quad TurnOffUDMotor \fatsemi OFF
\end{array}
$$

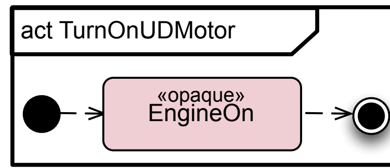


Figure 8.18: TurnOnUDMotor activity.

$$
\begin{array}{l}
\qquad\qquad \Box \\
\qquad\qquad local.disc?e : \{|\ UDMotorOn\ |\} \rightarrow ON \\
\qquad EQ = queue?e \rightarrow local?p!e \rightarrow EQ \\
\quad \text{within} \\
\qquad I_0\ [|\ \{|\ local\ |\}\ |]\ EQ
\end{array}
$$

$$UDMOTOR = UDMotor(udmotor, udmotorlocal)$$

$$
\begin{array}{l}
\alpha UDMOTOR = \\
\quad Union(\{\{|\ udmotor, udmotorlocal\ |\}, \alpha TurnOnUDMotor, \alpha TurnOffUDMotor\})
\end{array}
$$

The activities *TurnOnUDMotor* and *TurnOffUDMotor* are modelled using opaque actions. See Figure 8.18 for the *TurnOnUDMotor* activity.

$$
\begin{array}{l}
TurnOnUDMotor = \\
\quad \text{let} \\
\qquad I_0 = OA_0 \\
\qquad OA_0 = opaque.EngineOn \rightarrow F_0 \\
\qquad F_0 = Skip \\
\quad \text{within} \\
\qquad I_0
\end{array}
$$

$$\alpha TurnOnUDMotor = \{|\ opaque.EngineOn\ |\}$$

$$
\begin{array}{l}
TurnOffUDMotor = \\
\quad \text{let} \\
\qquad I_0 = OA_0 \\
\qquad OA_0 = opaque.EngineOff \rightarrow F_0 \\
\qquad F_0 = Skip \\
\quad \text{within} \\
\qquad I_0
\end{array}
$$

$$\alpha TurnOffUDMotor = \{|\ opaque.EngineOff\ |\}$$

Switch

The state machine corresponding to the switch is similar to that of the potentiometer. Local process *EQ* models the event queue. Figures 8.19 and 8.20 show the state machine and activities of the switch.

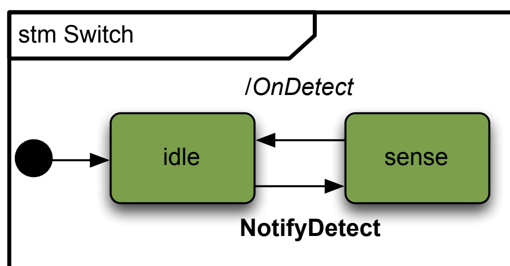


Figure 8.19: Switch state machine.

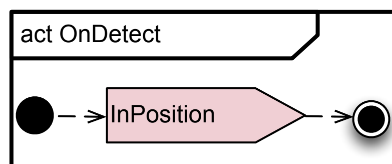


Figure 8.20: OnDetect activity.

$$
\begin{array}{l}
Switch(queue, local) = \\
\quad \text{let} \\
\quad\quad I_0 = IDLE \\
\quad\quad IDLE = \\
\quad\quad\quad local.proc.NotifyDetect \rightarrow SENSE \\
\quad\quad SENSE = \\
\quad\quad\quad OnDetect \fatsemi IDLE \\
\quad\quad\quad \Box \\
\quad\quad\quad local.disc?e : \{| \ NotifyDetect \ |\} \rightarrow SENSE \\
\quad\quad EQ = queue?e \rightarrow local?p!e \rightarrow EQ \\
\quad \text{within} \\
\quad\quad I_0 \ [| \ \{| \ local \ |\} \ |] \ EQ
\end{array}
$$

$$SWITCH = Switch(switch, switchlocal)$$

$$\alpha SWITCH = Union(\{\{| \ switch, switchlocal \ |\}, \alpha OnDetect\})$$

The CSP for *OnDetect* is shown below.

$$
\begin{array}{l}
OnDetect = \\
\quad \text{let} \\
\quad\quad I_0 = SS_0 \\
\quad\quad SS_0 = controller.InPosition \rightarrow F_0 \\
\quad\quad F_0 = Skip \\
\quad \text{within} \\
\quad\quad I_0
\end{array}
$$

Signal	Purpose
RTFBNextPlate	The rotary table asks the feed belt for the next plate.
RTFBInLoadPosition	The rotary table informs the feed belt that it is in the load position.
FBRTPlateLoaded	The feed belt informs the rotary table that the plate was loaded successfully.
RRRTNextPlate	The rotary robot asks the rotary table for the next plate.
RRRTPlateUnloaded	The rotary robot informs the rotary table that the plate was successfully unloaded.
FBRTPlateReady	The feed belt informs the rotary table that the plate is ready.
MPRRInUnloadPosition	The metal press informs the rotary robot that it is in the unload position.
MPRRInLoadPosition	The metal press informs the rotary robot that it is in the load position.
DBRRReady	The deposit belt informs the rotary robot that it is ready to accept a new plate.
RTRRInUnloadPosition	The rotary table informs the rotary robot that it is in the unload position.
RRMPSetLoadPosition	The rotary robot commands the metal press to the load position.
RRMPSetUnloadPosition	The rotary robot commands the metal press to the unload position.
RRMPPress	The rotary robot commands the metal press to forge the plate.
RRDBNextPlate	The rotary robot informs the deposit belt that the next plate is ready.
TCDBPlateUp	The travelling crane informs the deposit belt that the plate is picked up off the end.
TCDBReady	The travelling crane informs the deposit belt that it is ready to pick up another plate.
DBTCPlateEnd	The deposit belt tells the travelling crane that a plate is ready at the end.

Table 8.1: Signals used by the high level components of the production cell.

8.3.2 High Level Components

We briefly list the different signals communicated between the different high level blocks. We followed the convention that a signal is prefixed with two acronyms: the first corresponding to the sending component, and the second to the receiving component. The abbreviations is as follows: the feed belt is abbreviated FB; the elevating rotary table is abbreviated RT; the rotary robot RR; the metal press MP; the deposit belt DB; and the travelling crane TC. Table 8.1 lists the different signals of significance at the high level.

For each high level component, we present the abstract state machine (specification), associated activities and corresponding CSP processes associated with each diagram. The

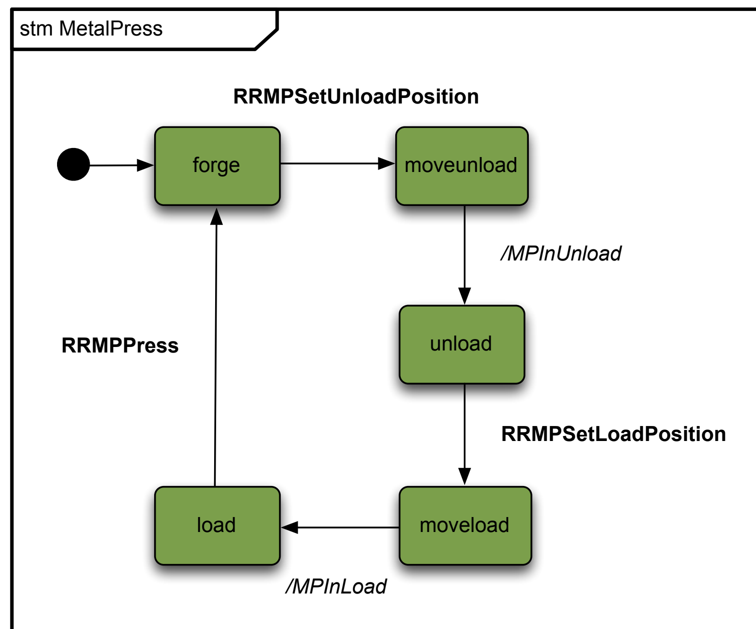


Figure 8.21: Metal press state machine.

alphabets of each CSP process follow the process definition.

Metal Press

The CSP modelling the state machine for the metal press follows. Note how each state of the state machine corresponds to a local CSP process definition. The events not expected in the currently active state (according to the definition of the state machine), are discarded without effect — the state machine does not change state. The state machine for the metal press is shown in Figure 8.21.

$$
\begin{array}{l}
MetalPress(queue, local) = \\
\quad \text{let} \\
\quad\quad I_0 = FORGE \\
\quad\quad FORGE = \\
\quad\quad\quad local.proc.RRMPSetUnloadPosition \rightarrow MOVEUNLOAD \\
\quad\quad\quad \Box \\
\quad\quad\quad local.disc?e : \{| \; RRMPSetLoadPosition, RRMPPress \; |\} \rightarrow FORGE \\
\quad\quad MOVEUNLOAD = \\
\quad\quad\quad MPInUnload \;\text{;}\; UNLOAD \\
\quad\quad\quad \Box \\
\quad\quad\quad local.disc?e : \{| \; RRMPSetLoadPosition, RRMPSetUnloadPosition, \\
\quad\quad\quad\quad\quad\quad RRMPPress \; |\} \rightarrow MOVEUNLOAD \\
\quad\quad UNLOAD = \\
\quad\quad\quad local.proc.RRMPSetLoadPosition \rightarrow MOVELOAD \\
\quad\quad\quad \Box \\
\quad\quad\quad local.disc?e : \{| \; RRMPSetUnloadPosition, RRMPPress \; |\} \rightarrow UNLOAD \\
\quad\quad MOVELOAD =
\end{array}
$$

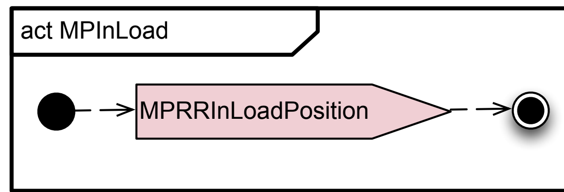


Figure 8.22: MPInLoad activity.

$$
\begin{array}{l}
\quad\quad MPInLoad \,{}_{9}^{\circ}\, LOAD \\
\quad\quad \Box \\
\quad\quad local.disc?e : \{\!|\, RRMPSetLoadPosition, RRMPSetUnloadPosition, \\
\quad\quad\quad\quad\quad\quad RRMPPress \,|\!\} \rightarrow MOVELOAD \\
\quad LOAD = \\
\quad\quad local.proc.RRMPPress \rightarrow FORGE \\
\quad\quad \Box \\
\quad\quad local.disc?e : \{\!|\, RRMPSetLoadPosition, RRMPSetUnloadPosition \,|\!\} \\
\quad\quad\quad \rightarrow LOAD \\
\quad EQ = queue?e \rightarrow local?p!e \rightarrow EQ \\
\text{within} \\
\quad I_0 \, [|\, \{\!|\, local \,|\!\} \,|] \, EQ
\end{array}
$$

$$METALPRESS = MetalPress(metalpress, metalpresslocal)$$

$$
\begin{array}{l}
\alpha METALPRESS = \\
\quad Union(\{\{\!|\, metalpress, metalpresslocal \,|\!\}, \alpha MPInLoad, \alpha MPInUnload\})
\end{array}
$$

The activities that execute within the context of the state machine of the metal press are presented below. *MPInLoad* informs the rotary robot that the metal press is in the load position; similarly, *MPInUnload* indicates that the metal press is currently in the unload position. The activity *MPInLoad* is shown in Figure 8.22.

$$
\begin{array}{l}
MPInLoad = \\
\quad \text{let} \\
\quad\quad I_0 = SS_0 \\
\quad\quad SS_0 = rotaryrobot.MPRRInLoadPosition \rightarrow F_0 \\
\quad\quad F_0 = Skip \\
\quad \text{within} \\
\quad\quad I_0
\end{array}
$$

$$\alpha MPInLoad = \{\!|\, rotaryrobot.MPRRInLoadPosition \,|\!\}$$

$$
\begin{array}{l}
MPInUnload = \\
\quad \text{let} \\
\quad\quad I_0 = SS_0 \\
\quad\quad SS_0 = rotaryrobot.MPRRInUnloadPosition \rightarrow F_0 \\
\quad\quad F_0 = Skip
\end{array}
$$

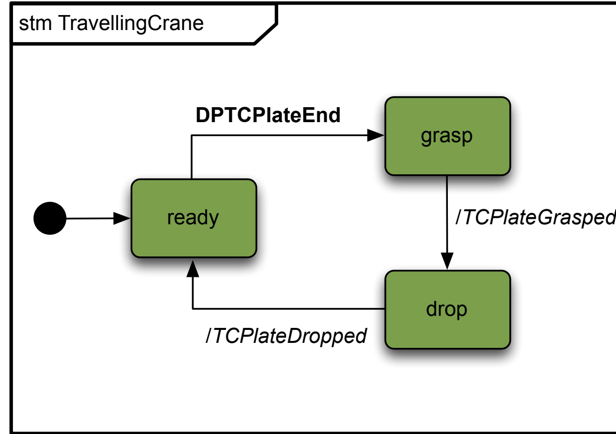


Figure 8.23: Travelling crane state machine.

$$
\begin{array}{l}
\quad \text{within} \\
\qquad I_0
\end{array}
$$

$$\alpha MPInUnload = \{| \, rotaryrobot.MPRRInUnloadPosition \, |\}$$

Travelling Crane

The CSP representation of the travelling crane, again, mirrors that of its corresponding state machine in Figure 8.23.

$$
\begin{array}{l}
TravellingCrane(queue, local) = \\
\quad \text{let} \\
\qquad I_0 = READY \\
\qquad READY = \\
\qquad\quad local.proc.DBTCPlateEnd \to GRASP \\
\qquad GRASP = \\
\qquad\quad TCPlateGrasped \fatsemi DROP \\
\qquad\quad \square \\
\qquad\quad local.disc?e : \{| \, DBTCPlateEnd \, |\} \to GRASP \\
\qquad DROP = \\
\qquad\quad TCPlateDropped \to READY \\
\qquad\quad \square \\
\qquad\quad local.disc?e : \{| \, DBTCPlateEnd \, |\} \to DROP \\
\qquad EQ = queue?e \to local?p!e \to EQ \\
\quad \text{within} \\
\qquad I_0 \, [| \, \{| \, local \, |\} \, |] \, EQ
\end{array}
$$

$$TRAVELLINGCRANE = TravellingCrane(travellingcrane, travellingcranelocal)$$

$$
\begin{array}{l}
\alpha TRAVELLINGCRANE = Union(\{\{| \, travellingcrane, travellingcranelocal \, |\}, \\
\qquad\qquad\qquad\qquad\qquad\quad \alpha TCPlateDropped, \alpha TCPlateGrasped\})
\end{array}
$$

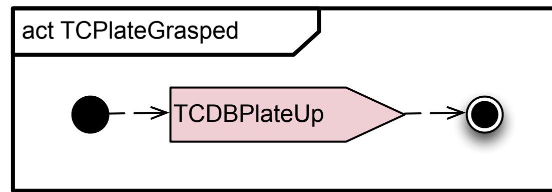


Figure 8.24: MPInLoad activity.

Again, the activities CSP of *TCPlateDropped* and *TCPlateGrasped* are shown below; Figure 8.24 shows the SysML diagram for TCPlateGrasped.

$$
\begin{array}{l}
TCPlateDropped = \\
\quad \text{let} \\
\quad\quad I_0 = SS_0 \\
\quad\quad SS_0 = depositbelt.TCDBReady \rightarrow F_0 \\
\quad\quad F_0 = Skip \\
\quad \text{within} \\
\quad\quad I_0
\end{array}
$$

$$\alpha TCPlateDropped = \{\mid depositbelt.TCDBReady \mid\}$$

$$
\begin{array}{l}
TCPlateGrasped = \\
\quad \text{let} \\
\quad\quad I_0 = SS_0 \\
\quad\quad SS_0 = depositbelt.TCDBPlateUp \rightarrow F_0 \\
\quad\quad F_0 = Skip \\
\quad \text{within} \\
\quad\quad I_0
\end{array}
$$

$$\alpha TCPlateGrasped = \{\mid depositbelt.TCDBPlateUp \mid\}$$

The rest of the components are designed similarly. The high level state machine for each of the components serves as the abstract specification that the concrete implementation, formed from the process algebraic composition of the components, must satisfy.

The interested reader is referred to the appendices for more on the diagrams and associated CSP related to the complete case study.

8.4 Results

We evaluated the effectiveness and suitability of the formal translations presented in Chapters 4–7 within the context of a moderately complex system.

Low Level Components

We derived process definitions for each of the six low level components. Using FDR and process animators, we were able to verify that the behaviour of the individual components

were as expected. We additionally performed refinement checks to ensure freedom from deadlock.

Intermediate Components

The robotic arm, our running case study, is the only intermediate component. We performed a refinement check between the behavioural specification of the arm block (abstract block), and the sensory and actuator components that make up the whole (concrete implementation). In addition, we performed a check to ensure that the composite implementation is free form deadlock. It follows that we performed two refinement checks.

High Level Components

High level components consist of a combination of sensors, actuators, and the robotic arm. There are six high level components: the feed belt, the elevating rotary table, the rotary robot, the metal press, the deposit belt, and the travelling crane. If we performed a deadlock analysis and characteristic refinement check on each component, it follows that we have 12 refinement checks at this level.

Summary

We performed a total of 21 refinement checks in order to assert behavioural conformance of the model: six low level refinement checks, two refinement checks related to the intermediate level, 12 refinement checks for the high level components, and a liveness check for the complete system. In addition, for each requirement assigned a behavioural formalism and subsequently expressed via a characteristic process, an additional refinement check is possible.

The compositional nature of CSP allowed us to partition the problem across the different levels. A similar approach would not be as straightforward with conventional model checkers, where there is no concept of refinement. Furthermore, conventional model checkers operate by checking properties, specified in a temporal logic, against the formal model. The key observation here is that the language used to specify the formal model and the language used to assert safety properties are different. In contrast, CSP and FDR utilise a characteristic behavioural process (the specification process) to compare against the concrete implementation process. We should point out that in conventional model checking approaches, the modeling language — used to specify the formal model — is typically quite close to the implementation language. It is still the case that temporal logics can be a barrier to those not familiar with formal methods.

Ultimately, we are able to break down the problem due to FDR's ability to compare process behaviours against each other, and the compositional nature of CSP. The ability to check the conformance of the behaviours between different levels, using the principle of refinement, is an elegant way to address complexity and alleviate the state space explosion problem.

The refinement checks related to liveness are presented below.

assert $PRODUCTIONCELL : [deadlockfree[F]]$
assert $TRAVELLINGCRANE : [deadlockfree[F]]$
assert $DEPOSITBELT : [deadlockfree[F]]$

assert $METALPRESS$: $[deadlock free[F]]$
assert $ROTARYROBOT$: $[deadlock free[F]]$
assert $ROTARYTABLE$: $[deadlock free[F]]$
assert $FEEDBELT$: $[deadlock free[F]]$
assert ARM : $[deadlock free[F]]$
assert $PESENSOR$: $[deadlock free[F]]$
assert $SWITCH$: $[deadlock free[F]]$
assert $PDMETER$: $[deadlock free[F]]$
assert $BDMOTOR$: $[deadlock free[F]]$
assert $UDMOTOR$: $[deadlock free[F]]$
assert $MAGNET$: $[deadlock free[F]]$

Refinement checks linking the respective layers can be stated thus.

assert $TRAVELLINGCRANE \sqsubseteq_F TRAVELLINGCRANECONCRETE$
assert $DEPOSITBELT \sqsubseteq_F DEPOSITBELTCONCRETE$
assert $METALPRESS \sqsubseteq_F METALPRESSCONCRETE$
assert $ROTARYROBOT \sqsubseteq_F ROTARYROBOTCONCRETE$
assert $ROTARYTABLE \sqsubseteq_F ROTARYTABLECONCRETE$
assert $FEEDBELT \sqsubseteq_F FEEDBELTCONCRETE$
assert $ARM \sqsubseteq_F ARMCONCRETECONCRETE$

In the above, the composition of *TRAVELLINGCRANECONCRETE* is given by

$$TRAVELLINGCRANECONCRETE = \| \, p : P \bullet [\alpha p] p$$

where

$$P = \{TCCONTROLLER, ARM, BDMOTOR, SWITCH, SWITCH'\}$$

The processes modelling the concrete composition of other high level blocks are similar.

8.5 Conclusion

One of the biggest difficulties encountered during the specification and design of the case study was coming up with a coherent SysML model. We followed an iterative approach:

- We started with a high level behavioral specification of the constituent components of the production cell: feed belt, elevating rotary table, rotary robot, metal press, deposit belt and traveling crane. We then decomposed each of the high level components into its constituent parts and came up with corresponding behavioral specifications. The state machines were then translated into their corresponding CSP processes. This process was iterative, as the resulting CSP descriptions pointed out inconsistencies in our original SysML model. These included:

  1. a missing triggering event on transitions of the target state machine, resulting in the state machine not responding to received events (instead the event is consumed and discarded by the target state machine);

2. a missing send event in the activities corresponding to state-based or transition related behaviors (the target state machine remains in an incorrect state);
3. the combined behavior of the concrete implementation state machines do not conform to the behavioral specification of the abstract block's classifying state machine;
4. the violation of liveness properties; and
5. the violation of safety properties.

The first three items above essentially imply the incorrect formulation of the SysML model, which was then faithfully translated. This was where we encountered the majority of the problems: we had initially specified incompatible behavioral specifications (in SysML) that did not function together as originally intended. FDR and refinement allowed us to address these issues.

- Another minor problem we encountered was erroneously translating between the source and target models. However, we these errors were usually picked up when using the process animator to explore the resulting behaviour. An automated translation approach would conceivably eliminate such problems (assuming a correct implementation).

We roughly took two months to come up with both the CSP and SysML models.

9

DISCUSSION AND CONCLUSION

This chapter summarises the contributions of the thesis, comments on the weaknesses and limitations of the various techniques employed, and outlines possible avenues of further work.

9.1 Contributions

In this section we set out to answer the research questions stated in Chapter 1 as well as summarise the contributions of the thesis.

Research Questions

> *"Can a formal refinement framework, defined over SysML diagrams, be helpful in pointing out inconsistencies in a SysML model, and ultimately lead to a more cohesive and well-formed SysML model?"*

We provided formal semantics for state machines, activities and interactions in Chapters 4–6. These constructs were evaluated within the context of a systems engineering methodology, and subsequently applied to a case study of sizeable complexity. Our semantics emphasised the communication model of individual communicating state machines. Moreover, activities, frequently used in SysML to augment state machine behaviour, were firmly integrated with our approach. Interactions were then used to validate possible scenarios between different components. The use of refinement checking, as well as state exploration techniques, were invaluable in pointing out behavioural inconsistencies. These techniques did not only provide a level of assurance in the behavioural model of the design, but also led to a more cohesive and well-formed SysML model.

The type of inconsistencies that can be detected can broadly be summarised thus.

- Inconsistencies between the classifier behaviour of a concrete block, and the combined behaviour of the concrete implementation blocks.
- Inconsistencies between communicating state machines (for example missing send or receive signal events).

- Deadlocks can be detected with ease.
- Requirements can be verified against the model via formal requirements traceability: safety and functional requirements are best suited to our approach.

  "Is it possible to define a framework that supports requirements traceability through formal refinement checking?"

Requirements are an integral part of SysML: a requirement diagram is used to depict, at a graphical level, relationships between different requirements. In this thesis we formalised these relationships in terms of the refinement relation of CSP. Requirements traceability is undoubtedly a good thing: a formal representation of requirement diagrams in CSP gives rise to a formal means of requirements traceability. This provides a formal foundation for this otherwise informal technique that serves as justification for the development of a formal framework.

Research Contributions

The research contributions, introduced in Chapter 1, can be summarised thus.

1. We defined a formal, comprehensive, behavioural semantics for SysML in terms of CSP. The contribution is novel in that it considers a substantive subset of the behavioural constructs in terms of a unified, overarching framework. This differs from previous contributions where the emphasis was placed on a single, isolated, diagram.
2. We used refinement to detect inconsistencies in SysML models. We considered the combined behaviours of different diagram kinds — state machines, activity diagrams and sequence diagrams — within the context of SysML.
3. We used a refinement framework to automate requirements traceability in SysML models. Requirements were formalised via SysML behavioural diagrams. Refinement was then used as a basis for determining whether the stated requirements imposed on the model held.

The overarching framework defined above also had the property of compositionality. A SysML block is composed out of other blocks, termed parts. Once a refinement relation between a block and its parts had been established, the block could be used in the place of its parts higher up in the system architecture. This approach, due to the compositional nature of CSP, offers substantial advantage over traditional model checking approaches. The use of this compositional approach is demonstrated as part our substantive case study that serves as a secondary research contribution.

Secondary research contributions can be summarised thus.

1. We provided a detailed account of past and current research on the topic of integrating process-algebraic approaches with behavioural formalisms.
2. We evaluated our methodology within the context of a safety critical system of moderate size and complexity. To the best of our knowledge, this is the first time an integrated formal approach has been validated in this context: other approaches are typically demonstrated only on a few diagrams.

9.2 Shortcomings and Limitations

The shortcomings and limitations of our semantics for each SysML diagram are discussed in the respective chapters, and, as such, are not repeated here.

Currently, the process of translating SysML diagrammatic constructs into CSP processes is a manual one. As long as this remains the case, questions can and will (quite correctly) be raised about the practicality of our approach. For example, errors in such translations may lead to the possibility of falsely asserting refinement. However, we have made significant efforts in defining a precise mapping between the two paradigms. Our emphasis in this thesis was on defining a comprehensive behavioural semantics for SysML in terms of CSP, rather than on the development of a toolset that implemented our framework. However, we do not foresee any significant problems with implementing the proposed framework: model-to-model transformations can be used to translate between meta-models of our source and target notations. For example, we can use the *Epsilon Transformation Language* (ETL) [**125**] to translate between meta-models of SysML and CSP. A subsequent model-to-text transformation, using the *Epsilon Generation Language* (EGL) [**125**], can then be used to translate the CSP meta-model into machine-readable CSP.

9.3 Future Work

It is possible to extend the work presented in this thesis in several ways.

- Generalisation allows behavioural features of blocks to be inherited or overwritten. It would be interesting to see if this can be taken into account and how this can be verified using CSP. Although generalisation is possible in SysML, it is not frequently encountered, and was therefore excluded from consideration. Furthermore, due to the fact that the characteristic behaviour of blocks are usually formalised using state machines, the investigation would have to consider how generalisation is applicable to state machines within the context of SysML.

- It might be possible to model some of the structural aspects using tools better suited for modelling state-based behaviour, such as the Alloy [**94**] model finder. In particular, it would be interesting to see if tools such as Alloy are useful in verifying OCL [**72**] constraints imposed on SysML models. Parametrics, another feature of SysML, used to place performance or physical constraints on blocks within a certain context, would be worth investigating.

- Timed CSP [**112**] extensions of our work would allow for the modelling of temporal information on sequence diagrams. The sequence diagrams in our formalisation merely considered the order of occurrence observations: any reference to the physical time intervals between observations on a lifeline were abstracted away. Untimed CSP is well suited to modelling at this level of abstraction. However, it is possible to explicitly impose timing constraints on sequence diagrams — shown using a standard constraint expression in braces and attached by a line to the constrained occurrence. It would be possible to use Tock CSP [**15**], where, in standard CSP, spe-

cial tick events denote the passage of time. Timed CSP, however, would be better suited to model sequence diagrams with explicit timing constraints.

- State machine diagrams can be extended to incorporate timed events. Again, Tock CSP can potentially be used to model such state machines, but timed extensions might be better suited.

- Investigating the integration of SysML and methodologies such as Circus [**126**] or CSP ‖ B [**68**] would be worthwhile. CSP does not allow for capturing state information in an eloquent fashion; notations such as B [**33**] or Z [**59**] are more appropriate.

- The probabilistic nature of SysML activities can be formalised with the help of model checkers such as PRISM [**116**]. The interested reader is referred to the work of Debbabi *et al.* [**127**]. Alternatively, probabilistic extensions of CSP [**114**; **115**] might be used to model probabilities on activity diagrams.

- The automatic translation between meta-models of SysML and CSP is another avenue of work worth pursuing. The development of a model-driven framework that automates the translation process, paired with investigations into how best to shield the SysML modeller from the underlying formal method, is a notable goal. For example, one way to isolate the modeller from the CSP domain is to provide feedback in the form of a sequence diagram, rather than a CSP counterexample. The benefit of this approach would enable the modeller to focus on the task at hand — the specification of a suitable SysML model — and would require no detailed knowledge of the underlying formal method.

9.4 Concluding Remarks

This thesis used CSP and SysML to demonstrate the use of formal methods within the context of systems modelling. We utilised the abstract syntax of the diagrammatic constructs found in SysML to underpin the translation between the different notations. In addition to providing a level of assurance in the validity of the design, the use of a formal description technique also provides the ability to validate that requirements are maintained in the model. The work of this thesis has provided the first steps towards the development of a framework that facilitates the automated translation between a significant subset of the behavioural features of SysML and the process algebra CSP.

List of References

[**1**] SysML: *Systems Modeling Language Specification, version 1.3*. Object Management Group, 2012. Available at: `http://www.omg.org/spec/SysML/1.3`, [2014, March].

[**2**] UML: *Unified Modeling Language Specification, version 2.4.1*. Object Management Group, 2011. Available at: `http://www.omg.org/spec/UML/2.4.1`, [2014, March].

[**3**] IEEE 1076: *IEEE 1076-2008: VHSIC Hardware Description Language Specification*. IEEE, 2008. Available at: `http://standards.ieee.org/findstds/standard/1076-2008.html`, [2014, March].

[**4**] Hoare, C.A.R.: *Communicating Sequential Processes*. Prentice Hall, 1985. ISBN 978-0-13-153271-7.

[**5**] Roscoe, A.W.: Model-checking CSP. In: *A Classical Mind: Essays in Honour of C.A.R. Hoare*, chap. 21. Prentice Hall, 1994. ISBN 978-0-13-294844-9.

[**6**] Armstrong, P., Goldsmith, M.H., Lowe, G., Ouaknine, J., Palikareva, H., Roscoe, A.W. and Worrell, J.: Recent developments in FDR. In: *Proceedings of the 24th International Conference on Computer Aided Verification (CAV 2012)*, vol. 7358 of *Lecture Notes in Computer Science*, pp. 699–704. Springer, 2012. ISBN 978-3-642-31424-7.

[**7**] FDR2: *Failures Divergences Refinement User Manual, version 2.94*. Department of Computer Science, University of Oxford, 2012. Available at: `http://www.cs.ox.ac.uk/projects/concurrency-tools/download/fdr2manual-2.94.pdf`, [2014, March].

[**8**] FDR3: *Failures Divergences Refinement User Manual, version 3*. Department of Computer Science, University of Oxford, 2013. Available at: `https://www.cs.ox.ac.uk/projects/fdr/manual/`, [2014, March].

[**9**] Lowe, G.: Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In: *Proceedings of the 2nd International Workshop on Tools and Algorithms for Construction and Analysis of Systems (TACAS 1996)*, vol. 1055 of *Lecture Notes in Computer Science*, pp. 147–166. Springer, 1996. ISBN 978-3-540-61042-7.

[**10**] Lowe, G. and Roscoe, A.W.: Using CSP to detect errors in the TMN protocol. *IEEE Transactions on Software Engineering*, vol. 23, no. 10, pp. 659–669, 1997. ISSN 0098-5589.

[**11**] Simpson, A.C.: *Safety through Security*. Doctor of Philosophy thesis, Department of Computer Science, University of Oxford, 1996.

[**12**] McInnes, A.I.S.: *A Formal Approach to Specifying and Verifying Spacecraft Behavior*. Doctor of Philosophy thesis, Department of Electrical and Computer Engineering, Utah State University, 2007.

[**13**] Selic, B.: Using UML for modeling complex real-time systems. In: *Proceedings of the 1998 Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES 1998)*, vol. 1474 of *Lecture Notes in Computer Science*, pp. 250–260. Springer, 1998. ISBN 978-3-540-65075-1.

[**14**] Friedenthal, S., Moore, A. and Steiner, R.: *A Practical Guide to SysML: The Systems Modeling Language*. Morgan Kaufmann Publishers, 2008. ISBN 978-0-12-385206-9.

[**15**] Roscoe, A.W.: *The Theory and Practice of Concurrency*. Prentice Hall, 1997. ISBN 978-0-13-674409-2.

[**16**] Sangiorgi, D. and Walker, D.: *The π-calculus: A Theory of Mobile Processes*. Cambridge University Press, 2003. ISBN 978-0-521-54327-9.

[**17**] Roscoe, A.W.: *Understanding Concurrent Systems*. Springer, 2010. ISBN 978-1-84882-257-3.

[**18**] Holt, J. and Perry, S.: *SysML for Systems Engineering*. The Institution of Engineering and Technology, 2008. ISBN 978-0-86341-825-9.

[**19**] Schneider, S.A.: *Concurrent and Real Time Systems: The CSP Approach*. John Wiley & Sons, 1999. ISBN 978-0-471-62373-1.

[**20**] Lewerentz, C. and Lindner, T.: Case study Production Cell. In: *Formal Development of Reactive Systems*, vol. 891 of *Lecture Notes in Computer Science*. Springer, 1995. ISBN 978-3-540-49133-0.

[**21**] Estefan, J.A.: *A Survey of Model-Based Systems Engineering (MBSE) Methodologies*. Technical report INCOSE-TD-2007-003-02, International Council on Systems Engineering, 2008.

[**22**] ISO/IEC 15288: *ISO/IEC 15288-2008: Systems and Software Engineering — System Life Cycle Processes*. International Standards Organization, 2008. Available at: `http://www.iso.org`, [2014, March].

[**23**] IEEE 1220: *IEEE 1220-2005: Application and Management of the Systems Engineering Process*. IEEE, 2005. Available at: `https://standards.ieee.org/findstds/standard/1220-2005.html`, [2014, March].

[**24**] Haskins, C. (ed.): *Incose Systems Engineering Handbook*. Incose, 2012. ISBN 978-1-937076-02-3.

[**25**] Baier, C. and Katoen, J.-P.: *Principles Of Model Checking*. MIT Press, 2008. ISBN 978-0-262-02649-9.

[**26**] Sun, J., Liu, Y. and Dong, J.S.: Model checking CSP revisited: introducing a Process Analysis Toolkit. In: *Proceedings of the 3rd International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2008)*, vol. 17 of *Communications in Computer and Information Science*, pp. 307–322. Springer, 2008. ISBN 978-3-540-88479-8.

[**27**] Kitchenham, B., Charters, S., Budgen, D., Brereton, P., Turner, M., Linkman, S., Jørgensen, M., Mendes, E. and Visaggio, G.: *Guidelines for Performing Systematic Literature Reviews in Software Engineering*. Technical report EBSE-2007-01, School of Computer Science and Mathematics, Keele University & Department of Computer Science, University of Durham, 2007.

[**28**] Genero, M., Fernández-Saez, A.M., Nelson, H.J., Poels, G. and Piattini, M.: A systematic literature review on the quality of UML models. *Journal of Database Management*, vol. 22, no. 3, pp. 46–70, 2011. ISSN 1063-8016.

[**29**] Usman, M., Nadeem, A., Kim, T.-H. and Cho, E.-S.: A survey of consistency checking techniques for UML models. In: *Proceedings of the 2008 Advanced Software Engineering and its Applications (ASEA 2008)*, pp. 57–62. IEEE, 2008. ISBN 978-0-7695-3432-9.

[**30**] Lucas, F.J., Molina, F. and Toval, A.: A systematic review of UML model consistency management. *Information and Software Technology*, vol. 51, no. 12, pp. 1631–1645, 2009. ISSN 0950-5849.

[**31**] Harel, D. and Naamad, A.: The Statemate semantics of statecharts. *ACM Transactions on Software Engineering and Methodology*, vol. 5, no. 4, pp. 293–333, 1996. ISSN 1049-331X.

[**32**] Leuschel, M. and Butler, M.: ProB: a model checker for B. In: *Proceedings of the 12th International Symposium of Formal Methods Europe (FME 2003)*, vol. 2805 of *Lecture Notes in Computer Science*, pp. 855–874. Springer, 2003. ISBN 978-3-540-45236-2.

[**33**] Schneider, S.A.: *The B-Method: An Introduction*. Macmillan Publishers, 2001. ISBN 978-0-333-79284-1.

[**34**] Cleaveland, R., Parrow, J. and Steffen, B.: The Concurrency Workbench. In: *Proceedings of the 1989 International Workshop on Automatic Verification Methods for Finite State Systems*, vol. 407 of *Lecture Notes in Computer Science*, pp. 24–37. Springer, 1990. ISBN 978-3-540-52148-8.

[**35**] Milner, R.: *A Calculus of Communicating Systems*. Springer, 1982. ISBN 978-0-387-10235-1.

[**36**] Victor, B. and Moller, F.: *The Mobility Workbench: A Tool for the π-calculus*. Technical report ECS-LFCS-94-285, School of Informatics, The University of Edinburgh, 1994.

[**37**] McMillan, K.L.: *Symbolic Model Checking: An Approach to the State Explosion Problem*. Doctor of Philosophy thesis, School of Computer Science, Carnegie Mellon University, 1992.

[**38**] Cimatti, A., Clarke, E., Giunchiglia, F. and Roveri, M.: NuSMV: a new symbolic model checker. In: *Proceedings of the 11th International Conference on Computer Aided Verification (CAV 1999)*, vol. 1633 of *Lecture Notes in Computer Science*, pp. 495–499. Springer, 1999. ISBN 978-3-540-66202-0.

[**39**] Holzmann, G.J.: The model checker SPIN. *IEEE Transactions on Software Engineering*, vol. 23, no. 5, pp. 279–295, 1997. ISSN 0098-5589.

[**40**] Bengtsson, J., Larsen, K.G., Larsson, F., Pettersson, P. and Yi, W.: UPPAAL — a tool suite for automatic verification of real-time systems. In: *Proceedings of the 1995 Workshop on Verification and Control of Hybrid Systems*, vol. 1066 of *Lecture Notes in Computer Science*, pp. 232–243. Springer, 1995. ISBN 978-3-540-61155-4.

[**41**] Alur, R. and Dill, D.: Automata for modeling real-time systems. In: *Proceedings of the 17th International Colloquium on Automata, Languages and Programming (ICALP 1990)*, vol. 443 of *Lecture Notes in Computer Science*, pp. 322–335. Springer, 1990. ISBN 978-3-540-52826-5.

[**42**] Abdelhalim, I., Schneider, S.A. and Treharne, H.: An integrated framework for checking the behaviour of fUML models using CSP. *International Journal on Software Tools for Technology Transfer*, 2012. ISSN 1433-2787.

[**43**] Abdelhalim, I., Schneider, S.A. and Treharne, H.: Towards a practical approach to check UML/fUML models consistency using CSP. In: *Proceedings of the 13th International Conference on Formal Engineering Methods (ICFEM 2011)*, vol. 6991 of *Lecture Notes in Computer Science*, pp. 33–48. Springer, 2011. ISBN 978-3-642-24558-9.

[**44**] fUML: *Foundational Subset for Executable UML Models Specification, version 1.1*. Object Management Group, 2013. Available at: `http://www.omg.org/spec/FUML/1.1`, [2014, March].

[**45**] Dan, L.: QVT based model transformation from sequence diagram to CSP. In: *Proceedings of the 15th International Conference on Engineering of Complex Computer Systems (ICECCS 2010)*, pp. 349–354. IEEE, 2010. ISBN 978-1-4244-6639-9.

[**46**] Ng, M.Y. and Butler, M.: Towards formalizing UML state diagrams in CSP. In: *Proceedings of the 1st International Conference on Software Engineering and Formal Methods (SEFM 2003)*, pp. 138–147. IEEE, 2003. ISBN 978-0-7695-1949-4.

[**47**] Davies, J.W.M. and Crichton, C.R.: Concurrency and refinement in the Unified Modeling Language. *Electronic Notes in Theoretical Computer Science*, vol. 70, no. 3, pp. 217–243, 2002. ISSN 1571-0661.

[**48**] Bolton, C. and Davies, J.W.M.: Using relational and behavioural semantics in the verification of object models. In: *Proceedings of the 4th International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS 2000)*, vol. 49 of *IFIP Advances in Information and Communication Technology*, pp. 163–182. Springer, 2000. ISBN 978-1-4757-1018-2.

[**49**] Küster, J.M.: Towards inconsistency handling of object-oriented behavioral models. *Electronic Notes in Theoretical Computer Science*, vol. 109, pp. 57–69, 2004. ISSN 1571-0661.

[**50**] Küster, J.M. and Engels, G.: Consistency management within model-based object-oriented development of components. In: *Proceedings of the 2nd International Symposium on Formal Methods for Components and Objects (FMCO 2003)*, vol. 3188 of *Lecture Notes in Computer Science*, pp. 157–176. Springer, 2004. ISBN 978-3-540-22942-1.

[**51**] Engels, G., Heckel, R. and Küster, J.M.: The Consistency Workbench: a tool for consistency management in UML-based development. In: *Proceedings of the 6th International Conference on The Unified Modeling Language: Modeling Languages and Applications (UML 2003)*, vol. 2863 of *Lecture Notes in Computer Science*, pp. 356–359. Springer, 2003. ISBN 978-3-540-20243-1.

[**52**] Küster, J.M. and Stroop, J.: Consistent design of embedded real-time systems with UML-RT. In: *Proceedings of the 4th International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2001)*, pp. 31–40. IEEE, 2001. ISBN 978-0-7695-1089-7.

[**53**] Engels, G., Küster, J.M., Heckel, R. and Groenewegen, L.: A methodology for specifying and analysing consistency of object-oriented behavioral models. *ACM SIGSOFT Software Engineering Notes*, vol. 26, no. 5, pp. 186–195, 2001. ISSN 0163-5948.

[**54**] Engels, G., Heckel, R., Küster, J.M. and Groenewegen, L.: Consistency-preserving model evolution through transformations. In: *Proceedings of the 5th International Conference on The Unified Modeling Language: Model Engineering, Concepts, and Tools (UML 2002)*, vol. 2460 of *Lecture Notes in Computer Science*, pp. 212–227. Springer, 2002. ISBN 978-3-540-44254-7.

[**55**] Engels, G., Küster, J.M., Heckel, R. and Groenewegen, L.: Towards consistency-preserving model evolution. In: *Proceedings of the 2002 International Workshop on Principles of Software Evolution (IWPSE 2002)*, pp. 22–28. ACM, 2002. ISBN 978-1-58113-545-9.

[**56**] Heckel, R. and Küster, J.M.: Behavioral constraints for visual models. *Electronic Notes in Theoretical Computer Science*, vol. 50, no. 3, pp. 257–265, 2001. ISSN 1571-0661.

[**57**] Ng, M.Y. and Butler, M.: Tool support for visualizing CSP in UML. In: *Proceedings of the 4th International Conference on Formal Engineering Methods (ICFEM 2002)*, vol. 2495 of *Lecture Notes in Computer Science*, pp. 287–298. Springer, 2002. ISBN 978-3-540-00029-7.

[**58**] Rasch, H. and Wehrheim, H.: Checking consistency in UML diagrams: classes and state machines. In: *Proceedings of the 6th International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS 2003)*, vol. 2884 of *Lecture Notes in Computer Science*, pp. 229–243. Springer, 2003. ISBN 978-3-540-20491-6.

[**59**] Woodcock, J.C.P. and Davies, J.W.M.: *Using Z: Specification, Refinement, and Proof*. Prentice Hall, 1996. ISBN 978-0-13-948472-8.

[**60**] Fischer, C.: CSP-OZ: a combination of Object-Z and CSP. In: *Proceedings of the 1997 International Workshop on Formal Methods for Open Object-Based Distributed Systems (FMOODS 1997)*, pp. 423–438. Chapman & Hall, 1997. ISBN 978-0-412-82040-3.

[**61**] Rasch, H. and Wehrheim, H.: Checking the validity of scenarios in UML models. In: *Proceedings of the 7th International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS 2005)*, vol. 3535 of *Lecture Notes in Computer Science*, pp. 67–82. Springer, 2005. ISBN 978-3-540-26181-0.

[**62**] Varró, D., Asztalos, M., Bisztray, D., Boronat, A., Dang, D.-H., Geiß, R., Greenyer, J., Van Gorp, P., Kniemeyer, O., Narayanan, A., Rencis, E. and Weinell, E.: Transformation of UML models to CSP: a case study for graph transformation tools. In: *Proceedings of the 3rd International Symposium on Applications of Graph Transformations (AGTIVE 2007)*, vol. 5088 of *Lecture Notes in Computer Science*, pp. 540–565. Springer, 2008. ISBN 978-3-540-89019-5.

[**63**] Xu, D., Philbert, N., Liu, Z. and Liu, W.: Towards formalizing UML activity diagrams in CSP. In: *Proceedings of the 2008 International Symposium on Computer Science and Computational Technology (ISCSCT 2008)*, pp. 450–453. IEEE, 2008. ISBN 978-1-4244-3746-7.

[**64**] Xu, D., Miao, H. and Philbert, N.: Model checking UML activity diagrams in FDR. In: *Proceedings of the 8th International Conference on Computer and Information Science (ICIS 2009)*, pp. 1035–1040. IEEE, 2009. ISBN 978-0-7695-3641-5.

[**65**] Yeung, W.L., Leung, K.R.P.H., Dong, W. and Wang, J.: Improvements towards formalizing UML state diagrams in CSP. In: *Proceedings of the 12th Asia-Pacific Software Engineering Conference (APSEC 2005)*, pp. 176–182. IEEE, 2005. ISBN 978-0-7695-2465-8.

[**66**] Zhang, S.J. and Liu, Y.: An automatic approach to model checking UML state machines. In: *Proceedings of the 4th International Conference on Secure Software Integration and Reliability Improvement Companion (SSIRI-C 2010)*, pp. 1–6. IEEE, 2010. ISBN 978-0-7695-4087-0.

[**67**] Turner, E., Treharne, H., Schneider, S.A. and Evans, N.: Automatic generation of CSP || B skeletons from xUML models. In: *Proceedings of the 5th International Colloquium on Theoretical Aspects of Computing (ICTAC 2008)*, vol. 5160 of *Lecture Notes in Computer Science*, pp. 364–379. Springer, 2008. ISBN 978-3-540-85761-7.

[**68**] Schneider, S.A. and Treharne, H.: CSP theorems for communicating B machines. *Formal Aspects of Computing*, vol. 17, no. 4, pp. 390–422, 2005. ISSN 0934-5043.

[**69**] Mellor, S.J. and Balcer, M.J.: *Executable UML: A Foundation for Model-Driven Architecture*. Addison-Wesley, 2002. ISBN 978-0-201-74804-8.

[**70**] Yeung, W.L.: Checking consistency between UML class and state models based on CSP and B. *Journal of Universal Computer Science*, vol. 10, no. 11, pp. 1540–1558, 2004. ISSN 0948-6968.

[**71**] Küster-Filipe, J.: Modelling concurrent interactions. In: *Proceedings of the 10th International Conference on Algebraic Methodology and Software Technology (AMAST 2004)*, vol. 3116 of *Lecture Notes in Computer Science*, pp. 304–318. Springer, 2004. ISBN 978-3-540-22381-8.

[**72**] OCL: *Object Constraint Language Specification, version 2.3.1*. Object Management Group, 2012. Available at: `http://www.omg.org/spec/OCL/2.3.1/`, [2014, March].

[**73**] Lam, V.S.W. and Padget, J.: Consistency checking of sequence diagrams and statechart diagrams using the π-calculus. In: *Proceedings of the 5th International Conference on Integrated Formal Methods (IFM 2005)*, vol. 3771 of *Lecture Notes in Computer Science*, pp. 347–365. Springer, 2005. ISBN 978-3-540-30492-0.

[**74**] Cunha, E., Custodio, M., Rocha, H. and Barreto, R.: Formal verification of UML sequence diagrams in the embedded systems context. In: *Proceedings of the 2011 Brazilian Symposium on Computing System Engineering (SBESC 2011)*, pp. 39–45. IEEE, 2011. ISBN 978-1-4673-0427-6.

[**75**] Reisig, W.: *Petri Nets: An Introduction*. Springer, 1985. ISBN 978-0-387-13723-0.

[**76**] Yang, N., Yu, H., Sun, H. and Qian, Z.: Modeling UML sequence diagrams using extended Petri nets. *Telecommunication Systems*, vol. 51, no. 2-3, pp. 147–158, 2012. ISSN 1018-4864.

[**77**] Eshuis, R.: Symbolic model checking of UML activity diagrams. *ACM Transactions on Software Engineering and Methodology*, vol. 15, no. 1, pp. 1–38, 2006. ISSN 1049-331X.

[**78**] Lam, V.S.W.: A formalism for reasoning about UML activity diagrams. *Nordic Journal of Computing*, vol. 14, no. 1, pp. 43–64, 2007. ISSN 1236-6064.

[**79**] del Mar Gallardo, M., Merino, P. and Pimentel, E.: Debugging UML designs with model checking. *Journal of Object Technology*, vol. 1, no. 2, pp. 101–117, 2002. ISSN 1660-1769.

[**80**] Guelfi, N. and Mammar, A.: A formal semantics of timed activity diagrams and its PROMELA translation. In: *Proceedings of the 12th Asia-Pacific Software Engineering Conference (APSEC 2005)*, pp. 283–290. IEEE, 2005. ISBN 978-0-7695-2465-8.

[**81**] Inverardi, P., Muccini, H. and Pelliccione, P.: Automated check of architectural models consistency using SPIN. In: *Proceedings of the 16th International Conference on Automated Software Engineering (ASE 2001)*, pp. 346–356. IEEE, 2001. ISBN 978-0-7695-1426-0.

[**82**] Jing, L., Jinhua, L. and Fangning, Z.: Model checking UML activity diagrams with SPIN. In: *Proceedings of the 2009 International Conference on Computational Intelligence and Software Engineering (CiSE 2009)*, pp. 1–4. IEEE, 2009. ISBN 978-1-4244-4507-3.

[**83**] Mikk, E., Lakhnechi, Y. and Siegel, M.: Hierarchical automata as model for statecharts. In: *Proceedings of the 3rd Asian Computing Science Conference on Advances in Computing Science (ASIAN 1997)*, vol. 1345 of *Lecture Notes in Computer Science*, pp. 181–196. Springer, 1997. ISBN 978-3-540-63875-9.

[**84**] Latella, D., Majzik, I. and Massink, M.: Automatic verification of a behavioural subset of UML statechart diagrams using the SPIN model-checker. *Formal Aspects of Computing*, vol. 11, no. 6, pp. 637–664, 1999. ISSN 0934-5043.

[**85**] Mikk, E., Lakhnech, Y., Siegel, M. and Holzmann, G.: Implementing statecharts in PROMELA/SPIN. In: *Proceedings of the 2nd Workshop on Industrial Strength Formal Specification Techniques (WIFT 1998)*, pp. 90–101. IEEE, 1998. ISBN 978-0-7695-0081-2.

[**86**] Lilius, J. and Paltor, I.P.: vUML: a tool for verifying UML models. In: *Proceedings of the 14th International Conference on Automated Software Engineering (ASE 1999)*, pp. 255–258. IEEE, 1999. ISBN 978-0-7695-0415-5.

[**87**] Lima, V., Talhi, C., Mouheb, D., Debbabi, M., Wang, L. and Pourzandi, M.: Formal verification and validation of UML 2.0 sequence diagrams using source and destination of messages. *Electronic Notes in Theoretical Computer Science*, vol. 254, pp. 143–160, 2009. ISSN 1571-0661.

[**88**] Liu, S., Liu, Y., Sun, J., Zheng, M., Wadhwa, B. and Dong, J.S.: USMMC: a self-contained model checker for UML state machines. In: *Proceedings of the 9th Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE 2013)*, pp. 623–626. ACM, 2013. ISBN 978-1-4503-2237-9.

[**89**] Schäfer, T., Knapp, A. and Merz, S.: Model checking UML state machines and collaborations. *Electronic Notes in Theoretical Computer Science*, vol. 55, no. 3, pp. 357–369, 2001. ISSN 1571-0661.

[**90**] Zhao, X., Long, Q. and Qiu, Z.: Model checking dynamic UML consistency. In: *Proceedings of the 8th International Conference on Formal Engineering Methods (ICFEM 2006)*, vol. 4260 of *Lecture Notes in Computer Science*, pp. 440–459. Springer, 2006. ISBN 978-3-540-47460-9.

[**91**] Huang, X., Sun, Q., Li, J. and Zhang, T.: MDE-based verification of SysML state machine diagram by UPPAAL. In: *Proceedings of the 2012 International Conference on Trustworthy Computing and Services (ISCTCS 2012)*, vol. 320 of *Communications in Computer and Information Science*, pp. 490–497. Springer, 2013. ISBN 978-3-642-35794-7.

[**92**] Diethers, K. and Huhn, M.: Vooduu: verification of object-oriented designs using UPPAAL. In: *Proceedings of the 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, vol. 2988 of *Lecture Notes in Computer Science*, pp. 139–143. Springer, 2004. ISBN 978-3-540-21299-7.

[**93**] Knapp, A., Merz, S. and Rauh, C.: Model checking timed UML state machines and collaborations. In: *Proceedings of the 7th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT 2002)*, vol. 2469 of *Lecture Notes in Computer Science*, pp. 395–414. Springer, 2002. ISBN 978-3-540-44165-6.

[**94**] Jackson, D.: *Software Abstractions: Logic, Language, and Analysis*. MIT Press, 2006. ISBN 978-0-262-10114-1.

[**95**] Garis, A., Paiva, A.C.R., Cunha, A. and Riesco, D.: Specifying UML protocol state machines in Alloy. In: *Proceedings of the 9th International Conference on Integrated Formal Methods (IFM 2012)*, vol. 7321 of *Lecture Notes in Computer Science*, pp. 312–326. Springer, 2012. ISBN 978-3-642-30729-4.

[**96**] van der Straeten, R., Jonckers, V. and Mens, T.: A formal approach to model refactoring and model refinement. *Software and Systems Modelling*, vol. 6, no. 2, pp. 139–162, 2007. ISSN 1619-1374.

[**97**] van der Straeten, R. and D'Hondt, M.: Model refactorings through rule-based inconsistency resolution. In: *Proceedings of the 2006 Symposium on Applied computing (SAC 2006)*, pp. 1210–1217. ACM, 2006. ISBN 978-1-59593-108-5.

[**98**] Baader, F., Calvanese, D., McGuinness, D.L., Nardi, D. and Patel-Schneider, P.F.: *The Description Logic Handbook: Theory, Implementation and Applications*. Cambridge University Press, 2003. ISBN 978-0-521-78176-3.

[**99**] Börger, E. and Stärk, R.: *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer, 2003. ISBN 978-3-540-00702-9.

[**100**] Börger, E., Cavarra, A. and Riccobene, E.: An ASM semantics for UML activity diagrams. In: *Proceedings of the 8th International Conference on Algebraic Methodology and Software Technology (AMAST 2000)*, vol. 1816 of *Lecture Notes in Computer Science*, pp. 293–308. Springer, 2000. ISBN 978-3-540-67530-3.

[**101**] Börger, E., Cavarra, A. and Riccobene, E.: Modeling the dynamics of UML state machines. In: *Proceedings of the 2000 International Workshop on Abstract State Machines: Theory and Applications (ASM 2000)*, vol. 1912 of *Lecture Notes in Computer Science*, pp. 223–241. Springer, 2000. ISBN 978-3-540-67959-2.

[**102**] Kim, S.-K. and Carrington, D.A.: A formal model of the UML metamodel: the UML state machine and its integrity constraints. In: *Proceedings of the 2nd International Conference of B and Z Users on Formal Specification and Development in Z and B (ZB 2002)*, vol. 2272 of *Lecture Notes in Computer Science*, pp. 497–516. Springer, 2002. ISBN 978-3-540-43166-4.

[**103**] Duke, R., King, P., Rose, G. and Smith, G.: *The Object-Z Specification Language*. Kluwer Academic Publishers, 2000. ISBN 978-0-7923-8684-1.

[**104**] Carneiro, E., Maciel, P., Callou, G., Tavares, E. and Nogueira, B.: Mapping SysML state machine diagram to time Petri net for analysis and verification of embedded real-time systems with energy constraints. In: *Proceedings of the 2008 International Conference on Advances in Electronics and Micro-electronics (ENICS 2008)*, pp. 1–6. IEEE, 2008. ISBN 978-0-7695-3370-4.

[**105**] Andrade, E., Maciel, P., Callou, G. and Nogueira, B.: A methodology for mapping SysML activity diagram to time Petri net for requirement validation of embedded real-time systems with energy constraints. In: *Proceedings of the 3rd International Conference on Digital Society (ICDS 2009)*, pp. 266–271. IEEE, 2009. ISBN 978-0-7695-3526-5.

[**106**] Cimatti, A. and Tonetta, S.: Contracts-refinement proof system for component-based embedded systems. *Science of Computer Programming*, vol. 97, pp. 333–348, 2015. ISSN 0167-6423.

[**107**] Sljivo, I., Gallina, B., Carlson, J., Hansson, H. and Puri, S.: A method to generate reusable safety case fragments from compositional safety analysis. In: *Proceedings of the 14th International Conference on Software Reuse (ICSR 2015)*, vol. 8919 of *Lecture Notes in Computer Science*, pp. 253–268. Springer, 2015. ISBN 978-3-319-14129-9.

[**108**] Roscoe, A.W. and Wu, Z.: Verifying Statemate statecharts using CSP and FDR. In: *Proceedings of the 8th International Conference on Formal Engineering Methods (ICFEM 2006)*, vol. 4260 of *Lecture Notes in Computer Science*, pp. 324–341. Springer, 2006. ISBN 978-3-540-47460-9.

[**109**] Alur, R., Holzmann, G.J. and Peled, D.: An analyzer for message sequence charts. In: *Proceedings of the 2nd International Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 1996)*, vol. 1055 of *Lecture Notes in Computer Science*, pp. 35–48. Springer, 1996. ISBN 978-3-540-49874-2.

[**110**] Jacobs, J. and Simpson, A.C.: Towards a process algebra framework for supporting behavioural consistency and requirements traceability in SysML. In: *Proceedings of the 15th International Conference on Formal Engineering Methods (ICFEM 2013)*, vol. 8144 of *Lecture Notes in Computer Science*, pp. 266–281. Springer, 2013. ISBN 978-3-642-41201-1.

[**111**] Simons, A.J.H.: On the compositional properties of UML statechart diagrams. In: *Proceedings of the 2000 International Conference on Rigorous Object-Oriented Methods (ROOM 2000)*. British Computer Society, 2000.

[**112**] Armstrong, P., Lowe, G., Ouaknine, J. and Roscoe, A.W.: Model checking Timed CSP. In: *Proceedings of the 2011 Higher-Order Workshop on Automated Runtime Verification and Debugging (HOWARD-60 2011)*. 2012.

[**113**] Jacobs, J. and Simpson, A.C.: A formal model of SysML blocks using CSP for assured systems engineering. In: *Proceedings of the 3rd International Workshop on Formal Techniques for Safety-Critical Systems (FTSCS 2014)*, Communications in Computer and Information Science. Springer, 2015. (To appear).

[**114**] Lowe, G.: *Probabilities and Priorities in Timed CSP*. Doctor of Philosophy thesis, Department of Computer Science, University of Oxford, 1993.

[**115**] Goldsmith, M.H.: CSP: the best concurrent-system description language in the world - probably! In: *Proceedings of the 2004 International Conference on Communicating Process Architectures (CPA 2004)*, vol. 62 of *Concurrent Systems Engineering Series*, pp. 227–232. IOS Press, 2004. ISBN 978-1-58603-458-0.

[**116**] Kwiatkowska, M., Norman, G. and Parker, D.: PRISM: probabilistic symbolic model checker. In: *Proceedings of the 12th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation (TOOLS 2002)*, vol. 2324 of *Lecture Notes in Computer Science*, pp. 200–204. Springer, 2002. ISBN 978-3-540-43539-6.

[**117**] Song, S., Sun, J., Liu, Y. and Dong, J.S.: A model checker for hierarchical probabilistic real-time systems. In: *Proceedings of the 24th International Conference on Computer Aided Verification (CAV 2012)*, vol. 7358 of *Lecture Notes in Computer Science*, pp. 705–711. Springer, 2012. ISBN 978-3-642-31423-0.

[**118**] Jacobs, J. and Simpson, A.C.: On a process algebraic representation of sequence diagrams. In: *Proceedings of the 1st International Workshop on Safety and Formal Methods (SaFoMe 2014)*, vol. 8938 of *Lecture Notes in Computer Science*, pp. 71–85. Springer, 2015. ISBN 978-3-319-15201-1.

[**119**] QVT: *Query/View/Transformation Specification, version 1.1*. Object Management Group, 2011. Available at: `http://www.omg.org/spec/QVT/1.1`, [2014, March].

[**120**] Jacobs, J. and Simpson, A.C.: A process algebraic approach to decomposition of communicating SysML blocks. In: *Proceedings of the 2nd International Conference on System Engineering and Modeling (ICSEM 2013)*, pp. 153–157. IACSIT, 2013.

[**121**] Jacobs, J. and Simpson, A.C.: On the formal interpretation of SysML blocks using a safety critical case study. In: *Proceedings of the 8th Brazilian Symposium on Software Components, Architectures, and Reuse (SBCARS 2014)*, pp. 95–104. IEEE, 2015. ISBN 978-1-4799-7860-1.

[**122**] Leveson, N.G.: *Engineering a Safer World: Systems Thinking Applied to Safety*. MIT Press, 2012. ISBN 978-0-262-01662-9.

[**123**] Woodcock, J.C.P., Cavalcanti, A.L.C., Fitzgerald, J., Larsen, P., Miyazawa, A. and Perry, S.: Features of CML: a formal modelling language for systems of systems. In: *Proceedings of the 7th International Conference on System of Systems Engineering (SoSE 2012)*, pp. 1–6. IEEE, 2012. ISBN 978-1-4673-2974-3.

[**124**] Graves, H. and Bijan, Y.: Using formal methods with SysML in aerospace design and engineering. *Annals of Mathematics and Artificial Intelligence*, vol. 63, no. 1, pp. 53–102, 2011. ISSN 1012-2443.

[**125**] Kolovos, D., Rose, L., Paige, R. and García-Domínguez, A.: *The Epsilon Book*. Online book, Department of Computer Science, University of York, 2012.

[**126**] Woodcock, J.C.P. and Cavalcanti, A.L.C.: A concurrent language for refinement. In: *Proceedings of the 5th Irish Workshop on Formal Methods (IW-FM 2001)*, pp. 93–115. British Computer Society, 2001.

[**127**] Debbabi, M., Hassaïne, F., Jarraya, Y., Soeanu, A. and Alawneh, L.: Probabilistic model checking of SysML activity diagrams. In: *Verification and Validation in Systems Engineering*, pp. 153–166. Springer, 2010. ISBN 978-3-642-15227-6.

APPENDICES

SysML Model

A.1 Low Level Components

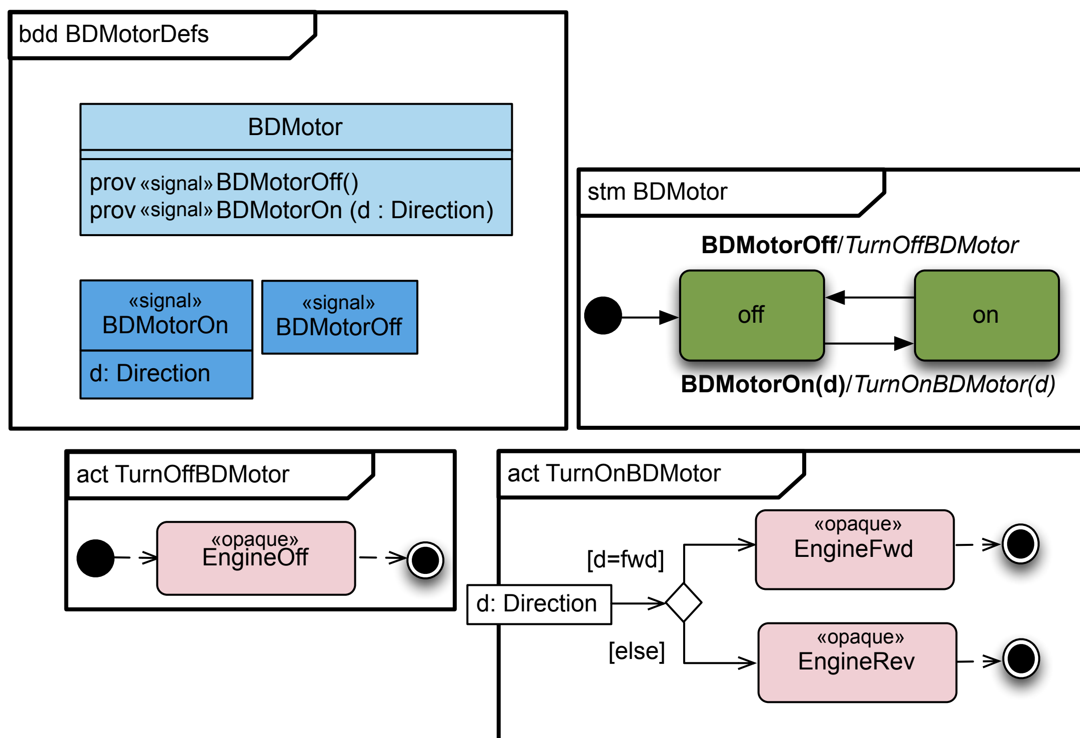


Figure A.1: Bidirectional motor diagrams.

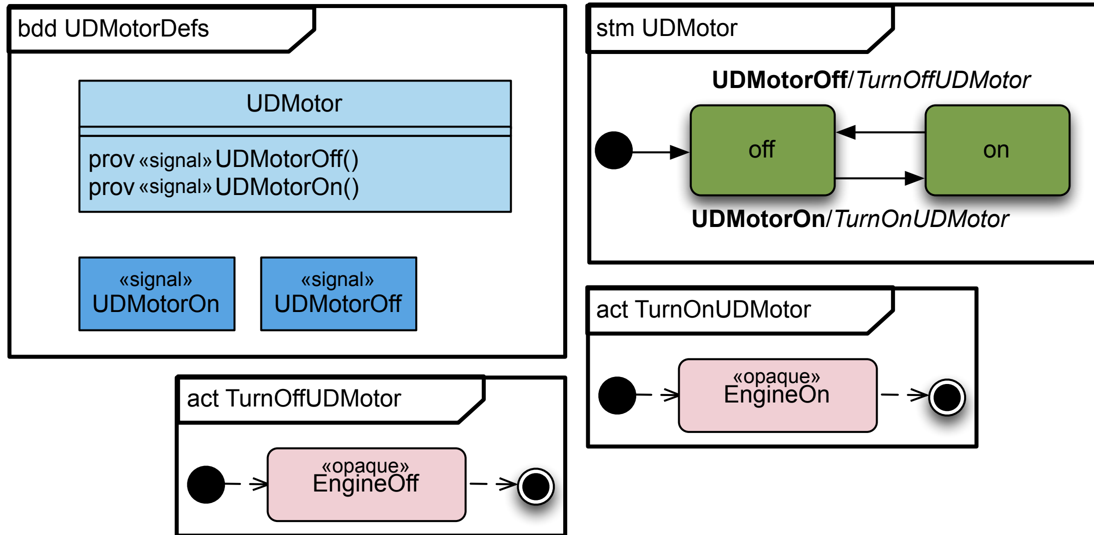


Figure A.2: Unidirectional motor diagrams.

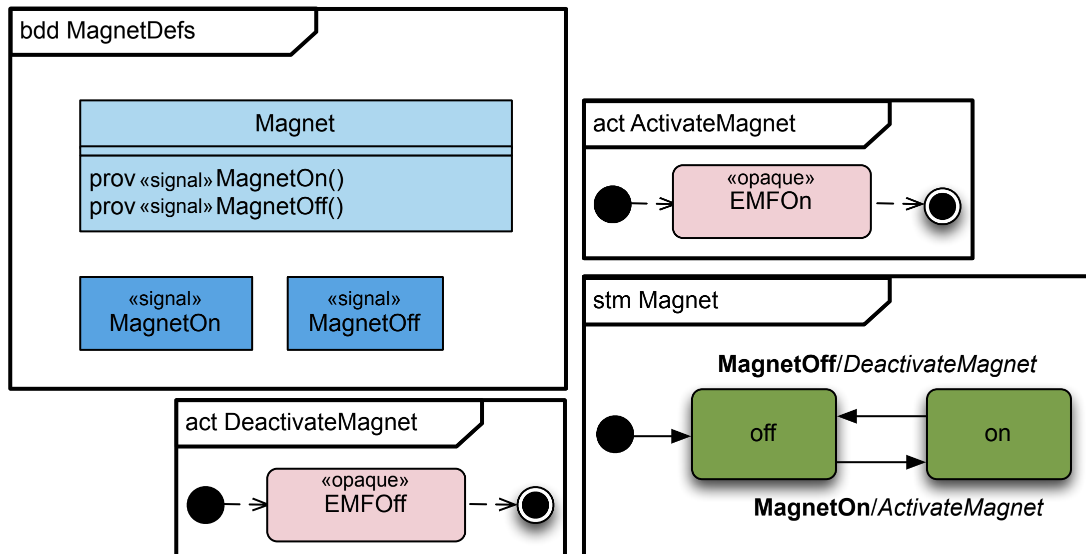


Figure A.3: Magnet diagrams.

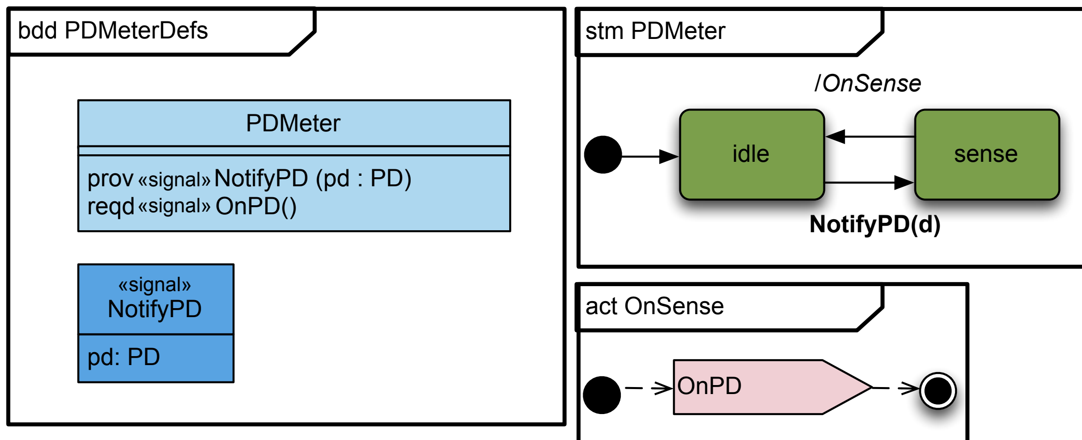


Figure A.4: Potentiometer diagrams.

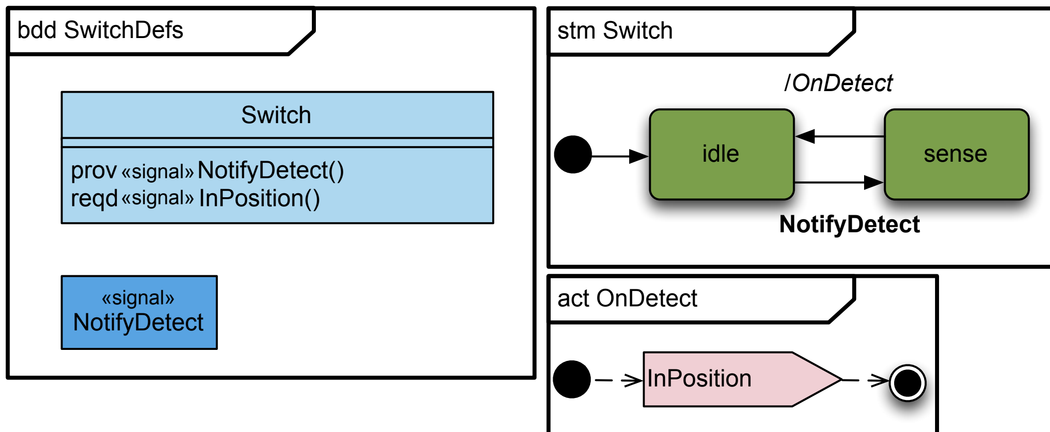


Figure A.5: Switch diagrams.

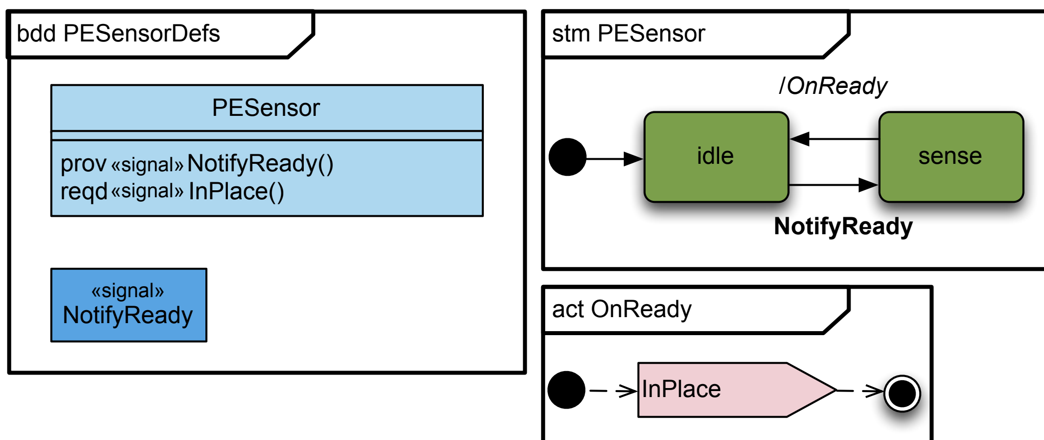


Figure A.6: Photoelectric sensor diagrams.

A.2 Intermediate Components

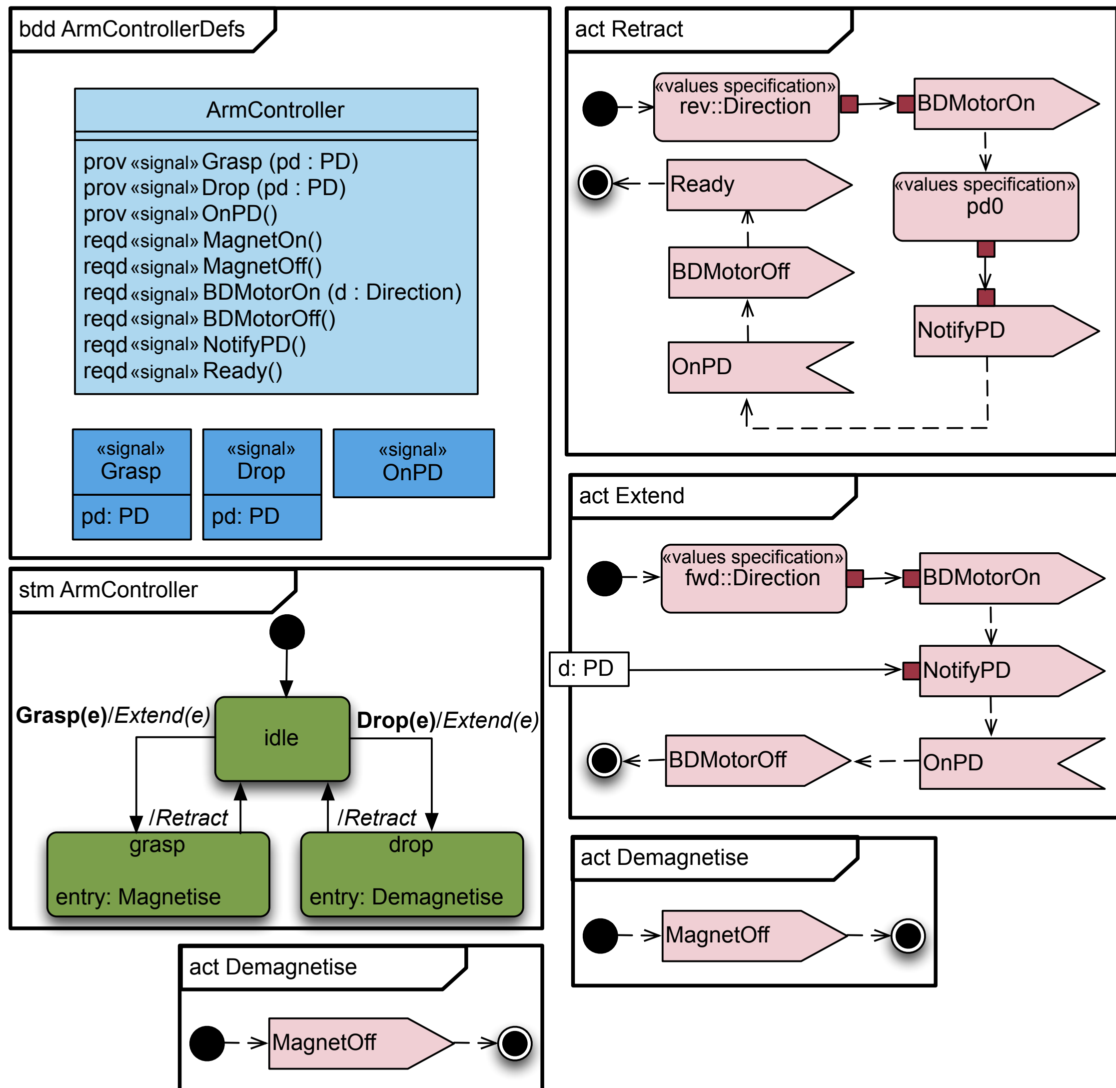


Figure A.7: Arm controller diagrams.

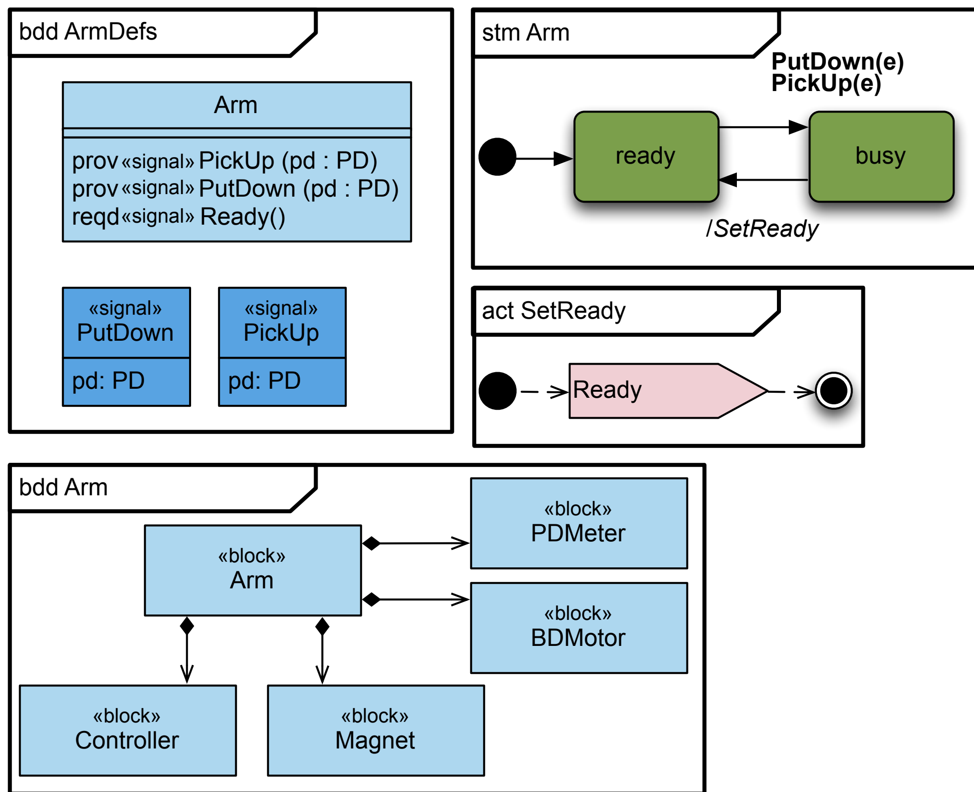


Figure A.8: Arm diagrams.

A.3 High Level Components

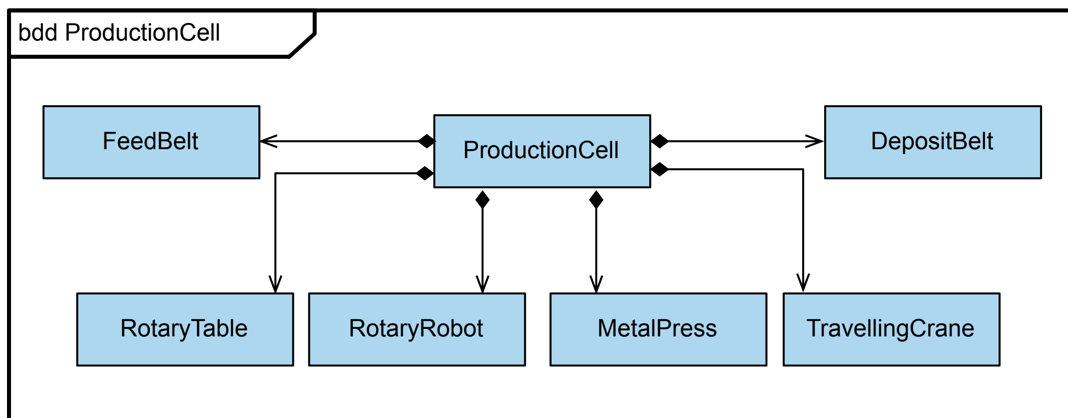


Figure A.9: Block definition diagram of the complete system.

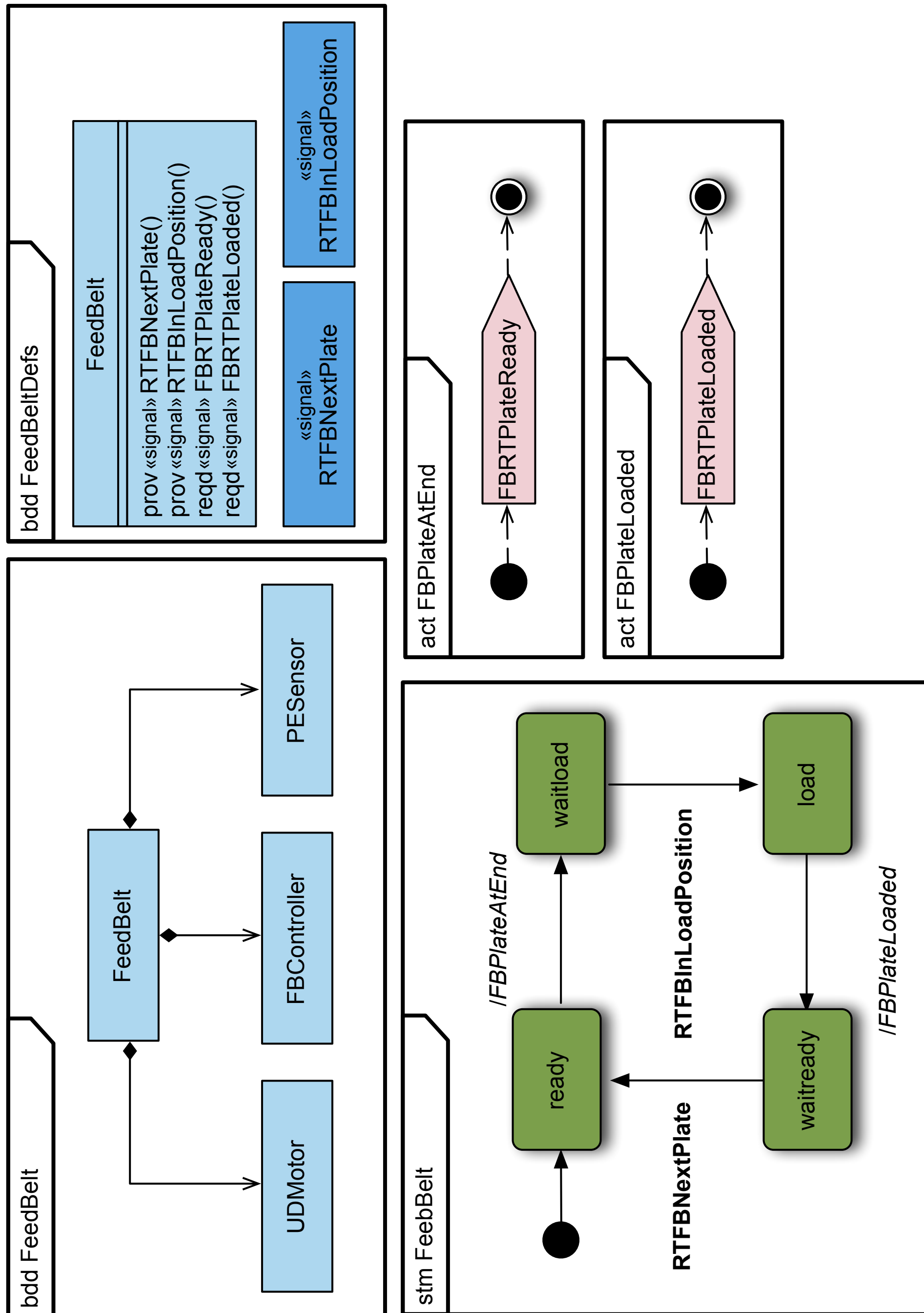


Figure A.10: Feed belt diagrams.

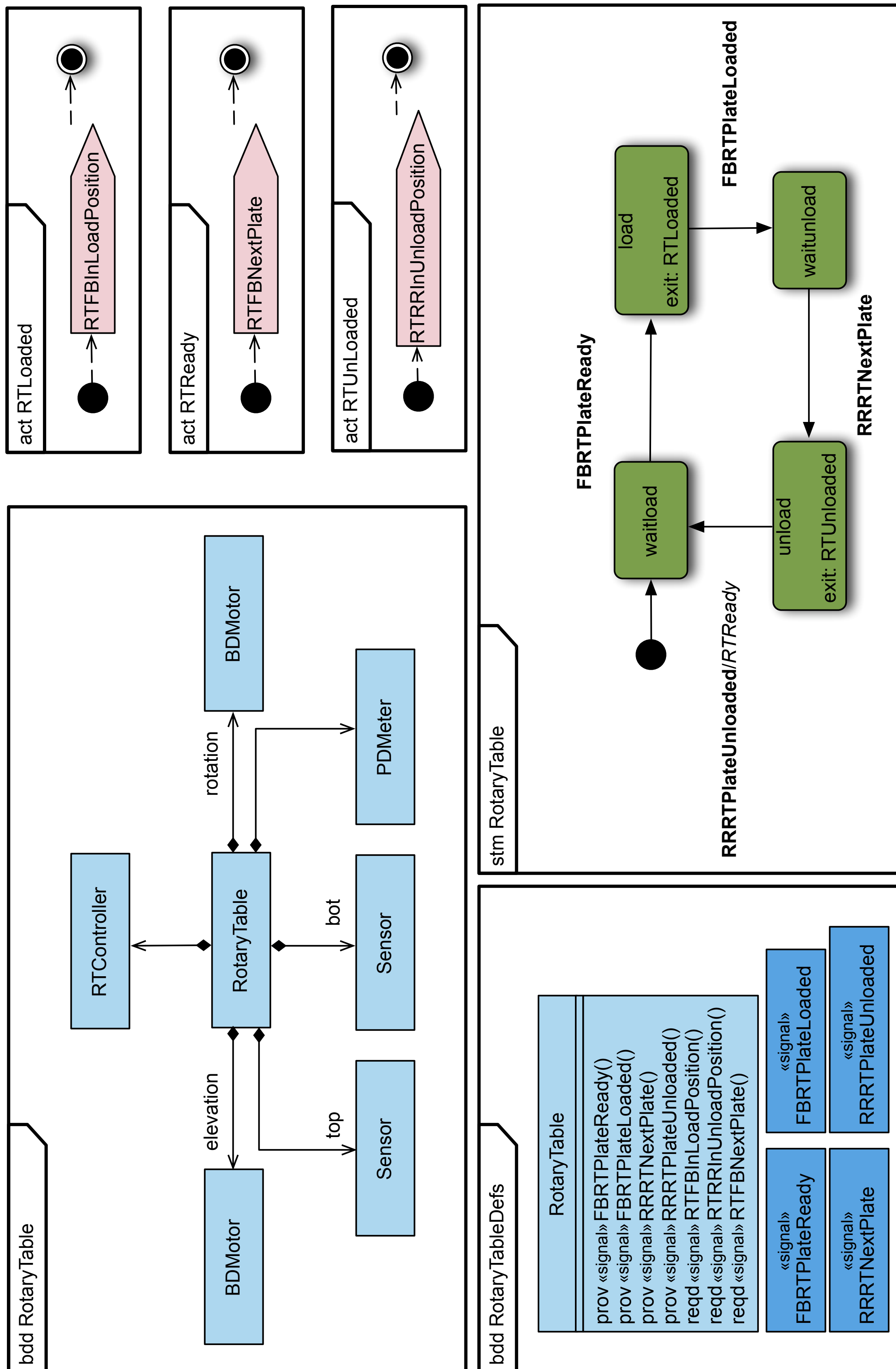


Figure A.11: Rotary table diagrams.

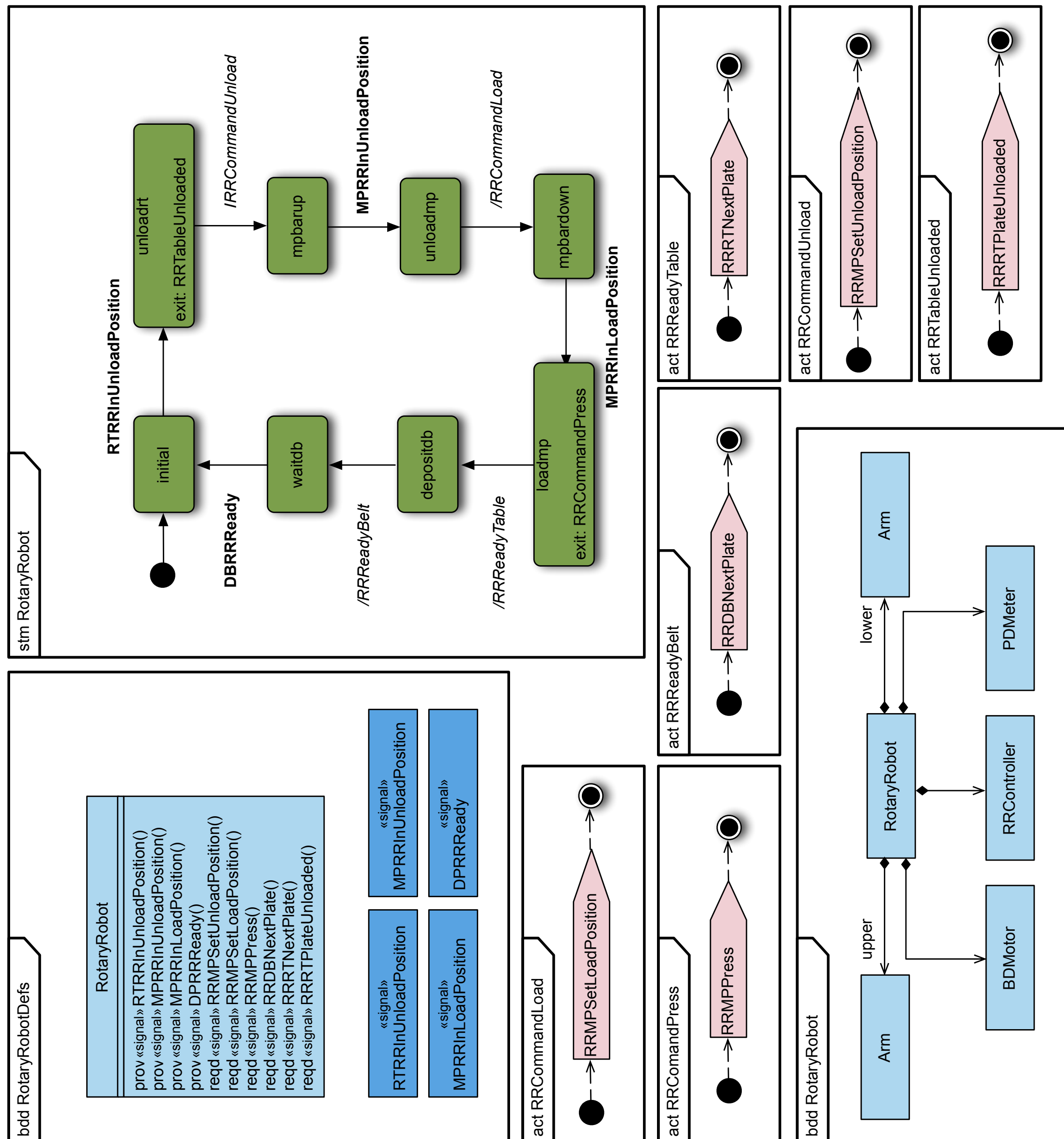


Figure A.12: Rotary robot diagrams.

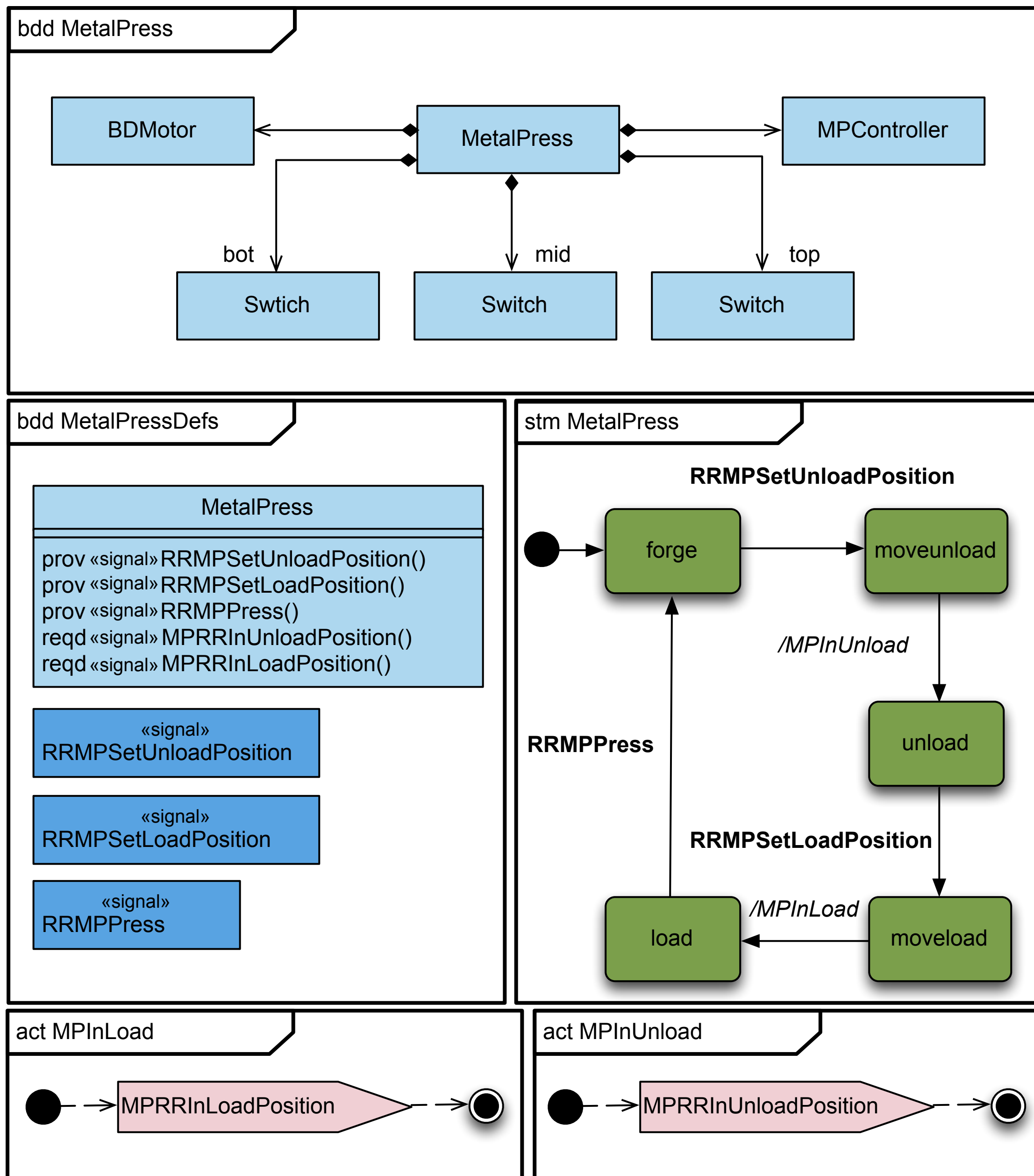


Figure A.13: Metal press diagrams.

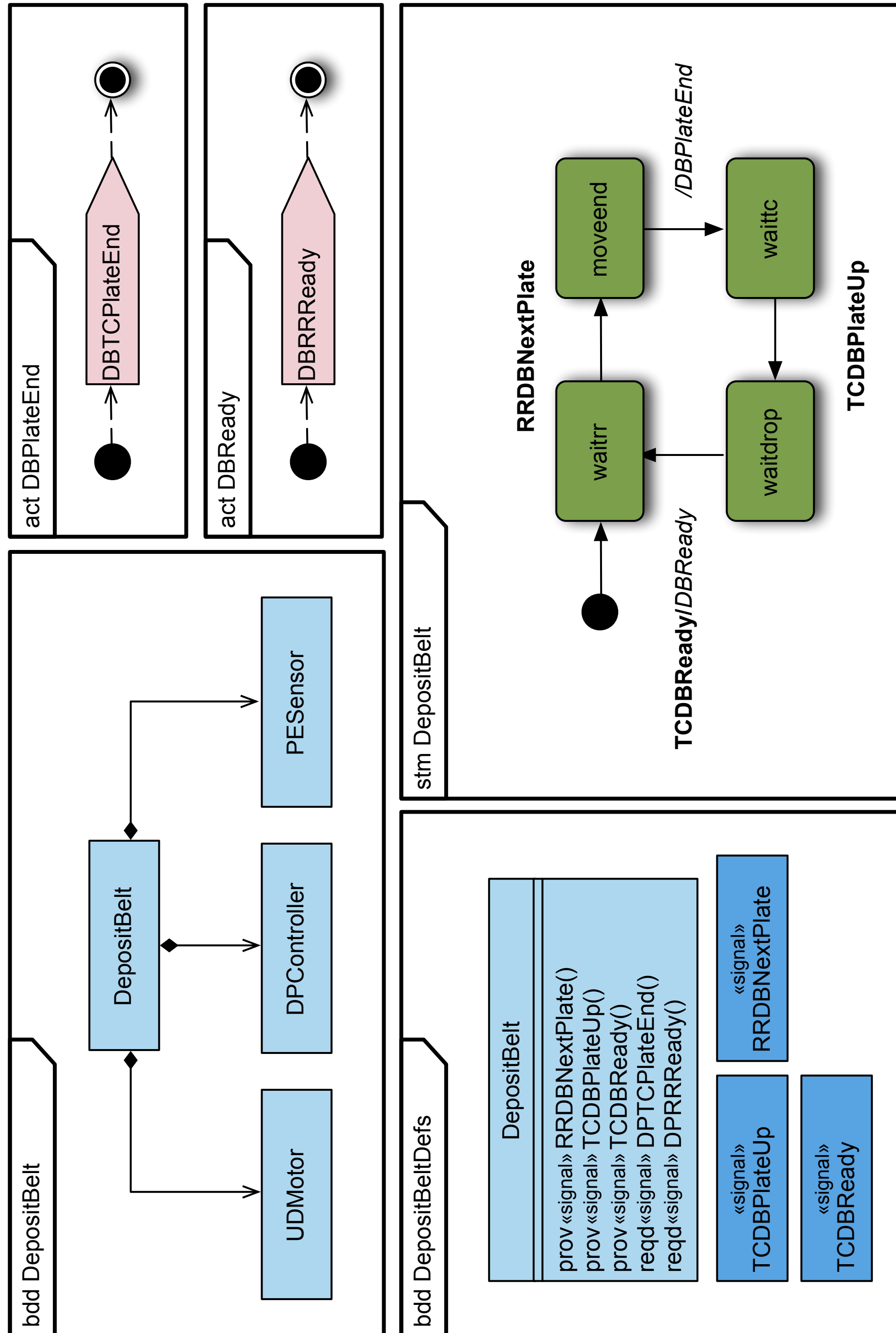


Figure A.14: Deposit belt diagrams.

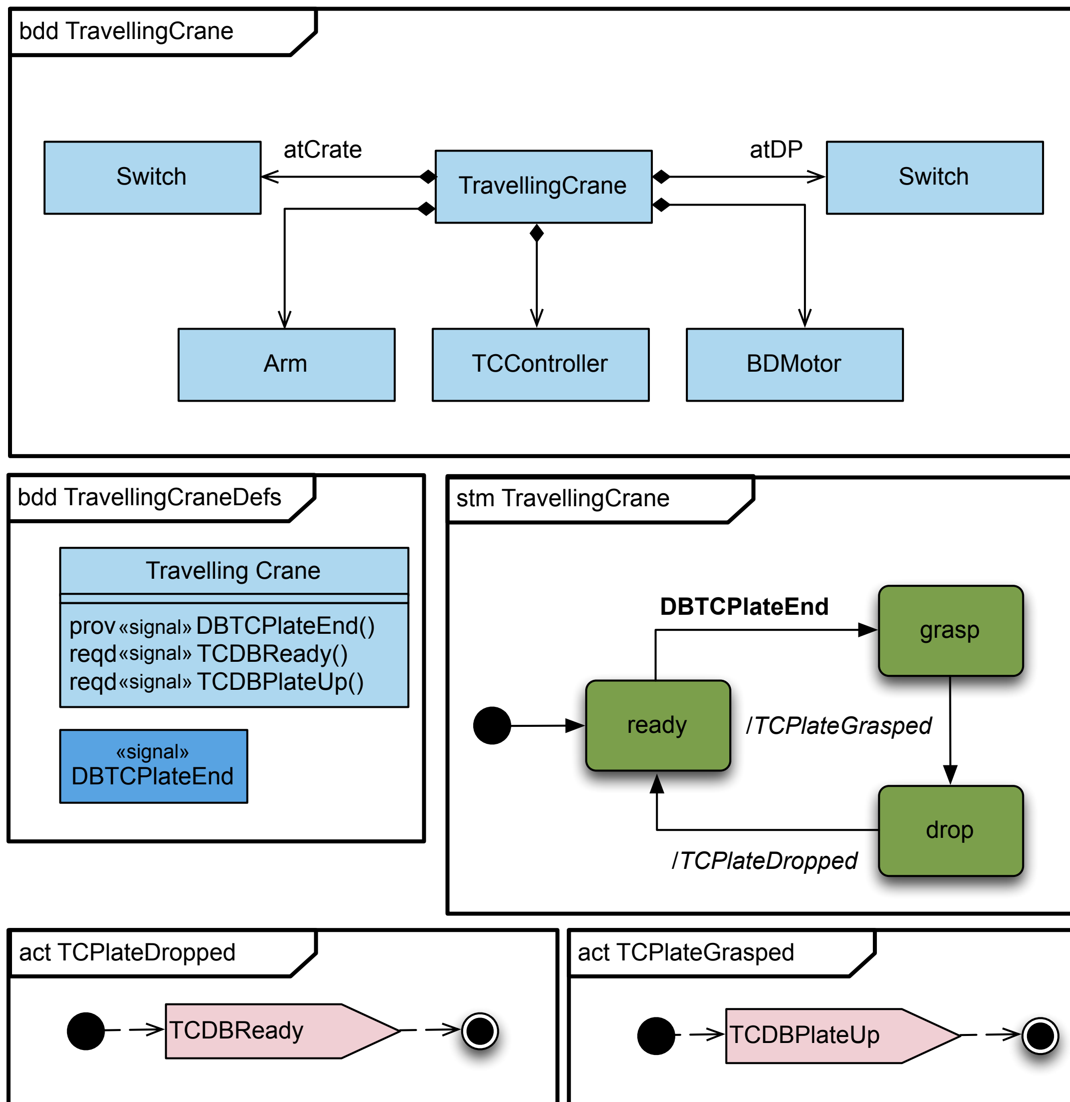


Figure A.15: Travelling Crane diagrams.

CSP Model

B.1 Low Level Components

B.1.1 Bidirectional Motor

$$
\begin{array}{l}
BDMotor(queue, local) = \\
\quad \text{let} \\
\quad\quad I_0 = OFF \\
\quad\quad OFF = \\
\quad\quad\quad local.proc.BDMotorOn?d \rightarrow \\
\quad\quad\quad\quad TurnOnBDMotor(d) \mathbin{;} ON \\
\quad\quad\quad \Box \\
\quad\quad\quad local.disc?e : \{\!|\ BDMotorOff\ |\!\} \rightarrow OFF \\
\quad\quad ON = \\
\quad\quad\quad local.proc.BDMotorOff \rightarrow \\
\quad\quad\quad\quad TurnOffBDMotor \mathbin{;} OFF \\
\quad\quad\quad \Box \\
\quad\quad\quad local.disc?e : \{\!|\ BDMotorOn\ |\!\} \rightarrow ON \\
\quad\quad EQ = queue?e \rightarrow local?p!e \rightarrow EQ \\
\quad \text{within} \\
\quad\quad I_0 \ [|\ \{\!|\ local\ |\!\}\ |]\ EQ
\end{array}
$$

$$BDMOTOR = BDMotor(bdmotor, bdmotorlocal)$$

$$
\begin{array}{l}
\alpha BDMOTOR = \\
\quad Union(\{\{\!|\ bdmotor, bdmotorlocal\ |\!\}, \alpha TurnOnBDMotor, \alpha TurnOffBDMotor\})
\end{array}
$$

$$
\begin{array}{l}
TurnOnBDMotor(d) = \\
\quad \text{let} \\
\quad\quad DEC_0 = \\
\quad\quad\quad \text{if } d == fwd \text{ then}
\end{array}
$$

OA_0
else
OA_1
$OA_0 = opaque.EngineFwd \rightarrow F_0$
$OA_1 = opaque.EngineRev \rightarrow F_0$
$F_0 = Skip$
within
DEC_0

$\alpha TurnOnBDMotor = \{| \; opaque.EngineFwd, opaque.EngineRev \; |\}$

$TurnOffBDMotor =$
let
$I_0 = OA_0$
$OA_0 = opaque.EngineOff \rightarrow F_0$
$F_0 = Skip$
within
I_0

$\alpha TurnOffBDMotor = \{| \; opaque.EngineOff \; |\}$

B.1.2 Unidirectional Motor

$UDMotor(queue, local) =$
let
$I_0 = OFF$
$OFF =$
$local.proc.UDMotorOn \rightarrow$
$TurnOnUDMotor \; ; \; ON$
$\Box$
$local.disc?e : \{| \; UDMotorOff \; |\} \rightarrow OFF$
$ON =$
$local.proc.UDMotorOff \rightarrow$
$TurnOffUDMotor \; ; \; OFF$
$\Box$
$local.disc?e : \{| \; UDMotorOn \; |\} \rightarrow ON$
$EQ = queue?e \rightarrow local?p!e \rightarrow EQ$
within
$I_0 \; [| \; \{| \; local \; |\} \; |] \; EQ$

$UDMOTOR = UDMotor(udmotor, udmotorlocal)$

$\alpha UDMOTOR =$
$Union(\{\{| \; udmotor, udmotorlocal \; |\}, \alpha TurnOnUDMotor, \alpha TurnOffUDMotor\})$

$TurnOnUDMotor =$

let
$I_0 = OA_0$
$OA_0 = opaque.EngineOn \rightarrow F_0$
$F_0 = Skip$
within
I_0

$\alpha TurnOnUDMotor = \{| \ opaque.EngineOn \ |\}$

$TurnOffUDMotor =$
let
$I_0 = OA_0$
$OA_0 = opaque.EngineOff \rightarrow F_0$
$F_0 = Skip$
within
I_0

$\alpha TurnOffUDMotor = \{| \ opaque.EngineOff \ |\}$

B.1.3 Magnet

$Magnet(queue, local) =$
let
$I_0 = OFF$
$OFF =$
$local.proc.MagnetOn \rightarrow$
$ActivateMagnet \mathbin{;} ON$
$\Box$
$local.disc?e : \{| \ MagnetOff \ |\} \rightarrow OFF$
$ON =$
$local.proc.MagnetOff \rightarrow$
$DeactivateMagnet \mathbin{;} OFF$
$\Box$
$local.disc?e : \{| \ MagnetOn \ |\} \rightarrow ON$
$EQ = queue?e \rightarrow local?p!e \rightarrow EQ$
within
$I_0 \ [| \ \{| \ local \ |\} \ |] \ EQ$

$MAGNET = Magnet(magnet, magnetlocal)$

$\alpha MAGNET =$
$Union(\{\{| \ magnet, magnetlocal \ |\}, \alpha ActivateMagnet, \alpha DeactivateMagnet\})$

$ActivateMagnet =$
let
$I_0 = OA_0$

$OA_0 = opaque.EMFOn \rightarrow F_0$
$F_0 = Skip$
within
I_0

$\alpha ActivateMagnet = \{| opaque.EMFOn |\}$

$DeactivateMagnet =$
let
$I_0 = OA_0$
$OA_0 = opaque.EMFOff \rightarrow F_0$
$F_0 = Skip$
within
I_0

$\alpha DeactivateMagnet = \{| opaque.EMFOff |\}$

B.1.4 Potentiometer

$PDMeter(queue, local) =$
let
$I_0 = IDLE$
$IDLE =$
$local.proc.NotifyPD?pd \rightarrow SENSE$
$SENSE =$
$OnSense \fatsemi IDLE$
$\square$
$local.disc?e : \{| NotifyPD |\} \rightarrow SENSE$
$EQ = queue?e \rightarrow local?p!e \rightarrow EQ$
within
$I_0 [| \{| local |\} |] EQ$

$PDMETER = PDMeter(pdmeter, pdmeterlocal)$

$\alpha PDMETER = Union(\{\{| pdmeter, pdmeterlocal |\}, \alpha OnSense\})$

$OnSense =$
let
$I_0 = SS_0$
$SS_0 = controller.OnPD \rightarrow F_0$
$F_0 = Skip$
within
I_0

B.1.5 Switch

$Switch(queue, local) =$
let
$I_0 = IDLE$
$IDLE =$
$local.proc.NotifyDetect \rightarrow SENSE$
$SENSE =$
$OnDetect \fatsemi IDLE$
$\Box$
$local.disc?e : \{| NotifyDetect |\} \rightarrow SENSE$
$EQ = queue?e \rightarrow local?p!e \rightarrow EQ$
within
$I_0 [| \{| local |\} |] EQ$

$SWITCH = Switch(switch, switchlocal)$

$\alpha SWITCH = Union(\{\{| switch, switchlocal |\}, \alpha OnDetect\})$
$OnDetect =$
let
$I_0 = SS_0$
$SS_0 = controller.InPosition \rightarrow F_0$
$F_0 = Skip$
within
I_0

B.1.6 Photoelectric Sensor

$PESensor(queue, local) =$
let
$I_0 = IDLE$
$IDLE =$
$local.proc.NotifyReady \rightarrow SENSE$
$SENSE =$
$OnReady \fatsemi IDLE$
$\Box$
$local.disc?e : \{| NotifyReady |\} \rightarrow SENSE$
$EQ = queue?e \rightarrow local?p!e \rightarrow EQ$
within
$I_0 [| \{| local |\} |] EQ$

$PESENSOR = PESensor(pesensor, pesensorlocal)$

$\alpha PESENSOR = Union(\{\{| pesensor, pesensorlocal |\}, \alpha OnReady\})$
$OnReady =$
let
$I_0 = SS_0$

$SS_0 = controller.InPlace \rightarrow F_0$
$F_0 = Skip$
within
I_0

B.2 Intermediate Components

B.2.1 Arm Controller

$Controller(queue, local) =$
let
$I_0 = IDLE$
$IDLE =$
$local.proc.Grasp?e \rightarrow$
$Extend(local, e) \fatsemi Magnetise \fatsemi GRASP$
$\Box$
$local.proc.Drop?e \rightarrow$
$Extend(local, e) \fatsemi Demagnetise \fatsemi DROP$
$\Box$
$local.disc?e : \{| OnPD |\} \rightarrow IDLE$
$GRASP =$
$Retract(local) \fatsemi IDLE$
$\Box$
$local.disc?e : \{| Grasp, Drop, OnPD |\} \rightarrow GRASP$
$DROP =$
$Retract(local) \fatsemi IDLE$
$\Box$
$local.disc?e : \{| Grasp, Drop, OnPD |\} \rightarrow DROP$
$EQ = queue?e \rightarrow local?p!e \rightarrow EQ$
within
$I_0 [| \{| local |\} |] EQ$

$CONTROLLER = Controller(controller, controllerlocal)$

$\alpha CONTROLLER = Union(\{\{| controller, controllerlocal |\}, \alpha Magnetise, \alpha Demagnetise, \alpha Extend, \alpha Retract\})$

$Extend(local, pd) =$
let
$I_0 = VS_0$
$VS_0 = SS_0(fwd)$
$SS_0(o) = bdmotor.BDMotorOn.o \rightarrow SS_1$
$SS_1 = pdmeter.NotifyPD.pd \rightarrow RS_0$
$RS_0 =$
$local.proc.OnPD \rightarrow SS_2$
$\Box$

$local.disc?ev : \{| Grasp, Drop |\} \rightarrow RS_0$
$SS_2 = bdmotor.BDMotorOff \rightarrow F_0$
$F_0 = Skip$
within
I_0

$\alpha Extend =$
$\{| bdmotor.BDMotorOn.fwd, bdmotor.BDMotorOff, pdmeter.NotifyPD |\}$

$Retract(local) =$
let
$I_0 = VS_0$
$VS_0 = SS_0(rev)$
$SS_0(o) = bdmotor.BDMotorOn.o \rightarrow VS_1$
$VS_1 = SS_1(pd_0)$
$SS_1(o) = pdmeter.NotifyPD.0 \rightarrow RS_0$
$RS_0 =$
$local.proc.OnPD \rightarrow SS_2$
$\Box$
$local.disc?ev : \{| Grasp, Drop |\} \rightarrow RS_0$
$SS_2 = bdmotor.BDMotorOff \rightarrow SS_3$
$SS_3 = client.Ready \rightarrow F_0$
$F_0 = Skip$
within
I_0

$\alpha Retract =$
$\{| bdmotor.BDMotorOn.rev, bdmotor.BDMotorOff,$
$pdmeter.NotifyPD.pd_0, client.Ready |\}$

$Magnetise =$
let
$I_0 = SS_0$
$SS_0 = magnet.magnetOn \rightarrow F_0$
$F_0 = Skip$
within
I_0

$\alpha Magnetise = \{| magnet.MagnetOn |\}$

$Demagnetise =$
let
$I_0 = SS_0$
$SS_0 = magnet.magnetOff \rightarrow F_0$
$F_0 = Skip$
within
I_0

$\alpha Demagnetise = \{| magnet.MagnetOff |\}$

B.2.2 Arm

$Arm(queue, local) =$
  let
    $I_0 = READY$
    $READY =$
      $local.proc.PickUp?e \rightarrow BUSY$
      $\Box$
      $local.proc.PutDown?e \rightarrow BUSY$
    $BUSY =$
      $SetReady \fatsemi READY$
      $\Box$
      $local.disc?e : \{| PickUp, PutDown |\} \rightarrow BUSY$
    $EQ = queue?e \rightarrow local?p!e \rightarrow EQ$
  within
    $I_0 [| \{| local |\} |] EQ$

$ARM = Arm(arm, armlocal)$

$\alpha ARM =$
  $Union(\{\{| arm, armlocal |\}, \alpha SetReady\})$

$SetReady =$
  let
    $I_0 = SS_0$
    $SS_0 = client.Ready \rightarrow F_0$
    $F_0 = Skip$
  within
    I_0

$\alpha SetReady = \{| client.Ready |\}$

B.3 High Level Components

B.3.1 Feed Belt

$FeedBelt(queue, local) =$
  let
    $I_0 = READY$
    $READY =$
      $FBPlateAtEnd \fatsemi WAITLOAD$
      $\Box$
      $local.disc?e : \{| RTFBNextPlate, RTFBInLoadPosition |\} \rightarrow READY$
    $WAITLOAD =$

$local.proc.RTFBInLoadPosition \rightarrow LOAD$
$\Box$
$local.disc?e : \{| RTFBNextPlate |\} \rightarrow WAITLOAD$
$LOAD =$
$FBPlateLoaded \fatsemi WAITREADY$
$\Box$
$local.disc?e : \{| RTFBNextPlate, RTFBInLoadPosition |\} \rightarrow LOAD$
$WAITREADY =$
$local.proc.RTFBNextPlate \rightarrow READY$
$\Box$
$local.disc?e : \{| RTFBInLoadPosition |\} \rightarrow WAITREADY$
$EQ = queue?e \rightarrow local?p!e \rightarrow EQ$
within
$I_0 [| \{| local |\} |] EQ$

$FEEDBELT = FeedBelt(feedbelt, feedbeltlocal)$

$\alpha FEEDBELT =$
$Union(\{\{| feedbelt, feedbeltlocal |\}, \alpha FBPlateAtEnd, \alpha FBPlateLoaded\})$
$FBPlateAtEnd =$
let
$I_0 = SS_0$
$SS_0 = rotarytable.FBRTPlateReady \rightarrow F_0$
$F_0 = Skip$
within
I_0

$\alpha FBPlateAtEnd = \{| rotarytable.FBRTPlateReady |\}$
$FBPlateLoaded =$
let
$I_0 = SS_0$
$SS_0 = rotarytable.FBRTPlateLoaded \rightarrow F_0$
$F_0 = Skip$
within
I_0

$\alpha FBPlateAtEnd = \{| rotarytable.FBRTPlateLoaded |\}$

B.3.2 Rotary Table

$RotaryTable(queue, local) =$
let
$I_0 = WAITLOAD$
$WAITLOAD =$
$local.proc.FBRTPlateReady \rightarrow LOAD$
$\Box$

$$
\begin{array}{l}
\quad\quad local.disc?e : \{|\ FBRTPlateLoaded, RRRTNextPlate, \\
\quad\quad\quad\quad\quad\quad RRRTPlateUnloaded\ |\} \rightarrow WAITLOAD \\
\quad LOAD = \\
\quad\quad local.proc.FBRTPlateLoaded \rightarrow RTLoaded \fatsemi WAITUNLOAD \\
\quad\quad \Box \\
\quad\quad local.disc?e : \{|\ FBRTPlateReady, RRRTNextPlate, \\
\quad\quad\quad\quad\quad\quad RRRTPlateUnloaded\ |\} \rightarrow LOAD \\
\quad WAITUNLOAD = \\
\quad\quad local.proc.RRRTNextPlate \rightarrow UNLOAD \\
\quad\quad \Box \\
\quad\quad local.disc?e : \{|\ FBRTPlateLoaded, FBRTPlateReady, \\
\quad\quad\quad\quad\quad\quad RRRTPlateUnloaded\ |\} \rightarrow WAITUNLOAD \\
\quad UNLOAD = \\
\quad\quad local.proc.RRRTPlateUnloaded \rightarrow RTUnloaded \fatsemi RTReady \fatsemi WAITLOAD \\
\quad\quad \Box \\
\quad\quad local.disc?e : \{|\ FBRTPlateLoaded, FBRTPlateReady, RRRTNextPlate\ |\} \\
\quad\quad\quad \rightarrow UNLOAD \\
\quad EQ = queue?e \rightarrow local?p!e \rightarrow EQ \\
\text{within} \\
\quad I_0\ [|\ \{|\ local\ |\}\ |]\ EQ
\end{array}
$$

$$ROTARYTABLE = RotaryTable(rotarytable, rotarytablelocal)$$

$$
\begin{array}{l}
\alpha ROTARYTABLE = Union(\{\{|\ rotarytable, rotarytablelocal\ |\}, \\
\quad\quad\quad\quad\quad\quad \alpha RTLoaded, \alpha RTUnloaded, \alpha RTReady\})
\end{array}
$$

$$
\begin{array}{l}
RTLoaded = \\
\quad \text{let} \\
\quad\quad I_0 = SS_0 \\
\quad\quad SS_0 = feedbelt.RTFBInLoadPosition \rightarrow F_0 \\
\quad\quad F_0 = Skip \\
\quad \text{within} \\
\quad\quad I_0
\end{array}
$$

$$\alpha RTLoaded = \{|\ feedbelt.RTFBInLoadPosition\ |\}$$

$$
\begin{array}{l}
RTUnloaded = \\
\quad \text{let} \\
\quad\quad I_0 = SS_0 \\
\quad\quad SS_0 = rotaryrobot.RTRRInUnloadPosition \rightarrow F_0 \\
\quad\quad F_0 = Skip \\
\quad \text{within} \\
\quad\quad I_0
\end{array}
$$

$$\alpha RTUnloaded = \{|\ rotaryrobot.RTRRInUnloadPosition\ |\}$$

$$RTReady =$$

let
$I_0 = SS_0$
$SS_0 = feedbelt.RTFBNextPlate \to F_0$
$F_0 = Skip$
within
I_0

$\alpha RTReady = \{| feedbelt.RTFBNextPlate |\}$

B.3.3 Rotary Robot

$RotaryRobot(queue, local) =$
let
$I_0 = INITIAL$
$INITIAL =$
$local.proc.RTRRInUnloadPosition \to UNLOADRT$
$\square$
$local.disc?e : \{| MPRRInUnloadPosition, MPRRInLoadPosition, DBRRReady |\} \to INITIAL$
$UNLOADRT =$
$RRTableUnloaded \fatsemi RRCommandUnload \fatsemi MPBARUP$
$\square$
$local.disc?e : \{| MPRRInUnloadPosition, MPRRInLoadPosition, DBRRReady, RTRRInUnloadPosition |\} \to UNLOADRT$
$MPBARUP =$
$local.proc.MPRRInUnloadPosition \to UNLOADMP$
$\square$
$local.disc?e : \{| RTRRInUnloadPosition, MPRRInLoadPosition, DBRRReady |\} \to MPBARUP$
$UNLOADMP =$
$RRCommandLoad \to MPBARDOWN$
$\square$
$local.disc?e : \{| MPRRInUnloadPosition, MPRRInLoadPosition, DBRRReady, RTRRInUnloadPosition |\} \to UNLOADMP$
$MPBARDOWN =$
$local.proc.MPRRInLoadPosition \to LOADMP$
$\square$
$local.disc?e : \{| DBRRReady, RTRRInUnloadPosition, MPRRInUnloadPosition |\} \to MPBARDOWN$
$LOADMP =$
$RRCommandPress \fatsemi RRReadyTable \fatsemi DEPOSITDB$
$\square$
$local.disc?e : \{| MPRRInUnloadPosition, MPRRInLoadPosition, DBRRReady, RTRRInUnloadPosition |\} \to LOADMP$
$DEPOSITDB =$
$RRReadyBelt \fatsemi WAITDB$

$\square$
$local.disc?e : \{| MPRRInUnloadPosition, MPRRInLoadPosition,$
$DBRRReady, RTRRInUnloadPosition |\} \rightarrow DEPOSITDB$
$WAITDB =$
$local.proc.DBRRReady \rightarrow INITIAL$
$\square$
$local.disc?e : \{| MPRRInUnloadPosition, MPRRInLoadPosition,$
$RTRRInUnloadPosition |\} \rightarrow WAITDB$
$EQ = queue?e \rightarrow local?p!e \rightarrow EQ$
within
$I_0 \; [| \{| local |\} |] \; EQ$

$ROTARYROBOT = RotaryRobot(rotaryrobot, rotaryrobotlocal)$

$\alpha ROTARYROBOT = Union(\{\{| rotaryrobot, rotaryrobotlocal |\},$
$\alpha RRCommandLoad, \alpha RRCommandUnload,$
$\alpha RRReadyTable, \alpha RRTableUnloaded,$
$\alpha RRCommandPress, \alpha RRReadyBelt\})$

$RRCommandLoad =$
let
$I_0 = SS_0$
$SS_0 = metalpress.RRMPSetLoadPosition \rightarrow F_0$
$F_0 = Skip$
within
I_0

$\alpha RRCommandLoad = \{| metalpress.RRMPSetLoadPosition |\}$

$RRCommandUnload =$
let
$I_0 = SS_0$
$SS_0 = metalpress.RRMPSetUnloadPosition \rightarrow F_0$
$F_0 = Skip$
within
I_0

$\alpha RRCommandUnload = \{| metalpress.RRMPSetUnloadPosition |\}$

$RRCommandPress =$
let
$I_0 = SS_0$
$SS_0 = metalpress.RRMPPress \rightarrow F_0$
$F_0 = Skip$
within
I_0

$\alpha RRCommandPress = \{| \; metalpress.RRMPPress \; |\}$

$RRReadyTable =$
let
$I_0 = SS_0$
$SS_0 = rotarytable.RRRTNextPlate \rightarrow F_0$
$F_0 = Skip$
within
I_0

$\alpha RRReadyTable = \{| \; rotarytable.RRRTNextPlate \; |\}$

$RRTableUnloaded =$
let
$I_0 = SS_0$
$SS_0 = rotarytable.RRRTPlateUnloaded \rightarrow F_0$
$F_0 = Skip$
within
I_0

$\alpha RRTableUnloaded = \{| \; rotarytable.RRRTPlateUnloaded \; |\}$

$RRReadyBelt =$
let
$I_0 = SS_0$
$SS_0 = depositbelt.RRDBNextPlate \rightarrow F_0$
$F_0 = Skip$
within
I_0

$\alpha RRReadyBelt = \{| \; depositbelt.RRDBNextPlate \; |\}$

B.3.4 Metal Press

$MetalPress(queue, local) =$
let
$I_0 = FORGE$
$FORGE =$
$local.proc.RRMPSetUnloadPosition \rightarrow MOVEUNLOAD$
$\square$
$local.disc?e : \{| \; RRMPSetLoadPosition, RRMPPress \; |\} \rightarrow FORGE$
$MOVEUNLOAD =$
$MPInUnload \fatsemi UNLOAD$
$\square$
$local.disc?e : \{| \; RRMPSetLoadPosition, RRMPSetUnloadPosition,$
$RRMPPress \; |\} \rightarrow MOVEUNLOAD$

$UNLOAD =$
$\quad local.proc.RRMPSetLoadPosition \rightarrow MOVELOAD$
$\quad \square$
$\quad local.disc?e : \{| RRMPSetUnloadPosition, RRMPPress |\} \rightarrow UNLOAD$
$MOVELOAD =$
$\quad MPInLoad \fatsemi LOAD$
$\quad \square$
$\quad local.disc?e : \{| RRMPSetLoadPosition, RRMPSetUnloadPosition,$
$\qquad RRMPPress |\} \rightarrow MOVELOAD$
$LOAD =$
$\quad local.proc.RRMPPress \rightarrow FORGE$
$\quad \square$
$\quad local.disc?e : \{| RRMPSetLoadPosition, RRMPSetUnloadPosition |\}$
$\qquad \rightarrow LOAD$
$EQ = queue?e \rightarrow local?p!e \rightarrow EQ$
within
$\quad I_0 [| \{| local |\} |] EQ$

$METALPRESS = MetalPress(metalpress, metalpresslocal)$

$\alpha METALPRESS =$
$\quad Union(\{\{| metalpress, metalpresslocal |\}, \alpha MPInLoad, \alpha MPInUnload\})$

$MPInLoad =$
let
$\quad I_0 = SS_0$
$\quad SS_0 = rotaryrobot.MPRRInLoadPosition \rightarrow F_0$
$\quad F_0 = Skip$
within
$\quad I_0$

$\alpha MPInLoad = \{| rotaryrobot.MPRRInLoadPosition |\}$

$MPInUnload =$
let
$\quad I_0 = SS_0$
$\quad SS_0 = rotaryrobot.MPRRInUnloadPosition \rightarrow F_0$
$\quad F_0 = Skip$
within
$\quad I_0$

$\alpha MPInUnload = \{| rotaryrobot.MPRRInUnloadPosition |\}$

B.3.5 Deposit Belt

$DepositBelt(queue, local) =$

let
$I_0 = WAITRR$
$WAITRR =$
$local.proc.RRDBNextPlate \rightarrow MOVEEND$
$\square$
$local.disc?e : \{| TCDBPlateUp, TCDBReady |\} \rightarrow WAITRR$
$MOVEEND =$
$DBPlateEnd \fatsemi WAITTC$
$\square$
$local.disc?e : \{| RRDBNextPlate, TCDBPlateUp, TCDBReady |\}$
$\rightarrow MOVEEND$
$WAITTC =$
$local.proc.TCDBPlateUp \rightarrow WAITDROP$
$\square$
$local.disc?e : \{| RRDBNextPlate, TCDBReady |\} \rightarrow WAITTC$
$WAITDROP =$
$local.proc.TCDBReady \fatsemi DBReady \fatsemi WAITRR$
$\square$
$local.disc?e : \{| RRDBNextPlate, TCDBPlateUp |\} \rightarrow WAITDROP$
$EQ = queue?e \rightarrow local?p!e \rightarrow EQ$
within
$I_0 \; [| \; \{| \; local \; |\} \; |] \; EQ$

$DEPOSITBELT = DepositBelt(depositbelt, depositbeltlocal)$

$\alpha DEPOSITBELT =$
$Union(\{\{| depositbelt, depositbeltlocal |\}, \alpha DBPlateEnd, \alpha DBReady\})$

$DBPlateEnd =$
let
$I_0 = SS_0$
$SS_0 = travellingcrane.DBTCPlateEnd \rightarrow F_0$
$F_0 = Skip$
within
I_0

$\alpha DBPlateEnd = \{| travellingcrane.DBTCPlateEnd |\}$

$DBReady =$
let
$I_0 = SS_0$
$SS_0 = rotaryrobot.DBRRReady \rightarrow F_0$
$F_0 = Skip$
within
I_0

$\alpha DBReady = \{| rotaryrobot.DBRRReady |\}$

B.3.6 Travelling Crane

$TravellingCrane(queue, local) =$
let
$I_0 = READY$
$READY =$
$local.proc.DBTCPlateEnd \rightarrow GRASP$
$GRASP =$
$TCPlateGrasped \,;\, DROP$
$\square$
$local.disc?e : \{| \, DBTCPlateEnd \, |\} \rightarrow GRASP$
$DROP =$
$TCPlateDropped \rightarrow READY$
$\square$
$local.disc?e : \{| \, DBTCPlateEnd \, |\} \rightarrow DROP$
$EQ = queue?e \rightarrow local?p!e \rightarrow EQ$
within
$I_0 \; [| \, \{| \, local \, |\} \, |] \; EQ$

$TRAVELLINGCRANE = TravellingCrane(travellingcrane, travellingcranelocal)$

$\alpha TRAVELLINGCRANE = Union(\{\{| \, travellingcrane, travellingcranelocal \, |\},$
$\alpha TCPlateDropped, \alpha TCPlateGrasped\})$

$TCPlateDropped =$
let
$I_0 = SS_0$
$SS_0 = depositbelt.TCDBReady \rightarrow F_0$
$F_0 = Skip$
within
I_0

$\alpha TCPlateDropped = \{| \, depositbelt.TCDBReady \, |\}$

$TCPlateGrasped =$
let
$I_0 = SS_0$
$SS_0 = depositbelt.TCDBPlateUp \rightarrow F_0$
$F_0 = Skip$
within
I_0

$\alpha TCPlateGrasped = \{| \, depositbelt.TCDBPlateUp \, |\}$