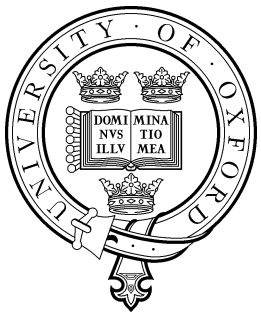
Model-Driven Development of Information Systems



Chen-Wei Wang
Kellogg College
University of Oxford

A thesis submitted for the degree of
Doctor of Philosophy

Hilary 2012

Abstract

The research presented in this thesis is aimed at developing reliable information systems through the application of model-driven and formal techniques. These are techniques in which a precise, formal model of system behaviour is exploited as source code. As such a model may be more abstract, and more concise, than source code written in a conventional programming language, it should be easier and more economical to create, to analyse, and to change. The quality of the model of the system can be ensured through certain kinds of formal analysis and fixed accordingly if necessary. Most valuably, the model serves as the basis for the automated generation or configuration of a working system.

This thesis provides four research contributions. The first involves the analysis of a proposed modelling language targeted at the model-driven development of information systems. Logical properties of the language are derived, as are properties of its compiled form—a guarded substitution notation. The second involves the extension of this language, and its semantics, to permit the description of workflows on information systems. Workflows described in this way may be analysed to determine, in advance of execution, the extent to which their concurrent execution may introduce the possibility of deadlock or blocking: a condition that, in this context, is synonymous with a failure to achieve the specified outcome. The third contribution concerns the validation of models written in this language by adapting existing techniques of software testing to the analysis of design models. A methodology is presented for checking model consistency, on the basis of a generated test suite, against the intended requirements. The fourth and final contribution is the presentation of an implementation strategy for the language, targeted at standard, relational databases, and an argument for its correctness, based on a simple, set-theoretic semantics for structure and operations.

Acknowledgements

I offer my sincerest respect and deepest gratitude to my supervisor, Professor Jim Davies, without whose guidance the completion of this thesis would never have been possible. Jim has always been patient, thorough, encouraging, and supportive at each stage of my doctorate. Jim's unique, charming, and decent character has impacted upon the kind of researcher and teacher which I wish to become. Jim: Thank you for opening up for me a new world, new perspectives, and new expectations.

Jonathan: Thank you for giving me many insights and advice, for sharing your wealth of knowledge, and for demonstrating your high self-discipline. Professor Jonathan Ostroff at York University in Canada was my undergraduate instructor of a software design course. I had the greatest pleasure of working under his supervision on an Eiffel software verification project. Without the working experience with Jonathan, my privilege of pursuing a degree in Oxford indeed would not have been possible in the first place.

I am truly grateful to the Software Engineering Programme, by which I had been funded as a teaching assistant throughout my five-year study in Oxford. I benefited a lot from discussions with researchers working in the same BOOSTER project: Dr. Ib Holm Sørensen, may he rest in peace, who was always thorough in explaining stories about the B method and the BOOSTER technology; Edward Crichton, James Welch, and Dr. David Faitelson who were always accessible when I had difficulties.

Many thanks to all of my examiners whose feedback contributed to the ultimate merit of this thesis: Professor Jeremy Gibbons and Dr. Niki Trigoni in my transfer of status, Dr. Andrew Martin and Dr. Steve McKeever in my confirmation of status, and Dr. Steve King and Dr. Steve McKeever in my final viva. I also thank Dr. Alessandra Cavarra and James Welch for their collaborations on papers which contribute toward this thesis.

I am deeply indebted to two of my undergraduate instructors at York: Dr. Peter Cribb and Dr. Natalija Vlajic. They were both incredibly friendly, helpful, and encouraging to me when I first began to study computer science. I very much appreciate the after-class hours they spent with me working through basic concepts and problems.

A grateful note goes to Dr. Danielle Russell, from Glendon College of York University in Canada, who helped me polish the language of this thesis. I shall owe my deepest apology to Danielle, for her expertise is in fact on English literature and thus she could only parse these hundreds of pages, while being prevented from performing any further semantic analysis.

I owe a very great deal to my friends in Oxford. I thank my fellow Canadian Dr. Eric Kerfoot who used to work with me in the summer of 2005 under Jonathan's supervision at York. Eric then started his doctorate in the same department (called the computing laboratory at that time) when I was in my final year of undergraduate studies, and it was his vivid stories about being a freshman in Oxford that encouraged me to apply abroad.

I am much obliged to Hsiang-Shang (Josh) Ko, currently a second-year doctoral student from the same department, who helped me tremendously on both typographical and technical aspects of those countless thesis drafts. Josh always stayed patient and encouraged me in dealing with my writer's block.

I will always cherish the quality time spent with my best friend from the Department of Education, Dr. Jang-Ho (Chris) Lee, who was as close to me as the best high-school playmate while we shared a flat together in Summertown. Chris always enjoy teasing me on my thesis subject of concurrency, just as I do on his subject of teacher's code switching.

Another friend of mine to thank is Machi Sato, also from the Department of Education, with whom I shared pleasant moments of classic concerts, formal dinners, and musicals in Oxford colleges and theatres.

Last but not least, I dedicate this thesis to my dearest, beloved parents: Sarah and Albert. Both myself and my elder brother, Blake, who has been awarded his S.J.D. in the United States, would not have achieved what we have achieved without our parents' persistent, unreserved love since forever.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

This thesis arose from the challenge of developing a reliable information system. We will utilise the following definition[1] of information systems:

Definition 1.1.1 (***Information Systems***) Information systems are computing systems for the collection and provision of (large quantities of) business data. These data normally have subtle relationships, constrained by complex business rules, and are updated through intertwining patterns of operations. □

Information systems are costly to develop and maintain, and the value of the data that they hold may be greater than the cost of the system itself. If the integrity of this data is compromised, queries about the system may fail to return a response, or they may return spurious results. For example, navigation between two runtime objects may become impossible if the link between them has not been properly maintained; or a system may assert that the same person exists in two places at once.

One approach to guaranteeing the proper maintenance of integrity constraints is through calculating and implementing appropriate *guards*—predicates to be evaluated before operations are invoked. The validity of operation guards is sufficient to guarantee that the integrity of the system is maintained in all subsequent states; however, the manual calculation of guards for operations is expensive and prone to

[1] We will elaborate on this definition of information systems in Chapter 2.

error. Even after operations are suitably guarded against the violation of system integrity, a new challenge arises when multiple, related operations are grouped into workflows that operate upon a shared system. As updates from workflow instances are interleaved, the state of the system may be modified in such a way that some of these workflow instances cannot proceed to completion as the guards of their next operations are not satisfied.

Model-driven development [1] advocates the idea of directing the attention of developers to an abstract, yet informative, description of the system—i.e. a *model*—rather than low-level code with an overwhelming volume of platform-dependent details. Our goal is to calculate appropriate guards for operations at the model level—instead of code—level. Techniques drawn from formal methods [2] can help us apply mathematical reasoning to make such a process of calculation both precise and systematic. Moreover, in the context of model-driven development, a model of system forms the basis of an automated process of model transformation which generates working code that implements its intended functionalities. On the one hand, it is important for us to ensure that code produced by the transformation engine is faithful to its source model. On the other hand, it is even more imperative that the source model is consistent—with respect to the intended requirements—in the first place.

In light of the above challenges of developing an information system, we will adopt the following notion of system *reliability*. The first criterion is that the system design must be consistent with the developers' understanding of the requirements—not necessarily for all possible behaviours of the system, but at least for those use-case scenarios that they are concerned with. The second criterion is that operations and workflows must be suitably guarded in order to preserve the system integrity and to ensure that workflow instances, once initiated, are able to complete successfully. Finally, we require that the runtime behaviour—resulting from the implementation of the system—must be faithful to its abstract design.

This thesis will build upon the existing work at Oxford on Booster [3]—a language and compiler targeted at the model-driven development of information systems.

Currently the BOOSTER compiler automatically generates—from object models—in-memory programs written in C. The aim of this thesis is to extend the range of BOOSTER, in several aspects. Nonetheless, theories that we build atop the BOOSTER notation are transferrable to other common approaches of object modelling (e.g. [4]).

As a case study for this dissertation, we will apply the developed methodologies and theories to a typical information system—a hotel reservation system. In this case study all possible scenarios of relating entity classes are addressed, and model operations exemplify those of an information system—finite operations that perform simple transformations on the system data. In other words, another information system of a larger scale will differ in only the numbers of entity classes and supported operations, but its complexity of entity relationship and of each individual operation shall resemble that of the case study. Therefore, our case study of the hotel reservation system is sufficiently representative for building the thesis argument. In the following four subsections we will discuss how the research contributions of this thesis fit into the development context of the case study.

We envisage in Figure 1.1[2] how the four contributions of this thesis fit into the development context of the hotel reservation system.

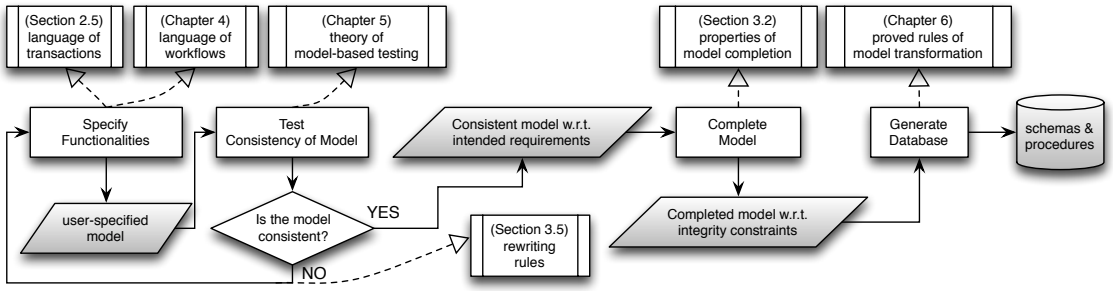


Figure 1.1: Contributions in the Development Context of an Information System

[2]The three *shaded parallelograms* denote the same design model of the system at different stages of development; the *shaded cylinder* denotes the ultimate, automatically-generated relational database; the four *rectangles* represent processes that gradually transform the source model to the ultimate working system; the six *double-lined rectangles* are formal theories that are developed in this dissertation to support the corresponding processes; the *diamond* denotes a point of decision; and finally *solid lines* represent the directed flow of processes, and *dashed lines* the direction of dependency.

Developers need a modelling language (Section 2.5) for describing the intended behavioural functionality as operations, as well as the structural and semantic integrity that will constrain the runtime *availability* of these operations. That is, the system integrity constraints define under what circumstances the implementing guarded transactions of operations will be blocked at runtime. Developers also need a modelling language (Chapter 4) for specifying workflows that are composed of operations. The semantics of workflow combinators should be defined on the basis of that of the referenced operations. Furthermore, in an execution context, where workflow instances are initiated by distributed clients, the specification of workflows should facilitate a systematic calculation of an appropriate precondition—which incorporates the account of their interference with each other—for them to successfully complete.

When a version of the design model arrives, developers need a process of model-based testing (Chapter 5) to be applied to this model. Developers will inspect the test results against their understandings of the requirements. In cases where inconsistency is revealed, it would be valuable to support this process through a list of rewriting laws (Section 3.5), in order to alter the relevant specifications of operations. Developers may repeat this testing process until they arrive at a version of the model whose test results are considered satisfactory.

A consistent model may not, however, necessarily be itself complete. We may still need to amend (Section 3.2), for some operations, the guarding constraints and intended updates in order for the maintenance of system integrity. Finally, once a version of the model that is both consistent and complete arrives, developers need an automated process of model transformation (Chapter 6) that implements abstract model operations. We will implement operations that update attributes and associations as Structured Query Language (SQL) [5] database queries that update the corresponding tables and columns. We need the guarantee that queries produced by the transformation engine are not only correct, but also efficient. In the following four sub-sections we will state the research problems and how we plan to tackle them.

1.1.1 Guarded Transactions

The *integrity* of data stored in an information system is characterised through complex constraints and business rules which must not be compromised between updates on the system. The maintenance of such integrity gives value to the stored data. Let us consider the development scenario of a hotel reservation system where we outline how the developers should describe the system integrity constraints and operations.

Motivating Scenario 1.1.1 (***Guarded Transactions***) On designing a hotel reservation system, there are three aspects that we ought to be concerned with—each of these three aspects requires a clean and easy-to-use syntax that is also amenable to later phases of formal analysis. First, we must declare the classes or entities that are involved and the relationship between them, e.g. a reservation belongs to a host hotel, which has that reservation in its list of records. Second, we may wish to impose business rules—which reflect facts about and constraints on the stored entities—upon the system, e.g. a reservation can be made at a hotel only if its current number of bookings does not exceed a certain limit. Third, we must carefully describe update operations that we wish to apply on the system to change the state of relevant data components, e.g. to make a reservation at a hotel. In addition, we have not only to properly maintain the relationship (or links) between all reservation and hotel objects, but also to ensure that we invoke that operation only when allowed by all the business rules in context. □

In light of the above scenario, we state our first research problem (**RP1**): the specification of transactions as relational constraints upon system attributes, together with a well-defined formal semantics, suitably reflecting the blocking policy that we wish to adopt at runtime.

Since our emphasis on system updates is upon data rather than computation, we will adopt the transformational style of specification as can be found in constraint-oriented formal languages such as VDM [6], Z [7], and OCL [8] notation for annotating

UML class diagrams [4]. We will capture integrity constraints as *invariants*, the intended effect of each operation as its *post-condition*, and the circumstances under which that operation can be safely invoked as its *pre-condition*. More precisely, the pre-condition of an operation is a constraint on values of data items and input variables before it is invoked. Its post-condition, on the other hand, is a transformation on states—a mathematical relation in which output variables and values of data items (*after* the operation is completed) are expressed in terms of those of input variables and data items (*before* it is invoked).

At runtime we will implement each model operation as a *guarded transaction*—the post-condition of it a transaction, and the pre-condition of it a guard. A transaction is atomic in its execution—which either completes successfully or leaves the state of the system intact—and during its execution no interference from other updates is possible. A guard will *block* its associated operation unless its corresponding precondition is satisfied by the current state of the system. Any transaction that is not *blocked* is guaranteed to succeed, provided that the state of the system does not change between the successful evaluation of its guard and the initiation of its transactional update.

We will design a modelling language that enables system developers to describe model operations as simple data transformations. We will demonstrate that first-order predicates are sufficiently expressive for our purpose of specification. We will define a formal semantics for this modelling language which suitably incorporates the runtime policy (i.e. blocking upon failures of guards) that we wish to adopt. We will then explore the formal consequence of such formal semantics by developing a list of rewriting rules for our predicative specifications. In Chapter 3 of this dissertation, we will cover all these aspects of the proposed modelling language: syntax, semantics, and analysis. This first contribution has been validated in part through conducting formal proofs for the list of refinement and equivalence laws, and through a number of applications of these laws on the case study.

1.1.2 Guarded Workflows

Most updates upon an information system will not be performed as single, isolated operations—either because a series of separate interactions are required (e.g. for requests of inputs), or because it is not feasible to restrict access to the data items involved for the entire duration of the updating process. Instead, these updates will be performed as *workflows*—patterns of operations enacted to achieve a specific goal, while allowing interference by other operations or workflows.

Workflows composed of guarded transactions will preserve the integrity of the data, as each component transaction will do so, but they are not guaranteed to complete successfully. These workflows may interfere with one another, even to the extent of updating the shared state in such a way that one or more workflow instances, currently executing, may be unable to continue. Let us consider the scenario of a hotel reservation system that amply demonstrates this issue of successful completion.

Motivating Scenario 1.1.2 (***Guarded Workflows***) If a hotel reservation system allows a new booking workflow to proceed while a customer is in the process of confirming a booking for the last available room, some disappointment may ensue. The customer who initiated the new booking workflow will suddenly be notified that, although they were allowed by the system to start, they cannot proceed by reserving a new room. □

In light of the above scenario, we state our second research problem (**RP2**): the specification and regulation of concurrent workflows, when running the generated system, in such a way that their interference with each other does not cause any of them to not be able to complete.

Indeed, systematically regulating to what extent instances of workflows may be executed in parallel *safely*—as far as producing a particular outcome in terms of completion, outputs, and desired resulting state of system is concerned—makes an essential contribution to the quality of an information system. In this situation, what we will aim for are *guarded workflows*. Knowing in advance which set of workflows are

to be run in parallel, we calculate a precondition whose validity can guarantee that these workflows reach completion from the current state of system. Not surprisingly, the challenge is that it can be difficult to predict the effects of a combination of workflows, executed concurrently, each composed of a number of guarded operations.

The calculation should be on the basis of both the formal specification of their component operations and the formal semantics of the workflow language. As each operation referenced in a workflow is characterised as a state transformation, and any unbounded iteration or recursion is syntactically forbidden, we are then able to calculate a precondition for the successful completion of a given combination of workflows. Should such a precondition prove too strong—with respect to the intended requirements—then the design of involved workflows can be modified; otherwise, this calculated precondition will be implemented as the guard upon the workflows involved. In Chapter 4 of this dissertation we will show that such an analysis is possible. This second contribution has been validated in part through publication [9].

1.1.3 Testing of Model Consistency

The process of implementing a system model may either be manual or automatic. In the former case, the source model is written in a general-purpose programming language such as Java [10] or C# [11]; it is then necessary to formally establish—via e.g. automated or interactive theorem proving—that the resulting implementation conforms to its source model. In the latter case, the model is written in a notation specific to the problem domain (e.g. that of information systems), whose assumptions and pre-defined rules form the basis of the generation process; it is still necessary to verify—though at the level of transformation rules—that any generated implementation is guaranteed to preserve the behaviour of its source model.

However, if the various structural and semantic constraints of integrity expressed in the model are *inconsistent*, then either the model will admit no implementation, or the implementation produced will not behave according to the intended requirements. Even worse is the fact that as the number of business rules that are imposed upon the

system model grows, the interplay between these various integrity constraints makes it difficult for the exact behaviour of the model to be fully predicated—even by its own developers. Let us consider scenarios where the developers of a hotel reservation system may perform certain types of tests directly upon the model they have designed.

Motivating Scenario 1.1.3 (***Testing of Model Consistency***) On completing the design model of a hotel reservation system, the developers may not be willing to predicate the behaviour of its entirety. Instead, they may find it useful to test their model by "running" use cases on it. We will consider three example use cases here. The first case is to test under what circumstances the data attribute, which denotes the set of reservations of a hotel, can be modified; if the result is that it is possible only when the current number of reservations is below, say 200, and if the developers find this restriction unnecessary, then they may eliminate the relevant invariant in the model accordingly. The second case is to test if it is possible to confirm a booking after it has been cancelled; if the result is that this is possible, then it is most likely that the developers did not equip the "confirm" operation with a suitable pre-condition, i.e. the booking to be confirmed must exist. The third case is the opposite of the second—we do not care about the order in which we cancel (an unconfirmed booking) and deallocate (an existing, confirmed booking), but only wish to test how possible it is for both operations to be applied. □

In light of the above scenario, we state our third research problem (**RP3**): the validation of the consistency of a system model at both levels of its defined operations and workflows with respect to the intended requirements.

We will tackle this issue of model consistency by not requiring the developers to conduct an exhaustive exploration of the model state space, and to consider all possible interactions between operations or workflows. That is, the developers ought to focus only on aspects that interest them. The novelty of our approach is that tests are performed on abstract models that are described through declarations of entities and their relationship, and through formal specification of operations and workflows.

We will adapt techniques drawn from both program slicing [12] and data-flow testing [13]—conventionally applied to low-level, imperative implementation code—to the context of formal, declarative design models. The rationale is that running such tests on design models, and eliminating any found inconsistency accordingly in the first place, is far more effective than later running tests on the concrete systems.

In Chapter 5 of this dissertation we will introduce a formal and model-based approach—as visualised in Figure 1.2—to testing design models. To test the model we supply test cases which represent use-case scenarios that we are interested in exploring. A use-case scenario may be that certain patterns of data usages or of operations must occur, that specific set(s) of operations may occur by any ordering, or that a combination of several such scenarios must (or must not) occur. Each result we obtain is the combination of an operation *trace* and a guarding *constraint*. The trace represents an instance of the model behaviour that demonstrates the use-case scenarios we supplied. The constraint abstractly characterises the circumstances under which the corresponding instance of model behaviour is guaranteed to complete successfully. This third contribution has been validated in part through publication [14, 15].

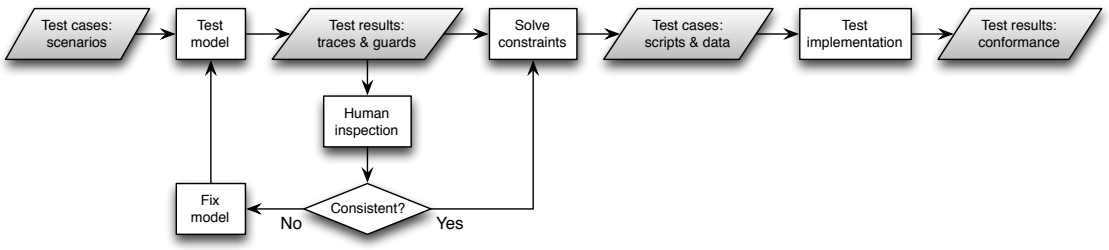


Figure 1.2: Model-Based Testing of Design Models (elaborated in Figure 5.1 on page 107)

The value of our approach is that developers may examine these *trace-guard* pairs in relation to their expectations and fix—if necessary—the relevant model constraints accordingly; this process may go iteratively until they are satisfied with all test results. An obvious by-product of our approach is that results from tests of models (i.e. traces and constraints) can be straightforwardly adapted—via constraint solving, e.g. [16]—to test cases (i.e. test scripts and concrete data values) for their implementing systems.

1.1.4 Model-Driven Engineering of Relational Databases

Our analysis and tests of design models are based upon the assumption that their implementations at runtime will behave consistently with their formal, precise specification. However, when the implementation is done by a third party, there is no guarantee of whether or not the transformation process—from an abstract model to its implementing code—is provably correct. Let us consider the development scenario of a hotel reservation system where not only a mechanism of automatic implementation is desired, but also its production process should provide certain formal guarantees on the correctness of the generated system.

Motivating Scenario 1.1.4 (***MDE of RDB***) The developers have decided to store the hotel reservation system data in a relational database. However, it would be a tedious and error-prone process for them to switch between contexts of objects and tables when manually implementing the model operations as stored procedures of SQL queries. Moreover, although we would expect any implementation platform of the SQL standard to have satisfactory support on indexing and caching, there is no guarantee on this matter, and this is subject to empirical analysis after the actual system has already been deployed. In the case where developers find that the support on these mechanisms is not satisfactory, it would be inconvenient to migrate to another implementation platform as all model operations have to be (manually) re-implemented. For this reason, it would be advantageous for us to equip all stored procedures with a caching mechanism; however, this task is destined to be error-prone if it has to be completed manually. □

In light of the above scenario, we state our final research problem (**RP4**): the automatic generation—from a system model—of an implementing relational database that is both correct and efficient on evaluating access paths.

At one extreme, if the system were written manually by programmers who referred to the model at appropriate points, then the guarantee of its correctness would be only as promising as their skills and experience. At the other extreme, we might hope

for a *verifying compiler* [17], which would be able to check—formally—whether any proposed code was faithful to the constraints of the model.

In Chapter 6 of this dissertation we will explore a model-driven approach—as visualised in Figure 1.3—which is situated somewhere between the above two extremes and produces a *verified compiler* for relational databases. This fourth contribution has been validated in part through a prototype implementation of the transformation using a functional programming language to mirror the mathematical descriptions, upon which the correctness proof of the transformation is based. The source code is a Booster system and the target platform is that of a Sql database. In the source Booster system we declare associations and invariants for expressing, respectively, the structural and semantic integrity. In addition, we adopt our own programming notion, which extends Abrial's generalised substitution language [18], to formally describe model operations.

The textual Booster system is first compiled into an Object model that formalises its features, particularly paths that access entities by navigating between associated classes. We then transform this Object model into a Table model that reflects our strategy of implementing classes and associations as Sql tables. We must also adapt paths in the Object model into their equivalents in the Table model accordingly. We again transform this Table model into a Sql model which supports an abstract syntax of database tables and queries; and we also impose a caching mechanism on paths that effectively rules out unnecessary re-evaluations of their values. Finally, from tables and queries contained in the Sql model, we can generate a working Sql database by straightforwardly rewriting them in their corresponding concrete syntax [5].

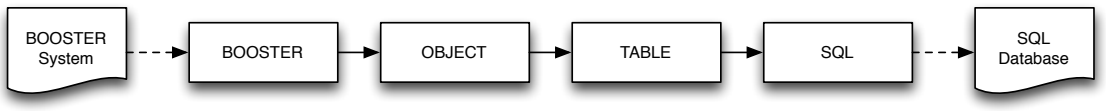


Figure 1.3: An Abridged Overview of Booster-to-Sql Model Transformation

The nature of our transformation pipeline as presented in Figure 1.3 is sophisticated; however, such sophistication is unavoidable in order to separate concerns that are essential to our transformation, e.g. implementation strategy, caching of paths, etc. More valuably, we will establish that the resulting model from each of the intermediate transformations preserves the semantics of its source model. The value of our model-driven approach is that the solution space of developers will be altered—from one that consists of scattered tables and rows to one that is defined through classes, associations, and entities. On the one hand, the conceptual gap between the problem space (of information systems) and the solution space in our approach is undoubtedly far narrower. On the other hand, the generated table schemas, as well as update queries that are both correct and efficient, are as if they were written by experienced SQL programmers in the first place. We have included, in Section E.12, (part of) the database—comprising table schema, key constraints, and queries—which correctly and efficiently implements our hotel reservation in Section A.2.

The most critical parts of our model transformation are Tables E.1 (on single-valued bi-associations), E.2 (on set-valued associations), E.3 (on sequence-valued associations), and E.4 (on primitive attributes)—as included in Appendix E.6—that summarise the rules of handling the 36 patterns of basic assignments. These patterns result from all possible ways of declaring properties—as either attributes or bi-directional associations—and changing their values. As a brief demonstration, consider a case where there is a 1-to-*optional* bi-directional association:

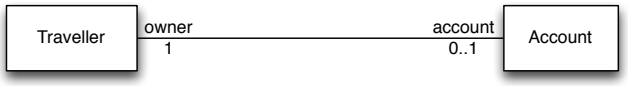


Figure 1.4: An example bi-directional association in the hotel reservation system

A traveller may or may not have an account, but an account must have an owner. In the context of class `Account`, we would expect an assignment `this.owner := t?`—`t?` is input traveller object—to be implemented as follows. As we are updating a

1-to-*optional* bi-association, we will refer to Patterns #2 and #5 in Table E.1 (on page 250). Furthermore, since we are updating the association by assigning a new value to the association end that is mandatory, i.e. *owner*, we will apply the rules on Pattern #2. As a result, we will expect the following two queries to be automatically generated to update the corresponding table that stores the association:

```
DELETE FROM 'Account_owner_Traveller_account'
       WHERE 'account' = 'this';
INSERT INTO 'Account_owner_Traveller_account'
       ('owner', 'account') VALUE ('t?', 'this');
```

where the first query clears the current link of the input, if any, and the second query adds the new link into the storing table. We have included a complete model specification of the hotel reservation system in Appendix A.2, where each assignment is annotated with a comment which specifies which assignment pattern it will trigger the model transformation engine to apply. The value of incorporating all the identified 36 patterns of basic assignments into the transformation engine is arguable—having these written in a manual for developers to refer to, for each encounter of a primitive assignment, is destined to be error-prone.

1.2 Dissertation Structure

We will summarise the structure and contents of this dissertation as follows:

Chapter 1: Introduction presents the motivation of this thesis, lists its research problems and contributions, and outlines the structure of this dissertation.

Chapter 2: Context gives particular interpretations to the concept of model-driven development of information systems, sketches the theoretical context, presents the thesis hypotheses, and overviews the BOOSTER notation and the extended notion of substitution language that we use in later chapters.

Chapter 3: A Theory of BOOSTER Predicates presents laws of BOOSTER predicates—derived from a weakest precondition semantics of its abstract implemen-

tation language—and demonstrates how these laws may enable an automated process of symbolic execution that safely rewrites the predicative specification.

Chapter 4: Guarded Workflows defines a workflow language on information systems and its formal semantics, and suggests a strategy—proved for its correctness—of calculating the precondition for parallel workflows to complete.

Chapter 5: Testing of Model Consistency introduces an approach that adapts techniques of program slicing and data-flow testing to validate the consistency of design models, and demonstrates that testing the model is more effective than testing the working system that is derived from it.

Chapter 6: Database Development presents the core transformation functions for turning a given BOOSTER object model into a relational SQL database, provides a formal account of the model transformation, and establishes that the behavioural semantics of the relational database is consistent with that of the object model.

Chapter 7: Discussion summarises how the application of our developed theories and methodologies on the case study has tested and confirmed the thesis hypotheses, reflects limitations of the thesis, and suggests future research accordingly.

We construct a road map of this dissertation in Figure 1.5: the four result chapters, starting from Chapter 3, address the above research problems in order; the results presented in these chapters constitute the four research contributions of this thesis.

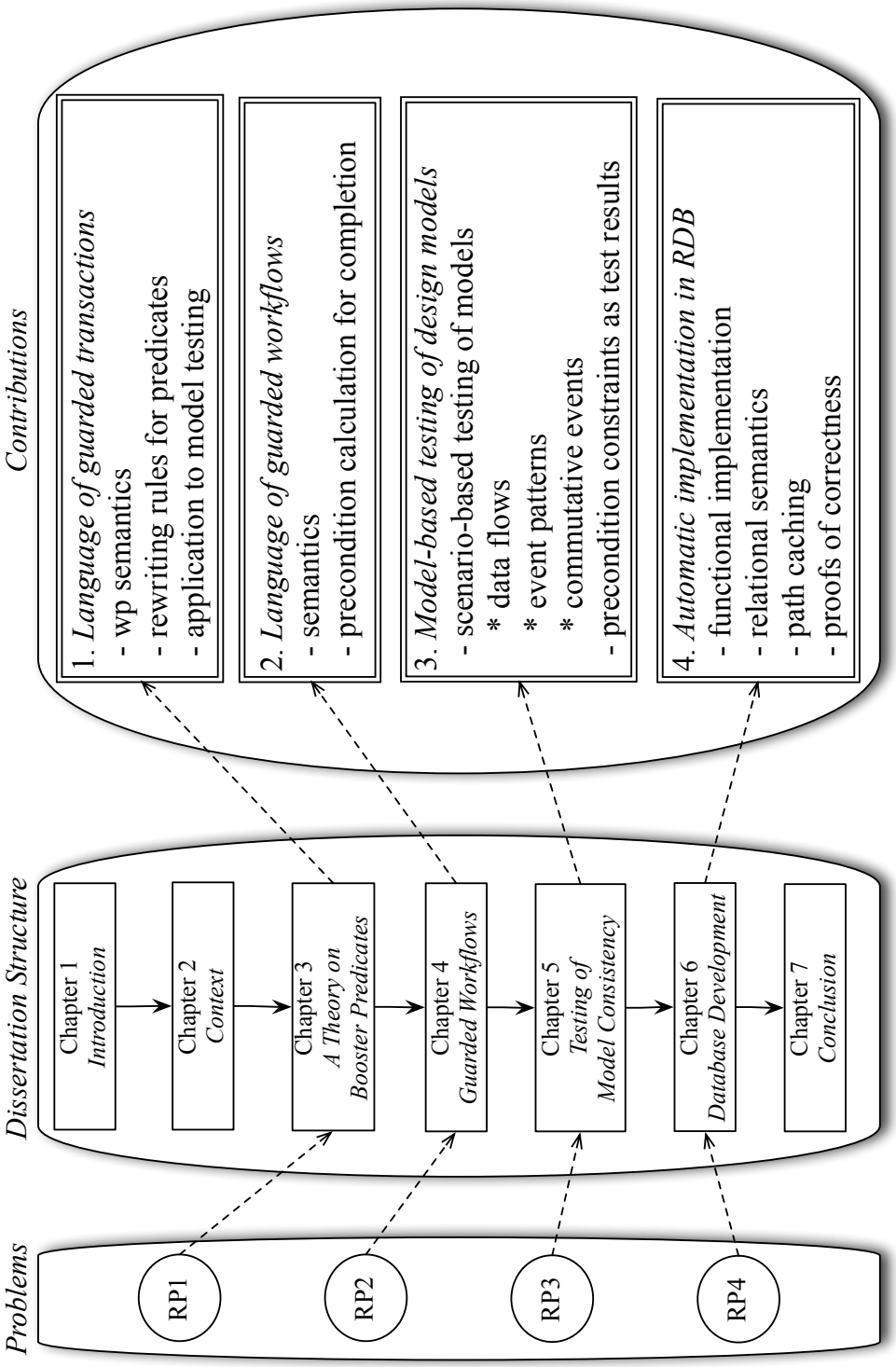


Figure 1.5: Roadmap of the Dissertation

Chapter 2

Context

Throughout the course of this chapter we will build the hypothesis of this thesis in regard to how to build a reliable information system. Our thesis hypothesis comprises four parts, each of which is claimed on the basis of the inability of relevant existing works to handle issues that arise in the four motivating scenarios as discussed in Chapter 1. As we already argued in Chapter 1, we use methods of the hotel reservation system to exemplify model operations in the application domain of information systems: simple data transformations rather than complex algorithmic procedures. Furthermore, all possible declarations of entity relationships will be covered by the case study. For this reason, this dissertation will not present another case study of a larger scale—e.g. the current website of *Oxford University Department of Computer Science*[1]. Although the number of its declared classes is much larger, the complexity of each of its model operations would resemble those presented in this case study. In Sections 2.1 to 2.4 we will state our thesis hypothesis on the basis of building a reliable hotel reservation system. We will also review features of BOOSTER that are relevant to our study in later result chapters.

[1]This website (`http://www.cs.ox.ac.uk/`) has been created and maintained using the BOOSTER technology. Its design model contains nearly 90 classes.

2.1 Semantic Integrity of Information Systems

Information Systems emerged as a distinct field of study in the 1980s, marked by the appearance of, for example, the ACM's *Transactions on Office Information Systems* journal. The first edition of that journal [19] argued strongly for the interdisciplinary nature of the field, involving the application of computing theory in the context of a working office. The ACM characterisation [20] emphasises *information access*, *knowledge representation*, and *user interactions*. The *Journal of Universal Computer Science* [21] describes the field, from a computer science perspective, by listing the following categories for research: models and principles; database management; information storage and retrieval; applications; interfaces and presentation.

Semantic integrity, on the other hand, has been a recurring theme in the research into database design. Integrity constraints are seen as defining the boundary of the "world", or the domain of interest, for the facts in the database. If all updates maintain the "truth" of the stored facts, then all queries are guaranteed to obtain only "true" results. For relational databases, we may express integrity constraints at the SQL programming level [22] with different *enforcement granularities*—row, table, or inter-table—and define *procedural triggers* to respond to violations of stated constraints. However, expressing constraints directly at the programming level can be problematic: developers tend to concentrate upon functionality, rather than constraints, and it can be difficult to choose an appropriate enforcement granularity or trigger type. More seriously, it is difficult to strike a balance between the *normalisation* [23]—which eliminates dependencies that could cause inconsistent updates—and *denormalisation* [24]—which benefits the performance of query processing—of the database in question.

For the purpose of this dissertation, we adapt the programming notion of semantic integrity to the design context of information systems—against objects, entities, associations, or relationships [25, 26]. As redundant data and dependencies provide for a more accessible view of the system in terms of conceptual integrity [27], the

developers can more easily express complex integrity constraints that correspond to domain properties, e.g. business rules. Our key concern is the *meaning* of data—which constitutes the integrity of their storing system—given through business or semantic constraints, and through relationships with other data. It is then critical to properly maintain this meaning as the data are updated—through either operations or their compositions, i.e. workflows. In light of the emphasis on meanings of data, we need to elaborate on our Definition 1.1.1 of information systems.

Definition 2.1.1 (***Information Systems***) Information systems are computing systems in which emphasis is placed upon: 1) the representation of the information they possess—stored as concrete values of data; and 2) the update of this information—through simple transformations, rather than complex, algorithmic computations, on the stored data. □

Upon each system update, there are three options available as to how we enforce the intended integrity constraints: 1) run a check upon its completion, and roll back if the integrity has been compromised; 2) run a check upon its completion, and fire triggers to repair the integrity if necessary; and 3) modify the update in such a way that it incorporates guarding constraints or actions that are needed to maintain the integrity.

Research on the first approach has focussed upon improving the efficiency of the checking process by, for example, simplifying logic constraints to be checked upon the completion of updates [28], or restricting the check to a subset of the database, based upon a formal analysis of the specification of the update operation [29]. However, this can still lead to a significant performance overhead, and failed transactions can be costly and inconvenient to roll back.

The second approach effectively focuses upon building *active database systems* [30]—ones that extend relational databases by *Rules Triggering Systems* (RTSs) [31]. The process requires the developers to formulate a set of repairing actions in response to events causing integrity violations. Research efforts have been dedicated to support-

ing constraint and trigger facilities—in either OODBMS [32, 33] or RDBMS [34]—and automatically generating at compile time, based on the stated constraints, a set of production rules [35] for repairing the database. The data, however, may be at greater risk: integrity has been violated, but there is nothing to prevent the specific "repair" action from making matters worse. The "repair" action may trigger an unbounded sequence of further "repairs", and it can be difficult to ensure that the intended effect of the user-defined transition will be preserved [36].

The third approach is more promising in that the task of maintaining integrity is handed over to a compiler that automatically rewrites proposed updates. A theoretical basis for this has been defined in terms of the *Greatest Consistent Specialisation* (GCS) of an update [37]: a *consistent* update is one that is guaranteed to satisfy all of the database consistency constraints; a *specialisation* is an update guaranteed to achieve the same (specified) goal, in at least the same set of circumstances; and the *greatest* consistent specialisation is the most *nondeterministic* consistent specialisation. The last characteristic, however, concerns only the effect of update, but ignores its availability. More precisely, GCS has Dijkstra's [38] notion of preconditions—i.e. $wp(g \to p, \varphi) = g \Rightarrow wp(p, \varphi)$—where the weakest precondition for a program p, guarded by g, to achieve φ includes the possibility that g is *false*. Instead, what we will adopt is Nelson's [39] notion of guards—i.e. $wp(g \to p, \varphi) = g \wedge wp(p, \varphi)$—which reflects our desired runtime property—the operation is *blocked* if its precondition, implemented as a guard, is not satisfied. Moreover, the notion of *specialisation* used in [40] relies upon the definition of a frame for the update; a frame is the set of attributes whose values may be updated, which may be a superset of those mentioned in the description of the "effect". It is shown in [40] that a GCS can be defined for any update with any frame, but the definition is not constructive, and attempts to establish a set of compositional rules for determining a GCS have foundered.

The first part of our thesis hypothesis will address the limitations of existing works. Let us first consider how the proposed solution should be applied to a hotel reservation system.

Example (*Maintaining System Integrity*) We use the system attribute *allocations* to denote the set of confirmed bookings of a hotel, and its cardinality is bounded by the number of rooms existing in that hotel. If the effect of an operation *allocate* has the potential of updating *allocations* such that its cardinality exceeds the physical limit, then we would expect the compiler to have incorporated a proper precondition in the specification of *allocate*—if not already specified by the developers—so that we will be able to block it when necessary at runtime. Obviously, the intent of *allocate* and the relevant model constraint(s) upon *allocations* must be precisely captured—as we have learnt from Scenario 1.1.1—in order for the compiler to proceed. □

We now state the first part of our thesis hypothesis:

Thesis Hypothesis 2.1.1 (***Guarded transactions***) We may develop a reliable information system by using a language that allows developers to describe constraints and updates at the modelling level; the language semantics should allow its compiler to rewrite updates—for the purpose of maintaining the integrity—in a compositional manner; more generally, a list of rewriting rules should be available as to how we may alter specifications of operations when desired. □

We will now proceed to build the second part of our thesis hypothesis.

2.2 Workflows

According to the *Oxford English Dictionary*, a business workflow is a sequence of processes through which a piece of work passes from initiation to completion. It will often be a recurring pattern of activity addressing a specific customer or business requirement [41]. Researchers working in the areas of databases incorporate transaction support into the workflow management system [42, 43, 44]. Individual activities are treated as atomic transactions. Their specified control-flow and data-flow dependencies are exploited to support their communication and to query the state of their containing workflows; failures are handled by means of rollback, compensation,

or exception handling. In this thesis, however, we are interested in exploring the specification and analysis—rather than the implementation—of workflows.

Modelling Notations

Existing workflow languages assume the provision of a set of interfaces of available services (e.g. [45]). Workflow languages may be *design-oriented*—serving as a visual notation aimed at the communication of design intentions, rather than at a concrete implementation. Examples include graphics-based BPMN [46], UML activity diagrams [47], and pattern-based YAWL [48]. They may also be *execution-oriented*—designed as an extended form of imperative programming languages with constructs specific to the domain of business process modelling. Examples include WSCI [49], BPEL [50], XPDL [51], WWF [52], and WebWorkFlow [53] that support familiar features of object orientation. However, the design of existing workflow languages does not reflect any anticipation that the success or failure of a task—or a sequence of tasks—can be determined in advance; neither is there the concept of "blocking" tasks. In this respect, workflow languages adopt a similar perspective to that of imperative programming: where there are preconditions, their role is to indicate whether success can be guaranteed, not to act as barriers to further execution.

Formal Semantics

There has been considerable interest in the formalisation of these languages, usually by means of translation into an existing formal, behavioural notation: for example, from BPMN to CSP [54, 55], to Petri Nets [56]; from BPEL to Petri Nets [57, 58], to event calculus [59], to timed automata [60], to pattern-based constructs [61, 62], or to the axiomatic system of Hoare-logic [63]; or from workflow patterns to Petri Nets [64, 65, 66]. This allows the formal analysis of descriptions written in these languages, checking properties such as *reachability* of states, *safety* of execution traces, or *boundedness* of transition sequences. To date, this work has focussed upon the correctness of implementations with respect to specifications of properties of individual

workflows. The correctness of a workflow in the context of global constraints upon the information system has not been considered.

More closely related is the work of [67], where a process notation is proposed that combines both abstract synchronisation events and imperative programming tasks. The notation has been used successfully in the analysis of various algorithms, and has strong tool support [68]. However, the work has focused upon the verification of abstract algorithms. Moreover, although tasks are modelled as transactions, there is no notion of their applicability or availability. Also closely related is the work on combining state- and event-based modelling notations, such as $CSP \parallel B$ [69] and *Circus* [70]. However, the orientation of the specifications is not quite what we aim for, and issues of successful completion and systematic calculation of workflow (or process) preconditions have yet to be explored in either formalism.

The second part of our thesis hypothesis addresses the above limitations of existing works by focusing upon a restricted environment of execution—updates, through either operations or workflows, are applied to a single, sequential data component. Consequently, the effects of updates—initiated by distributed clients—are interleaved on the single, shared state. Let us first consider how the proposed solution should be applied to a hotel reservation system.

Example (*Specifying & Guarding Workflows*) There will be—as we have learnt from Scenario 1.1.2—multiple instances of workflows that are initiated to perform updates on a shared state—that of the same hotel reservation system. If a simple workflow *Booking* is to first *reserve* a room and then to actually *allocate* a room, then the effect of *Booking* should be derived on the basis of the effect of *reserve*, of the effect of *allocate*, and of the semantics of "then" as to how it combines the effects of its two operands. Such property of *compositionality* will facilitate our calculation of a precondition of completion when, for example, multiple instances of *Booking* are considered to be run in parallel. The resulting guarding constraint should be that the hotel in question, in its current state, is able to accommodate the need of all instances

of *Booking*; otherwise, all of them will be blocked. □

We now state the second part of our thesis hypothesis:

Thesis Hypothesis 2.2.1 (***Guarded workflows***) We may develop a reliable information system through predicting the model behaviour under intertwining updates from distributed clients. More precisely, given a set of workflows to be run in parallel, we calculate a *precondition* for their successful completion. We derive the effects of workflows on the basis of the effects of individual activities—characterised as state transformations—and of the semantics of the workflow combinators as to how effects of activities should be combined. □

We will now proceed to build the third part of our thesis hypothesis.

2.3 Model-Based Testing

In the context of model-driven development, the system model itself can serve as the basis for test planning and execution. The results of the test can be exploited to determine—instead of the conformance of implementation to its model—if the model is consistent with the developer's understanding of requirements.

In our study of information systems, we wish to employ data-flow testing [13, 71]: white box testing guided by the patterns of data definition and usage. In the conventional setting of imperative code, for each variable v we identify all statements that *define* (i.e. initialise or modify) its value, as well as all statements that refer to its value for either *computational use* (i.e. in part of a side-effect-free value expression) or *predicate use* (i.e. in part of the predicate for a conditional branch or loop). We denote, respectively, locations (i.e. line numbers) of these three sets of statements—with respect to a variable v—as *def(v)*, *p-use(v)*, and *c-use(v)*. If a procedure calls another as its subroutine, then we need to follow through the (non-circular) dependency tree and keep track of the usage patterns of variables. In the context of model-driven

development, code execution will be replaced by the evaluation of complex predicates, variables by class attributes, and the use of line numbers to identify relevant statements by *preconditions*—satisfying these constraints would guarantee that the operation will demonstrate the data usage pattern in question.

Existing approaches to testing object-oriented code written in languages such as Eiffel [72] and C++ [73], or to testing web services ([74, 75]), treat the contractual specification as the test oracle. Invariants and method pre-conditions are used to derive valid input data, whereas whether the tests pass or fail is determined by invariants and method post-conditions. The goal, however, is to expose errors in the implementation rather than the model. Moreover, the user-specified contracts are not guaranteed to be complete, and the semantics of compound methods—those that reference the specifications of others—is not compositional.

There are approaches to automatically generating states from object-oriented classes (e.g. [76]), but their analysis is upon code rather than the specification model. More relevant approaches such as [77] target the generation of test data from functional and state-based specification. However, their model interface is not object-oriented and hence may not be suitable for our problem domain of information systems. Moreover, they lack the notion of invariant properties and their maintenance upon completion of operations, and hence the test cases that they generate cannot be used to validate the design as in our approach.

Many research efforts have been devoted to testing concurrent software (e.g. [78]), and more recent work [79] is reported on compiling shared-variable programs composed of these routines into a formal language of patterns for the purpose of model checking. However, their focus is on the low-level routines and race conditions. Of more relevance is the work on model-based testing based upon the use of an abstract model to drive the generation of test cases; for example, UML activity [80] or state [81, 82] diagrams, BPEL web service language [83], dependency graph [84], and data-flow-oriented specification language [85]. These modelling languages are also suitable for specifying workflows. However, their behavioural model descriptions are

not as formal as mathematical predicates, and are thus unable to calculate the precise intents of workflows that reference the model operations, let alone to calculate the precondition as we wish to do.

Testing the model of design through use-case scenarios—e.g. data flows, operation patterns, and events that commute as we suggested in Scenario 1.1.3—can help developers reveal any inconsistencies. As such an approach is not found in existing works, we state the third part of our thesis hypothesis accordingly. Let us first consider how the proposed solution should be applied to a hotel reservation system.

Example (*Model Consistency*) Developers have specified a number of workflows on the hotel reservation system and are interested in exploring how they might interfere with each other. The obvious brute-force approach is to explore all possible traces of the parallel composition of these workflows; however, the well-known challenge of combinatorial explosion will soon arise either as the number of workflows to be analysed increases, or as the extent of non-determinism or parallelism within individual workflows increases. Fortunately, as we have learnt from Scenario 1.1.3, developers may prefer to test the model through use cases rather than contemplating the model in its entirety. □

We now state the third part of our thesis hypothesis:

Thesis Hypothesis 2.3.1 (***Testing model consistency***) We may develop a reliable information system through validating the model consistency—by means of testing the model of design—against the intended requirements. □

We will now proceed to build the fourth, final part of our thesis hypothesis.

2.4 Model-Driven Development

Much of the work currently described as *model-based development* is based upon the description of functionality in terms of the target programming language—operations

are written in an imperative language such as Java or C#. Within the domain of information systems, object-oriented methodologies [86, 4, 87] and the specific approach of *design by contract* (DbC) [88] are a common basis for specification, implementation, and verification. Research in this area has involved modelling extensions to the Java and C# object-oriented programming languages, such as *Java Modelling Language* (JML) [10] and Spec# [11], respectively. With this approach, the emphasis, and hence the challenge, shifts from the generation of appropriate update code to the verification of the proposed implementation. However, the task of establishing that an arbitrary programming statement in a language such as Java or C# meets a particular specification is intractable, but considerable progress may be made by requiring that the programs follow particular patterns or templates, and that the programmer insert appropriate annotations to guide the process of verification.

One approach to achieving more reliable, more correct results in software development is through increasing levels of abstraction in programming languages. With each increase in level, the amount of program code to be understood by the human is reduced, and the amount of routine, error-prone implementation work to be done is also reduced. Mainstream languages have evolved from machine code, to assembler, to imperative languages such as C and Java, and finally to domain-specific languages such as SQL and MATLAB. However, further progress has been harder to achieve: existing "fifth generation" languages, such as Prolog, that solve problems using constraints, do not guarantee the automatic production of working code (or the equivalent) from a correct program—human insight is still required.

Model-driven development [89, 90] addresses this problem by combining the domain-specific approach of the fourth generation with the more abstract, constraint-oriented approach that characterises the fifth generation. The task of identifying an appropriate algorithm to satisfy some abstract constraint can be greatly simplified when it is to be performed within a specific domain. The extra information needed to transform the abstract model into a concrete working system comes from domain knowledge or assumptions, expressed through the application of specific transformation rules.

Model-Driven Architecture

The *Model-Driven Architecture* (MDA) [1, 91, 92] is a (trademarked) approach to model-driven development closely associated with the notations and tools of object modelling, as promoted by the *Object Management Group* (OMG). It is intended as an environment where business requirements and the chosen implementation platform are digested and unambiguously represented as, respectively, a *platform-independent model* (PIM) and a *platform-specific model* (PSM); a process of *model transformation* is then applied to a source PIM, amending it with details specifying how that specific platform should be used, turning it into a target PSM. The PSM is itself ready for a second automatic transformation for turning it into a working system. The term *architecture* in MDA refers to a set of other various OMG standards that can be used together in model-driven development: for the definition [93], serialisation [94], and interchange [95] of various *metamodels* of domain specific languages, specification of visual *models* of a system [4], expression of logical constraints [8], and finally the querying, projection (into views), and transformation of *models* [96, 97].

In parallel with OMG's standardisation of MDA and its technology foundation, the *Eclipse Modelling Framework* (EMF) [98] has been developed as the implementation framework for MDD under the open source Eclipse development and programming environment. The OMG's meta-meta-model for language definition—the Meta Object Facility (MOF)—has its equivalent here in *Ecore*, a core subset of the MOF API that targets generating efficient Java implementations. There are various tools developed on top of EMF which specialise in operating upon Ecore models, in particular, for model management [99, 100] and transformation [101]. However, research in this area is still limited to the provision of a modelling environment—dedicating the necessary efforts to defining and proving a correct process of model transformation is still at the developers' discretion.

Model Transformation and Interpretation

Some works focus upon transformation or elaboration of models independently of the execution. For a chosen problem domain, e.g. electronic marketplace [102], a common model that already contains details of the most common features is defined for all potential applications to elaborate upon. Alternatively, an abstract solution model that is not oriented towards any particular implementation platform is transformed into another that is more amenable to either automatic or manual code generation—e.g. from an analysis model to one with concrete software architectures [103], from a customised design model for web services to one that fits a standard profile [104]. It is also explored in terms of adding constraints to a solution model to facilitate a later phase of code generation, e.g. imposing OCL constraints upon UML classes [105, 106].

Other works effectively define a virtual machine for some executable model, in particular, the executable UML [107, 108]. Well-understood action or process languages are assigned into various diagrammatic UML constructs: activity diagrams specifying workflows (e.g. [109, 110]), state diagrams describing possible transitions between system states in response to external events (e.g. [111, 112, 113, 114]), collaboration diagrams representing abstract views of interactions among runtime objects (e.g. [115]), sequence diagrams detailing specific orders of object interactions (e.g. [116]), and even the treatment of a mix of these diagrams (e.g. [117, 118]).

However, without a compositional, behavioural semantics, it is impossible to calculate the resulting behaviour for systems of large size or complexity (and the diagrams themselves become hard to read). Also, these works do not address the issues of how system functionalities may, in general, be composed to define new ones, or how other constraints in the specification—such as database consistency constraints—may be properly maintained.

Of more relevance is the approach that transforms models into their implementing systems; this category of works is the closest to Fowler's original notion [1] of model-driven architecture: *MDA uses modeling languages as programming languages rather*

than merely as design languages. The process of model transformation is implemented using a collection of language-specific or domain-specific constructs. There are works on implementing UML associations in Java (e.g. [119, 120]), but without considering how compound methods may be specified and implemented—they have no notion of composition. In the specific domain of web services, there are works on translating from some abstract service component model into its implementation as imperative service requests: from OMG's diagrammatic *Business Process Modelling Notation* (BPMN) into OASIS' *Business Process Execution Language* (BPEL) (e.g. [121, 122]) or WfMC's XML *Process Definition Language* (XPDL) (e.g. [123]), from W3C's *Ontology Web Language-Service* (OWL-S) into BPEL (e.g. [124]), or from a specific UML profile, customised for modelling service components, into BPEL (e.g. [125, 126]). However, these works have consideration of neither integrity nor composition.

The work on *Object-Relational Mapping* (ORM) [127] also focuses on programming with models. *Enterprise Java Beans* [128] and Hibernate [129] are two example tools for supporting the *Java Data Objects* (JDO) API [130], a standard interface-based Java model abstraction of persistence. Applications are written in Java classes and mappings are specified in some custom structure, such as an XML configuration file, which to a degree decouples the database access from the application code; however, the developer is still required to consider the lower-level view of tables in specifying the mapping. The *ADO.NET Entity Framework* [131] is an EDM-to-Relational mapping platform, where EDM stands for the *Entity Data Model*, an extended entity-relationship model, from which object services like views are provided. The Entity Framework claims to be capable of handling the scenario where multiple applications request varied views of the same data. However, like the other ORM tools, there is no means of specifying semantic integrity constraints as part of the original model, and the updates are described directly as imperative Java or C# methods, making the correct maintenance of business rules a difficult, manual task.

A more formal approach to object-relational mapping has been adopted by Mammar [132, 133, 134]. Her research focusses upon the generation of relational databases

from precise UML models, in which constraints and intended functionality are described by annotating diagrams with fragments of the Abstract Machine Notation (AMN) of the B method. Semantic integrity is addressed through the calculation of preconditions for the maintenance of structural and user-supplied invariants. However, the satisfaction of these preconditions may not be enough to guarantee integrity following the update operation, and a proof must be performed (using the B prover) to determine whether or not this is the case. This lack of automation is connected to the requirement that the modeller should (manually) specify appropriate guards for calls of elementary methods—those that are predefined to modify associations—within the definition of a transaction. Also, Mammar's work is limited to a single level of composition: transactions may be defined in terms of elementary methods, but not in terms of other transactions. Furthermore, the only means of composition allowed is parallel composition; elementary methods may not be composed sequentially in the definition of a transaction.

To state the final part of our thesis hypothesis, we describe how we expect the problem should be solved on a hotel reservation system.

Example (*MDD of RDB*) As we have learnt from Scenario 1.1.4, manually implementing operations on objects as queries on tables, as well as imposing a caching strategy to avoid unnecessary re-evaluations of paths, are error prone. In particular, the process of implementing all possible patterns of updating bi-directional associations is one that should be pre-defined in the transformation engine and proved correct once and for all. That is, what we will aim for is not just a correct and efficient implementation of the hotel reservation system, but a transformation engine that turns any valid input system—which may well be an evolved version of the same system—into its correct and efficient implementation. □

However, we have not found any existing works that address these matters. Consequently, we state the final part of our thesis hypothesis accordingly as follows:

Thesis Hypothesis 2.4.1 (***Automatic database implementation***) We may develop a reliable information system through automating the process of its implementation, in a SQL relational database, from its design model; the guarantee should be that the generated queries are both correct and efficient. □

We have now built all four parts of our thesis hypothesis. As the thesis proposes to extend the range of the BOOSTER notation, in the next section we will review its relevant aspects that we will use in later result chapters.

2.5 The BOOSTER Notation

BOOSTER [3] is a formal, object-based modelling notation intended for the model-driven development of information systems. Its characteristic features are

object orientation: we write object models—classes, attributes, associations, and methods—to specify the intended structures and behaviours.

logical constraints: each operation (or method) is specified entirely in terms of predicative, two-state constraints upon the values of attributes or associations before and after it is executed; business rules are described as system- and class-level invariants that define the (semantic) integrity of data.

compound method: a method may reference names of other methods as part of its definition, effectively re-using their *complete* specifications.

automatic elaboration/completion: each user-supplied, partial specification is extended automatically to reflect assumptions about users' intentions, to guarantee the maintenance of integrity, and to address the possibility of aliasing.

guarded transactions: the intended effect of each method is implemented as an atomic, isolated transaction; its precondition is implemented as a *guard*—the method is available, i.e. not blocked, only when its precondition is satisfied. That is, it is not possible to call a method when its effect would be undefined.

A compiler for the language maps the predicative methods into statements in a formal, programming notation that extends the substitutions language of B [18]. In

the next section we will discuss how to describe—using the BOOSTER notation—the intended structural and behaviour features of a hotel reservation system.

2.5.1 The Hotel Reservation System

We describe the language of BOOSTER by presenting how it is used to specify our hotel reservation system; its complete grammar is included in Appendix A.1; and the complete hotel reservation system specified in BOOSTER is listed in Appendix A.2. Furthermore, each method in Appendix A.2 is annotated with comments—each of which indicates which assignment pattern—as listed in Tables E.1 to E.4 —the corresponding statement belongs to. This is to relate the behavioural specification in Appendix A.2 to our model transformation engine (as already noted in Section 1.1.4 and will be discussed in Chapter 6).

Data Model: a BOOSTER system consists of a list of classes.

```
system HotelReservation { ... }
```

The hotel reservation system model has been designed to include all possible declarations of bi-directional associations[2] (also as visualised in the standard UML class diagrams [4] as shown in Figure 2.1):

– *One-to-One association:* a hotel can legally exist if and only if a licence is issued for it.

```
class Hotel {
  attributes
    permit : Licence . licensee
}
```

```
class Licence {
  attributes
    licensee : Hotel . permit
}
```

– *One-to-Optional association:* a traveller may or may not have a bank account, but each existing account must have an associated owner.

```
class Traveller {
  attributes
    account : [ Account . owner ]
}
```

```
class Account {
  attributes
    owner : Traveller . account
}
```

[2]The most important entity type is that of bi-directional associations, where we declare a mirror attribute that represents the role of the current class. For example, for the *one*-to-*one* association that exists between classes `Hotel` and `Licence`, class `Hotel` plays the role of the licensee and class `Licence` plays the role of the permit.

– *One-to-Set association:* a hotel has a set of allocations (i.e. confirmed bookings), and each allocation must have an associated hotel as its host. The `[*]` syntax specifies the cardinality of the set-valued attribute—zero or larger.

```
class Hotel {                                    class Allocation {
  attributes                                       attributes
    allocations : set(Allocation . host) [*]         host : Hotel . allocations
}                                                }
```

– *One-to-Sequence association:* a hotel has its collection of reservations (i.e. unconfirmed bookings) in order, and each reservation must have an associated hotel as its host.

```
class Hotel {                                    class Reservation {
  attributes                                       attributes
    reservations : seq(Reservation.host) [*]         host : Hotel.reservations
}                                                }
```

– *Optional-to-Optional association:* a member of staff may be the mentee of another staff, and this suggests that he or she may be the mentor of some other staff. This example also demonstrates a class that associates with itself.

```
class Staff {
  attributes
    mentor : [ Staff . mentee ]
}
```

– *Optional-to-Set association:* a confirmed booking may already have allocated a room, and a room may be assigned to a set of allocations.

```
class Allocation {                  class Room {
  attributes                          attributes
    room : [ Room.allocations ]         allocations : set(Allocation.room)[*]
}                                   }
```

– *Optional-to-Sequence association:* an unconfirmed booking may specify a room, and a room may be reserved to non-clashing periods.

```
class Reservation {                 class Room {
  attributes                          attributes
    room: [ Room.reservations ]         reservations : seq(Reservation.room)[*]
}                                   }
```

– *Set-to-Set association:* a hotel has a set of registered clients that travel frequently, and each such traveller has a set of hotels that they register.

```
class Hotel {
  attributes
    registered: set(Traveller.reglist)[*]
}
```

```
class Traveller {
  attributes
    reglist: set(Hotel.registered)[*]
}
```

– *Set-to-Sequence association:* a hotel has a sequence of hiring consultants (where the order represents their priorities), and each staff being a consultant has a set of client hotels to which they provide their consulting service.

```
class Hotel {
  attributes
    consultants: seq(Staff.clients)[*]
}
```

```
class Staff {
  attributes
    clients: set(Hotel.consultants)[*]
}
```

– *Sequence-to-Sequence association:* a hotel has a sequence of employees, arranged by their starting dates, and each employee has a sequence of hotels that they have worked or transferred between.

```
class Hotel {
  attributes
    employees: seq(Staff.employers)[*]
}
```

```
class Staff {
  attributes
    employers: seq(Hotel.employees)[*]
}
```

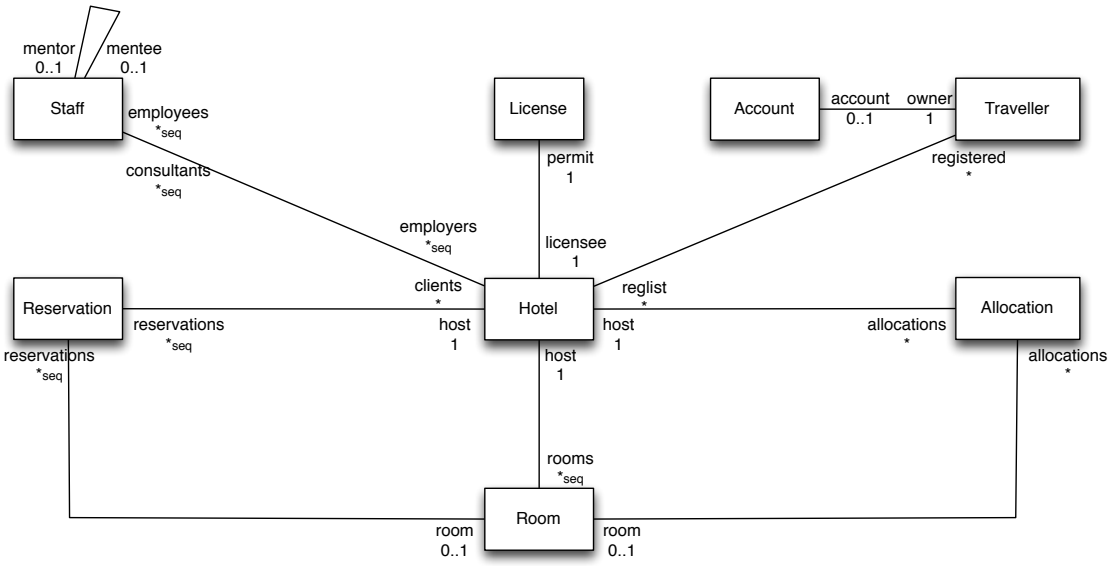


Figure 2.1: Hotel Reservation System

Behavioural Model: each BOOSTER method is specified as a first-order, two-state predicate. For example, in the context of class `Hotel`, method `deallocate` deletes an existing allocation.

```
deallocate { a? : allocations => (a? /: allocations' &
                                  a? /: a?.room.allocations') }
```

The expression `a?` : allocations imposes the constraint that input `a?` is an existing confirmed booking. Each operand of `&` in the consequence specifies that input `a?` is not included in the target set in its post state; this will be implemented as a deletion of `a?` from the target set. In the next section we will discuss how to complete—automatically—such partial, user-specified descriptions with respect to the structural and semantic integrity constraints that are declared in the same system.

2.5.2 Model Completion

We consider user-specified method predicates as *incomplete* because it is possible to overlook model constraints that should have been incorporated in the specification. To derive the specification of a complete transaction upon the system, the compiler should 1) produce additional updates that are necessary for instances of bi-directional associations to remain consistent; and 2) calculate a guard, strong enough to ensure the maintenance of all integrity constraints and user-supplied invariants. A compiler has been designed perform both tasks [135, 136].

We introduce a language that is based upon the *Guarded Substitution Language* (GSL) for the B Toolkit [18] for the abstract implementation of BOOSTER predicates. We extend the GSL notion of variables to *attribute paths*—chains of attribute names. Moreover, we include an implementation-level construct of bounded looping $! \ x \ : \ E \ . \ S$. Our extended GSL notation is also similar to Dijkstra's *Guarded Command Langauge* (GCL) [38], but with two specification-level constructs that are inherited from GSL: parallel substitution $S \parallel T$ and unbounded choice $@ \ x \ : \ E \ . \ S$. Here is

the abstract syntax for our extended GSL notation:

$$
\begin{array}{lcll}
\mathit{Substitution} & ::= & \mathit{skip} & \\
& | & \_ := \_ & \langle\langle \mathit{PATH} \times \mathit{Expression} \rangle\rangle \\
& | & \_ \longrightarrow \_ & \langle\langle \mathit{Predicate} \times \mathit{Substitution} \rangle\rangle \\
& | & \_ \parallel \_ & \langle\langle \mathit{Substitution} \times \mathit{Substitution} \rangle\rangle \\
& | & \_ \,;\, \_ & \langle\langle \mathit{Substitution} \times \mathit{Substitution} \rangle\rangle \\
& | & \_ \,\Box\, \_ & \langle\langle \mathit{Substitution} \times \mathit{Substitution} \rangle\rangle \\
& | & !\_ : \_ . \_ & \langle\langle \mathit{NVariable} \times \mathit{Expression} \times \mathit{Substitution} \rangle\rangle \\
& | & @\_ : \_ . \_ & \langle\langle \mathit{NVariable} \times \mathit{Expression} \times \mathit{Substitution} \rangle\rangle
\end{array}
$$

We support programming constructs of immediate termination (*skip*), assignment (:=), guarded substitution ($\longrightarrow$), bounded ($\Box$) and unbounded (@) choices, parallel ($\parallel$) and sequential (;) compositions, and the generalised parallel composition (!).

Completed Model Operations: completed model operations are written in our extended GSL notation. We consider four operations in the context of the `Hotel` class: `reserve`, `cancel`, `allocate`, and `deallocate`. Inputs or outputs to be manipulated in a method will be declared (via set membership) in its guard—the compiler will derive their types if not already specified. We also assume that two auxiliary sequence functions `ins` and `del` exist to, respectively, insert and delete an item at a certain position in some sequence.

Given as inputs a room and a proposed period of stay—and provided the current number of confirmed bookings does not exceed a specific limit—a hotel may create an unconfirmed booking:

```
reserve {
     r! : extent(Reservation) & dates? : set(Date) & m? : extent(Room)
  &  #allocations < 100
  ==>
     r!.dates        := r!.dates \/ dates?
  || r!.status       := "unconfirmed"
     -- insert into one-to-seq
  || r!.host         := this
  || reservations    := ins(reservations, #reservations + 1, r!)
     -- insert into opt-to-seq
  || r!.room         := m?
  || m?.reservations := ins(m?.reservations, #m?.reservations + 1, r!)
}
```

The expression `m? : extent(Room)` specifies the type of the input: a reference drawn

from the extent of class `Room`. The reservation object to be output is initialised for its date, status, host, and room. The last four parallel updates are grouped in such a way that each pair of them properly updates a bi-directional association. For example, the parallel updates

```
    r!.host      := this
 || reservations := ins(reservations, #reservations + 1, r!)
```

consistently modify the two sides of the association between class `Room` and `Reservation`. Similarly, method `deallocate` as presented in its predicative form in Section 2.5.1 is now completed with its updates on the two associations.

```
deallocate {
     a? : extent(Allocation) & a? : allocations
  ==>
     -- remove from one-to-set
     allocations         := allocations - {a?}
     -- remove from opt-to-set
  || a?.room             := null
  || a?.room.allocations := a?.room.allocations - {a?}
}
```

2.6 Conclusion

By identifying the lack of existing works dealing with challenges that arise in the development context (Figure 1.1) of our hotel reservation system—a representative information system—we have stated the four parts of our thesis hypothesis as to how we may build a reliable information system in Sections 2.1 to 2.4. We elaborated on our definition of information systems in Section 2.1 and stated our assumption in regard to the restricted environment of execution—single, sequential data component—in Section 2.2. We also introduced the BOOSTER notation in Section 2.5—by considering its specification for the hotel reservation system—that we will adopt in later chapters to explore the specification, analysis, and implementation of information systems. In the subsequent result chapters, we will, respectively, test and verify the four parts of our thesis hypothesis—on the basis of the hotel reservation system.

Chapter 3

A Theory of BOOSTER Predicates

In this chapter we will test and verify the hypothesis 2.1.1 (on page 21) regarding the expression of properties or constraints.

3.1 Introduction

As envisaged in Figure 1.1, in Section 2.5 we described the intended behavioural functionality of the hotel reservation system—a representative information system—using BOOSTER methods—our language of transactions. Moreover, we adopted our extended notion of substitution as the abstract implementation language for these predicative methods. However, as requirements evolve, or are not properly reflected, these behavioural specifications will often be in need of revision. One way to reveal the latter case is what we will present in Chapter 5: a model-based approach to testing—on use-case scenarios that the developers are concerned with—the consistency of design models. On this account, if the developers discover inconsistencies—through such a test on the model—and wish to revise their BOOSTER predicate methods, it would be crucial to ensure that each derivation step in the rewriting process is formally supported.

What we will ultimately deliver in this chapter is Section 3.5.2, also summarised in Table 3.1 (on page 62): a list of equivalence and refinement laws on BOOSTER predicates, each of which involves two expressions of BOOSTER predicates. In the case of an *equivalence* law, its LHS may be replaced by its RHS, and vice versa; in

the case of a *refinement* law, its LHS may be replaced by its RHS, but not vice versa. Since both relations of equivalence ($\equiv$) and refinement ($\sqsubseteq$) are *transitive*, this list of laws may be exploited in two flavours. One is for developers to apply—manually—a sequence of laws at their discretion in order to arrive at a resulting specification that is guaranteed to be either an equivalence or a refinement of the original. The other is for developers to supply a predicate that may not appear as so related to the original and to validate—automatically—if there exists a sequence of derivation steps between the two predicate expressions.

To pave the way for the results in Table 3.1, we structure the rest of this chapter as illustrated in Figure 3.1.

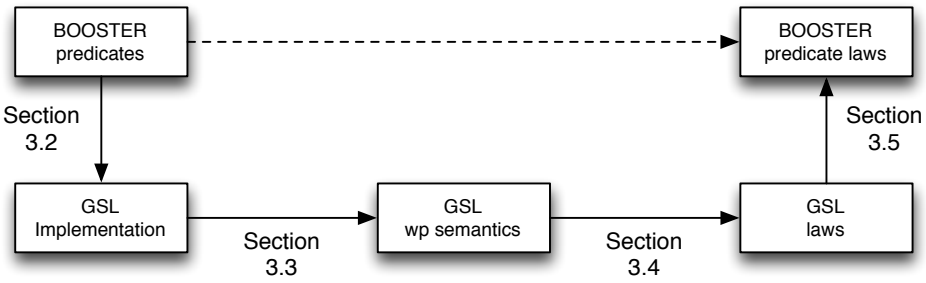


Figure 3.1: Section Outline—Deriving Laws of BOOSTER Predicates

First, we present a set of axiomatic properties (Section 3.2) that characterise our expectation of the BOOSTER compiler being *compositional* in completing—through the generation of extra guarding constraints and updates—partial, user-defined BOOSTER methods. Our extended GSL notation (Section 2.5.2) is adopted as the abstract implementation language for the completed BOOSTER predicates. Then, we give a weakest precondition (*wp*) semantics—which reflects the blocking policy that we wish to adopt at runtime—to this abstract implementation language (Section 3.3). On the basis of our *wp* semantics, we derive a list of equivalence and refinement laws for our extended notion of substitutions (Section 3.4). On the basis of the axiomatic properties of its implementation pattern and the laws on its abstract implementation language, we derive a list of equivalence and refinement laws on BOOSTER predicates (Section 3.5).

We included the majority of our proofs for laws and theorems in Appendix B. Finally, we demonstrate the use of BOOSTER predicate laws as listed in Table 3.1 on our hotel reservation system (Section 3.6).

3.2 Properties of Model Completion

Any compiler that implements the automatic completion of model operations—within the context of a given BOOSTER model with declared associations (and the structural constraints they impose) and semantic invariants—should satisfy a set of axiomatic properties. In particular, we are interested in properties regarding the generation of both guarding constraints and extra updates in order for each method to maintain—upon the effect of its transactional update—the relevant integrity constraints. From these axiomatic properties and laws of the abstract implementation language (Section 3.4), we may derive a set of algebraic laws, each stating that we may substitute one (BOOSTER) predicate expression by another, with the assurance that the resulting predicate behaves at least as a formal refinement of the original one, if not an equivalent of it.

We state axiomatic properties of two semantic functions $\mathcal{G}$ and $\mathcal{P}$ that operate upon the syntax of BOOSTER methods, with respect to the declared associations and invariants in context.

3.2.1 Calculating Guards

The first semantic function $\mathcal{G} : \mathit{Predicate} \to \mathit{Predicate}$ calculates for a given method predicate its guarding constraint. Function $\mathcal{G}$ is consistent with its more concrete counterpart *wp* (Section 3.3) that operates upon substitution programs: $\mathcal{G}$ ought to calculate a guarding constraint that is no weaker than the standard notion of *wp* [38, 18].

In this section we are interested in studying a list of axiomatic properties of $\mathcal{G}$ that operate at the more abstract level of mathematical predicates.

We write $before\,(p)$ to denote the "unprimed" version of a method predicate p, which transforms each post-state reference of a variable to its pre-state reference. We shall use an important property of *before* to state side conditions for later laws of predicate equivalence and refinement to hold.

Property 3.2.1 (***Before-state Predicates***)

$$\forall\, p : Predicate \mid p = before\,(p) \bullet \\ \mathcal{G}\,[\![\, p \,]\!] = p \wedge \mathcal{P}\,[\![\, p \,]\!] = p \longrightarrow skip$$

□

To achieve a negated predicate $\neg\ p$, we require that the operand predicate p does not refer to the post-state value of any attribute, i.e. no "primed" reference, for otherwise the resulting program is likely to be useless. For example, the implementing program to achieve $\neg\ (x' = x + 1)$ would have a guard $before\,(x' = x + 1) = x = x + 1 = \mathbf{false}$, which means that the program is never applicable. Therefore, we guard $\neg\ p$ by simply requiring that $before\,(\neg\ p)$ is already satisfied in the pre-state.

$$\mathcal{G}\,[\![\, \neg\ p \,]\!] = before\,(\neg\ p)$$

Similarly, we require that *true* and *false* are already satisfied when, respectively, we intend for **true** and **false** to be established.

$$\begin{array}{rcl} \mathcal{G}\,[\![\, \mathbf{true} \,]\!] & = & true \\ \mathcal{G}\,[\![\, \mathbf{false} \,]\!] & = & false \end{array}$$

The guard of $p \vee q$ is the disjunction of the guards for method predicates p and q. We generalise accordingly the guard of $\vee$ to that of $\exists$.

$$\begin{array}{rcl} \mathcal{G}\,[\![\, p \vee q \,]\!] & = & \mathcal{G}\,[\![\, p \,]\!] \vee \mathcal{G}\,[\![\, q \,]\!] \\ \mathcal{G}\,[\![\, \exists x : A \bullet p \,]\!] & = & \exists x : A \bullet \mathcal{G}\,[\![\, p \,]\!] \end{array}$$

The result of $\mathcal{G}\,[\![\, p \wedge q \,]\!]$ is equal to the conjunction of $\mathcal{G}\,[\![\, p \,]\!]$ and $\mathcal{G}\,[\![\, q \,]\!]$ when aliasing is not possible and hence triggers no extra guarding constraint; we formalise this side condition as $frame\,(p) \cap frame\,(q) = \emptyset$, where $frame\,(p)$ denotes the set of attributes that the implementation of predicate p may modify. We will later

need $read\,(p)$, where $p = before\,(p)$, to denote the set of attributes upon which the evaluation of p relies. We generalise accordingly the guard of $\wedge$ to that of $\forall$.

$$\begin{array}{rcll} \mathcal{G}\,[\![\, p \wedge q \,]\!] & = & \mathcal{G}\,[\![\, p \,]\!] \wedge \mathcal{G}\,[\![\, q \,]\!] & \textbf{if } frame\,(p) \cap frame\,(q) = \emptyset \\ \mathcal{G}\,[\![\, \forall x : A \bullet p \,]\!] & = & \forall x : A \bullet \mathcal{G}\,[\![\, p \,]\!] & \textbf{if } \forall\, x_1, x_2 : A \bullet \\ & & & \qquad frame\,(p[x_1/x]) \cap frame\,(p[x_2/x]) = \emptyset \end{array}$$

The guard of $\mathcal{G}\,[\![\, p \Rightarrow q \,]\!]$ reflects the blocking semantics that we wish to adopt at runtime: it is either that the guard for method predicate p does not hold, in which case we block the corresponding method, or that $\mathcal{G}\,[\![\, p \,]\!]$ holds and we also require that the guard of the method predicate q holds before its effect occurs.

$$\begin{array}{rcl} \mathcal{G}\,[\![\, p \Rightarrow q \,]\!] & = & \mathcal{G}\,[\![\, p \,]\!] \wedge \mathcal{G}\,[\![\, q \,]\!] \\ & & \vee \\ & & \neg\, \mathcal{G}\,[\![\, p \,]\!] \end{array}$$

We assume that the antecedent of $p \Rightarrow q$ contains no intent of update, i.e. $p = before\,(p)$. If we also know that $q = before\,(q)$, then we will treat $p \Rightarrow q$ as a normal predicate; otherwise, we will treat it as a guarded program (i.e. $(p \wedge \mathcal{G}\,[\![\, q \,]\!] \longrightarrow \mathcal{P}\,[\![\, q \,]\!]) \,\Box\, (\neg\, p \longrightarrow skip)$).

3.2.2 Producing Programs

The second semantic function $\mathcal{P} : Predicate \rightarrow Substitution$ calculates for a given method predicate its protecting updates, where *Substitution* is the type of a language that we use both to assign a formal semantics to the BOOSTER method language, and to suggest a possible pattern of implementation for the later phase of code generation.

Since $\mathcal{G}\,[\![\, \neg\, p \,]\!]$ ensures that the negation already holds, the method need not do anything (i.e. *skip*) to achieve it.

$$\mathcal{P}\,[\![\, \neg\, p \,]\!] = before\,(\neg\, p) \longrightarrow skip$$

Similarly, we turn **true** and **false** to programs that need do nothing to achieve the intended condition:

$$\begin{array}{rcl} \mathcal{P}\,[\![\, \textbf{true} \,]\!] & = & true \longrightarrow skip \\ \mathcal{P}\,[\![\, \textbf{false} \,]\!] & = & false \longrightarrow skip \end{array}$$

To achieve $p \vee q$, we produce a choice of guarded programs: one that achieves method predicate p via $\mathcal{P}[\![p]\!]$, suitably guarded by $\mathcal{G}[\![p]\!]$; and similarly for the other that achieves method predicate q. We generalise the program production for $\vee$ to that for $\exists$ accordingly.

$$\begin{array}{rcl} \mathcal{P}[\![p \vee q]\!] & = & \mathcal{G}[\![p]\!] \longrightarrow \mathcal{P}[\![p]\!] \\ & & \Box \\ & & \mathcal{G}[\![q]\!] \longrightarrow \mathcal{P}[\![q]\!] \\ \mathcal{P}[\![\exists x : A \bullet p]\!] & = & \exists x : A \bullet \mathcal{G}[\![p]\!] \longrightarrow @\, x \ : \ A \,.\, \mathcal{P}[\![p]\!] \end{array}$$

We produce a parallel composition of two programs, $\mathcal{P}[\![p]\!]$ and $\mathcal{P}[\![q]\!]$, to simultaneously achieve both predicates p and q, and we guard it by the conjunction of their guards. We generalise the program production for $\wedge$ to that for $\forall$ accordingly.

$$\begin{array}{rcll} \mathcal{P}[\![p \wedge q]\!] & = & \mathcal{G}[\![p]\!] \wedge \mathcal{G}[\![q]\!] & \textbf{if } \mathit{frame}\,(p) \cap \mathit{frame}\,(q) = \emptyset \\ & & \longrightarrow \mathcal{P}[\![p]\!] \parallel \mathcal{P}[\![q]\!] & \\ \mathcal{P}[\![\forall x : A \bullet p]\!] & = & \forall x : A \bullet \mathcal{G}[\![p]\!] & \textbf{if } \forall x_1, x_2 : A \bullet \\ & & \longrightarrow !\, x \ : \ A \,.\, \mathcal{P}[\![p]\!] & \quad \mathit{frame}\,(p[x_1/x]) \cap \\ & & & \quad \mathit{frame}\,(p[x_2/x]) = \emptyset \end{array}$$

To achieve $p \Rightarrow q$, we reflect the blocking semantics we adopt through a choice of guarded programs: one that achieves method predicate q via $\mathcal{P}[\![q]\!]$, suitably guarded by $\mathcal{G}[\![q]\!]$ and when the antecedent holds; and the other program that does nothing, i.e. we block the method, when the antecedent does not hold.

$$\begin{array}{l} \mathcal{P}[\![p \Rightarrow q]\!] = \mathcal{G}[\![p]\!] \wedge \mathcal{G}[\![q]\!] \longrightarrow \mathcal{P}[\![q]\!] \\ \qquad\qquad\quad \Box \\ \qquad\qquad\quad \neg\, \mathcal{G}[\![p]\!] \longrightarrow \mathit{skip} \end{array}$$

We may simplify the output of function $\mathcal{P}$ when we know the input predicate contains no intention of updates to the system.

Property 3.2.2 (***Simplifying Output of*** $\mathcal{P}$ ***via*** *before*)
Given BOOSTER predicates p, q such that $p = \mathit{before}\,(p)$ and $q = \mathit{before}\,(q)$ we

obtain:

$$\begin{array}{lcll}
\mathcal{P}\,[\![\, p \vee q \,]\!] & = & p \vee q \longrightarrow \mathit{skip} & \texttt{(before-}\vee\texttt{)} \\
\mathcal{P}\,[\![\, \exists x : A \bullet p \,]\!] & = & \exists\, x : A \bullet p \longrightarrow \mathit{skip} & \texttt{(before-}\exists\texttt{)} \\
\mathcal{P}\,[\![\, p \wedge q \,]\!] & = & p \wedge q \longrightarrow \mathit{skip} & \texttt{(before-}\wedge\texttt{)} \\
\mathcal{P}\,[\![\, \forall x : A \bullet p \,]\!] & = & \forall\, x : A \bullet p \longrightarrow \mathit{skip} & \texttt{(before-}\forall\texttt{)} \\
\mathcal{P}\,[\![\, p \Rightarrow q \,]\!] & = & (p \wedge q \longrightarrow \mathit{skip}) \,\Box\, (\neg\, p \longrightarrow \mathit{skip}) & \texttt{(before-}\Rightarrow\texttt{)}
\end{array}$$

For cases `before-`$\wedge$ and `before-`$\forall$, we require, respectively, that $(\mathit{frame}\,(p) \cap \mathit{frame}\,(q) = \emptyset)$ and $(\forall\, x_1, x_2 : A \bullet \mathit{frame}\,(p[x_1/x]) \cap \mathit{frame}\,(p[x_2/x]) = \emptyset)$. □

Proof of Property 3.2.2 From the definition of $\mathcal{P}$ and manipulation of *wp*. □

We observe that the calculation of $\mathcal{P}$ incorporates that of $\mathcal{G}$. For example, the guard of $\mathcal{P}\,[\![\, \neg\, p \,]\!]$, i.e. $\mathit{before}\,(\neg\, p)$, is equivalent to $\mathcal{G}\,[\![\, \neg\, p \,]\!]$, and the calculation of $\mathcal{P}\,[\![\, p \vee q \,]\!]$ invokes two instances of $\mathcal{G}$. As a result, we would expect that imposing an extra guard $\mathcal{G}\,[\![\, p \,]\!]$ upon the guarded program $\mathcal{P}\,[\![\, p \,]\!]$, for some predicate p, results in an equivalent program.

Property 3.2.3 (***Incorporation of*** $\mathcal{G}$ ***in*** $\mathcal{P}$)

$$\forall\, p \in \mathit{Predicate} \bullet \mathcal{P}\,[\![\, p \,]\!] = (\mathcal{G}\,[\![\, p \,]\!] \longrightarrow \mathcal{P}\,[\![\, p \,]\!])$$

□

We have now seen the implementation pattern of BOOSTER predicates—functions $\mathcal{G}$ and $\mathcal{P}$ output programs that are described in our extended GSL notation (Section 2.5.2). In the next section we will define a semantics on this abstract implementation language.

3.3 A Semantics for Substitutions

The standard notion of Dijkstra's predicate transformer [38]—also known as the weakest precondition (*wp*)—transforms from a goal predicate I to another $\mathit{wp}\,(S,\ I)$, where S is a composition of programming statements. Starting from a state where $\mathit{wp}\,(S,\ I)$ holds, program S is guaranteed to terminate in a state that satisfies I. Our notion of

wp is specific to the extended substitution language (Section 2.5.2) and the BOOSTER predicates (Section 2.5.1):

$$wp : Substitution \times Predicate \rightarrow Predicate$$

We implement the calculated *wp* as *guards*. That is, at any given state, we *block* those operations whose associated constraints of *wp* are not satisfied. Such intended usage of our *wp* is fundamentally different from the standard, which treats those constraints as preconditions and allows updates to proceed even in non-satisfying states—with the expectation that the outcome is undefined. Later in this section we will see how our intended usage of *wp* induces a definition that is distinguished from the standard, notably in cases of assignment ($:=$), guarded substitution ($\longrightarrow$), and choice ($\square$). Moreover, in Section 3.4 we will also see how our interpretation of *wp* as guards affects the notion of refinement between substitution programs accordingly.

In fact, our notion of *wp* returns stronger constraints than the standard [38, 18] in two aspects. First, we impose an aliasing constraint on every parallel composition: for each pair of property updates that appear in a parallel composition $(cp_1.a' = e_1) \wedge (cp_2.a' = e_2)$ if both context paths cp_1 and cp_2 are typed to the same class, then we will need to impose the constraint $(cp_1 = cp_2 \Rightarrow e_1 = e_2)$. As long as cp_1 and cp_2 refer to the same object at runtime (i.e. an instance of aliasing), values of the two source expressions e_1 and e_2 match. Second, we reflect compiler-specific heuristics—maintenance of semantic integrity may be achieved through strengthening the guard of the method. For example, say an invariant property $s \cap t$ is declared. Then, to ensure that a predicate method $s' = s \cup \{x\}$ maintains this invariant, the compiler may generate an extra guard $x \notin t$, instead of producing an extra update $t' = t \setminus \{x\}$. Let us consider the first case of aliasing constraints in detail:

Definition 3.3.1 (***Constraint on Aliasing***) We define a function which generates cases where aliasing may occur:

$$pathsets : (NClass \times NBAttribute) \times Predicate \rightarrow \mathbb{P}(\mathbb{P}\, PATH)$$

Given an attribute a from class A and a constraint C, we write $pathsets\,(A \mapsto a,\ C)$ to denote a set of path sets drawn from constraint C, each of which is a set of context paths that also refer to the same a from class A. The returned value of *pathsets* provides the basis for a thorough case-analysis of aliasing scenarios—the member set $\{\}$ represents the case where the aliasing is absent. We also define a function that infers the exact aliasing constraint for each case generated by *pathsets*:

$$matches : \mathbb{P}\, PATH \rightarrow Predicate$$

Given a set of paths s—where $s \in pathsets\,(A \mapsto a,\ c)$, i.e. all paths in s are context paths to access attribute a—we write $matches\,(s)$ to ensure that if all paths in s refer to the same object, then they also access the same value for attribute a. □

Let us consider an example of imposing the aliasing constraints.

Example (*Aliasing Constraint*) We consider a model in which a `Person` may have at most one `partner`, another instance of the same class. The method `marry` should have the effect of ensuring: that the `partner` of the current object becomes the input object `person?`, and that the `partner` of `person?` becomes the current object.

```
class Person {
  attributes
    partner : [ Person . partner ]
  methods
    marry {
         partner /= null ==> partner.partner := null
      || this.partner := person?
      || person?.partner := this
      || person?.partner /= null ==> person?.partner.partner := null
    }
}
```

The BOOSTER compiler considers the set of context paths for attribute `partner`: $\{partner,\ this,\ person?,\ person?.partner\}$. A pair-wise analysis upon this set of paths—via *pathsets*—results in six cases:

$$\begin{array}{l} pathsets\,(Person \mapsto partner,\ marry) = \\ \quad \{\,\{this,\ person?\},\ \{partner,\ person?.partner\},\ \{partner, this\}, \\ \quad\quad \{partner,\ person?\},\ \{this,\ person?.partner\},\ \{person?,\ person?.partner\}\,\} \end{array}$$

In each case, if the two context paths refer to the same object, then we require

that—via *matches*—the assigned expressions have the same value. We end up having the following guarded method[1].

```
marry {
     this : extent (Person) & person? : extent (Person)
   & (this = person? => person? = this)              -- case 1
   & (partner /= null & person?.partner /= null =>
       partner = person?.partner => null = null)     -- case 2
   & (partner /= null =>
       partner = this => person? = null)             -- case 3
   & (partner /= null =>
       partner = person? => this = null)             -- case 4
   & (person?.partner /= null =>
       this = person?.partner => person? = null)     -- case 5
   & (person?.partner /= null =>
       person? = person?.partner => this = null)     -- case 6
  ==> ...
}
```

□

We are now able to present the definition of our *wp* which incorporates the constraints of aliasing. The rules of *wp* on *skip* and sequential composition (;) are just the same as those in [18, 137]: to establish I through *skip*, i.e. doing nothing, I must already be satisfied; to establish I through $S\ ;\ T$, we must ensure that the post-state of executing S is strong enough for T to establish I.

$$\begin{aligned} wp\,(skip,\ I) &= I \\ wp\,(S\ ;\ T,\ I) &= wp\,(S,\ wp\,(T,\ I)) \end{aligned}$$

In our treatment of assignment $(v := e)$, it is insufficient to perform for each occurrence of variable v within predicate I merely a textual substitution by expression e. In our context, we assume that each variable v is an attribute path that is formed by a target attribute a and its context path cp. We then need to take into account the fact that paths of distinct textual forms may refer to the same object at runtime. Each possible case of aliasing is expressed on a member set x drawn from

[1]We observe that the guard of the above substitution has its second conjunct (i.e. $this = person? \Rightarrow person? = this$) automatically reduced to *true*, the consequence of its third conjunct (i.e. $null = null$) reduced to *true*, and consequences of the remaining conjuncts (i.e. $person? = null$ and $this = null$) reduced to *false*. We would also expect the compiler to simplify the above completed method and yield a more palatable expression.

$pathsets\,(target\,[\![cp.a]\!],\ I)$[2]: we write $(matches\,(x) \Rightarrow I[e/x_1.a, e/x_2.a, \ldots, e/x_n.a])$[3] to express the constraint that if the set of paths under consideration all refer to the same object (i.e. $matches\,(x)$, where $x = \{x_1, x_2, \ldots, x_n\}$), then we expect the version of predicate I—with all occurrences of the attribute paths $x_1.a, x_2.a, \ldots, x_n.a$ substituted by the expression e—to be satisfied.

$$wp\,(cp.a := e,\ I) = \forall\, x : pathsets\,(target\,[\![cp.a]\!],\ I) \bullet \\ matches\,(x) \Rightarrow I[e/x_1.a, e/x_2.a, \ldots, e/x_n.a] \\ \textbf{where } x = \{x_1, x_2, \ldots, x_n\}$$

Since we adopt the blocking semantics for guarded substitutions, only when the antecedent p is satisfied will we consider the establishment of I via S.

$$wp\,(p \longrightarrow S,\ I) = p \wedge wp\,(S,\ I)$$

This is different from the standard that returns $p \Rightarrow wp\,(S,\ I)$, which means that executing a program outside its guard will miraculously establish I.

We characterise our definition of wp on $S \parallel T$[4] through the notion of commutativity: regardless of the order of executing S and T, we ought to establish I at the end. We generalise the rule of wp on $\parallel$ accordingly to cover the iterator (i.e. $!\ x\ :\ A\ .\ S$), effectively considering all possible execution orders of the set $\{S[y/x] \mid y \in A\}$ of substitution programs.

$$\begin{aligned} wp\,(S \parallel T,\ I) &= wp\,(S,\ wp\,(T,\ I)) \wedge wp\,(T,\ wp\,(S,\ I)) \\ wp\,(!\ x\ :\ A\ .\ S,\ I) &= \forall x : A \bullet \left(\begin{array}{l} wp\,(S,\ wp\,(S',\ I)) \\ \wedge \\ wp\,(S',\ wp\,(S,\ I)) \end{array} \right) \end{aligned}$$

$$\textbf{where } S' = !\ y\ :\ A \setminus \{x\}\ .\ S[y/x]$$

We assume that programs S and T do not contain the parallel composition operator ($\parallel$) in their definitions. When multiple parallel composition operators are present, we

[2]We write $target\,[\![cp.a]\!] \in NClass \times NBAttribute$ to denote the attribute (and its class type) that path $cp.a$ refers to.

[3]We write $I[e/x_1.a, e/x_2.a, \ldots, e/x_n.a] \in Predicate$ to denote the version of predicate I with all occurrences of paths $x_i.a$, where $1 \leq i \leq n$, substituted by expression e.

[4]There was no rule on parallel composition in the standard definition of wp, as it is not part of Dijkstra's language of guarded commands.

consider all interleaving orders accordingly[5]:

$$wp\,(S_1 \parallel S_2 \parallel \ldots \parallel S_n,\ I) = \bigwedge \{wp\,(t,\ I) \mid t \in \lll_{i \in 1..n} S_i\}$$

Our definition of *wp* on a bounded or unbounded choice states that we do not insist all branches of the choice are available in any given state; instead, we block those whose guards are not satisfied.

$$\begin{array}{lcl} wp\,(S \,\Box\, T,\ I) & = & wp\,(S,\ I) \vee wp\,(T,\ I) \\ wp\,(@\,x\ :\ A\ .\ S,\ I) & = & \exists y : A \bullet wp\,(S[y/x],\ I) \end{array}$$

We will use two standard properties of *wp*—it is monotonic on implication ($\Rightarrow$), and it distributes through conjunction ($\wedge$) in the goal predicate.

Property 3.3.1

$$\begin{array}{lcll} wp\,(S,\ p) & \Rightarrow & wp\,(S,\ q) & (\texttt{mono-wp-}\Rightarrow) \\ wp\,(S,\ p \wedge q) & \Leftrightarrow & wp\,(S,\ p) \wedge wp\,(S,\ q) & (\texttt{dist-wp-}\wedge) \end{array}$$

provided $p \Rightarrow q$ in the case of (`mono-wp-`$\Rightarrow$). □

We conclude this section by stating the following property that holds between *wp* (Section 3.3), $\mathcal{G}$ (Section 3.2.1), and $\mathcal{P}$ (Section 3.2.2).

Property 3.3.2 (***programs*** $\mathcal{P}$***, guards*** $\mathcal{G}$***, and predicate transformer*** *wp*)

$$\forall\, p : Predicate \bullet \mathcal{G}\,[\![\, p \,]\!] \Leftrightarrow wp\,(\mathcal{P}\,[\![\, p \,]\!],\ p)$$

□

Given a method predicate p, we expect $\mathcal{P}$ to first produce updates (i.e. $\mathcal{P}\,[\![\, p \,]\!]$) that both implement the stated intent and maintain the structural integrity, then we shall expect *wp* to generate the suitable guarding constraint (i.e. $wp\,(\mathcal{P}\,[\![\, p \,]\!],\ p)$)—consistent with what $\mathcal{G}$ axiomatically requires (i.e. $\mathcal{G}\,[\![\, p \,]\!]$)—upon which at runtime we decide whether or not to block the method.

To facilitate the proofs of laws on BOOSTER predicates, we will derive a list of laws at the level of their compiled form—induced directly from our notion of *wp*.

[5]This ensures that (`assoc-`$\wedge$) of Law 3.5.3 is the case.

3.4 Reasoning about Substitutions

3.4.1 Refinement

Our notion of refinement relation is similar to [18, 137] in that we define it in terms of our specific notion of *wp* (Section 3.3):

Definition 3.4.1 (***Refinement of Substitution Programs***) A program S is refined by another program T, denoted as $S \sqsubseteq T$, if and only if T is a suitable, safe replacement for S—at any state that we might observe T's execution (i.e. it is not blocked), we must also be able to observe S's execution. Consequently, substituting T for S will guarantee not to result in a system trace that would not have been possible for T.

$$\_ \sqsubseteq \_ : \mathit{Substitution} \leftrightarrow \mathit{Substitution}$$

$$\forall\, I : \mathit{Predicate};\ S, T : \mathit{Substitution} \bullet S \sqsubseteq T \Leftrightarrow \mathit{wp}\,(T,\ I) \Rightarrow \mathit{wp}\,(S,\ I)$$

□

However, the standard definition of program refinement is defined by a reversed relation of implication (i.e. $S \sqsubseteq T \Leftrightarrow \mathit{wp}\,(S,\ I) \Rightarrow \mathit{wp}\,(T,\ I)$). We have to reverse the implication as our specific definition of *wp* reflects the blocking semantics we adopt—particularly the rule on the choice □ combinator that disjoins, rather than conjoins, the recursive calculations of both operands. Consequently, the laws that we prove in later Sections 3.4.2 and 3.5.2 will still carry interpretations that are consistent with the standard.

For proving algebraic laws in later sections, we claim that combinators of our extended substitution language are *monotonic* on the $\sqsubseteq$ relation.

Law 3.4.1 (***Monotonicity of program combinators***)

$$
\begin{array}{lcll}
g \longrightarrow S & \sqsubseteq & g \longrightarrow U & \texttt{(mono-}\longrightarrow\texttt{)} \\
S \ ; \ T & \sqsubseteq & U \ ; \ V & \texttt{(mono-;)} \\
S \parallel T & \sqsubseteq & U \parallel V & \texttt{(mono-}\parallel\texttt{)} \\
S \ \Box \ T & \sqsubseteq & U \ \Box \ V & \texttt{(mono-}\Box\texttt{)} \\
! \, x \ : \ A \ . \ S & \sqsubseteq & ! \, x \ : \ A \ . \ U & \texttt{(mono-!)} \\
@ \, x \ : \ A \ . \ S & \sqsubseteq & @ \, x \ : \ A \ . \ U & \texttt{(mono-@)}
\end{array}
$$

provided $S \sqsubseteq U$ and $T \sqsubseteq V$. $\Box$

We obtain immediately that $\sqsubseteq$ is transitive, for $\Rightarrow$ is. Accordingly, we say two programs are equivalent if they are refined by each other:

Definition 3.4.2 (***Equivalence of Substitution Programs***)

$$
\begin{array}{|l}
\_ = \_ : \mathit{Substitution} \leftrightarrow \mathit{Substitution} \\
\hline
\forall \, S, T : \mathit{Substitution} \bullet S = T \Leftrightarrow (S \sqsubseteq T \wedge T \sqsubseteq S)
\end{array}
$$

$\Box$

3.4.2 Laws

Based upon the weakest precondition semantics of substitutions, we present a list of equivalence and refinement laws on them.

Laws of Equivalence

Some basic laws whose proofs through simple manipulations of *wp* are trivial:

Law 3.4.2 (***Basic***)

$$
\begin{array}{lcll}
(p \longrightarrow S) \ \Box \ (q \longrightarrow S) & = & p \vee q \longrightarrow S & \texttt{(a)} \\
p \longrightarrow (q \longrightarrow S) & = & p \wedge q \longrightarrow S & \texttt{(b)} \\
\mathit{true} \longrightarrow S & = & S & \texttt{(c)}
\end{array}
$$

$\Box$

We have $\longrightarrow$ distribute over $\parallel$ and over $\Box$. We also have $\parallel$ distribute over $\Box$.

Law 3.4.3 (***Distributivity***)

$$\begin{array}{rcl} p \longrightarrow (S \parallel T) & = & (p \longrightarrow S) \parallel (p \longrightarrow T) \\ p \longrightarrow (S \,\Box\, T) & = & (p \longrightarrow S) \,\Box\, (p \longrightarrow T) \\ p \parallel (S \,\Box\, T) & = & (p \parallel S) \,\Box\, (p \parallel T) \end{array}$$

□

A branch of choice with a *false* guard is effectively not available, as it is always *blocked*.

Law 3.4.4 (***Non-contributing branch with*** *false* ***guard in Choice***)

$$S \,\Box\, (\mathit{false} \longrightarrow T) = S$$

□

In a parallel composition, if one of the operand processes does nothing, then the other process is not constrained by it at all.

Law 3.4.5 (***Non-contributing*** *skip* ***in Parallel Composition***)

$$wp\,(S \parallel \mathit{skip},\ I) = wp\,(S,\ I)$$

□

When there are two nested guards, we may eliminate the one that is weaker.

Law 3.4.6 (***Eliminating the Weaker in Nested Guards***)

$$\begin{array}{rcll} p \wedge q \longrightarrow (q \longrightarrow S) & = & p \wedge q \longrightarrow S & \texttt{(a)} \\ p \longrightarrow (p \vee q \longrightarrow S) & = & p \longrightarrow S & \texttt{(b)} \end{array}$$

□

Laws of Refinement

Under our specific notion of refinement, we may refine a substitution by strengthening its guard, resulting in a behaviour represented by a subset of observable traces. However, as we will see in Law 3.4.11, our notion of refinement here is not completely identical to the traces refinement in a process setting [139].

Law 3.4.7 (***Refinement by Strengthening Guard***)

$$\begin{array}{lcll} q \longrightarrow S & \sqsubseteq & p \longrightarrow S & \texttt{(a) prov.} \quad p \Rightarrow q \\ p \vee q \longrightarrow S & \sqsubseteq & p \longrightarrow S & \texttt{(b)} \\ p \longrightarrow S & \sqsubseteq & p \wedge q \longrightarrow S & \texttt{(c)} \end{array}$$

□

Replacing one substitution by a more refined one will result in a refinement.

Law 3.4.8 (***Refinement through Refining the Body Substitution***)

$$p \longrightarrow S \sqsubseteq p \longrightarrow T$$

provided $S \sqsubseteq T$ □

In the context of a guarded substitution, where the guard is a disjunction ($\vee$) of predicates and the substitution a bounded choice ($\Box$), we obtain a refinement by distributing operands of the disjunction to both branches of the choice.

Law 3.4.9 (***Refinement by Distributing Guards over*** $\Box$)

$$(p \vee q) \longrightarrow S \;\Box\; T \sqsubseteq (p \longrightarrow S) \;\Box\; (q \longrightarrow T)$$

□

Symmetrically, in the context of a guarded substitution, where the guard is a conjunction ($\wedge$) of predicates and the substitution a parallel composition ($\|$), we obtain a refinement by distributing operands of the conjunction to both operands of the parallel composition.

Law 3.4.10 (***Refinement by Distributing Guards over*** $\|$)

$$(g \wedge h) \longrightarrow (S \parallel T) \;\sqsubseteq\; (g \longrightarrow S) \parallel (h \longrightarrow T)$$

We are able to strengthen the refinement relation ($\sqsubseteq$) to an equivalence ($=$) if 1) $g = \mathcal{G}[\![\, p \,]\!], S = \mathcal{P}[\![\, p \,]\!]$ and $h = \mathcal{G}[\![\, q \,]\!], T = \mathcal{P}[\![\, q \,]\!]$, for some predicates p and q; and 2) the model invariant in context I is such that $I = p = q$. □

In the above case of equivalence (=) of Law 3.4.10, since $p = q$, we have $S = T$ and $g = h$, and the above is simplified to:

$$g \longrightarrow (S \parallel S) \quad \sqsubseteq \quad (g \longrightarrow S) \parallel (g \longrightarrow S)$$

The RHS is more *refined* than the LHS because it requires that g is satisfied before the second execution of S.

The fact that our substitution combinators should be interpreted as relational operators makes our notion of refinement distinct from that of traces refinement in a process setting [139]. As an example, a choice between the possible interleavings of two programs (i.e. $(S\ ;\ T) \Box (T\ ;\ S)$) is not equivalent to running them in parallel (i.e. $S \parallel T$). The latter has the additional constraint that after one program has finished executing, the other cannot be blocked; consequently, the former is refined by the latter. However, in [139] the former is equivalent to the latter.

Law 3.4.11 (***Refinement of Sequential Composition***)

$$\begin{array}{lll} (S\ ;\ T) \Box (T\ ;\ S) & \sqsubseteq & S\ ;\ T \quad \texttt{(a)} \\ S\ ;\ T & \sqsubseteq & S \parallel T \quad \texttt{(b)} \end{array}$$

By the transitivity of $\sqsubseteq$, we have $(S\ ;\ T) \Box (T\ ;\ S) \sqsubseteq S \parallel T$. $\Box$

We have now completed all the requirements, as illustrated in Figure 3.1, for exploring BOOSTER predicates laws: 1) the properties of the BOOSTER compiler in Section 3.2; 2) the *wp* semantics in Section 3.3; and 3) the laws on the compiled form of BOOSTER predicates in Section 3.4.2. Before our exploration in Section 3.5, we conclude our reasoning about the extended notion of substitutions by presenting a normal form of substitutions.

3.4.3 A Normal Form of Substitutions

Besides using our notion of *wp* to prove a set of laws on program equivalence and refinement, we may also use it to derive a normal form of substitutions, which any future formal analysis can be performed upon, instead of upon the entire syntax.

We define a semantic function $\mathcal{N}\,[\![\_]\!]\_ : \mathit{Substitution} \to \mathit{Predicate} \to \mathit{Substitution}$ that rewrites each substitution as a choice between sequential guarded assignments. Given a program S and context invariant I, $\mathcal{N}\,[\![S]\!]_I$ has the following shape:

$$G \longrightarrow \left(\begin{array}{lllll} g_{11} \longrightarrow x_{11} := e_{11} & ; & g_{12} \longrightarrow x_{12} := e_{12} & ; \quad \ldots \quad ; & g_{1m} \longrightarrow x_{1m} := e_{1m} \\ \Box & & & & \\ g_{21} \longrightarrow x_{21} := e_{21} & ; & g_{22} \longrightarrow x_{22} := e_{22} & ; \quad \ldots \quad ; & g_{2m} \longrightarrow x_{2m} := e_{2m} \\ \Box & & & & \\ \ldots & & & & \\ \Box & & & & \\ g_{n1} \longrightarrow x_{n1} := e_{n1} & ; & g_{n2} \longrightarrow x_{n2} := e_{n2} & ; \quad \ldots \quad ; & g_{nm} \longrightarrow x_{nm} := e_{nm} \end{array} \right)$$

where $n, m \in \mathbb{N}$, and we assume that each of the n branches has m phases of guarded assignments.

We abbreviate the above pattern as:

$$G \longrightarrow \underset{i \in 1..n}{\Box} \left(\underset{j \in 1..m}{\fatsemi} g_{ij} \longrightarrow x_{ij} := e_{ij} \right)$$

To define $\mathcal{N}$ recursively, given any programs S and T, we have

$$\begin{array}{rcl} \mathcal{N}\,[\![S]\!]_I & = & P \longrightarrow \underset{i \in 1..n}{\Box} \left(\underset{j \in 1..m}{\fatsemi} p_{ij} \longrightarrow y_{ij} := e_{ij} \right) \\ \mathcal{N}\,[\![T]\!]_I & = & Q \longrightarrow \underset{i \in 1..n}{\Box} \left(\underset{j \in 1..m}{\fatsemi} q_{ij} \longrightarrow z_{ij} := f_{ij} \right) \end{array}$$

where we assume that, without the loss of generality, S and T have the same bounds $m, n \in \mathbb{N}$. We are then able to define $\mathcal{N}$ on the syntax of substitution programs (Section 2.5.2).

The rationale of letting $\mathcal{N}$ carry around the context invariant is specific to our notion of *wp* (Section 3.3). As we argue in Law 3.4.11, a parallel composition in our setting of *wp* cannot be rewritten as a choice of sequential compositions. Consequently, to transform a parallel composition to the above pattern of normal form we ought to strengthen the outer guard, satisfying which must guarantee that each

interleaving order is able to establish the context invariant I.

$$\mathcal{N}\,[\![S \parallel T]\!]_I = \\ c \longrightarrow \Box_{i,\,i' \in 1..n} \left(\begin{array}{l} \left(\fatsemi_{j \in 1..m} p_{ij} \longrightarrow y_{ij} := e_{ij} \right) ; \left(\fatsemi_{j' \in 1..m} q_{i'j'} \longrightarrow z_{i'j'} := f_{i'j'} \right) \\ \Box \\ \left(\fatsemi_{j' \in 1..m} q_{i'j'} \longrightarrow z_{i'j'} := f_{i'j'} \right) ; \left(\fatsemi_{j \in 1..m} p_{ij} \longrightarrow y_{ij} := e_{ij} \right) \end{array} \right)$$

$$\textbf{where } c = wp\,(S,\ wp\,(T,\ I)) \wedge wp\,(T,\ wp\,(S,\ I))$$

We define the generalised parallel simply in terms of $\mathcal{N}$ on $\parallel$, considering all possible ways of instantiating the parameterised substitution.

$$\mathcal{N}\,[\![! \ x \ : \ A \ . \ S]\!]_I = \mathcal{N}\,[\![\Box_{x' \in A} \left(\begin{array}{l} S\,[x'/x] \\ \parallel \\ ! \ x \ : \ A \setminus \{x'\} \ . \ S \end{array} \right)]\!]_I$$

Definitions of $\mathcal{N}$ on all other combinators are conceivable. Both *skip* and assignment are special cases of the above pattern as there is only one choice and the intended update is guarded by nothing.

$$\begin{array}{lcl} \mathcal{N}\,[\![skip]\!]_I & = & true \longrightarrow (true \longrightarrow x := x) \\ \mathcal{N}\,[\![x := e]\!]_I & = & true \longrightarrow (true \longrightarrow x := e) \end{array}$$

To normalise a guarded substitution $r \longrightarrow S$, we simply strengthen the outer guard of S (i.e. P) by r.

$$\mathcal{N}\,[\![r \longrightarrow S]\!]_I \ = \ (r \wedge P) \longrightarrow \Box_{i \in 1..n} \left(\fatsemi_{j \in 1..m} p_{ij} \longrightarrow y_{ij} := e_{ij} \right)$$

To normalise a choice $S \ \Box \ T$, we disjoin their outer guards and combine their choices of guarded assignments.

$$\mathcal{N}\,[\![S \ \Box \ T]\!]_I = (P \vee Q) \longrightarrow \left(\begin{array}{l} \Box_{i \in 1..n} \left(\fatsemi_{j \in 1..m} p_{ij} \longrightarrow y_{ij} := e_{ij} \right) \\ \Box \\ \Box_{i \in 1..n} \left(\fatsemi_{j \in 1..m} q_{ij} \longrightarrow z_{ij} := f_{ij} \right) \end{array} \right)$$

The treatment of the generalised choice @ $x \ : \ A \ . \ S$ considers all possible instantiations of the parameterised substitution S, each of which is based on a member drawn

from the set A—essentially we combine the choices of guarded assignments from $\mid A \mid$ versions of S.

$$\mathcal{N}\,[\![@\,x\ :\ A\ .\ S]\!]_I = \\ (\exists\, x : A \bullet P) \longrightarrow \underset{i\in 1..n,\ x'\in A}{\Box} \left(\left(\underset{j\in 1..m}{\overset{\circ}{9}} p_{ij} \longrightarrow y_{ij} := e_{ij} \right) [x'/x] \right)$$

The treatment of ; is similar to that of $\longrightarrow$: we propagate the outer guard Q of the second substitution T, for each of the choices it contains, to the first guarded assignment.

$$\mathcal{N}\,[\![S\ ;\ T]\!]_I = P \longrightarrow \underset{i,\ i'\in 1..n}{\Box} \left(\begin{array}{l} \left(\underset{j\in 1..m}{\overset{\circ}{9}} p_{ij} \longrightarrow y_{ij} := e_{ij} \right) ; \\ \left(Q \wedge q_{i'1} \longrightarrow z_{i'1} := f_{i'1} \right) ; \\ \left(\underset{j'\in 2..m}{\overset{\circ}{9}} q_{i'j'} \longrightarrow z_{i'j'} := f_{i'j'} \right) \end{array} \right)$$

Theorem 3.4.1 (***Normalised Form of GSL***)
Our above notion of normal form of substitutions is valid. More precisely,

$$\forall\, S : \mathit{Substitution};\ I : \mathit{Predicate} \bullet \mathit{wp}\,(S,\ I) = \mathit{wp}\,(\mathcal{N}\,[\![S]\!]_I,\ I)$$

□

Proof of Theorem 3.4.1 We prove this theorem by comparing the results of applying *wp* to both sides of each rule of the semantic function $\mathcal{N}$. □

3.5 Reasoning about BOOSTER Predicates

We have paved the way for deriving laws that can be exploited to rewrite behavioural specifications that are written as BOOSTER predicates—the properties of the BOOSTER compiler (Section 3.2), as well as the *wp* semantics (Section 3.3) and laws (Section 3.4.2) on its compiled form. We will prove that laws for BOOSTER predicate are distinguishable from those for standard first-order predicates [138].

3.5.1 Refinement vs. Equivalence

We already formalised our notion of refinement for programs of substitutions (Section 3.4.1). Since function $\mathcal{P}$ generates a guarded program on each BOOSTER method predicate p, we can then impose a refinement relation upon our BOOSTER method predicates accordingly, on the basis of the refinement relationship that exists among the guarded programs that we generate from them.

We are now able to impose a refinement relation upon the BOOSTER method predicates whose compiled form is that of guarded programs: a refinement relationship exists between two predicates p and q, denoted as $p \sqsubseteq q$, if and only if there exists one between the guarded programs we produce from them.

Definition 3.5.1 (***Refinement of* BOOSTER *Predicates***)

$$\begin{array}{|l} \_ \sqsubseteq \_ : \mathit{Predicate} \leftrightarrow \mathit{Predicate} \\ \hline \forall\, p, q : \mathit{Predicate} \bullet p \sqsubseteq q \Leftrightarrow \mathcal{P}\,[\![\, p \,]\!] \sqsubseteq \mathcal{P}\,[\![\, q \,]\!] \end{array}$$

□

We claim that all logical combinators of BOOSTER predicates are *monotonic* on the $\sqsubseteq$ relation.

Law 3.5.1 (***Monotonicity of* BOOSTER *predicate combinators***)

$$\begin{array}{lcll} g \Rightarrow p & \sqsubseteq & g \Rightarrow r & \texttt{(mono-}\Rightarrow\texttt{)} \\ p \wedge q & \sqsubseteq & r \wedge s & \texttt{(mono-}\wedge\texttt{)} \\ p \vee q & \sqsubseteq & r \vee s & \texttt{(mono-}\vee\texttt{)} \\ \forall x : A \bullet p & \sqsubseteq & \forall x : A \bullet r & \texttt{(mono-}\forall\texttt{)} \\ \exists x : A \bullet p & \sqsubseteq & \exists x : A \bullet r & \texttt{(mono-}\exists\texttt{)} \end{array}$$

provided $p \sqsubseteq r$ and $q \sqsubseteq s$. □

Proof of Law 3.5.1 As our notion of predicate refinement ($\sqsubseteq$) is defined in terms of the program refinement ($\sqsubseteq$) between programs that are generated via $\mathcal{P}$, the monotonicity property follows directly from Law 3.4.1. □

Accordingly, we impose an equivalence relation ($\equiv$) upon BOOSTER method predicates: two predicates are equivalent if and only if they are refined by each other, i.e. their generated programs via $\mathcal{P}$ are equivalent.

Definition 3.5.2 (***Equivalence of* BOOSTER *Predicates***)

$$\begin{array}{|l} \_ \equiv \_ : \mathit{Predicate} \leftrightarrow \mathit{Predicate} \\ \hline \forall\, p, q : \mathit{Predicate} \bullet p \equiv q \Leftrightarrow p \sqsubseteq q \wedge q \sqsubseteq p \end{array}$$

□

From the above definition and Law 3.5.1, it is straightforward to derive that all logical combinators of BOOSTER predicates are *monotonic* on the $\equiv$ relation.

When two BOOSTER method predicates are not equivalent, we will attempt to prove that one is mathematically more refined than the other and hence can safely be its substitute.

3.5.2 Laws

The syntactic structure of BOOSTER predicates resembles that of standard first-order predicates as presented in, for example, the seminal text by Gries [138]. A reasonable step forward would be to explore the consequence of equipping the abstract implementation language of BOOSTER predicates with *wp* semantics. More precisely, we examine the relevant theorems[6] on predicates in [138] and observe whether or not the relations of equivalence and implication still hold in the context of BOOSTER predicate methods.

All equivalence and refinement laws that we have proved and will present in this section are summarised in Table 3.1. Proofs for the majority of these laws are included in Appendix B. We give examples to these laws in the context of our hotel reservation system—we use *reserve*, *allocate*, and *cancel* to denote, respectively, the predicative

[6]We do not attempt to explore algebraic laws on $=$, as it reflects the compiler-specific heuristics on solving the constraints, through specific patterns of implementation. We will not examine laws on relational composition (;), either, as it is not listed as one of the predicate combinators in [138].

specification of reserving a hotel room, allocating a room to an existing reservation, and cancelling an existing reservation.

Laws of Equivalence

Conjunction and disjunction are symmetric in the BOOSTER context.

Law 3.5.2 (***Symmetry***)

$$\begin{array}{lcll} p \vee q & \equiv & q \vee p & \texttt{(sym-}\vee\texttt{)} \\ p \wedge q & \equiv & q \wedge p & \texttt{(sym-}\wedge\texttt{)} \end{array}$$

provided $\mathit{frame}\,(p) \cap \mathit{frame}\,(q) = \emptyset$ in the case of `(sym-∧)`. □

We state also the following basic laws on BOOSTER $\wedge$ and $\vee$:

Law 3.5.3 (***Associativity, Idempotency, and Distributivity***)

$$\begin{array}{lcll} (p \vee q) \vee r & \equiv & p \vee (q \vee r) & \texttt{(assoc-}\vee\texttt{)} \\ (p \wedge q) \wedge r & \equiv & p \wedge (q \wedge r) & \texttt{(assoc-}\wedge\texttt{)} \\ p \wedge p & \equiv & p & \texttt{(idem-}\wedge\texttt{)} \\ p \vee p & \equiv & p & \texttt{(idem-}\vee\texttt{)} \\ p \vee (q \vee r) & \equiv & (p \vee q) \vee (p \vee r) & \texttt{(dist-}\vee\texttt{-over-}\vee\texttt{)} \end{array}$$

provided $p = \mathit{before}\,(p)$ for case `(idem-∧)`. □

The side condition $p = \mathit{before}\,(p)$ is necessary for the case of `(idem-∧)` as in the LHS two instances of $\mathcal{P}\,[\![\, p \,]\!]$ are run in parallel, whereas in the RHS just one instance. Also, we have not included in Law 3.5.3 the distributivity of $\wedge$ over $\wedge$, as its LHS and RHS denote different numbers of program instances running in parallel and thus are not in general equivalent. However, we will see later, in Law 3.5.15, that the refinement relationship ($\sqsubseteq$) exists instead between the two sides.

Law 3.5.4 (***Zero, Identity***)

$$\begin{array}{lcll} p \vee \mathbf{true} & \equiv & \mathbf{true} & \texttt{(zero-}\vee\texttt{)} \\ p \wedge \mathbf{false} & \equiv & \mathbf{false} & \texttt{(zero-}\wedge\texttt{)} \\ p \vee \mathbf{false} & \equiv & p & \texttt{(iden-}\vee\texttt{)} \\ p \wedge \mathbf{true} & \equiv & p & \texttt{(iden-}\wedge\texttt{)} \end{array}$$

Laws	#	BOOSTER Predicates	Before Predicate(s)
Equivalence Laws			
Symmetry	3.5.2	$p \vee q \equiv q \vee p$	
		$p \wedge q \equiv q \wedge p$	see also main text
Associativity	3.5.3	$(p \vee q) \vee r \equiv p \vee (q \vee r)$	
		$(p \wedge q) \wedge r \equiv p \wedge (q \wedge r)$	
Idempotency	3.5.3	$p \vee p \equiv p$	
		$p \wedge p \equiv p$	p
Distributivity	3.5.3	$p \vee (q \vee r) \equiv (p \vee q) \vee (p \vee r)$	
	3.5.9	$p \Rightarrow (q \Rightarrow r) \equiv (p \Rightarrow q) \Rightarrow (p \Rightarrow r)$	p, q
Zero	3.5.4	$p \vee \mathbf{true} \equiv \mathbf{true}$	p
		$p \wedge \mathbf{false} \equiv \mathbf{false}$	
Identity	3.5.4	$p \vee \mathbf{false} \equiv p$	
		$p \wedge \mathbf{true} \equiv p$	
Def. of Implication	3.5.5	$p \Rightarrow q \equiv \neg\, p \vee (p \wedge q)$	p
Iden. of Implication	3.5.6	$\mathbf{true} \Rightarrow p \equiv p$	
True Antecedent	3.5.7	$p \wedge (p \Rightarrow q) \equiv p \wedge q$	p, see also main text
True Consequence	3.5.16	$p \equiv p \wedge (q \Rightarrow p)$	p, q
Shunting	3.5.8	$(p \wedge q) \Rightarrow r \equiv p \Rightarrow (q \Rightarrow r)$	p, q
Miscellaneous	3.5.10	$(p \Rightarrow r) \wedge (q \Rightarrow r) \equiv (p \vee q) \Rightarrow r$	p, q, r
		$(p \Rightarrow r) \wedge (\neg\, p \Rightarrow r) \equiv r$	p, r
		$p \vee (p \Rightarrow q) \equiv \mathbf{true}$	p, q
		$p \Rightarrow p \equiv \mathbf{true}$	p
		$p \Rightarrow \mathbf{true} \equiv \mathbf{true}$	p
		$p \Rightarrow \mathbf{false} \equiv \neg\, p$	p
		$\mathbf{false} \Rightarrow p \equiv \mathbf{true}$	
Refinement Laws			
Weakening/ Strengthening	3.5.11	$p \vee q \sqsubseteq p$	
	3.5.14	$p \sqsubseteq p \wedge q$	q
	3.5.18	$p \vee q \sqsubseteq p \wedge q$	q
	3.5.19	$p \vee q \sqsubseteq p \vee (q \wedge r)$	r
	3.5.20	$p \wedge (q \vee r) \sqsubseteq p \wedge q$	
Eliminating Unguarded Branch	3.5.13	$p \vee (q \Rightarrow p) \sqsubseteq (q \Rightarrow p)$	q
Zero of $\vee$	3.5.12	$p \vee \mathbf{true} \sqsubseteq \mathbf{true}$	
Dist. of $\wedge$ over $\wedge$	3.5.15	$p \wedge (q \wedge r) \sqsubseteq (p \wedge q) \wedge (p \wedge r)$	p
Absorption	3.5.17	$p \sqsubseteq p \wedge (p \vee q)$	p, q
		$p \vee (p \wedge q) \sqsubseteq p$	
		$p \wedge (\neg\, p \vee q) \sqsubseteq p \wedge q$	p
		$p \vee q \sqsubseteq p \vee (\neg\, p \wedge q)$	p
Refine. by $\Rightarrow$	3.5.21	$\neg\, p \vee q \sqsubseteq p \Rightarrow q$	p
Distributivity	3.5.22	$(p \vee q) \wedge (p \vee r) \sqsubseteq p \vee (q \wedge r)$	
	3.5.23	$p \wedge (q \vee r) \sqsubseteq (p \wedge q) \vee (p \wedge r)$	
Modus ponens	3.5.24	$q \sqsubseteq p \wedge (p \Rightarrow q)$	p

Table 3.1: Equivalence and Refinement Laws of BOOSTER Predicates

provided $p = \mathit{before}\,(p)$ in the case of `(zero-∨)`. □

In the BOOSTER context, we define an implication as either the antecedent not being satisfied (in which case the corresponding method is blocked and thus does not achieve anything else), or both the antecedent and the consequence are satisfied.

Law 3.5.5 (***Definition of Implication in* BOOSTER**)

$$p \Rightarrow q \;\equiv\; \neg\, p \vee (p \wedge q) \quad \texttt{(def-⇒)}$$

provided $p = \mathit{before}\,(p)$. □

Example (*Law 3.5.5*) When p = **true**, we have (**true** $\Rightarrow$ *reserve*) equivalent to **false** $\vee$ (**true** $\wedge$ *reserve*)—the effects of both are just *reserve*. When $p = (\#\mathit{reservations} < 1000)$, we have $(\#\mathit{reservations} < 1000 \Rightarrow \mathit{reserve})$ equivalent to $(\#\mathit{reservations} \geq 1000) \vee (\#\mathit{reservations} < 1000 \wedge \mathit{reserve})$—in both cases, the effect of *reserve* is applicable if the number of reserved rooms is below a limit, or otherwise it is blocked. □

In the context of conventional predicate logic [138], the RHS of Law 3.5.5 is equivalent to $(\neg\ p \vee q)$ and hence the standard definition $(p \Rightarrow q \equiv \neg\ p \vee q)$. However, we will see in Law 3.5.21 that this standard definition of implication does not hold in the BOOSTER context.

In the BOOSTER context, just as a program guarded by **true** is equivalent to itself, a **true** antecedent has no effect upon its consequence.

Law 3.5.6 (***Left Identity of Implication***)

$$\mathbf{true} \Rightarrow p \;\equiv\; p \quad \texttt{(left-iden-⇒)}$$

□

In a parallel composition, if one of the operands is a before-state constraint, then imposing it as an extra guard upon the other operand results in the same behaviour.

Law 3.5.7 (***True Antecedent***)

$$p \wedge (p \Rightarrow q) \equiv p \wedge q$$

provided $p = \mathit{before}\,(p)$. □

Example (*Law 3.5.7*) When $p = (\#\mathit{reservations} < 1000)$, it is the case that $((\#\mathit{reservations} < 1000) \wedge ((\#\mathit{reservations} < 1000) \Rightarrow \mathit{reserve}))$ is not equivalent to $((\#\mathit{reservations} < 1000) \wedge \mathit{reserve})$ because $\mathit{read}\,(p) \cap \mathit{frame}\,(\mathit{reserve}) = \{\,\mathit{reservations}\,\}$—in the latter case, after the effect of *reservations* is applied, the number of reserved rooms may just increment to 1000. Instead, we may choose, for example, $p = (\#\mathit{allocations} < 1000)$, provided that attribute *allocations* is not a member of $\mathit{frame}\,(\mathit{reserve})$. □

In the BOOSTER context, just as we can conjoin the outer and inner guards of a program, we can conjoin the outer and inner antecedents of an implication.

Law 3.5.8 (***Shunting***)

$$(p \wedge q) \Rightarrow r \equiv p \Rightarrow (q \Rightarrow r)$$

provided $p = \mathit{before}\,(p)$ and $q = \mathit{before}\,(q)$. □

Example (*Law 3.5.8*) Let $p = (r?.\mathit{room} \neq \mathit{null})$ and $q = (\mathit{earlier}\,(r?.\mathit{date},\ d?))$. We then have $((r?.\mathit{room} \neq \mathit{null}) \wedge (\mathit{earlier}\,(r?.\mathit{date},\ d?))) \Rightarrow \mathit{cancel}$ equivalent to $(r?.\mathit{room} \neq \mathit{null}) \Rightarrow ((\mathit{earlier}\,(r?.\mathit{date},\ d?)) \Rightarrow \mathit{cancel})$. Both have the effect that it is allowed to cancel when there has not been a specific room assigned to the input reservation, and when the reservation was not made later than an input date. □

We are able to distribute a BOOSTER implication over another.

Law 3.5.9 (***Distributivity of $\Rightarrow$ over $\Rightarrow$***)

$$p \Rightarrow (q \Rightarrow r) \quad \equiv \quad (p \Rightarrow q) \Rightarrow (p \Rightarrow r) \quad \texttt{(dist-⇒-over-⇒)}$$

provided $p = \mathit{before}\,(p)$ and $q = \mathit{before}\,(q)$. □

Example (*Law 3.5.9*) Similar to the Example of Law 3.5.8, imposing the constraint $p = (r?.room \neq null)$ twice is effectively the same as imposing it once. □

We also have equivalence laws where base predicates **true** and **false** are present.

Law 3.5.10 (***Miscellaneous*** $\equiv$)

$(p \Rightarrow r) \wedge (q \Rightarrow r)$	$\equiv$	$(p \vee q) \Rightarrow r$		(a)
$(p \Rightarrow r) \wedge (\neg\, p \Rightarrow r)$	$\equiv$	r		(b)
$p \vee (p \Rightarrow q)$	$\equiv$	**true**		(c)
$p \Rightarrow p$	$\equiv$	**true**	(reflex-$\Rightarrow$)	(d)
$p \Rightarrow$ **true**	$\equiv$	**true**	(right-zero-$\Rightarrow$)	(e)
$p \Rightarrow$ **false**	$\equiv$	$\neg\, p$		(f)
false $\Rightarrow p$	$\equiv$	**true**		(g)

provided $LHS = before\,(LHS)$ and $RHS = before\,(RHS)$ for cases (a) - (f); no side condition for case (g). □

Laws of Refinement

When two BOOSTER predicates are not interchangeable, we may instead be interested in finding if one is refined by the other, in the sense of $\sqsubseteq$. We may refine a disjunction by eliminating one of its operands.

Law 3.5.11 (***Refinement of Disjunction***)

$$p \vee q \sqsubseteq p$$

□

Example (*Law 3.5.11*) Let $p = reserve$ and $q = allocate$. We obtain a refinement *reserve* from $(reserve \vee allocate)$ through reducing the non-determinism, eliminating the blocking scenario resulting from the guard of *allocate* not being satisfied. □

By Law 3.5.11, we see immediately the Zero property of BOOSTER disjunction.

Law 3.5.12 (***Zero of*** $\vee$)

$$p \vee \textbf{true} \sqsubseteq \textbf{true}$$

□

We note that the LHS and RHS of Law 3.5.12 are identical to those of case (zero-$\vee$) in Law 3.5.4; in the case of refinement, we do not need the side condition $p = \mathit{before}\,(p)$.

Also an immediate derivation from Law 3.5.11 is: when there is a choice between a predicate p and a version of p with a non-trivial antecedent, we are able to eliminate the former and result in a refinement.

Law 3.5.13 (***Refinement by Eliminating the Unguarded Branch***)

$$p \vee (q \Rightarrow p) \sqsubseteq (q \Rightarrow p)$$

provided $q = \mathit{before}\,(q)$. □

We may refine a BOOSTER predicate p by requiring it to be simultaneously satisfied with a before-state constraint; this effectively imposes that constraint as an invariant upon the implementing program of p, i.e. to be maintained before and after its execution.

Law 3.5.14 (***Refinement by Conjunction***)

$$p \sqsubseteq p \wedge q$$

provided $q = \mathit{before}\,(q)$. □

Example (*Law 3.5.14*) When $q = (\#\mathit{reservations} < 1000)$, we can refine *reserve* by placing it in parallel with q—$(\#\mathit{reservations} < 1000) \wedge \mathit{reserve}$. This would mean that for the effect of *reserve* to be applicable, it is not only before its application that the number of reservations is below a specified limit, but also after its application—a stronger constraint than just imposing q as an extra guard for *reserve*. □

By applying Law 3.5.14, we are now able to complete the listing of Law 3.5.3:

Law 3.5.15 (***Distributivity of*** $\wedge$ ***over*** $\wedge$)

$$p \wedge (q \wedge r) \;\sqsubseteq\; (p \wedge q) \wedge (p \wedge r) \quad \texttt{(assoc-}\wedge\texttt{-over-}\wedge\texttt{)}$$

provided that $p = \mathit{before}\,(p)$. □

In the LHS the constraint p only has to be satisfied once—before, in-between, or after the parallel composition of q and r; in the RHS, we further require that p has to be satisfied twice.

Example (*Law 3.5.15*) Let q and r be *reserve*. Let $p = (\#\mathit{reservations} < 1000)$, such that $\mathit{read}\,(p)$ and $\mathit{frame}\,(\mathit{reserve})$ overlap—the singleton set $\{\,\mathit{reservations}\,\}$. The LHS—i.e. $(\#\mathit{reservations} < 1000) \wedge (\mathit{reserve} \wedge \mathit{reserve})$—is equivalent to the case where p is first placed with the first instance of *reserve*—i.e. $((\#\mathit{reservations} < 1000) \wedge \mathit{reserve}) \wedge \mathit{reserve}$—by `(assoc-∧)` of Law 3.5.3. We then obtain the RHS as a refinement by placing the same constraint in parallel with the second instance of *reserve*—i.e. $((\#\mathit{reservations} < 1000) \wedge \mathit{reserve}) \wedge ((\#\mathit{reservations} < 1000) \wedge \mathit{reserve})$—by Law 3.5.14 and the `mono-∧` of Law 3.5.1. One constrained scenario resulting from the RHS, but not the LHS, is that $(\#\mathit{reservations} < 1000)$ holds in between both instances of *reserve* are applied. □

Placing a predicate p in parallel with another before-state predicate, where p appears as the consequence, results in a refined predicate by Law 3.5.14—this would also automatically strengthen the relation to that of an equivalence.

Law 3.5.16 (***True Consequence***)

$$p \equiv p \wedge (q \Rightarrow p)$$

provided $p = \mathit{before}\,(p)$ and $q = \mathit{before}\,(q)$ □

By Laws 3.5.11 and 3.5.14, we can see immediately, also via Law 3.5.1 of monotonicity, the refinement relationship in the laws of absorption.

Law 3.5.17 (***Absorption***)

$$
\begin{array}{lcll}
p & \sqsubseteq & p \wedge (p \vee q) & \texttt{(Absorption (a))} \\
p \vee (p \wedge q) & \sqsubseteq & p & \texttt{(Absorption (b))} \\
p \wedge (\neg\, p \vee q) & \sqsubseteq & p \wedge q & \texttt{(Absorption (c))} \\
p \vee q & \sqsubseteq & p \vee (\neg\, p \wedge q) & \texttt{(Absorption (d))}
\end{array}
$$

There are side conditions because of the use of Law 3.5.14: for case `(a)` $p = \mathit{before}\,(p)$ and $q = \mathit{before}\,(q)$; for cases `(c)` and `(d)`, $p = \mathit{before}\,(p)$. □

Example (*Law 3.5.17*) Case `(c)` means that we obtain a refinement by eliminating a branch of choice from one of the parallel operands. For example, say $p = (\#\mathit{reservations} < 1000)$, then we have $(\#\mathit{reservations} < 1000) \wedge ((\#\mathit{reservations} \geq 1000) \vee \mathit{reserve})$ refined by $(\#\mathit{reservations} < 1000) \wedge \mathit{reserve}$. The RHS is more refined as it resolves the choice—the constraint of *reserve* must be satisfied, but not $(\#\mathit{reservations} \geq 1000)$. Case `(d)` means that we obtain a refinement by imposing an extra constraint on one of the parallel operands. For example, with the same p, then we have $(\#\mathit{reservations} < 1000) \vee \mathit{reserve}$ refined by $(\#\mathit{reservations} < 1000) \vee ((\#\mathit{reservations} \geq 1000) \wedge \mathit{reserve})$. The RHS is more refined as it further requires that $(\#\mathit{reservations} \geq 1000)$ holds before, or after, the effect of *reserve* is applied. □

We may refine disjoined predicates by their conjunction:

Law 3.5.18 (***Refining Disjunction by Conjunction***)

$$p \vee q \sqsubseteq p \wedge q$$

provided $q = \mathit{before}\,(q)$. □

Example (*Law 3.5.18*) Let $p = \mathit{reserve}$ and $q = (\#\mathit{reservations} < 1000)$. We then refine $\mathit{reserve} \vee (\#\mathit{reservations} < 1000)$ by $\mathit{reserve} \wedge (\#\mathit{reservations} < 1000)$. The former is a choice—between an unguarded *reserve* and a guarded action that achieves nothing. The latter is a refinement in that the number of reservations must be bounded—before and after the application of *reserve*. □

We may refine a choice by adding a parallel constraint on one of its branches:

Law 3.5.19 (***Refinement of Choice by Adding Parallel Constraint***)

$$p \vee q \sqsubseteq p \vee (q \wedge r)$$

provided $r = \mathit{before}\,(r)$. □

Proof of Law 3.5.19 Provided that $r = \mathit{before}\,(r)$, we know $q \sqsubseteq q \wedge r$ by Law 3.5.14, and then we complete the proof by `mono-`$\vee$ of Law 3.5.1. □

Example (*Law 3.5.19*) Let $p = \mathit{reserve}$, $q = \mathit{allocate}$, and $r = (\#\mathit{allocations} < \mathit{NUM\_OF\_ROOMS})$. We then refine $(\mathit{reserve} \vee \mathit{allocate})$ by $(\mathit{reserve} \vee (\mathit{allocate} \wedge (\#\mathit{allocations} < \mathit{NUM\_OF\_ROOMS})))$. The former is a choice—between (unguarded) *reserve* and *allocate*. The latter is a refinement in that before and after the application of *allocate*, the number of allocated rooms does not exceed the physical limit of the hotel. That is, this constraint is maintained as an invariant of *allocate*. □

We may refine a parallel composition by making one of its operands more deterministic, i.e. reducing one of its choices.

Law 3.5.20 (***Refinement of Parallel by Removing Choice***)

$$p \wedge (q \vee r) \sqsubseteq p \wedge q$$

□

Proof of Law 3.5.20 We know $q \vee r \sqsubseteq q$ by Law 3.5.11, and then we complete the proof by `mono-`$\wedge$ of Law 3.5.1. □

Example (*Law 3.5.20*) Let $p = \mathit{allocate}$, $q = \mathit{reserve}$, and $r = \mathit{cancel}$. We then have $\mathit{allocate} \wedge (\mathit{reserve} \vee \mathit{cancel})$ by $\mathit{allocate} \wedge \mathit{reserve}$. The right parallel operand of the former has a choice of cancelling an existing booking, requiring that a valid room reservation is supplied. The latter is a refinement in that we make that operand more deterministic—only the effect of *reserve* is to be applied. □

Laws 3.5.11, 3.5.14, 3.5.18, 3.5.19, 3.5.20 correspond to the list of theorems on weakening/strengthening in [138], except that our notion of predicate refinement ($\sqsubseteq$) reverses the direction of logical implication.

When there is a choice between a constraint and a program, we can refine it by negating the constraint and using it to guard the program.

Law 3.5.21 (***Redefining Standard Implication in*** **Booster**)

$$\neg\ p \vee q \sqsubseteq p \Rightarrow q$$

provided $p = \mathit{before}\,(p)$. □

Example (*Law 3.5.21*) Let $p = (\#\mathit{reservations} < 1000)$ and $q = \mathit{reserve}$. We then have $((\#\mathit{reservations} \geq 1000) \vee \mathit{reserve})$ refined by $((\#\mathit{reservations} < 1000) \Rightarrow \mathit{reserve})$. The former either achieves nothing when the number reservations exceeds a specified limit, or it achieves—unguardedly—the effect of *reserve*. The latter is a refinement in that it is more deterministic—achieving nothing is not allowed—and that the effect of *reserve* is now guarded by the number of reservations. □

When there is a parallel composition, each of its operands being a bounded choice and sharing a common branch, we may refine it by a choice between that common branch and a parallel composition of the other two branches.

Law 3.5.22 (***Distributivity of*** $\vee$ ***over*** $\wedge$)

$$(p \vee q) \wedge (p \vee r) \sqsubseteq p \vee (q \wedge r)$$

□

Example (*Law 3.5.22*) Let $p = \mathit{allocate}$, $q = \mathit{reserve}$, and $r = (\#\mathit{reservations} < \mathit{1000})$. We then have $((\mathit{allocate} \vee \mathit{reserve}) \wedge (\mathit{allocate} \vee (\#\mathit{reservations} < \mathit{1000})))$ refined by $(\mathit{allocate} \vee (\mathit{reserve} \wedge (\#\mathit{reservations} < \mathit{1000})))$. In the former, it is non-deterministic whether to impose $(\#\mathit{reservations} < \mathit{1000})$ as an invariant upon the application of *reserve*. The latter is a refinement in that its right branch of choice—i.e. $\mathit{reserve} \wedge (\#\mathit{reservations} < \mathit{1000})$—makes this matter deterministic. □

Distributing parallel composition over a choice results in a refinement.

Law 3.5.23 (***Distributivity of*** $\wedge$ ***over*** $\vee$)

$$p \wedge (q \vee r) \sqsubseteq (p \wedge q) \vee (p \wedge r)$$

□

When p is specified as a before-state constraint in Law 3.5.23, p will then be imposed before and after the effect of q or r is applied; this would strengthen the relation of $\sqsubseteq$ to that of $\equiv$. We demonstrate this special case below.

Example (*Law 3.5.23*) Let $p = (\#reservations < 1000)$, $q = allocate$, and $r = reserve$. We then have $((\#reservations < 1000) \wedge (allocate \vee reserve))$ refined by $(((\#reservations < 1000) \wedge allocate) \vee ((\#reservations < 1000) \wedge reserve))$. In both cases, the constraint $(\#reservations < 1000)$ is imposed as an invariant upon the application of either *allocate* or *reserve*. □

We may refine a predicate by adding to it a constraint as its antecedent and requiring the simultaneous satisfaction of both the resulting implication and the same antecedent.

Law 3.5.24 (***Modus Ponens***)

$$q \sqsubseteq p \wedge (p \Rightarrow q)$$

provided $p = before\,(p)$. □

Example (*Law 3.5.24*) Let $p = (\#reservations < 1000)$ and $q = reserve$. We then refine *reserve* by $(\#reservations < 1000) \wedge ((\#reservations < 1000) \Rightarrow reserve)$. The former is an unguarded application of *reserve*. The latter is a refinement in that $\#reservations < 1000$ is imposed as an invariant upon the application of *reserve*. □

A Theory of BOOSTER Predicates

We identify a class of BOOSTER predicates that can be treated as if they were the standard predicates.

Theorem 3.5.1 (**BOOSTER *predicates vs. standard predicates***)
Given two predicates p and q, we have

$$p \equiv q \;\Leftrightarrow\; p \equiv q \;\texttt{provided}\; p = before\,(p) \;\texttt{and}\; q = before\,(q) \quad \texttt{(a)}$$
$$p \Rightarrow q \;\Leftrightarrow\; q \sqsubseteq p \quad \texttt{(b)}$$

where the LHS corresponds to theorems[7] in the standard predicate calculus, whereas the RHS corresponds to, respectively, the equivalence ($\equiv$) and refinement ($\sqsubseteq$) relations on BOOSTER predicates. □

3.6 Example

We have seen examples for the majority of the equivalence and refinement laws of BOOSTER predicates (Section 3.5). We will now demonstrate a case which involves multiple steps of derivation. We write *reserve* to denote the predicative specification of making a room reservation, constants MAX_RES and MAX_ALLO to denote, respectively, the predefined (upper) bounds on the numbers of reservations and allocations in the hotel, and $LIMIT$ to denote the (lower) bound on a traveller's bank account balance.

$$
\begin{array}{ll}
 & reserve \vee (\#reservations \geq MAX\_RES) \\
\sqsubseteq & \{\ \texttt{Law 3.5.14, (mono-}\vee\texttt{) in Law 3.5.1}\ \} \\
 & ((\#allocation < MAX\_ALLO) \wedge reserve) \vee (\#reservations \geq MAX\_RES) \\
\sqsubseteq & \{\ \texttt{Law 3.5.21}\ \} \\
 & (\#reservations < MAX\_RES) \Rightarrow ((\#allocation < MAX\_ALLO) \wedge reserve) \\
\equiv & \{\ \texttt{Law 3.5.8}\ \} \\
 & (\#reservations < MAX\_RES \wedge \#allocation < MAX\_ALLO) \Rightarrow reserve \\
\sqsubseteq & \{\ \texttt{Law 3.5.24, (mono-}\Rightarrow\texttt{) in Law 3.5.1}\ \} \\
 & (\#reservations < MAX\_RES \wedge \#allocation < MAX\_ALLO) \Rightarrow \\
 & \quad (t?.balance \geq LIMIT) \wedge ((t?.balance \geq LIMIT) \Rightarrow reserve)
\end{array}
$$

As discussed in the beginning of this chapter, the above series of derivation (or rewriting) steps may have occurred in two distinct application scenarios. On the one hand, the developers may have completed the above rewriting process manually, and their intent of revision has been on a per-step basis—this is similar to how we interpreted each equivalence and refinement law in Section 3.5. On the other hand, the developers may have directly specified the resulting predicative specification to reflect how they think the availability of *reserve* should be constrained. More precisely, a traveller can only make a reservation when 1) the numbers of reservations and allocations of

[7] These do not include those theorems on quantifiers, which we do not consider in this dissertation.

the hotel have not exceeded certain predefined limits; and 2) that traveller's account balance has not gone, and will not go, below its credit limit before and after the reservation is made. The developers, however, were not sure if such an intent is connected to the original. As a result, assuming there has been a tool of symbolic execution implemented on the basis of our developed rules in Table 3.1, the developers may have launched the tool to search—automatically—for the series of derivation steps.

3.7 Conclusion

In this chapter we have addressed three issues as envisaged in Figure 1.1 (on page 3): *a language of transactions*, *properties of model completion*, and *rewriting rules*. We explored the logical consequence of both a weakest precondition semantics (*wp*) and a list of axiomatic properties of the BOOSTER compiler. The *wp* semantics (Section 3.3), together with the derived list of laws and a normal form of substitutions (Section 3.4), reflect the blocking policy that we wish to adopt at runtime. This semantics is given to an extended notion of program substitutions—first introduced in Section 2.5.2—that is used as the abstract implementation language of BOOSTER predicates. We expect any implementation of the BOOSTER compiler to abide by axioms (Section 3.2) as to how predicative specifications are completed—i.e. equipped with extra guarding constraints and updates—in a compositional manner. Sections 3.3 to 3.4 paved the way for deriving (Section 3.5) a list of laws—summarised in Table 3.1 on page 62—that facilitates the revision, when desired, of developers' predicative specifications.

We demonstrated on rewriting BOOSTER predicates by applying these laws—through one step (Section 3.5) or multiple steps (Section 3.6) of derivations. Each step of derivation is guaranteed—by the proof that we conducted for the corresponding law—to result in a predicate that is either an equivalent to or a refinement of the original one. In other words, not only can the BOOSTER language of predicates be adopted to describe the intended behavioural functionality, but it is also supported by a list of rewriting rules. Therefore, we have tested and verified the hypothesis 2.1.1

(on page 21). We will address: the issue of *language of workflows* that extends the solution space of transactions to that of workflows (Chapter 4); and the issue of *a theory of model-based testing* that helps developers reveal inconsistency in their design models (Chapter 5). The methodology that we will present in Chapter 5 provides a suitable context of model-based testing for exploiting the contribution of this chapter.

Chapter 4

Guarded Workflows

In this chapter we will test and verify the hypothesis 2.2.1 (on page 24) regarding workflows.

4.1 Introduction

As shown in Figure 1.1 (on page 3), we have introduced the BOOSTER notation as a language of transactions (Section 2.5)—we describe information systems as (abstract) object models. We write classes, attributes, associations, and invariants—these prototype, respectively, the participating entities, their defining features and relationship, and the intended business rules. More importantly, we write each update operation as a two-state, first-order predicate—whose pre-condition and post-condition will be implemented as, respectively, a guard and a transaction.

However, it is not sufficient to perform updates upon an information system as single operations; instead, it is useful to form *workflows*: meaningful flows or patterns of operations enacted to achieve a specific result, through a series of separate interactions, particularly to collect inputs at different stages. In regard to the roadmap of this thesis, as visualised in Figure 1.1, the resulting behavioural model of the system—which consists of operations and workflows—will be subject to a testing process (Chapter 5) to reveal any inconsistency, of particular kinds, that the model possesses. Fortunately, if the model is judged by its developers as needing revision, then the list of rewriting rules as developed in Chapter 3 usefully provides the devel-

opers with hints as to how the revision may be completed.

As it is not feasible to restrict access to the core data store for the entire duration of the updating, long-running workflow, we ought to allow its stages of execution to be interleaved with other operations or workflows, unavoidably causing them to interfere with each other. More specifically, although workflows that are composed of guarded transactions are guaranteed to preserve the integrity of the data—as each of the component transactions does so—the execution of workflows, when interference between them exists, is not guaranteed to *complete* successfully. As we have learnt from Scenario 1.1.2 (on page 7) and Example 2.2 (on page 23), during the execution of a *Booking* workflow of hotel rooms, the state of the system may be updated, by another operation or workflow running in parallel, in such a way that the next operation to be performed by *Booking* is now blocked, i.e. because the last room of the hotel has been taken by others. Workflows should be designed to accommodate possible changes in their shared execution environment. Upon finding in advance that the new workflow instance may be blocked halfway through its execution—due to the interference from other running instances—or that, symmetrically, it may block other workflows, we should then prevent it from being initiated.

On this account, what we will ultimately deliver in this Chapter is Section 4.7: a systematic approach to calculating a *precondition* for the successful completion, or achievement of a particular outcome, of a given set of workflows that are run in parallel. The challenge is that concurrent workflows interfere with each other through their component operations that modify the same shared state. However, we will argue that such a challenge can be met in the BOOSTER context of this thesis—each referenced operation in a workflow is a guarded transaction, the intended effect of each transaction is characterised as a state transformation, and any iteration is syntactically forbidden at the level of workflows.

To pave the way, we first define a workflow language (Section 4.2) that combines operations that are specified as relational constraints and implemented as guarded transactions. We give this language a structural operational semantics (SOS) [140]

in order to explain the *availability* of individual operations and the intended interpretation of each syntactic construct (Section 4.3). We then provide a denotational trace semantics (Section 4.4) that associates each workflow expression with a set of traces, i.e. sequences of action events that may be allowed by the current state to occur, and that characterises the constraint information contained within workflow expression. We also show that this trace semantics is consistent with the operational semantics (Section 4.5). The trace semantics can be used in combination with a relational semantics (Section 4.6) to formalise the precondition of a combination of workflows. We included the majority of our proofs for laws, lemmas, and theorems in Appendix C. Finally, we illustrate this application, together with a concrete syntax for the language, using an example involving the definitions of workflows upon a hotel reservation system.

4.2 A Guarded Workflow Language

Existing, graphical workflow notations such as BPMN [46] have not been defined with guarded transactions in mind: instead they assume that tasks are non-atomic, and that they begin execution as soon as the workflow is ready. As a result, the graphical notation of UML state diagrams is more appropriate for the illustration of workflows in our new language. This pattern of interaction with a hotel booking system can be illustrated using the state diagram of Figure 4.1.

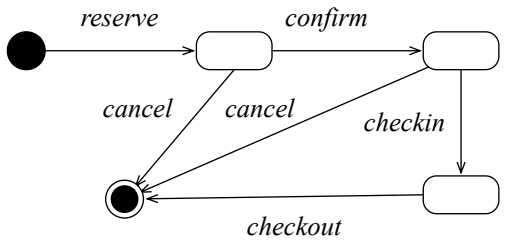


Figure 4.1: A simple booking workflow

A key consideration in the design of our language is the representation of successful completion. In keeping with the standard approach in CSP, we will write *skip*

to denote a workflow that does nothing but terminate successfully. Furthermore, although we will not have a use for such a constant in workflow specifications, it will be convenient to introduce *stop* to denote a workflow that has stopped permanently *without* terminating successfully.

We will use *actions* to represent the tasks performed by a workflow. For any action a and workflow W, we will write $a \rightarrow W$ to denote a workflow that is ready to perform a and then behave as W. Actions are not communication *events*, in the sense of Hoare's Communicating Sequential Processes (CSP) notation [141]: they may be performed without reference to other workflows. However, their availability is determined in part by the state of the information system itself: each action is associated with a *precondition*, a constraint upon the values of state, input, and local attributes, and cannot be performed unless that precondition is satisfied.

If *W1* and *W2* are workflows, then: we will write $W1 \Box W2$ to denote a workflow that is ready to behave as either *W1* or *W2*; we will write $W1 \; ; \; W2$ to denote a workflow that will behave as *W1* until that workflow terminates successfully reaching a stage denoted by *skip*, and then behave as *W2*; as workflows do not synchronise on their effects, we write $W1 \;|||\; W2$ to denote a workflow that behaves as *W1* and *W2* executing in parallel, through interleaving their component operations. However, not until Chapter 5 will we, for the purpose of testing the design model, introduce a workflow combinator of synchronous parallel that exploits the idea of a concurrent process as a constraint upon the pattern of behaviour, exemplified in the design of the CSP notation [141]. In the context of this chapter, when we say two workflows are running in parallel, we mean they do so without any synchronisation (i.e. interleaving).

We may declare variables for use in a workflow, providing transient, local state and allowing the parameters of one task to be related to parameters of another. If v is a variable and W is a workflow, then we write var v . W to denote a workflow that behaves exactly as W, but is able also to make use of v for storing values. To make use of a local variable, we include it in a *guard*: an expression that constrains the values of input and output variables of the next action to be performed. If g is a

guard, and W is a workflow, we write $g \,\&\, W$ to denote a workflow in which the first action to be performed by W is subject to the constraints of g.

The abstract syntax we have just introduced can be summarised by the following grammar:

$$W ::= \mathit{stop} \mid \mathit{skip} \mid \text{var } V \;.\; W \mid a \to W \mid g \,\&\, W \mid W \,\Box\, W \mid W \,;\, W \mid W \,|||\, W$$

For the purposes of this thesis, we will omit any treatment of iteration: in order that our preconditions can be automatically calculated, any iteration must be tightly controlled—or else the designer must assert a suitable, transformational specification for that part of the workflow, i.e. loop invariant and variant.

As an example, we define the textual form of Figure 4.1 using our guarded workflow language:

$$\begin{aligned} \mathit{Booking} &= \mathit{reserve} \to (\mathit{confirm} \to \mathit{Booked} \\ &\qquad\qquad \Box \\ &\qquad\qquad \mathit{cancel} \to \mathit{skip}) \\ \mathit{Booked} &= (\mathit{checkin} \to \mathit{checkout} \to \mathit{skip} \\ &\qquad \Box \\ &\qquad \mathit{cancel} \to \mathit{skip}) \end{aligned}$$

4.3 Operational semantics

We may use a collection of transition rules to characterise the intended behaviour associated with a workflow expression. The consequent part of each rule describes the effect of performing a particular action; the antecedent describes a set of conditions that must apply if the action is to be performed. As we may be maintaining information about the values of local variables as the workflow progresses, we consider a transition relation on pairs of expressions: the first component denoting a workflow, the second denoting a valuation of local variables.

Each transition is labelled by a semantic event, recording not only the name of the action but also the valuation of input, output, and local variables. In addition to the events formed from actions and valuations, we will introduce a special event ✓

to denote termination, and another special event τ to denote the transfer of control between components in a sequential composition or an interleaving.

The rule for *skip* is entirely straightforward: this is a workflow that may perform ✓ under any circumstances, and then performs no further actions.

$$\frac{}{(skip, \rho) \xrightarrow{\checkmark} (\Omega, \rho)}$$

where Ω represents a process that has successfully terminated. Although the behaviour of this process is similar to that of *stop*, it differs in that this semantic process will allow the performance of ✓ by another component running in interleaving, providing for the form of distributed termination adopted for CSP in [139]. *stop* itself does not require a rule, as no transitions are possible.

To present a rule for the action prefix operator $\rightarrow$, we must first consider the semantic properties of that action, considered as a transformation of the shared state of the information system. We may characterise each action as a relation, describing the possible effects of that action upon the state, in the context of some valuation of input, output, and local variables. Each state or valuation may be represented as a *binding*: a partial function from attributes to values. If we write Act to denote the set of all possible actions, and Bin to denote the set of all possible bindings, then we may define a function $\mathcal{R} \in Act \rightarrow (Bin \leftrightarrow Bin)$ that maps each action to the corresponding relation on bindings.

The performance of an action will be associated with a particular pair of bindings: one giving the values of state, input, and local variables before the action is performed; the other giving the values of state, output, and local variables afterwards. An action may be performed only if the first of these bindings lies within the *precondition* of the action, represented as the domain of the corresponding relation. We may record the fact of its performance as an event, with components of the event denoting those parts of the bindings corresponding to the input, output, and local variables: the visible attributes that take their values during the execution of the workflow.

If a is an action, we write $name\,(a)$ to denote its name, and $in\,(a)$ and $out\,(a)$ to denote the corresponding sets of input and output variables, respectively. If b is a binding, and s is a set of attributes, then we will write $s \lhd b$ to denote the part of b referring to attributes in s—the domain restriction of b, considered as a partial function. If ρ is a valuation of local variables, then we will write $\mathrm{dom}\,(\rho)$ to denote the set of variables currently bound: this includes variables currently bound to the special value $\perp$; these are variables that have been declared but not yet assigned a specific value.

We use Σ denotes the set of all events. If (b, b') is a pair of bindings associated with the performance of action a in the context ρ—the current valuation of any local variables—then the corresponding event is calculated by a function

$$\varepsilon \in (Act \times Bin \times Bin) \to \Sigma$$

Each event $\varepsilon(a, b, b') \in \Sigma$ contains the following information: the name of a; the domain restriction of b to inputs $in(a)$ and local variables $\mathrm{dom}(\rho)$; the domain restriction of b' to outputs $out(a)$ and local variables, which is again $\mathrm{dom}(\rho)$, as the performance of an action neither declares nor eliminates local variables from the scope of the workflow. The valuation of local variables afterwards is given by $\mathrm{dom}(\rho) \lhd b'$.

With these auxiliary definitions completed, we may exhibit the transition rule for action prefix:

$$\frac{(b, b') \in \mathcal{R}[\![a]\!]}{(a \to W, \rho) \overset{\varepsilon\,(a,b,b')}{\longrightarrow} (W, \mathrm{dom}(\rho) \lhd b')}$$

If the relational semantics of an action a includes the pair (b, b'), then a workflow ready to do a may perform that action, with the input variables bound in b, the outputs bound in b', and the local variables updated accordingly.

In the following, we will write ε for the expression $\varepsilon\,(a, b, b')$, where we assume that $(b, b') \in \mathcal{R}[\![a]\!]$, and ρ' for the expression $\mathrm{dom}\,(\rho) \lhd b'$. The rules for choice are

straightforward and familiar:

$$\frac{(W1, \rho) \xrightarrow{\varepsilon} (W1', \rho')}{(W1 \,\Box\, W2, \rho) \xrightarrow{\varepsilon} (W1', \rho')} \qquad \frac{(W2, \rho) \xrightarrow{\varepsilon} (W2', \rho')}{(W1 \,\Box\, W2, \rho) \xrightarrow{\varepsilon} (W2', \rho')}$$

The $\Box$ operator is used to introduce an exclusive choice of workflows. The choice is resolved only when an event occurs: the workflow that performed the corresponding action then proceeds, while the other is discarded.

The rules for interleaving make explicit the fact that our actions are not communication events in the sense of CSP: they are not shared between workflows, and hence either workflow in an interleaving may proceed without reference to the other:

$$\frac{(W1, \rho) \xrightarrow{\varepsilon} (W1', \rho')}{(W1 \,|||\, W2, \rho) \xrightarrow{\varepsilon} (W1' \,|||\, W2, \rho')} \qquad \frac{(W2, \rho) \xrightarrow{\varepsilon} (W2', \rho')}{(W1 \,|||\, W2, \rho) \xrightarrow{\varepsilon} (W1 \,|||\, W2', \rho')}$$

A side condition for these rules is that the workflows $W1$ and $W2$ do not share any local variables. If αW denotes the set of variables appearing freely in W, then for any combination $W1 \,|||\, W2$, we insist that $\alpha W1 \cap \alpha W2 = \{\}$.

When one of the interleaving workflows is ready to complete, it can do so without the consent of the other, and the ✓ transition is encapsulated as τ. An interleaving completes when both of its component workflows complete.

$$\frac{(W1,\ \rho) \xrightarrow{\checkmark} (W1',\ \rho')}{(W1 \,|||\, W2,\ \rho) \xrightarrow{\tau} (\Omega \,|||\, W2,\ \rho')} \qquad \frac{}{\Omega \,|||\, \Omega \xrightarrow{\checkmark} \Omega}$$

A sequential composition of workflows will proceed as the first workflow until the special event ✓ is performed, indicating that *skip* has been reached. When ✓ is performed, the composition starts to behave as the second component, $W2$.

$$\frac{(W1, \rho) \xrightarrow{\varepsilon} (W1', \rho') \qquad \varepsilon \neq \checkmark}{(W1 \,;\, W2, \rho) \xrightarrow{\varepsilon} (W1' \,;\, W2, \rho')} \qquad \frac{(W1, \rho) \xrightarrow{\checkmark} (W1', \rho')}{(W1 \,;\, W2, \rho) \xrightarrow{\tau} (W2, \rho')}$$

The ✓ is not visible outside the sequential composition, and its occurrence is marked by the special event τ.

A local variable declaration has the effect of extending the domain of the valu-

ation ρ. The initial binding of the new variable is to the special value $\perp$, denoting undefinedness.

$$\frac{(W, \rho \oplus \{ V \mapsto \perp \}) \xrightarrow{\varepsilon} (W', \rho'')}{(\text{var } V \ . \ W, \rho) \xrightarrow{\varepsilon} (W', \rho'')}$$

where $\rho'' = (\text{dom}\,(\rho) \cup \{ V \}) \lhd b'$.

A guard has the effect of constraining the availability of the next action, and hence the choice of bindings in the next visible event. If workflow W is ready to perform an action that would transform between bindings b and b', then $g \ \& \ W$ is ready to perform that same action only if the union of these bindings satisfies the constraint of guard g (denoted as $b \cup b' \models g$):

$$\frac{(W, \rho) \xrightarrow{\varepsilon} (W', \rho) \qquad b \cup b' \models g}{(g \ \& \ W, \rho) \xrightarrow{\varepsilon} (W', \rho')}$$

In most cases, only the first of these additional conjuncts needs to be considered: most guards will refer only to inputs and attributes of the state before the action. However, as our workflow descriptions are *specifications*, there is no reason why we should not also constrain the occurrence of an action on the basis of its possible outcome, or the value of an output.

4.4 Trace Semantics

We may give a denotational semantics to our language of workflows in terms of traces of action events, similar in many respects to the trace semantics of the CSP notation. The difference lies in the nature of action events: these record the observable information about each action as bindings of input, output, and local variables. As actions cannot be shared between workflows, our action events do not represent synchronisations.

Our trace semantics maps workflow expressions to sets of traces. If *Wfw* denotes the set of all valid workflow expressions, Σ denotes the set of all events, and Σ^* the

set of all traces, then our semantics is characterised by a function $\mathcal{T} \in \mathit{Wfw} \to \mathbb{P}\,\Sigma^*$, where $\mathbb{P}$ denotes the powerset operator. This function is itself defined in terms of a subsidiary function which holds the current valuation of local variables as a second parameter:

$$\mathcal{T}[\![W]\!] = \mathcal{T}[\![W]\!]_{\rho_0}$$

where ρ_0 denotes the initial, empty valuation.

The semantics of *skip* contains precisely two traces:

$$\mathcal{T}[\![\mathit{skip}]\!]_\rho = \{\langle\rangle, \langle\checkmark\rangle\}$$

while that of *stop* contains precisely one:

$$\mathcal{T}[\![\mathit{stop}]\!]_\rho = \{\langle\rangle\}$$

Neither of these workflows is affected by the current valuation of any local variables.

The semantics of the action prefix operator $\to$ is given by the following equation:

$$\mathcal{T}[\![a \to W]\!]_\rho = \{\langle\rangle\} \cup \{\langle\varepsilon\rangle \frown t' \mid (b, b') \in \mathcal{R}[\![a]\!] \wedge t' \in \mathcal{T}[\![W]\!]_{\rho'}\}$$

where $\varepsilon = \varepsilon\,(a, b, b')$ and $\rho' = (\mathrm{dom}(\rho) \lhd b')$, as before.

A choice of workflows may perform any trace of events that is possible for at least one of the two components. The semantics of the choice operator is thus given by:

$$\mathcal{T}[\![\mathit{W1} \,\Box\, \mathit{W2}]\!]_\rho = \mathcal{T}[\![\mathit{W1}]\!]_\rho \cup \mathcal{T}[\![\mathit{W2}]\!]_\rho$$

A sequential composition may perform any trace of the first component, together with any concatenation of traces composed as: a sequence of events from the first component, leading up to termination; followed by a sequence of events from the second component.

$$\begin{aligned}\mathcal{T}[\![\mathit{W1} \,;\, \mathit{W2}]\!]_\rho = {} & \{t \mid t \in \mathcal{T}[\![\mathit{W1}]\!]_\rho \wedge \checkmark \notin \mathrm{ran}(t)\} \\ & \cup \\ & \{\mathit{t1} \frown \mathit{t2} \mid \checkmark \notin \mathrm{ran}(\mathit{t1}) \wedge \mathit{t1} \frown \langle\checkmark\rangle \in \mathcal{T}[\![\mathit{W1}]\!]_\rho \wedge \mathit{t2} \in \mathcal{T}[\![\mathit{W2}]\!]_\rho\}\end{aligned}$$

Note that this leaves any environment variable declared in *W1* unbound in *W2*.

To define the semantics of an interleaving of workflows, we first introduce an operator upon traces. If *t1* and *t2* are traces of action events, then $t1 \,|||\, t2$ denotes the set of all possible interleavings, formed according to the familiar recursive definition: see e.g. [141]. The semantics we require is then simply:

$$\mathcal{T}[\![W1 \;|||\; W2]\!]_\rho = \bigcup \{ t1 \;|||\; t2 \mid t1 \in \mathcal{T}[\![W1]\!]_\rho \wedge t2 \in \mathcal{T}[\![W2]\!]_\rho \}$$

where $\bigcup$ denotes distributed union.

A local variable declaration is not associated with any observable event, but instead has the effect of extending the domain of the current valuation:

$$\mathcal{T}[\![\text{var } V \;.\; W]\!]_\rho = \mathcal{T}[\![W]\!]_{\rho \oplus \{ V \mapsto \perp \}}$$

where $\perp$ is the special value denoting undefinedness, as before.

The guard operator affects only the availability of the next action, and hence makes no difference to *stop*, and distributes through the choice and local variable operators:

$$\begin{array}{rcl} g \;\&\; stop & = & stop \\ g \;\&\; (W1 \;\Box\; W2) & = & (g \;\&\; W1) \;\Box\; (g \;\&\; W2) \\ g \;\&\; (\text{var } V \;.\; W) & = & \text{var } V \;.\; g \;\&\; W \end{array}$$

under the additional side condition that variable V cannot appear free in g if the last equation is to be valid. The guard operator does not distribute through the interleaving operator ($|||$) in general, but under certain side conditions.

Law 4.4.1 (***Distributivity of*** & ***through*** $|||$)

$$g \;\&\; (W1 \;|||\; W2) = (g \;\&\; W1) \;|||\; (g \;\&\; W2)$$

provided $W1 \mathrel{\widehat{=}} g_1 \;\&\; W1'$ and $W2 \mathrel{\widehat{=}} g_2 \;\&\; W2'$, where $g_1 \Rightarrow g$ and $g_2 \Rightarrow g$. $\Box$

In a sequential composition, the guard is associated with the first action to be performed, whether this is performed by the first or second component:

$$\begin{array}{rcl} g \;\&\; (W1 \;;\; W2) & = & (g \;\&\; W1) \;;\; W2 \\ (g \;\&\; skip) \;;\; W2 & = & g \;\&\; W2 \end{array}$$

The combination of two consecutive guards acts simply as their conjunction:

$$g \,\&\, (h \,\&\, W) \;=\; (g \wedge h) \,\&\, W$$

Its semantics is then characterised by its effect upon a workflow preceded by a single action.

$$g \,\&\, (a \to W) \;=\; (g \,\&\, a) \to W$$

where $(g \,\&\, a)$ denotes a guarded action, and the semantics of a guarded action prefix is given by

$$\begin{aligned}\mathcal{T}[\![(g \,\&\, a) \to W]\!]_\rho = \{\langle\rangle\} \\ \cup \\ \{\langle\varepsilon\rangle \frown t \mid (b, b') \in \mathcal{R}[\![a]\!] \wedge b \cup b' \models g \wedge t \in \mathcal{T}[\![W]\!]_{\rho'}\}\end{aligned}$$

where $\models$ denotes satisfaction, and ε and ρ' are the same abbreviations described above.

We may demonstrate that this denotational semantics is consistent with the operational semantics first used to define the language with a standard congruence proof. To do this, we introduce a multi-step transition relation on the graph formed by the existing transition relation, and then define an abstraction function to extract a term in our denotational semantic domain: that is, a set of possible traces for each workflow expression. We then show, inductively, that this is precisely the same set of traces identified by the semantic function $\mathcal{T}$.

If we wish simply to ensure that a workflow is guaranteed to complete, given that other operations may update the state while it is executing, we may consider this extended form of trace semantics. This is equivalent to considering each step of the workflow, and checking that the domain of the next operation is *total*: that is, it extends to the whole range of possible values for state, input, and local variables.

4.5 Congruence

We have described the workflow language as both a transition system in Section 4.3 and a list of inductive semantic rules (as mathematical relations) in Section 4.4.

We use the type synonym $Node == Wfw \times Env$ to represent each node in our transition system. As opposed to the normal one-step transition $\longrightarrow : (Node \times \Sigma) \leftrightarrow Node$, we defined a generalised version—i.e. $\Longrightarrow : (Node \times \operatorname{seq} \Sigma) \leftrightarrow Node$—that transits the system for multiple steps:

$$\begin{array}{rcl} P \overset{\langle\rangle}{\Longrightarrow} Q & \Leftrightarrow & P = Q \\ P \overset{\langle e \rangle^\frown s}{\Longrightarrow} Q & \Leftrightarrow & \exists R \bullet P \overset{e}{\longrightarrow} R \wedge R \overset{s}{\Longrightarrow} Q \end{array}$$

To prove that both semantic representations are *congruent*, we define an abstraction function $\Phi : Node \to \mathbb{P}\,\Sigma^*$ that extracts from the transition system a term in our denotational framework—a set of allowed traces with respect to some environment $\rho \in Env$—from each workflow $W \in Wfw$:

$$\Phi[\![W]\!]_\rho = \{ t : \Sigma^* \mid \exists (W', \rho') : Node \bullet (W, \rho) \overset{t}{\Longrightarrow} (W', \rho') \}$$

That is, we state the congruence theorem of our guarded workflow language as follows:

Theorem 4.5.1 The trace semantics of our guarded workflow language is congruent with its operational semantics: $\Phi[\![W]\!]_\rho = \mathcal{T}[\![W]\!]^\Phi_\rho$, where the superscript Φ in $\mathcal{T}[\![\_]\!]^\Phi_\_$ means that we substitute each recursive invocation of $\mathcal{T}$ by that of Φ. □

4.6 Relational Semantics

The trace semantics has been carefully designed to be compositional: the semantics of any compound workflow expression can be inferred directly from those of its components. Accordingly, the set of traces associated with a workflow represents the possible behaviours of that workflow in an arbitrary context: in particular, we have placed no constraint upon the way in which the state of the information system might

be updated while the workflow is executing.

We can say more about the progress properties of a workflow if we assume that no other workflows will act upon the same state variables while this workflow is executing. To do this, however, the trace semantics is not enough: we need also to consider the state of the underlying information system, for this is what will determine the availability and effect of each operation.

We now trade compositionality for specific analysis, and use our trace semantics to derive a relational semantics for workflows, mapping each workflow to a relation on bindings. In this relation, the first component represents the state of the system before the workflow begins, and the second represents the state upon successful termination of the workflow.

Writing ε for $\varepsilon\,(a, b, b')$ as before, the relation $\Delta\,(\varepsilon)$ is formed as follows:

$$\Delta\,(\varepsilon\,(a, b, b')) = \{\,(c, c') \mid (c \uplus (\theta \lhd b), c' \uplus (\theta \lhd b')) \in \mathcal{R}[\![a]\!]\,\}$$

where θ denotes the set of attributes of the state of the underlying information system, $\uplus$ denotes disjoint union, and $\lhd$ denotes domain subtraction. That is, $\Delta\,(\varepsilon)$ is the set of all pairs of states that could combine with the valuations of input, output, and local variables to form a pair of bindings in the relational semantics of action a.

The effect of performing a particular sequence of events s is given by the expression $\mathit{apply}\,(s)$, defined as follows:

$$\begin{array}{l}\mathit{apply}\,(\langle\rangle) = \mathit{id}\\ \mathit{apply}\,(\langle\varepsilon\rangle \frown s) = \Delta\,(\varepsilon) \fatsemi \mathit{apply}\,(s)\end{array}$$

where id denotes the identity relation and $\fatsemi$ denotes a particular form of relational composition. If $r \in X \leftrightarrow Y$ and $s \in Y \leftrightarrow Z$, then

$$\begin{array}{l}r \fatsemi s = \{\,(x, z) \mid (\exists\, y : Y \bullet (x, y) \in r \land (y, z) \in s) \land\\ \qquad\qquad (\forall\, y : Y \bullet (x, y) \in r \Rightarrow (y, z) \in s\,\}\end{array}$$

That is, the composition relates x to z if, and only if, there is a route from x to z by means of r and then s and every such route is guaranteed to succeed. This "pessimistic" form of relational composition is different from the familiar form of

composition used in, for example, the Z notation [7]. We consider the consequence of this in an example on our hotel reservation system.

Example (*"Pessimistic" Relational Composition*) Let *reserveOrAllocate* denote the relational constraint upon the system that either makes a new reservation—which is always allowed—or confirms an existing reservation and allocates to it a room. We also use function *extent* to retrieve, for a given class, the references of its set of instantiated objects.

$$\begin{aligned} \mathit{reserveOrAllocate} = {} & (r! \in \mathit{extent}\,(\mathit{Reservation}) \land r! \in \mathit{reservations}') \\ & \lor \\ & \left(\begin{array}{l} \#\mathit{allocations} < \mathit{ROOMS} \land a! \in \mathit{extent}\,(\mathit{Allocation}) \\ \quad \land\ a! \in \mathit{allocations}' \land a!.\mathit{room} = \mathit{room}? \end{array} \right) \end{aligned}$$

We consider *reserveOrAllocate* ⨾ *reserveOrAllocate*—the effect of running two instances of this action in order. There are four possible effects of this sequence—two consecutive reservations or allocations, or one reservation (or allocation) followed by the other. If we adopted the conventional notion of ⨾ as in Z, then for this sequence to complete, the precondition would be *true*, because making a reservation is never constrained. However, our intention is to consider all four cases and derive the strongest necessary precondition accordingly: the precondition resulting from two consecutive allocations, i.e. two rooms are initially available in the hotel. □

If we wish to examine whether a workflow is guaranteed to complete under the assumption that no other workflows interfere, we only have to check that every trace allowed by the relational semantics can be extended with a trace that leads to successful termination. If W is a workflow, then we define

$$\mathcal{T}[\![W]\!]\checkmark = \{t \in \mathcal{T}[\![W]\!] \mid \mathit{last}(t) = \checkmark\}$$

to denote the set of all successfully-terminating traces of W.

If t is a trace of workflow W, then we define $W \,/\, t$ to be the set of all possible characterisations of W after trace t has been performed, with the traces

$$\mathcal{T}[\![W \,/\, t]\!] = \{\, t' \mid t \frown t' \in \mathcal{T}[\![W]\!] \,\}$$

We may then observe that W is guaranteed to complete successfully from a state σ if, and only if,

$$\forall\, t, \sigma' \bullet t \in \mathcal{T}[\![W]\!] \land (\sigma, \sigma') \in \mathit{apply}\,(t) \Rightarrow \exists\, t' \in \mathcal{T}[\![W \,/\, t]\!] \checkmark \bullet \sigma' \in \operatorname{dom} \mathit{apply}(t') \tag{4.1}$$

The empty trace of events $\langle\rangle$ is always applicable, and we have $W \,/\, \langle\rangle = W$. We may thus derive the set of initial states and valuations of input and local variables from which a workflow can be guaranteed to complete. We refer to this set as $\mathit{pre}(W)$, the *precondition* of workflow W.

The same approach allows us to examine which combinations of workflows can be safely executed together without the need for a locking protocol. We have only to consider the precondition of an interleaving of workflows: for example, to determine the set of initial conditions under which workflows *W1* and *W2*, executing in combination, can be guaranteed to complete successfully, we have only to substitute $W1 \mid\mid\mid W2$ for W in the above constraint.

The lack of *compositionality* should be evident, and each combination of workflows will require a separate, fresh analysis: the value of $\mathit{pre}(W1 \mid\mid\mid W2)$ cannot be inferred from those of $\mathit{pre}(W1)$ and $\mathit{pre}(W2)$. While this may seem unsatisfying, it is a natural consequence of abstracting from the system state in our representation of traces. Fortunately, this does not prevent us from deriving value from the approach in practice.

4.7 A Strategy for Precondition Calculation

We present a concrete implementation for calculating the guard of *completion*, given a set of workflows running in interleaving. It is *efficient*, in that we do not have to explicitly consider all possible interleaving orders of the workflows involved. We will prove that this implementation is *correct*, in that it establishes the trace property (Equation 4.1) from the previous section.

Each event in traces that are computed by the function $\mathcal{T} : \mathit{Wfw} \to \mathbb{P} \operatorname{seq} \Sigma$

(Section 4.4) consists of the name of an action $a \in Act$ and a *concrete* binding of states, input and output values. This is not suitable for our purpose of statically calculating the guard of completion directly upon the workflow syntax, as the resulting constraints themselves will already characterise the set of legitimate bindings from which the interleaving execution of the set of workflows in question is guaranteed to complete. As a result, we define $\mathcal{T}' : Wfw \to \mathbb{P}(\text{seq}\, Act)$ as follows:

$$\begin{array}{rcl}
\mathcal{T}'[\![Stop]\!] & = & \{\langle\rangle\} \\
\mathcal{T}'[\![Skip]\!] & = & \{\langle\rangle\} \\
\mathcal{T}'[\![a \to W]\!] & = & \{\langle a\rangle \frown s \mid s \in \mathcal{T}'[\![W]\!]\} \\
\mathcal{T}'[\![W_1 \,\Box\, W_2]\!] & = & \mathcal{T}'[\![W_1]\!] \cup \mathcal{T}'[\![W_2]\!] \\
\mathcal{T}'[\![g \,\&\, W]\!] & = & \mathcal{T}'[\![W]\!] \\
\mathcal{T}'[\![W_1 \,|||\, W_2]\!] & = & \{u \mid u \in s \,|||\, t \wedge s \in \mathcal{T}'[\![W_1]\!] \wedge t \in \mathcal{T}'[\![W_2]\!]\} \\
\mathcal{T}'[\![W_1 \,;\, W_2]\!] & = & \{s \frown t \mid s \frown \langle\checkmark\rangle \in \mathcal{T}'[\![W_1]\!] \wedge t \frown \langle\checkmark\rangle \in \mathcal{T}'[\![W_2]\!]\}
\end{array}$$

For any given workflow $W \in Wfw$, the set of action sequences returned by $\mathcal{T}'[\![W]\!]$ is similar to the set of events returned by $\mathcal{T}[\![W]\!]\checkmark$ in that they both represent ways to successfully terminate W, should the binding of its starting state and inputs allow. But the set $\mathcal{T}'[\![W]\!]$ does not, however, include any concrete binding information.

The most naive approach to calculating the guard of completion for an interleaving $W_1 \,|||\, W_2$ is by defining a function $\mathcal{G} : Wfw \to Predicate$[1] that explicitly considers all possible interleaving orders of their successfully-terminating traces.

$$\begin{array}{l}
\mathcal{G}[\![W_1 \,|||\, W_2]\!] = \\
\quad \bigwedge g : \{complete\,(u) \mid u \in s \,|||\, t \wedge s \in \mathcal{T}'[\![W_1]\!] \wedge t \in \mathcal{T}'[\![W_2]\!]\} \bullet g
\end{array} \tag{4.2}$$

The auxiliary function *complete* calculates the guard for completing a single action sequence:

$$\begin{array}{rcl}
complete\,(\langle o\rangle) & = & guard\,(o) \\
complete\,(\langle o\rangle \frown os) & = & wp\,(intent\,(o), complete\,(os))
\end{array}$$

Without loss of generality, we focus our calculation on actions that are *deterministic* in their effects. Equivalently, we assume that for each $a \in Act$, its semantic interpretation $\mathcal{R}[\![a]\!]$ is functional. The auxiliary function *wp* is as defined in

[1] We overload the use of the symbol $\mathcal{G}$, which in the context of Chapter 3 operates on BOOSTER predicates and generates extra guarding constraints.

Section 3.3 that calculates the weakest precondition of a given substitution with respect to a desired predicate property. We also assume that the query function $guard : Action \rightarrow Predicate$ retrieves the guard constraint from a given action, and $intent : Action \rightarrow Substitution$ its intended update.

Obviously, the calculation of $\mathcal{G}$ (Equation 4.2) will quickly become infeasible because of the combinatorial explosion that results from the increase in either the number of interleaving workflows involved or the lengths of their traces.

Our solution addresses the following questions:

1. In the simplest scenario where W_1 and W_2 are *sequential workflows*, where the use of $|||$ is forbidden, is there a more efficient way to calculate the guard for $W_1 ||| W_2$ to complete without explicitly enumerating all their possible interleaving orders (i.e. $W_1 ||| W_2$)?

2. In the general case where one or both of W_1 and W_2 are themselves interleaving workflows, is it possible to generate a representative set S of interleaving orders whose guard of completion is equivalent to Equation 4.2? Of course, we should expect that $S \subset W_1 ||| W_2$ and $\#S$ is substantially smaller than $\#(W_1 ||| W_2)$.

Our solution will demonstrate that the calculation of a constraint upon which actions of sequential workflows W_1 and W_2 *commute*[2] is equivalent to that of $\mathcal{G}$ in Equation 4.2. The intuition is that if all actions from W_1 and W_2 commute, all their possible interleaved executions should bring us to the "identical" states, in that these states satisfy a common property: *completion* for our purpose.

The standard notion of *commutativity* (e.g. [142]) defines that two deterministic actions a and b *commute* if, and only if,

$$\begin{aligned} \forall\, \sigma \in Bin \mid \sigma \in (\operatorname{dom} \mathcal{R}[\![a]\!] \cap \operatorname{dom} \mathcal{R}[\![b]\!]) \bullet\ & \mathcal{R}[\![a]\!]\,(\sigma) \in \operatorname{dom} \mathcal{R}[\![b]\!] \\ & \wedge \\ & \mathcal{R}[\![b]\!]\,(\sigma) \in \operatorname{dom} \mathcal{R}[\![a]\!] \\ & \wedge \\ & \mathcal{R}[\![b]\!]\,(\mathcal{R}[\![a]\!]\,(\sigma)) = \mathcal{R}[\![a]\!]\,(\mathcal{R}[\![b]\!]\,(\sigma)) \end{aligned}$$

[2]Our notion that two actions commute is weaker than the standard notion.

For any state where both actions are enabled, it does not matter which action we execute first: the other will still be available and the resulting states from the two possible orders are the same.

However, as far as calculating the guard of completion is concerned, a weaker notion of commutativity suffices: actions a and b in the interleaving execution

$$(a \rightarrow W_1) \,|||\, (b \rightarrow W_2)$$

commute when neither of their interleaving orders $\langle a, b\rangle$ and $\langle b, a\rangle$ prevents the remaining part of the interleaving (i.e. $W_1 \,|||\, W_2$) from completing. More precisely, we weaken the third conjunct in the standard notion above as

$$\begin{array}{l} \mathcal{R}[\![b]\!]\,(\mathcal{R}[\![a]\!]\,(\sigma)) \models \mathcal{G}[\![\,W_1 \,|||\, W_2]\!] \\ \wedge \\ \mathcal{R}[\![a]\!]\,(\mathcal{R}[\![b]\!]\,(\sigma)) \models \mathcal{G}[\![\,W_1 \,|||\, W_2]\!] \end{array}$$

The post-states of actions a and b need not be identical as required in the standard notion; instead, $W_1 \,|||\, W_2$ being able to complete from both states will suffice.

We represent, respectively, patterns of sequential workflows W_1 and W_2 as action sequences $s, t : \mathrm{seq}\, Act$. We define a function *commute*:

$$commute : (\mathrm{seq}\, Act \times \mathrm{seq}\, Act) \rightarrow Predicate$$

The application of $commute\,(s, t)$ calculates a guard of completion that incorporates the guards of completion for all possible interleaving orders in $s \,|||\, t$.

To implement *commute*, we consider the inductive case of the interleaving operator (e.g. [141]):

$$\begin{array}{rl} \langle a\rangle \frown s \,|||\, \langle b\rangle \frown t = & \{\langle a\rangle \frown u \mid u \in s \,|||\, \langle b\rangle \frown t\} \\ & \cup \\ & \{\langle b\rangle \frown u \mid u \in \langle a\rangle \frown s \,|||\, t\} \end{array}$$

We will argue, and prove, that the calculation of the commutativity constraint for all interleaving orders $\langle a\rangle \frown s \,|||\, \langle b\rangle \frown t$ to complete can reuse those for $s \,|||\, \langle b\rangle \frown t$ and $\langle a\rangle \frown s \,|||\, t$.

To implement this, we apply the technique of dynamic programming [143] that

is substantially more efficient than explicitly considering the entire $s \,|||\, t$. We give the algorithmic specification of *commute* in Figure 4.2, where we assume that $s = \langle s_1, s_2, \ldots, s_m \rangle$, $t = \langle t_1, t_2, \ldots, t_n \rangle$, and array indices start with 1.

```
0   commute (s : Action[], t : Action[]) : Predicate {

1     int m, n             := s.length, t.length;
2     Predicate[][] matrix := new Predicate[m+1][n+1];
3     matrix[m+1][n+1]     := true;

4     for i : m .. 1
5       matrix[i][n+1] := wp(intent s[i], matrix[i+1][n+1]);

6     for j : n .. 1
7       matrix[m+1][j] := wp(intent t[j], matrix[m+1][j+1]);

8     for j : n .. 1
9       for i : m .. 1
10        matrix[i][j] := And (wp(intent s[i], matrix[i+1][j]),
11                             wp(intent t[j], matrix[i][j+1]));

12    return matrix[1][1]; }
```

Figure 4.2: Calculating Guard for Completion

The local variable *matrix* is a two-dimensional array (Figure 4.3) with its horizontal width $matrix.length = s.length + 1$ and vertical length $matrix[0].length = t.length + 1$. The horizontal indices start with 1 and increase rightwards up to $m + 1$; the vertical indices start with 1 and increase downwards up to $n + 1$.

The complexity of *commute* corresponds to the number of cells in *matrix* (i.e. $(\#s+1) \times (\#t + 1)$). The computation of *commute* consists of four stages:

1. We assign (at line 3) to the bottom-right cell (i.e. $matrix[m + 1][n + 1]$) a predicate value that characterises the ultimate goal to be achieved by every $u \in s \,|||\, t$ upon its completion. For now we are only concerned with the completion, hence the assigned value *true*.

2. We initialise (at lines 4 - 5 and 6 - 7, respectively) the cells at the bottom row

and the right-most column. The first iterations of both processes base their calculations upon the previously assigned value at $matrix[m+1][n+1]$; later iterations base theirs upon the previously calculated values that are stored in neighbouring cells at the immediate right and down sides, respectively.

3. We calculate and assign values for the remaining cells of *matrix* in the nested loop at lines 8 - 11. For each row we start with the right-most cell (i.e. with the horizontal index m) and gradually move to the left until reaching the left-most cell (i.e. with the horizontal index 1), in which case we go up to the row above and iterate through the process again. To calculate the value for a cell $matrix[i][j]$, where $1 \leq i \leq m$ and $1 \leq j \leq n$, we reuse those that are stored in neighbouring cells at its right side (i.e. $matrix[i+1][j]$) and down side (i.e. $matrix[i][j+1]$). These two cells are guaranteed to have already been assigned.

4. We finally return the value at the top-left cell (i.e. $matrix[1][1]$), which should, as we will prove below, characterise the commutativity constraint upon which all interleaving orders $s \mid\mid\mid t$ are guaranteed to—as specified at $matrix[m+1][n+1]$—complete.

`[1][1]`	...			...		`[m + 1][1]`
...						...
					`[m][`l` - 1]`	`[m + 1][`l` - 1]`
		`[k - 1][`l`]`	`[k][`l`]`		`[m][`l`]`	
		`[k - 1][`l` + 1]`				
...						...
					`[m][n]`	
`[1][n + 1]`	...			...		`[m + 1][n + 1]`

Figure 4.3: Variable *matrix* in function *commute* (Figure 4.2)

We will abbreviate the tail of a sequence u starting from an index i as $u_{i...}$. We state two lemmas upon the local variable *matrix* as declared in *commute* (Figure 4.2) and visualised in Figure 4.3.

Firstly, the bottom row of *matrix*—with its vertical indices fixed as $n+1$ (or respectively, right-most column with its horizontal indices fixed as $m+1$)—stores values that characterise the constraints upon which the suffix of s (or respectively, of t) is guaranteed to complete, without any interference from the other.

Lemma 4.7.1

$$\forall\, i, j : int \mid i \in 1\,..\,m \wedge j \in 1\,..\,n \bullet \begin{array}{l} matrix\,[i][n+1] = complete\,(s_{i..}) \\ \wedge \\ matrix\,[m+1][j] = complete\,(t_{j..}) \end{array}$$

where *complete* is defined as above. □

The second lemma on *matrix*—whose soundness relies upon that of Lemma 4.7.1—states that each cell $matrix[i][j]$ stores the commutativity constraint upon which $s_{i..} \mid\mid\mid t_{j..}$ is guaranteed to complete.

Lemma 4.7.2

$$\forall\, i, j : int \mid i \in 1\,..\,m \wedge j \in 1\,..\,n \bullet \\ matrix\,[i][j] = \bigwedge g : \{complete\,(u) \mid u \in s_{i..} \mid\mid\mid t_{j..}\} \bullet g$$

□

We now argue that whereas *commute* is computationally more efficient than $\mathcal{G}$, their returned results are equivalent.

Theorem 4.7.1 (***Correctness of*** *commute*) Given two action sequences s and t, we have

$$\begin{array}{rcl} commute\,(s,t) & = & \mathcal{G}[\![s \mid\mid\mid t]\!] \\ & = & \bigwedge g : \{complete\,(u) \mid u \in s \mid\mid\mid t\} \bullet g \end{array}$$

□

Proof of Theorem 4.7.1 The proof of Theorem 4.7.1 automatically follows by observing a direct implication from Lemma 4.7.2:

$$\begin{array}{rl} & matrix\,[1][1] \\ = & \{\ \texttt{Lemma 4.7.2}\ \} \\ & \bigwedge g : \{complete\,(u) \mid u \in s_{1..} \mid\mid\mid t_{1..}\} \bullet g \\ = & \{\ \texttt{def.of}\ \mathcal{G},\ s_{1..} = s,\ t_{1..} = t\ \} \\ & \mathcal{G}[\![s \mid\mid\mid t]\!] \end{array}$$

□

Therefore, we can elaborate the definition of $\mathcal{G}$ (Equation 4.2 on page 91) as

$$\begin{aligned} &\mathcal{G}_{com}[\![W_1 \mid\mid\mid W_2]\!] = \\ &\quad \bigwedge g : \{ \boxed{commute\,(s,t)} \mid s \in \mathcal{T}'[\![W_1]\!] \wedge t \in \mathcal{T}'[\![W_2]\!] \} \bullet g \end{aligned} \tag{4.3}$$

The calculations of $\mathcal{G}$ (Equation 4.2) and $\mathcal{G}_{com}$ both require the explicit enumerations of $\mathcal{T}'[\![W_1]\!]$ and $\mathcal{T}'[\![W_2]\!]$. But in $\mathcal{G}_{com}$ we have replaced in $\mathcal{G}$ the set of applications of $complete\,(u), u \in s \mid\mid\mid t$ with a single application of $commute\,(s,t)$. This means that for each pair of traces $(s,t) \in \mathcal{T}'[\![W_1]\!] \times \mathcal{T}'[\![W_2]\!]$, whereas $\mathcal{G}_{com}$ just performs a single application of $commute\,(s,t)$ to calculate the guard for their completion, $\mathcal{G}$ applies *complete* for a (potentially) much more substantial number (i.e. $\#(s \mid\mid\mid t)$) of times.

Undeniably, $\mathcal{G}_{com}$ considers sequences that are drawn from the trace function $\mathcal{T}'$, which may in turn apply the interleave operator $\mid\mid\mid$ if either or both of W_1 and W_2 are defined using the interleave operator $\mid\mid$. We now deliberate the following question:

Is it possible to substitute occurrences of $\mathcal{T}'[\![W]\!]$ *in Equation 4.3 by* $f\,[\![W]\!]$*, where* f *is a function* $f : Wfw \to \mathbb{P}(\mathrm{seq}\, Act)$ *such that* $f\,[\![W]\!] \subseteq \mathcal{T}'[\![W]\!]$*?*

In fact, the more substantial the difference in size is between $\mathcal{T}'[\![W]\!]$ and $f\,[\![W]\!]$, the greater the extent to which we can reduce the amount of applying *commute* in Equation 4.3.

However, we argue that the calculation of Equation 4.3 cannot be improved for its efficiency by applying *commute* to a strictly smaller subset of $\{(s,t) \mid s \in \mathcal{T}'[\![W_1]\!] \wedge t \in \mathcal{T}'[\![W_2]\!]\}$, because the returned predicate value of $\mathcal{G}_{com}$ may be weaker than it should be. We state this formally as the following theorem:

Theorem 4.7.2 (***Non-reducibility of*** $\mathcal{T}'$)

$$\forall f : Wfw \to \mathbb{P}(\mathrm{seq}\, Act) \bullet \left(\begin{array}{l} \forall W_1, W_2 : Wfw \bullet \\ \quad f\,(W_1) \subseteq \mathcal{T}'[\![W_1]\!] \wedge f\,(W_2) \subseteq \mathcal{T}'[\![W_2]\!] \wedge \\ \quad \mathcal{G}_{com}[\![W_1 \mid\mid\mid W_2]\!] = \\ \qquad \bigwedge g : \{ commute\,(s,t) \mid s \in f\,(W_1) \wedge t \in f\,(W_2) \} \bullet g \end{array} \right) \Rightarrow f = \mathcal{T}'$$

□

We gain our intuition of the proof for theorem 4.7.2 by considering an example:

Example (*Theorem 4.7.2*) Say two workflows W_1 and W_2 are defined as follows: $W_1 = (a \rightarrow b \rightarrow Skip) \,|||\, (c \rightarrow Skip)$ and $W_2 = (d \rightarrow e \rightarrow Skip) \,|||\, (f \rightarrow Skip)$. We then have $\mathcal{T}'[\![W_1]\!] = \{\langle a, b, c\rangle, \langle a, c, b\rangle, \langle c, a, b\rangle\}$ and $\mathcal{T}'[\![W_2]\!] = \{\langle d, e, f\rangle, \langle d, f, e\rangle, \langle f, d, e\rangle\}$. Theorem 4.7.2 claims that the computation of

$$\begin{array}{l} \mathcal{G}_{com}[\![W_1 \,|||\, W_2]\!] = \\ \quad \bigwedge g : \{commute\,(u) \mid u \in s \,|||\, t \wedge s \in \mathcal{T}'[\![W_1]\!] \wedge t \in \mathcal{T}'[\![W_2]\!]\} \bullet g \end{array}$$

cannot be improved by making either $\mathcal{T}'[\![W_1]\!]$ or $\mathcal{T}'[\![W_2]\!]$ strictly smaller in its size.

This is quite obvious, because if we remove any interleaving order from $\mathcal{T}'[\![W_1]\!]$ or $\mathcal{T}'[\![W_2]\!]$ for $\mathcal{G}_{com}$ to consider, there will be some interleaving order(s) in $W_1 \,|||\, W_2$ that we overlook to apply *commute* on and conjoin as part of the result of $\mathcal{G}_{com}$. Say we decide to remove $\langle a, b, c\rangle$ from $\mathcal{T}'[\![W_1]\!]$, then we will have the following constraint missing from the final result of $\mathcal{G}_{com}[\![W_1 \,|||\, W_2]\!]$:

$$\begin{array}{l} \bigwedge g : \{commute\,(u) \mid u \in s \,|||\, t \wedge s \in \{\langle a, b, c\rangle\} \wedge \\ \qquad\qquad\qquad\qquad\qquad t \in \{\langle d, e, f\rangle, \langle d, f, e\rangle, \langle f, d, e\rangle\}\} \bullet g \end{array}$$

The set of all these interleaving orders is disjoint from that of $(\mathcal{T}'[\![W_1]\!] \setminus \{\langle a, b, c\rangle\}) \,|||\, \mathcal{T}'[\![W_2]\!]$, and this means that the sequence $\langle a, b, c\rangle$ cannot be removed from $\mathcal{T}'[\![W_1]\!]$ for the computation of $\mathcal{G}_{com}$. Similar arguments apply to all other interleaving orders in $\mathcal{T}'[\![W_1]\!]$ and $\mathcal{T}'[\![W_2]\!]$. □

Theorem 4.7.2 ensures that $\mathcal{G}_{com}$ as it is (Equation 4.3) cannot be improved by reducing the size of its two arguments. This implies that for any two arbitrary workflows W_1 and W_2, the fewer levels of nested interleavings that occur in the definitions of W_1 and W_2, the better the computation of $\mathcal{G}_{com}[\![W_1 \,|||\, W_2]\!]$ performs.

Finally, we state the following theorem:

Theorem 4.7.3 A set RUN of workflows is currently active. There is a request for a new workflow $W \in Wfw$ to be run in interleaving with RUN. Our decision about

whether W should be granted to begin will be based upon whether $RUN' = |||\, r : RUN \cup \{W\} \bullet r$ can complete from the current state. Equivalently, we care whether the current state satisfies the guard of completion G for RUN' that is calculated as

$$G = \mathcal{G}_{com}[\![W \,|||\, RUN]\!]$$

Say the current state is σ, we claim that G is a sufficiently strong antecedent for the consequence of the completion property 4.1:

$$\sigma \models G \Rightarrow \left(\begin{array}{c} \forall\, t, \sigma' \bullet t \in \mathcal{T}[\![RUN']\!] \land (\sigma, \sigma') \in \mathit{apply}\,(t) \Rightarrow \\ \exists\, t' \in \mathcal{T}[\![RUN'/t]\!]\checkmark \bullet \sigma' \in \mathrm{dom}\ \mathit{apply}\,(t') \end{array} \right)$$

Should the calculated completion guard be satisfied by the current state, there is always *at least one* operation available from some workflow $r \in RUN'$ to let RUN' progress further. If another new workflow is to be added before the entire RUN' completes, yet another new guard will be calculated. □

Proof of Theorem 4.7.3 Theorem 4.7.3 follows directly from the definition of $\mathcal{G}_{com}$ (Equation 4.3) and Theorem 4.7.1. The computation of $G = \mathcal{G}_{com}[\![W \,|||\, RUN]\!]$ considers all interleaving orders of $W \,|||\, |||_{r \in RUN} r$, so the new interleaving $|||_{r \in \{W\} \cup RUN} r$ is guaranteed to complete as long as the current state satisfies G. □

4.8 Example

We describe two workflows—both targeted at our hotel reservation system—`MakeBooking` makes a booking and `ChangeBooking` changes an existing booking.

```
MakeBooking {
  r = r! & reserve ->
    ( r? = r & confirm ->
        r? = r & allocate -> skip
      []
      r? = r & cancel -> skip )  }
```

```
ChangeBooking {
  ( a = a? & deallocate -> skip )
  []
  ( a? = a & change ->
      a? = a & deallocate ->
        r = r! & reserve ->
          r? = r & allocate -> skip ) }
```

`MakeBooking` applies to a single reservation—created and identified with the first operation `reserve`—and then either confirms it by allocating a room in the hotel, or

cancels it. The expression `r = r!` binds the reference of the output reservation object `r!` to that of the local variable `r`, whereas the expression `r? = r` sets the value of input `r?` to be that of `r`. Similarly, `ChangeBooking` applies to an existing allocation—created and identified with some earlier call to `allocate`—and then either cancels it, or changes it through a new reservation.

The operation `allocate` is applicable only in those states where the number of allocations is bounded by what is physically available in the hotel. Furthermore, to allocate a room for a reservation `r?`—via `allocate`—`r?` must already exist in the system and be in a confirmed state. Accordingly, the availability of operation `reserve` is predicated upon rooms that have not been allocated. The operation `cancel` (or respectively, `deallocate`) can be applied only when the reservation in question, i.e. `r?` (or respectively, the allocation in question, i.e. `a?`), exists in the system.

$$
\begin{array}{lcl}
\mathit{guard}\,(\texttt{reserve}) & = & (\#\mathit{reservations} < \mathit{ROOMS} - \#\mathit{allocations}) \\
\mathit{guard}\,(\texttt{cancel}) & = & (r? \in \mathit{reservations}) \\
\mathit{guard}\,(\texttt{allocate}) & = & \left(\begin{array}{l} \#\mathit{allocations} < \mathit{ROOMS} \\ \quad \wedge\ r? \in \mathit{reservations} \\ \qquad \wedge\ r?.\mathit{status} = \mathit{confirmed} \end{array} \right) \\
\mathit{guard}\,(\texttt{deallocate}) & = & (a? \in \mathit{allocations})
\end{array}
$$

Operations `confirm` and `change` are most likely interactions between the customer (or their agent) and the hotel reservation system. Other operations—`reserve`, `allocate`, `cancel`, and `deallocate`—will be performed by the hotel staff when handling the customer's request. As these operations have non-trivial preconditions, an examination—via $\mathcal{T}'$ in Section 4.7—of the successful traces of the two workflows

$$
\begin{array}{lcl}
\mathcal{T}'[\![\mathit{MakeBooking}]\!] & = & \{\, \langle\, \texttt{reserve}, \texttt{confirm}, \texttt{allocate} \,\rangle, \\
 & & \quad \langle\, \texttt{reserve}, \texttt{cancel} \,\rangle \,\} \\
\mathcal{T}'[\![\mathit{ChangeBooking}]\!] & = & \{\, \langle\, \texttt{deallocate} \,\rangle, \\
 & & \quad \langle\, \texttt{change}, \texttt{deallocate}, \texttt{reserve}, \texttt{allocate} \,\rangle \,\}
\end{array}
$$

will reveal that even if we assume that no other workflows are enacted in interleaving, both workflows are not guaranteed to succeed—for each of them to complete, the hotel in question cannot be fully booked, i.e. there exists at least one available room

that has not been either reserved or allocated.

As both workflows do not contain any ||| in their definitions, we can calculate a suitable precondition for each of them to successfully complete by $\mathcal{G}$ in Equation 4.2 (on page 91) that considers both of its traces. Trace sets of both workflows will result in the same precondition constraint. More precisely, we have

$$\begin{array}{rcl} \mathcal{G}[\![\texttt{MakeBooking}]\!] & = & \bigwedge g : \{\, \mathit{complete}\,(t) \mid t \in \mathcal{T}'[\![\mathit{MakeBooking}]\!] \,\} \\ & = & \left(\begin{array}{c} \#\mathit{reservations} < \mathit{ROOMS} - \#\mathit{allocations} \\ \quad \wedge\, \#\mathit{allocations} < \mathit{ROOMS} \end{array} \right) \\ & = & \bigwedge g : \{\, \mathit{complete}\,(t) \mid t \in \mathcal{T}'[\![\mathit{ChangeBooking}]\!] \,\} \\ & = & \mathcal{G}[\![\texttt{ChangeBooking}]\!] \end{array}$$

where intents of `reserve`, `allocate`, and `confirm` automatically satisfy the constraints, respectively, that $r? \in \mathit{reservations}$, that $a? \in \mathit{allocations}$, and that $r?.\mathit{status} = \mathit{confirm}$.

Finally, we consider running `MakeBooking` and `ChangeBooking` in parallel. Say an instance of `MakeBooking` has just been initiated—which means there is at least one room available—so we have $\mathit{RUN} = \{\, \texttt{MakeBooking} \,\}$. To decide if a new instance of `ChangeBooking` can be initiated in parallel with `MakeBooking`, we ask if $\mathit{RUN}' =$ (`ChangeBooking` ||| `MakeBooking`) is able to complete. However, we must admit that the precondition for this parallel composition to complete—characterised through $\mathcal{G}$ in Equation 4.2 (on page 91)—cannot be calculated on the basis of $\mathcal{G}[\![\texttt{MakeBooking}]\!]$ and $\mathcal{G}[\![\texttt{ChangeBooking}]\!]$. More precisely, we have $\mathcal{G}[\![\texttt{MakeBooking}]\!]$ not equivalent to $\mathcal{G}[\![\texttt{MakeBooking}]\!] \wedge \mathcal{G}[\![\texttt{ChangeBooking}]\!]$. As an example, the member trace

$$\langle\, \texttt{reserve},\ \texttt{confirm},\ \texttt{change},\ \texttt{deallocate} \,\rangle$$

of $\mathcal{T}'[\![\texttt{MakeBooking} \mid\!\mid\!\mid \texttt{ChangeBooking}]\!]$ may lead us to a state in which both workflows are ready to perform an allocation—if fewer than two rooms were initially available, then the combination would get stuck and neither of them could complete.

Our Theorem 4.7.3 (on page 98) has been developed specifically for this kind of problem. We apply the routine *commute*—as implemented in Figure 4.2 (on page 94)

and proved for its correctness in Theorem 4.7.1 (on page 96)—on RUN'.

$$\begin{array}{ll} & \mathcal{G}_{com}[\![\texttt{MakeBooking} \,|||\, \texttt{ChangeBooking}]\!] \\ = & \bigwedge g : \{commute\,(s,t) \mid s \in \mathcal{T}'[\![\texttt{MakeBooking}]\!] \wedge t \in \mathcal{T}'[\![\texttt{ChangeBooking}]\!]\} \bullet g \\ = & \left(\begin{array}{c} \#reservations < ROOMS - \#allocations \\ \wedge\ 2 \leq ROOMS - \#allocations \end{array} \right) \end{array}$$

Therefore, the calculated predicate from $\mathcal{G}_{com}[\![\texttt{MakeBooking} \,|||\, \texttt{ChangeBooking}]\!]$ will be evaluated on the current state—should it not be satisfied, we block the initiation of the new instance of *ChangeBooking* accordingly.

4.9 Conclusion

In this chapter we have addressed the issue of *a language of workflows* as envisaged in Figure 1.1 (on page 3). The ability to specify the intended behavioural functionality as first-order predicates (Section 2.5 and Chapter 3), as well as to combine such formal descriptions to specify recurring patterns of updates, complete the picture of behavioural specification in BOOSTER.

We introduced the syntax of a workflow language (Section 4.2)—references to the names of operations denote their relational constraints on the shared state. On the basis of the relational specification of these component activities, we defined an operational semantics (Section 4.3) and a trace semantics (Section 4.4) for our workflow language—the latter is *compositional* by considering all possible ways of interference that might occur between updates of actions. We also established that these two kinds of semantics are consistent (Section 4.5). Furthermore, we traded compositionality for analysis by defining a relational semantics (Section 4.6) which imposes a *closure* on the interference being allowed—this enables us to define the notion of a *precondition* for the successful completion of concurrent workflows. Sections 4.2 to 4.6 paved the way for a strategy—efficient for its adoption of the dynamic programming technique—of systematically calculating such a precondition—we also proved its correctness.

We demonstrated on our hotel reservation system (Section 4.8) how—given a set of workflows to be run in parallel—the calculation of a precondition for completion can help us take a precautionary step before a new workflow instance is initiated. We block—according to how the calculated precondition evaluates on the current state—the start of a new workflow instance if its interference on the shared state has the potential to block the progress of any other that is already running, or vice versa. This means we are able to effectively and efficiently predict the model behaviour under intertwining updates that are potentially initiated by distributed clients. Therefore, we have tested and verified the hypothesis 2.2.1 (on page 24). We will address: the issue of *a theory of model-based testing* that serves as the next step forwards for performing tests on such behavioural models and revealing any inconsistency of particular kinds (Chapter 5); and the issue of *proved rules of model transformation* that turns the behavioural model that is both complete and consistent into working SQL database queries (Chapter 6).

Chapter 5

Testing of Model Consistency

In this chapter we will test and verify the hypothesis 2.3.1 (on page 26) regarding testing the consistency of design models.

5.1 Introduction

As envisaged in Figure 1.1, we have presented a unified modelling notation for describing the intended behavioural functionality of an information system. This modelling notation consists of a language of transactions (Section 2.5) and a language of workflows (Chapter 4). We describe individual model operations—which will be implemented as guarded transactions—as BOOSTER predicates, whose syntax resembles that of the standard first-order predicates [7, 138]. We describe recurring patterns of operations as workflows, whose syntax is based upon Hoare's Communicating Sequential Processes (CSP) notation [141]. The semantics of our transaction and workflow languages are unified: workflow combinators effectively serve as relational operators that combine the relational constraints (imposed upon the shared state) of their component transactions.

However, if the behavioural model is *inconsistent*—with respect to the developers' understandings of the requirements—then either the model will admit no implementation, or the implementation produced will not behave according to the intended requirements. Fortunately, both BOOSTER model operations (written as predicates) and our guarded workflow language (equipped with a relational semantics on the

shared state) are amenable to a formal analysis of the design model. We hypothesise that a systematic manipulation of these method predicates will help us detect inconsistency (of particular kinds) in the source behavioural model and usefully suggest how it might be resolved accordingly. This effectively prevents us from generating an implementation from any inconsistent model.

What we will ultimately deliver in this chapter is Section 5.3: a model-based testing theory on behavioural design models that are written in our languages of transactions (Section 2.5) and workflows (Chapter 4). We will explore our notion of "model slicing" by adapting the conventional notion of program slicing [12] to the context of formal, declarative design models. Our testing process assists in detecting particular kinds of model inconsistency through revealing—i.e. slicing—the developers' (implicit) intents from the models under test. Our notion of model consistency in this chapter focuses on aspects of 1) the data flow of a particular attribute; 2) the occurrence of a particular pattern of events; 3) the notion of *commutativity* in which a set (or sets) of events can occur in any order; and 4) a well-formed combination of 1) to 3).

As visualised in Figure 5.1, our testing process is novel in two respects:

1. the ability of the users of our approach (e.g. the developers) to formally specify scenario(s) that they are concerned with or expect to (or not to) happen; and

2. the formal characterisation of a predicate that denotes the set of valid initial configurations (i.e. its precondition), starting from which guarantees that the model's execution will realise the corresponding scenario.

The systematic process that we will devise to generate the second item (i.e. the precondition of a specified scenario) essentially *tests* the given design model. As various structural and semantic model constraints can make it difficult for the developers to predict every possible behaviour, they may usefully specify scenarios (of the above kinds) as test cases. That is, they will test the precise circumstances under which the

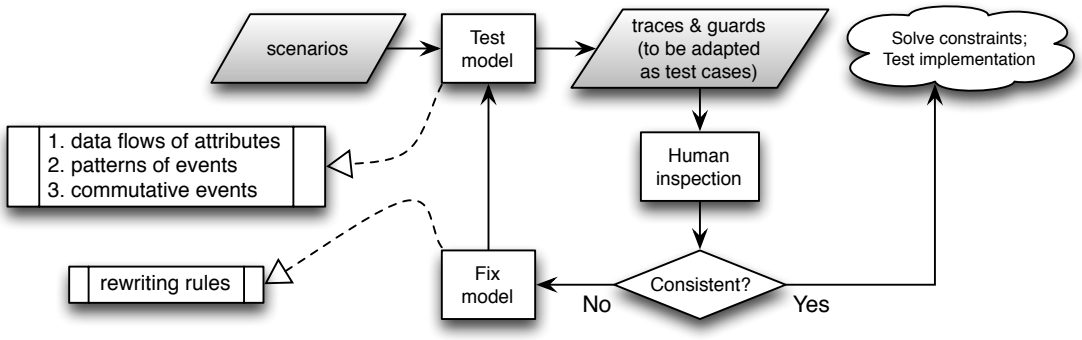


Figure 5.1: Model-Based Testing of Design Models (elaborating on Figure 1.2 on page 10)

occurrences of these scenarios are allowed—by constraints of the model under test. Let us consider a number of use cases of the testing theory that we will develop.

Example (*Testing Behavioural Design Models*) The developers of the hotel reservation system have now completed its structural and behavioural specification (Appendix A.2). As it is not feasible to explore all possible interactions between model operations, our testing methodology allows the developers to perform tests on their model. They can usefully query how possible it is to change the value of attribute *allocations*—which records the number of confirmed bookings—and later use its value. Indeed, attribute *allocations* may be referenced in many operations, but the developers may not be interested in exploring all of their interactions. Instead, we allow the developers to reduce the space of analysis by considering, for example, only scenarios where a cancellation occurs. One possible test result might be $(\langle\, reserve,\ allocate,\ cancel \,\rangle,\ false)$. This would mean that the current model is not able to perform a cancellation following a confirmed booking. It is then up to the developers to judge whether or not the precondition *false* is too strong. They may have to modify the specification of method *cancel* (or other relevant model constraints) and re-run the model test. □

Results from testing the model are *operation traces* and their associated preconditions; these constraints are to be implemented as *guards*. Each trace denotes an

execution scenario of the model behaviour under test—i.e. a concurrent execution of workflows—and its associated guard describes the circumstances under which that execution is guaranteed to be completed successfully. These results provide useful hints for developers to fix their models should they consider the generated precondition(s) inconsistent with their understandings of the requirements. Our developed rewriting rules of behavioural specifications (Section 3.5) provides a formal ground for the developers to revise their model if demanded. Such a testing process may be applied iteratively until the produced results are judged as consistent by the developers and thus ready to be adapted as (concrete) test cases for the model implementation. This is particularly valuable in the context of model-based development, where the manual development of systems gives no guarantee with respect to the specification model. In later sections of this chapter, we will refer to the results from our model test as test cases.

To pave the way, we will first adapt the standard notion of data-flow testing—applied to procedural programs—to the formal, declarative context of BOOSTER methods (Section 5.2). More specifically, we will infer two ranges of intended data-flow usages: those within scope of each individual BOOSTER predicate method (Section 5.2.1), as well as those that may occur in-between methods (Section 5.2.2). Our interpretation of data-flow testing on BOOSTER predicates will contribute to one of the kinds of scenario specification in our approach to testing workflows. We will then present our methodology on testing workflows in the presence of concurrency (Section 5.3): the design of its inputs and outputs (Section 5.3.1); a list of heuristics (Section 5.3.2) that enables us to specify scenarios with which we might be concerned and thus helps us deal with the potential combinatorial explosion that results from interleaving workflows; and a systematic process (Section 5.3.3) of calculating preconditions satisfying which would guarantee that the implemented model at runtime will exhibit the specified scenarios. We will discuss (Section 5.4) how the equivalence and refinement laws of BOOSTER predicates developed in Section 3.5 serve as a valid point of reference when the developers decide that their model under test is incon-

sistent. Finally, we will demonstrate our approach in the hotel reservation system (Section 5.5).

5.2 Data-Flow Analysis of BOOSTER Methods

Our data-flow analysis of BOOSTER methods focuses upon attributes instead of objects. We interpret BOOSTER model operations as constraints upon attribute values instead of upon objects. We perceive the system state as a list of binary relations. Each relation represents the state of an attribute that maps between object references of its current class and target class.

Example (*Formal Interpretation of System State*) Part of the system state of `HotelReservation` (Appendix A.2; see Figure 1.4 on page 13) is a partial function $account : extent\,(Traveller) \nrightarrow extent\,(Account)$, where sets $extent\,(Traveller)$ and $extent\,(Account)$ contain references to, respectively, traveller and account objects. Each model operation is then interpreted as performing the corresponding relational operation(s) on the relevant attribute function(s). For instance, changing the account of traveller t to another $a?$ is formalised as $account := account \oplus \{t \mapsto a?\}$. □

Therefore, in the BOOSTER context we concentrate our data-flow analysis entirely upon *definitions* and *uses* of attributes instead of objects. It is as if the analysis were done in the context of procedural programming. Each variable is an attribute function variable like *account*. Furthermore, the notion of code execution is replaced by the evaluation of complex predicates. A pair of line numbers—where an attribute is defined and used—is no longer a valid way for representing a test case that "locates" within each method predicate. Instead, we may represent each test case by calculating a *precondition* that is strong enough to guide the "execution path" of the method predicate—more precisely, the execution path of its implementing program—to cover the corresponding def-use pattern. As a result, for each attribute a, we redefine the standard notions (Section 2.3) of (re)defining its value (i.e. $def(a)$), using it for computation (i.e. $c\text{-}use(a)$), and using it for predicate (i.e. $p\text{-}use(a)$).

5.2.1 Intra-method Usage Patterns

Our goal in this section is to define a semantic function TC which operates on a single BOOSTER predicate method and extracts "test cases" for model consistency. Each "test case" consists of an attribute usage pattern and its associated precondition. For example, we write $TC\,(\mathit{allocations},\ \mathit{deallocate})$ to extract—from the predicative specification of method *deallocate*—all usage patterns of attribute *allocations* and under what circumstances they are allowed to occur. We will first elaborate on our notions of "usage patterns" and "test cases".

Usage Patterns

For an attribute a that is referenced in method m, it can be that a is: 1) defined without being used in m (*def-only*); 2) used (for predicate or computation) without being defined in m (*use-only*); 3) defined then used in m (*def-then-use*); and 4) used then defined in m (*use-then-def*). The last two cases generally represent a "phased" method specified via the relational composition operator ($;$). A special case to note is when a's value is used to update itself. Namely, we consider each of the predicate expressions $a' = a + x$, $x \in a'$, and $x \notin a'$ as a *use-then-def* pattern for attribute a. The disjunction operator (`or`) makes it possible for method m to contain multiple intra-method usage patterns for a.

The above four attribute usage patterns define how we may "slice" our behavioural model specification—each generated test case reflects how the current model constrains the availability of a usage pattern of interest. To collect the above four data usage patterns we define a collection of transformation rules that are implemented in the Haskell functional language [144]. Using a functional language to implement our model transformation—as opposed to an imperative language like Java—has the advantage that these functions resemble the corresponding mathematical, semantic functions that extract usage patterns from method predicates. That is, how exactly the data usage patterns should be extracted manifests itself in the series of Haskell functions. More specifically, these transformation functions use higher-order

functions (e.g. `map`, `foldr`) rather than explicit loops. We will present some transformation functions in the math form, but adapting it to the executable Haskell syntax is straightforward—e.g. $\mathbb{P}\, T$ implemented as `[T]`.

We define as follows an abstract syntax for expressing *intra*-method usage patterns (of an attribute).

$$\begin{aligned} \mathit{Usage} ::= {} & \mathit{Def}\langle\langle \mathit{Name} \rangle\rangle \mid \mathit{CUse}\langle\langle \mathit{Name} \rangle\rangle \mid \mathit{PUse}\langle\langle \mathit{Name} \rangle\rangle \\ \mid {} & \mathit{DefThenCUse}\langle\langle \mathit{Name} \rangle\rangle \mid \mathit{DefThenPUse}\langle\langle \mathit{Name} \rangle\rangle \\ \mid {} & \mathit{CUseThenDef}\langle\langle \mathit{Name} \rangle\rangle \mid \mathit{PUseThenDef}\langle\langle \mathit{Name} \rangle\rangle \end{aligned}$$

For example, we write $\mathit{PUse}\,(\mathit{reservations})$ to denote a *predicate-use* pattern of the attribute. A predicate method may contain zero or more of the above usage patterns of a chosen attribute. Now that we have a clear notion of attribute usage patterns, we will now explore how to extract "test cases"—each of which consists of a usage pattern and its associated precondition—from the predicate method under test.

Extracting Usage Patterns as Test Cases

We define *TCase* as a type synonym for $\mathit{Usage} \times \mathit{Predicate}$. A test case $(u, c) \in \mathit{TCase}$ is a pair that consists of an *intra*-method usage pattern u and its associated precondition constraint c. Should c be satisfied, the usage pattern u is guaranteed to be "executed" by the predicate method under test. We then define our semantic function

$$\mathit{TC} : (\mathit{Attribute} \times \mathit{Predicate}) \to \mathbb{P}\, \mathit{TCase}$$

which extracts—given an attribute and a method predicate—pairs of type *TCase*. We write $\mathit{TC}\,(a,\ m)$ to denote the set of *intra*-method test cases for attribute a in predicate method m. Therefore, we can define, for example, our previous notion of "attribute a is *def-then-use* in method m" formally in terms of *TC*

$$\begin{aligned} &\text{a } \texttt{is } \mathit{def\text{-}then\text{-}use} \texttt{ in m} \;\Leftrightarrow \\ &\quad \exists\, c_1, c_2 : \mathit{Predicate} \bullet (\mathit{DefThenCUse}\,(a),\ c_1) \in \mathit{TC}\,(a,\ m) \lor \\ &\qquad\qquad\qquad\qquad (\mathit{DefThenPUse}\,(a),\ c_2) \in \mathit{TC}\,(a,\ m) \end{aligned}$$

where constraint c_1 or c_2 may be *false*. This would mean that it is never possible for the model under test to perform the pattern. The developers may or may not decide to fix the relevant model constraints accordingly. Similar definitions and arguments apply to "attribute a is *use-then-def* in method m", etc. Let us consider an example of a test case for model consistency.

Example (*Attribute Usage Pattern & Precondition*) We observe the usage patterns of attribute *allocations*—which records the set of confirmed bookings—in the predicative specification of method *deallocate* (Section 2.5.1). Since *deallocate* contains just a single phase—i.e. the relational composition operator (`;`) is absent—we have $\neg\ (\exists\, c : \mathit{Predicate} \bullet (\mathit{DefThenCUse}\,(\mathit{allocations}),\ c) \in \mathit{TC}\,(\mathit{allocations},\ \mathit{deallocate}))$. On the other hand, attribute *allocations* is expanded with a new allocation $a? \in \mathit{extent}\,(\mathit{Allocation})$, provided $a?$ already exists in the system. As a result, we have $(\mathit{CUse}\,(\mathit{allocations}),\ a? \in \mathit{allocations}) \in \mathit{TC}\,(\mathit{allocations},\ \mathit{deallocate})$. □

We now begin to define the function TC that extracts attribute usage patterns—through rules on patterns of predicate methods to which TC can be applied. The three base cases of function TC are binary relations—i.e. `=`, `:`, and `/:`—that denote changes on the state.

$$
\begin{aligned}
\mathit{TC}\,(a,\ \texttt{e = f}) \ =\ & \textbf{if } e == a' \wedge \mathit{occurs}\,(a, f) \textbf{ then} \\
& \quad \{(\mathit{CUseThenDef}\,(a),\ \mathit{true})\} \cup \mathit{TC}'\,(a,\ f) \\
& \textbf{else if } e == a' \textbf{ then} \\
& \quad \{(\mathit{Def}\,(a),\ \mathit{true}),\ (\mathit{Def}\,(\mathit{opp}\,(a)),\ \mathit{true})\} \cup \mathit{TC}'\,(a,\ f) \\
& \textbf{else} \\
& \quad \mathit{TC}'\,(a,\ e) \cup \mathit{TC}'\,(a,\ f)
\end{aligned}
$$

When the left-hand side of an equality relation `e = f` denotes the after value (i.e. a') of the attribute a in question, then it is either that a's value is used to update itself (i.e. $\mathit{occurs}\,(a, f)$, or that it is considered as a *Def* usage. In the former, we consider it as a *CUseThenDef* usage; in the latter, we collect not only attribute a, but also its opposite $\mathit{opp}\,(a)$. For example, in Figure 1.4 (on page 13), an update to the owner of an account would also involve an update to its opposite: the account of that traveller.

Otherwise, we recurse on expressions—via another semantic function TC' which we will define below—from both sides of the equality. The same result applies to the other two relations of set membership $TC\,(a,\ f\texttt{:}e)$ (interpreted as an insertion) and non-membership $TC\,(a,\ f\texttt{/:}e)$ (interpreted as a deletion).

Just as the BOOSTER compiler amends users' partial specification with additional updates that maintain the association invariant—when given a user-specified predicate method—we should consider not only attributes that are explicitly referred to, but also those that are declared to be their mirrors.

Example (*Inferring Updates on Bi-Associations*) We consider the predicate method *activate* that updates the bi-association between classes `Traveller` and `Account` (see Figure 1.4 on page 13). The completed version of *activate* (under class `Traveller` in Appendix A.2) updates both sides of the association; however, method *activate* may just be written as: $(amount? \in int \wedge acc! \in extent\,(Account) \Rightarrow acc!.balance' = amount? \wedge this.account' = acc!)$. The application of $TC\,(account,\ activate)$ should also consider the mirror attribute of *account*, i.e. *owner*, and return test cases for both $Def\,(account)$ and $Def\,(owner)$. □

Before we continue to present the remaining rules of TC on relational combinators, we introduce another transformation function TC' that serves an identical purpose of extracting attribute usage patterns—but from primitive expression terms.

$$TC' : (Attribute \times Expression) \rightarrow \mathbb{P}\ TCase$$

We note the subtle difference between functions TC and TC': the former is applicable to the (top-level) logical and relational terms that evaluate to boolean values.

Our definition of TC' is as concise as one equation in Haskell:

```
TC' (a, e) = (everything (++) ([] `mkQ` (matchPath a))) e
```

where `everything` and `mkQ` are drawn from a library module that supports the design pattern[1] as reported in [145]. Instead of exhaustively enumerating all forms of

[1] A more detailed account on the relevant usage of this library is included in Appendix E.10.

expressions for pattern matching, we are able to specify only the specific "form" of expressions that we wish to consider while ignoring all others. More precisely, we express our concern by passing an auxiliary function

$$matchPath : Attribute \to Path \to \mathbb{P}\, TCase$$

$$matchPath\,(a)\,(path) = \textbf{if}\ target\,(path) == a\ \textbf{then}\ \{(CUse\,(a),\ true)\}\ \textbf{else}\ \{\}$$

Function *matchPath* compares the target of its input path expression *path* against its input attribute a. Should they match, we then consider it as a computational use by default. All results from *matchPath* on applicable expressions (i.e. of type *Path*) are combined using the concatenation operator `++`, and the initial combination starts with the empty sequence `[]`.

The above base cases of TC and TC' that detect *Def* and *CUse* patterns specify *true* as the default constraint. We will now present the remaining rules of TC on logical combinators; these inductive cases of function TC will consider operands of the four logical operators (i.e. `&`, `=>`, `or`, and `;`) and strengthen (or weaken) the constraint accordingly.

For an implication (`=>`), all computational usages of a in its antecedent are switched to predicate usages, and the antecedent p is taken to strengthen the constraint of each test case detected in its consequence q. We first consider an example.

Example (*Usage Patterns in Implication*) We consider the predicative specification of method *deallocate* (Section 2.5.1) that is described as an implication. Its antecedent $a? \in allocations$ requires that the input allocation $a?$ already exists in the system. We expect $TC'\,(allocations,\ deallocate)$ to return a test case $(CUse\,(allocations),\ true)$ by default. Since this usage of *allocations* occurs in the context of an antecedent, however, it will be modified—by the above rule of TC on `=>`—to a new test case $(PUse\,(allocations),\ true)$. We will use this antecedent to guard usage patterns that are extracted from the consequence. For instance, we expect TC to first extract from $a? \notin allocations'$ a test case $(CUseThenDef\,(allocations),\ true)$,

and then to guard it with the antecedent in context and result in a new test case $(CUseThenDef\ (allocations),\ a? \in allocations)$. □

We formalise what we expect from the above example scenario as follows.

$$TC\,(a,\ p\ \texttt{=>}\ q) = puse\,(TC\,(a, p)) \cup \{(u, p \wedge q') \mid (u, q') \in TC\,(a,\ q)\}$$

We replace—via the auxiliary function *puse* (of type $\mathbb{P}\,TCase \rightarrow \mathbb{P}\,TCase$)—each occurrence of *CUse* by *PUse*.

```
puse = map (\tc@(u, p) -> case u of CUse a    -> (PUse a, p)
                                    otherwise ->  tc)
```

Here we apply our assumption in Section 3.2 about the antecedent—$p = before\,(p)$ and consequently p contains `CUse` patterns only.

In both conjunction (`&`) and disjunction (`or`), we are concerned with test cases that appear in their two operands but refer to the same attribute usage pattern u, i.e. (u, c_1) and (u, c_2). As both operands of `&` must be satisfied simultaneously, we reduce them into a single test case by conjoining the two associated constraints, i.e. $(u, c_1 \wedge c_2)$. On the other hand, as the choice operator (`or`) contributes to the non-deterministic behaviour of the predicate method in question, our calculation should incorporate intended effects that are inferred from both branches. Consequently, we reduce by disjoining the two constraints, i.e. $(u, c_1 \vee c_2)$.

We will first consider an example:

Example (*Usage Patterns in Conjunction and Disjunction*) We consider the specification of an update on the hotel reservation system:

$$\begin{array}{l} \#allocations < 1000 \Rightarrow t?.account.withdraw(amount? = a?.rate) \\ \oplus \\ \#allocations \geq 1000 \Rightarrow t?.account.withdraw(amount? = a?.rate \times weight?) \end{array}$$

where $\oplus$ denotes either a conjunction or a disjunction. A traveller $t?$ may be charged—via method *withdraw* (see class `Account` in Appendix A.2)—with a higher, weighted rate for their existing allocation $a?$ when the hotel is currently booked above a specific

extent. As both implications have the intent to modify the same attribute *balance* (in class *Account*), we have two test cases generated—$(Def\ (balance),\ \#allocations < 1000)$ and $(Def\ (balance),\ \#allocations \geq 1000)$. If $\oplus$ is $\vee$, then we expect the model to execute either of the *CUse* patterns of *balance*—we have $(Def\ (balance),\ true)$ as the combined test case. On the other hand, if $\oplus$ is $\wedge$, we have a test case $(Def\ (balance),\ false)$—this informs the developers that the current model constraints do not allow both usage patterns to occur at the same time. Should the developers find this generated test case unsatisfactory, they may either change $\oplus$ to $\vee$, or modify the two antecedents such that they are not disjoint. □

We formalise what we expect from the above example scenario as follows.

$$\begin{array}{lcl} TC\,(a,\ p\ \texttt{\&}\ q) & = & reduce\,(TC\,(a,\ p) \cup TC\,(a,\ q))\ conjoin \\ TC\,(a,\ p\ \texttt{or}\ q) & = & reduce\,(TC\,(a,\ p) \cup TC\,(a,\ q))\ disjoin \end{array}$$

The union of test cases collected from both operands will include, if any, pairs of test cases that refer to the same attribute usage. We specify the reduction routine as the auxiliary function

$$reduce : \mathbb{P}\ TCase \rightarrow (TCase \rightarrow TCase \rightarrow TCase) \rightarrow \mathbb{P}\ TCase$$

Given a set of test cases *tcs*, we perform $reduce\,(tcs)\,(f)$ to reduce it based on f that specifies how pairs of test cases should be reduced, which can be either of the following:

$$\begin{array}{lcl} conjoin\,(u,\ p)\,(u,\ p') & = & (u,\ p \wedge p') \\ disjoin\,(u,\ p)\,(u,\ p') & = & (u,\ p \vee p') \end{array}$$

The reduction process is iterative and specified using the standard higher order function `foldr`. We reduce a set of test cases to another by searching for a "matching" pair of test cases that both refer to the same attribute usage and reducing them—via *conjoin* or *disjoin*—into a single test case. We repeat this process until no matching pair can be found.

```
reduce []              f = []
reduce (tc@(u, p):tcs) f = if null match then
                             tc:(reduce tcs f)
```

```
else
  (foldr f tc match) : reduce (tcs \\ (tc:match))
where match = [(u', p') | (u', p') <- tcs, u == u']
```

Finally, if a predicate method is divided into two phases via the relational composition operator (`;`), we have to consider generating test cases from usage patterns that occur in either or both of the two phases.

Example (*Usage Patterns in Relational Composition*) We consider the specification of an update on the hotel reservation system:

$$\#reservations \leq 1000 \Rightarrow r? \in reservations'$$
$$\overset{\circ}{9}$$
$$\left(\begin{array}{l} \#reservations \leq 1000 \Rightarrow \\ \quad s? \in reservations' \wedge t?.account.withdraw\,(amount? = r?.rate + s?.rate) \end{array} \right)$$

A traveller $t?$ may make two reservations—written as $r?$ and $s?$—and charged for their processing fees. For each reservation we require that so far the number of reservations has been bounded—to ensure that the hotel always has free rooms for priority or emergency use. We are interested in two *Def* usage patterns: 1) one for attribute *reservations* that is modified in the first phase; and 2) the other for attribute *balance* (see method `withdraw` of class `Account` in Appendix A.2) that is modified only in the second phase. We are also interested in the *PUse* pattern of attribute *reservations* that occurs in the antecedent of the second phase.

For pattern *Def* (*balance*) to occur, we need to ensure that not only the antecedent of the first phase is established, but also that the intended update in the first phase does not falsify the antecedent in the second phase. More precisely, we should have a test case (*Def* (*balance*), $\#reservations < 1000$), where the associated precondition for pattern *Def* (*balance*) ends up being stronger than the antecedent of the second phase. On the other hand, we may compose patterns *Def* (*reservations*) and *PUse* (*reservations*) that occur in the two phases. For the *PUse* pattern to occur, we need to ensure that the antecedent of the first phase is established. More precisely, we should have a test case (*DefThenPUse* (*reservations*), $\#reservations \leq 1000$). Of course, if there were a *CUse* (*reservations*) pattern in the second phase, then we

would expect a stronger precondition—the intent of the first phase has to establish the antecedent of the second phase. □

We formalise what we expect from the above example scenario as follows.

$$
\begin{aligned}
TC\,(a,\ p\ \ ;\ \ q) = \textbf{let}\ & \mathit{fst\_only} == TC\,(a,\ p) \setminus TC\,(a,\ q);\\
& \mathit{snd\_only} == TC\,(a,\ q) \setminus TC\,(a,\ p) \bullet\\
& \left(\begin{array}{ll} & \mathit{fst\_only}\\ \cup & \{\,(u,\ wp^*\,(p,\ c)) \mid (u,\ c) \in \mathit{snd\_only}\,\}\\ \cup & \mathit{merge}\ p\ \mathit{fst}\ \mathit{snd}\end{array}\right)\\
\textbf{where}\ & \mathit{fst} = TC\,(a,\ p)\\
& \mathit{snd} = TC\,(a,\ q)
\end{aligned}
$$

There are three cases to consider: 1) if a appears only in the first phase, then we collect its test cases via $TC\,(a,\ p) \setminus TC\,(a,\ q)$ without adding further constraints; 2) if a appears only in the second phase, then we require that the intent of the first phase (i.e. p) does not prevent its test cases, if any, from happening; and 3) if a appears on both phases, then we may construct for it a new usage pattern and hence a new test case accordingly.

The above auxiliary function

$$\mathit{merge} : \mathit{Predicate} \to \mathbb{P}\ \mathit{TCase} \to \mathbb{P}\ \mathit{TCase} \to \mathbb{P}\ \mathit{TCase}$$

takes as inputs the formal specification of the first phase (i.e. p), as well as the test cases collected from both phases (i.e. *fst* and *snd*).

```
merge _ []       snd = snd
merge p (tc:tcs) snd = (foldr ((++).(comp p tc)) [] snd) `union` (merge p tcs snd)
```

The merge process is iterative and specified using `foldr` (as in function *reduce*). For each test case *tc* in the first phase, we compare it against each test case in the second phase to see if they can be composed to make one of the $*\mathit{ThenDef}$ or $\mathit{DefThen}*$ patterns, where $*$ is either *CUse* or *PUse*. We specify this composition process in an auxiliary function

$$\mathit{comp} : \mathit{Predicate} \to \mathit{TCase} \to \mathit{TCase} \to \mathbb{P}\ \mathit{TCase}$$

As we have seen from the above example, if attribute *reservations* is defined in the first phase and used for predicate in the second phase, then we compose a new *DefThenPUse* pattern to test:

$$comp\, fst\, (Def\ (a),\ p)\, (PUse\, (a),\ p') = \{\ (\ DefThenPUse\, (a),\ p \wedge wp^*(fst,\ p')\ )\ \}$$

The precondition of this combined test case has to ensure two things: 1) the constraint p of $Def\ (a)$ in the first phase is satisfied; and 2) the intended state transformation in the first phase—as specified in *fst*—can establish the constraint p' of $PUse\, (a)$ in the second phase.

Here our notion of the weakest precondition above (wp^*) is consistent with wp in Section 3.3. Both calculations are with respect to a desired property to be established. On the one hand, the calculation of wp takes as input a program of substitutions, whose primitive updates are assignments. On the other hand, wp^* takes as input an abstract specification of operation (Section 2.5.1). The three base cases of wp^* reflect the intended effects on state:

$$\begin{array}{lcl} wp^*\, (x'\ \texttt{=}\ e,\ inv) & = & inv\, [\, e\, /\, x\,] \\ wp^*\, (e\ \texttt{:}\ \ x',\ inv) & = & inv\, [\, x \cup \{e\}\, /\, x\,] \\ wp^*\, (e\ \texttt{/:}\ \ x',\ inv) & = & inv\, [\, x \setminus \{e\}\, /\, x\,] \end{array}$$

We intend to interpret a negated predicate as side-effect free and use it directly as a guard constraint.

$$wp^*\, (\texttt{not}\ \ p,\ inv) = \neg\, p \wedge inv$$

And other inductive cases are defined in a manner similar to that of wp in Section 3.3.

$$\begin{array}{lcl} wp^*\, (p\ \texttt{=>}\ \ q,\ inv) & = & p \wedge wp^*\, (q,\ inv) \\ wp^*\, (p\ \texttt{;}\ \ q,\ inv) & = & wp^*\, (p,\ wp^*\, (q,\ inv)) \\ wp^*\, (p\ \texttt{\&}\ \ q,\ inv) & = & wp^*\, (p\ \texttt{;}\ \ q,\ inv) \wedge wp^*\, (q\ \texttt{;}\ \ p,\ inv) \end{array}$$

So far our calculation of TC extracts attribute usage patterns in the scope of a single predicate method. It is already sufficient for the developers to exploit the preconditions that are generated for these patterns and decide if individual methods are consistent. However, we will also want to explore usage patterns that exist across

methods. This is particularly useful when instances of workflows are executed upon the system. Rather than enumerating all possible interleaving orders, the developers may be more interested in those scenarios, i.e. traces, where usage patterns are observed through pairs of operations. On this account, we suggest in the next section how these *intra*-method usage patterns should be composed to create *inter*-method patterns.

5.2.2 Inter-method Usage Patterns

After detecting the *intra*-method usage pattern(s) for each attribute in every method, we then consider—as summarised in Figure 5.2—how to identify usage patterns that are achieved through pairs of methods.

m_1 \ m_2	*def-only*	*def-then-use*	*use-then-def*	*use-only*
def-only	C1: $\times$	C2: m_2	C4: $m_1 ; m_2$	C5: $m_1 ; m_2$
def-then-use		C3: m_1, m_2	C6: m_1, $m_1 ; m_2$	C7: m_1, $m_2 ; m_1$
use-then-def			C8: $m_1 ; m_2$, $m_2 ; m_1$	C9: $m_1 ; m_2$
use-only				C1:$\times$

Figure 5.2: Constructing Inter-Method Attribute Usage Patterns

Given an attribute a and two methods m_1 and m_2 we have a list of cases:

- C1: a is either *def-only* or *use-only* in both m_1 and m_2. In this case $m_1 ; m_2$ is not worth testing as there is no inter-method def-use association for a.
- C2: a is *def-only* in m_1 and *def-then-use* in m_2. In this case $m_1 ; m_2$ is not worth testing as the later definition makes the inter-method du-association not def-clear. But m_2's intra-method usage is worth testing.
- C3: a is *def-then-use* in both m_1 and m_2. In this case $m_1 ; m_2$ is not worth testing as it is not def-clear. However, m_1 and m_2 may be tested individually for their intra-method du-associations.
- In other cases either $m_1 ; m_2$ or $m_2 ; m_1$ (e.g. C7 for a *use-then-def* pattern) is worth testing for its inter-method du-associations:

- C4: a is *def-only* in m_1 and *use-then-def* in m_2.
- C5: a is *def-only* in m_1 and *use-only* in m_2.
- C6: a is *def-then-use* in m_1, *use-then-def* in m_2. In this case m_1 is also worth testing individually for its intra-method du-association.
- C7: a is *def-then-use* in m_1 and *use-only* in m_2. In this case m_1 is also worth testing individually for its intra-method du-association.
- C8: a is *use-then-def* in m_1 and m_2. In this case both orders of composing m_1 and m_2 are worth testing.
- C9: a is *use-then-def* in m_1, *use-only* in m_2.

All other six blank cells in Figure 5.2 are symmetric to cases C2 to C6 and C9 by swapping the identifiers of methods.

Example (*Inter-Method Usage Pattern*) We consider the specifications[2] of methods *reserve* and *allocate* (Appendix A.2). As attribute *allocations* is used in method *reserve* for guarding the intended update and it is modified in method *allocate*, we construct—according to case C5 in Figure 5.2—a method sequence *allocate*;*reserve* in which the usage pattern *DefThenPUse* (*allocations*) may occur. □

The method sequence $\langle$ *allocate*, *reserve* $\rangle$ that we derived from the above example "may" demonstrate the usage pattern *DefThenPUse* (*allocations*), provided its associated precondition is satisfied. However, in the presence of concurrency, there will possibly be interference that happens after *allocate* completes but before *reserve* is initiated. For example, another instance of *allocate* that occupies the last available room in the hotel may just cause *reserve* to be blocked. On this account, our calculation of preconditions for *inter*-method usage patterns ought to incorporate the effect of interference from operations or workflows that are known to be running.

[2]We do not provide the predicative specifications of both methods in Appendix A.2, but their completed version—in our extended GSL notation—is conceptually similar to its predicative counterparts and thus would serve the purpose of demonstration.

We will address this in Section 5.3.3—to reach that point, we will first introduce a methodology of testing workflows in the next section.

5.3 A Testing Methodology for Parallel Workflows

We have studied how data usage patterns of attributes may be extracted within individual model operations (Section 5.2.1) or in between (Section 5.2.2) operations—all of them are described as first-order predicates. What remains to be completed is to place these results in the execution context of reality—one that allows interference on the shared system state.

We will first introduce the design of our methodology for testing workflows: they are run in parallel and thus interfere with each other (Section 5.3.1). In order for the developers to test (the behavioural features of) their design models, we suggest three kinds of heuristics—each of them denotes a particular kind of use-case scenario—that can facilitate the reduction of our analysis space, i.e. the number of trace orders to consider. (Section 5.3.2). Finally, we will transform each use-case scenario—i.e. a trace that is chosen on the basis of the heuristic specified—into a precondition constraint that reflects how the current model specification constrains its availability (Section 5.3.3).

5.3.1 Design

We first characterise the interface—i.e. input and output—of our methodology. Users are expected to specify an input

$$in : \mathbb{P}\ Wfw \times Scenario \times Predicate$$

which is a 3-tuple (W, S, G) where

- $W \in \mathbb{P}\ Wfw$ specifies the set of workflows that we intend to run in interleaving (or in parallel with no synchronisation). This set is essentially a *closure* (Section 4.6) meaning that no further interference is allowed beyond W. In other

words, we are interested in analysing the behaviour of $\|\!|_{w \in W} w$.

– $S \in \mathit{Scenario}$ denotes a specification of use-case scenario(s) to test (Section 5.3.2). This effectively restricts our attention to a—hopefully substantially smaller—subset of $\mathcal{T}[\![\|\!|_{w \in W} w]\!]$. We will refer to a member of *Scenario* either as a restriction criterion, or as a use-case scenario(s) specification. Our calculation of preconditions (Section 5.3.3) concentrates upon this subset of interleaving traces accordingly.

– $G \in \mathit{Predicate}$ indicates a predicate constraint or goal that should be satisfied upon the completion of executing $\|\!|_{w \in W} w$. In this thesis we focus on the completion of parallel workflows and thus specify $G = \mathit{true}$; however, our results apply to cases where G represents a stronger constraint on system attributes.

Given an input as specified above, our approach generates an output

$$\mathit{out} : \mathit{Trace} \nrightarrow \mathit{Predicate}$$

which is a partial function. The domain of *out* is the set of traces—each of which is drawn from the interleaving behaviour of parallel workflows in consideration and restricted by the input S. More precisely, we have $\operatorname{dom} \mathit{out} = [\![S]\!]^{W'}_{\mathit{Scenario}} \cap \mathcal{T}[\![W']\!]$, where $W' = \|\!|_{w \in W} w$. The expression $[\![S]\!]^{W'}_{\mathit{Scenario}}$ assigns—via the semantic function $[\![\_]\!]^{-}_{\mathit{Scenario}}$ (Section 5.3.2)—a formal meaning to the input restriction criterion S as a subset of $\mathcal{T}[\![W']\!]$.

Each member in the range of *out* is a precondition constraint, satisfying which would guarantee that the corresponding interleaving order—i.e. use-case scenario of interest—successfully completes. More precisely, we have $\forall\, t \in \operatorname{dom} \mathit{out} \bullet \mathit{out}\,(t) = \operatorname{dom}(\mathit{apply}\,(t) \rhd G)$—where $\rhd$ is the range restriction operator, and the expression $\mathit{apply}\,(t)$ (Section 4.6) formalises the effect of trace t on the (shared) state as a binary relation. If we interpret each precondition constraint as its set of satisfying states, then we would expect $\mathit{out}\,(t)$ to be all (starting) states from which the relation $\mathit{apply}\,(t)$ maps to those that satisfy G. Furthermore, as $\mathit{apply}\,(t)$ gives an abstract

interpretation of its effect on the state, the mechanical calculation of $out\,(t)$ involves a systematic application (Section 5.3.3) of our wp^* on predicates as introduced in Section 5.2.1.

Given this output function, we represent the resulting test suite as its inverse. More precisely, we define $TS = \{(c, t) \mid t \in \mathrm{dom}\, out \wedge c = out\,(t)\}$. Each test case consists of a precondition constraint c and its associated trace t. Each trace t suggests the order in which we should invoke the operations—it essentially simulates a possible interleaving execution, i.e. a use case, of the input workflows. The constraint c characterises the precise availability of the use-case scenario that t describes, and it advises as to how we should set the data values in order to configure the starting state of the test case.

We have completed presenting the design of our testing methodology for testing parallel workflows. In the next section, we will elaborate on how we specify each kind of use-case scenario in order to concentrate the developers' tests on their models.

5.3.2 Specifying Use-Case Scenarios

In our testing methodology (Section 5.3.1) we intend for the developers to manually inspect each generated operation trace—which characterises a use-case scenario of interest—and its associated precondition constraint. However, the developers may not be interested in manually deciding the consistency of all possible behaviours that result from executing these workflows in parallel—even in the case of a system like our hotel reservation system (Appendix A.2) which contains smaller numbers of declared methods and workflows than other information systems. More formally, the feasibility of any analysis upon parallel workflows is predicated upon the number and complexity of the workflows involved. In the simplest case of a pair of sequential workflows, with m and n operations, the number of orders in which their operations might interact (and hence interfere with one another) is $\binom{n+m}{n}$—the number of ways for choosing n out of $n + m$ slots. Even when m and n are as small as 10, for example, the amount of interleaving orders to consider already reaches $184,756$. Moreover, each additional

use of interleave ($|||$) or choice ($\Box$) combinator further multiplies the possibilities to be considered.

To reduce the number of trace orders for consideration—from these traces we perform the calculation of preconditions and the generation of test cases—we propose three kinds of use-case scenarios to which the developers restrict their attention when testing their models. The first is on the occurrence of certain pattern of events; this pattern is described as a workflow. This approach is similar to that of the parallel operator ($\|$) in CSP [141]—events initiated from distinct workflow instances but with identical names are synchronised on their occurrences. The second is on the data flow of a particular attribute that occurs in a pair of operations in the trace. The third is on the notion of events that *commute*—we are not concerned about the order in which they appear in the trace. The specification of use-case scenarios is compositional. We may describe a scenario by combining the specifications of others. We define the abstract syntax of specifying use-case scenarios as follows. Here we also refer to the abstract syntax of *Usage* (Section 5.2.1) from our adapted data-flow analysis upon declarative methods.

$$\begin{aligned} \mathit{Scenario} ::= {} & \mathit{PROJ}\langle\!\langle \mathit{Wfw} \rangle\!\rangle \mid \mathit{DF}\langle\!\langle \mathit{Usage} \rangle\!\rangle \mid \mathit{Commute}\langle\!\langle \mathbb{P}(\mathbb{P}\,\Sigma) \rangle\!\rangle \\ \mid {} & \mathit{NOT}\langle\!\langle \mathit{Scenario} \rangle\!\rangle \\ \mid {} & \mathit{AND}\langle\!\langle \mathit{Scenario} \times \mathit{Scenario} \rangle\!\rangle \\ \mid {} & \mathit{OR}\langle\!\langle \mathit{Scenario} \times \mathit{Scenario} \rangle\!\rangle \end{aligned}$$

We formally interpret the above scenario specification by mapping it to the set of satisfying traces, when given a set of workflows W to be executed in interleaving. We define a semantic function

$$[\![\_]\!]^{-}_{\mathit{Scenario}} : \mathit{Wfw} \to \mathit{Scenario} \to \mathbb{P}\ \mathit{Trace}$$

and its definition is as follows. Traces of W that satisfy both S_1 and S_2 are the intersections of those that satisfy S_1 and those that satisfy S_2, their union for traces that satisfy either of the restrictions, and the difference between them for traces that

do not satisfy the restriction.

$$\begin{aligned} [\![AND\,(S_1, S_2)]\!]^W_{Scenario} &= [\![S_1]\!]^W_{Scenario} \cap [\![S_2]\!]^W_{Scenario} \\ [\![OR\,(S_1, S_2)]\!]^W_{Scenario} &= [\![S_1]\!]^W_{Scenario} \cup [\![S_2]\!]^W_{Scenario} \\ [\![NOT\,(S)]\!]^W_{Scenario} &= \mathcal{T}[\![W]\!] \setminus [\![S]\!]^W_{Scenario} \end{aligned}$$

We now consider the three testing scenarios that we may specify as the base cases of *Scenario*—event projections, attribute data flows, and commutative events.

Testing Scenario—Event Projections

For a trace $t \in \mathcal{T}[\![W]\!]$ to satisfy the projection constraint $PROJ\,(W')$—workflow W' describes the pattern of events that the developers expect to occur—at least one of $t' \in \mathcal{T}[\![W']\!]$ must equal to the "projected" version of t.

$$[\![PROJ\,(W')]\!]^W_{Scenario} = \{ t \mid t \in \mathcal{T}[\![W]\!] \wedge t \upharpoonright (\Sigma\, W') \in \mathcal{T}[\![W']\!] \}$$

The operator $s \upharpoonright r$ selects a (not necessarily consecutive) subsequence from sequence s that contains items solely from the range set r. Let us consider an example on our hotel reservation system.

Example (*Testing Scenarios—Event Projections*) Consider (a sketch of) the specification of a booking workflow:

$$Booking = \big(\ (\ldots) \,\Box\, (\ldots \rightarrow cancel \rightarrow \ldots)\ \big)$$

where there are two branches of choices for users of the hotel reservation system. Details of the first branch are irrelevant for our demonstration here, but it contains no occurrence of the method *cancel*. On the other hand, the second branch provides users of the system with an option to cancel the booking. Say the developers wish to explore the model behaviour when 100 instances of *Booking* are run in parallel. They may be interested in testing their system model by considering only those scenarios where a cancellation occurs in the booking process, or those where the booking does not fail. The former is tested by a 3-tuple input $(\ |||_{i \in 1..100}\ Booking,\ PROJ\,(cancel \rightarrow skip),\ true)$

and the latter is tested by negating the scenario criterion to $NOT\,(PROJ\,(cancel \rightarrow skip))$. Our use of event projection is similar to the synchronous parallel operator in CSP [141]—but here we exploit its usage in the context of model-based testing (of design models). □

Testing Scenario—Attribute Data Flows

A trace t satisfies the data usage criterion $DF\,(u)$ if at least one of its contained operations demonstrates the usage pattern u. Let us first consider an example on our hotel reservation system.

Example (*Testing Scenarios—Attribute Data Flows*) We elaborate on the (sketched) booking workflow that we discussed in the previous example of event projections:

$$Booking = \big(\ (\ldots \rightarrow allocate \rightarrow \ldots \rightarrow reserve \rightarrow \ldots)\ \Box\ (\ldots \rightarrow cancel \rightarrow \ldots)\ \big)$$

where the first branch contains occurrences of methods *allocate* and *reserve*. We already learnt from the example in Section 5.2.2 that there exists a *inter*-method data pattern $DefThenPUse\,(allocations)$ between *allocate* and *reserve*. Similar to our projection of *cancel* in the previous example, the developers may instead be interested in exploring the data flow of attribute *allocations*, and we can test this by an input $(\ |||_{i \in 1..100}\ Booking,\ DF\,(DefThenPUse\,(allocations)),\ true)$. □

To formalise what we expect from the above example, we have

$$[\![DF\,(u)]\!]^{W}_{Scenario} \quad = \quad \{t \mid t \in \mathcal{T}[\![W]\!] \wedge hasUsage\,(u, t)\}$$

Let us consider the case of usage $DefThenPUse\,(a)$. Function $hasUsage\,(u, t)$ is a boolean query

$$hasUsage : Usage \times Trace \rightarrow Boolean$$

that has one of its rules determining if attribute a has a *DefThenPUse* usage pattern, and its definition is built atop our calculation of *intra*-method usage patterns (Sec-

tion 5.2.1). We require not only that attribute a is defined and used for predicate in two methods, but also that a is def-clear between them, i.e. there is no re-definition of a. More precisely, we have:

$$hasUsage\,(DefThenPUse\,(a),\ t) = \left(\begin{array}{ll} \exists\, i, j : \mathrm{dom}\, t;\ c_1, c_2 : Predicate \mid i < j \bullet & \\ \qquad (Def\,(a),\ c_1) \in TC\,(a,\ t\,(i)) & \\ \quad \wedge\ (PUse\,(a),\ c_2) \in TC\,(a,\ t\,(j)) & \\ \quad \wedge\ isDefClear\,(i,\ j) & \end{array} \right)$$

where $isDefClear\,(i,\ j)$ is to ensure that none of the methods $t\,(i),\ \ldots,\ t\,(j)$ changes the value of attribute a. More precisely,

$$isDefClear\,(i,\ j) = \left(\begin{array}{l} \forall\, k \mid i < k < j \bullet \\ \qquad \nexists\, c : Predicate \bullet (Def\,(a),\ c) \in TC\,(a,\ t\,(k)) \end{array} \right)$$

It is obvious that from the above style of definition, when the restricted set of traces is generated, we know automatically which two methods to test via indices i and j.

Traces that we generate from a DF criterion, e.g. $DF\,(DefThenPUse\,(a))$, are guaranteed—via *isDefClear*—to contain no interference on the attribute a under test. However, there is still a possibility of interference on values of other attributes that causes the data usage pattern to fail to be demonstrated. Consider our example above—when multiple instances of *Booking* are run in parallel, operations that modify the same shared state may interfere before, in-between, or after the execution of *allocate* and *reserve*—each case may cause the data pattern not to be able to successfully complete. This is exactly the concern that we will address in Section 5.3.3.

Testing Scenario—Commutative Events

The $Commute\,(S)$ criterion—where $S = \bigcup_{i \in 1..n} s_i$—specifies patterns of events whose execution orders in the test cases do not concern the developers. Instead, for each event set s_i that is included in the specification of $Commute\,(S)$, the developers will be satisfied by exploring just a *representative* test case, which includes events that are drawn from s_i. Allowing the developers to specify such "don't-care" conditions facilitates our process of test case generation—for each subset of events that is drawn

from s_i, there is no need for us to enumerate all possible interactions of these events, but to pick one suffices. In general, given a workflow W to analyse and the huristics $Commute\,(\{s\})$ to apply, for each trace $tr \in \mathcal{T}[\![W]\!]$, applying the restriction reduces from $\mid tr \mid!$ traces to $\frac{|tr|!}{|(\mathrm{ran}\ tr) \cap s|!}$ traces to be considered. When there are multiple commutative event sets specified, assuming that they are disjoint, then the extent of reduction on tr is multiplied by those of all event sets that overlap with the range of tr. Let us consider an example on our hotel reservation system.

Example (*Testing Scenario—Commutative Events*) We will consider the interleaving of seven one-step workflows; each of them represents the execution of a transaction[3] upon the hotel reservation system.

$$(deposit \rightarrow skip \;|||\; register \rightarrow skip \;|||\; reserve \rightarrow skip \;|||\; allocate \rightarrow skip)$$
$$\Box$$
$$(cancel \rightarrow skip \;|||\; deallocate \rightarrow skip \;|||\; withdraw \rightarrow skip)$$

If for the first branch of choice the developers are not concerned about whether or not a traveller registers in the hotel before they make a reservation, they will supply an input with the restriction criterion $Commute\,(\{\,\{deposit,\ register\}\,\})$. We then observe that in each of the two example sets of traces below

$$\{\langle \boxed{register}, \boxed{deposit}, reserve, allocate\rangle, \langle \boxed{deposit}, \boxed{register}, reserve, allocate\rangle\}$$
$$\{\langle \boxed{register}, reserve, allocate, \boxed{deposit}\rangle, \langle \boxed{deposit}, reserve, allocate, \boxed{register}\rangle\}$$

the projection to $\{reserve, allocate\}$—i.e. events that do not commute—gives two equal sequences. For the first set, we may choose—nondeterministically—the trace $\langle\, register, deposit, reserve, allocate\,\rangle$ as the representative to test the model. If the developers are also not concerned about whether or not a traveller is charged for its cancellation before their allocated room is freed up by the hotel, they will specify a stronger criterion $Commute\,(\{\,\{deposit,\ register\},\ \{deallocate,\ withdraw\}\,\})$ to gain an even smaller analysis space for generating the test suite. Furthermore, we observe

[3]See in Appendix A.2 methods *register*, *reserve*, *cancel*, *allocate*, and *deallocate* in class *Hotel*, and methods *deposit* and *withdraw* in class *Account*.

that the set of traces that results from the second branch of the workflow

$$\left\{ \begin{array}{l} \{\langle cancel, deallocate, withdraw\rangle\},\ \{\langle cancel, withdraw, deallocate\rangle\}, \\ \{\langle deallocate, cancel, withdraw\rangle\},\ \{\langle withdraw, cancel, deallocate\rangle\}, \\ \{\langle deallocate, withdraw, cancel\rangle\},\ \{\langle withdraw, deallocate, cancel\rangle\} \end{array} \right\}$$

has to be enumerated in our process of test case generation. They are not affected by the above criterion of $Commute\,(\{\,\{deposit,\ register\},\ \{deallocate,\ withdraw\}\,\})$. □

To formalise our expectation as demonstrated in the above example, we define the rule of our semantic function $[\![\_]\!]^{-}_{Scenario}$ on *Commute*. The base case is when there is no set of events that commute—there is no restriction to the workflow under analysis.

$$[\![Commute\,(\{\})]\!]^{W}_{Scenario} = \mathcal{T}[\![W]\!]$$

The recursive case[4] is when there is at least one set of events which the developers think should commute.

$$[\![Commute\,(\{s\} \cup S)]\!]^{W}_{Scenario} = \bigcup\{rep\,(ts) \mid ts \in cts\} \cup ([\![Commute\,(S)]\!]^{W}_{Scenario} \setminus \bigcup cts)$$

$$\textbf{where } cts = \left\{ \begin{array}{l} ts \mid ts \subseteq [\![S]\!]^{W}_{Scenario} \\ \quad \wedge \\ \quad \left(\begin{array}{l} \forall\, t_1, t_2 : ts \bullet\ t_1 \upharpoonright s \in perm\,(t_2 \upharpoonright s) \\ \qquad\qquad \wedge \\ \qquad\qquad t_1 \upharpoonright (\Sigma \setminus s) = t_2 \upharpoonright (\Sigma \setminus s) \end{array} \right) \end{array} \right\}$$

Each member *ts* in set *cts* is a set of event traces. The projections of all event traces in *ts* to the commutativity event set *s* are permutations of each other (i.e. $t_1 \upharpoonright s \in perm\,(t_2 \upharpoonright s)$). Also, the projections of all traces in *ts* to the complement of *s* (i.e. $\Sigma \setminus s$) are equivalent to each other (i.e. $t_1 \upharpoonright (\Sigma \setminus s) = t_2 \upharpoonright (\Sigma \setminus s)$). From each member of *cts*—which is a set of event traces—we select a representative trace from it—via *rep*—for testing. Equally importantly, we include also those traces that do not involve at all the events that commute (i.e. $[\![Commute\,(S)]\!]^{W}_{Scenario} \setminus \bigcup cts$).

[4]Here we assume the existence of two functions. Function $perm : \Sigma^* \to \mathbb{P}\,\Sigma^*$ returns the set of permutations of a given trace. Function $rep : \mathbb{P}\,\Sigma^* \to \Sigma^*$ selects—nondeterministically—a trace to represent the given trace set (i.e. $rep\,(\{\}) = \{\}$ and $rep\,(ts) \in ts$, for $ts \neq \{\}$).

We have explored how the specification of test criteria—particularly via event projections and commutative events—may help the developers reduce the magnitude of the set of use-scenarios (and their associated precondition) that they will need to manually inspect. The generation of test cases for the data-flow use-case scenarios would require extra care. In between methods that may demonstrate an attribute usage pattern, operations initiated from other instances of workflows may interfere with the state in such a way that the pattern under test cannot successfully complete. We will address this issue in the next section to complete the presentation of our testing methodology for parallel workflows.

5.3.3 Preconditions for Inter-Method Usage Patterns

As noted in Section 5.2.2, we assume for each method sequence $\langle m_1, m_2 \rangle$ that we generate—e.g. $\langle allocate, reserve \rangle$—a restricted, sequential context of execution in order for it to exercise the attribute usage pattern in question. More precisely, there is no other method or workflow executed in interleaving with $\langle m_1, m_2 \rangle$ and hence no possibility of interference before, in between, and after the execution of m_1 and m_2. However, such assumption prevents us from simulating the common reality—a concurrent context.

In this section we will explore the consequence of relaxing such unrealistic assumption. More precisely, we want to test scenarios as outlined below:

$$before \frown \langle m_1 \rangle \frown middle \frown \langle m_2 \rangle \frown after$$

where *before*, *middle*, and *after* represent, respectively, the interference on state that occurs before, in between, and after the execution of $\langle m_1, m_2 \rangle$. Each interference can be any interleaving order of activities from other workflows. These three scenarios are worth testing because: 1) the interference of *before* may even prevent $\langle m_1, m_2 \rangle$ from being executed; 2) the interference of *middle* may prevent m_2 from being executed after m_1; and 3) the execution of $\langle m_1, m_2 \rangle$ itself may be the interference for *after*. That is, after completing $\langle m_1, m_2 \rangle$, the resulting state may cause the execution of

after to get stuck in the middle, or not even to be initiated. All of these three scenarios are important and should be reflected in our test suite.

Therefore, each test case $(c, t) \in TS$ as described in Section 5.3.1 may reflect a number of restriction criteria (Section 5.3.2) that are concerned with the data flows. Suppose that test trace t has exactly the generic form as we outlined above. Let us consider an example in our hotel reservation system.

Example (*Preconditions for Inter-Method Usage Patterns*) Say we are considering a trace t conforming to the form of $\mathit{before} \frown \langle \mathit{allocate} \rangle \frown \mathit{middle} \frown \langle \mathit{reserve} \rangle \frown \mathit{after}$. We also obtain from Section 5.2.2 that in method *allocate* we have pattern $(\mathit{Def}\ (\mathit{allocations}),\ p)$ and in method *reserve* we have pattern $(\mathit{PUse}\ (\mathit{allocations}),\ q)$. The exact definitions of *before*, *middle*, *after*, p, and q are irrelevant for our purpose of demonstration. As a result, $\mathit{out}\,(t)$ as described in Section 5.3.1 should be calculated as $\mathit{wp}^*\,(\fatsemi\ \mathit{before},\ p) \wedge \mathit{wp}^*\,((\fatsemi\ \mathit{before})\ ;\ \mathit{allocate}\ ;\ (\fatsemi\ \mathit{middle}),\ q) \wedge \mathit{wp}^*\,(\fatsemi\ t,\ \mathit{true})$. We use $\fatsemi$ to combine, via relational composition $(;)$, all operations in a given trace t into a single operation with the effect $\mathit{apply}\,(t)$ on the shared state. The three conjuncts address our three concerns on interference as outlined above. The first conjunct ensures that the execution of *before* will not prevent method *allocate* from exercising the usage $\mathit{Def}\,(\mathit{allocations})$. Similarly, the second conjunct ensures that the execution up to method *reserve* will not prevent it from exercising the usage $\mathit{CUse}\,(\mathit{allocations})$. Finally, we ensure that the test trace t does determinate and establish the user-specified goal *true*, i.e. successful completion. □

To implement our expectation from the above example is straightforward and not considered as a contribution of this thesis. We have now completed the presentation of our testing methodology for parallel workflows. Each application of this testing process may allow the developers to detect inconsistent precondition constraints for their associated use-case scenario. This suggests that our testing process ought to be applied iteratively. Upon fixing the relevant model constraints and inspecting the re-generated test cases, the developers may or may not decide to change their design.

In the next section, we will discuss how our results in Chapter 3 may provide the developers with a formal guarantee when revising their behaviour models.

5.4 Revising Behavioural Specifications

The developers are ideal users of our testing methodology (Section 5.3) as they have had their understandings about the requirements throughout the course of their development of the design model under test. Consequently, each generated test case will enable the developers to examine if the scenarios that they are concerned about—represented as an operation trace—are associated with precondition constraints that are consistent with how they expect the system to function. The predicative specification of a BOOSTER system model consists of invariants of the system and classes, as well as preconditions and postconditions of methods. These three categories of constraints form the basis of our precondition calculation in Section 5.3.3 and thus should be the location in the model the developers consider fixing, should they consider any of the generated test cases inconsistent.

Randomly altering the predicative specification of the model may cause the generated test suite to seem consistent. However, when the model is subject to a series of such evolutions, it should be to the benefits of developers to be guaranteed that each subsequent evolution of their model description results in a version that is more refined, if not equivalent, on its behaviours. Therefore, we suggest that developers take advantage of the repository of equivalence and algebraic laws that we listed in Table 3.1 (on page 62). Our hypothesis is that if the intention is to perform a refactoring of the model without changing the external observation of its behaviours, then the developers should consult the equivalence laws. On the other hand, if the purpose is to resolve any detected inconsistency, then the developers ought to attempt to find applicable patterns of refinement. Admittedly, if the original model is simply erroneous with respect to the requirements—e.g., a misuse of the logical or relational operators, a misunderstanding of the requirements, etc.—then it is possible

that the fix has to result in a version of the model that has no logical connection to the original. We already considered examples on the scenarios in which we may apply each rewriting rule as listed in Table 3.1. In cases where the resulting specification is equivalent to the original, we will expect the re-generated precondition to be also equivalent to its previously generated value. In cases where the resulting specification refines the original, however, we will expect the re-generated precondition to entail the previous one. Let us consider another example.

Example (*Revising Specifications & Re-Generating Preconditions*) The developers have tested the method $(\#reservations < 1000) \vee reserve$ on the use case of $Def\,(reservations)$ data pattern. As method *reserve* was not guarded by an antecedent, the test result was the precondition *true*. The developers may think that it should be the case that before and after the effect of *reserve* is applied, the number of reservations will be kept above certain lower limit; otherwise, the current billing mechanism in its specification would be incorrect. As a result, the developers will apply Law 3.5.17 `(d)` in Section 3.5 and obtain a refined specification: $(\#reservations < 1000) \vee ((\#reservations \geq 1000) \wedge reserve)$. They can expect that the re-generated precondition will be stronger: it has to ensure that within the use-case scenario trace in question, methods that occur before *reserve* will establish a state for $\#reservations \geq 1000$ to hold before and after it is applied. Moreover, methods that occur after *reserve* will not be blocked because of this new invariant that we imposed on *reserve*. Similar arguments apply to all other examples that we already considered in Sections 3.5 and 3.6. □

The value of our developed rewriting rules in Table 3.1 is two-fold: 1) when the developers wish to revise their specifications, they can re-use the interpretations that we have already demonstrated in Section 3.5; and 2) they can be ensured that the re-generated precondition will accurately reflect their intent of rewriting. We state two theorems to address formally how the developers may select the rewriting rules. Their proofs are included in Appendix D.

Theorem 5.4.1 (***Applying Equivalence Laws***) By following an equivalence rule in Table 3.1 (on page 62), to perform a refactoring on the predicative specification of some BOOSTER method p, it is the case that the resulting predicate q preserves in its behaviour what might be observed from the behaviour of p. □

Theorem 5.4.2 (***Applying Refinement Laws***) By following a refinement rule in Table 3.1 (on page 62), to perform a change on the predicative specification of some BOOSTER method p, it is the case that whatever observation we might make of the behaviour of the resulting predicate q is also possible for the behaviour of p. □

5.5 Example

We demonstrate our testing methodology on the hotel reservation system model (Appendix A.2).

5.5.1 User-Specified Methods

Say the method *allocate* presented in Appendix A.2 has been completed from the user-specified predicate:

```
allocate { r? : extent(Reservation) &
           r?.status = "confirmed"  &
           m? : extent(Room)        &
           # allocations < 100      &
           a! : extent(Allocation)
           =>
           Forall x : r?.dates @ (x : a!.dates') &
           a! : allocations'                     &
           a! : m?.allocations'
         }
```

where for each bi-directional association being updated, specifying a change of one end is sufficient for the compiler to add additional updates as we see in Section 2.5.2. Method *deallocate* is defined in a similar manner. Our testing approach will analyse such abstract descriptions, rather than the substitution programs as seen in Section A.2.

5.5.2 Attribute Usage Patterns

5.5.2.1 Intra-Method

Say we wish to test the usage pattern of attribute *allocations*. Its value is used to determine the truth of guards of both methods *allocate* and *deallocate*. As both methods contain just a single phase, the predicate use of *allocations* will not be constrained at all. More precisely,

$$(PUse\,(allocations),\ true) \in TC\,(allocations,\ allocate)$$
$$\wedge\ (PUse\,(allocations),\ true) \in TC\,(allocations,\ deallocate)$$

On the other hand, definitions on the value of *allocations* are guarded in both methods, and hence their associated constraints will be stronger. More precisely,

$$(Def\,(allocations),\ c) \in TC\,(allocations,\ allocate)$$

where $c = r?.status = confirmed \wedge \#\,allocations < 100$, and similarly for the calculation of $TC\,(allocations,\ deallocate)$.

5.5.2.2 Inter-Method

According to Section 5.2.2, we generate a method sequence $\langle allocate,\ deallocate\rangle$ that demonstrates a data usage pattern $DefThenPUse\,(allocations)$, by composing $Def\,(allocations)$ in method *allocate* and $PUse\,(allocations)$ in method *deallocate*.

To calculate a guard constraint that ensures this *inter*-method usage pattern will be exercised at runtime—assuming a sequential context of execution—we treat the sequence $\langle allocate,\ deallocate\rangle$ as if it were a stand-alone predicate method. More precisely, $(DefThenPUse\,(allocations),\ c') \in TC\,(allocations,\ (allocate\ \ ;\ \ deallocate))$, where $c' = c \wedge wp^*\,(allocate, true)$ and c was defined in Section 5.5.2.1. That is, constraint c' insists that we are able to exercise the definition of *allocations*, and that after executing method *allocate* we are still able to exercise the predicate use of *allocations* in method *deallocate* (which is constrained by nothing as calculated).

5.5.3 Workflows

We will consider workflows, that are similar to those considered in Section 4.8, to book a hotel room, and to cancel or change an existing booking.

```
MakeBooking {
  r = r! & reserve ->
    ( r? = r & confirm ->
        r? = r & allocate -> skip
      []
      r? = r & cancel -> skip ) }
```

```
CancelOrChangeBooking {
  ( r = r? & cancel ->
      r? = r & deallocate -> skip )
  []
  ( r = r? & change ->
      r? = r & deallocate ->
        r? = r & allocate -> skip ) }
```

5.5.4 Specifying Use-Case Scenarios

We put both workflows in interleaving: $W = \mathit{MakeBooking} \mid\mid\mid \mathit{CancelOrChangeBooking}$. The trace set of *MakeBooking* is $\{\langle \mathit{reserve}, \mathit{confirm}, \mathit{allocate}\rangle, \langle \mathit{reserve}, \mathit{cancel}\rangle\}$, and it is $\{\langle \mathit{cancel}, \mathit{deallocate}\rangle, \langle \mathit{change}, \mathit{deallocate}, \mathit{allocate}\rangle\}$ for *CancelOrMakeBooking*. Therefore, the total number of interleaving orders of both workflows is $\binom{5}{3}+\binom{6}{3}+\binom{4}{2}+\binom{5}{2} = 46$. This reinforces why we wish to focus the developers' attention to only those scenarios that they are concerned about. By observing a small-scale example of interleaving workflows like the above, we should be convinced that the combinatorial explosion resulting from the increasing use of choice and parallel combinators can quickly make the number of possible interleaving orders hardly tractable.

We will fix W as the first input argument (Section 5.3.1). We demonstrate cases of our approach by varying the second argument (i.e. restriction criterion) and the third one (i.e. expected goal).

5.5.4.1 Event Projections

We may impose the restriction that no cancellation is present by specifying a scenario $S_1 = \neg\ S$, where $S = \mathit{Proj}\ (\mathit{cancel} \rightarrow \mathit{skip})$. This will effectively force us to focus on studying the behaviour of $\mathit{MakeBooking}' \mid\mid\mid \mathit{CancelOrChangeBooking}'$. In $\mathit{CancelOrChangeBooking}'$ we eliminate the choice of `r = r? & cancel -> ...`, and in $\mathit{MakeBooking}'$ the choice of `r? = r & cancel -> skip`. That is, we have pruned

off a conditional branch from each workflow for our analysis. More precisely, we have reduced our problem space to $[\![S_1]\!]^W_{Scenario}$ that consists of $\binom{6}{3} = 20$ traces to be tested, roughly 43% of the complete interleaving set of W.

5.5.4.2 Attribute Dataflows

Another category of constraint that we may impose upon traces is to focus the data-flow analysis on a particular attribute. If we want to restrict our attention to attribute *allocations* in class *Hotel*, and since values of this chosen attribute are modified and used in the guards of methods *allocate* and *deallocate*, then we will wish to restrict our attention to the data flow by specifying the scenario $S_2 = DF\,(DefThenPUse\,(allocations))$. As a result, in workflow *MakeBooking*, only the first branch of the choice operator references method *allocate*, meaning that we can prune it and only focus on the traces of $MakeBooking' \;|||\; CancelOrChangeBooking$. That is, we have pruned off a conditional branch from workflow *MakeBooking* but left the other intact for consideration. Our problem space is reduced to $[\![S_2]\!]^W_{Scenario}$, containing $\binom{5}{3} + \binom{6}{3} = 30$ orders, roughly 65% of the complete interleaving set of W.

5.5.4.3 Compositionality of Restriction

We may compose and mix constraint criteria from different categories. For example, $S = S_1 \vee S_2$ restricts our attention to only those traces that either do not represent an execution scenario where a cancellation occurs, or demonstrate the computational usage pattern for attribute *allocations*. On the other hand, $S' = S_1 \wedge S_2$ restricts our attention even further to those that satisfy both properties.

5.5.5 Calculating Preconditions

The result from testing an input $(W, S_1 \wedge S_2,\ true)$ on our current model of hotel reservation system is a function *out* whose domain set contains only those satisfying traces of $S_1 \wedge S_2$ (i.e. $[\![S_1 \wedge S_2]\!]^W_{Scenario}$). Each test trace does not contain the event *cancel* and demonstrates an inter-method usage pattern of attribute *allocations*. From

each such trace t, the application of $out\,(t)$ will give us a precondition condition upon which the execution of t is guaranteed to complete (as the goal expected is the weakest as *true*). Say we pick t as $\langle reserve, confirm, \boxed{allocate}, change, \boxed{deallocate}, allocate\rangle$. We already calculated a constraint c' in Section 5.5.2.2 which guarantees that the *inter*-method usage pattern of *allocations*—in the call sequence $\langle allocate\ deallocate\rangle$—would successfully complete. However, interferences may occur before, in between, and after the call sequence in such a way that the interleaved execution cannot complete. As defined in Section 5.3.3, we calculate a new constraint c':
$wp^*\,(\S\ before,\ \ c)\ \wedge\ wp^*\,((\S\ before)\,;allocate\,;(\S\ middle),\ \ true)\ \wedge\ wp^*\,(\S\ t,\ \ true)$, where $\S\ before = (reserve\,;confirm)$, $\S\ middle = \langle change\rangle$, and c was calculated in Section 5.5.2.1.

5.5.6 Revising Specifications

Test case (c', t) is an output of our methodology (as described in Section 5.3.1). Should developers find c' inconsistent with their expectation on the use-case of t, they may revise the specifications of the relevant model operations and run through the generation process again. We already demonstrated this in Section 5.4.

5.6 Conclusion

In this chapter we have addressed the issue of *a theory of model-based testing* as envisaged in Figure 1.1 (on page 3). Our testing theory provides a suitable application context for both Chapter 3 and Chapter 6. The list of rewriting rules that we have developed for BOOSTER predicates (Chapter 3) may serve as a valuable reference should there be a need to revise the specification of an (inconsistent) operation. The revision of an operation affects the meaning of all its containing workflows accordingly.

For the purpose of testing the behaviour models of design, we adapted (Section 5.2) the standard notion of data-flow testing and program slicing—applicable to imperative implementation code—to our BOOSTER context of predicate methods. We defined transformation rules that extract—in the scope of a single predicate method

(Section 5.2.1)—data usage patterns and their associated precondition constraints for predicative specifications. We then suggested how methods should be composed accordingly (Section 5.2.2) to create inter-method usage patterns for testing. This result of data-flow analysis on behavioural specifications paved the way for a testing methodology for parallel workflows. We first presented the design of our methodology (Section 5.3.1). We then presented the use of three kinds of restriction criteria (Section 5.3.2) that may restrict the developers' attention to only those scenarios that they are concerned about—patterns of events, data flows of attributes, and events whose order of execution is not of interest. We finally addressed how we should calculate the precondition constraint for *inter*-method usage patterns (Section 5.3.3) in the presence of interferences on the shared state. We also drew a connection (Section 5.4) between the results of these chapters and those of Chapter 3. The developers may exploit the list of rewriting rules on BOOSTER predicates in Table 3.1 (on page 62) when there is a need to eliminate the inconsistency detected by our testing methodology.

We demonstrated our model-based testing theory on the hotel reservation system (Section 5.5). Given a behaviour design model and a combination of scenarios that are of interest, the generated precondition for each use-case scenario can facilitate the developers in validating the model under test against their understandings of the requirements. Should the precondition of a particular use case be judged as inconsistent, the developers can usefully exploit the list of rewriting rules (Section 3.5) on BOOSTER predicates. Our testing process may be applied iteratively until all generated preconditions are judged satisfactory by the developers. Therefore, we have tested and verified the hypothesis 2.3.1 (on page 26). Since the BOOSTER method language is given a formal, precise operational interpretation, we are able to automate a process that adapts these abstract, behavioural descriptions—once they are considered as consistent—to a chosen execution platform. We will address this issue of *proved rules of model transformation* in Chapter 6 as outlined in our thesis roadmap: a provably correct process that transforms a given BOOSTER object model into a working relational database.

Chapter 6

Database Development

In this chapter we will test and verify the hypothesis 2.4.1 (on page 32) regarding automating the process of producing correct and efficient implementations.

6.1 Introduction

As envisaged in Figure 1.1 (on page 3), from Chapters 3 to 5 we have laid the theoretical foundation for an approach to developing a reliable—both consistent and complete—information system model. We will not, however, be able to build the case of this thesis—an approach to building reliable information systems—until we can ensure that the implementation is faithful to the model we have designed. It would not be acceptable to assume that all BOOSTER model operations will be implemented correctly by some third party—this chapter will address exactly this concern.

What we will ultimately deliver in this chapter is a verified transformation engine—as we will claim via Theorem 6.5.4 (on page 173)—which produces implementing SQL databases—characterised through schemas of tables and stored procedures of queries. The product for our hotel reservation system is illustrated in code fragments as presented in Appendix E.12[1]. We generate the schemas of tables that store values of all attributes and associations that are declared in the source BOOSTER model. For each completed method written in our extended GSL notation—for example, method *reserve* as specified in Appendix A.2—we generate two kinds of SQL database ar-

[1]We refer interested readers to this section for details on the example transformation.

tifacts: 1) a query function which implements its precondition as a runtime guard; and 2) an update procedure which implements its intent of updates. Products of our transformation as exemplified in Appendix E.12 feature both correctness and efficiency. In particular, the code fragment in Appendix E.12.3 has been automatically generated to implement the intended updates of method *reserve*. There are substantially more subtleties—e.g. how to write queries that are consistent with the chosen implementation strategy and caching mechanism—in these generated queries and thus it is difficult to make them error-free if they were written by hand.

Our generated queries are *correct* because they are consistent with our implementation strategy of storing object properties as SQL tables and columns—this is ensured at the compiler, rather than instances, level. More precisely, the update performed by each generated query in the database domain *simulates* [146] a state transformation—which modifies values of the corresponding entities—in the object domain. Our generated queries are also *efficient* because the transformation process adopts a caching mechanism on object paths: their values will be (re-)evaluated only when necessary. As we will only focus on the issue of *correctness* in this chapter, details on this caching mechanism are included in Appendix E.7 and its example can be found in Appendix E.12; relevant sections for this caching mechanism will be referenced for the rest of this chapter.

To pave the way to achieving the above qualities of our generated queries—correctness and efficiency—in this chapter we will develop a unified implementation and semantic framework as summarised in Figure 6.1[2]. There are four kinds of models that are involved in our pipeline of transformation as illustrated in Figure 6.1; each of them conforms to its own metamodel, or data type: 1) a completed BOOSTER model compiled from its textual specification, written in the concrete syntax of our

[2]We use thin-lined, unshaded boxes to denote Haskell program data types, and thin arrows for the executable transformations between them. These constitute our implementation framework. We also use thick-lined, shaded boxes to denote the Z schema formalisation of the corresponding program data types, thick lines with circles at one end for the process of assigning a formal meaning, and arrows with circles at each end for the relationship between formalised concepts. These constitute our semantic framework for establishing the correctness of our transformation.

extended GSL notation; 2) an OBJECT model representing the formalisation of a BOOSTER model, particularly paths within its model operations. 3) an intermediate TABLE model reflecting our implementation strategy (i.e. how class properties are stored in different sorts of tables); 4) a SQL model represented by elements that correspond very closely to the syntax for creating tables and queries and making explicit the set of (primary and foreign) key constraints. A final model-to-text transformation will be applied to generate as output a well-formed SQL database schema. The beginning and ending transformations in our pipeline—from concrete into abstract BOOSTER and from abstract to concrete SQL—have straightforward justification of their correctness and are thus not considered in this chapter.

We will now discuss how this chapter is structured; to emphasise its correspondence to Figure 6.1, we will underline the relevant sections as we refer to them. To facilitate our ultimate proofs of transformation, we will use the functional programming language Haskell [144] to define metamodels of model structures and operations as data types; our rules of transformation on model structures and operations will be defined as Haskell functions accordingly. As our main focus of this chapter is about the correctness of *behavioural* transformation of models, we include details on the structural aspects of models in Appendices E.3 and E.4[3]. We declare a program data type to which structures of the corresponding model must conform (in each sub-section of Appendix E.3). Results from the series of structural transformations of these models—i.e. from BOOSTER to OBJECT, then to TABLE, and finally to SQL—are schemas of tables (Appendix E.4).

We will declare a program data type to which operations of the corresponding model must conform (in each sub-section of Section 6.3). In particular, we emphasize, in Figure 6.2, that the operations in contexts of BOOSTER, OBJECT, and TABLE models all conform to the same syntax: that of our extended GSL; the only difference is how we represent the paths. More precisely, paths used in a substitution program in the context of a BOOSTER model are of type *BPATH*, of type *OPATH* in the context

[3]For the rest of this chapter, we give references to the relevant sections of these two appendices.

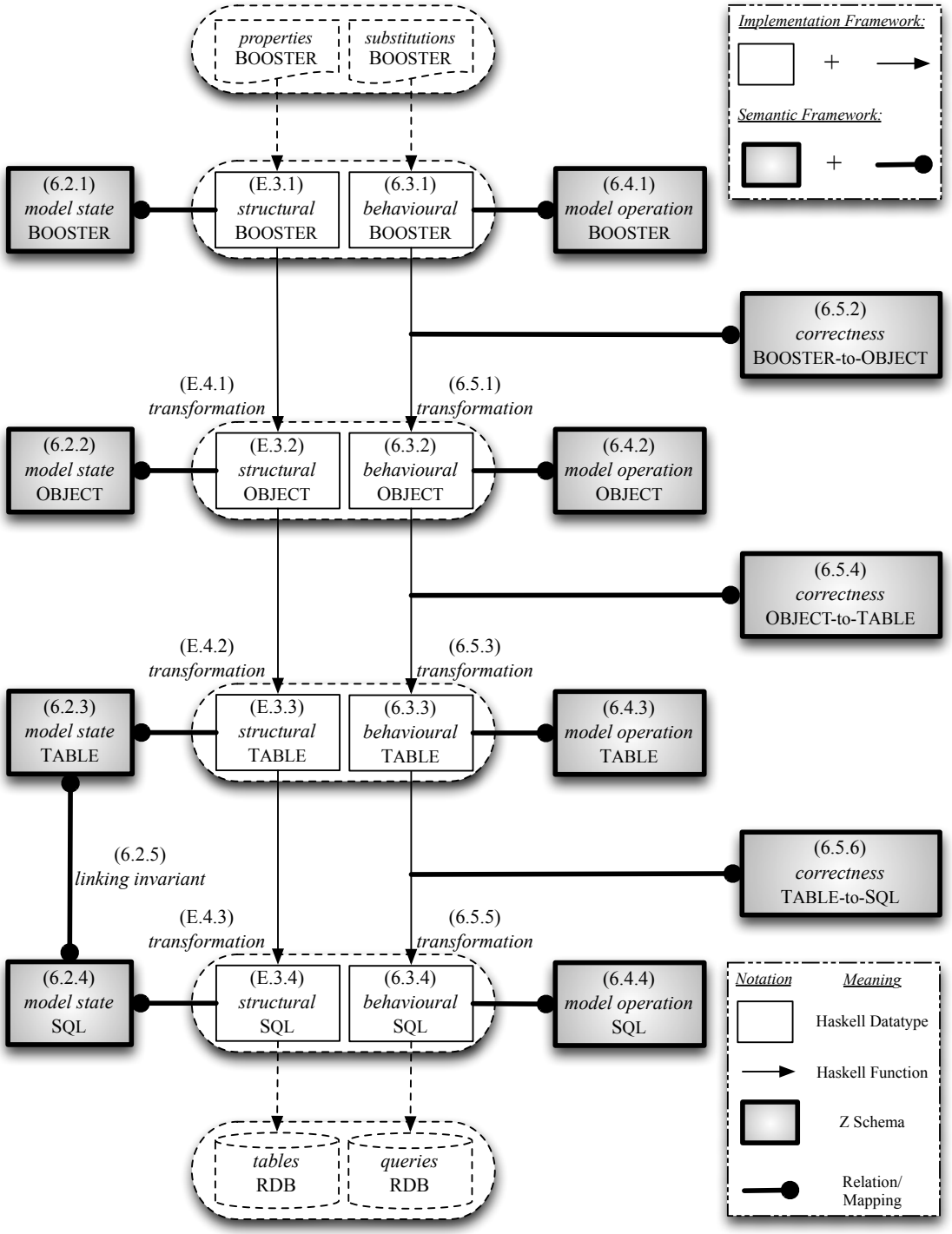


Figure 6.1: Booster Model to Sql Database: Implementation & Semantic Framework

of an OBJECT model, and of type *TPATH* in the context of a TABLE model. SQL database statements that implement these operations, however, adopt a completely different syntax, where paths are implemented using `SELECT` queries. In Section 6.5 we will address exactly the above issue of paths when performing transformations on operations, from one model context to another.

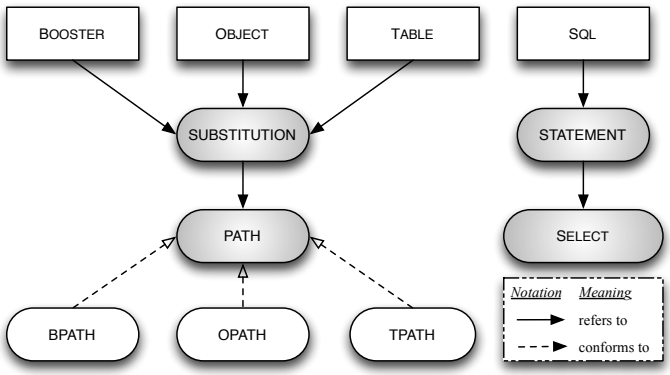


Figure 6.2: Datatypes of Behavioural Models

To establish the correctness of our transformation engine, we will formally characterise the state space of the corresponding model (in each sub-section of Section 6.2), and operations that act upon them accordingly as a relation (in each sub-section of Section 6.4). We will adopt the Z schema language [7] to define such a relational semantics for model states and operations. Our Haskell transformation functions that are written in a declarative manner express explicitly the intended functionalities—through higher-order functions and pattern matching—and thus make them close to the relational semantics.

For each of the three behavioural transformations—from BOOSTER to OBJECT (Section 6.5.1), then from OBJECT to TABLE (Section 6.5.3), and finally from TABLE to SQL (Section 6.5.5)—we will establish its correctness before we proceed to the next level. More precisely, we will prove the correctness of both BOOSTER-to-OBJECT (Section 6.5.2) and OBJECT-to-TABLE (Section 6.5.4) transformations by arguing just the consistency of the path representations as emphasised in Figure 6.2. To prove the correctness of the TABLE-to-SQL transformation (Section 6.5.6), however,

we also need the specification of linking invariants (Section 6.2.5), which relate the relevant model states, as operations in both models do not share the same syntax. In particular, it is fundamental to establish that we correctly translate the various cases of assignments that result from different combinations of declaring and updating properties. In light of this, our Haskell transformation rules will reflect the 36 patterns that we may have for basic assignments: Tables E.1 (on single-valued bi-associations), E.2 (on set-valued associations), E.3 (on sequence-valued associations), and E.4 (on primitive attributes) in Appendix E.6.

Finally, Table 6.1 summarises the results that we will establish—the proofs of which are included in Appendix E.9. For each of the three transformation contexts, we will first need to establish that it is correct on paths, then establish that it is correct on expressions. We will also establish that our caching mechanism for paths does not cause any inconsistency on querying values (Appendix E.7), and that the specific pattern of loop which we adopt to implement iterators (`ALL`, `ANY`) and quantifiers (`Forall`, `Exists`) guarantees termination (Appendix E.8). Finally, we will establish Theorem 6.5.4: we correctly handle the transformations on basic assignment patterns—as summarised in Tables E.1 to E.4—and the substitution program combinators (i.e. $\square$, ; , $\|$, and $\longrightarrow$). Our claims of correctness will be at the *metamodel* level: the database generated—via our transformation engine—from any given Booster model is automatically correct. We will now start presenting the results of this chapter by formalising states of models.

	Booster-to-Object	Object-to-Table	Table-to-Sql
paths	Lemma 6.5.1 (p. 165)	Lemma 6.5.2 (p. 168)	Lemma E.7.1 (p. 257)
expressions	Theorem 6.5.1 (p. 166)	Theorem 6.5.2 (p. 169)	Theorem 6.5.3 (p. 171)
caching	*n.a.*	*n.a.*	Property E.7.1 (p. 257)
termination	*n.a.*	*n.a.*	Property E.8.1 (p. 258)
correctness	*n.a.*	*n.a.*	Theorem 6.5.4 (p. 173)

Table 6.1: Claims of Transformations

6.2 Relational Semantics: Model States

6.2.1 Booster Model State

Since we formalise each Booster model as an Object model, changing particularly the orientation of its path expressions, there is no need to give a separate relational semantics to the Booster model. Both the Booster (Appendix E.3.1) and Object (Appendix E.3.2) models will share the same runtime structure: a set of links between object references. As a result, it is not necessary to define a retrieve relation between states of the Booster and Object models for the correctness proof of the transformation.

6.2.2 Object Model State

We denote the runtime state of an Object model as $\mathcal{S}_{obj}$, characterised through two mappings:

$$
\begin{array}{|l}
\mathcal{S}_{obj} \\
\hline
OBJECT\_MODEL \\
extent : N\_CLASS \pfun \mathbb{P}\, ObjectId \\
value : ObjectId \pfun N\_PROPERTY \pfun Value \\
\hline
\operatorname{dom} extent = \operatorname{dom} class \\
\forall\, c : N\_CLASS;\ o : ObjectId \mid c \in \operatorname{dom} extent \land o \in extent\,(c) \bullet \\
\qquad \operatorname{dom}\,(value\,(o)) = \operatorname{dom}\,((class\ c).property) \\
\hline
\end{array}
$$

The inclusion of *OBJECT_MODEL* enables us to constrain the two mappings according to (the type system of) the object model in question; its structure is similar to that of the programming data type `OBJECT_MODEL` (Appendix E.3.2) and is included in Appendix E.1.1. From a given class $c \in N\_CLASS$, $extent\,(c) \in \mathbb{P}\, ObjectId$ returns the identifiers of its set of instantiated objects; from a given object identifier $o \in ObjectId$ and property $p \in N\_PROPERTY$—presumably, p is declared in the class from which *oid* is instantiated—$value\,(o)\,(p) \in Value$ returns the runtime value of the access path $o.p$. We define *Value* as a structured type that encompasses the possibilities of undefined value (for optional properties), primitive value, and set and

sequence of values.

$$
\begin{array}{rcl@{\qquad}rcl}
\mathit{Value} & ::= & \mathit{null} & \mathit{Primitive} & ::= & \mathit{string}\langle\langle \mathit{String} \rangle\rangle \\
& | & \mathit{value}\langle\langle \mathit{Primitive} \rangle\rangle & & | & \mathit{number}\langle\langle \mathit{Integer} \rangle\rangle \\
& | & \mathit{seqValue}\langle\langle \mathrm{seq}\ \mathit{Primitive} \rangle\rangle & & | & \mathit{reference}\langle\langle \mathit{ObjectId} \rangle\rangle \\
& | & \mathit{setValue}\langle\langle \mathbb{P}\ \mathit{Primitive} \rangle\rangle & & &
\end{array}
$$

6.2.3 TABLE Model State

Given an OBJECT model, its corresponding TABLE model, while maintaining the same sets of operation and expression operators, changes the orientation of its paths from `BPATH` (Section 6.3.2) to `TPATH` (Section 6.3.3), with respect to the chosen implementation strategy reflected through a set of query functions (Appendix E.3.3). As a result, the state of a table model will just be composed of: 1) the type system of the object model in context; and 2) functions for querying the state of such a context object model. More precisely, we will formalise (part of) the table model as:

$$
\begin{array}{|l}
\hline
\mathit{TABLE\_MODEL} \\
\mathit{OBJECT\_MODEL} \\
\\
\mathit{nTableModel} : \mathit{N\_MODEL} \\
\mathit{assocTables}, \mathit{setTables}, \mathit{seqTables} : \mathbb{P}\ \mathit{IDEN\_PROPERTY} \\
\mathit{intProperty}, \mathit{strProperty}, \mathit{refProperty}, \\
\mathit{oppositeOf} : \mathit{IDEN\_PROPERTY} \pfun \mathit{IDEN\_PROPERTY} \\
\mathit{opSpec}, \mathit{opIntent} : \mathit{N\_CLASS} \pfun \mathit{N\_OPERATION} \pfun \mathit{SUBSTITUTION} \\
\hline
\end{array}
$$

where *assocTables*, *intProperty* and *oppositeOf*, and *opSpec* are example queries about, respectively, implementation details, model structures, and model behaviours. See Appendix E.1.2 for the complete characterisation of *TABLE_MODEL*. As a result, we define $\mathcal{S}_{tab} \mathrel{\widehat{=}} [\mathcal{S}_{obj};\ \mathit{TABLE\_MODEL}]$. Constraints on the field functions of schema *TABLE_MODEL* are as specified in the transformation rules in Appendix E.4.2. Both instances of *OBJECT_MODEL* from $\mathcal{S}_{obj}$ and *TABLE_MODEL* coincide with each other.

6.2.4 SQL Model State

We represent a SQL database as a set of named tables, each of which contains a set of tuples. A tuple is a set of column-value mappings. More precisely, we define $\mathcal{S}_{sql} \mathrel{\widehat=} [tuples : N\_TABLE \pfun \mathbb{P}\, Tuple]$ and $Tuple \mathrel{\widehat=} [values : N\_COLUMN \pfun ScalarValue]$. To implement structured, abstract values for the OBJECT model (Section 6.2.2), we adopt the type *ScalarValue* that is supported by the target SQL database, including string, integer, and Boolean values.

6.2.5 Linking Invariant

The correctness proofs of model transformation will require a formal characterisation of the relationship between the two semantic domains. We first define auxiliary mapping functions that retrieve values from a particular domain, that of TABLE or of SQL. In the context of a particular $\mathcal{S}_{obj}$, we define mapping functions $\mathcal{M}$ that retrieve for the various class properties their corresponding mappings, each of which consists of the reference of the current object and the value of its property. For example, we flatten structures of set-valued properties as relations:

$$\begin{array}{l}\mathcal{M}_{set} == \\ \quad (\lambda\, s : \mathcal{S}_{tab};\ p : N\_CLASS \times N\_PROPERTY \bullet \\ \quad\quad \bigcup \{ o : ObjectId;\ v : Value;\ vs : \mathbb{P}\, Primitive \mid \\ \quad\quad\quad o \in s.extent\,(fst\ p) \land v = s.value\,(o)\,(snd\ p) \land v = setValue\,(vs) \bullet \\ \quad\quad\quad\quad \{ v' : vs \bullet [\![o]\!]^{SV} \mapsto [\![v']\!]^{SV} \} \})\end{array}$$

Definitions for cases of other types of properties are similar (see Appendix E.2). In the context of a particular $\mathcal{S}_{sql}$, we retrieve from a given table a set of mappings that consist of values drawn from either two of its columns:

$$\begin{array}{l}\mathcal{M}_{2cols} == (\lambda\, s : \mathcal{S}_{sql};\ n : N\_TABLE;\ c_1, c_2 : N\_COLUMN \bullet \\ \quad\quad \{ row : s.tuples\,(n) \bullet row.values\,(c_1) \mapsto row.values\,(c_2) \})\end{array}$$

The definition for the case of three columns will be similar (see Appendix E.2). We now characterise the linking invariant by placing both domains in context.

TABLE ↔ SQL

$\mathcal{S}_{tab}$
$\mathcal{S}_{sql}$

$\text{dom}\ tuples =$
$\quad \text{dom}\ extent \cup \{p : assocTables \cup setTables \cup seqTables \bullet [\![p]\!]^{NTab}\}$

where `assocTables`, `setTables`, and `seqTables` are TABLE model queries as defined in Appendix E.4.2. Also, for each valid property identification $p \in IDEN\_PROPERTY$, we insist that its mappings that are retrieved from both $\mathcal{S}_{tab}$ and $\mathcal{S}_{sql}$ are equivalent. We have six cases to consider. Each of the linking invariants will impose an equality between the two sets of values that are retrieved from the TABLE and SQL domains. For example, on identifying a set-valued end of a bi-association, i.e. $p \in assocTables \cap setProperty$, we insist that

$$\mathcal{M}_{set}\,(\theta\,\mathcal{S}_{tab},\ p) = \mathcal{M}_{2cols}\,(\theta\,\mathcal{S}_{sql},\ [\![p]\!]^{NTab},\ [\![oppositeOf\ p]\!]^{NCol},\ [\![p]\!]^{NCol}) \qquad (6.1)$$

Similar definitions apply to other cases of properties (see Appendix E.2).

6.3 Behavioural Models

As structures of the three models (Appendix E.3) differ to reflect their domains of application, so do the behaviours that we define to perform upon them. In the series of transformations as outlined in Figure 6.1, our translation of *path expressions* matters the most to the *correctness* of the generated SQL database. Model operations of the BOOSTER, OBJECT, and TABLE models all conform to the syntax of GSL, but they use paths of different types (i.e. `BPATH`, `OPATH` and `TPATH`, respectively) to evaluate expressions or to locate targets of assignments. The same path expression that is defined in the BOOSTER model changes its orientation to reflect current semantic domain in context. Eventually we will derive equivalent path expressions that are written in nested SQL `SELECT` queries. Furthermore, we will use functions `everywhere`, `everything`, and `mkT` from a library module that supports the design pattern as re-

ported in [145]. A detailed account of the relevant usage of this library is included in Appendix E.10.

6.3.1 Booster Model

In the source Booster system model we specify methods using the substitution program notation as described in Section 2.5.2.

```
data METHOD = Method N_METHOD SUBSTITUTION
```

We assume there are auxiliary functions

```
prog   :: METHOD -> SUBSTITUTION
guard  ::           SUBSTITUTION -> PREDICATE
intent ::           SUBSTITUTION -> SUBSTITUTION
```

that retrieve from a Booster method, respectively, its program specification, guarding constraint, and intent of updates.

Booster models (Section 6.3.1), Object models (Section 6.3.2), and Table models (Section 6.3.3) adopt the notation of `SUBSTITUTION` to specify their behaviours, except that we have to transform the paths that appear in expressions to reflect their structural differences. Instead of creating three versions of `SUBSTITUTION`, each accepting one of these path notations, we place atop a type `PATH` that accommodates all possibilities.

```
data PATH = BPath BPATH | OPath OPATH | TPath TPATH
```

In Booster methods each path expression is of type `BPATH` that navigates via associations between classes:

```
data BPATH          = BPathSeq N_CLASS      PATH_START [PATH_COMPONENT]
data PATH_START     = This
                    | Variable N_VARIABLE
                    | Input    N_VARIABLE
                    | Output   N_VARIABLE
                    | SCStart  N_B_ATTRIBUTE
                    | SCInput  N_VARIABLE   EXPRESSION
                    | SCOutput N_VARIABLE   EXPRESSION
data PATH_COMPONENT = Entity   N_B_ATTRIBUTE
                    | SCEntity N_B_ATTRIBUTE EXPRESSION
```

A BOOSTER path is a sequence, contextualised by its enclosing class, that has a start and a tail. An object path can start by referring to either the current class, a dummy variable within the scope of a quantification[4], an input or output variable, a sequence component (e.g. `s[0]`, where `s` is a sequence-valued property), or input variable or output variable that refers to a sequence-valued entity. The tail of an object path is modelled as a sequence of entity names. When the entity referred to is sequence-valued, we may also access one of its members via a valid index. See Example E.11.1 in Appendix E.11 for a demonstration.

6.3.2 OBJECT Model

Each class (Appendix E.3.2) contains an indexed set of operations. Each operation has a set of inputs and outputs, and a complete specification in the substitution language that characterises both its intent and the necessary guard. We may also query the type of any valid input or output.

```
data OPERATION = Operation { nInputs, nOutputs :: [N_VARIABLE]
                           , inputs, outputs   ::  N_VARIABLE -> TYPE
                           , spec              ::  SUBSTITUTION       }
```

We specify operations of the OBJECT using the same `SUBSTITUTION` notation as we do for BOOSTER model. However, paths of type `OPATH` are used to indicate explicitly which properties/classes are accessed, along with its chain of navigation starting from the current class.

```
data OPATH     = BaseOPath                 REF_START
               | RecOPath     OPATH        TARGET
data REF_START = ThisRef                                BASE
               | VariableRef  N_VARIABLE                BASE
               | InputRef     N_VARIABLE                BASE
               | OutputRef    N_VARIABLE                BASE
               | SCRef        IDEN_PROPERTY EXPRESSION  BASE
               | SCInputRef   N_VARIABLE    EXPRESSION  BASE
               | SCOutputRef  N_VARIABLE    EXPRESSION  BASE
data TARGET    = EntityTarget IDEN_PROPERTY
               | SCTarget     IDEN_PROPERTY EXPRESSION
```

[4]Presumably, a path $a.\ \ldots$ starting with a property a of the current class should have been transformed into $this.a.\ \ldots$.

An object path is a left-heavy binary tree, where the left-most child refers to its starting reference and all right children represent target classes/properties that are accessed. The starting reference of an object path is similar to that of a BOOSTER path (i.e. `PATH_START`) but provides strictly more information about the base type (Appendix E.3.2) of that reference. All intermediate and the ending targets of an object path contextualise the properties with their enclosing classes (i.e. `IDEN_PROPERTY` rather than `N_B_ATTRIBUTE` as in `PATH_COMPONENT`). See Example E.11.2 in Appendix E.11 for a demonstration.

6.3.3 TABLE Model

Invoking any of the field functions `opSpec`, `opIntent`, and `opGuard` triggers the transformation from a substitution or predicate expression upon the OBJECT model structure into its equivalent upon the corresponding TABLE model.

The implementation strategy of an `OBJECT_MODEL` is incorporated in a `TABLE_MODEL`, where we use a path of type `TPATH` to indicate, for each navigation to a property in the OBJECT model, the corresponding access to a table which stores that property.

```
data TPATH   = BaseTPath                 REF_START
             | RecTPath     TPATH        TACCESS
data T_ACCESS
  = ClassTAccess IDEN_PROPERTY
  | AssocTAccess IDEN_PROPERTY
  | SetTAccess   IDEN_PROPERTY
  | SeqTAccess   IDEN_PROPERTY             -- retrieve all indexed components
  | SeqTCAccess  IDEN_PROPERTY EXPRESSION  -- retrieve an indexed component
```

A table path is left-heavy (as is an `OPATH`), where the left-most child refers to its starting reference and all right children represent target tables that are accessed. The starting reference of a table path provides exactly the same information as its `OPATH` counterpart (i.e. `REF_START`). On the other hand, all intermediate and the ending targets of a table path denote accesses to a variety of tables, predicated upon our implementation strategy. When the target property is sequence-valued, we distinguish between the two cases where one of its indexed components is to be accessed (`SeqTCAccess`) and where all indexed components are to be accessed (`SeqTAccess`).

See Example E.11.3 in Appendix E.11 for a demonstration.

6.3.4 SQL Model

We characterise the behaviour aspect of the SQL model by the concept of routines, referring to both state-changing procedures and side-effect free functions. A routine contains lists of input and output parameters, a return value if it is a function, and a list of statements as its body implementation.

```
data ROUTINE       = Routine { parameters :: [PARAMETER]
                             , return     ::  Maybe SQL_DATA_TYPE
                             , statements :: [STATEMENT]
                             }
data SQL_DATA_TYPE = SqlString | SqlInteger | SqlBoolean | SqlDate
data PARAMETER     = InParam  N_VARIABLE SQL_DATA_TYPE
                   | OutParam N_VARIABLE SQL_DATA_TYPE
```

We select only the suitable subset of SQL statements that is sufficient to implement (Appendix E.5)—with our current implementation strategy—the modelling concepts of BOOSTER methods. We may insert a row, update all satisfying rows of an expression, delete all satisfying rows of an expression, select a collection of values from a table into a local variable, declare or set a local variable, or execute statements conditionally or iteratively. Moreover, to implement iterators (`ALL`, `ANY`) and quantifiers (`Forall`, `Exists`), we support a list of cursor functions: we may declare a cursor, make it start or stop working, set the collection of values it should traverse over, and return the value at its current position. Numerical, relational, and logical expressions in SQL overlap with those of BOOSTER to a great extent. We present only expressions related to queries upon tables. An expression may refer to the value of a named value, table, or library function. It may also refer to the selection of values from a table, and the union, intersection, and difference of two other expressions. Finally, it may refer to an aggregate function that may appear in a select expression. We will implement table paths that access cells in tables as nested `SELECT` expressions; see Example E.11.4 in Appendix E.11 for a demonstration.

6.4 Relational Semantics: Model Operations

Our semantic model builds from the abstract syntax as presented in Section 6.3, which has its obvious counterpart in Z as included in Appendix E.1.

6.4.1 Object Model Operations

Each method/operation will be implemented as an atomic transaction, whose list of inputs is collected upon its initiation, and outputs produced upon its completion. For each operation, we use $\mathcal{R}_{obj}$ to denote the formal context, under which its transformational effect on the state is defined as a binary relation ($\leftrightarrow$), accommodating the possibility that the effect on state is *non-deterministic*.

$$
\begin{array}{|l}
\mathcal{R}_{obj} \\
\hline
input : \mathbb{P}\, N\_VARIABLE \\
output : \mathbb{P}\, N\_VARIABLE \\
effect : (\mathcal{S}_{obj} \times IO_{obj}) \leftrightarrow (\mathcal{S}_{obj} \times IO_{obj}) \\
\hline
effect \in \mathcal{S}_{obj} \times (input \rightarrow Value) \leftrightarrow \mathcal{S}_{obj} \times (output \rightarrow Value) \\
\hline
\end{array}
$$

where a member of $IO_{obj} == N\_VARIABLE \nrightarrow Value$ represents the collection of inputs or outputs as a mapping from variables to their values.

Using the above formal notions of $\mathcal{S}_{obj}$ and $\mathcal{R}_{obj}$, we write $System_{obj}$ to denote the set of possible object system configurations, each characterised through its current state (of type $\mathcal{S}_{obj}$) and its set of indexed operations (of type $\mathcal{R}_{obj}$). More precisely, $System_{obj} \mathrel{\widehat{=}} [state : \mathcal{S}_{obj};\ relation : N\_CLASS \nrightarrow N\_OPERATION \nrightarrow \mathcal{R}_{obj}]$. We begin by characterising the precise effect of a primitive assignment (:=). We then define inductively the effect of all other GSL combinators (such as $\|$ and ;). This enables us to derive the semantics of each operation that is defined through a valid composition of GSL combinators.

We formally characterise the effect of an assignment as transforming from one object system state to another, affecting specifically upon its *value* mappings according to a pair of given path (as the target) and expression (as the source). More precisely,

we define $AssignInput \mathrel{\widehat=} [path? : PATH;\ e? : EXPRESSION]$ and

AssignEffect

$s, s' : \mathcal{S}_{obj}$
$AssignInput$

$s.nObjModel = s'.nObjModel \land s.sets = s'.sets \land s.classes = s'.classes$

$s.extent = s'.extent$

let $p == target\,[\![path?]\!]$;
$o == context\,[\![path?]\!] \bullet s'.value = s.value \oplus \{o \mapsto$
$s.value\,(o) \oplus \{p \mapsto eval\,(e?)\}\}$

The three conjuncts in the first line of the constraint compartment of *AssignEffect* specify that an assignment does not modify the underlying static type system. The input path *path*? above encompasses both possibilities of `OPATH` and `TPATH`; in the former case, the other input expression $e?$ presumably consists of paths, if any, of type `OPATH`, and of type `TPATH` in the latter case.

We start by relating domains of the OBJECT model and TABLE model, where assignment paths are specified in, respectively, `OPATH` (Section 6.3.2) and `TPATH` (Section 6.3.3). In the OBJECT model domain, an assignment is parameterised by a path of type `BPATH` and an expression that consists of paths, if any, of the consistent type. We formalise each OBJECT model assignment under the formal context of $\mathcal{R}_{obj}$, by defining its *effect* mapping though the constraint of *AssignEffect* and requiring that the sets of external inputs and outputs are empty.

$Assign_{obj}$

$\mathcal{R}_{obj}$
$op? : OPATH$
$oe? : EXPRESSION$

$input = output = \{\}$

$\forall\, s, s' : \mathcal{S}_{obj};\ AssignInput \mid$
$path? = OPath\,(op?) \land e? = oe? \bullet$
$AssignEffect \Leftrightarrow (s, \{\}) \mapsto (s', \{\}) \in effect$

6.4.2 TABLE Model Operations

The characterisation of an assignment in the TABLE model domain ought to be similar to that of $Assign_{obj}$, except that its target is of type `TPATH`, and that its source is of type *EXPRESSION*, where all containing path expressions are of type `TPATH`.

$Assign_{tab}$

$\mathcal{R}_{obj}$
$tp? : TPATH$
$te? : EXPRESSION$

$input = output = \emptyset$

$\forall\, s, s' : \mathcal{S}_{obj};\ AssignInput \mid$
$\quad path? = TPath\,(tp?) \land e? = te? \bullet$
$\quad\quad AssignEffect \Leftrightarrow (s, \{\}) \mapsto (s', \{\}) \in effect$

We define a TABLE model assignment under the same formal context of $\mathcal{R}_{obj}$, reusing the constraint *AssignEffect* and requiring empty sets of inputs and outputs to define its *effect* mapping. Note that input path $tp? \in TPATH$ and expression $te? \in EXPRESSION$ in $Assign_{tab}$ are distinct representations from $op? \in OPATH$ and $oe? \in EXPRESSION$ in $Assign_{obj}$.

We will now move to the next stage of the transformation pipeline: from TABLE model to SQL model. To prove that the transformation of operations from TABLE model to SQL model is correct, we still have, on the side of the TABLE model, to characterise the effect of each substitution operation (which contain paths of type `TPATH`) upon the state $\mathcal{S}_{obj}$; on the other side of the SQL, we need to characterise both its state (i.e. $\mathcal{S}_{sql}$) and queries, as well as the relationship between the two model states (i.e. $\mathcal{S}_{obj}$ and $\mathcal{S}_{sql}$).

We map our extended GSL substitution into a relation

$$[\![\_]\!]_{obj} : SUBSTITUTION \rightarrow ((\mathcal{S}_{obj} \times IO_{obj}) \leftrightarrow (\mathcal{S}_{obj} \times IO_{obj}))$$

whose type is consistent with that of the field relation *effect* of $\mathcal{R}_{obj}$. We use the

identity relation to represent the *skip* operation, as it has no effect on the state.

$$[\![skip]\!]_{obj} = \mathrm{id}\,(\mathcal{S}_{obj} \times IO_{obj})$$

Given a TABLE path $tp?$ and an expression $te?$ (which presumably contains all paths, if any, as table paths), we represent the assignment substitution $tp? := te?$ by the effect relation of $Assign_{tab}$ that exists uniquely with respect to $tp?$ and $te?$.

$$[\![tp? := te?]\!]_{obj} = (\mu\, Assign_{tab}).effect$$

We interpret a guarded substitution $g \longrightarrow S$ as a relation that has the same effect as $[\![S]\!]_{obj}$ within the domain of satisfying states of guard g (denoted as $[\![g]\!]^{states}_{obj}$); otherwise, it just behaves like *skip* as it will be blocked and cannot achieve anything.

$$[\![g \longrightarrow S]\!]_{obj} = \mathrm{id}\,(\mathcal{S}_{obj} \times IO_{obj}) \oplus ([\![g]\!]^{states}_{obj} \lhd [\![S]\!]_{obj})$$

The sequencing combinator has a direct correspondence to the operator of relational composition ($\fatsemi$).

$$[\![S\ ;\ T]\!]_{obj} = [\![S]\!]_{obj} \fatsemi [\![T]\!]_{obj}$$

With a consistent interpretation of parallel composition in our definition of predicate transformer, discussed in Section 3.3, we derive from $S \parallel T$ (and $!\ x\ :\ A\ .\ S$ as its generalisation accordingly) the relation with the effect on state that incorporates both (and all) possible orders of execution.

$$\begin{array}{lcl}
[\![S \parallel T]\!]_{obj} & = & ([\![S]\!]_{obj} \fatsemi [\![T]\!]_{obj}) \cap ([\![T]\!]_{obj} \fatsemi [\![S]\!]_{obj}) \\
[\![!\ x\ :\ \{\}\ .\ S]\!]_{obj} & = & \mathrm{id}\,(\mathcal{S}_{obj} \times IO_{obj}) \\
[\![!\ x\ :\ \{y\} \cup A\ .\ S]\!]_{obj} & = & [\![S[y/x]]\!]_{obj} \fatsemi [\![!\ x\ :\ A\ .\ S]\!]_{obj} \\
 & & \cap \\
 & & [\![!\ x\ :\ A\ .\ S]\!]_{obj} \fatsemi [\![S[y/x]]\!]_{obj}
\end{array}$$

The choice combinator has the obvious correspondence to the union operator ($\cup$); we generalise this relation using $\bigcup$ for the @ iterator.

$$\begin{array}{lcl}
[\![S \,\Box\, T]\!]_{obj} & = & [\![S]\!]_{obj} \cup [\![T]\!]_{obj} \\
[\![@\ x\ :\ A\ .\ S]\!]_{obj} & = & \bigcup\{y : A \bullet [\![S[y/x]]\!]_{obj}\}
\end{array}$$

6.4.3 SQL Model Operations

Each transaction is composed of SQL queries, and similar to $\mathcal{R}_{obj}$, we collect and produce, respectively, its list of inputs and outputs upon its initiation and completion. We use $\mathcal{R}_{sql}$ to denote such formal context, under which again the transformational effect on the state of database is defined accordingly as a function, reflecting the fact that the database implementation is deterministic in its effect.

$$
\begin{array}{|l}
\mathcal{R}_{sql} \\
\hline
input, output : \mathbb{P}\, N\_VARIABLE \\
effect : (\mathcal{S}_{sql} \times IO_{sql}) \rightarrow (\mathcal{S}_{sql} \times IO_{sql}) \\
\hline
this \in input \\
\hline
\end{array}
$$

The mechanism of referencing the current object (via *this*) is simulated through providing by default the value of *this* for each generated stored procedure or function. We model inputs and outputs in the same way as we do for IO_{obj}, except that the range of values is now of type *ScalarValue*. We assume the existence of an auxiliary function $[\![\_]\!]^{sql}_{io} : IO_{obj} \rightarrow IO_{sql}$ that convert between these values.

We represent the execution stack of each stored procedure by three properties: the set of local variables with their bound values and, for each declared cursor (or iterator), its current position and the list of values it stores.

$$
\begin{array}{|l}
Stack \\
\hline
vars : N\_VARIABLE \pfun ScalarValue \\
cursor_{pos} : N\_VARIABLE \pfun \mathbb{N} \\
cursor_{vals} : N\_VARIABLE \pfun \mathrm{seq}\, ScalarValue \\
\hline
\mathrm{dom}\, cursor_{pos} = \mathrm{dom}\, cursor_{vals} \\
\forall\, cur : \mathrm{dom}\, cursor_{pos} \bullet \\
\quad cursor_{pos}\,(cur) = 0 \lor cursor_{pos}\,(cur) \in \mathrm{dom}(cursor_{vals}\,(cur)) \\
\hline
\end{array}
$$

A cursor will return a value only when it has been both declared and activated, i.e. its current position is not zero. By defining a function $eval_{sql} : SQL\_EXPR \rightarrow ScalarValue$ we will write $eval_{sql}\,(e)$ to denote the resulting value that is obtained by

evaluating the SQL expression e; also, by defining a function $sat : SQL\_EXPR \rightarrow \mathbb{P}\, Tuple \rightarrow \mathbb{P}\, Tuple$ we will wirte $sat\,(e)\,(ts)$ to denote the set of satisfying tuples, in the range of tuples ts, of the Boolean expression e.

State-Changing Queries: we first define the formal semantics of queries that update the database state (of type $\mathcal{S}_{sql}$) but not the execution stack. The query `UPDATE` *table*? `SET` *sets*? `WHERE` *cond*? modifies in a table those tuples that satisfy a condition and takes as inputs *table*? a table name, *sets*? a mapping that specifies how relevant columns should be modified, and *cond*? a Boolean condition that chooses the range of tuples to be modified. We require that *table*? must be the name of an existing table in the current state, and that *sets*? intends to modify only columns that exist in the table referred to by *table*?.

UpdateInput

$$
\begin{array}{l}
s : \mathcal{S}_{sql} \\
table? : N\_TABLE \\
sets? : N\_COLUMN \pfun SQL\_EXPR \\
cond? : SQL\_EXPR \\
\hline
table? \in \mathrm{dom}\, s.tuples \\
\forall\, t : s.tuples\,(table?) \bullet \mathrm{dom}\, sets? \subseteq \mathrm{dom}\, t.values
\end{array}
$$

Locally, for each chosen tuple to be modified, we override its column-to-value mappings by those specified in the input *sets*?.

UpdateTuple

$$
\begin{array}{l}
t, t' : Tuple \\
sets? : N\_COLUMN \pfun SQL\_EXPR \\
\hline
t'.values = t.values \oplus \{col : N\_COLUMN;\ e : SQL\_EXPR \mid \\
\qquad\qquad (col, e) \in sets? \bullet col \mapsto eval_{sql}\,(e)\}
\end{array}
$$

Globally, on identifying the set of satisfying tuples as the target for update, we insist that there exists a bijection ($\rightarrowtail\!\!\!\!\!\twoheadrightarrow$) between the pre-state and post-state values of these target tuples; how tuple values are mapped to each other in this bijection is constrained by *UpdateTuple*. Values of the non-satisfying tuples remain intact.

UpdateEffect

$$
\begin{array}{l}
s, s' : \mathcal{S}_{sql} \\
\mathit{UpdateInput} \\
\hline
\textbf{let}\ \mathit{target} == \mathit{sat}\,(\mathit{cond}?)\,(s.\mathit{tuples}\ \mathit{table}?) \bullet \\
\textbf{let}\ \mathit{unchanged} == s.\mathit{tuples}\ \mathit{table}? \setminus \mathit{target} \bullet \\
\textbf{let}\ \mathit{changed} == s'.\mathit{tuples}\ \mathit{table}? \setminus \mathit{unchanged} \bullet \\
\quad \mathit{unchanged} \subseteq s.\mathit{tuples}\,(\mathit{table}?) \land \mathit{unchanged} \subseteq s'.\mathit{tuples}\,(\mathit{table}?) \\
\quad \land \\
\quad (\exists f : \mathit{target} \rightarrowtail\!\!\!\!\!\!\rightarrow \mathit{changed} \bullet (\forall t, t' : \mathit{Tuple} \mid t \mapsto t' \in f \bullet \mathit{UpdateTuple}))
\end{array}
$$

Synthesising the above schemas, we are now able to define the formal semantics of the `UPDATE` query by placing $\mathcal{R}_{sql}$ into context and defining its *effect* relation accordingly. We also insist that the `UPDATE` query does not have an effect on the execution stack.

`UPDATE`

$$
\begin{array}{l}
\Xi \mathit{Stack} \\
\mathcal{R}_{sql} \\
\mathit{UpdateInput} \\
\hline
\mathit{input} = \mathit{output} = \{\} \\
\forall s, s' : \mathcal{S}_{sql} \bullet \mathit{UpdateEffect} \Leftrightarrow (s, \{\}) \mapsto (s', \{\}) \in \mathit{effect}
\end{array}
$$

Other state-changing queries `INSERT` and `DELETE` and side-effect-free queries will be similarly defined (see Appendix E.1.3).

For each SQL statement, we assign to it a relational semantics by mapping it to a relation on states (of type $\mathcal{S}_{sql}$). This is a similar process to that for $[\![\_]\!]_{obj}$.

$$[\![\_]\!]_{sql} : \mathit{Statement} \rightarrow ((\mathcal{S}_{sql} \times \mathit{IO}_{sql}) \leftrightarrow (\mathcal{S}_{sql} \times \mathit{IO}_{sql}))$$

For simple query statements, we refer to the schema definitions above. For example,

$$[\![\texttt{UPDATE}\,(t, \mathit{sets}, \mathit{cond})]\!]_{sql} \quad = \quad (\mu\,\texttt{UPDATE}).\mathit{effect}$$

For the convenience of defining our semantics on query combinators, we also define a semantic function

$$[\![\_]\!]_{\mathrm{seq}\ sql} : \mathrm{seq}\ \mathit{Statement} \rightarrow ((\mathcal{S}_{sql} \times \mathit{IO}_{sql}) \leftrightarrow (\mathcal{S}_{sql} \times \mathit{IO}_{sql}))$$

which maps from a sequence of statements to a relation on states, with its calculation depending upon that of $[\![\_]\!]_{sql}$:

$$\begin{array}{lcl} [\![\langle\rangle]\!]_{\text{seq } sql} & = & \text{id}\,(\mathcal{S}_{sql} \times IO_{sql}) \\ [\![\langle stmt\rangle \frown stmts]\!]_{\text{seq } sql} & = & [\![stmt]\!]_{sql} \fatsemi [\![stmts]\!]_{\text{seq } sql} \end{array}$$

The formal semantics of an `IF` ... `THEN` ... `ELSE` ... statement is the union of the semantic interpretations of the two sequences of statements in its body, each suitably restricted on its domain.

$$\begin{array}{rl} [\![\texttt{IF}\ b\ \texttt{THEN}\ stmts1\ \texttt{ELSE}\ stmts2]\!]_{sql} = & [\![b]\!]^{states}_{sql} \lhd [\![stmts1]\!]_{\text{seq } sql} \\ & \cup \\ & [\![\texttt{NOT}\ b]\!]^{states}_{sql} \lhd [\![stmts2]\!]_{\text{seq } sql} \end{array}$$

To define the semantics of a `WHILE` loop, we intend for the following equation to hold

$$\begin{array}{rl} & [\![\texttt{WHILE}\ b\ \texttt{DO}\ stmts\ \texttt{END WHILE}]\!]_{sql} \\ = & [\![\texttt{IF} \quad b \quad \texttt{THEN} \quad (stmts \frown \langle \texttt{WHILE}\ b\ \texttt{DO}\ stmts\ \texttt{END WHILE}\rangle) \quad \texttt{ELSE} \quad \langle\rangle]\!]_{sql} \end{array}$$

By applying the definition of $[\![\_]\!]_{sql}$ on `IF` ... `THEN` ... `ELSE` ... and $[\![\_]\!]_{\text{seq } sql}$ on $\langle\rangle$, we have

$$\begin{array}{rl} & \boxed{[\![\texttt{WHILE}\ b\ \texttt{DO}\ stmts\ \texttt{END WHILE}]\!]_{sql}} \\ = & [\![b]\!]^{states}_{sql} \lhd ([\![stmts]\!]_{\text{seq } sql} \fatsemi \boxed{[\![\texttt{WHILE}\ b\ \texttt{DO}\ stmts\ \texttt{END WHILE}]\!]_{sql}}) \\ & \cup \\ & [\![\texttt{NOT}\ b]\!]^{states}_{sql} \lhd \text{id}\,(\mathcal{S}_{sql} \times IO_{sql}) \end{array}$$

Let us define a function

$$F\,(X) = [\![b]\!]^{states}_{sql} \lhd ([\![stmts]\!]_{\text{seq } sql} \fatsemi X) \cup [\![\texttt{NOT}\ b]\!]^{states}_{sql} \lhd \text{id}\,(\mathcal{S}_{sql} \times IO_{sql})$$

When $X = [\![\texttt{WHILE}\ b\ \texttt{DO}\ stmts\ \texttt{END WHILE}]\!]_{sql}$, we obtain

$$[\![\texttt{WHILE}\ b\ \texttt{DO}\ stmts\ \texttt{END WHILE}]\!]_{sql} = F\,([\![\texttt{WHILE}\ b\ \texttt{DO}\ stmts\ \texttt{END WHILE}]\!]_{sql})$$

which means that $[\![\texttt{WHILE}\ b\ \texttt{DO}\ stmts\ \texttt{END WHILE}]\!]_{sql}$ should be a fixed-point of function F. The least fixed-point (LFP) of function F—i.e. $\bigcup_{n\in\mathbb{N}} F^n\,(\emptyset)$—exists by Kleene's

fixed-point theorem, since F is easily provable to be continuous[5]. We choose this LFP of F for the value of $[\![\texttt{WHILE}\ b\ \texttt{DO}\ stmts\ \texttt{END WHILE}]\!]_{sql}$, because it encompasses all possibilities of finite iterations of the loop: $F^1(\emptyset)$ considers only the case where the stay condition does not hold in the very beginning, $F^2(\emptyset) \supseteq F^1(\emptyset)$ also considers the case where the stay condition holds for one iteration, ..., $F^{i+1} \supseteq F^{\mathrm{i}}(\emptyset)$ also considers the case where the stay condition holds for $i \in \mathbb{N}$ iterations, etc.

6.5 Behavioural Model Transformation

6.5.1 Booster to Object Models

Transformation on Paths: we will first consider the transformation on Booster paths. We define a transformation from a Booster model path (`BPATH` in Section 6.3.1) to an Object model path (`OPATH` in Section 6.3.2). In the context of Object model, we amend the data type of each iterator (`ALL`, `ANY`) and quantifiers (`Forall`, `Exists`) by deriving the type of its bound variable. We demonstrate such amendment by comparing

```
data PATH_START = Variable    N_VARIABLE      | ...
data REF_START  = VariableRef N_VARIABLE BASE | ...
```

where type `REF_START` in the context of an Object model possesses strictly more information about its bound variable: its type, derived from the set-valued expression that it quantifies over. We implement this functionality as `dVarType` that takes as input the enclosing system and substitution program (which contains instantiations of `PATH_START`) and returns an amended substitution (which contains instantiations of `REF_START`). See Example E.11.5 in Appendix E.11 for a demonstration.

We proceed with the path transformation assuming that the above amendment has been completed. We contextualise such a transformation by the Booster system `sys` ∈ `B_SYSTEM` and substitution program `bprog` that enclose the source Booster

[5]We will need to prove that continuity is preserved under function composition ($\circ$). We can also easily prove that operators of domain restriction ($\lhd$), union ($\cup$), and sequential composition ($\fatsemi$) are all continuous. Since F can be expressed as the composition of these operators, we conclude that F is also continuous.

path in question. Also, we maintain an environment `env` that contains the type information of bound variables in scope. We then write `(booToObjPath sys bprog env)` to transform from a BOOSTER path to its corresponding OBJECT path.

```
type ENV = [(N_VARIABLE, BASE)]
booToObjPath :: B_SYSTEM -> SUBSTITUTION -> ENV -> PATH -> PATH
booToObjPath sys bprog env (BPath bpath) = OPath (booToObjPath' sys bprog env bpath)
```

We delegate the transformation to the auxiliary function `booToObjPath'`.

```
booToObjPath' :: B_SYSTEM -> SUBSTITUTION -> ENV -> BPATH -> OPATH
```

There are two cases for `booToObjPath'` to consider. The base case is a BOOSTER path that contains only the starting reference.

```
booToObjPath' sys bprog env (BPathSeq c start [])
  | start' <- refStart sys c bprog env start = BaseOPath start'
```

We transform (via `refStart`) a reference (of type `PATH_START`) into another (of type `REF_START`), from which we construct a base object path (i.e. `BaseBPath`).

```
refStart :: B_SYSTEM -> N_CLASS -> SUBSTITUTION -> ENV -> PATH_START -> REF_START
refStart sys c _ _    This          = ThisRef       (ClassBase c)
refStart sys c _ env (Variable x)   | baseT <- head ([t | (y, t) <- env, y == x])
                                    = VariableRef x baseT
refStart sys _ s env (SCInput  i e) | t  <- varTypeInSubs i s
                                    , e' <- everywhere
                                              (mkT (booToObjPath sys s env)) e
                                    = SCInputRef i e' (base t)
...
```

A reference of `This` to the current class is translated by passing its enclosing class `c`. A bound variable reference triggers a lookup for its type in the environment *env*. An input reference will trigger the derivation (via `varTypeInSubs`) of its type from the enclosing method program `s`. The same procedure applies to the transformation on the pattern of an output reference. A component of some input sequence `i` will trigger both the transformation on the index expression `e` (via `everywhere (mkT (booToObjPath ...))`) and the derivation of `i`'s type. See Example E.11.6 in Appendix E.11 for a demonstration.

The recursive case for `booToObjPath'` is a BOOSTER path that involves class navigation (i.e. with one or more dots in its textual form).

```
booToObjPath' sys bprog env p@(BPathSeq c start pcs)
  | bpath_prefix <- BPathSeq c start (init pcs)
  , op_prefix    <- booToObjPath' sys bprog env bpath_prefix
  , tar          <- target        sys bprog env p
  = RecOPath op_prefix tar
```

The transformation process for such kind of path is recursive. For an object path $r.e_1.....e_{n-1}.e_n$, where $r \in$ `PATH_START` denotes its starting reference and the sequence $\langle e_1, \ldots, e_{n-1}, e_n \rangle$ its chain of class navigation, we perform three tasks: 1) to recursively transform its prefix (i.e. $r.e_1.....e_{n-1}$) into an object path (via a recursive call to `booToObjPath'`); 2) to calculate the target property (via `target`) of the entire path (i.e. $r.e_1.....e_{n-1}.e_n$); and 3) to construct a recursive object path (via the constructor `RecBPath`) out of these. The auxiliary function `target` locates exactly the property (and its enclosing class) that an object path intends to access. See Example E.11.7 in Appendix E.11 for a demonstration.

Lemma 6.5.1 (***Correctness:* BOOSTER-*to*-OBJECT *Path Transformation***)
In the context of 1) a BOOSTER system $sys \in B\_SYSTEM$ 2) a BOOSTER method specification $prog \in SUBSTITUTION$ 3) an environment $env \in Env$ of dummy variables, a given BOOSTER path $path_{boo} \in BPATH$ and its translated OBJECT path $booToObjPath\,(sys)\,(prog)\,(env)\,(path_{boo}) \in OPATH$ refer to the same property and containing object. Formally[6],

$$\forall\, path_{boo} : BPATH;\ path_{obj} : OPATH \mid$$
$$path_{obj} = booToObjPath\,(path_{boo}) \bullet \left(\begin{array}{l} context\,[\![path_{boo}]\!] = context\,[\![path_{obj}]\!] \\ \wedge \\ target\,[\![path_{boo}]\!] = target\,[\![path_{obj}]\!] \end{array} \right)$$

□

Transformation on Methods: we will now consider the transformation on BOOSTER methods. Part of the transformation of `omOperation` (Appendix E.4.1)

[6]To express the correctness property of the path translation *booToObjPath*, we define two semantic functions $target\,[\![\_]\!] : PATH \rightarrow N\_PROPERTY$ and $context\,[\![\_]\!] : PATH \rightarrow ObjectId$ that inform us of, respectively, the name of property that a given path refers to and the reference of object that contains this property. For example, in a BOOSTER path $this.a.b$, we obtain $context\,[\![this.a.b]\!] = this.a$ and $target\,[\![this.a.b]\!] = b$.

```
omOperation :: B_SYSTEM -> METHOD -> OPERATION
omOperation sys m = Operation { ...
                              , spec = booToObjProg sys (prog m)
                              }
```

initialises the field function `spec` of the OBJECT model `OPERATION` (Appendix E.3.2) by transforming (via `booToObjProg`)—given a context BOOSTER system `sys` and within a method `m`—each occurrence of a path of type `BPATH` (Section 6.3.1) to a path of type `OPATH` (Section 6.3.2).

```
booToObjProg :: B_SYSTEM -> SUBSTITUTION -> SUBSTITUTION
booToObjProg sys prog = let prog' = dVarType sys prog
                        in  everywhere (mkT (booToObjPath sys prog [])) prog'
```

where `prog' = dVarType sys prog` denotes the first stage that we mentioned and assumed correct—each occurrence of an iterator or a quantifier is derived for the type of its bound variable. The second stage locates each occurrence of paths of type `BPATH` (via `everywhere` and `mkT` functions) and transforms it (via `booToObjPath`) to its corresponding path of type `OPATH`.

6.5.2 Correctness: BOOSTER-to-OBJECT Transformation

Although both BOOSTER and OBJECT models adopt the same substitution notation for their model operations, the types of their path expressions (i.e. `BPATH` in Section 6.3.1 and `OPATH` in Section 6.3.2) are distinct in how they retrieve values. Therefore, we ought to prove the correctness of the BOOSTER-to-OBJECT model transformation by arguing that path expressions in BOOSTER are correctly represented in OBJECT.

Theorem 6.5.1 (*Correctness:* **BOOSTER-*to*-OBJECT** *Model Transformation*) OBJECT model operations that are transformed from those of the BOOSTER model perform consistent updates upon the system state. □

6.5.3 OBJECT to TABLE Models

Transformation on Paths: we will first consider the transformation on OBJECT paths. We define a transformation from an OBJECT path (`OPATH` in Section 6.3.2) to

a TABLE path (`TPATH` in Section 6.3.3).

```
objToTabPath :: OBJECT_MODEL -> PATH -> PATH
objToTabPath om    (OPath opath) = TPath (objToTabPath' om opath)
```

We delegate the task of transforming an object path to its corresponding table path to `objToTabPath'`. The base case considers basic object paths:

```
objToTabPath' :: OBJECT_MODEL -> OPATH -> TPATH
objToTabPath' om bop@(BaseOPath start) | SCRef att oe t <- start
                                       = let te = (objToTabExpr om oe)
                                         in  BaseTPath (SCRef att te t)
objToTabPath' _  (BaseOPath start)     = BaseTPath  start
```

Since both object and table paths use `REF_START` to specify their starting references, there is no need for transformation on this. However, if a basic object path refers to a sequence component (i.e. `(SCRef att oe t)`, then we have to invoke a call of `objToTabExpr` to transform its index expression `oe` that contains object paths to its corresponding table expression that contains table paths. Function `objToTabExpr` traverses through the given object expression (via `everywhere`) and applies `objToTabPath` to each of its containing paths of type `OPATH`.

```
objToTabExpr :: OBJECT_MODEL -> EXPRESSION -> EXPRESSION
objToTabExpr om e = everywhere (mkT (objToTabPath om)) e
```

The recursive case of `objToTabPath'` considers recursive object paths:

```
objToTabPath' om (RecOPath op tar) =
  case tar of
    EntityTarget iden@(c, p)
      | iden `elem` biAssoc'    om c -> RecTPath tp (AssocTAccess iden)
      | iden `elem` classTables tm c -> RecTPath tp (ClassTAccess iden)
      | iden `elem` setTables   tm   -> RecTPath tp (SetTAccess   iden)
      | iden `elem` seqTables   tm   -> RecTPath tp (SeqTAccess   iden)
    SCTarget iden oe                 -> let te = objToTabExpr om oe
                                        in  RecTPath tp (SeqTCAccess iden te)
  where tm = objToTab      om
        tp = objToTabPath' om op
```

The above transformation is designed to correspond to our implementation strategy (Appendix E.4.2). Each recursive object path is structured as `RecOPath op tar`: `op` is its prefix of type `OPATH`, which we recursively transform into a table path equivalent `tp`; and `tar` is its target property. We then decide for each given `tar` which sort

of table access it demands, by checking its membership against various domains: for example, an access to the corresponding association table if it is a bi-association, class table if it is a member of `classTables`, etc. We assume that if the target property is sequence-valued, it cannot be accessed for its entirety, but only one of its members through indexing.

Lemma 6.5.2 (***Correctness:* OBJECT-*to*-TABLE *Path Transformation***)
In the context of an $om \in OBJECT\_MODEL$, for an object path $path_{obj} \in OPATH$, the transformation from it to a table path $objToTabPath\,(om)\,(path_{obj}) \in TPATH$ preserves the intended property of access and its context object.

$$\forall\, path_{obj} : OPATH;\ path_{tab} : TPATH \mid$$
$$path_{tab} = objToTabPath\,(om)\,(path_{obj}) \bullet \left(\begin{array}{l} context\,[\![path_{obj}]\!] = context\,[\![path_{tab}]\!] \\ \wedge \\ target\,[\![path_{obj}]\!] = target\,[\![path_{tab}]\!] \end{array} \right)$$

□

Transformation on Operations: we will now consider the transformation on OBJECT operations. As part of the transformation of `objToTable` (Appendix E.4.2),

```
objToTable om = TableModel { ...
                           , opSpec  = opSpec'  om
                           , opGuard = opGuard' om, opIntent = opIntent' om }
```

Field functions `opSpec`, `opGuard`, and `opIntent` of the TABLE model (Appendix E.3.3) should return predicates and substitutions that adopt paths of type `TPATH` (Section 6.3.3).

To achieve this, we transform (via `objToTabPath`) each occurrence of an object path to its corresponding table path.

```
opSpec' :: OBJECT_MODEL -> N_CLASS -> N_OPERATION -> SUBSTITUTION
opSpec'   om c o = let ospec = spec (operations (classes om c) o)
                   in  everywhere (mkT (objToTabPath om)) ospec
```

We write `spec (operations (classes om c) o)`, for each operation `o` that is declared in class `c` and OBJECT model `om`, to retrieve `o`'s specification written in substitution programs. From the transformed specification that `opSpec'` returns, its predicate

guard and substitution intent is automatically adapted to the TABLE model context. For example,

```
opGuard' :: OBJECT_MODEL -> N_CLASS -> N_OPERATION -> PREDICATE
opGuard'  om c   = guard  . (opSpec' om c)
```

6.5.4 Correctness: OBJECT-to-TABLE Transformation

The argument here resembles that of Theorem 6.5.1: both OBJECT and TABLE models adopt the same substitution notation for their model operations, the types of their path expressions (i.e. `OPATH` in Section 6.3.2 and `TPATH` in Section 6.3.3) are distinct in how they retrieve values. Therefore, we ought to prove the correctness of the OBJECT-to-TABLE model transformation by arguing that path expressions in the two domains are consistent.

Theorem 6.5.2 (***Correctness:* OBJECT-*to*-TABLE *Model Transformation***) Operations upon TABLE model that are transformed from those of the OBJECT model perform consistent updates upon the system state. □

6.5.5 TABLE to SQL Models

In this section we will transform each substitution program that contains paths of type `TPATH` (Section 6.3.3) into a database stored procedure, i.e. a sequential composition of SQL queries (Section 6.3.4). We will present our transformations on expressions, on iterations, on assignments, and on combinators of substitution programs. The correctness of our TABLE-to-SQL expression transformation (Theorem 6.5.3 on page 171) relies on that of the TABLE-to-SQL path transformation (Lemma E.7.1 on page 257), which relies on the consistency property of our caching mechanism (Property E.7.1 on page 257). In Appendix E.7.2 we establish that our generated queries are efficient, in that within the scope of each procedure, we re-evaluate nested `SELECT` queries—which implement path expressions—only when necessary; if not necessary to re-evaluate, then we simply refer to variables that cache values of the corresponding path expressions. For an example of how this caching mechanism works, see Appendix E.12.

Transformation on Expressions: we transform both predicates and expressions on TABLE model into those on SQL model, by specifying two separate functions.

```
toSqlExpr  :: TABLE_MODEL -> PREDICATE  -> SQL_EXPR
toSqlExpr' :: TABLE_MODEL -> EXPRESSION -> SQL_EXPR
```

For each predicate, relational, and numerical combinator, as well as certain set operators (e.g. membership, union, etc), we have a direct correspondence of SQL expression combinator. For example,

```
toSqlExpr  tm (And       p q) = toSqlExpr  tm p `AND` toSqlExpr  tm q
toSqlExpr  tm (Lt        e f) = toSqlExpr' tm e :<:   toSqlExpr' tm f
toSqlExpr  tm (NotMember e f) = toSqlExpr' tm e `NIN` toSqlExpr' tm f
toSqlExpr' tm (Plus      e f) = toSqlExpr' tm e :+:   toSqlExpr' tm f
```

For predicates and expressions (of the TABLE model) that are not directly supported in the SQL model, we transform them to those that are formally equivalent. For example,

```
toSqlExpr' tm (Card e) | isPathExpr e
                       = SELECT [COUNT (VAR oid)] (toSqlExpr' tm e) TRUE
toSqlExpr' tm (Tail e) | isPathExpr e
                       , Path (TPath rtp@(RecTPath tp _)) <- e
                       = let col     = targetColumn    rtp
                             tab     = targetTable  tm rtp
                             context = VAR (nCacheVar tp)
                         in  SELECT [VAR index, VAR col]
                                    tab
                                    ((VAR oid :=: context)
                                       `AND` (VAR index :>: NUMBER 1))
```

where we restrict expression `e` to be a path.

An important case of transformation to note is the treatment of equality.

```
toSqlExpr tm (Equals e f)
  | f == Null, isPathExpr e, Path (TPath p@(RecTPath _ ta)) <- e
  = let tab  = targetTable  tm p
        col  = targetColumn    p
        cond = targetRow    tm p
    in  case ta of
          ClassTAccess _ -> SELECT [       VAR col ] tab cond :=: NULL
          AssocTAccess _ -> SELECT [COUNT (VAR col)] tab cond :=: NUMBER 0
  | otherwise
  = (toSqlExpr' tm e) :=:   (toSqlExpr' tm f)
```

In a special pattern of equality `e = f`, where `e` and `f` denote, respectively, a table access and the `Null` value, then we ought to refer back to our implementation strategy

(Appendix E.4.2): `Null` in the domain of TABLE is represented as either `NULL` in class tables, or as non-existing rows in the SQL domain.

Theorem 6.5.3 (*Correctness:* **TABLE-*to*-SQL** *Expression Transformation*) Predicates or expressions in the TABLE model, if supported, are correctly represented in the SQL model via, respectively, `toSqlExpr` and `toSqlExpr'`. □

Transformation on Assignments: corresponding to the possible ways of declaring bi-directional associations as we demonstrated in Section 2.5.1, our transformation on assignments must take account of possible ways of updating these associations. Tables E.1 (on single-valued bi-associations), E.2 (on set-valued associations), E.3 (on sequence-valued associations), and E.4 (on primitive attributes) summarise the patterns of basic assignments that we handle. Each table is self-contained: the columns from left to right specify declarations of properties, numerical identifiers of patterns, their BOOSTER predicative specifications, their abstract implementation in the substitution program, and their concrete implementation in database queries. The pattern numbers in these tables also correspond to those that are documented as comments in Appendix A.2. The function `toSqlProc` delegates the task of transforming assignment to another auxiliary function `transAssign` that implements exactly the 36 patterns as specified in the three tables.

```
toSqlProc tm _ s@(Assign _ _) = transAssign tm s
transAssign :: TABLE_MODEL -> SUBSTITUTION -> [STATEMENT]
```

For demonstrations, see Sections 1.1.4 and Appendix E.12.3.

Transformation on Combinators: a transformation function `toSqlProc` that turns a substitution into a list of SQL query statements is defined as follows. For a parallel composition (`Parallel`), we ensure that no further cache updates take place upon common read variables from both sides, as the order in which both of its operands run should not matter, and we select to execute first the output of the left operand from `toSqlProc` by default. The treatment of a sequential composition (`Sequence`) is trivial:

execute outputs of both operands from `toSqlProc` in their specified order. For guarded substitution (`Guard`), we transform it into an `IfThenElse` pattern that corresponds to the relational semantics of `Guard` in Section 6.4.2. Our treatment of bounded choice (`Choice`) assumes that its first operand has a guard, upon which we decide which branch to select. Finally, we transform iterators (`ALL`, `ANY`) to instantiations of the `iterate` pattern. As a demonstration, see Appendix E.12.3 for how we transform on the iterator `ALL` in method `reserve` (Appendix A.2).

6.5.6 Correctness: TABLE-to-SQL Transformation

Finally, we are now able to establish that our transformation on any given substitution program is correct, with respect to the linking invariant defined in Section 6.2.5. We illustrate the correctness criterion in Figure 6.3, where $\mathcal{S}_{obj}$ refers to the underlying OBJECT model state that a TABLE model contains. Similarly, $[\![prog]\!]_{obj}$ in Figure 6.3 refers to a TABLE program that has exactly the same program structure (see Figure 6.2 on page 145) as its source OBJECT program, except that all `OPAHT` path expressions have been translated to those of `TPATH`.

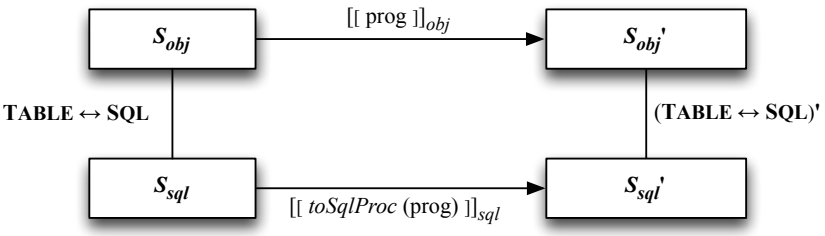


Figure 6.3: Correctness of Model Transformation

Given a substitution program *prog* and its context table model, we first assume that the linking invariant is satisfied both at the before state (i.e. TABLE ↔ SQL) and at the after state (i.e. (TABLE ↔ SQL)′). Then we will need to establish that for each state transformation that is characterised by the relational effect of the generated SQL program from *prog* (i.e. $[\![toSqlProc\,(\theta\ TABLE\_MODEL)\,(prog)]\!]_{sql}$), there is at least a corresponding state transformation that is characterised by the relational effect of

the source TABLE program (i.e. $[\![prog]\!]_{obj}$). More precisely, we state our correctness theorem as, based on the standard notion of simulation as in [146]:

Theorem 6.5.4 Let the schema *TransInput* denote the inputs of transformation $TransInput \mathrel{\widehat=} [TABLE\_MODEL;\ prog : SUBSTITUTION]$, then

$$
\begin{array}{l}
\forall\, TransInput;\ \Delta\mathcal{S}_{obj};\ \Delta\mathcal{S}_{sql} \mid \\
\quad \textsc{Table} \leftrightarrow \textsc{Sql} \wedge \\
\quad\quad \mathcal{R}_{sql}[effect_{sql}/effect] \wedge \\
\quad\quad\quad effect_{sql} = [\![toSqlProc\,(\theta\ TABLE\_MODEL)\,(prog)]\!]_{\mathrm{seq}\ sql} \bullet \\
\quad\quad\quad \left(\begin{array}{l} \exists \mathcal{S}'_{obj} \bullet (\textsc{Table} \leftrightarrow \textsc{Sql})' \wedge \\ \qquad\qquad \mathcal{R}_{obj}[effect_{obj}/effect] \wedge effect_{obj} = [\![prog]\!]_{obj} \end{array} \right)
\end{array}
$$

where transformation function *toSqlProc* is as defined in the above section.

□

We have now completed the final theorem of this chapter. See Appendix E.12 for an application of our transformation rules to the hotel reservation system case study. We will now proceed to conclude this chapter.

6.6 Conclusion

In this chapter we have completed our thesis roadmap—as envisaged in Figure 1.1 (on page 3)—by addressing the issue of *proved rules of model transformation*. As illustrated in Figure 6.1, we presented a unified implementation and semantic framework, where we correctly transform properties and methods in a given BOOSTER model into tables and queries in a SQL relational database. We established Theorem 6.5.4 that ensures the correctness of our transformation on the behavioural features of BOOSTER. We first defined the structural models (Appendix E.3) and behavioural models (Section 6.3) of all three model domains BOOSTER, OBJECT, and TABLE that are involved in our pipeline of transformation. We then performed transformations on the structures of these models (Appendix E.4) and their operations (Sections 6.5.1, 6.5.3, and 6.5.5). To establish the correctness of these model transformations, we took a formal account of the model states of all three domains (Section 6.2)

and their operations that are characterised as relations of states (Section 6.4). Finally, we established the list of claims of our transformations (Sections 6.5.2, 6.5.4, and 6.5.6) as summarised in Table 6.1.

We applied the implemented Haskell transformation engine to the hotel reservation system (Appendix A.2) and presented part of the generated database artifacts—tables, key constraints, stored functions, and stored procedures—in Appendix E.12. We observed the value of our automated, verified transformation: the solution space of the hotel reservation system developers was altered—from one that consists of scattered tables and rows to one that is defined through classes, associations, and entities. The generated update queries—correct and efficient—are as if they were written by experienced SQL programmers in the first place. Therefore, we have tested and verified the hypothesis 2.4.1 (on page 32). We have also built the case of this thesis—on the basis that, as argued in Chapters 1 and 2, our hotel reservation case study is a representative information system—through testing and verifying all four parts of our thesis hypothesis as to build a reliable information system.

Chapter 7

Conclusion

In this chapter we will summarise the research findings of this thesis (Section 7.1), address its limitations (Section 7.2), and suggest future topics of research (Section 7.3).

7.1 Summary

This thesis presents the theory behind an approach to developing reliable information systems. We describe updates either as atomic operations, or as long-running workflows that represent patterns of these operations. We combine the emphasis of Model Driven Development (MDD) [1]—on abstraction and automation—with the rigour and analysis capabilities offered by formal methods [2]. This combination makes it possible to formalise and prove properties and transformations of models. Throughout the course of developing the thesis argument we referred to a typical information system—a hotel reservation system (HRS). All possible scenarios of relating entity classes have been addressed, and we have model operations which exemplify those of an information system—finite operations that perform simple transformations on the system data. Consequently, our HRS is sufficiently representative to support the research findings.

The results consist of theories, techniques, and tools in which the consistency of an IS model can be validated through testing. A provably correct implementation can be automatically generated from an IS model, and the efficient execution of complex workflows can be ensured. The roadmap of this thesis is constructed in Figure 1.5

(on page 16): each research problem is addressed in a corresponding chapter, and its results constitute a research contribution of this thesis. In the following sub-sections we will summarise how each contribution is situated in the development context of an information system (as illustrated in Figure 1.1 on page 3).

Specifying guarded transactions: the first contribution is presented in Chapter 3 which tests and verifies Hypothesis 2.1.1 (on page 21). We perform a formal analysis of a proposed modelling language—named BOOSTER—whose syntax resembles that of the standard first-order predicates, targeted at the model-driven development of information systems. A guarded substitution notation is adopted as the abstraction implementation language. We formally characterise its behaviour with a weakest pre-condition semantics, which reflects the policy of blocking that we wish to adopt at runtime. We then explore the logical consequence through proving a list of equivalence and refinement laws as summarised in Table 3.1 (on page 62). These laws provide a formal base for rewriting predicative specifications of operations. We demonstrate such rewriting on the HRS and interprete the meaning of each step of law application. We also suggest an application in the context of model-based testing: when an operation is considered as inconsistent and needs to be altered, we are able to rewrite, if applicable, its predicative specification into another, with a formal guarantee that its new behaviour is at the least a refinement, if not an equivalent.

Analysing parallel workflows: the second contribution is presented in Chapter 4 which tests and verifies Hypothesis 2.2.1 (on page 24). We define a workflow language whose semantics is built atop that of operations written as BOOSTER predicate methods. A formal semantics is given to this workflow language that includes not only the sequencing information, but also the data transformation constraints derived from the individual operations involved. We prove that this semantics is consistent, and that the operators in the language are appropriately compositional. We then specify and prove (Section 4.7) the correctness of a concrete calculation of a guard constraint for workflows that are executed in parallel. Should such a constraint be satisfied,

all the involved workflows are guaranteed to complete. We also apply the developed technique on the HRS.

Testing model consistency: the third contribution is presented in Chapter 5 which tests and verifies Hypothesis 2.3.1 (on page 26). We theorise how the semantics of operations and workflows can be used as the basis of model-level testing (of design models). By calculating the transformational semantics of the model, the developers are able to examine the precise circumstances—defined by constraints declared in the model under test—under which the given use-case scenarios are guaranteed to occur. Use-case scenarios that we allow the developers to specify include 1) data flows of attributes; 2) patterns of event occurrences; and 3) operations which commute. The developers may fix the relevant model constraints accordingly should they find any of the circumstances inconsistent with their intended requirements. This process may be repeated until no inconsistency is revealed. On the HRS we demonstrate our testing methodology and how the list of rewriting rules developed from Chapter 3 can be exploited to revise the behavioural model when needed.

Generating a working database: the fourth contribution is presented in Chapter 6 which tests and verifies Hypothesis 2.4.1 (on page 32). We define a transformation that allows automatic implementations of BOOSTER object models on a relational database platform. The novelty is our focus upon the automatic implementation of behavioural features, and upon how object paths can be efficiently implemented as nested queries through a simple caching mechanism. We demonstrate what the final product of our transformation engine looks like: a relational SQL database schema comprising definitions of tables and stored procedures, i.e. sequences of database queries. Furthermore, we give a relational semantics—which characterises transformations on model states—to operations in both object and database domains. We then exploit this semantics in order to establish that the BOOSTER-to-SQL transformation is correct.

7.2 Limitations

Guarded Transactions: for purposes of calculating preconditions for concurrent workflows either to achieve a specific goals (Section 4.7), e.g. completion, or to execute user-specific scenarios (Section 5.3.3), e.g. data flows of attributes, we assume that all model operations are *finite*, by forbidding the specification of unbounded loop and recursion. Furthermore, the list of equivalence and refinement laws that we prove on BOOSTER predicates is chosen on the sole basis of Gries' book [138]. The results as summarised in Table 3.1 (on page 62) constitute only a starting point: further rewriting rules will still have to be identified and proved.

Guarded Workflows: in Chapter 4, for the purpose of calculating a precondition for workflows that modify a shared state, the guarded workflow language does not support the synchronous parallel combinator; neither does it support constructs of recursive and bounded looping. Furthermore, we have assumed that all workflow instances are operating upon a single, sequential data component, which possesses a centralised control to interleave transactions that are initiated by distributed clients.

Model-Based Testing: in Chapter 5 the restriction criteria—data-flow of attributes, occurrences of event patterns, and commutativity of events—are most suitable in the context of abstract design models. In the setting of implementation, for imperative programming models, the state explosion problem may not be as efficiently alleviated by these criteria as we would expect on design models. Admittedly, there is still a lack of tool support for simplifying the generated predicates, for generating concrete test cases via certain techniques of constraint solving, and for tracing both operation traces and their associated preconditions back to their corresponding constraints in the source model. Nonetheless, the main contribution of this chapter is a formal underpinning of a theory on model-based testing.

Database Development: in Chapter 6 we focus on the formal proofs of critical functions that matter the most to the correctness of the transformation: paths, expressions, and program combinators. There are many small auxiliary functions, with

their intended functionalities implemented in a declarative manner and thus it is trivial to argue that they are correct. Alternatively, these functions have been validated through testing. However, the relational semantics and functional implementation have made it obvious to discharge the necessary proof obligations. Consequently, the work presented in this chapter constitutes an adequate justification of the correctness of the compiler. The listing of patterns of basic assignments on bi-associations also gives a valuable guideline on stating the proof obligations.

7.3 Future Work

In the following sections we suggest future work that may be undertaken to address the limitations.

Specification & Verification of Infinite Operations: we may relax our assumption that model operations are finite through introducing unbounded loop and recursion into the syntax of both our method language and workflow language. The immediate consequence is that our presented results on calculating preconditions for finite workflows and operations no longer apply. The *correct-by-construction* approach may be worth exploring to tackle the problem of infinite or recursive operations. A series of *refinement* steps will be applied to derive from an initial, abstract description of intended functionalities to an ultimate, concrete description that is sufficiently close for machine execution. On the one hand, the *theorem-proving* branch (e.g. [147]) will allow us to prove, with the aid of an automated or interactive theorem prover, that the list of critical properties is maintained between state changes made by operations, and that the resulting system is deadlock-free. On the other hand, the *model-checking* branch (e.g. [67]) will allow us to give process-oriented descriptions of how system components cooperate, and to check exhaustively that all possible states satisfy the intended safety and liveness properties that are encoded in temporal logic formulas. Not surprisingly, the exploration of either branch will have to deal with its own inherent problem: that any theorem prover cannot be complete, with re-

spect to the set of proof obligations it is capable of discharging, and that any model checker cannot deal with infinite types (e.g. real numbers) due to the problem of state explosion.

Distributed Context: although we have concentrated our study on the presence of a single BOOSTER machine, its design model may well be the *virtual* model of communicating components that are physically or geographically distributed. The focus then may be to reason about the consistency of our locking protocols at both the abstract and concrete levels: we ought to establish that any proposed, efficient runtime evaluations of preconditions, implemented through distributed algorithms, produce results as if all data items were stored in a single state component. Furthermore, as actual data components may be geographically distributed, we must also consider real-time constraints and incorporate them in our reasoning.

Tool Support: Although our theories on workflow preconditions and model-based testing are sound and the motivating scenarios of application are clearly indicated, mature tool support is necessary in order for the actual systems to benefit. In particular, the implementation of *traceability* in the context of our model-based testing, namely pairs of operation traces and their associated preconditions, requires more labour. For each referenced operation in the trace, there should be a link bringing us back to its original relational specification. For each generated precondition, assuming that it is a list of logical conjuncts, we should be able to link back to the source model constraint that caused the generation of a specific conjunct. However, all these become non-trivial when we are in a distributed context: for example, attributes that are involved in a specific precondition may be physically distributed and requires a robust mechanism for their evaluation. Furthermore, we may want to automate the term-rewriting process of our predicative specifications, through implementing a compiler that identifies applicable terms and performs symbolic executions accordingly, to generate equivalent or refined specifications. We shall expect new rules of predicate rewriting to be identified and evaluated for their validity via the same proof process.

References

[1] D. S. Frankel. *Model Driven Architecture: Applying MDA to Enterprise Computing*. Wiley, 2003.

[2] J. Woodcock, P. Gorm Larsen, J. Bicarregui, and J. Fitzgerald. Formal methods: Practice and experience. *ACM Computing Surveys (CSUR)*, 41:19:1–19:36, 2009.

[3] J. Davies, C. Crichton, E. Crichton, D. Neilson, and I. H. Sørensen. Formality, evolution, and model-driven software engineering. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 130:39–55, 2005.

[4] H.-E. Eriksson, M. Penker, and D. Fado. *UML 2 Toolkit*. Wiley, 2003.

[5] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. McGraw-Hill, 2003.

[6] C. B. Jones. *Systematic software development using VDM*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2nd edition, 1990.

[7] J. Woodcock and J. Davies. *Using Z*. Prentice Hall, 1996.

[8] OMG. Object Constraint Language, version 2.2. OMG document formal/2010-02-01, Object Management Group, 2010. `http://www.omg.org/spec/OCL/2.2/`.

[9] C.-W. Wang, J. Davies, and J. Welch. A guarded workflow language and its formal semantics. In *Theoretical Aspects of Software Engineering (TASE)*, pages 25–34. IEEE, 2010.

[10] L. Burdy, Y. Cheon, D. Cok, M. D. Ernst, J. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. *Software Tools for Technology Transfer*, 7(3):212–232, June 2005.

[11] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In *CASSIS 04': Construction and Analysis of Safe, Secure and Interoperable Smart devices*, volume 3362 of *LNCS*, pages 49–69, 2005.

[12] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, 1984.

[13] J. W. Laski and B. Korel. A data flow oriented program testing strategy. *IEEE Transactions on Software Engineering*, 9(3):347–354, 1983.

[14] C.-W. Wang and A. Cavarra. Checking model consistency using data-flow testing. In *Asia-Pacific Software Engineering Conference (APSEC)*, pages 414–421.

IEEE, 2009.

[15] C.-W. Wang, A. Cavarra, and J. Davies. Formal and model-based testing of concurrent workflows. In *International Conference on Quality Software (QSIC)*, pages 252–259. IEEE, 2011.

[16] Andrea Arcuri Shaukat Ali, Muhammad Zohaib Z. Iqbal and Lionel Briand. A search-based OCL constraint solver for model-based test data generation. In *International Conference on Quality Software (QSIC)*. IEEE, 2011.

[17] J. C. Bicarregui, C. A. R. Hoare, and J. C. P. Woodcock. The verified software repository: a step towards the verifying compiler. *Formal Aspects of Computing*, 18(2):143 – 151, 2006.

[18] J.-R. Abrial. *The B-book: assigning programs to meanings*. Cambridge University Press, 1996.

[19] J. O. Limb. Editor's introduction. *ACM Transactions on Information Systems (TOIS)*, 1(1):1–2, 1983.

[20] Editorial Charter. ACM Transactions on Information Systems. `http://tois.acm.org/charter.html`.

[21] Journal of Universal Computer Science. Information systems. `http://www.jucs.org/jucs_info/acm_categories`.

[22] C. Türker and M. Gertz. Semantic integrity support in SQL:1999 and commercial (object-)relational database management systems. *The VLDB Journal*, 10(4):241–269, 2001.

[23] W. Kent. A simple guide to five normal forms in relational database theory. *Communications of the ACM*, 26(2):120–125, 1983.

[24] G. Sanders and S. Shin. Denormalization effects on performance of RDBMS. In *Hawaii International Conference on System Sciences (HICSS)*, pages 9–17. IEEE, 2001.

[25] D. Costal, C. Gómez, A. Queralt, R. Raventós, and E. Teniente. Improving the definition of general constraints in UML. *Software and System Modeling*, 7(4):469–486, 2008.

[26] F. Devos and E. Steegmans. Specifying business rules in object-oriented analysis. *Software and System Modeling*, 4(3):297–309, 2005.

[27] F. P. Brooks. *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley Professional, 20th anniversary edition, August 1995.

[28] J.-M. Nicolas. Logic for improving integrity checking in relational data bases. *Acta Informatica*, 18:227–253, 1982.

[29] J. Cabot and E. Teniente. Computing the relevant instances that may violate an OCL constraint. In *Conference on Advanced Information Systems Engineering (CAiSE)*, volume 3520 of *LNCS*, pages 48–62, 2005.

[30] N. W. Paton and O. Díaz. Active database systems. *ACM Computing Surveys*

(CSUR), 31:63–103, 1999.

[31] Y. Zhou and M. Hsu. A theory for rule triggering systems. In *Extending Database Technology (EDBT)*, pages 407–421. Springer-Verlag, 1990.

[32] N. H. Gehani and H. V. Jagadish. ODE as an active database: Constraints and triggers. In *Very Large Data Bases (VLDB)*, pages 327–336. Morgan Kaufmann Publishers Inc., 1991.

[33] H. Ishikawa and K. Kubota. An active object-oriented database: A multi-paradigm approach to constraint management. In *Very Large Data Bases (VLDB)*, pages 467–478. Morgan Kaufmann Publishers Inc., 1993.

[34] J. Widom and S. J. Finkelstein. Set-oriented production rules in relational database systems. In *ACM SIGMOD international conference on Management of data*, pages 259–270. ACM, 1990.

[35] S. Ceri, P. Fraternali, S. Paraboschi, and L. Tanca. Automatic generation of production rules for integrity maintenance. *ACM Transactions on Database Systems (TODS)*, 19(3):367–422, 1994.

[36] K.-D. Schewe and B. Thalheim. Limitations of rule triggering systems for integrity maintenance in the context of transition specifications. *Acta Cybernetica*, 13(3):277–304, 1998.

[37] K.-D. Schewe and B. Thalheim. Towards a theory of consistency enforcement. *Acta Informatica*, 36(2):97–141, 1999.

[38] E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, 1975.

[39] G. Nelson. A generalization of Dijkstra's calculus. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 11(4):517–561, 1989.

[40] S. Link and K.-D. Schewe. Computability and decidability issues in the theory of consistency enforcement. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 42:174–196, 2001.

[41] D. Georgakopoulos, M. Hornick, and A. Sheth. An overview of workflow management: from process modeling to workflow automation infrastructure. *Distributed and Parallel Databases*, 3(2):119–153, 1995.

[42] U. Dayal, M. Hsu, and R. Ladin. A transactional model for long-running activities. In *Very Large Data Bases (VLDB)*, pages 113–122, 1991.

[43] H. Schuldt, G. Alonso, and H.-J. Schek. Concurrency control and recovery in transactional process management. In *Principles of database systems (PODS)*, pages 316–326. ACM, 1999.

[44] P. Grefen, J. Vonk, and P. Apers. Global transaction support for workflow management systems: from formal specification to practical implementation. *The VLDB Journal*, 10(4):316–333, 2001.

[45] W3C. Web Services Description Language (WSDL) 1.1. Technical report, World

Wide Web Consortium, 2001. http://www.w3.org/TR/wsdl.

[46] OMG. Business Process Modeling Notation (BPMN), Version 2.0. Technical report, Object Management Group, 2011. http://www.omg.org/spec/BPMN/.

[47] M. Dumas and A. H. M. ter Hofstede. UML activity diagrams as a workflow specification language. In *The Unified Modeling Language (UML)*, volume 2185 of *LNCS*, pages 76–90, 2001.

[48] W.M.P. van der Aalst and A. H. M. ter Hofstede. YAWL: yet another workflow language. *Information Systems*, 30(4):245–275, 2005.

[49] W3C. Web Service Choreography Interface (WSCI) 1.0. Technical report, World Wide Web Consortium, 2002. http://www.w3.org/TR/wsci/.

[50] OASIS. Web Services Business Process Execution Languages Version 2.0. Technical report, OASIS, 2007. http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html.

[51] WfMC. Process Definition Interface–XML Process Definition Language Version 2.1. Technical Document WFMC-TC-1025, Workflow Management Coalition, 2008. http://www.wfmc.org/xpdl.html.

[52] MSDN. Windows Workfow Foundation at .NET Framework 3.5. Online documentation, Microsoft Developer Network, 2007. http://msdn.microsoft.com/en-us/library/ms735967.aspx.

[53] Z. Hemel, R. Verhaaf, and E. Visser. WebWorkFlow: An object-oriented workflow modeling language for web applications. In *Model Driven Engineering Languages and Systems (MoDELS)*, volume 5301 of *LNCS*, pages 113–127, 2008.

[54] W. L. Yeung. Mapping WS-CDL and BPEL into CSP for behavioural specification and verification of web services. In *European Conference on Web Services (ECOWS)*, pages 297–305. IEEE, 2006.

[55] P. Y. H. Wong and J. Gibbons. Property specifications for workflow modelling. In *Integrated Formal Methods (IFM)*, volume 5423 of *LNCS*, pages 56–71, 2009.

[56] R. M. Dijkman, M. Dumas, and C. Ouyang. Semantics and analysis of business process models in BPMN. *Information and Software Technology*, 50(12):1281–1294, 2008.

[57] C. Ouyang, E. Verbeek, W. M. P. van der Aalst, S. Breutel, M. Dumas, and A. H. M. ter Hofstede. Formal semantics and analysis of control flow in WS-BPEL. *Science of Computer Programming*, 67(2-3):162–198, 2007.

[58] H. Dun, H. Xu, and L. Wang. Transformation of BPEL processes to Petri Nets. In *Theoretical Aspects of Software Engineering (TASE)*, pages 166–173, 2008.

[59] M. Rouached, W. Fdhila, and C. Godart. A semantic framework to engineering WSBPEL processes. *Information Systems and E-Business Management*, 7:223–250, 2009.

[60] G. Pu, X. Zhao, S. Wang, and Z. Qiu. Towards the semantics and verification

of BPEL4WS. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 151(2):33–52, 2006.

[61] A. Brogi and R. Popescu. Workflow semantics of peer and service behaviour. In *Theoretical Aspects of Software Engineering (TASE)*, pages 143–150, 2008.

[62] A. Brogi and R. Popescu. From BPEL processes to YAWL workflows. In *Web Services and Formal Methods (WS-FM)*, volume 4184 of *LNCS*, 2006.

[63] C. Luo, S. Qin, and Z. Qiu. Verifying BPEL-like programs with Hoare logic. In *Theoretical Aspects of Software Engineering (TASE)*, pages 151–158, 2008.

[64] W. M. P. Van Der Aalst, A. H. M. Ter Hofstede, B. Kiepuszewski, and A. P. Barros. Workflow patterns. *Distributed and Parallel Databases*, 14(1):5–51, 2003.

[65] N. Russell, A.H.M. ter Hofstede, D. Edmond, and W.M.P. van der Aalst. Workflow data patterns. Technical Report FIT-TR-2004-01, Queensland University of Technology, 2004.

[66] N. Russell, A.H.M. ter Hofstede, W.M.P. van der Aalst, and N. Mulyar. Workflow control-flow patterns: A revised view. BPM Center Report BPM-06-22, BPMcenter.org, 2006. `http://www.workflowpatterns.com/`.

[67] J. Sun, Y. Liu, J. S. Dong, and C. Chen. Integrating specification and programs for system modeling and verification. In *Theoretical Aspects of Software Engineering (TASE)*, pages 127–135, 2009.

[68] J. Sun, Y. Liu, J. S. Dong, and J. Sun. Bounded model checking of compositional processes. In *Theoretical Aspects of Software Engineering (TASE)*, pages 23–30, 2008.

[69] S. Schneider and H. Treharne. CSP theorems for communicating B machines. *Formal Aspects of Computing*, 17(4):390–422, 2005.

[70] Jim Woodcock and Ana Cavalcanti. A concurrent language for refinement. In *5th Irish Workshop on Formal Methods (IWFM)*, 2001.

[71] P. Ammann and J. Offutt. *Introduction to Software Testing*. Cambridge University Press, 2008.

[72] B. Meyer, I. Ciupa, A. Leitner, and L. L. Liu. Automatic testing of object-oriented software. In *Current Trends in Theory and Practice of Computer Science (SOFSEM)*, volume 4362 of *LNCS*, pages 114–129, 2007.

[73] Y. Singh and A. Saha. Enhancing data flow testing of classes through design by contract. In *International Conference on Computer and Information Science (ICIS)*, pages 567–574. IEEE, 2008.

[74] G. Dai, X. Bai, Y. Wang, and F. Dai. Contract-based testing for web services. In *Computer Software and Applications Conference (COMPSAC)*, pages 517–526. IEEE, 2007.

[75] R. Heckel and M. Lohmann. Towards contract-based testing of web services.

Electronic Notes in Theoretical Computer Science (ENTCS), 82(6):2003, 2004.

[76] T. Ball, D. Hoffman, F. Ruskey, R. Webber, and L. White. State generation and automated class testing. *Software Testing, Verification and Reliability*, 10(3):149–170, 2000.

[77] J. Offutt, S. Liu, A. Abdurazik, and P. Ammann. Generating test data from state-based specifications. *Software Testing, Verification and Reliability*, 13(1):25–53, 2003.

[78] R.N. Taylor, D.L. Levine, and C.D. Kelly. Structural testing of concurrent programs. *IEEE Transactions on Software Engineering*, 18(3):206 – 215, 1992.

[79] A. W. Roscoe. Compiling shared variable programs into CSP. In *PROGRESS workshop*, 2001.

[80] B. Lei, L. Wang, and X. Li. UML activity diagram based testing of Java concurrent programs for data race and inconsistency. In *International Conference on Software Testing, Verification, and Validation (ICST)*, pages 200–209, 2008.

[81] C. Schwarzl and B. Peischl. Test sequence generation from communicating UML state charts: An industrial application of symbolic transition systems. In *International Conference on Quality Software (QSIC)*, pages 122 –131, 2010.

[82] S.-K. Kim, L. Wildman, and R. Duke. A UML approach to the generation of test sequences for Java-based concurrent systems. In *Australian Software Engineering Conference (ASWEC)*, pages 100 – 109, 2005.

[83] S.-S. Hou, L. Zhang, Q. Lan, H. Mei, and J.-S. Sun. Generating effective test sequences for BPEL testing. In *International Conference on Quality Software (QSIC)*, pages 331 – 340, 2009.

[84] Y.-H. Tung, S.-S. Tseng, T.-J. Lee, and J.-F. Weng. A novel approach to automatic test case generation for web applications. In *International Conference on Quality Software (QSIC)*, pages 399 – 404, 2010.

[85] Y. Chen, S. Liu, and L. Wang. An extension to data-flow-oriented formal specification language for specifying concurrent software systems. In *International Conference on Quality Software (QSIC)*, pages 214–219, 2010.

[86] O.-J. Dahl and K. Nygaard. SIMULA: an ALGOL-based simulation languages. *Communications of the ACM*, 9(9):671–678, 1966.

[87] R. B. France, S. Ghosh, T. Dinh-Trong, and A. Solberg. Model-driven development using UML 2.0: Promises and pitfalls. *Computer*, 39(2):59–66, 2006.

[88] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, 2nd edition, 2000.

[89] D. C. Schmidt. Model-driven engineering. *IEEE Computer*, 39(2):25 – 31, 2006.

[90] C. Atkinson and T. Kühne. Model-driven development: A metamodeling foundation. *IEEE Software*, 20(5):36–41, 2003.

[91] OMG. MDA guide version 1.0.1. OMG document omg/2003-06-01, Object

Management Group, 2003. http://www.omg.org/mda/specs.htm.

[92] OMG. Model Driven Architecture – a technical perspective. OMG document ormsc/01-07-01, Object Management Group, 2001. http://www.omg.org/mda/specs.htm.

[93] OMG. Meta Object Facility (MOF) Specification 2.4.1. OMG document formal/2011-08-07, Object Management Group, 2011. http://www.omg.org/mof/.

[94] OMG. MOF/XMI Mapping, version 2.4.1. OMG document formal/2011-08-09, Object Management Group, 2007. http://www.omg.org/spec/XMI/.

[95] OMG. Common Warehouse Metamodel (CWM) Specification. OMG document formal/03-03-02, Object Management Group, 2003. Version 1.1: http://www.omg.org/spec/CWM/.

[96] OMG. MOF 2.0 Query/Views/Transformations (QVT). OMG document formal/2011-01-01, Object Management Group, 2002. http://www.omg.org/spec/QVT.

[97] OMG. Revised submission for MOF 2.0 query/views/transformation rfp (ad/2002-04-10). OMG document ad/2005-03-02, Object Management Group, 2005. Submitted by QVT-Merge Group: http://www.omg.org/docs/ad/05-03-02.pdf.

[98] F. Budinsky, D. Steinberg, E. Merks, R. Ellersick, and T. J. Grose. *Eclipse Modeling Framework*. the eclipse series. Addison-Wesley Professional, 2003.

[99] D. S. Kolovos, R. F. Paige, and F. A.C. Polack. The Epsilon Object Language (EOL). In *European Conference in Model Driven Architecture (EC-MDA)*, 2006.

[100] L. M. Rose, R. F. Paige, D. S. Kolovos, and F. Polack. The Epsilon Generation Language. In *European Conference on Modelling Foundations and Applications (ECMDA-FA)*, pages 1–16, 2008.

[101] F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev. ATL: A model transformation tool. *Science of Computer Programming*, 72(1–2):31–39, 2008.

[102] A. Queralt and E. Teniente. A platform independent model for the electronic marketplace domain. *Software and System Modeling*, 7(2):219–235, 2008.

[103] J. E. Pérez-Martínez and A. Sierra-Alonso. From Analysis Model to Software Architecture: A PIM2PIM Mapping. In *European Conference on Modelling Foundations and Applications (ECMDA-FA)*, pages 25–39, 2006.

[104] R. Grønmo, M. C. Jaeger, and H. Hoff. Transformations between UML and OWL-S. In *European Conference on Modelling Foundations and Applications (ECMDA-FA)*, pages 269–283, 2005.

[105] P. Kelsen, E. Pulvermueller, and C. Glodt. Specifying executable platform-independent models using OCL. *ECEASST*, 9, 2008.

[106] K. Jiang, L. Zhang, and S. Miyake. Using OCL in executable UML. *ECEASST*,

9, 2008.

[107] S. J. Mellor and M. Balcer. *Executable UML: A Foundation for Model-Driven Architectures*. Addison-Wesley, 2002.

[108] C. Raistrick, P. Francis, and J. Wright. *Model Driven Architecture with Executable UML(TM)*. Cambridge University Press, 2004.

[109] A. K. Bhattacharjee and R. K. Shyamasundar. Activity diagrams: A formal framework to model business processes and code generation. *Journal of Object Technology*, 8(1):189–220, 2009.

[110] G. Engels, A. Kleppe, A. Rensink, M. Semenyak, C. Soltenborn, and H. Wehrheim. From UML activities to TAAL — towards behaviour-preserving model transformations. In *European Conference on Modelling Foundations and Applications (ECMDA-FA)*, volume 5095 of *LNCS*, pages 94–109, 2008.

[111] F. Barbier. Supporting the UML state machine diagrams at runtime. In *European Conference on Modelling Foundations and Applications (ECMDA-FA)*, volume 5095 of *LNCS*, pages 338–348, 2008.

[112] T. Waheed, M. Z. Z. Iqbal, and Z. I. Malik. Data flow analysis of UML action semantics for executable models. In *European Conference on Modelling Foundations and Applications (ECMDA-FA)*, volume 5095 of *LNCS*, pages 79–93, 2008.

[113] L. Fuentes and P. Sánchez. Execution of aspect oriented UML models. In *European Conference on Modelling Foundations and Applications (ECMDA-FA)*, volume 4530 of *LNCS*, pages 83–98, 2007.

[114] S. Haustein and J. Pleumann. A model-driven runtime environment for web applications. *Software and System Modeling*, 4(4):443–458, 2005.

[115] A. Bercovici, F. Fournier, and A. J. Wecker. From business architecture to SOA realization using MDD. In *European Conference on Modelling Foundations and Applications (ECMDA-FA)*, volume 5095 of *LNCS*, pages 381–392, 2008.

[116] M. Banci, A. Fantechi, S. Gnesi, and G. Lombardi. Model driven development and code generation: An automotive case study. In *Design for Dependable Systems (SDL)*, volume 4745 of *LNCS*, pages 19–34, 2007.

[117] M. L. Crane and J. Dingel. Towards a formal account of a foundational subset for executable UML models. In *Model Driven Engineering Languages and Systems (MoDELS)*, volume 5301 of *LNCS*, pages 675–689, 2008.

[118] K. Jiang, L. Zhang, and S. Miyake. An executable UML with OCL-based action semantics language. In *Asia-Pacific Software Engineering Conference (APSEC)*, pages 302–309. IEEE, 2007.

[119] D. H. Akehurst, G. Howells, and K. D. McDonald-Maier. Implementing associations: UML 2.0 to Java 5. *Software and System Modeling*, 6(1):3–35, 2007.

[120] D. Gessenharter. Mapping the UML2 semantics of associations to a Java code generation model. In *Model Driven Engineering Languages and Systems (MoD-*

ELS), volume 5301 of *LNCS*, pages 813–827, 2008.

[121] C. Ouyang, W. M.P. van der Aalst, M. Dumas, and A. H.M. ter Hofstede. From business process models to process-oriented software systems: The BPMN to BPEL way. Technical report, Queensland University of Technology, 2006. http://eprints.qut.edu.au/archive/00005266/.

[122] C. Ouyang, M. Dumas, A. H. M. ter Hofstede, and W. M. P. van der Aalst. From BPMN process models to BPEL web services. In *International Conference on Web Services (ICWS)*, pages 285–292, 2006.

[123] T. Hornung, A. Koschmider, and J. Mendling. Integration of heterogeneous BPM schemas: The case of XPDL and BPEL. In *Conference on Advanced Information Systems Engineering (CAiSE)*, pages 23–26, 2006.

[124] B. Bordbar, G. Howells, M. Evans, and A. Staikopoulos. Model transformation from OWL-S to BPEL via SiTra. In *European Conference on Modelling Foundations and Applications (ECMDA-FA)*, volume 4530 of *LNCS*, pages 43–58, 2007.

[125] X. Yu, Y. Zhang, T. Zhang, L. Wang, J. Zhao, G. Zheng, and X. Li. Towards a model driven approach to automatic BPEL generation. In *European Conference on Modelling Foundations and Applications (ECMDA-FA)*, volume 4530 of *LNCS*, pages 204–218, 2007.

[126] B. Karakostas, Y. Zorgios, and C. C. Alevizos. Automatic derivation of BPEL4WS from IDEF0 process models. *Software and System Modeling*, 5(2):208–218, 2006.

[127] C. Russell. Bridging the object-relational divide. *ACM Queue*, 6(3):18–28, 2008.

[128] ORACLE. Enterprise JavaBeans Technology. http://www.oracle.com/technetwork/java/index.html.

[129] JBoss Community. Hibernate — relational persistence for Java and .NET. http://www.hibernate.org/.

[130] ORACLE. Java Data Objects API: Java model abstraction of persistence. http://www.oracle.com/technetwork/java/index.html.

[131] A. Adya, J. A. Blakeley, S. Melnik, and S. Muralidhar. Anatomy of the ADO.NET Entity Framework. In *ACM SIGMOD International Conference on Management of Data*, pages 877–888, 2007.

[132] A. Mammar. A systematic approach to generate B preconditions: application to the database domain. *Software Systems Modeling*, 8(3):385–401, 2009.

[133] A. Mammar and R. Laleau. A formal approach based on UML and B for the specification and development of database applications. *Automated Software Engineering*, 13(4):497–528, 2006.

[134] A. Mammar and R. Laleau. From a B formal specification to an executable code — application to the relational database domains. *Information and Software*

Technology, 48(4):253–279, 2006.

[135] J. Welch, D. Faitelson, and J. Davies. Automatic maintenance of association invariants. *Software and Systems Modeling*, 7(3):287–301, 2008.

[136] D. Faitelson, J. Welch, and J. Davies. From predicates to programs: The semantics of a method language. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 184:171–187, 2007.

[137] P. Gardiner and C. Morgan. Data refinement of predicate transformers. *Theoretical Computer Science*, 87:143–162, 1991.

[138] D. Gries and F. B. Schneider. *A logical approach to discrete math.* Springer-Verlag, 1993.

[139] A. W. Roscoe. *The Theory and Practice of Concurrency.* Prentice Hall PTR, 1997.

[140] G. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, University of Aarhus, 1981.

[141] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21:666–677, 1985.

[142] E. M. Clarke Jr., O. Grumberg, and D. A. Peled. *Model Checking.* MIT Press, 1999.

[143] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms.* MIT Press, 2001.

[144] R. Bird. *Introduction to Functional Programming using Haskell.* Prentice Hall, 1998.

[145] R. Lämmel and S. P. Jones. Scrap your boilerplate: a practical design pattern for generic programming. In *International Workshop on Types in Languages Design and Implementation (TLDI)*, pages 26–37. ACM, 2003.

[146] W. DeRoever and K. Engelhardt. *Data Refinement: Model-Oriented Proof Methods and Their Comparison.* Cambridge University Press, 1999.

[147] J.-R. Abrial. *Modeling in Event-B: System and Software Engineering.* Cambridge University Press, 2010.

Appendix A

Appendix to Chapter 2

A.1 Grammar of the BOOSTER Language

In describing the language of BOOSTER, we will adopt the following notation: keywords and symbols are in boldface; terminals are in uppercase; non-terminals are capitalised and italics; | denotes alternation; ? denotes an optional instance, * for an arbitrary number of instances, and + for at least one instance.

A model in BOOSTER consists of a single **system** declaration, followed by a system name, an optional system-level invariant, and a number of **set** and **class** declarations:

```
System ::= system IDENTIFIER {
             Invariant?
             (Set | Class)*
           }
```

System-level invariants are unnamed predicates:

```
Invariant ::= invariant { Predicate }
```

Each set declaration introduces a named list of identifiers:

```
Set ::= set IDENTIFIER { IDENTIFIER (, IDENTIFIER)* }
```

Classes and Attributes

Each class declaration begins with its name and an optional list of superclasses, followed by a list of declarations for *attributes*, *methods*, *invariants*, and *named constraints*:

```
Class ::= class IDENTIFIER ( extends IDENTIFIER (, IDENTIFIER)* )? {
            ( attributes    Attribute+      )?
            ( methods       Method+         )?
            ( invariant   { Predicate }     )?
            ( constraints   NamedPredicate+ )?
          }
```

Named constraints are enclosed within round parentheses, to be substituted as macros within definitions of other predicates:

```
NamedPredicate ::= IDENTIFIER = ( Predicate )
```

A standard attribute declaration includes a name and a type, but may also include a list of methods. We have found it useful to declare a method not merely within the context of a class, but also within the context of an individual attribute. This is most obviously true of the basic methods that update the attribute itself: those that set or clear an optional attribute; those that add or remove an element from a set-valued attribute; those that update a reference-valued attribute while also creating or deleting a referenced object.

```
Attribute ::= IDENTIFIER : Type ( { Method+ } )?
            |  IDENTIFIER = Expression
```

A *derived* attribute is declared using = and without a type: its type will be determined by the value of the specified expression.

There are three categories of type in BOOSTER: *simple* type, *collection* type, and *optional* type:

```
Type           ::= SimpleType | OptionalType | CollectionType

SimpleType     ::= int | string | IDENTIFIER | ReferenceType
OptionalType   ::= [ SimpleType ]
CollectionType ::= ( set | seq ) ( SimpleType ) [ Cardinality ]

ReferenceType  ::= IDENTIFIER . IDENTIFIER
Cardinality    ::= NUMBER | *
                |  NUMBER .. NUMBER | .. NUMBER | NUMBER .. | NUMBER .. *
```

Associations in BOOSTER models are normally bi-directional: that is, any implementation consists of a pair of matching links, corresponding to a pair of opposing attributes. Thus a `ReferenceType` has a pair of identifiers: the first being the name of the class referenced, the second being the name of the opposite attribute. References to user-defined sets are normally uni-directional, and are made using a single `IDENTIFIER`.

A collection may be a set (`set`), without ordering or repetition, or a sequence (`seq`); the elements of a collection must be of `SimpleType`. We must specify the intended cardinality of a collection using a conventional `..` and * notation. An attribute of optional type is rather like a collection of cardinality `0..1`; however, its value is referred to using the attribute name directly, rather than extracting the contents of a singleton, and if it is empty, it has the special value `null`.

Predicates and methods

Our use of an information system places an emphasis on simple transformations on its data, subject to its set of structural and semantic integrity constraints, rather than

sophisticated algorithms. We thus characterise the effect of each model operation as a transformation on the system state: a mathematical predicate that relates values of the modified data items after the operation to those beforehand. This transformational style of specification is also adopted in formal state-based languages such as VDM [6] and Z [7]; it is also the basis for the constraint-oriented specification of operations in UML [4], where class diagrams are annotated with OCL [8] constraints that describe post-conditions.

Methods are declared simply as named predicates:

```
Method ::= IDENTIFIER { Predicate? }
```

Predicates may be described using *logical* and *relational* operators:

```
Predicate ::= true | false
            | not Predicate
            | Predicate ( & | or | => | ; ) Predicate
            | ( forall | exists ) IDENTIFIER : Expression @ Predicate
            | Relation
            | ( Predicate )
            | IDENTIFIER  /* named constraints */
            | MethodReference
```

The sequential composition operator allows a transaction to be built up through a number of phases, and hence through a number of intermediate states. These states are not required to satisfy the user-specified integrity constraints, but they must satisfy any representational constraints arising from the choice of platform technology (for example, the requirement that an integer-valued attribute should not be assigned a value greater than "maxint").

The standard equality and arithmetic operators are included, using the same syntax as AMN

```
Relation       ::= Expression ( NumberRelation | SetRelation ) Expression
NumberRelation ::= <= | >= | <   | >   | = | /=
SetRelation    ::= <: | :> | <<: | :>> | : | /:

Expression ::= ValueExpression
             | ( Expression )
             | - Expression
             | Expression ( NumberOp | ++ | \/ | /\ ) Expression
             | ( card | head | tail ) Expression
NumberOp   ::= + | - | * | / | min
```

An expression may refer to the value of a primitive, a collection, or an attribute of the model, expressed as a *Path*.

```
ValueExpression ::= STRING | NUMBER | string | int | null
                  |  SetExpression | SeqExpression
                  |  Path
SetExpression   ::= { (Expression)* }
```

```
SeqExpression   ::= < (Expression)* >
Path            ::= PathStart
                  | PathStart . PathComponent ( . PathComponent )*
PathStart       ::= this
                  | IDENTIFIER ( ? | ! | ' )? Index?
PathComponent   ::= IDENTIFIER Index?
Index           ::= [ Expression ]
```

We use the conventional name `this` to denote the current object, and *Index* to denote sequence indexing, if applicable. Set expressions are specified as comma-separated list of expressions enclosed within curly braces (i.e. `{}`) and angle brackets (i.e. <>) for sequence expressions. Inputs to an operation are introduced as free variables decorated with question marks `?`, and outputs—or new objects to be created—are introduced as free variables decorated with exclamation marks `!`.

Method references represent partial applications of input values for the referenced methods:

```
MethodReference    ::= Path InputSubstitutions?
InputSubstitutions ::= ( InputSubstitution ( & InputSubstitution )* )
InputSubstitution  ::= IDENTIFIER? = Expression
```

A.2 Hotel Reservation System in BOOSTER

```
system HotelReservation {

  class Hotel {
    attributes
      name         : string
      permit       : Licence . licensee
      rooms        : seq(Room        . host     ) [*]
      reservations : seq(Reservation . host     ) [*]
      allocations  : set(Allocation  . host     ) [*]
      registered   : set(Traveller   . reglist  ) [*]
      consultants  : seq(Staff       . clients  ) [*]
      employees    : seq(Staff       . employers) [*]
    methods
      renew {
           lic? : extent(Licence)
        ==>
           -- assign to one-to-one
           permit        := lic? -- #1
        || lic?.licensee := this -- #1
      }
      hire {
           s? : extent(Staff)
        ==>
           -- insert into seq-to-seq
           employees    := ins(employees, #employees + 1, s?)         -- #27
           ||
           s?.employers := ins(s?.employers, #s?.employers + 1, this) -- #27
      }
      contract {
           s? : extent(Staff)
        &  s? : ran(employees)
        ==>
           -- insert into set-to-seq
           s?.clients  := s?.clients \/ {this}                   -- #19
           ||
           consultants := ins(consultants, #consultants + 1, s?) -- #25
      }
      contracts {
           s? : set(Staff)
        &  forall x : s? @ (x.mentor /= null & x : ran(employees))
        ==>
           -- insert into set-to-seq
           (ALL x : s? @ x.clients  := x.clients \/ {this})      -- #19
           ||
```

```
     (ALL x : s? @ consultants := ins(consultants, #consultants + 1, x)) -- #25
}
reserve {
     r!     : extent(Reservation)
  &  dates? : set(Date)
  &  m?     : extent(Room)
  &  #allocations < 100
  ==>
     r!.dates        := r!.dates \/ dates?                                -- #34
  || r!.status       := "unconfirmed"                                     -- #29
     -- insert into one-to-seq
  || r!.host         := this                                              -- #4
  || reservations    := ins(reservations, #reservations + 1, r!)          -- #21
     -- insert into opt-to-seq
  || r!.room         := m?                                                -- #11
  || m?.reservations := ins(m?.reservations, #m?.reservations + 1, r!) -- #23
}
cancel {
     r? : extent(Reservation)
  &  r? : reservations
  ==>
     r?.dates := r?.dates - r?.dates              -- #35
     -- remove from one-to-seq
  || reservations         :=
        del(reservations,
              indexOf(reservations, r?))          -- #22
     -- remove from opt-to-seq
  || r?.room              := null                 -- #12
  || r?.room.reservations :=
        del(r?.room.reservations,
              indexOf(r?.room.reservations, r?))  -- #24
}
allocate {
     r? : extent(Reservation)
  &  r? : reservations
  &  r?.status = "confirmed"
  &  m? : extent(Room)
  &  #allocations < 100
  &  a! : extent(Allocation)
  ==>
     ALL x : r?.dates @ (a!.dates := a!.dates \/ {x})
     -- insert into one-to-set
  || a!.host        := this                    -- #3
  || allocations    := allocations \/ {a!}     -- #13
     -- insert into opt-to-set
  || a!.room        := m?                      -- #9
  || m?.allocations := m?.allocations \/ {a!} -- #15
```

```
    }
    deallocate {
         a? : extent(Allocation)
      &  a? : allocations
      ==>
         -- remove from one-to-set
         allocations         := allocations - {a?}         -- #14
         -- remove from opt-to-set
      || a?.room             := null                      -- #10
      || a?.room.allocations := a?.room.allocations - {a?} -- #16
    }
    register {
         t? : extent(Traveller)
      &  t? /: registered
      ==>
         -- insert into set-to-set
         registered := registered \/ {t?}   -- #17
      || t?.reglist := t?.reglist \/ {this} -- #17
    }
    unregister {
         t? : extent(Traveller)
      &  t? : registered
      ==>
         -- remove from set-to-set
         registered := registered - {t?}   -- #18
      || t?.reglist := t?.reglist - {this} -- #18
    }
 }

class Licence {
  attributes
    id       : string
    licensee : Hotel . permit
}

class Staff {
  attributes
    id        :  string
    name      :  string
    clients   :  set(Hotel . consultants)[*]
    employers :  seq(Hotel . employees  )[*]
    mentor    : [ Staff . mentee ]
    mentee    : [ Staff . mentor ]
 methods
    assignMentor {
         men? : extent(Staff)
      &  men? /= this
```

```
      &  this.mentor = null
      &  men?.mentee = null
      ==>
         -- assign to opt-to-opt
         mentor      := men? -- #7
      || men?.mentee := this -- #7
   }
   changeMentor {
        men? : extent(Staff)
      &  men? /= this
      &  mentor /= null
      &  mentor /= men?
      &  men?.mentee /= null => men?.mentee /= this
      ==>
         -- delete opt-to-opt
      (   mentor        := null  -- #8
       || mentor.mentee := null) -- #8
      ;
      (   mentor        := men?  -- #7
       || mentee        := this) -- #7
   }
   resign { -- resign from a job
        i? : int
      &  j? : int
      &  i? : dom(employers)
      &  i? = employers[ j? ]
      &  j? : dom(employers[ i? ].employees)
      &  this = employers[ i? ].employees[ j? ]
      ==>
      -- remove from seq-to-seq
      employers                 := del(employers, i?)                 -- #28
      ||
      employers[ i? ].employees := del(employers[ i? ].employees, j?) -- #28
   }
   terminate { -- terminate an existing contract of consulting
        i? : int
      &  i? : dom(h?.consultants)
      &  this = clients.consultants[ i? ]
      &  h? : extent(Hotel)
      &  h? : clients
      ==>
         -- remove from set-to-seq
         clients        := clients - {h?}          -- #20
         ||
         h?.consultants := del(h?.consultants, i?) -- #26
   }
}
```

```
class Reservation {
  attributes
    host   : Hotel . reservations
    room   : [ Room . reservations ]
    status : string
    dates  : set(Date) [*]
}

class Allocation {
  attributes
    host  : Hotel . allocations
    room  : [ Room . allocations ]
    dates : set(Date) [*]
  methods
    extend { -- extend the tenure
         d? : extent(Date)
      &  e? : extent(Date)
      &  d? /: dates
      &  e? /: dates
      ==>
      dates := dates \/ {d?, e?} -- #32
    }
    shorten { -- shorten the tenure
         d? : extent(Date)
      &  e? : extent(Date)
      &  d? : dates
      &  e? : dates
      ==>
      dates := dates - {d?, e?}  -- #33
    }
}

class Room {
  attributes
    host         : Hotel.rooms
    id           : string
    available    : set(Date) [*]
    reservations : seq(Reservation . room) [*]
    allocations  : set(Allocation  . room) [*]
}

class Traveller {
  attributes
    name    : string
    reglist : set(Hotel.registered) [*]
    account : [ Account . owner ]
```

```
      address : [ string ]
    methods
      activate {
           amount? : int
        &  amount? > 0
        &  acc! : extent(Account)
        ==>
           acc!.balance := amount?  -- #29
           -- assign to one-to-opt
        || account      := acc!     -- #2
        || acc!.owner   := this     -- #5
      }
      deactivate {
        account := null             -- #6
      }
  }

  class Account {
    attributes
      owner         : Traveller . account
      balance       : int
      beneficiaries : seq(Account)[*]
   methods
      deposit {
           amount? : int
        &  amount? > 0
        ==>
           balance := balance + amount? -- #29
      }
      withdraw {
           amount? : int
        &  amount? <= balance
        ==>
           balance := balance - amount? -- #29
      }
      change {
           acc? : extent(Account)
        &  i? : int
        &  i? : dom(beneficiaries)
        ==>
           beneficiaries[ i? ] := acc?
      }
  }
}
```

Appendix B

Appendix to Chapter 3

B.1 Proofs of Substitution Laws

Proof of Law 3.4.1 As we are provided that $S \sqsubseteq U$ and $T \sqsubseteq V$, by the definition of $\sqsubseteq$ on substitutions we obtain

$$\forall\, J : \mathit{Predicate} \bullet \mathit{wp}\,(U,\ J) \Rightarrow \mathit{wp}\,(S,\ J) \wedge \mathit{wp}\,(V,\ J) \Rightarrow \mathit{wp}\,(T,\ J)$$

Using this assumption, we argue the case of $\|$ as follows:

$$\begin{array}{ll} & \mathit{wp}\,(U \parallel V,\ I) \\ \Leftrightarrow & \{\ \texttt{def. of}\ \mathit{wp}\ \texttt{on}\ \| \ \} \\ & \mathit{wp}\,(U,\ \mathit{wp}\,(V,\ I)) \wedge \mathit{wp}\,(V,\ \mathit{wp}\,(U,\ I)) \\ \Leftrightarrow & \{\ \mathit{wp}\,(U,\ J) \Rightarrow \mathit{wp}\,(S,\ J)\ \texttt{and}\ \mathit{wp}\,(V,\ J) \Rightarrow \mathit{wp}\,(T,\ J)\ \texttt{for all}\ J\ \} \\ & \mathit{wp}\,(S,\ \mathit{wp}\,(V,\ I)) \wedge \mathit{wp}\,(T,\ \mathit{wp}\,(U,\ I)) \\ \Rightarrow & \{\ \mathit{wp}\,(U,\ I) \Rightarrow \mathit{wp}\,(S,\ I)\ \texttt{and}\ \mathit{wp}\,(V,\ I) \Rightarrow \mathit{wp}\,(T,\ I)\texttt{, Property 3.3.1 twice}\ \} \\ & \mathit{wp}\,(S,\ \mathit{wp}\,(T,\ I)) \wedge \mathit{wp}\,(T,\ \mathit{wp}\,(S,\ I)) \\ \Leftrightarrow & \{\ \texttt{def. of}\ \mathit{wp}\ \texttt{on}\ \| \ \} \\ & \mathit{wp}\,(S \parallel T,\ I) \end{array}$$

This monotonicity is generalised to that of !. Similar steps of reasoning apply to other combinators. □

Proof of Law 3.4.4

$$\begin{array}{ll} & \mathit{wp}\,(S \,\square\, (\mathit{false} \longrightarrow T),\ I) \\ \Leftrightarrow & \{\ \mathit{wp}\ \texttt{on}\ \longrightarrow\ \texttt{then on}\ \square\ \} \\ & \mathit{wp}\,(S,\ I) \vee (\mathit{false} \wedge \mathit{wp}\,(T,\ I)) \\ \Leftrightarrow & \{\ \texttt{predicate logic: zero of}\ \wedge\texttt{, identity of}\ \vee\ \} \\ & \mathit{wp}\,(S,\ I) \end{array}$$

□

Proof of Law 3.4.5

$$
\begin{array}{ll}
 & wp\,(S \parallel skip,\ I) \\
\Leftrightarrow & \{\ \texttt{def. of}\ wp\ \texttt{on}\ \parallel\ \} \\
 & wp\,(S,\ wp\,(skip,\ I)) \wedge wp\,(skip,\ wp\,(S,\ I)) \\
\Leftrightarrow & \{\ \texttt{def. of}\ wp\ \texttt{on}\ skip\ \texttt{twice}\ \} \\
 & wp\,(S,\ I) \wedge wp\,(S,\ I) \\
\Leftrightarrow & \{\ \texttt{predicate logic: idempotency of}\ \wedge\ \} \\
 & wp\,(S,\ I)
\end{array}
$$

□

Proof of Law 3.4.6 We prove by definitions of wp on $\longrightarrow$. □

Proof of Law 3.4.7 We argue case (a) as follows:

$$
\begin{array}{ll}
 & wp\,(p \longrightarrow S,\ I) \\
\Leftrightarrow & \{\ \texttt{def. of}\ wp\ \texttt{on}\ \longrightarrow\ \} \\
 & p \wedge wp\,(S,\ I) \\
\Rightarrow & \{\ p \Rightarrow q\texttt{, predicate logic: monotonicity of}\ \wedge\ \} \\
 & q \wedge wp\,(S,\ I) \\
\Leftrightarrow & \{\ \texttt{def. of}\ wp\ \texttt{on}\ \longrightarrow\ \} \\
 & wp\,(q \longrightarrow S,\ I)
\end{array}
$$

Cases (b) and (c) immediately from (a) for the theorems of strengthening/weakening in predicate logic [138] (i.e. $p \Rightarrow p \vee q$ and $p \wedge q \Rightarrow p$). □

Proof of Law 3.4.9

$$
\begin{array}{ll}
 & (p \vee q) \longrightarrow S \,\Box\, T \\
= & \{\ \texttt{Law 3.4.3: } \longrightarrow \texttt{ over } \Box\ \} \\
 & (p \vee q \longrightarrow S) \,\Box\, (p \vee q \longrightarrow T) \\
\sqsubseteq & \{\ \texttt{Law 3.4.7 (b) twice; mono-}\Box\texttt{ in Law 3.4.1}\ \} \\
 & (p \longrightarrow S) \,\Box\, (q \longrightarrow T)
\end{array}
$$

□

Proof of Law 3.4.10 We first prove that the refinement relation holds in general:

$$
\begin{array}{ll}
 & wp\,(g \longrightarrow S \parallel h \longrightarrow T,\ I) \\
\Leftrightarrow & \{\ \texttt{def. of}\ wp\ \texttt{on}\ \parallel\ \} \\
 & wp\,(g \longrightarrow S,\ wp\,(h \longrightarrow T,\ I)) \wedge wp\,(h \longrightarrow T,\ wp\,(g \longrightarrow S,\ I)) \\
\Leftrightarrow & \{\ \texttt{def. of}\ wp\ \texttt{on}\ \longrightarrow\ \texttt{four times, rearranging terms}\ \} \\
 & (g \wedge h) \wedge wp\,(S,\ h \wedge wp\,(T,\ I)) \wedge wp\,(T,\ g \wedge wp\,(S,\ I)) \\
\Rightarrow & \{\ \texttt{Property 3.3.1, predicate logic}\ \} \\
 & (g \wedge h) \wedge wp\,(S,\ wp\,(T,\ I)) \wedge wp\,(T,\ wp\,(S,\ I)) \\
\Leftrightarrow & \{\ \texttt{def. of}\ wp\ \texttt{on}\ \parallel\ \} \\
 & wp(g \wedge h \longrightarrow S \parallel T, I)
\end{array}
$$

(Continued from the proof of Law 3.4.10)
In the specific context where $g = \mathcal{G}[\![p]\!], S = \mathcal{P}[\![p]\!], h = \mathcal{G}[\![q]\!], T = \mathcal{P}[\![q]\!]$, and $I = p = q$ for some predicates p and q, we argue by Property 3.3.2 that the $\Rightarrow$ step above can be strengthened to $=$, i.e. $\mathcal{G}[\![p]\!] \Leftrightarrow wp\,(\mathcal{P}[\![p]\!],\ p)$ means that $g \Leftrightarrow wp\,(S,\ I)$ and $\mathcal{G}[\![q]\!] \Leftrightarrow wp\,(\mathcal{P}[\![q]\!],\ q)$ means that $h \Leftrightarrow wp\,(T,\ I)$. □

Proof of Law 3.4.11 For case `(b)`, we argue as follows, for any predicate I:

$$
\begin{array}{ll}
 & wp\,(S \parallel T,\ I) \\
\Leftrightarrow & \{\ \texttt{def. of } wp \texttt{ on } \parallel\ \} \\
 & wp\,(S,\ wp\,(T,\ I)) \wedge wp\,(T,\ wp\,(S,\ I)) \\
\overset{*}{\Rightarrow} & \{\ \texttt{predicate logic: } p \wedge q \Rightarrow p\ \} \\
 & wp\,(S,\ wp\,(T,\ I)) \\
\Rightarrow & \{\ \texttt{predicate logic: } p \Rightarrow p \vee q\ \} \\
 & wp\,(S,\ wp\,(T,\ I)) \vee wp\,(T,\ wp\,(S,\ I)) \\
\Leftrightarrow & \{\ \texttt{def. of } wp \texttt{ on } \Box\texttt{, then on ; twice}\ \} \\
 & wp\,((S\ ;\ T) \Box (T\ ;\ S),\ I)
\end{array}
$$

Immediately after the intermediate derivation step that is marked with $*$, we essentially complete the proof of case `(b)`, with the definition of wp on sequential composition $(;)$. From then on until the last derivation step we complete the proof of case `(a)`. By the transitivity of $\sqsubseteq$, we conclude that $(S\ ;\ T) \Box (T\ ;\ S) \sqsubseteq S \parallel T$. □

B.2 Proofs of BOOSTER Predicate Laws

Proof of Law 3.5.2 We argue the case of `(sym-∧)` as follows:

$$
\begin{array}{ll}
 & \mathcal{P}[\![p \wedge q]\!] \\
= & \{\ \texttt{axiom of } \mathcal{P} \texttt{ on } \wedge\ \} \\
 & (\mathcal{G}[\![p]\!] \wedge \mathcal{G}[\![q]\!]) \longrightarrow (\mathcal{P}[\![p]\!] \parallel \mathcal{P}[\![q]\!]) \\
= & \{\ \texttt{predicate logic: symmetry of } \wedge\texttt{, symmetry of } \parallel\ \} \\
 & (\mathcal{G}[\![q]\!] \wedge \mathcal{G}[\![p]\!]) \longrightarrow (\mathcal{P}[\![q]\!] \parallel \mathcal{P}[\![p]\!]) \\
= & \{\ \texttt{axiom of } \mathcal{P} \texttt{ on } \wedge\ \} \\
 & \mathcal{P}[\![q \wedge p]\!]
\end{array}
$$

We prove the case of `(sym-∨)` in almost an identical manner, using the symmetry properties of logical disjunction $(\vee)$ and deterministic choice $(\Box)$ instead. □

Proof of Law 3.5.4 In the case of `(zero-∨)`, as the LHS is a choice between programs $\mathcal{P}[\![p]\!]$ and $true \longrightarrow skip$, it is not equivalent to the RHS where the choice of $\mathcal{P}[\![p]\!]$ is not available; however, if $p = before\,(p)$, then the LHS becomes $(p \longrightarrow skip) \Box (true \longrightarrow skip)$, which equals $(p \vee true) \longrightarrow skip$ by Law 3.4.2 `(a)` and further equals $true \longrightarrow skip$ by Zero of $\vee$ in predicate logic. Consequently, LHS equals RHS when $p = before\,(p)$. We will see later in Law 3.5.12 that a refinement relation exists for the case of `(zero-∨)` instead.

(Continued from the proof of Law 3.5.4) We argue the case of `(zero-∧)` as follows:

$$
\begin{array}{ll}
 & wp\,(\mathcal{P}\,[\![\,p \wedge \mathbf{false}\,]\!],\ I) \\
\Leftrightarrow & \{\ \texttt{axioms of } \mathcal{P} \texttt{ on } \wedge\texttt{, } \mathcal{G} \texttt{ and } \mathcal{P} \texttt{ on } \mathbf{false}\ \} \\
 & wp\,(\mathcal{G}\,[\![\,p\,]\!] \wedge \mathit{false} \longrightarrow \mathcal{P}\,[\![\,p\,]\!] \parallel (\mathit{false} \longrightarrow \mathit{skip}),\ I) \\
\Leftrightarrow & \{\ \texttt{def. of } wp \texttt{ on } \longrightarrow\texttt{, predicate logic: zero of } \wedge\ \} \\
 & \mathit{false} \\
\Leftrightarrow & \{\ \texttt{def. of } wp \texttt{ on } \mathit{false} \longrightarrow \mathit{skip}\ \} \\
 & wp\,(\mathit{false} \longrightarrow \mathit{skip},\ I) \\
\Leftrightarrow & \{\ \texttt{axiom of } \mathcal{P} \texttt{ on } \mathbf{false}\ \} \\
 & wp\,(\mathcal{P}\,[\![\,\mathbf{false}\,]\!],\ I)
\end{array}
$$

Similar arguments apply to the cases of `(iden-∧)` and `(iden-∨)` via, respectively, theorems on the identity of $\wedge$ and $\vee$ in predicate logic. □

To prove Law 3.5.5, we need a lemma: we may simplify a parallel composition provided that the programs and guards involved are generated from $\mathcal{P}$ and $\mathcal{G}$ (Section 3.2).

Lemma B.2.1 (***Simplifying Parallel Composition***)

$$(g \wedge h) \longrightarrow S \parallel T = (g \wedge h) \longrightarrow T$$

provided $g = \mathcal{P}\,[\![\,p\,]\!], S = \mathcal{P}\,[\![\,p\,]\!]$ and $h = \mathcal{P}\,[\![\,q\,]\!], T = \mathcal{P}\,[\![\,q\,]\!]$ for some predicates p, q, where $\mathit{before}\,(p) = p$ □

Proof of Lemma B.2.1 Since $\mathit{before}\,(p) = p$, we have $\mathcal{G}\,[\![\,p\,]\!] = p$ and $\mathcal{P}\,[\![\,p\,]\!] = p \longrightarrow \mathit{skip}$ by Property 3.2.1.

$$
\begin{array}{ll}
 & wp\,(\mathcal{G}\,[\![\,p\,]\!] \wedge \mathcal{G}\,[\![\,q\,]\!] \longrightarrow \mathcal{P}\,[\![\,p\,]\!] \parallel \mathcal{P}\,[\![\,q\,]\!],\ I) \\
\Leftrightarrow & \{\ \mathcal{G}\,[\![\,p\,]\!] = p \ \texttt{ and } \mathcal{P}\,[\![\,p\,]\!] = p \longrightarrow \mathit{skip}\ \} \\
 & wp\,(p \wedge \mathcal{G}\,[\![\,q\,]\!] \longrightarrow ((p \longrightarrow \mathit{skip}) \parallel \mathcal{P}\,[\![\,q\,]\!]),\ I) \\
\Leftrightarrow & \{\ \texttt{distributivity of } \longrightarrow \texttt{ over } \parallel\ \} \\
 & wp\,(\left(\begin{array}{l} p \wedge \mathcal{G}\,[\![\,q\,]\!] \longrightarrow (p \longrightarrow \mathit{skip}) \\ \parallel \\ p \wedge \mathcal{G}\,[\![\,q\,]\!] \longrightarrow \mathcal{P}\,[\![\,q\,]\!] \end{array}\right),\ I) \\
\Leftrightarrow & \{\ \texttt{Law 3.4.6 (a)}\ \} \\
 & wp\,(\left(\begin{array}{l} p \wedge \mathcal{G}\,[\![\,q\,]\!] \longrightarrow \mathit{skip} \\ \parallel \\ p \wedge \mathcal{G}\,[\![\,q\,]\!] \longrightarrow \mathcal{P}\,[\![\,q\,]\!] \end{array}\right),\ I) \\
\Leftrightarrow & \{\ \texttt{distributivity of } \longrightarrow \texttt{ over } \parallel\ \} \\
 & wp\,(p \wedge \mathcal{G}\,[\![\,q\,]\!] \longrightarrow (\mathit{skip} \parallel \mathcal{P}\,[\![\,q\,]\!]),\ I) \\
\Leftrightarrow & \{\ \texttt{Law 3.4.5}\ \} \\
 & wp\,(p \wedge \mathcal{G}\,[\![\,q\,]\!] \longrightarrow \mathcal{P}\,[\![\,q\,]\!],\ I) \\
\Leftrightarrow & \{\ \mathcal{G}\,[\![\,p\,]\!] = p\ \} \\
 & wp\,(\mathcal{G}\,[\![\,p\,]\!] \wedge \mathcal{G}\,[\![\,q\,]\!] \longrightarrow \mathcal{P}\,[\![\,q\,]\!],\ I)
\end{array}
$$

□

Proof of Law 3.5.5

$$
\begin{array}{ll}
 & \mathcal{P}[\![p \Rightarrow q]\!] \\
= & \{ \texttt{axiom of } \mathcal{P} \texttt{ on } \Rightarrow \} \\
 & (\neg\, \mathcal{G}[\![p]\!] \longrightarrow \mathit{skip}) \,\Box\, (\mathcal{G}[\![p]\!] \wedge \mathcal{G}[\![q]\!] \longrightarrow \mathcal{P}[\![q]\!]) \\
= & \{ \mathit{before}\,(p) = p \} \\
 & (\mathcal{G}[\![\neg\, p]\!] \longrightarrow \mathit{skip}) \,\Box\, (\mathcal{G}[\![p]\!] \wedge \mathcal{G}[\![q]\!] \longrightarrow \mathcal{P}[\![q]\!]) \\
= & \{ \mathit{before}\,(p) = p, \texttt{ Lemma B.2.1} \} \\
 & (\mathcal{G}[\![\neg\, p]\!] \longrightarrow \mathit{skip}) \,\Box\, (\mathcal{G}[\![p]\!] \wedge \mathcal{G}[\![q]\!]) \longrightarrow \mathcal{P}[\![p]\!] \parallel \mathcal{P}[\![q]\!]) \\
= & \{ \texttt{Law 3.4.2 (b), predicate logic: idempotency of } \wedge \} \\
 & \mathcal{G}[\![\neg\, p]\!] \longrightarrow \mathit{skip} \\
 & \Box \\
 & (\mathcal{G}[\![p]\!] \wedge \mathcal{G}[\![q]\!]) \longrightarrow (\mathcal{G}[\![p]\!] \wedge \mathcal{G}[\![q]\!]) \longrightarrow \mathcal{P}[\![p]\!] \parallel \mathcal{P}[\![q]\!]) \\
= & \{ \texttt{axiom of } \mathcal{P} \texttt{ on } \neg \texttt{ and } \wedge\texttt{, axiom of } \mathcal{G} \texttt{ on } \mathcal{P}[\![p \wedge q]\!] \} \\
 & (\mathcal{G}[\![\neg\, p]\!] \longrightarrow \mathcal{P}[\![\neg\, p]\!]) \,\Box\, (\mathcal{G}[\![p \wedge q]\!] \longrightarrow \mathcal{P}[\![p \wedge q]\!]) \\
= & \{ \texttt{axiom of } \mathcal{P} \texttt{ on } \vee \} \\
 & \mathcal{P}[\![\neg\, p \vee (p \wedge q)]\!]
\end{array}
$$

□

Proof of Law 3.5.6

$$
\begin{array}{ll}
 & \mathcal{P}[\![\mathbf{true} \Rightarrow p]\!] \\
= & \{ \texttt{axiom of } \mathcal{P} \texttt{ on } \Rightarrow \} \\
 & \mathbf{true} \wedge \mathcal{G}[\![p]\!] \longrightarrow \mathcal{P}[\![p]\!] \\
 & \Box \\
 & \mathbf{false} \longrightarrow \mathit{Skip} \\
= & \{ \texttt{Law 3.4.4, predicate logic: idem. of } \mathbf{true}\texttt{, then } \mathbf{true} \longrightarrow S = S \} \\
 & \mathcal{P}[\![p]\!]
\end{array}
$$

□

Proof of Law 3.5.7

$$
\begin{array}{ll}
 & \mathcal{P}[\![p \wedge (p \Rightarrow q)]\!] \\
= & \{ \texttt{axiom of } \mathcal{P} \texttt{ on } \wedge \} \\
 & \mathcal{G}[\![p]\!] \wedge \mathcal{G}[\![p \Rightarrow q]\!] \longrightarrow \mathcal{P}[\![p]\!] \parallel \mathcal{P}[\![p \Rightarrow q]\!] \\
= & \{ \texttt{axioms of } \mathcal{G} \texttt{ and } \mathcal{P} \texttt{ on } \Rightarrow \} \\
 & \left(\begin{array}{l} \mathcal{G}[\![p]\!] \\ \wedge \\ (\mathcal{G}[\![p]\!] \wedge \mathcal{G}[\![q]\!]) \vee \neg\, \mathcal{G}[\![p]\!] \end{array} \right) \longrightarrow \left(\begin{array}{l} \mathcal{P}[\![p]\!] \\ \parallel \\ \left(\begin{array}{l} \mathcal{G}[\![p]\!] \wedge \mathcal{G}[\![q]\!] \longrightarrow \mathcal{P}[\![q]\!] \\ \Box \\ \neg\, \mathcal{G}[\![p]\!] \longrightarrow \mathit{skip} \end{array} \right) \end{array} \right) \\
= & \{ \texttt{predicate logic: absorption } p \wedge (\neg\, p \vee q) \equiv p \wedge q, \texttt{ idempotency of } \wedge \} \\
 & \mathcal{G}[\![p]\!] \wedge \mathcal{G}[\![q]\!] \longrightarrow \left(\begin{array}{l} \mathcal{P}[\![p]\!] \\ \parallel \\ \left(\begin{array}{l} \mathcal{G}[\![p]\!] \wedge \mathcal{G}[\![q]\!] \longrightarrow \mathcal{P}[\![q]\!] \\ \Box \\ \neg\, \mathcal{G}[\![p]\!] \longrightarrow \mathit{skip} \end{array} \right) \end{array} \right) \\
= & \{ p = \mathit{before}(p) \texttt{ means } \mathcal{P}[\![p]\!] = \mathit{skip}, \texttt{ then Law 3.4.5} \} \\
 & \mathcal{G}[\![p]\!] \wedge \mathcal{G}[\![q]\!] \longrightarrow \left(\begin{array}{l} \mathcal{G}[\![p]\!] \wedge \mathcal{G}[\![q]\!] \longrightarrow \mathcal{P}[\![q]\!] \\ \Box \\ \neg\, \mathcal{G}[\![p]\!] \longrightarrow \mathit{skip} \end{array} \right) \\
= & \{ \texttt{distributivity of } \longrightarrow \texttt{ over } \Box \} \\
 & \mathcal{G}[\![p]\!] \wedge \mathcal{G}[\![q]\!] \longrightarrow (\mathcal{G}[\![p]\!] \wedge \mathcal{G}[\![q]\!] \longrightarrow \mathcal{P}[\![q]\!]) \\
 & \Box \\
 & \mathcal{G}[\![p]\!] \wedge \mathcal{G}[\![q]\!] \longrightarrow (\neg\, \mathcal{G}[\![p]\!] \longrightarrow \mathit{skip}) \\
= & \{ \texttt{Law 3.4.2 (b) twice, idempotency of } \wedge, \texttt{ excluded middle, zero of } \wedge \} \\
 & \mathcal{G}[\![p]\!] \wedge \mathcal{G}[\![q]\!] \longrightarrow \mathcal{P}[\![q]\!] \\
 & \Box \\
 & \mathit{false} \longrightarrow \mathit{skip} \\
= & \{ \texttt{Law 3.4.4} \} \\
 & \mathcal{G}[\![p]\!] \wedge \mathcal{G}[\![q]\!] \longrightarrow \mathcal{P}[\![q]\!] \\
= & \{ p = \mathit{before}(p) \texttt{ means } \mathcal{P}[\![p]\!] = \mathit{skip}, \texttt{ then Law 3.4.5, then commutativity of } \parallel \} \\
 & \mathcal{G}[\![p]\!] \wedge \mathcal{G}[\![q]\!] \longrightarrow \mathcal{P}[\![p]\!] \parallel \mathcal{P}[\![q]\!] \\
= & \{ \texttt{def. of } \mathcal{P} \texttt{ on } \wedge \} \\
 & \mathcal{P}[\![p \wedge q]\!]
\end{array}
$$

□

There is a step of derivation which the proof of Law 3.5.8 requires:

Lemma B.2.2 (***Simplifying Inner Guard***)

$$
\left(\begin{array}{l} p \wedge q \\ \vee \\ p \wedge r \end{array} \right) \longrightarrow \left(\begin{array}{l} p \wedge q \longrightarrow S \\ \Box \\ p \wedge r \longrightarrow T \end{array} \right) = \left(\begin{array}{l} p \wedge q \\ \vee \\ p \wedge r \end{array} \right) \longrightarrow \left(\begin{array}{l} q \longrightarrow S \\ \Box \\ r \longrightarrow T \end{array} \right)
$$

□

Proof of Lemma B.2.2

$$
\begin{array}{ll}
 & wp\,((p \wedge q) \vee (p \wedge r) \longrightarrow (p \wedge q \longrightarrow S \,\square\, p \wedge r \longrightarrow T),\ I) \\
\Leftrightarrow & \{\ wp\ \texttt{on}\ \longrightarrow\ \texttt{three times}\ \} \\
 & ((p \wedge q) \vee (p \wedge r)) \wedge ((p \wedge q) \wedge wp\,(S,\ I) \wedge (p \wedge r) \wedge wp\,(T,\ I)) \\
\Leftrightarrow & \{\ \texttt{distributivity of}\ \wedge\ \texttt{over}\ \vee\ \} \\
 & (\boxed{\mathrm{p}} \wedge (q \vee r)) \wedge ((\boxed{\mathrm{p}} \wedge q) \wedge wp\,(S,\ I) \wedge (\boxed{\mathrm{p}} \wedge r) \wedge wp\,(T,\ I)) \\
\Leftrightarrow & \{\ \texttt{idempotency of}\ \wedge\ \texttt{twice}\ \} \\
 & (p \wedge (q \vee r)) \wedge (q \wedge wp\,(S,\ I) \wedge r \wedge wp\,(T,\ I)) \\
\Leftrightarrow & \{\ \texttt{distributivity of}\ \wedge\ \texttt{over}\ \vee\texttt{, then}\ wp\ \texttt{on}\ \longrightarrow\ \texttt{three times}\ \} \\
 & wp\,((p \wedge q) \vee (p \wedge r) \longrightarrow (q \longrightarrow S \,\square\, r \longrightarrow T),\ I)
\end{array}
$$

□

There is also a step of derivation which both the proof of Law 3.5.8 and that of Law 3.5.23 require:

Lemma B.2.3 (***Eliminating Outer Guard***)

$$p \vee q \longrightarrow ((p \longrightarrow S) \,\square\, (q \longrightarrow T)) = (p \longrightarrow S) \,\square\, (q \longrightarrow T)$$

On the LHS, in the context of a guarded substitution, we may remove the guard $(p \vee q)$ if it disjoins the guards of both of its guarded branches of choice. □

Proof of Lemma B.2.3

$$
\begin{array}{ll}
 & wp\,(p \vee q \longrightarrow ((p \longrightarrow S) \,\square\, (q \longrightarrow T)),\ I) \\
\Leftrightarrow & \{\ wp\ \texttt{on}\ \longrightarrow\texttt{, then on}\ \square\texttt{, then on}\ \longrightarrow\ \texttt{twice}\ \} \\
 & (p \vee q) \wedge ((p \wedge wp\,(S,\ I)) \vee (q \wedge wp\,(T,\ I))) \\
\Leftrightarrow & \{\ \texttt{predicate logic: distributivity of}\ \wedge\ \texttt{over}\ \vee\ \} \\
 & ((p \vee q) \wedge (p \wedge wp\,(S,\ I))) \vee ((p \vee q) \wedge (q \wedge wp\,(T,\ I))) \\
\Leftrightarrow & \{\ \texttt{absorption in predicate logic:}\ p \wedge (p \vee q) = p\texttt{, twice}\ \} \\
 & (p \wedge wp\,(S,\ I)) \vee (q \wedge wp\,(T,\ I)) \\
\Leftrightarrow & \{\ wp\ \texttt{on}\ \longrightarrow\texttt{, then on}\ \square\ \} \\
 & wp\,((p \longrightarrow S) \,\square\, (q \longrightarrow T),\ I)
\end{array}
$$

□

Proof of Law 3.5.8

$$
\begin{array}{ll}
 & \mathcal{P}[\![(p \wedge q) \Rightarrow r]\!] \\
= & \{\, \texttt{axiom of } \mathcal{P} \texttt{ on } \Rightarrow \,\} \\
 & (\mathcal{G}[\![p \wedge q]\!] \wedge \mathcal{G}[\![r]\!] \longrightarrow \mathcal{P}[\![r]\!]) \,\Box\, (\neg\, \mathcal{G}[\![p \wedge q]\!] \longrightarrow \mathit{skip}) \\
= & \{\, \texttt{axiom of } \mathcal{G} \texttt{ on } \wedge \texttt{ twice} \,\} \\
 & ((\mathcal{G}[\![p]\!] \wedge \mathcal{G}[\![q]\!]) \wedge \mathcal{G}[\![r]\!] \longrightarrow \mathcal{P}[\![r]\!]) \,\Box\, (\neg\, ((\mathcal{G}[\![p]\!] \wedge \mathcal{G}[\![q]\!])) \longrightarrow \mathit{skip}) \\
= & \{\, \texttt{predicate logic: associativity of } \wedge\texttt{, De Morgan} \,\} \\
 & (\mathcal{G}[\![p]\!] \wedge (\mathcal{G}[\![q]\!] \wedge \mathcal{G}[\![r]\!]) \longrightarrow \mathcal{P}[\![r]\!]) \,\Box\, (\neg\, \mathcal{G}[\![p]\!] \vee \neg\, \mathcal{G}[\![q]\!] \longrightarrow \mathit{skip}) \\
= & \{\, \texttt{predicate logic: } p \wedge (\neg\, p \vee q) \equiv p \wedge q \texttt{ (absorption)} \,\} \\
 & (\mathcal{G}[\![p]\!] \wedge (\mathcal{G}[\![q]\!] \wedge \mathcal{G}[\![r]\!]) \longrightarrow \mathcal{P}[\![r]\!]) \,\Box\, (\boxed{(\mathcal{G}[\![p]\!] \wedge \neg\, \mathcal{G}[\![q]\!]) \vee \neg\, \mathcal{G}[\![p]\!]} \longrightarrow \mathit{skip}) \\
= & \{\, \texttt{Law 3.4.2 (a), associativity of } \Box \,\} \\
 & \left(\begin{array}{l} \mathcal{G}[\![p]\!] \wedge (\mathcal{G}[\![q]\!] \wedge \mathcal{G}[\![r]\!]) \longrightarrow \mathcal{P}[\![r]\!] \\ \Box \\ \boxed{\mathcal{G}[\![p]\!] \wedge \neg\, \mathcal{G}[\![q]\!]} \longrightarrow \mathit{skip} \end{array} \right) \Box\, \boxed{\neg\, \mathcal{G}[\![p]\!]} \longrightarrow \mathit{skip} \\
= & \{\, \texttt{Law 3.4.2 (b) twice, predicate logic: idempotency of } \wedge \,\} \\
 & \left(\begin{array}{l} \mathcal{G}[\![p]\!] \wedge (\mathcal{G}[\![q]\!] \wedge \mathcal{G}[\![r]\!]) \longrightarrow (\mathcal{G}[\![q]\!] \wedge \mathcal{G}[\![r]\!] \longrightarrow \mathcal{P}[\![r]\!]) \\ \Box \\ \mathcal{G}[\![p]\!] \wedge \neg\, \mathcal{G}[\![q]\!] \longrightarrow (\neg\, \mathcal{G}[\![q]\!] \longrightarrow \mathit{skip}) \end{array} \right) \Box\, \neg\, \mathcal{G}[\![p]\!] \longrightarrow \mathit{skip} \\
= & \{\, \texttt{Law 3.4.6} \,\} \\
 & \left(\begin{array}{l} \mathcal{G}[\![p]\!] \wedge (\mathcal{G}[\![q]\!] \wedge \mathcal{G}[\![r]\!]) \longrightarrow \mathcal{P}[\![r]\!] \\ \Box \\ \mathcal{G}[\![p]\!] \wedge \neg\, \mathcal{G}[\![q]\!] \longrightarrow \mathit{skip} \end{array} \right) \Box\, \neg\, \mathcal{G}[\![p]\!] \longrightarrow \mathit{skip} \\
= & \{\, \texttt{Lemma B.2.3, then Lemma B.2.2} \,\} \\
 & \left(\begin{array}{lcl} \mathcal{G}[\![p]\!] \wedge (\mathcal{G}[\![q]\!] \wedge \mathcal{G}[\![r]\!]) & & \mathcal{G}[\![q]\!] \wedge \mathcal{G}[\![r]\!] \longrightarrow \mathcal{P}[\![r]\!] \\ \vee & \longrightarrow & \Box \\ \mathcal{G}[\![p]\!] \wedge \neg\, \mathcal{G}[\![q]\!] & & \neg\, \mathcal{G}[\![q]\!] \longrightarrow \mathit{skip} \end{array} \right) \Box\, \neg\, \mathcal{G}[\![p]\!] \longrightarrow \mathit{skip} \\
= & \{\, \texttt{predicate logic: distributivity of } \wedge \texttt{ over } \vee \,\} \\
 & \mathcal{G}[\![p]\!] \wedge (\neg\, \mathcal{G}[\![q]\!] \vee (\mathcal{G}[\![q]\!] \wedge \mathcal{G}[\![r]\!]) \longrightarrow \left(\begin{array}{l} \mathcal{G}[\![q]\!] \wedge \mathcal{G}[\![r]\!] \longrightarrow \mathcal{P}[\![r]\!] \\ \Box \\ \neg\, \mathcal{G}[\![q]\!] \longrightarrow \mathit{skip} \end{array} \right) \\
 & \Box \\
 & \neg\, \mathcal{G}[\![p]\!] \longrightarrow \mathit{skip} \\
= & \{\, \texttt{axiom of } \mathcal{G} \texttt{ on } \Rightarrow \,\} \\
 & \mathcal{G}[\![p]\!] \wedge \mathcal{G}[\![q \Rightarrow r]\!] \longrightarrow ((\mathcal{G}[\![q]\!] \wedge \mathcal{G}[\![r]\!] \longrightarrow \mathcal{P}[\![r]\!]) \,\Box\, (\neg\, \mathcal{G}[\![q]\!] \longrightarrow \mathit{skip})) \\
 & \Box \\
 & \neg\, \mathcal{G}[\![p]\!] \longrightarrow \mathit{skip} \\
= & \{\, \texttt{axiom of } \mathcal{P} \texttt{ on } \Rightarrow \texttt{ twice} \,\} \\
 & \mathcal{P}[\![p \Rightarrow (q \Rightarrow r)]\!]
\end{array}
$$

□

Proof of Law 3.5.9

$$
\begin{array}{ll}
 & (p \Rightarrow q) \Rightarrow (p \Rightarrow r) \\
\equiv & \{ \texttt{Law 3.5.8} \} \\
 & ((p \Rightarrow q) \wedge p) \Rightarrow r \\
\equiv & \{ \texttt{Law 3.5.2, Law 3.5.7} \} \\
 & p \wedge q \Rightarrow r \\
\equiv & \{ \texttt{Law 3.5.8} \} \\
 & p \Rightarrow (q \Rightarrow r)
\end{array}
$$

□

Proof of Law 3.5.10

The LHS and RHS in cases `(a)` - `(c)` represent different patterns of guarded programs: in cases `(a)` and `(b)`, two guarded programs are run in parallel on the LHS, whereas there is just one running on the RHS. In cases `(d)` - `(f)`, as the the predicate p appears as the antecedent, we would expect that it does not contain intent to modify the state, i.e. $p = \mathit{before}\,(p)$.

For case `(a)`, we argue as follows:

$$
\begin{array}{ll}
 & \mathcal{P}\,[\![\,(p \Rightarrow r) \wedge (q \Rightarrow r)\,]\!] \\
= & \{ \texttt{Property 3.2.1: before-}\wedge \} \\
 & ((p \Rightarrow r) \wedge (q \Rightarrow r)) \longrightarrow \mathit{skip} \\
= & \{ \texttt{predicate logic: } (p \Rightarrow r) \wedge (q \Rightarrow r) \equiv (p \vee q) \Rightarrow r \} \\
 & ((p \vee q) \Rightarrow r) \longrightarrow \mathit{skip} \\
= & \{ \texttt{Property 3.2.1: before-}\wedge \} \\
 & \mathcal{P}\,[\![\,(p \vee q) \Rightarrow r\,]\!]
\end{array}
$$

Similar arguments apply to cases `(b)` - `(f)`. For case `(g)`, we argue as follows:

$$
\begin{array}{ll}
 & \mathcal{P}\,[\![\,\mathbf{false} \Rightarrow p\,]\!] \\
= & \{ \texttt{axiom of } \mathcal{P} \texttt{ on } \Rightarrow\texttt{, predicate logic: negation of } \mathit{false} \} \\
 & (\mathit{false} \wedge \mathcal{G}\,[\![\,p\,]\!] \longrightarrow \mathcal{P}\,[\![\,p\,]\!]) \,\square\, (\mathit{true} \longrightarrow \mathit{skip}) \\
= & \{ \texttt{Law 3.4.2 (b), pred. logic: zero of } \wedge \} \\
 & (\mathit{false} \longrightarrow \mathcal{P}\,[\![\,p\,]\!]) \,\square\, (\mathit{true} \longrightarrow \mathit{skip}) \\
= & \{ \texttt{Law 3.4.4} \} \\
 & \mathit{true} \longrightarrow \mathit{skip} \\
= & \{ \texttt{axiom of } \mathcal{P} \texttt{ on } \mathbf{true} \} \\
 & \mathcal{P}\,[\![\,\mathbf{true}\,]\!]
\end{array}
$$

□

Proof of Law 3.5.11 We prove by arguing that for any predicate I:

$$
\begin{array}{ll}
 & wp\,(\mathcal{P}\,[\![\,p\,]\!],\ I) \\
\Leftrightarrow & \{\ \texttt{Property 3.2.3}\ \} \\
 & wp\,(\mathcal{G}\,[\![\,p\,]\!] \longrightarrow \mathcal{P}\,[\![\,p\,]\!],\ I) \\
\Leftrightarrow & \{\ \texttt{def. of } wp \texttt{ on } \longrightarrow\ \} \\
 & \mathcal{G}\,[\![\,p\,]\!] \wedge wp\,(\mathcal{P}\,[\![\,p\,]\!],\ I) \\
\Rightarrow & \{\ \texttt{predicate logic: } p \Rightarrow (p \vee q)\ \} \\
 & (\mathcal{G}\,[\![\,p\,]\!] \wedge wp\,(\mathcal{P}\,[\![\,p\,]\!],\ I)) \vee (\mathcal{G}\,[\![\,q\,]\!] \wedge wp\,(\mathcal{P}\,[\![\,q\,]\!],\ I)) \\
\Leftrightarrow & \{\ \texttt{def. of } wp \texttt{ on } \Box\texttt{, then on } \longrightarrow \texttt{ twice}\ \} \\
 & wp\,((\mathcal{G}\,[\![\,p\,]\!] \longrightarrow \mathcal{P}\,[\![\,p\,]\!]) \,\Box\, (\mathcal{G}\,[\![\,q\,]\!] \longrightarrow \mathcal{P}\,[\![\,q\,]\!]),\ I) \\
\Leftrightarrow & \{\ \texttt{def. of } \mathcal{P} \texttt{ on } \vee\ \} \\
 & wp\,(\mathcal{P}\,[\![\,p \vee q\,]\!],\ I)
\end{array}
$$

□

Proof of Law 3.5.14 Since $q = \mathit{before}\,(q)$, we have $\mathcal{P}\,[\![\,q\,]\!] = q \longrightarrow \mathit{skip}$. We prove by arguing that, for any predicate I, $wp\,(\mathcal{P}\,[\![\,p \wedge q\,]\!],\ I) \Rightarrow wp\,(\mathcal{P}\,[\![\,p\,]\!],\ I)$.

$$
\begin{array}{ll}
 & wp\,(\mathcal{P}\,[\![\,p \wedge q\,]\!],\ I) \\
\Leftrightarrow & \{\ \texttt{def. of } \mathcal{G} \texttt{ and } \mathcal{P} \texttt{ on } \wedge\ \} \\
 & wp\,((\mathcal{G}\,[\![\,p\,]\!] \wedge \mathcal{G}\,[\![\,q\,]\!]) \longrightarrow (\mathcal{P}\,[\![\,p\,]\!] \parallel \mathcal{P}\,[\![\,q\,]\!]),\ I) \\
\Leftrightarrow & \{\ \texttt{def. of } wp \texttt{ on } \longrightarrow\texttt{, then on } \parallel\ \} \\
 & (\mathcal{G}\,[\![\,p\,]\!] \wedge \mathcal{G}\,[\![\,q\,]\!]) \wedge wp\,(\mathcal{P}\,[\![\,p\,]\!],\ \boxed{wp\,(\mathcal{P}\,[\![\,q\,]\!],\ I)}) \wedge wp\,(\mathcal{P}\,[\![\,q\,]\!],\ wp\,(\mathcal{P}\,[\![\,p\,]\!],\ I)) \\
\overset{*}{\Leftrightarrow} & \{\ \mathcal{P}\,[\![\,q\,]\!] = q \longrightarrow \mathit{skip}\ \} \\
 & (\mathcal{G}\,[\![\,p\,]\!] \wedge \mathcal{G}\,[\![\,q\,]\!]) \wedge wp\,(\mathcal{P}\,[\![\,p\,]\!],\ wp\,(q \longrightarrow \mathit{skip},\ I)) \wedge wp\,(\mathcal{P}\,[\![\,q\,]\!],\ wp\,(\mathcal{P}\,[\![\,p\,]\!],\ I)) \\
\Leftrightarrow & \{\ \texttt{def. of } wp \texttt{ on } \longrightarrow\texttt{, then on } \mathit{skip}\ \} \\
 & (\mathcal{G}\,[\![\,p\,]\!] \wedge \mathcal{G}\,[\![\,q\,]\!]) \wedge wp\,(\mathcal{P}\,[\![\,p\,]\!],\ q \wedge I) \wedge wp\,(\mathcal{P}\,[\![\,q\,]\!],\ wp\,(\mathcal{P}\,[\![\,p\,]\!],\ I)) \\
\Rightarrow & \{\ \texttt{predicate logic: } p \wedge q \Rightarrow p\texttt{, Property 3.3.1}\ \} \\
 & wp\,(\mathcal{P}\,[\![\,p\,]\!],\ \boxed{I})
\end{array}
$$

In general, the refinement relationship does not hold without the side condition $q = \mathit{before}\,(q)$, because in the derivation that is marked with $*$, there is in general no connection between $wp\,(\mathcal{P}\,[\![\,q\,]\!],\ I)$ and I—e.g. say I is $x > 0$ and $\mathcal{P}\,[\![\,q\,]\!]$ is $x := 1$, then the calculated result by wp is simply *true* and hence weaker than I. □

Proof of Law 3.5.15

$$
\begin{array}{ll}
 & p \wedge (q \wedge r) \\
\equiv & \{\ \texttt{Law 3.5.3 assoc-}\wedge\ \} \\
 & (p \wedge q) \wedge r \\
\sqsubseteq & \{\ p = \mathit{before}\,(p)\texttt{, Law 3.5.14, case (mono-}\wedge\texttt{) of Law 3.5.1}\ \} \\
 & (p \wedge q) \wedge (p \wedge r)
\end{array}
$$

□

Proof of Law 3.5.16 We obtain that $LHS \sqsubseteq RHS$ immediately from Law 3.5.14, with the side condition that $q \Rightarrow p = \mathit{before}\,(q \Rightarrow p)$. This would mean that $LHS = \mathit{before}\,(LHS)$ and $RHS = \mathit{before}\,(RHS)$. By Property 3.2.1 we obtain that $\mathcal{P}\,[\![\, LHS \,]\!] = \mathit{before}\,(LHS) \longrightarrow \mathit{skip}$ and $\mathcal{P}\,[\![\, RHS \,]\!] = \mathit{before}\,(RHS) \longrightarrow \mathit{skip}$. As we already know $\mathit{before}\,(LHS) \equiv \mathit{before}\,(RHS)$ in standard predicate logic, we conclude that $\mathcal{P}\,[\![\, LHS \,]\!] = \mathcal{P}\,[\![\, RHS \,]\!]$. □

Proof of Law 3.5.18

$$
\begin{array}{ll}
 & p \vee q \\
\sqsubseteq & \{\, \texttt{Law 3.5.11} \,\} \\
 & p \\
\sqsubseteq & \{\, q = \mathit{before}\,(q), \texttt{ Law 3.5.14} \,\} \\
 & p \wedge q
\end{array}
$$

□

Proof of Law 3.5.21

$$
\begin{array}{ll}
 & \neg\, p \vee q \\
\sqsubseteq & \{\, p = \mathit{before}\,(p), \texttt{ Law 3.5.14, Law 3.5.1 of monotonicity} \,\} \\
 & \neg\, p \vee (p \wedge q) \\
\equiv & \{\, \texttt{Law 3.5.5} \,\} \\
 & p \Rightarrow q
\end{array}
$$

□

Proof of Law 3.5.22

$$
\begin{array}{ll}
 & \mathcal{P}[\![(p \vee q) \wedge (p \vee r)]\!] \\
= & \{\, \texttt{axiom of } \mathcal{P} \texttt{ on } \wedge \,\} \\
 & (\mathcal{G}[\![p \vee q]\!] \wedge \mathcal{G}[\![p \vee r]\!]) \longrightarrow (\mathcal{P}[\![p \vee q]\!] \parallel \mathcal{P}[\![p \vee r]\!]) \\
= & \{\, \texttt{axiom of } \mathcal{G} \texttt{ on } \vee\texttt{, twice; axiom of } \mathcal{P} \texttt{ on } \vee\texttt{, twice} \,\} \\
 & \left(\begin{array}{l} \mathcal{G}[\![p]\!] \vee \mathcal{G}[\![q]\!] \\ \wedge \\ \mathcal{G}[\![p]\!] \vee \mathcal{G}[\![r]\!] \end{array}\right) \longrightarrow \left(\begin{array}{l} \mathcal{G}[\![p]\!] \vee \mathcal{G}[\![q]\!] \longrightarrow \mathcal{P}[\![p]\!] \,\square\, \mathcal{P}[\![q]\!] \\ \parallel \\ \mathcal{G}[\![p]\!] \vee \mathcal{G}[\![r]\!] \longrightarrow \mathcal{P}[\![p]\!] \,\square\, \mathcal{P}[\![r]\!] \end{array}\right) \\
\overset{*}{\sqsubseteq} & \{\, \texttt{explained below} \,\} \\
 & \left(\begin{array}{l} \mathcal{G}[\![p]\!] \vee \mathcal{G}[\![q]\!] \\ \wedge \\ \mathcal{G}[\![p]\!] \vee \mathcal{G}[\![r]\!] \end{array}\right) \longrightarrow \left(\begin{array}{lll} \mathcal{G}[\![p]\!] \vee \mathcal{G}[\![q]\!] & & \mathcal{P}[\![p]\!] \,\square\, \mathcal{P}[\![q]\!] \\ \wedge & \longrightarrow & \parallel \\ \mathcal{G}[\![p]\!] \vee \mathcal{G}[\![r]\!] & & \mathcal{P}[\![p]\!] \,\square\, \mathcal{P}[\![r]\!] \end{array}\right) \\
= & \{\, \texttt{Law 3.4.2 (b)} \,\} \\
 & \begin{array}{lll} \mathcal{G}[\![p]\!] \vee \mathcal{G}[\![q]\!] & & \mathcal{P}[\![p]\!] \,\square\, \mathcal{P}[\![q]\!] \\ \wedge & \longrightarrow & \parallel \\ \mathcal{G}[\![p]\!] \vee \mathcal{G}[\![r]\!] & & \mathcal{P}[\![p]\!] \,\square\, \mathcal{P}[\![r]\!] \end{array} \\
= & \{\, \texttt{distributivity of } \vee \texttt{ over } \wedge\texttt{, distributivity of } \square \texttt{ over } \parallel \,\} \\
 & \begin{array}{lll} \mathcal{G}[\![p]\!] & & \mathcal{P}[\![p]\!] \\ \vee & \longrightarrow & \square \\ \mathcal{G}[\![q]\!] \wedge \mathcal{G}[\![r]\!] & & \mathcal{P}[\![q]\!] \parallel \mathcal{P}[\![r]\!] \end{array} \\
\sqsubseteq & \{\, \texttt{Law 3.4.9} \,\} \\
 & (\ \mathcal{G}[\![p]\!] \longrightarrow \mathcal{P}[\![p]\!]\ ) \,\square\, (\ \mathcal{G}[\![q]\!] \wedge \mathcal{G}[\![r]\!] \longrightarrow \mathcal{P}[\![q]\!] \parallel \mathcal{P}[\![r]\!]\ ) \\
= & \{\, \texttt{Law 3.4.2 (b), predicate logic: idempotency of } \wedge \,\} \\
 & (\ \mathcal{G}[\![p]\!] \longrightarrow \mathcal{P}[\![p]\!]\ ) \,\square\, \left(\left(\begin{array}{l} \mathcal{G}[\![q]\!] \\ \wedge \\ \mathcal{G}[\![r]\!] \end{array}\right) \longrightarrow \left(\begin{array}{lll} \mathcal{G}[\![q]\!] & & \mathcal{P}[\![q]\!] \\ \wedge & \longrightarrow & \parallel \\ \mathcal{G}[\![r]\!] & & \mathcal{P}[\![r]\!] \end{array}\right)\right) \\
= & \{\, \texttt{axioms of } \mathcal{G} \texttt{ and } \mathcal{P} \texttt{ on } \wedge \,\} \\
 & (\ \mathcal{G}[\![p]\!] \longrightarrow \mathcal{P}[\![p]\!]\ ) \,\square\, (\ \mathcal{G}[\![q \wedge r]\!] \longrightarrow \mathcal{P}[\![q \wedge r]\!]\ ) \\
= & \{\, \texttt{axiom of } \mathcal{P} \texttt{ on } \vee \,\} \\
 & \mathcal{P}[\![p \vee (q \wedge r)]\!]
\end{array}
$$

In the derivation step above that is marked with $*$, we cannot choose to use Law 3.4.10 as $\mathcal{P}[\![p]\!] \,\square\, \mathcal{P}[\![q]\!]$ and $\mathcal{P}[\![p]\!] \,\square\, \mathcal{P}[\![r]\!]$ are not exactly equal to, respectively, $\mathcal{P}[\![p \vee q]\!]$ and $\mathcal{P}[\![p \vee r]\!]$. However, proving that $=$ holds at this step would be very similar to Law 3.4.10:

$$
\begin{array}{ll}
 & wp\,(LHS,\ I) \\
\Leftrightarrow & \{\ \texttt{def. of } wp \texttt{ on } \|, \texttt{ then on } \longrightarrow\ \} \\
 & (\mathcal{G}[\![\,p\,]\!] \vee \mathcal{G}[\![\,q\,]\!]) \wedge (\mathcal{G}[\![\,p\,]\!] \vee \mathcal{G}[\![\,r\,]\!]) \\
 & \wedge \\
 & wp\,(\mathcal{P}[\![\,p\,]\!] \,\Box\, \mathcal{P}[\![\,q\,]\!],\ (\mathcal{G}[\![\,p\,]\!] \vee \mathcal{G}[\![\,r\,]\!]) \wedge wp\,(\mathcal{P}[\![\,p\,]\!] \,\Box\, \mathcal{P}[\![\,r\,]\!],\ I)) \\
 & \wedge \\
 & wp\,(\mathcal{P}[\![\,p\,]\!] \,\Box\, \mathcal{P}[\![\,r\,]\!],\ (\mathcal{G}[\![\,p\,]\!] \vee \mathcal{G}[\![\,q\,]\!]) \wedge wp\,(\mathcal{P}[\![\,p\,]\!] \,\Box\, \mathcal{P}[\![\,q\,]\!],\ I)) \\
\Leftrightarrow & \{\ \texttt{def. of } wp \texttt{ on } \Box \texttt{ twice}\ \} \\
 & (\mathcal{G}[\![\,p\,]\!] \vee \mathcal{G}[\![\,q\,]\!]) \wedge (\mathcal{G}[\![\,p\,]\!] \vee \mathcal{G}[\![\,r\,]\!]) \\
 & \wedge \\
 & wp\,(\mathcal{P}[\![\,p\,]\!] \,\Box\, \mathcal{P}[\![\,q\,]\!],\ (\mathcal{G}[\![\,p\,]\!] \vee \mathcal{G}[\![\,r\,]\!]) \wedge (wp\,(\mathcal{P}[\![\,p\,]\!],\ I) \vee wp\,(\mathcal{P}[\![\,r\,]\!],\ I))) \\
 & \wedge \\
 & wp\,(\mathcal{P}[\![\,p\,]\!] \,\Box\, \mathcal{P}[\![\,r\,]\!],\ (\mathcal{G}[\![\,p\,]\!] \vee \mathcal{G}[\![\,q\,]\!]) \wedge (wp\,(\mathcal{P}[\![\,p\,]\!],\ I) \vee wp\,(\mathcal{P}[\![\,q\,]\!],\ I))) \\
\Rightarrow & \{\ \texttt{predicate logic: } p \wedge q \Rightarrow p\texttt{, mono-wp-}\Rightarrow \texttt{ of Property 3.3.1}\ \} \\
 & (\mathcal{G}[\![\,p\,]\!] \vee \mathcal{G}[\![\,q\,]\!]) \wedge (\mathcal{G}[\![\,p\,]\!] \vee \mathcal{G}[\![\,r\,]\!]) \\
 & \wedge \\
 & wp\,(\mathcal{P}[\![\,p\,]\!] \,\Box\, \mathcal{P}[\![\,q\,]\!],\ (wp\,(\mathcal{P}[\![\,p\,]\!],\ I) \vee wp\,(\mathcal{P}[\![\,r\,]\!],\ I))) \\
 & \wedge \\
 & wp\,(\mathcal{P}[\![\,p\,]\!] \,\Box\, \mathcal{P}[\![\,r\,]\!],\ (wp\,(\mathcal{P}[\![\,p\,]\!],\ I) \vee wp\,(\mathcal{P}[\![\,q\,]\!],\ I))) \\
\Leftrightarrow & \{\ \texttt{def. of } wp \texttt{ on } \Box\texttt{, then on } \|\texttt{, then on } \longrightarrow\ \} \\
 & wp\,(RHS,\ I)
\end{array}
$$

□

Proof of Law 3.5.23

$$
\begin{array}{ll}
 & \mathcal{P}[\![\,p \wedge (q \vee r)\,]\!] \\
= & \{\ \texttt{axiom of } \mathcal{P} \texttt{ on } \wedge\ \} \\
 & \mathcal{G}[\![\,p\,]\!] \wedge \mathcal{G}[\![\,q \vee r\,]\!] \longrightarrow \mathcal{P}[\![\,p\,]\!] \,\|\, \mathcal{P}[\![\,q \vee r\,]\!] \\
= & \{\ \texttt{axioms of } \mathcal{G} \texttt{ and } \mathcal{P} \texttt{ on } \vee\ \} \\
 & \mathcal{G}[\![\,p\,]\!] \wedge (\mathcal{G}[\![\,q\,]\!] \vee \mathcal{G}[\![\,r\,]\!]) \longrightarrow \mathcal{P}[\![\,p\,]\!] \,\|\, (\mathcal{P}[\![\,q\,]\!] \,\Box\, \mathcal{P}[\![\,r\,]\!]) \\
= & \{\ \texttt{predicate logic: distributivity of } \wedge \texttt{ over } \vee\texttt{, distributivity of } \| \texttt{ over } \Box\ \} \\
 & \begin{array}{lcl}
   \mathcal{G}[\![\,p\,]\!] \wedge \mathcal{G}[\![\,q\,]\!] & & \mathcal{P}[\![\,p\,]\!] \,\|\, \mathcal{P}[\![\,q\,]\!] \\
   \vee & \longrightarrow & \Box \\
   \mathcal{G}[\![\,p\,]\!] \wedge \mathcal{G}[\![\,r\,]\!] & & \mathcal{P}[\![\,p\,]\!] \,\|\, \mathcal{P}[\![\,r\,]\!]
   \end{array} \\
\sqsubseteq & \{\ \texttt{Law 3.4.9, (mono-}\longrightarrow\texttt{) of Law 3.4.1, then Lemma B.2.3}\ \} \\
 & \left(\begin{array}{l}
   \mathcal{G}[\![\,p\,]\!] \wedge \mathcal{G}[\![\,q\,]\!] \\
   \vee \\
   \mathcal{G}[\![\,p\,]\!] \wedge \mathcal{G}[\![\,r\,]\!]
   \end{array}\right) \longrightarrow \left(\begin{array}{l}
   \mathcal{G}[\![\,p\,]\!] \wedge \mathcal{G}[\![\,q\,]\!] \longrightarrow \mathcal{P}[\![\,p\,]\!] \,\|\, \mathcal{P}[\![\,q\,]\!] \\
   \Box \\
   \mathcal{G}[\![\,p\,]\!] \wedge \mathcal{G}[\![\,r\,]\!] \longrightarrow \mathcal{P}[\![\,p\,]\!] \,\|\, \mathcal{P}[\![\,r\,]\!]
   \end{array}\right) \\
= & \{\ \texttt{axiom of } \mathcal{G} \texttt{ on } \wedge\texttt{, twice; axiom of } \mathcal{P} \texttt{ on } \|\texttt{, twice}\ \} \\
 & \mathcal{G}[\![\,p \wedge q\,]\!] \vee \mathcal{G}[\![\,p \wedge r\,]\!] \longrightarrow \mathcal{P}[\![\,p \wedge q\,]\!] \,\Box\, \mathcal{P}[\![\,p \wedge r\,]\!] \\
\sqsubseteq & \{\ \texttt{Law 3.4.9}\ \} \\
 & (\mathcal{G}[\![\,p \wedge q\,]\!] \longrightarrow \mathcal{P}[\![\,p \wedge q\,]\!]) \,\Box\, (\mathcal{G}[\![\,p \wedge r\,]\!] \longrightarrow \mathcal{P}[\![\,p \wedge r\,]\!]) \\
= & \{\ \texttt{axiom of } \mathcal{P} \texttt{ on } \vee\ \} \\
 & \mathcal{P}[\![\,(p \wedge q) \vee (p \wedge r)\,]\!]
\end{array}
$$

□

Proof of Law 3.5.24

$$
\begin{array}{ll}
 & q \\
\sqsubseteq & \{\ p = \mathit{before}\,(p)\text{, Law 3.5.14, sym-}\wedge\text{ in Law 3.5.2}\ \} \\
 & p \wedge q \\
\equiv & \{\ \text{Law 3.5.7}\ \} \\
 & p \wedge (p \Rightarrow q)
\end{array}
$$

□

Proof of Theorem 3.5.1 For (a), it is in general not the case for the BOOSTER predicate equivalence to correspond exactly to the equivalence on standard predicates, because of our implicit interpretation of BOOSTER predicates as guarded programs. A counter example is that we interpret $(p \Rightarrow r) \wedge (q \Rightarrow r)$ as the parallel composition of two guarded programs, whereas $(p \vee q \Rightarrow r)$ is interpreted as just one guarded program. Therefore, we insist, like how we argue the case of Law 3.5.16, the side conditions $p = \mathit{before}\,(p)$ and $q = \mathit{before}\,(q)$, i.e. no updates on the state are intended. This would mean that, by Property 3.2.1, two guarded programs $(p \longrightarrow \mathit{skip})$ and $(q \longrightarrow \mathit{skip})$ will be generated with their guards equivalent. For (b), it is because our notion of program refinement has the reverse relation of implication, hence affecting its source predicates. □

Appendix C

Appendix to Chapter 4

C.1 Proof of Trace Property

Proof of Law 4.4.1 We consider the general patterns of the two workflows as event prefixes: $a1 \to W1$ and $a2 \to W2$. For $(g \,\&\, (a1 \to W1 \mid\mid\mid a2 \to W2)) \,/\, \langle a1 \rangle$ to proceed by executing $a2$, the guard g does not need to be satisfied. This is not equivalent to $(g \,\&\, (a1 \to W1) \mid\mid\mid g \,\&\, (a2 \to W2)) \,/\, \langle a1 \rangle$, for which to proceed by executing $a2$, the guard g needs to be satisfied.

$$
\begin{array}{ll}
 & \mathcal{T}[\![(g \,\&\, (a1 \to W1)) \mid\mid\mid (g \,\&\, (a2 \to W2))]\!] \\
= & \{\ \texttt{guarded action}\ \} \\
 & \mathcal{T}[\![(g \,\&\, a1) \to W1 \mid\mid\mid (g \,\&\, a2) \to W2]\!] \\
= & \{\ \mathcal{T}\ \texttt{on}\ \mid\mid\mid\ \} \\
 & \bigcup \{ t1 \mid\mid\mid t2 \mid t1 \in \mathcal{T}[\![(g \,\&\, a1) \to W1]\!] \wedge t2 \in \mathcal{T}[\![(g \,\&\, a2) \to W2]\!] \} \\
= & \{\ \mathcal{T}\ \texttt{on}\ (g \,\&\, a) \to W\ \} \\
 & \bigcup \{ t1 \mid\mid\mid t2 \mid \\
 & \qquad t1 \in \{\langle\rangle\} \cup \{ \varepsilon(\boxed{\text{a1}}, b, b') \frown t \mid (b, b') \in \mathcal{R}[\![a1]\!] \wedge b \cup b' \models g \wedge t \in \boxed{\mathcal{T}[\![W1]\!]} \} \\
 & \qquad \wedge \\
 & \qquad t2 \in \{\langle\rangle\} \cup \{ \varepsilon(\boxed{\text{a2}}, b, b') \frown t \mid (b, b') \in \mathcal{R}[\![a2]\!] \wedge b \cup b' \models g \wedge t \in \boxed{\mathcal{T}[\![W2]\!]} \} \} \\
\subseteq & \{\ \texttt{relaxing the constraint that}\ g\ \texttt{must be satisfied after}\ a1\ \texttt{or}\ a2\ \} \\
 & \{\langle\rangle\} \cup \{ \varepsilon(\boxed{\text{a1}}, b, b') \frown t \mid (b, b') \in \mathcal{R}[\![a1]\!] \wedge b \cup b' \models g \\
 & \qquad\qquad \wedge \\
 & \qquad\qquad t \in \bigcup \{ t1 \mid\mid\mid t2 \mid t1 \in \boxed{\mathcal{T}[\![W1]\!]} \wedge t2 \in \boxed{\mathcal{T}[\![a2 \to W2]\!]} \} \} \\
 & \cup \\
 & \{\langle\rangle\} \cup \{ \varepsilon(\boxed{\text{a2}}, b, b') \frown t \mid (b, b') \in \mathcal{R}[\![a2]\!] \wedge b \cup b' \models g \\
 & \qquad\qquad \wedge \\
 & \qquad\qquad t \in \bigcup \{ t1 \mid\mid\mid t2 \mid t1 \in \boxed{\mathcal{T}[\![a1 \,\&\, W1]\!]} \wedge t2 \in \boxed{\mathcal{T}[\![W2]\!]} \} \} \\
= & \{\ \mathcal{T}\ \texttt{on}\ \mid\mid\mid\ \} \\
 & \{\langle\rangle\} \cup \{ \varepsilon(a1, b, b') \frown t \mid (b, b') \in \mathcal{R}[\![a1]\!] \wedge b \cup b' \models g \wedge t \in \mathcal{T}[\![W1 \mid\mid\mid a2 \to W2]\!] \} \\
 & \cup \\
 & \{\langle\rangle\} \cup \{ \varepsilon(a2, b, b') \frown t \mid (b, b') \in \mathcal{R}[\![a2]\!] \wedge b \cup b' \models g \wedge t \in \mathcal{T}[\![a1 \to W1 \mid\mid\mid W2]\!] \}
\end{array}
$$

(Continued from the proof of Law 4.4.1)

$$
\begin{array}{ll}
= & \{\, \mathcal{T} \texttt{ on } (g \,\&\, a) \to W \,\} \\
& \mathcal{T}[\![(g \,\&\, a1) \to (W1 \,|||\, a2 \to W2)]\!] \cup \mathcal{T}[\![(g \,\&\, a2) \to (a1 \to W1 \,|||\, W2)]\!] \\
= & \{\, \mathcal{T} \texttt{ on } \Box \,\} \\
& \mathcal{T}[\![(g \,\&\, a1) \to (W1 \,|||\, a2 \to W2) \,\Box\, (g \,\&\, a2) \to (a1 \to W1 \,|||\, W2)]\!] \\
= & \{\, \texttt{guarded action} \,\} \\
& \mathcal{T}[\![g \,\&\, (a1 \to (W1 \,|||\, a2 \to W2)) \,\Box\, g \,\&\, (a2 \to (a1 \to W1 \,|||\, W2))]\!] \\
= & \{\, \& \texttt{ distributes } \Box \,\} \\
& \mathcal{T}[\![g \,\&\, (\, a1 \to (W1 \,|||\, a2 \to W2) \,\Box\, a2 \to (a1 \to W1 \,|||\, W2)\,)]\!] \\
= & \{\, |||\texttt{-step} \,\} \\
& \mathcal{T}[\![g \,\&\, (a1 \to W1 \,|||\, a2 \to W2)]\!]
\end{array}
$$

□

C.2 Proofs of Congruence

Proof of Theorem 4.5.1 We prove by considering all possible syntactic components of the workflow language (i.e. all type constructors of *Wfw*).

Base cases are *stop* and *skip*. For *stop*, we argue as follows:

$$
\begin{array}{ll}
& \Phi[\![stop]\!]_\rho \\
= & \{\, \texttt{def. of } \Phi \,\} \\
& \{t : \Sigma^* \mid (\exists\, Q : Node \bullet (stop, \rho) \stackrel{t}{\Longrightarrow} Q)\} \\
= & \{\, \texttt{case analysis: } \Sigma^* = \{\langle\rangle\} \cup \Sigma^+ \,\} \\
& \{\, \langle\rangle \,\} \cup \{t : \Sigma^+ \mid (\exists\, Q : Node \bullet (stop, \rho) \stackrel{t}{\Longrightarrow} Q)\} \\
= & \{\, \texttt{no transition rule for } stop \,\} \\
& \{\, \langle\rangle \,\} \\
= & \{\, \texttt{def. of } \mathcal{T} \,\} \\
& \mathcal{T}[\![stop]\!]_\rho
\end{array}
$$

Similarly, we argue the congruence of *skip* as follows:

$$
\begin{array}{ll}
& \Phi[\![skip]\!]_\rho \\
= & \{\, \texttt{def. of } \Phi \,\} \\
& \{t : \Sigma^* \mid (\exists\, Q : Node \bullet (skip, \rho) \stackrel{t}{\Longrightarrow} Q)\} \\
= & \{\, \texttt{case analysis: } \Sigma^* = \{\langle\rangle\} \cup \Sigma^+ \,\} \\
& \{\, \langle\rangle \,\} \cup \{t : \Sigma^+ \mid (\exists\, Q : Node \bullet (skip, \rho) \stackrel{t}{\Longrightarrow} Q)\} \\
= & \{\, \texttt{transition rule for } skip\texttt{, operation of } \cup \,\} \\
& \{\langle\rangle, \langle\checkmark\rangle\} \\
= & \{\, \texttt{def. of } \mathcal{T} \,\} \\
& \mathcal{T}[\![skip]\!]_\rho
\end{array}
$$

For the case of action prefix, we want to prove that

$$\begin{array}{l}\forall\, \rho : \mathit{Env};\ a : \mathit{Act};\ W : \mathit{Wfw} \bullet \\ \quad \Phi[\![a \to W]\!]_\rho \ = \ \{\langle\rangle\} \cup \{b : \mathcal{R}[\![a]\!]\,(\rho);\ t' : \Sigma^* \mid t' \in \boxed{\Phi[\![W]\!]_{\mathrm{dom}\,\rho \lhd b'}} \\ \qquad\qquad\qquad\qquad \bullet\ \langle \varepsilon\,(a, b, b')\rangle \frown t'\} \\ \qquad\qquad\quad = \ \mathcal{T}[\![a \to W]\!]^\Phi_\rho\end{array}$$

where we assume that $(b, b') \in \mathcal{R}[\![a]\!]$ and we argue as follows:

$$\begin{array}{ll} & \Phi[\![a \to W]\!]_\rho \\ = & \{\ \texttt{def. of}\ \Phi\ \} \\ & \{t : \Sigma^* \mid (\exists\, Q : \mathit{Node} \bullet (a \to W, \rho) \stackrel{t}{\Longrightarrow} Q)\} \\ = & \{\ \texttt{case analysis:}\quad \Sigma^* = \{\langle\rangle\} \cup \Sigma^+\ \} \\ & \{\langle\rangle\} \cup \{t : \Sigma^+ \mid (\exists\, Q : \mathit{Node} \bullet (a \to W, \rho) \stackrel{t}{\Longrightarrow} Q)\} \\ = & \{\ \texttt{transition rule for}\ \to\ \} \\ & \{\langle\rangle\} \cup \{\langle \varepsilon\,(a, b, b')\rangle \frown t \mid (b, b') \in \mathcal{R}[\![a]\!] \wedge t \in \Sigma^+ \wedge \\ & \qquad\qquad \left(\begin{array}{c} \exists\, Q : \mathit{Node} \bullet (a \to W, \rho) \stackrel{\varepsilon\,(a,b,b')}{\longrightarrow} (W, \rho') \wedge \\ (W, \rho') \stackrel{t}{\Longrightarrow} Q \end{array} \right)\} \\ = & \{\ \texttt{def. of}\ \Phi\ \} \\ & \{\langle\rangle\} \cup \{\langle \varepsilon\,(a, b, b')\rangle \frown t \mid (b, b') \in \mathcal{R}[\![a]\!] \wedge t \in \Phi[\![W]\!]_{\rho'}\} \\ = & \{\ \texttt{def. of}\ \mathcal{T}[\![\_]\!]^\Phi_\_\ \} \\ & \mathcal{T}[\![a \to W]\!]^\Phi_\rho \end{array}$$

The proofs for other workflow combinators are proved in a similar way:

$$\begin{array}{l}\forall\, \rho : \mathit{Env};\ \mathit{W1}, \mathit{W2} : \mathit{Wfw} \bullet \\ \quad \Phi[\![\mathit{W1} \,\Box\, \mathit{W2}]\!]_\rho \ = \ \boxed{\Phi[\![\mathit{W1}]\!]_\rho} \cup \boxed{\Phi[\![\mathit{W2}]\!]_\rho} \\ \quad \Phi[\![\mathit{W1}\ ;\ \mathit{W2}]\!]_\rho \ = \ \{t \mid t \in \boxed{\Phi[\![\mathit{W1}]\!]_\rho} \wedge \checkmark \notin \mathrm{ran}(t)\} \\ \qquad\qquad\qquad\quad \cup \\ \qquad\qquad\qquad\quad \{\mathit{t1} \frown \mathit{t2} \mid \checkmark \notin \mathrm{ran}(\mathit{t1}) \wedge \\ \qquad\qquad\qquad\qquad\qquad \mathit{t1} \frown \langle\checkmark\rangle \in \boxed{\Phi[\![\mathit{W1}]\!]_\rho} \wedge \mathit{t2} \in \boxed{\Phi[\![\mathit{W2}]\!]_\rho}\} \\ \quad \Phi[\![\mathit{W1} \,|||\, \mathit{W2}]\!]_\rho \ = \ \bigcup\{\mathit{t1} \,|||\, \mathit{t2} \mid \mathit{t1} \in \boxed{\Phi[\![\mathit{W1}]\!]_\rho} \wedge \mathit{t2} \in \boxed{\Phi[\![\mathit{W2}]\!]_\rho}\} \\ \quad \Phi[\![\mathrm{var}\ X\ .\ W]\!]_\rho \ = \ \boxed{\Phi[\![W]\!]_{\rho \oplus \{X \mapsto \bot\}}}\end{array}$$

□

C.3 Proofs of Precondition Calculation

Proof of Lemma 4.7.1 We prove lemma 4.7.1 by a structural induction on the number of iterations that have been executed. For the first conjunct we consider lines 4 - 5 in Figure 4.2.

In the first iteration, $i = m$ and $matrix[m+1][n+1] = true$ (line 2), we argue that

$$
\begin{array}{ll}
 & matrix[m][n+1] \\
= & \{ \texttt{line 5 of } \textit{commute} \} \\
 & wp\,(intent\ s[m],\ matrix[m+1][n+1]) \\
= & \{ \texttt{line 3 of } \textit{commute} \} \\
 & wp\,(intent\ s[m],\ true) \\
= & \{ \texttt{def. of } \textit{complete} \} \\
 & complete\,(s_{m..})
\end{array}
$$

We state our inductive hypothesis as follows, after the k_{th} iteration where $1 < k < m$ (i.e. not the very last iteration of lines 4 - 5).

$$
\begin{array}{l}
\forall\, i : int \mid i \in (m-k+1)\,..\,m \bullet \\
\qquad matrix\,[i][n+1] = complete\,(s_{i..})
\end{array}
$$

The lower bound expression $m-k+1$ characterises the fact that we look at operations of s in their reverse order (i.e. right to left in $matrix$), where in the 1st iteration $k = 1$ and $i \in m\,..\,m$, in the 2nd iteration $k = 2$ and $i \in m-1\,..\,m$, ..., and in the very last iteration $k = m$ and $i \in 1\,..\,m$.

For the $(k+1)_{th}$ iteration, where i is decremented to $m-k$, we argue that

$$
\begin{array}{ll}
 & matrix[m-k][n+1] \\
= & \{ \texttt{line 5 of } \textit{commute} \} \\
 & wp\,(intent\ t[m-k],\ matrix[m-k+1][n+1]) \\
= & \{ \texttt{inductive hypothesis} \} \\
 & wp\,(intent\ t[m-k],\ complete\,(s_{m-k+1..})) \\
= & \{ \texttt{def. of } \textit{complete} \} \\
 & complete\,(s_{m-k..})
\end{array}
$$

We have proved the first conjunct of lemma 4.7.1, and a similar argument applies to its second conjunct. □

Proof of Lemma 4.7.2 We prove lemma 4.7.2 by a structural induction on the number of iterations that have been executed between lines 8 and 11 in Figure 4.2.

In the very first iteration, where $i = m$ and $j = n$, we argue that

$$
\begin{array}{ll}
 & matrix[m][n] \\
= & \{\ \texttt{lines 10, 11 of } \textit{commute}\ \} \\
 & wp\,(intent\ s[m],\ matrix[m+1][n]) \wedge wp\,(intent\ t[n],\ matrix[m][n+1]) \\
= & \{\ \texttt{Lemma 4.7.1}\ \} \\
 & wp\,(intent\ s[m],\ complete\,(t_{n..})) \wedge wp\,(intent\ t[n],\ complete\,(s_{m..})) \\
= & \{\ \texttt{def. of } \textit{complete}\ \} \\
 & complete\,(\langle s[m]\rangle \frown t_{n..}) \wedge complete\,(\langle t[n]\rangle \frown s_{m..}) \\
= & \{\ \texttt{def. of } t_{n..},\ s_{m..}\ \} \\
 & complete\,(\langle s[m]\rangle \frown \langle t[n]\rangle) \wedge complete\,(\langle t[n]\rangle \frown \langle s[m]\rangle) \\
= & \{\ \texttt{def. of } |||\ \} \\
 & \bigwedge g : \{complete\,(u) \mid u \in s_{m..} \mid\mid\mid t_{n..}\} \bullet g
\end{array}
$$

We state our inductive hypothesis as follows, after executing the iteration where $i = k$ and $j = l$, with $(k, l) \in \{k, l \mid 1 \leq k \leq m \wedge 1 \leq l \leq n\} \setminus \{(m, n)\}$.

$$
\begin{array}{l}
\forall\, i, j : int \mid (i \in k\,..\,m \wedge j = l) \wedge (i \in 1\,..\,m \wedge j \in l+1\,..\,n) \bullet \\
\qquad matrix\,[i][j] = \bigwedge g : \{complete\,(u) \mid u \in s_{i..} \mid\mid\mid t_{j..}\} \bullet g
\end{array}
$$

The range of i and j is a conjunction. The left conjunct denotes all cells—at the same line l—that are to the "right" of cell $matrix\,[i][j]$. The right conjunct denotes all rows that are "below" line l where cell $matrix\,[i][j]$ is located.

There are two cases for the next iteration.

In the first case, we stay in the current row l, let j retain its value as l, and decrement i to $k-1$ to move to the next cell at the left. We argue that

$$
\begin{array}{ll}
 & matrix[k-1][l] \\
= & \{\ \texttt{lines 10 - 11 of } \textit{commute}\ \} \\
 & wp\,(intent\ s[k-1],\ matrix[k][l]) \wedge wp\,(intent\ t[l],\ matrix[k-1][l+1]) \\
= & \{\ \texttt{inductive hypothesis}\ \} \\
 & wp\,(intent\ s[k-1],\ \bigwedge g : \{complete\,(u) \mid u \in s_{k..} \mid\mid\mid t_{l..}\} \bullet g) \\
 & \wedge \\
 & wp\,(intent\ t[l],\ \bigwedge g : \{complete\,(u) \mid u \in s_{k-1..} \mid\mid\mid t_{l+1..}\} \bullet g) \\
= & \{\ \texttt{dist-wp-}\wedge\ \texttt{of Law 3.3.1 on page 50}\ \} \\
 & \bigwedge g : \{wp\,(intent\ s[k-1], complete\,(u)) \mid u \in s_{k..} \mid\mid\mid t_{l..}\} \bullet g \\
 & \wedge \\
 & \bigwedge g : \{wp\,(intent\ t[l], complete\,(u)) \mid u \in s_{k-1..} \mid\mid\mid t_{l+1..}\} \bullet g \\
= & \{\ \texttt{def. of } \textit{complete}\ \} \\
 & \bigwedge g : \{complete\,(\langle s[k-1]\rangle \frown u) \mid u \in s_{k..} \mid\mid\mid t_{l..}\} \bullet g \\
 & \wedge \\
 & \bigwedge g : \{complete\,(\langle t[l]\rangle \frown u) \mid u \in s_{k-1..} \mid\mid\mid t_{l+1..}\} \bullet g \\
= & \{\ \texttt{def. of } |||\texttt{, also see below}\ \} \\
 & \bigwedge g : \{complete\,(u) \mid u \in s_{k-1..} \mid\mid\mid t_{l..}\} \bullet g
\end{array}
$$

There are facts about how we expand the tails of s and t in *commute*: $s_{k-1..} = \langle s[k-1] \rangle \frown s_{k..}$ and $t_{l..} = \langle t[l] \rangle \frown t_{l+1..}$. These imply that $s_{k-1..} \mid\mid\mid t_{l..}$ is equivalent to the union of sets $\{\langle s[k-1] \rangle \frown u \mid s_{k..} \mid\mid\mid t_{l..}\}$ and $\{\langle t[l] \rangle \frown u \mid u \in s_{k-1..} \mid\mid\mid t_{l+1..}\}$.

In the second case, we decrement j to $l-1$ to move up to the row above and reset i to m to refer to the "right-most" cell in that row. We argue that

$$
\begin{array}{ll}
 & matrix[m][l-1] \\
= & \{ \texttt{lines 10 - 11 of } \textit{commute} \} \\
 & wp\,(intent\ s[m],\ matrix[m+1][l-1]) \wedge wp\,(intent\ t[l-1],\ matrix[m][l]) \\
= & \{ \texttt{lemma 4.7.1, inductive hypothesis} \} \\
 & wp\,(intent\ s[m],\ complete\,(t_{l-1..})) \\
 & \wedge \\
 & wp\,(intent\ t[l-1],\ \bigwedge g : \{complete\,(u) \mid u \in s_{m..} \mid\mid\mid t_{l..}\} \bullet g) \\
= & \{ \texttt{dist-wp-}\wedge \texttt{ of Law 3.3.1 on page 50} \} \\
 & wp\,(intent\ s[m],\ complete\,(t_{l-1..})) \\
 & \wedge \\
 & \bigwedge g : \{wp\,(intent\ t[l-1], complete\,(u)) \mid u \in s_{m..} \mid\mid\mid t_{l..}\} \bullet g \\
= & \{ \texttt{def. of } complete \} \\
 & complete\,(\langle s[m] \rangle \frown t_{l-1..}) \\
 & \wedge \\
 & \bigwedge g : \{complete\,(\langle t[l-1] \rangle \frown u) \mid u \in s_{m..} \mid\mid\mid t_{l..}\} \bullet g \\
= & \{ \texttt{def. of } \mid\mid\mid \texttt{, also see below} \} \\
 & \bigwedge g : \{complete\,(u) \mid u \in s_{m..} \mid\mid\mid t_{l-1..}\} \bullet g
\end{array}
$$

The argument for the last step of the second case is similar to that for the first case. We have the facts $s_{m..} = \langle s[m] \rangle \frown \langle\rangle$ and $t_{l-1..} = \langle t[l-1] \rangle \frown t_{l..}$, which imply that $s_{m..} \mid\mid\mid t_{l-1..}$ is equivalent to the union of sets $\{\langle s[m] \rangle \frown t_{l-1..}\}$ and $\{\langle t[l-1] \rangle \frown u \mid u \in s_{m..} \mid\mid\mid t_{l..}\}$. Therefore, we have proved lemma 4.7.2. □

Proof of Theorem 4.7.2 We prove by contradiction. We assume the negation of Theorem 4.7.2:

$$\exists f : \mathit{Wfw} \to \mathbb{P}(\mathrm{seq}\, \mathit{Act}) \bullet \left(\begin{array}{l} \forall\, W_1, W_2 : \mathit{Wfw} \bullet \\ \quad f\,(W_1) \subseteq \mathcal{T}'[\![W_1]\!] \wedge \\ \quad f\,(W_2) \subseteq \mathcal{T}'[\![W_2]\!] \wedge \\ \quad \mathcal{G}_{com}[\![W_1 \parallel\!\parallel W_2]\!] = \\ \qquad \bigwedge g : \{\mathit{commute}\,(s,t) \mid s \in f\,(W_1) \wedge t \in f\,(W_2)\} \bullet g \end{array} \right) \wedge f \neq \mathcal{T}'$$

The second conjunct implies that $\exists\, W : \mathit{Wfw} \bullet f\,(W) \neq \mathcal{T}'[\![W]\!]$, which can be further strengthened because of the first conjunct: $\exists\, W : \mathit{Wfw} \bullet f\,(W) \subset \mathcal{T}'[\![W]\!]$. Say we pick such a workflow W and we argue as follows:

$$\begin{array}{ll} & \mathcal{G}_{com}[\![W \parallel\!\parallel W]\!] \\ = & \{\ \texttt{def. of}\ \mathcal{G}_{com}\ \} \\ & \bigwedge g : \{\mathit{commute}\,(s,t) \mid s \in \mathcal{T}'[\![W]\!] \wedge t \in \mathcal{T}'[\![W]\!]\} \bullet g \\ = & \{\ f(W) \subset \mathcal{T}'[\![W]\!]\ \} \\ & \bigwedge g : \{\mathit{commute}\,(s,t) \mid s \in f(W) \wedge t \in f(W)\} \bullet g \\ & \wedge \\ & \bigwedge g : \{\mathit{commute}\,(s,t) \mid s \in \mathcal{T}'[\![W]\!] \setminus f(W) \wedge t \in \mathcal{T}'[\![W]\!] \setminus f(W)\} \bullet g \\ \neq & \{\ f(W) \cap (\mathcal{T}'[\![W]\!] \setminus f(W)) = \emptyset\ \} \\ & \bigwedge g : \{\mathit{commute}\,(s,t) \mid s \in f(W) \wedge t \in f(W)\} \bullet g \end{array}$$

But this contradicts with the first conjunct of our assumption, so our initial assumption (i.e. the negation of theorem 4.7.2) must be false.

We hence prove theorem 4.7.2. □

Appendix D

Appendix to Chapter 5

Proof of Theorem 5.4.1 Each equivalence law $p \equiv q$ in Table 3.1 is proved, by Definitions 3.5.2 and 3.4.2, that

$$\forall\, I : \mathit{Predicate} \bullet \mathit{wp}\,(\mathcal{P}\,[\![\, p \,]\!],\ I) = \mathit{wp}\,(\mathcal{P}\,[\![\, q \,]\!],\ I)$$

This suggests that in a context where the behaviour of p (i.e. $\mathcal{P}\,[\![\, p \,]\!]$) is able to maintain the invariant I, then so is the behaviour of q (i.e. $\mathcal{P}\,[\![\, q \,]\!]$), and vice versa. □

Proof of Theorem 5.4.2 Each refinement law $p \sqsubseteq q$ in Table 3.1 is proved, by Definitions 3.5.1 and 3.4.1, that

$$\forall\, I : \mathit{Predicate} \bullet \mathit{wp}\,(\mathcal{P}\,[\![\, q \,]\!],\ I) \Rightarrow \mathit{wp}\,(\mathcal{P}\,[\![\, p \,]\!],\ I)$$

This suggests that in a context where the behaviour of q (i.e. $\mathcal{P}\,[\![\, q \,]\!]$) is able to maintain the invariant I, then so is the behaviour of p (i.e. $\mathcal{P}\,[\![\, p \,]\!]$), but the converse is not true—it may be that in certain contexts where the execution of $\mathcal{P}\,[\![\, p \,]\!]$ is able to maintain the invariant I, the execution of $\mathcal{P}\,[\![\, q \,]\!]$ is blocked as it contains a strictly guarding constraint. □

Appendix E

Appendix to Chapter 6

E.1 Z Models

We first define a list of predefined sets: the set of possible names of models, classes, operations, properties, attributes, and variables.

$$[N\_MODEL, N\_CLASS, N\_OPERATION]$$

$$[N\_PROPERTY, N\_B\_ATTRIBUTE, N\_VARIABLE]$$

Also, we have *TYPE* and *PREDICATE* to denote the sets of, respectively, entity types and predicates that have been defined in Section A.1.

$$[TYPE, PREDICATE]$$

To identify a property—either an attribute or an association—we supply its name and its declaring class.

$$IDEN\_PROPERTY == N\_CLASS \times N\_B\_ATTRIBUTE$$

E.1.1 Object Model Structure

We now define the type system contained in an object model, and this corresponds to the Haskell programming type as defined in Section E.3.2 (on page 236). On the top level we have an object model which consists of a name, a list of set declarations, and a list of class mappings.

ObjectModel

$$nObjectModel : N\_MODEL$$
$$sets : N\_SET \pfun \mathbb{P}\ Value$$
$$classes : N\_CLASS \pfun CLASS$$

Each class consists of a list of properties, including both attributes and associations, and operations.

$$
\begin{array}{|l}
\hline
CLASS \\
properties : N\_PROPERTY \pfun PROPERTY \\
operations : N\_OPERATION \pfun OPERATION \\
\hline
\end{array}
$$

To express the constraint that a member field is optional, we model its value as a set that can be either singleton or empty.

$$T[0 \,..\, 1] == \{s : \mathbb{P}\, T \mid \#s \leq 1\}$$

A class property has a kind, a type, and may have an opposite if it is declared as a bi-directional association.

$$
\begin{array}{|l}
\hline
PROPERTY \\
kind : KINDtype : TYPE \\
opposite : OPPOSITE[0 \,..\, 1] \\
\hline
\end{array}
$$

The kind of a property may either be an attribute or an association.

$$Kind ::= Attribute \mid Association$$

The type of a property consists of its minimum and maximum multiplicities, its ordering of member elements if applicable, and its base type.

$$
\begin{array}{|l}
\hline
TYPE \\
min, max : MULTIPLICITY \\
ordering : ORDER[0 \,..\, 1] \\
base : BASE \\
\hline
\end{array}
$$

The multiplicity of a property may be either finite or infinite.

$$MULTIPLICITY ::= Multiplicity\langle\langle \mathbb{N} \rangle\rangle \mid Infinity$$

The base type of a property may be that of a class, of a declared set, an integer, or a string.

$$
\begin{array}{rcl}
Base & ::= & ClassBase\langle\langle N\_CLASS \rangle\rangle \\
& \mid & SetBase\langle\langle N\_SET \rangle\rangle \\
& \mid & IntBase \\
& \mid & StrBase
\end{array}
$$

The ordering mechanism of a property, if applicable, may be either a set of q sequence.

$ORDER ::= Set \mid Seq$

Finally, in the case of an association property, its opposite consists of the target class and the property that navigates back to the current class.

Opposite

$class : N\_CLASS$
$property : N\_PROPERTY$

We do not include the schema type of *OPERATION* here; its definition is similar to the Haskell programming data type as defined in Section 6.3.2 (on page 152).

E.1.2 Table Model State

We define the structure of a Table model state which reflects our implementation strategy as presented in Section E.3.3.

TableModel

$ObjectModel$

$nTableModel : N\_MODEL$
$nClassTables : \mathbb{P}\, N\_CLASS$
$classTables : N\_CLASS \rightarrow \mathbb{P}\, IDEN\_PROPERTY$
$assocTables, setTables, seqTables : \mathbb{P}\, IDEN\_PROPERTY$
$intProperty, strProperty, refProperty,$
$manProperty, optProperty, setProperty, seqProperty : \mathbb{P}\, IDEN\_PROPERTY$
$oppositeOf : IDEN\_PROPERTY \nrightarrow IDEN\_PROPERTY$
$opNames : N\_CLASS \rightarrow \mathbb{P}\, N\_OPERATION$
$opInputs, opOutputs : N\_CLASS \nrightarrow N\_OPERATION \nrightarrow N\_VARIABLE$
$opIOType : N\_CLASS \nrightarrow N\_OPERATION \nrightarrow N\_VARIABLE \nrightarrow TYPE$
$opSpec, opIntent : N\_CLASS \nrightarrow N\_OPERATION \nrightarrow SUBSTITUTION$
$opGuard : N\_CLASS \nrightarrow N\_OPERATION \nrightarrow PREDICATE$

E.1.3 Sql Queries

We supply detailed definitions of queries that related to Section 6.4.3. The query

`INSERT INTO` *table*? (*cols*?) `VALUE` (*vals*?)

inserts a new tuple into a table and takes as inputs *table*? a table name, *cols*? the list of columns to be constructed for the new tuple, and *vals*? the list of values to be assigned accordingly to those columns. We require that *table*? must be the name of an existing table in the current state, that *cols*? intends to construct a tuple whose

columns already exist in the table referred to by *table*?, and that the two lists *cols*? and *vals*? are equal in their sizes.

InsertInput

$$
\begin{array}{l}
s : \mathcal{S}_{sql} \\
table? : N\_TABLE \\
cols? : \operatorname{seq} N\_COLUMN \\
vals? : \operatorname{seq} SQL\_EXPR \\
\hline
table? \in \operatorname{dom} s.tuples \\
\forall\, t : s.tuples\,(table?) \bullet \operatorname{ran} cols? \subseteq \operatorname{dom} t.values \\
\#cols? = \#vals?
\end{array}
$$

Locally, we specify how to construct the new tuple to be inserted, through collecting column-to-value pairs, each of which drawn from items at the same valid position of *cols*? and *vals*?.

NewTuple

$$
\begin{array}{l}
Tuple \\
cols? : \operatorname{seq} N\_COLUMN \\
vals? : \operatorname{seq} SQL\_EXPR \\
\hline
values = \{i : \operatorname{dom} cols? \bullet cols?\,(i) \mapsto eval_{sql}\,(vals?\,(i))\}
\end{array}
$$

Globally, we update contents of the table that *table*? refers to, by adding to it the new tuple as specified in the schema *NewTuple*.

InsertEffect

$$
\begin{array}{l}
s, s' : \mathcal{S}_{sql} \\
InsertInput \\
NewTuple \\
\hline
s'.tuples = s.tuples \oplus \{table? \mapsto \\
\qquad\qquad s.tuples\,(table?) \cup \{\theta\ Tuple\}\}
\end{array}
$$

We are now able to define the formal semantics of the `INSERT` query by placing $\mathcal{R}_{sql}$ and other relevant schemas above into context and defining its *effect* relation accordingly. The `INSERT` query, similar to `UPDATE`, has no effect on the execution stack.

INSERT

$\Xi Stack$
$\mathcal{R}_{sql}$
$InsertInput$
$NewTuple$

$input = output = \{\}$

$\forall\, s, s' : \mathcal{S}_{sql} \bullet InsertEffect \Leftrightarrow$
$(s, \{\}) \mapsto (s', \{\}) \in effect$

The query

DELETE FROM *table*?
WHERE *cond*?

removes from a table the set of tuples that satisfy a condition and takes as inputs *table*? a table name and *cond*? a Boolean condition that specifies the range of tuples to be deleted. We require that *table*? must be the name of an existing table in the current state.

DeleteInput

$s : \mathcal{S}_{sql}$
$table? : N\_TABLE$
$cond? : SQL\_EXPR$

$table? \in \operatorname{dom} s.tuples$

We represent the effect of **DELETE** as removing, from the table referred to by *table*?, those satisfying tuples of *cond*?.

DeleteEffect

$s, s' : \mathcal{S}_{sql}$
$DeleteInput$

$s'.tuples = s.tuples \oplus \{table? \mapsto$
$s.tuples\,(table?) \setminus sat\,(cond?)\,(s.tuples\ table?)\}$

We are now able to define the formal semantics of the **DELETE** query by placing $\mathcal{R}_{sql}$ and the relevant schemas above into context and defining its *effect* relation accordingly. The **INSERT** query, similar to **UPDATE** and **INSERT**, has no effect on the execution stack.

DELETE

$$\Xi Stack$$
$$\mathcal{R}_{sql}$$
$$DeleteInput$$

$$input = output = \{\}$$

$$\forall\, s, s' : \mathcal{S}_{sql} \bullet$$
$$DeleteEffect \Leftrightarrow (s, \{\}) \mapsto (s', \{\}) \in effect$$

We now define the formal semantics of queries that modify the execution stack only but not the database state (of type $\mathcal{S}_{sql}$). We define a schema to represent the fact that each of these queries has no effect on the state, by insisting that its *effect* is the identity relation on the state. More precisely, we define

SideEffectFreeQuery

$$\mathcal{R}_{sql}$$

$$input = output = \{\} \land effect = \mathrm{id}\,(\mathcal{S}_{sql} \times \{\})$$

The query **DECLARE** *var*? *type*? declares a local variable and takes as inputs *var*? a variable name and *type*? a valid SQL data type. We require that *var*? has not been declared already. More precisely, we define

DeclVarInput

$$Stack$$
$$var? : N\_VARIABLE$$
$$type? : SQL\_DATA\_TYPE$$

$$var? \notin \mathrm{dom}\, vars$$

The effect of declaring a local variable is only on the execution stack, by initialising the new variable as undefined (i.e. $\bot$). More precisely, we define

DeclVarStackEffect

$$\Delta Stack$$
$$DeclVarInput$$

$$vars' = vars \cup \{var? \mapsto \bot\} \land cursor'_{pos} = cursor_{pos} \land cursor'_{vals} = cursor_{vals}$$

We are now able to define the formal semantics of a local variable declaration in terms of its effect on the execution stack and database state:

DECLARE$_{var}$

$$DeclVarStackEffect;\ SideEffectFreeQuery$$

The query

`SELECT`	*col*?
`INTO`	*var*?
`FROM`	*table*?
`WHERE`	*cond*?

redefines the value of a local variable and takes as inputs *col*? the target column, *var*? the variable to be assigned to, *table*? the containing table, and *cond*? the condition that specifies the range of tuples to draw the column values. In our use of the `SELECT_INTO` query, we require that *cond*? identifies a single tuple in the table. We also require that *table*? refers to an existing table in the current state and *col*? is a valid column of its, and *var*? is already declared.

SelectIntoInput

$$
\begin{array}{l}
Stack \\
s : \mathcal{S}_{sql} \\
col? : N\_COLUMN \\
var? : N\_VARIABLE \\
table? : N\_TABLE \\
cond? : SQL\_EXPR \\
\hline
table? \in \operatorname{dom} s.tuples \\
\exists_1\, t : Tuple \bullet t \in sat\,(cond?)\,(s.tuples\,(table?)) \\
\forall\, t : s.tuples\,(table?) \bullet col? \in \operatorname{dom} t.values \\
var? \in \operatorname{dom} vars
\end{array}
$$

The `SELECT_INTO` query retrieves from the current database state the value of the input column from the (sole) satisfying tuple in the input table.

SelectIntoQuery

$$
\begin{array}{l}
s : \mathcal{S}_{sql} \\
SelectIntoInput \\
q! : ScalarValue \\
\hline
q! = (\textbf{let}\ row == (\mu\, t : Tuple \mid t \in sat\,(cond?)\,(s.tuples\,(table?))) \bullet \\
\qquad\qquad row.values\,(col?))
\end{array}
$$

The effect of `SELECT_INTO` on the execution stack is on replacing the existing value of the local variable in question with a value as specified in *SelectIntoQuery*.

SelectIntoStackEffect

$$\Delta Stack$$
$$SelectIntoInput$$
$$SelectIntoQuery$$
$$cursor'_{pos} = cursor_{pos}$$
$$cursor'_{vals} = cursor_{vals}$$
$$vars' = vars \oplus \{var? \mapsto q!\}$$

We are now able to define the formal semantics of redefining the value of a local variable in terms of its effect on the execution stack and database state.

`SELECT_INTO`

$$SelectIntoStackEffect$$
$$SideEffectFreeQuery$$

The query

`DECLARE` *cur*? `CURSOR FOR` *list*?

declares a cursor over a collection of values for later iterations and takes as inputs *cur*? the name of the cursor and *list*? the expression of collection. We require that *cur*? has not been declared as a cursor before.

DeclCurInput

$$Stack$$
$$cur? : N\_VARIABLE$$
$$list? : SQL\_EXPR$$
$$cur? \notin \mathrm{dom}\, cursor_{pos}$$

When a cursor is first declared but not yet activated, although we evaluate the collection of values over which it is supposed to iterate, we give it an invalid current position (i.e. *0*) so that it is yet able to return a value.

DeclCurStackEffect

$$\Delta Stack$$
$$DeclCurInput$$
$$vars' = vars$$
$$cursor'_{pos} = cursor_{pos} \cup \{cur? \mapsto 0\}$$
$$cursor'_{vals} = cursor_{vals} \cup \{cur? \mapsto eval^*_{sql}\,(list?)\}$$

We are now able to define the formal semantics of a cursor declaration in terms of its effect on the execution stack and database state.

DECLARE$_{cur}$

$$DeclCurStackEffect$$
$$SideEffectFreeQuery$$

The query

OPEN $cur?$

activates the iteration of a cursor and takes as input $cur?$ a cursor name. We require that $cur?$ has been declared already.

OpenCurInput

$$Stack$$
$$cur? : N\_VARIABLE$$
$$cur? \in \mathrm{dom}\ cursor_{vals}$$

The effect of activating a cursor is on the execution stack: it assigns that cursor's current position to a valid value (i.e. *1*) such that a later call to fetch its value can succeed.

OpenCurStackEffect

$$\Delta Stack$$
$$OpenCurInput$$
$$vars' = vars$$
$$cursor'_{pos} = cursor_{pos} \oplus \{cur? \mapsto 1\}$$
$$cursor'_{vals} = cursor_{vals}$$

We are now able to define the formal semantics of a cursor activation in terms of its effect on the execution stack and database state.

OPEN

$$OpenCurStackEffect$$
$$SideEffectFreeQuery$$

The query

CLOSE $cur?$

deactivates the iteration capability of a cursor and takes as input $cur?$ a cursor name. We require that $cur?$ has been declared already.

CloseCurInput

$Stack$
$cur? : N\_VARIABLE$

$cur? \in \mathrm{dom}\, cursor_{vals}$

The effect of deactivating a cursor is to restore to it back to state where it was first declared: it assigns that cursor's current position to a invalid value (i.e. 0) such that it has to be activated again before its use.

CloseCurStackEffect

$\Delta Stack$
$CloseCurInput$

$vars' = vars$
$cursor'_{pos} = cursor_{pos} \oplus \{cur? \mapsto 0\}$
$cursor'_{vals} = cursor_{vals}$

We are now able to define the formal semantics of a cursor deactivation in terms of its effect on the execution stack and database state.

CLOSE

$CloseCurStackEffect$
$SideEffectFreeQuery$

The query

`DECLARE` $cur?$ `INTO` $x?$

assigns a local variable to a value fetched from a collection currently being iterated by a cursor and takes as input $cur?$ a cursor name and $x?$ the local variable to be assigned. We require that both $x?$ and $cur?$ have already been declared.

FetchCurInput

$Stack$
$cur?, x? : N\_VARIABLE$

$cur? \in \mathrm{dom}\, cursor_{vals}$
$x? \in \mathrm{dom}\, vars$

The effect of this query is on retrieving the value currently pointed by the cursor and overriding it for that of $x?$, as well as shifting the cursor position by one place accordingly.

FetchCurStackEffect

$\Delta Stack$
FetchCurInput

$vars' = vars \oplus \{x? \mapsto cursor_{vals}\,(cur?)\,(cursor_{pos}\,(cur?))\}$
$cursor'_{pos} = cursor_{pos} \oplus \{cur? \mapsto cursor_{pos}\,(cur?) + 1\}$
$cursor'_{vals} = cursor_{vals}$

We are now able to define the formal semantics of this two-step query in terms of its effect on the execution stack and database state.

FETCH

FetchCurStackEffect
SideEffectFreeQuery

E.2 Linking Mode States

In the context of a particular $\mathcal{S}_{obj}$, we define mapping functions $\mathcal{M}$ that retrieve for the various class properties their corresponding mappings, each of which consists of the reference of the current object and the value of its property. For primitive properties, we keep *null* value as it is.

$$\begin{array}{l}\mathcal{M}_{prim} == \\ \quad (\lambda\, s : \mathcal{S}_{tab};\ p : NClass \times NProperty \bullet \\ \quad\quad \{o : ObjectId;\ v : Value \mid o \in s.extent\,(fst\,p) \land v = s.value\,(o)\,(snd\,p) \bullet \\ \quad\quad\quad\quad [\![o]\!]^{SV} \mapsto [\![v]\!]^{SV}\})\end{array}$$

For associations, we represent *null* link as empty.

$$\begin{array}{l}\mathcal{M}_{assoc} == \\ \quad (\lambda\, s : \mathcal{S}_{tab};\ p : NClass \times NProperty \bullet \\ \quad\quad \bigcup\{o : ObjectId;\ v : Value \mid o \in s.extent\,(fst\,p) \land v = s.value\,(o)\,(snd\,p) \bullet \\ \quad\quad\quad\quad \textbf{if } v = null \textbf{ then } \{\} \textbf{ else } \{[\![o]\!]^{SV} \mapsto [\![v]\!]^{SV}\}\})\end{array}$$

For seq-valued properties, we flatten their structures as sets of triples, the middle element of which denotes the index value.

$$\begin{array}{l}\mathcal{M}_{seq} == \\ \quad (\lambda\, s : \mathcal{S}_{tab};\ p : NClass \times NProperty \bullet \\ \quad\quad \bigcup\{o : ObjectId;\ v : Value;\ vs : \mathrm{seq}\, Primitive \mid \\ \quad\quad\quad o \in s.extent\,(fst\,p) \land v = s.value\,(o)\,(snd\,p) \land v = seqValue\,(vs) \bullet \\ \quad\quad\quad\quad \{i : \mathrm{dom}\, vs \bullet [\![o]\!]^{SV} \mapsto [\![i]\!]^{SV} \mapsto [\![vs\ i]\!]^{SV}\}\})\end{array}$$

In the context of a particular $\mathcal{S}_{sql}$, we retrieve from a given table a set of mappings that consist of values drawn from either three of its columns:

$$\begin{array}{l}\mathcal{M}_{3cols} == \\ \quad (\lambda\, s : \mathcal{S}_{sql};\ n : NTable;\ c_1, c_2, c_3 : NColumn \bullet \\ \quad\quad \{row : s.tuples\,(n) \bullet row.values\,(c_1) \mapsto row.values\,(c_2) \mapsto row.values\,(c_3)\})\end{array}$$

On identifying a primitive, single-valued attribute that is stored in some class table, i.e. $p \in classTables\,(fst\,p)$, we insist that

$$\begin{array}{l}\mathcal{M}_{prim}\,(\theta\,\mathcal{S}_{tab},\ p) = \\ \quad \mathcal{M}_{2cols}\,(\theta\,\mathcal{S}_{sql},\ [\![p]\!]^{NTab},\ \texttt{oid},\ [\![p]\!]^{NCol})\end{array} \tag{E.1}$$

On identifying a primitive, sequence-valued attribute, i.e. $p \in seqTables$, we insist that

$$\begin{array}{l}\mathcal{M}_{seq}\,(\theta\,\mathcal{S}_{tab},\ p) = \\ \quad \mathcal{M}_{3cols}\,(\theta\,\mathcal{S}_{sql},\ [\![p]\!]^{NTab},\ \texttt{oid},\ \texttt{index},\ [\![p]\!]^{NCol})\end{array} \tag{E.2}$$

On identifying a single-valued end of a bi-association, i.e. $p \in assocTables \cap (manProperty \cup optProperty)$, we insist that

$$\begin{aligned} &\mathcal{M}_{assoc}\,(\theta\,\mathcal{S}_{tab},\ p) = \\ &\quad \mathcal{M}_{2cols}\,(\theta\,\mathcal{S}_{sql},\ [\![p]\!]^{NTab},\ [\![oppositeOf\ p]\!]^{NCol},\ [\![p]\!]^{NCol}) \end{aligned} \tag{E.3}$$

We represent each null-valued link as an non-existing row, i.e. {}.

On identifying a sequence-valued end of a bi-association, i.e. $p \in assocTables \cap seqProperty$, we insist that

$$\begin{aligned} &\mathcal{M}_{seq}\,(\theta\,\mathcal{S}_{tab},\ p) = \\ &\quad \mathcal{M}_{3cols}\,(\theta\,\mathcal{S}_{sql},\ [\![p]\!]^{NTab},\ [\![oppositeOf\ p]\!]^{NCol},\ \texttt{index},\ [\![p]\!]^{NCol}) \end{aligned} \tag{E.4}$$

When p refers to a primitive, set- (or sequence-valued) attribute, i.e. $p \in setTables$ (or $p \in seqTables$), the relationship is defined in a similar way as the above case of set- (or sequence-valued) association.

$$\begin{aligned} &\mathcal{M}_{set}\,(\theta\,\mathcal{S}_{tab},\ p) = \\ &\quad \mathcal{M}_{2cols}\,(\theta\,\mathcal{S}_{sql},\ [\![p]\!]^{NTab},\ \texttt{oid},\ [\![p]\!]^{NCol}) \end{aligned} \tag{E.5}$$

Since there is no opposite end, we refer to the default `oid` column.

E.3 Structural Model

E.3.1 Booster Model

The system structure of a Booster model and property types will correspond to the concrete syntax as presented in Section A.1 in a straightforward way. The naming convention in our Haskell implementation is to use block capitals for types, e.g.`BPATH`, and use capitalised words for type constructors, e.g. `BPath`. This ensure that it is then straightforward to derive the corresponding data type definitions in Z. `N_SYSTEM` denotes the set of name for the corresponding model element; the same naming convention applies for rest of the current chapter. We use the names `B_CLASS`, `B_ATTRIBUTE`, and `B_TYPE` to be distinguished from `CLASS`, `ATTRIBUTE`, and `TYPE` of an Object model in the next section.

E.3.2 Object Model

An Object model formalises data declarations in its source Booster model, by organising relevant model elements in a hierarchical tree structure. An Object model has a name, a list of declared sets, each with a collection of values, and a list of classes.

```
type VALUE = String
data OBJECT_MODEL
  = ObjectModel { nObjModel ::  N_MODEL
                , nSets     :: [N_SET]  , sets      ::  N_SET    -> [VALUE]
                , nClasses  :: [N_CLASS], classes   ::  N_CLASS  ->  CLASS  }
```

A class consists of a list of properties and operations.

```
data CLASS = Class { nProperties :: [N_PROPERTY]
                   , properties  ::  N_PROPERTY   -> PROPERTY
                   , nOperations :: [N_OPERATION]
                   , operations  ::  N_OPERATION  -> OPERATION }
```

Function `operations` belongs to the behaviour aspect of an Object model, which we will discuss in Section 6.3.2.

A class property consists of its kind, type, and optionally an opposite end of association.

```
data PROPERTY = Property {kind :: KIND, type_ :: TYPE, opposite :: Maybe OPPOSITE}
```

The kind of a property may be an attribute or an association. An attribute is of scalar, primitive type, whereas an association enables us to navigate from the current class to another. A property that is of the association kind may have its opposite defined (i.e. a bi-directional association) or undefined (i.e. a uni-directional association).

```
data KIND  = Attribute | Association
```

The type of an attribute specifies its minimum and maximum multiplicities, its ordering, and the sort of entity set its value is based upon. A multiplicity can be specified either by an integer capacity or as unconstrained. The base set of entities can refer to all instantiated objects of a class, all values of a declared set, all integer values, or all strings. An ordering may be either a generic set or a sequence, with indices and (possible) repetition of elements.

```
data TYPE = Type {min_, max_:: MULTIPLICITY, ordering:: Maybe ORDER, base:: BASE}
data MULTIPLICITY = Multiplicity Integer | Infinity
data BASE = ClassBase N_CLASS | SetBase N_SET | IntBase | StrBase
data ORDER = Set | Seq
```

The opposite of a property, if defined, is composed of an associated class and one of its declared properties that associates back to the current class.

```
data OPPOSITE = Opposite { class_ :: N_CLASS, property :: N_PROPERTY }
```

For example, as shown in Figure 2.1 (on page 35), the `allocations` property of a hotel has the opposite `(Opposite Allocation host)`, and symmetrically, the `host` property of an allocation has the opposite `(Opposite Hotel allocations)`.

E.3.3 Table Model

It is imperative to create an intermediate level of abstraction between the source Object model and the target Sql model. We document in such an intermediate model our implementation strategy, for storing properties/classes into columns/tables, as well as how each navigation of class association should be adapted accordingly to an access to the right table. Neither the Object model nor the Sql model is the appropriate placeholder to incorporate these two pieces of information. As our implementation strategy may vary from time to time, we intend to maintain both Object and Sql models in their most generic forms possible, such that they always remain intact, and so do definitions for the Booster-to-Object compilation and the Sql-to-Script generation.

Moreover, the Object model structure in Section E.3.2 is somehow "distributed", in that we cannot query a model element that resides at the lower level of the tree via a single function application, but a nested one is required. This also motivates our need to create the intermediate `TABLE_MODEL`, whose field functions collectively provide a useful and "centralised" interface. More precisely, these field functions factorise common queries which we will (Section 6.5.5) apply intensively when generating the Sql model. In short, a Table model is essentially an interface for more conveniently querying the Object model.

A Table model has a matching name to its source Object model and consists of three sorts of field functions (see also Appendix E.1.2 for its formal structure). First, *implementation* functions determine which properties are stored in a particular kind of table. These include those that return the collection of properties that are stored in the table that represents their enclosing class (`classTables`), tables for bi-directional associations (`assocTables`), tables for set-value attributes (`setTables`), and tables for

sequence-valued attributes (`seqTables`). The range of each implementation function is a set `IDEN_PROPERTY` of property identifications—each of which is a pair consisting of a class and a property it encloses. For example, we write `(Hotel, allocations)` to identify the property `allocations` in class `Hotel`. Second, *model structure* functions query the type information. These are concerned with the base set (`baseOf`), or the opposite, if any, of a property (`oppositeOf`). We may also query which properties are of primitive values (`intProperty`), object reference values (`refProperty`), or non-null values (`manProperty`), etc. Moreover, for a particular operation we may query the input set (`opInputs`), output set (`opOutputs`), or type of input or output (`opIOType`). Finally, *model behaviour* functions query the behaviour specification. These may give us either the full specification (`opSpec`), predicate guard (`opGuard`), or intent of update (`opIntent`) of a particular operation. Calculations of all these functions are with respect to the source OBJECT model that is supplied for transformation. Field functions `opSpec`, `opGuard`, and `opIntent` belong to the behavioural aspect of a TABLE model and we will discuss them in Section 6.3.3.

E.3.4 SQL Model

We present a SQL model that does not embed any knowledge of the implementation strategy, which we expect to reflect via a separate TABLE model from the previous stage of transformation. Consequently, when a new strategy of implementation is to be adopted, we then modify only the TABLE model, without touching the SQL model and its model-to-text definitions.

A SQL model has a matching name to its source TABLE model and consists of collections of tables, (primary and foreign) key constraints, and database routines.

```
type COL_DECL   = (N_COLUMN, SQL_DATA_TYPE)
type SCHEMA     = (N_TABLE, [COL_DECL])
type FK         = ([N_COLUMN], N_TABLE, [N_COLUMN])
data SQL_MODEL = SqlModel { nSchema  ::  N_SCHEMA
                          , nTables  :: [N_TABLE]
                          , tables   :: [SCHEMA]
                          , primary  :: [(N_TABLE  , [N_COLUMN])]
                          , foreign  :: [(N_TABLE  , [FK]     )]
                          , routines :: [(N_ROUTINE,  ROUTINE )]   }
```

The schema of each table is standard: its name and a list of column-type declaration pairs. The form of each foreign key constraint, with respect to a referencing table, is also standard: its name and list of referencing columns, as well as the name of the referenced table and the list of its referenced columns. The field function `routines` belongs to the behavioural aspect of the SQL model, which we will discuss in Section 6.3.4.

E.4 Structural Model Transformation

There are four kinds of models involved in our pipeline of transformation: we first transform the source BOOSTER model (Section E.3.1) into an OBJECT model (Section E.3.2) that formalise both its system structure and access path expressions. We then transform from this OBJECT model to an intermediate TABLE model (Section E.3.3) that reflects our strategy of implementation, from which we generate a SQL model (Section E.3.4) of implementation. These transformations are straightforward once we are decided on the implementation strategy: 1) each class is implemented as a *class table*, where we store single-valued, mandatory or optional properties only, including primitives and non-collection uni-associations; 2) each bi-directional association is implemented as an *association table*, where we store properties at its both ends; and 3) each primitive or uni-association that is set-valued or sequence-valued is implemented as, respectively, a *set table* or *sequence table*[1]. In the case of an optional attribute, whose value might be `Null`, we would expect the corresponding cell in its class table to contain `NULL` if appropriate; on the other hand, when an optional association has its value as `Null`, we would expect it to be represented as a non-existing row in its association table. Our treatment of transforming equality expressions and assignments in Section 6.5.5 will consistently reflect this strategy of representation. We refer to Appendix E.12.1 for examples of the product of such structural transformations.

E.4.1 BOOSTER to OBJECT Models

To transform the BOOSTER structure (Section E.3.1) to that of the OBJECT model (Section E.3.2), we define a function of the expected types of domain and range.

```
booToObj :: System -> ObjectModel
booToObj sys = ObjectModel { nObjModel = nObjModel' sys
                           , nSets     = nSets'     sys
                           , sets      = sets'      sys
                           , nClasses  = nClasses'  sys
                           , classes   = classes'   sys
                           }
```

Given a valid BOOSTER system `sys` $\in$ `System`, the resulting model `booToObj(sys)` $\in$ `ObjectModel` represents the corresponding OBJECT model of `sys`.

The source BOOSTER system `sys` will be encapsulated once the field functions are initialised; as a result, users wishing to query an OBJECT model need not know anything at all about the underlying BOOSTER system. Such encapsulation occurs at each stage of our transformation pipeline. For example, users interested in querying a SQL model need not (and cannot) access the underlying TABLE, OBJECT, or BOOSTER model from the previous stages of transformation, but can only perform those queries whose domain is SQL model.

Definitions of each "primed" function are straightforward as they correspond to the tree structure of `ObjectModel`, and presumably they can be validated just through

[1]Set-valued or sequence-valued bi-associations are stored in association tables

testing rather than formal proofs. As an example, function *classes′* is defined as follows.

```
classes' :: System -> NClass -> Class
classes' sys c = omClass sys (findBClass sys c)

findBClass :: System -> NClass -> BClass
findBClass sys c = head (filter (matchNBClass c) (bClassList sys))
```

We first find out the BOOSTER class (via `findBClass`) that matches a given class name `c`, then we transform (via `omClass`) the returned BOOSTER class (of type `BClass`) into an OBJECT model class (of type `Class`). Finding a particular BOOSTER class involves filtering out all classes from the system (via `bClassList`) except those (presumably a singleton list) that match its name (via `matchNBClass`) with that of the class `c` in question. Transformation function `omClass` is defined in the same manner as `booToObj`.

```
omClass :: System -> BClass -> Class
omClass sys c = Class { nProperties = nProperties'     c
                      , properties  = properties'      c
                      , nOperations = nOperations'     c
                      , operations  = operations'  sys c
                      }
```

The function `properties'` is defined similarly to initialise all field functions at different nested levels. The more interesting function to investigate is `operations'`

```
nOperations' :: BClass -> NOperation
operations' sys c m = omOperation sys (findMethod c m)

omOperation :: System -> Method -> Operation
omOperation sys   m = Operation { nInputs  = nInputs'  m
                                , inputs   = ioType    m
                                , nOutputs = nOutputs' m
                                , outputs  = ioType    m
                                , spec     = ...
                                }
```

which will retrieve the BOOSTER method in question (via `findMethod`) and transform it (via `omOperation`) into an OBJECT model operation. Field functions `nInputs'` and `nOutputs'` retrieve names of, respectively, inputs and outputs of a given BOOSTER method.

We assume that types of inputs and outputs, if any, must be declared as conjuncts in the guard of the operation in question. For example, in the model of hotel reservation system (Appendix A.2), the method `allocate` declares in its guard (i.e. the left operand of the guard operator `==>`) the input `r?` of class type `Reservation` and the output `a!` of class type `Allocation`. To query the type of a valid input or output (via `ioType`), we look at the guard of its enclosing operation.

The initialisation of field function `spec` would involve the transformation from a BOOSTER method substitution that adopts paths of type `BPath` (Section 6.3.1) into a corresponding OBJECT model substitution that uses paths of type `OPath` (Section 6.3.2). We already discussed this behavioural transformation in detail in Section 6.5.1.

E.4.2 Object to Table Models

The transformation from the Object model structure (Section E.3.2) to that of its corresponding Table model (Section E.3.3) reflects how we decide to store properties/classes into columns/tables (Section E.4 on page 239). At the level of abstraction of the Table model, we do not specify how the (primary and foreign) key constraints are defined. More precisely, whether to define both stored properties in an association table as a composite primary key, or to declare an extra column to identify that association, is left to the later Table-to-Sql transformation (Section 6.5.5). Similarly, how we store the indices of a sequence-valued property is also left to that later transformation.

Similar to the case of Section E.4.1, we define a transformation function between the Object and Table models as follows

```
objToTab :: ObjectModel -> TableModel
objToTab om = TableModel { ... }
```

where the part of ... initialises all field functions (e.g. `assocTables`) of the `TableModel` data type, by encapsulating the source Object model `om`.

Implementation Functions

We define a repository of query functions upon a given source Object model `om`. For example, we may query at the model level about the sets of bi-associations (via `biAssoc om`), set-valued properties (via `setDecls`), uni-associations that are either mandatory (via `mandatoryUniAssoc`) or optional (`optionalUniAssoc`).

The implementation functions `classTables`, `assocTables`, `setTables`, `seqTables` reflect exactly our implementation strategy as specified above. For example,

```
objToTab om = TableModel { assocTables = assocTables' om, ... }

assocTables' :: ObjectModel -> [IdenProperty]
assocTables' om   = foldr (\assoc -> append om assoc) [] (biAssoc om)
```

The definition of the above function `append` ensures for each bi-association, there is only one table that represents it.

```
append :: ObjectModel -> IdenProperty -> [IdenProperty] -> [IdenProperty]
append om p pList | oppositeOf' om p `elem` pList = pList
                  | otherwise                     = p : pList
```

For example, we consider the bi-association between the `hotel` and `room` classes (Figure 2.1 on page 35). In the `hotel` class we declare a property `rooms` that has an mirror attribute `host`; symmetrically, a property `host` in the `room` class has `rooms` as its mirror attribute. The property identification pairs `(Hotel, rooms)` and `(Room, host)` indeed refer to the same bi-association, so it is important for us to ensure that either one of them, but not both, is within the range of `assocTables`.

Property E.4.1 Each bi-association will only be stored in a single table. More precisely,

$$\forall\, om : OBJECT\_MODEL;\ tm : TABLE\_MODEL;\ p, q : IDEN\_PROPERTY \mid \left(\begin{array}{l} tm = objToTab\,(om) \wedge \\ p \neq q \wedge \\ \{p, q\} \subseteq assocTables\,(tm) \wedge \\ \{p, q\} \subseteq biAssoc\,(om) \end{array} \right) \bullet q \neq oppositeOf'\,(om)\,(p)$$

□

Proof of Property E.4.1 Given a context OBJECT model om, a candidate property p that refers to an end of some bi-association, and a list $pList$ of properties that we have so far collected to store as the association tables, $append\,(om)\,(p)\,(pList)$ will check if p's opposite end has already been chosen for storage, in which case p is ignored. Otherwise, p is chosen for storage, which suggests that its opposite end will be ignored in later iterations. □

Model Structure Functions

The model structure functions return type-related information about the underlying OBJECT model. Functions

```
baseOf     :: ObjectModel -> IdenProperty -> Base
oppositeOf :: ObjectModel -> IdenProperty -> IdenProperty
```

infer, respectively, the set of entities upon which a given property is based and the opposite property of, if defined, the bi-association. We may also query the set of properties of a particular type in the OBJECT model. For example, `refProperty` returns all properties that are reference-valued. We support also queries

```
opInputs  :: ObjectModel -> NClass -> NOperation -> [NVariable]
opOutputs :: ObjectModel -> NClass -> NOperation -> [NVariable]
opIOType  :: ObjectModel -> NClass -> NOperation ->  NVariable -> Type
```

on the types of inputs and outputs of operations.

Model Behaviour Functions

The model behaviour functions `opSpec`, `opGuard`, and `opIntent` will return predicates and substitutions that use paths of type `TPath` (Section 6.3.3), which are transformed from those of the underlying OBJECT model that use paths of type `OPath` (Section 6.3.2). We already characterised such transformations in Section 6.5.3.

E.4.3 TABLE to SQL Models

The transformation from a TABLE model structure (Section E.3.3) to that of its corresponding SQL model (Section E.3.4) manifests the primary and foreign (or referential)

key constraints on tables that we identified in the TABLE model (Section E.4.2). As before, we define a transformation function with the appropriate types of domain and range.

```
tabToSql :: TableModel -> SqlModel
tabToSql tm = SqlModel { nSchema  = nTabModel tm
                       , nTables  = nTables'  tm, tables   = tables'  tm
                       , primary  = primary'  tm, foreign  = foreign' tm
                       , routines = ...
                       }
```

The initialisation of the field function `routines` will involve the transformation of operations of the TABLE model (Section 6.3.3) to those of the SQL model (Section 6.3.4). We already address this transformation in detail in Section 6.5.5.

Naming of Tables

The field function

```
nTables :: TableModel -> [NTable]
nTables tm = genNAssocTables tm ++ ...
```

returns a set of string names to be declared for all sorts of tables. For example, each property identification `(c, p)` $\in$ `assocTables tm`, for a given TABLE model `tm`, will trigger the creation of an association table with its name incorporating both association ends.

```
genNAssocTables :: TableModel -> [NTable]
genNAssocTables tm = map (nAssocTable tm) (assocTables tm)

nAssocTable :: TableModel -> IdenProperty -> NTable
nAssocTable tm p =  nPropertyIden p ++ "_" ++ nPropertyIden (oppositeOf tm p)
```

For example, a table named `Hotel_rooms_Room_host` is created for the corresponding bi-association.

Structures of Tables

The field function `tables` returns a set of schema structures to be declared for the named tables.

```
tables :: TableModel -> [Schema]
tables tm =  classTableSchemas  tm ++ assocTableSchemas tm
          ++ setTableSchemas    tm ++ seqTableSchemas   tm
```

Each sort of table contains a default column that represents the identifier of the properties in question. For example, in the case of class tables, the `oid` column identifies a unique object, while in the case of association tables, it identifies a unique pair of objects.

We define an auxiliary function `columnStruct`. Given a context TABLE model `tm` and the identification of a property `iden` that is to be stored, `columnStruct tm iden` returns a list of column declarations that implement `iden`.

```
columnStruct :: TableModel -> IdenProperty -> [ColDecl]
columnStruct tm iden@(_, p)
  = [(p, columnDecl)]
    where columnDecl
            | iden `elem` intProperty tm = SqlInteger
            | iden `elem` strProperty tm = SqlString
            | iden `elem` refProperty tm = case baseOf tm iden of
                                             ClassBase "Date" -> SqlDate
                                             otherwise        -> SqlInteger
```

Both integer properties and reference properties will be stored as SQL integers, and SQL strings for string properties. A property of type `Date` is implemented directly by the `Date` data type in SQL. Another auxiliary function `columnStructs` collects outputs by applying *columnStruct* upon a list of identified properties.

```
columnStructs :: TableModel -> [IdenProperty] -> [(NColumn, SqlDataType)]
columnStructs tm ps = foldr ((++).columnStruct tm) [] ps
```

For each class table `c`, we declare a default integer column `oid` to represent references of all instantiated objects of that class, and we declare this column as the primary key. We also declare columns for all properties that are to be stored in that class table, which are determined by the query `classTables tm c`.

```
classTableSchemas :: TableModel -> [Schema]
classTableSchemas tm =
  foldr ((:).(\c -> (c,
                     oidColumn : (columnStructs tm (classTables tm c)))))
        []
        (nClassTables tm)
```

Example (*Schema of class table*) The class `Hotel` (Appendix A.2) is implemented by

```
CREATE TABLE `Hotel`(`oid` INTEGER AUTO_INCREMENT,
                     `name` CHAR(30),
                     PRIMARY KEY (`oid`));
```

where `name` is its only property that is not bi-directional, set-valued, or sequence-valued. □

For each association table, we declare a default integer column `oid` to represent references of all instances of that association, and we use this column as the primary key. We also declare columns to represent properties at both ends of the association. The rationale for not choosing the alternative strategy—we could have declared a composite primary key upon properties at both ends of the bi-association—is to take into account that we may later decide to make the current bi-association an association class, in which case its instance should be consistently identified by an `oid` column as we do for class tables.

```
assocTableSchemas :: TableModel -> [Schema]
assocTableSchemas tm =
  foldr ((:).(\p -> (nAssocTable tm p,
                     let p' = oppositeOf tm p
                     in (oidColumn : columnStructs tm [p, p']) ++
                           genIndexColumns tm p p')))
  []
  (assocTables tm)
```

In the case where none of the association ends is sequence-valued, the `genIndexColumns` above returns an empty list.

Example (*Schema of association table*) The bi-association between property `registered` of class `Hotel` and property `reglist` of class `Traveller` (Figure 2.1) is implemented by

```
CREATE TABLE ‘Traveller_reglist_Hotel_registered‘(‘oid‘ INTEGER AUTO_INCREMENT,
                                                  ‘reglist‘ INTEGER,
                                                  ‘registered‘ INTEGER,
                                                  PRIMARY KEY (‘oid‘));
```

□

In cases where one or both of the association ends are sequence-valued, we need to declare accordingly an index column or columns.

```
genIndexColumns :: TableModel -> IdenProperty -> IdenProperty -> [ColDecl]
genIndexColumns tm p p' | p  ‘elem‘ seqProperty tm &&
                          p' ‘elem‘ seqProperty tm       -- seq-to-seq bi-association
                        = [(indexColumn p , SqlInteger),
                           (indexColumn p', SqlInteger)]
                        | p  ‘elem‘ seqProperty tm ||
                          p' ‘elem‘ seqProperty tm       -- seq-to-_   bi-association
                        = [(index, SqlInteger)]
                        | otherwise                      -- _-to-_     bi-association
                        = []
```

Example (*Schema of association table for sequence-valued property*) The bi-association between property `employees` of class `Hotel` and property `employers` of class `Staff` (Figure 2.1) is implemented by

```
CREATE TABLE ‘Staff_employers_Hotel_employees‘(‘oid‘ INTEGER AUTO_INCREMENT,
                                               ‘employers‘ INTEGER,
                                               ‘employees‘ INTEGER,
                                               ‘employers_index‘ INTEGER,
                                               ‘employees_index‘ INTEGER,
                                               PRIMARY KEY (‘oid‘));
```

□

For each set table that implements an identified set-valued property `(c, p)` $\in$ `setTables tm`, we declare an integer column `oid` to represent objects (presumably instantiated from class `c`) that possess `p`, and a column that represents values of `p`.

```
setTableSchemas :: TableModel -> [Schema]
setTableSchemas tm =
  foldr ((:).(\p -> (nSetTable p,
                     oidColumn : columnStructs tm [p])))
  []
  (setTables tm)
```

For each sequence table, similarly, we stored both an integer column `oid` and a column that represents values of the property in question. In addition, we store an integer column `index` that represents indices of that property.

Foreign Key Constraints

The field function `foreign` returns a collection of foreign/referential key constraints on tables.

```
foreign :: TableModel -> [(NTable, [FK])]
foreign tm =  classTableFK tm ++ assocTableFK tm
           ++ setTableFK   tm ++ seqTableFK   tm
```

We define an auxiliary function

```
objRefFK :: TableModel -> IdenProperty -> [FK]
```

such that given a context TABLE model `tm` $\in$ `TABLE_MODEL` and an identified property `iden`, the transformation of `objRefFK tm iden` returns the set of foreign constraints upon the table that implements `iden`. For any identified reference-valued property `(c, p)` $\in$ `refProperty tm`, we figure out the associated class of `p`, say `c'`, then we derive `([p], c', [oid])` as a foreign key constraint. There is no foreign key constraint for properties of other types, including any data type that is directly implemented as a SQL primitive type, e.g. `Date`.

As an example, for each association table there are exactly two foreign key constraints: one on the identified property that is in the domain of `assocTables`, and the other on its opposite (via query of `oppositeOf`).

```
assocTableFK :: TableModel -> [(NTable, [FK])]
assocTableFK tm =
  foldr ((:).(\p -> (nAssocTable tm p,
                     objRefFK tm p ++ objRefFK tm (oppositeOf tm p))))
  []
  (assocTables tm)
```

Example (*Foreign key on association table*) We impose a foreign key constraint on the association table in Example E.4.3:

```
ALTER TABLE ‘Traveller_reglist_Hotel_registered‘
  ADD FOREIGN KEY (‘reglist‘) REFERENCES ‘Hotel‘ (‘oid‘);
ALTER TABLE ‘Traveller_reglist_Hotel_registered‘
  ADD FOREIGN KEY (‘registered‘) REFERENCES ‘Traveller‘ (‘oid‘);
```

□

For each class table we select all stored properties that are reference-valued, each of which is transformed into a foreign key constraint. The foreign key constraints on set and sequence tables are similar. There are at most two foreign keys declared: one on the `oid` column that represents objects possessing the identified set-valued (or respectively, sequence-valued) property, and the other on the column that represents values of that set-valued property, if it is also reference-valued.

E.5 Programming Data Types for SQL Statements

We define the `Statement` data type to model queries that are used to implement programs of substitutions.

```
data Statement
  = INSERT          SQL_EXPR   [N_COLUMN]              [SQL_EXPR]
  | UPDATE          SQL_EXPR   [(N_COLUMN, SQL_EXPR)]  SQL_EXPR
  | DELETE          SQL_EXPR    SQL_EXPR

  -- local variables
  | DECLARE         N_VARIABLE  SQL_DATA_TYPE
  | SELECT_INTO     SQL_EXPR    N_VARIABLE SQL_EXPR SQL_EXPR
  | SET             N_VARIABLE  SQL_EXPR
  | TEMP_TABLE      N_VARIABLE  SQL_EXPR
  | DROP_TEMP_TABLE N_VARIABLE
  | RETURN          SQL_EXPR

  -- statement combinators
  | IfThenElse      SQL_EXPR   [STATEMENT] [STATEMENT]
  | WHILE           SQL_EXPR   [STATEMENT]

  -- cursor operations
  | DECLARE_CUR     N_VARIABLE  SQL_EXPR
  | OPEN            N_VARIABLE
  | CLOSE           N_VARIABLE
  | FETCH           N_VARIABLE  N_VARIABLE
```

We define the `SQL_EXPR` data type to model queries that are used to implement expressions that occur in programs of substitutions.

```
data SQL_EXPR
  = SELECT          [SQL_EXPR] SQL_EXPR SQL_EXPR          | NULL
  | SELECT_AS_TAB   [SQL_EXPR] SQL_EXPR SQL_EXPR N_TABLE | STRING String
  | SELECT_AS_ROW   [(SQL_EXPR, N_COLUMN)]               | NUMBER Integer
  | SQL_EXPR `AS`    N_COLUMN                           |          NEG SQL_EXPR
  | COUNT SQL_EXPR                                      | SQL_EXPR :+: SQL_EXPR
                                                        | SQL_EXPR :-: SQL_EXPR
  | VAR   N_VARIABLE                                    | SQL_EXPR :*: SQL_EXPR
  | TABLE N_TABLE                                       | SQL_EXPR :/: SQL_EXPR
  | FUNC  N_FUNCTION [SQL_EXPR]
  | AllCols                                            | SQL_EXPR :=:   SQL_EXPR
  | SQL_EXPR `UNION`     SQL_EXPR                      | SQL_EXPR :<>:  SQL_EXPR
  | SQL_EXPR `INTERSECT` SQL_EXPR                      | SQL_EXPR :>:   SQL_EXPR
  | SQL_EXPR `EXCEPT`    SQL_EXPR                      | SQL_EXPR :<:   SQL_EXPR
  | TRUE                                               | SQL_EXPR :>=:  SQL_EXPR
  | FALSE                                              | SQL_EXPR :<=:  SQL_EXPR
  |           NOT  SQL_EXPR                            | SQL_EXPR `IN`  SQL_EXPR
  | SQL_EXPR `AND` SQL_EXPR                            | SQL_EXPR `NIN` SQL_EXPR
  | SQL_EXPR `OR`  SQL_EXPR
```

E.6 Transformation Patterns of Basic Assignments

Corresponding to the possible ways of declaring bi-directional associations as we demonstrated in Section 2.5.1, our transformation on assignments must take account of possible ways of updating these associations. The following four tables summarise the patterns of basic assignments that our Haskell transformation handles.

- Table E.1 lists assignment patterns on single-valued bi-associations.
- Table E.2 lists assignment patterns on set-valued associations.
- Table E.3 lists assignment patterns on sequence-valued associations.
- Table E.4 lists assignment patterns on primitive attributes

Each table is self-contained: the columns from left to right specify declarations of properties, numerical identifiers of patterns, their abstract implementation in the substitution program, and their concrete implementation in database queries. The pattern numbers in these tables also correspond to the numbers that are documented as comments in the complete of our hotel reservation system in BOOSTER as in Appendix A.2.

Table E.1: Specifying & Implementing Booster Bi-Associations: *Single-Valued Ends*

Bi-Association	#	Booster Predicate	GSL Substitution	Sql Queries

1-*to*-1
`b: B.a`
`a: A.b`
	1	$b' = that \wedge$
		$that.a' = this$
			$\underline{b := that} \parallel$
			$that.a := this$
				`DELETE FROM` t
				`WHERE` $a = this$ `OR` $b = that$`;`
				`INSERT INTO` t (b, a) `VALUE` $(that, this)$`;`

1-*to*-opt.
`b: B.ao`
`a: [A.b]`
	2	$b' = that \wedge$
		$that.ao' = this$
			$\underline{b := that} \parallel$
			$that.ao := this$
				`DELETE FROM` t `WHERE` $ao = this$`;`
				`INSERT INTO` t (b, ao) `VALUE` $(that, this)$`;`

1-*to*-set
`b: B.as`
`as: set(A.b)`
	3	$b' = that \wedge$
		$this \in that.as'$
			$\underline{b := that} \parallel$
			$that.as := that.as \cup \{this\}$
				`DELETE FROM` t `WHERE` $as = this$`;`
				`INSERT INTO` t (b, as) `VALUE` $(that, this)$`;`

1-*to*-seq
`b: B.as`
`as: seq(A.b)`
	4	$b' = that \wedge$
		$that.as' = \text{ins}\,(that.as, i, this)$
			$\underline{b := that} \parallel$
			$that.as := \text{ins}\,(that.as, i, this)$
				No updates triggered—see pattern 21.

opt.-*to*-1
`bo: [B.a]`
`a: A.bo`
	5	$bo' = that \wedge$
		$that.a' = this$
			$\underline{bo := that} \parallel$
			$that.a := this$
				No updates triggered—see pattern 2.

	6	$bo' = null$	$bo := null$	`DELETE FROM` t `WHERE` $a = this$`;`

opt.-*to*-opt
`bo: [B.ao]`
`ao: [A.bo]`
	7	$bo' = that \wedge$
		$that.ao' = this$
			$\underline{bo := that} \parallel$
			$that.ao := this$
				`INSERT INTO` t (bo, ao) `VALUE` $(that, this)$`;`

	8	$bo' = null \wedge$ $that.ao' = null$	$\underline{bo := null} \parallel$ $that.ao := null$	`DELETE FROM` t `WHERE` $ao = this$`;`

opt.-*to*-set
`bo: [B.as]`
`as: set(A.bo)`
	9	$bo' = that \wedge$
		$this \in that.as'$
			$\underline{bo := that} \parallel$
			$that.as := that.as \cup \{this\}$
				`INSERT INTO` t (bo, as) `VALUE` $(that, this)$`;`

	10	$bo' = null \wedge$ $this \notin that.as'$	$\underline{bo := null} \parallel$ $that.as := that.as \setminus \{this\}$	`DELETE FROM` t `WHERE` $as = this$`;`

opt.-*to*-seq
`bo: [B.as]`
`as: seq(A.bo)`
	11	$bo' = that \wedge$
		$that.as' = \text{ins}\,(that.as, i, this)$
			$\underline{bo := that} \parallel$
			$that.as := \text{ins}\,(that.as, i, this)$
				No updates triggered—see pattern 23.

	12	$bo' = null \wedge$ $that.as' = \text{del}\,(that.as, i)$	$\underline{bo := null} \parallel$ $that.as := \text{del}\,(that.as, i)$	*No updates triggered—see pattern 24.*

Table E.2: Specifying & Implementing Booster Bi-Associations: *Set-Valued Ends*

Bi-Association	#	Booster Predicate	GSL Substitution	Sql Queries

set-*to*-1
`bs: set(B.a)`
`a: A.bs`
	13	$that \in bs' \wedge$
		$that.a' = this$
			$\underline{bs := bs \cup \{that\}} \parallel$
			$that.a := this$
				No updates triggered—see pattern 3.

	14	$that \notin bs'$	$bs := bs \setminus \{that\}$	`DELETE FROM` t `WHERE` $bs = that$`;`

set-*to*-opt
`bs: set(B.ao)`
`ao: [A.bs]`
	15	$that \in bs' \wedge$
		$that.ao' = this$
			$\underline{bs := bs \cup \{that\}} \parallel$
			$that.ao := this$
				No updates triggered—see pattern 9.

	16	$that \notin bs' \wedge$ $that.ao' = null$	$\underline{bs := bs \setminus \{that\}} \parallel$ $that.ao := null$	*No updates triggered—see pattern 10.*

set-*to*-set
`bs: set(B.as)`
`as: set(A.bs)`
	17	$that \in bs' \wedge$
		$this \in that.as'$
			$\underline{bs := bs \cup \{that\}} \parallel$
			$that.as := that.as \cup \{this\}$
				`INSERT INTO` t (bs, as) `VALUE` $(that, this)$`;`

	18	$that \notin bs' \wedge$ $this \notin that.as'$	$\underline{bs := bs \setminus \{that\}} \parallel$ $that.as := that.as \setminus \{this\}$	`DELETE FROM` t `WHERE` $as = this$ `AND` $bs = that$`;`

set-*to*-seq
`bs: set(B.as)`
`as: seq(A.bs)`
	19	$that \in bs' \wedge$
		$that.as' = \text{ins}\,(that.as, i, this)$
			$\underline{bs := bs \cup \{that\}} \parallel$
			$that.as := \text{ins}\,(that.as, i, this)$
				No updates triggered—see pattern 25.

	20	$that \notin bs' \wedge$ $that.as' = \text{del}\,(that.as, i)$	$\underline{bs := bs \setminus \{that\}} \parallel$ $that.as := \text{del}\,(that.as, i)$	*No updates triggered—see pattern 26.*

Table E.3: Specifying & Implementing BOOSTER Bi-Associations: *Seq-Valued Ends*

Bi-Association	#	Booster Predicate	GSL Substitution	SQL Queries

seq-*to*-1
`bs: seq(B.a)`
`a: A.bs`
	21	$bs' = \text{ins}\,(bs, i, that) \wedge$
		$that.a' = this$
			$\underline{bs := \text{ins}\,(bs, i, that)} \parallel$
			$that.a := this$
				`UPDATE` t `SET` $index = index - 1$
				`WHERE` $a = ($`SELECT` a
				`FROM` t `WHERE` $bs = that)$
				`AND` $index > i$`;`
				`DELETE FROM` t `WHERE` $bs = that$`;`
				`UPDATE` t `SET` $index = index + 1$
				`WHERE` $a = this$ `AND` $index \geq i$`;`
				`INSERT INTO` t $(bs, a, index)$ `VALUE` $(that, this, i)$`;`

	22	$bs' = \text{del}\,(bs, i)$	$bs := \text{del}\,(bs, i)$	`DELETE FROM` t `WHERE` $a = this$ `AND` $index = i$`;`
				`UPDATE` t `SET` $index = index - 1$
				`WHERE` $a = this$ `AND` $index > i$`;`

seq-*to*-opt
`bs: seq(B.ao)`
`ao: [A.bs]`
	23	$bs' = \text{ins}\,(bs, i, that) \wedge$
		$that.ao' = this$
			$\underline{bs := \text{ins}\,(bs, i, that)} \parallel$
			$that.ao := this$
				`UPDATE` t `SET` $index = index + 1$
				`WHERE` $ao = this$ `AND` $index \geq i$`;`
				`INSERT INTO` t $(bs, ao, index)$ `VALUE` $(that, this, i)$`;`

	24	$bs' = \text{del}\,(bs, i) \wedge$
		$that.ao' = null$
			$\underline{bs := \text{del}\,(bs, i)} \parallel$
			$that.ao := null$
				`DELETE FROM` t `WHERE` $ao = this$ `AND` $index = i$`;`
				`UPDATE` t `SET` $index = index - 1$
				`WHERE` $ao = this$ `AND` $index > i$`;`

seq-*to*-set
`bs: seq(B.as)`
`as: set(A.bs)`
	25	$bs' = \text{ins}\,(bs, i, that) \wedge$
		$this \in that.as'$
			$\underline{bs := \text{ins}\,(bs, i, that)} \parallel$
			$that.as := that.as \cup \{this\}$
				`UPDATE` t `SET` $index = index + 1$
				`WHERE` $as = this$ `AND` $index \geq i$`;`
				`INSERT INTO` t $(bs, as, index)$ `VALUE` $(that, this, i)$`;`

	26	$bs' = \text{del}\,(bs, i) \wedge$
		$this \notin that.as'$
			$\underline{bs := \text{del}\,(bs, i)} \parallel$
			$that.as := that.as \setminus \{this\}$
				`DELETE FROM` t `WHERE` $as = this$ `AND` $index = i$`;`
				`UPDATE` t `SET` $index = index - 1$
				`WHERE` $as = this$ `AND` $index > i$`;`

seq-*to*-seq
`bs: seq(B.as)`
`as: seq(A.bs)`
	27	$bs' =$
		$\text{ins}\,(bs, i, that) \wedge$
		$that.as' =$
		$\text{ins}\,(that.as, j, this)$
			$\underline{bs :=}$
			$\underline{\text{ins}\,(bs, i, that)}$
			$\parallel$
			$\underline{that.as :=}$
			$\underline{\text{ins}\,(that.as, j, this)}$
				`UPDATE` t `SET` $index_{bs} = index_{bs} + 1$
				`WHERE` $as = this$ `AND` $index_{bs} \geq i$`;`
				`UPDATE` t `SET` $index_{as} = index_{as} + 1$
				`WHERE` $bs = that$ `AND` $index_{as} \geq j$`;`
				`INSERT INTO` t $(bs, index_{bs}, as, index_{as})$
				`VALUE` $(that, i, this, j)$`;`

	28	$bs' =$
		$\text{del}\,(bs, i) \wedge$
		$that.as' =$
		$\text{del}\,(that.as, j)$
			$\underline{bs :=}$
			$\underline{\text{del}\,(bs, i)}$
			$\parallel$
			$\underline{that.as :=}$
			$\underline{\text{del}\,(that.as, j)}$
				`DELETE FROM` t `WHERE` $(as = this$ `AND` $index_{bs} = i)$ `AND`
				$(bs = that$ `AND` $index_{as} = j)$`;`
				`UPDATE` t `SET` $index_{bs} = index_{bs} - 1$
				`WHERE` $as = this$ `AND` $index_{bs} > i$`;`
				`UPDATE` t `SET` $index_{as} = index_{as} - 1$
				`WHERE` $bs = that$ `AND` $index_{as} > j$`;`

Table E.4: Specifying & Implementing BOOSTER Attributes: *Primitive*

Attribute	#	Booster Predicate	GSL Substitution	SQL Queries
Mandatory `b: B`	29	$b' = that$	$b := that$	`UPDATE` t `SET` $b = that$ `WHERE` $oid = this$`;`
Optional `b: [B]`	30	$b' = that$	$b := that$	`UPDATE` t `SET` $b = that$ `WHERE` $oid = this$`;`
	31	$b' = null$	$b := null$	`UPDATE` t `SET` $b = null$ `WHERE` $oid = this$`;`

Set-Valued
`b: set(B)`
	32	$\{e_1, \ldots e_n\} \subseteq b'$	$b := b \cup \{e_1, \ldots e_n\}$	`INSERT INTO` t (oid, b) `VALUE` $(this, e_1)$`;`
				$\ldots$
				`INSERT INTO` t (oid, b) `VALUE` $(this, e_n)$`;`

	33	$\{e_1, \ldots e_n\} \not\subseteq b'$	$b := b \setminus \{e_1, \ldots e_n\}$	`DELETE FROM` t `WHERE` $oid = this$ `AND` $b = e_1$`;`
				$\ldots$
				`DELETE FROM` t `WHERE` $oid = this$ `AND` $b = e_n$`;`

	34	$es \subseteq b'$	$b := b \cup es$	$\{ x \in es \bullet$ `INSERT INTO` t (oid, b) `VALUE` $(this, x)$`; }`
	35	$es \not\subseteq b'$	$b := b \setminus es$	$\{ x \in es \bullet$ `DELETE FROM` t `WHERE` $oid = this$ `AND` $b = x$`; }`
Seq-Valued `b: seq(B)`	36	$b[i]' = that$	$b[i] := that$	`UPDATE` t `SET` $b = that$ `WHERE` $oid = this$ `AND` $index = i$`;`

E.7 A Caching Mechanism for Paths

E.7.1 Motivation

As the requirements evolve, the class association graph of our hotel reservation system (Figure 2.1 on page 35) will grow in its complexity either when we add more nodes through declaring new classes, or when we add more vertices through declaring new associations between existing classes. This would, unavoidably, lead us to object paths involving a long list of class navigations. We may, for example, specify the following intent on our hotel reservation system (Figure 2.1) to update the accounts of all registered travellers of a hotel, being the host of the set of all reservations of a given room `r?`:

```
ALL res : r?.reservations @ (
  ALL t : res.host.registered @ (t.account := ...))
```

where four edges in the association graph are involved to navigate between classes *Room*, *Reservation*, *Hotel*, *Traveller*, and *Account*. When the number of instances of each association is substantial, then it would not be acceptable—as far as the performance is concerned—for the generated queries simply to access each traveller right from the input room; this causes similar look-ups to be repeatedly performed on large association tables.

More precisely, we want to avoid the naive approach in which we implement each given object path $this.p_1.p_2.\ldots.p_{n-1}.p_n$ as the following nested SQL `SELECT` query (assuming that property p_i is stored in table T_i, $1 \leq i \leq n$):

```
SELECT p_n
FROM T_n
WHERE oid = (SELECT p_{n-1}
             FROM T_{n-1}
             WHERE oid = ... (SELECT p_2
                              FROM T_2
                              WHERE oid = (SELECT p_1
                                           FROM T_1
                                           WHERE oid = This)) ... )
```

Starting from the n_{th} nested level that locates the row with its `oid` column matching the starting reference `This`, each lookup in table T_i ($2 \leq i \leq n$) uses the lookup result from a deeper nested level (i.e. in table T_{i-1}). We observe that paths in a BOOSTER method which share a common prefix will cause their transformed SQL queries to share a common subquery. There is certainly a scope for improvement on the performance: instead of continually evaluating the same common subquery, we *cache* its value and let all "relevant" paths reuse its value by referencing it. As we learnt from Scenario 1.1.4 (on page 11), we may not just assume that there will be satisfactory support on optimising queries which implement long paths. Without satisfactory support on indexing or caching, even moderately-sized data sets could—depending on the implementation—cause performance problems.

E.7.2 Details

We will first present three critical query functions, upon which our transformation will rely, that—given a table path of type `TPATH`—collectively define an appropriate representation of it in the domain of SQL models: target table, column, and rows.

Target Tables, Columns, and Rows: given a table path (of type `TPATH` in Section 6.3.3), we can calculate its stored location in the generated database, characterised by the intersection of a table, a column, and a row[2]. Thanks to the embedded information in type `TAccess` for table access (section 6.3.3), the calculation is straightforward. We will define three functions:

```
targetTable  :: TABLE_MODEL -> TPATH -> SQL_EXPR
targetColumn ::                 TPATH -> N_COLUMN
targetRow    :: TABLE_MODEL -> TPATH -> SQL_EXPR
```

Given a table path `tp` ∈ `TPATH` and its context model `tm` ∈ `TABLE_MODEL`, for the entity that is referred to by `tp`, we write `targetTable tm tp` to denote the name of its storing table, `targetColumn tp` the column it is stored under, and `targetRow tm tp` the condition that identifies the specific row(s) corresponding to its runtime value.

Example E.7.1 We assume `tm` ∈ `TABLE_MODEL` is the table model in context. We consider the first part of the recursive table path from Example E.11.3, which is also a recursive table path (which we call it `tp` here):

```
RecTPath (BaseTPath (ThisRef (ClassBase Account)))
         (AssocTAccess (Account, owner))
```

To determine the stored table for `(Account, owner)`, one of the checks is to see if it is a bi-association, in which case it is stored in an association table. By Property E.4.1, we know that it may be the case that `(Account, owner)` ∉ `(assocTables tm)`, as it was simply its opposite `(Traveller, account)` that we adopted to name the corresponding association table. In either case, we would expect `(targetTable tm tp)` to return a SQL expression that denotes the table name `Account_owner_Traveller_account`. Determining the target column in the stored table for `(Account, owner)` is trivial, as we name the column according to the property—`(targetColumn (Account, owner))` ought to return `owner`. Since owner is a single-valued property, it corresponds to just one row in its stored table, we shall expect to find the corresponding row via the reference of its current object, of type `Account`. As a result, the condition that `(targetRow tm tp)` returns should be `account = this` (i.e. find the `owner` whose `account` is `this`). On the other hand, say we are now in the context of the `Traveller` class, the call to `targetRow` on a path that intends to retrieve the value of `reglist` (a set-valued property) would return `registered = this`, but this effectively denotes all hotels whose registered clients include `this` traveller. □

Caching Values of Paths: we have constructed each object path (`OPATH` in Section 6.3.2) or table path (`TPATH` in Section 6.3.3) as a nested tree to represent the source BOOSTER path sequence (`BPATH` in Section 6.3.1). Thanks to the fact that each table path embeds the information about which table columns to access, it is straightforward to transform it into its corresponding (nested) SQL `SELECT` query. We will briefly present a caching mechanism, including how we collect paths to be cached, how we declare the collected such paths as cache variables, and how we update the values of these variables in due course. We

[2]Or rows if the target property is set-valued.

encourage the readers to first consider illustrations of this caching mechanism as presented in Sections E.12.2 and E.12.3.

Collecting Paths to be Cached: given a table path `tp` $\in$ `TPATH` (Section 6.3.3), we write `prefixesInTPath tp` to infer a set of table paths as its prefixes (or subtrees). The shortest prefix is the starting reference of the table path in question, and the longest the table path itself. Access to an indexed component that acts as a starting reference of a path (e.g. `sc[0]`, `sci?[0]`) will be cached as, respectively, variables `‘sc[0]‘` and `‘sci?[0]‘`. Other base table paths such as `ThisRef` and `InputRef` should have already been declared as the procedure inputs. Dummy variables (`VariableRef`) that appear in quantifiers (`Forall`, `Exists`) and iterators (`ALL`, `ANY`) will be declared as local variables and used in combination with cursors.

Our goal is to cache path expressions that occur in either program substitutions or guard predicates. For example, we would expect `(cacheVarsInSubs o)` to return a sorted sequence of prefix paths to be cached, and its order suggests the order in which we (re-)evaluate and cache their values.

Example E.7.2 The table path as we have seen from Example E.11.3 has the following set of three prefix paths:

$$\left\{\begin{array}{r} \texttt{BaseTPath}\,(\mathit{ThisRef}\;(\mathit{ClassBase}\;\mathit{Account})), \\ \\ (\texttt{RecTPath}\,(\texttt{BaseTPath}\,(\mathit{ThisRef}\;(\mathit{ClassBase}\;\mathit{Account})))\\ \qquad (\texttt{AssocTAccess}\,(\mathit{Account},\;\mathit{owner}))), \\ \\ \texttt{RecTPath}\,(\texttt{RecTPath}\,(\texttt{BaseTPath}\,(\mathit{ThisRef}\;(\mathit{ClassBase}\;\mathit{Account})))\\ \qquad (\texttt{AssocTAccess}\,(\mathit{Account},\;\mathit{owner})))\\ \quad (\texttt{SetTAccess}\,(\mathit{Traveller},\;\mathit{reglist})) \end{array}\right\}$$

And this corresponds to the set of BOOSTER path sequences `this`, `this.owner`, as well as `this.owner.reglist`. The value of the starting reference `this` should have already been declared as one of the procedure arguments, the value of which is reused to evaluate the path `this.owner` and cache its value in variable `‘this.owner‘` accordingly, the value of which is reused to evaluate the path `this.owner.reglist` and cache its value in variable `‘this.owner.reglist‘` for later references. □

Declaring Cache Variables of Paths: This leads to our simple strategy of declaring cache variables, as implemented in function `declareCacheVar`: in the beginning of each model operation, we declare all paths to be used across its phases, if any. For each table path that is a prefix of another, we declare it as a cache variable. In cases where the property to be accessed is set- or sequence-valued, we create temporary tables to capture their values.

Updating Cache Variables of Paths: throughout each operation, where necessary, we update, via `updateCache`, the cache variables by re-evaluating its values. We rely on the three functions as defined above 1) to calculate the target table, column, and row; and 2) to re-evaluate the corresponding table path accordingly. Provided that the collection of paths, as well as the declaration and update of cache variables are done properly, the valuable outcome is that within each scope in the procedure where the re-evaluation of a path is not necessary, we can simply refer to its cache variable. For instance, in Example E.11.3, we would expect two cache variables `‘this.owner‘` and `‘this.owner.reglist‘`

to be referenced. The trivial definition of our transformation from a path `tp` ∈ `TPATH` to its corresponding expression reflects exactly this:

```
toSqlExpr' tm (Path (TPath btp@(BaseTPath _    ))) = VAR (nCacheVar btp)
toSqlExpr' tm (Path (TPath rtp@(RecTPath  tp ta))) =
  case ta of
    SetTAccess   _   -> multiVar
    SeqTAccess   _   -> multiVar
    AssocTAccess p
      | p `elem` (setProperty tm) -> multiVar
      | p `elem` (seqProperty tm) -> multiVar
      | otherwise                 -> var
    ClassTAccess _   -> var
    SeqTCAccess  _ _ -> var
  where var       = VAR (nCacheVar rtp)
        col       = targetColumn rtp
        multiVar  = SELECT [VAR col] var TRUE
```

We transform both base and recursive table paths to a reference to their corresponding cache variable (via `nCacheVar`). But there is a subtlety here, if the table path denotes a set- or sequence-valued property, we transform it to a `SELECT` query that retrieves the value from its cache variable.

Lemma E.7.1 (***Correctness:* TABLE-*to*-SQL *Path Transformation***) Each occurrence of `updateCache` correctly assigns, from their corresponding TABLE paths, values to cache variables. □

Property E.7.1 (***Consistency of Caching***) For each reference to a variable `v` that caches the value of table path `tp`, it is the case that retrieving the value of `v` is as if we re-evaluated `tp`. □

E.8 Implementing Iterators

Iterators and quantifiers will be implemented as `WHILE` loops that iterate, via cursors, through their set-valued ranges. For the purpose of correctness, our transformation adopts a specific pattern of loop (`iterate`) that is explicitly bounded (i.e with a loop variant). An instantiation `iterate tm x tp head body` represents a loop that iterates for exactly the number of times that corresponds to the size of the set-valued range, referred to by the table path `tp`; statements `head` and `body` are those to be executed, respectively, right before the loop starts and inside the loop.

A use case of the `iterate` pattern is our treatment of quantifiers (`Forall`, `Exists`) and iterators (`ALL`, `ANY`). We encourage the reader to first consider an example of this in Appendix E.12.3. We declare a cache variable for each expression of universal or existential quantification, as there is no direct support of them in SQL. In the beginning of each stored function or procedure, we declare cache variables to represent all its quantifier expressions. Updates (via `updateQCacheVar`) to variables that cache the universal and existential expressions both instantiate the `iterate` pattern, differing only in how they accumulate (i.e. conjoin versus disjoin) the ultimate Boolean result.

Property E.8.1 (***Termination of Iterators & Quantifiers***) Our transformation on iterators via `toSqlProc` and on quantifiers via `updateQCacheVar` generate loop implementations that are guaranteed to terminate. □

This means that our relational semantics for `WHILE` in Section 6.4.3 is guaranteed to unwrap in a bounded number of times, and hence it is possible to show that its behaviour is consistent with the corresponding iterators.

E.9 Proofs

Proof of Lemma 6.5.1 We assume that the auxiliary function `target` correctly identifies the property, given a context path, and the function `dVarType` correctly derives types of bound variables by properly maintaining the environment. We prove by a structural induction on the structure of a BOOSTER path $bp \in BPATH$. The base case (except for `SCStart`, `SCInput`, and `SCOutput`) is trivial, because it uses the auxiliary function `refStart` that queries from the environment about the type of bound variable. Our inductive hypothesis is that each recursive call to `booToObjPath` returns the correct result as stated. We first prove the case on transforming, for example, `SCStart`, particularly its index expression, by our inductive hypothesis on `booToObjPath`. We then prove all other cases where the given BOOSTER path is of recursive structure: 1) by our inductive hypothesis on the path transformation function, we know the prefix of the BOOSTER path has been transformed into the correct OBJECT path; 2) since `target` correctly locates the entity target from the prefix BOOSTER path; we conclude that a new OBJECT path that combines 1) and 2) correctly corresponds to the given BOOSTER path. □

Proof of Theorem 6.5.1 Lemma 6.5.1 as proved in Section 6.5.1 ensures that paths of type `OPath` in the OBJECT model consistently retrieve values, with respect to those of type `BPath`. Since we adopt the same set of expression operators for both model domains, we argue that two items are consistently represented in the OBJECT model: 1) paths that appear as targets of assignments ($:=$), as well as 2) expressions that appear in sources of assignments ($:=$) or in guards of guarded substitutions ($\longrightarrow$). By an induction on the same set of operation combinators that we adopt for both models—bounded choice ($\Box$), unbounded choice (@), parallel ($\|$), generalised parallel (!), sequential composition ($;$), and guarded substitution ($\longrightarrow$)—we prove this theorem. □

Proof of Lemma 6.5.2 Similar to the proof for Lemma 6.5.1, we prove by a structural induction on the structure of an OBJECT path $op \in OPATH$. The base case (except for `SCRef`) is trivial, as paths from both domains share the same data type `RefStart`. Our inductive hypothesis is that each recursive call to `objToTabPath` and `objToTabExpr` that uses it returns the correct result as stated. We first prove the case on transforming `SCRef`, particularly its index expression, by our inductive hypothesis on `objToTabExpr`. We then prove all other cases where the given object path is of recursive structure: 1) by our inductive hypothesis on the two transformation functions, we know the prefix of the object path has been transformed into the correct table path; 2) we can correctly locate the corresponding table for the entity target from the prefix object path; we conclude that a new table path that combines 1) and 2) correctly corresponds to the given object path. □

Proof of Theorem 6.5.2 Similar to the proof for Theorem 6.5.1, except we build upon Lemma 6.5.2 as proved above. □

Proof of Lemma E.7.1 Thanks to our specific design of the `TPATH` data type for table paths (Section 6.3.3) that carry information regarding what sorts of tables to access along the way, it is then straightforward (Section 6.5.5) to figure out the target tables, columns, and row conditions. □

Proof of Property E.7.1 As far as the consistency is concerned, all we ought to argue is that when `tp` changes its value, the mechanism is able to update its value before the next

phase, if any, begins. Obviously the sequential composition is the sole cause of the problem: as soon as `tp` is modified in the first phase, a reference to `v` in the second phase, without re-caching its value, would return an inconsistent value. As we have examined the complete specification of our transformation engine—not included in this dissertation—we update all declared cache variables (via `updateCache`) both right before and in-between each sequential composition. This guarantees that when each of the two phases starts, values of all cache variables are up-to-date. □

Proof of Theorem 6.5.3 The linking invariants we define in Section 6.2.5 precisely capture how we retrieve values of properties in the TABLE model domain and values of columns in the SQL model domain. Since each mapping function defined in Section 6.2.5 returns either a pair set or a triple set, we can use them just like relations. It is then straightforward to define two semantic mappings: $[\![\_]\!]^{expr}_{tab}$ and $[\![\_]\!]^{expr}_{sql}$ that map, respectively, TABLE and SQL expression combinators into ordinary set operators and set comprehensions. In particular, the `SELECT` statement maps exactly to the set comprehension expression.

For example, we consider the transformation of `Tail e` as shown above. In the TABLE domain, assuming that `e` is a path expression that has its identification $p \in$ `IdenProperty` and $context\,[\![e]\!] = ce$.

$$[\![\texttt{Tail e}]\!]^{expr}_{tab} = \{(i-1,\ v) \mid (o,i,v) \in \mathcal{M}_{seq}\,(e) \wedge o = ce \wedge i > 1\}$$

In the SQL domain, assuming that the target table and column of path `e` are, respectively, *tab* and *col*.

$$[\![toSqlExpr'\,(tm)\,(\texttt{Tail e})]\!]^{expr}_{tab} = \{(\texttt{index}-1,\ col) \mid (o,i,v) \in \mathcal{M}_{3cols}\,(s_{sql},\ tab,\ \texttt{index},\ col)\}$$

It is then obvious to see that

$$[\![\texttt{Tail e}]\!]^{expr}_{tab} = [\![toSqlExpr'\,(tm)\,(\texttt{Tail e})]\!]^{expr}_{tab}$$

In the TABLE domain we define the base case as:

$$\begin{aligned}[\![e]\!]^{expr}_{tab} = \ & \textbf{if } p \in \texttt{primitiveProperty } \textbf{then} \\ & \quad \{v \mid (o,v) \in \mathcal{M}_{prim}\,(s_{tab},\ p) \wedge o = ce\,\} \\ & \textbf{else if } p \in \texttt{seqProperty } \textbf{then} \\ & \quad \{(i,v) \mid (o,i,v) \in \mathcal{M}_{seq}\,(s_{tab},\ p) \wedge o = ce\,\} \\ & \textbf{else if } p \in \texttt{setProperty } \textbf{then} \\ & \quad \{v \mid (o,v) \in \mathcal{M}_{set}\,(s_{tab},\ p) \wedge o = ce\,\} \\ & \textbf{else if } p \in \texttt{biAssoc } \textbf{then} \\ & \quad \{v \mid (o,v) \in \mathcal{M}_{assoc}\,(s_{tab},\ p) \wedge o = ce\,\}\end{aligned}$$

We define the recursive case as:

$$\begin{aligned}[\![\texttt{Card e}]\!]^{expr}_{tab} &= \#[\![e]\!]^{expr}_{tab} \\ [\![\texttt{Head e}]\!]^{expr}_{tab} &= \{v \mid (i,v) \in [\![e]\!]^{expr}_{tab} \wedge i = 1\} \\ [\![\texttt{Tail e}]\!]^{expr}_{tab} &= \{(i-1,\ v) \mid (i,v) \in [\![e]\!]^{expr}_{tab} \wedge i > 1\} \\ [\![\texttt{Concat e f}]\!]^{expr}_{tab} &= [\![e]\!]^{expr}_{tab} \cup \{(i + \#[\![e]\!]^{expr}_{tab},\ v) \mid (i,v) \in [\![f]\!]^{expr}_{tab}\}\end{aligned}$$

In the SQL domain we define:

$$[\![\texttt{SELECT}\ col\ \texttt{FROM}\ t\ \texttt{WHERE}\ c]\!]^{expr}_{sql} = \{row : s_{sql}.tuples\,(t) \mid row \models c \bullet row.values\,(col)\}$$
$$[\![\texttt{SELECT}\ col_1, col_2\ \texttt{FROM}\ t\ \texttt{WHERE}\ c]\!]^{expr}_{sql} = \{row : s_{sql}.tuples\,(t) \mid row \models c \bullet row.values\,(col_1) \mapsto row.values\,(col_2)\}$$

Therefore, correctness of the base cases of predicates (`TRUE`, `FALSE`) and of expressions (such as `StringLiteral`, `Null`, etc) is trivial. An important base case is the treatment of path expressions, for which we prove by applying the correctness and consistency of the caching mechanism (Properties E.7.1 and E.7.1).

For other cases of transformation `toSqlExpr(e) = f`, where have either a direct or indirect correspondence between the two semantic domains, we will argue just as we do as the above example.

$$[\![e]\!]^{expr}_{tab} = [\![f]\!]^{expr}_{sql}$$

The proof of each case is straightforward as our rewriting rules of set or sequence operators are standard and can be easily justified from the returned values of $[\![\_]\!]^{expr}_{tab}$ and $[\![\_]\!]^{expr}_{sql}$. □

Proof of Property E.8.1 We observe that in definitions of `toSqlProc` and `updateQCacheVar`, the transformations on iterators and quantifiers are both defined as instantiating the `iterate` pattern. We also observe from the definition of the `iterate` pattern that a variable is dedicated to function as the loop variant, being decremented in each iteration and counting from the size of range expression in question towards zero. We then conclude that any instantiation to this pattern is guaranteed to terminate. □

Proof of Theorem 6.5.4 The proof will be built upon lemmas concerning the 36 patterns of basic assignments (see Tables E.1, E.2, E.3, and E.4), as well as lemmas concerning the combinators of `Substitution`. We will first prove the correctness of transformation on basic assignments. For each identified pattern we have the proof obligation:

$$\begin{array}{l} \forall\ TransInput;\ \Delta\mathcal{S}_{obj};\ \Delta\mathcal{S}_{sql} \mid \\ \quad prog = (tp? := te?) \land \\ \quad \textsc{Table} \leftrightarrow \textsc{Sql} \land \\ \quad \mathcal{R}_{sql}[effect_{sql}/effect] \land effect_{sql} = [\![toSqlProc\,(\theta\ TableModel)\,(prog)]\!]_{\mathrm{seq}\ sql} \bullet \\ \qquad \left(\begin{array}{l} \exists \mathcal{S}'_{obj} \bullet (\textsc{Table} \leftrightarrow \textsc{Sql})' \land \\ \qquad\qquad \mathcal{R}_{obj}[effect_{obj}/effect] \land effect_{obj} = [\![prog]\!]_{obj} \end{array} \right) \end{array}$$

where the context of the assignment statement $tp? := te?$ may be instantiated to any of the 36 patterns that we have identifed.
Let us first consider the pattern transformation as discussed in Section 1.1.4 (on page 11). The relevant linking invariant is E.3:

$$\mathcal{M}_{assoc}\,(\theta\,\mathcal{S}_{tab},\ p) = \mathcal{M}_{2cols}\,(\theta\,\mathcal{S}_{sql},\ [\![p]\!]^{NTab},\ [\![oppositeOf\ p]\!]^{NCol},\ [\![p]\!]^{NCol})$$

We prove by assuming the antecendent

$$\begin{array}{l}\textsc{Table} \leftrightarrow \textsc{Sql} \wedge \\ \quad \mathcal{R}_{sql}[\mathit{effect}_{sql}/\mathit{effect}] \wedge \\ \quad\quad \mathit{effect}_{sql} = [\![\mathit{toSqlProc}\,(\theta\ \mathit{TableModel})\,(\mathit{tp}? := \mathit{te}?)]\!]_{sql}\end{array}$$

It is then obvious that the linking invariant still holds between $\mathcal{S}_{obj}$' resulting from $\mathit{tp}? := \mathit{te}?$ and $\mathcal{S}_{sql}$' resulting from the `DELETE` and `INSERT` queries: by assigning $\mathit{te}?$ to $\mathit{tp}?$, that means $\mathit{te}?$ would lose its previously linked object (e.g. an account being deleted, hence also loses its owner) and now link to $\mathit{te}?$ (e.g. the same account being inserted back, with a new owner).

The proofs of all the remaining 35 patterns differ from the above proof in only the specific, relevant linking invariant to be applied, and in the number of times that the relational composition is applied to argue the corresponding steps of Sql statements.

We will now prove the correctness of transformation on combinators. We then state an inductive hypothesis: recursive invocations of *toSqlProc* correctly transform operands of all combinators. By this hypothesis, we immediately prove the case as the implementing SQL queries for each operand are put in sequence, from which we can easily argue the effect of their relational composition is equivalent to that of the GSL combinators. For example, it is obvious to see that

$$\begin{array}{rcl}[\![S\ ;\ T]\!]_{obj} & = & [\![S]\!]_{obj} \fatsemi [\![T]\!]_{obj} \\ & = & [\![\mathit{toSqlProc}\,(S)]\!]_{\mathrm{seq}\ sql} \fatsemi [\![\mathit{toSqlProc}\,(T)]\!]_{\mathrm{seq}\ sql}\end{array}$$

because by our inductive hypothesis

$$\begin{array}{rcl}[\![S]\!]_{obj} & = & [\![\mathit{toSqlProc}\,(S)]\!]_{\mathrm{seq}\ sql} \\ [\![T]\!]_{obj} & = & [\![\mathit{toSqlProc}\,(T)]\!]_{\mathrm{seq}\ sql}\end{array}$$

□

E.10 Library for Model Transformation

Instead of considering the entire tree-structured *Substitution* and *EXPRESSION* for pattern matching, we define auxiliary functions to specify the "forms" of model elements upon which we care to perform the path transformation. The structure for the rest of the tree is preserved.

Example Say we have $s \in SUBSTITUTION$ and want to increment all number literals that reside in s. The most straightforward approach is to explicitly consider the tree structure at different nesting levels.

```
incSubs (Sequence s t)    = Sequence (incSubs s) (incSubs t)
... -- other cases of Substitution

incPred (And      p q)    = And      (incPred p) (incPred q)
... -- other cases of Predicate

incExpr (NumberLiteral i) = NumberLiteral (i + 1)
incExpr  e                = e
```

The only function that does the transformation is *incExpr*, whereas all others just recurse while preserving the tree structure. There will be a number of scenarios like this in our transformation, so we want to specify an expression that is as easy as

```
everywhere (mkT inc) s
```

which will walk through the entire tree of s and apply *incExpr* to each subtree of type *Expression*. This is a much neater way because we define only the function that performs the transformation, whereas the library functions *everywhere* ensures that the structure for rest of the tree is preserved. □

We note that the function *everywhere* expects transformation function such as *incExpr* : *Expression* → *Expression* that has its domain and range of the same type. As far as our model transformation is concerned, we will perform transformations from, for example, Object to Table paths (Section 6.5.3), for which we will define, respectively, transformation functions of types $BPATH \rightarrow OPATH$ and $OPATH \rightarrow TPATH$. However, the function *everywhere* cannot accept either of these functions, as each of them has distinct types of domain and range. To resolve this, since $BPATH$, $OPATH$, and $TPATH$ construct the common type $PATH$, we will define functions of type $PATH \rightarrow PATH$ that in turn apply, respectively, those two transformation functions.

E.11 Examples of Path Transformation

Example E.11.1 In the context of the `Account` class (Figure 2.1 on page 35), a path `this.owner.reglist` that retrieves its owner's list of registered hotels has the abstract representation: `BPath (BPathSeq (Account This [Entity owner, Entity reglist]))`. □

Example E.11.2 The object path in Example E.11.1 has its `OPATH` counterpart

```
RecOPath (RecOPath (BaseOPath (ThisRef (ClassBase Account)))
                   (EntityTarget (Account, owner)))
         (EntityTarget (Traveller reglist))
```

□

Example E.11.3 The object path in Example E.11.2 has its `TPATH` counterpart

```
RecTPath (RecTPath (BaseTPath (ThisRef (ClassBase Account)))
                   (AssocTAccess (Account, owner)))
         (AssocTAccess (Traveller reglist))
```

where properties `owner` (an end of a bi-directional association) and `reglist` are accessed in the two corresponding association tables. □

Example E.11.4 The table path in Example E.11.3 has its SQL counterpart

```
SELECT (VAR   'reglist')
       (TABLE 'Hotel_registered_Traveller_reglist')
       (VAR   'oid' = (SELECT (VAR   'owner')
                              (TABLE 'Account_owner_Traveller_account')
                              (VAR    oid = VAR this)))
```

where `oid` is the default column (declared as the primary key) for each table that implements an association (Section E.4.2). The model-to-text transformation—from the `STATEMENT` datatype—to this concrete `SELECT` query is trivial. □

Example E.11.5 In the context of the `Account` class, from the predicate expression

```
forall h : owner.reglist @ card (h.employees) > 100
```

we would expect a transformation from `(PATH_START (Variable h))` (in the context of BOOSTER models) to `(REF_START (VariableRef h (ClassBase Staff)))` (in context of OBJECT model). The bound variable `h` has its type `extent (Hotel)` derived from the set-valued expression `owner.reglist`. The path expression `card (h.employees) > 100` is then valid because `employees` is a defined property of the `Hotel` class. □

Example E.11.6 In a program `m { i? : seq(Hotel) ==> ... i?[0] ... }` that declares in its guard an input sequence `i?`, we will transform the path expression `i?[0]` into the object path `BaseOPath (SCInputRef i (NumberLiteral 0) (ClassBase Hotel))`. □

Example E.11.7 We consider the transformation on the BOOSTER path from Example E.11.1 `BPath (BPathSeq (Account This [Entity owner, Entity reglist]))`. We delegate to a recursive call of `booToObjPath'` the task of transforming its prefix `BPath (BPathSeq (Account This [Entity owner]))` and the result from this is

```
RecOPath (BaseOPath (ThisRef (ClassBase Account)))
         (EntityTarget (Account, owner))
```

We will expect function `target` to infer that accessing property `reglist` through this context path navigates us to the class `Traveller` and thus returns `(EntityTarget (Traveller reglist))`. We then expect the suspending call of `objToObjPath'` to combine the two outputs, and we obtain exactly the OBJECT path as presented in Example E.11.2. □

E.12 Generated Database for HRS

In this section we will demonstrate the capability of our BOOSTER-to-SQL transformation engine: we will achieve this by generating a working relational database comprising table schemas and queries that implement, respectively, properties and methods of our hotel reservation system (Figure 2.1 and Appendix A.2). In particular, we show both the behavioural transformation on method `reserve` in class `Hotel` that makes a room yet to be confirmed, as well as the structural transformation on the attributes and associations involved. The transformation process and product on the rest of the system are similar and thus not discussed. We will first present the generated schemas of tables and their referential constraints (Appendix E.12.1). We will then present the generated stored function that implements the guard of *reserve* (Appendix E.12.2). We will finally present the generated stored procedure that implements the intent of update of *reserve* (Appendix E.12.3). The complete specification of method *reserve* in Appendix A.2 has been annotated by comments for its update substitutions—each of which corresponds to one of the 36 patterns that we have identified in Tables E.1 (on single-valued bi-associations), E.2 (on set-valued associations), E.3 (on sequence-valued associations), and E.4 (on primitive attributes). Our functional implementation on transforming basic assignments (Section 6.5.5) abides by patterns rules as specified in these tables.

E.12.1 Schemas of Tables & Referential Constraints

For each class we store only those attributes that are not declared as bi-associations; by default we store an `oid` column declared as its primary key. Class `Reservation` has only its `status` of primitive type and hence the only property stored in the corresponding class table[3]. Set-valued properties, like attribute `dates` in class `Reservation` is stored in a separate table with also an `oid` column to identify the current object in a given method call. We also store each association—such as attributes `host` and `reservations` between `Reservation` and `Hotel` (Figure 2.1)—in a separate table, with an `oid` column to identify the exact association instance. Since attribute `reservations` is also sequence-valued, we will also store an `index` column in its association table.

Schema of Tables Updated by 'reserve'

```
1  CREATE TABLE `Reservation`(`oid` INTEGER AUTO_INCREMENT,
2                             `status` CHAR(30),
3                              PRIMARY KEY (`oid`));

4  CREATE TABLE `Reservation_dates`(`oid` INTEGER AUTO_INCREMENT,
5                                   `dates` Date,
6                                   PRIMARY KEY (`oid`, `dates`));

7  CREATE TABLE `Reservation_host_Hotel_reservations`(`oid` INTEGER AUTO_INCREMENT,
8                                                     `host` INTEGER,
9                                                     `reservations` INTEGER,
10                                                    `index` INTEGER,
11                                                    PRIMARY KEY (`oid`));
```

[3] `AUTO_INCREMENT` here is for the target SQL platform to generate a unique identifier each time when we insert a new row.

```
12  CREATE TABLE ‘Room_reservations_Reservation_room‘(‘oid‘ INTEGER AUTO_INCREMENT,
13                                                    ‘reservations‘ INTEGER,
14                                                    ‘room‘ INTEGER,
15                                                    ‘index‘ INTEGER,
16                                                    PRIMARY KEY (‘oid‘));
```

End of Schema of Tables Updated by 'reserve'

We also generate referential constraints for association tables which will impose upon both ends of the association in question. In our example we require that the `host` column stores references to the class table for `Hotel`; the `reservations` column stores references to the class table for `Reservation`. Similar constraints are generated for the bi-association between `reservations` (of `Hotel`) and `room` (of `Reservation`).

Referential Constraints of Tables Updated by 'reserve'

```
1  ALTER TABLE ‘Reservation_host_Hotel_reservations‘
2    ADD FOREIGN KEY (‘host‘) REFERENCES ‘Hotel‘ (‘oid‘);
3  ALTER TABLE ‘Reservation_host_Hotel_reservations‘
4    ADD FOREIGN KEY (‘reservations‘) REFERENCES ‘Reservation‘ (‘oid‘);

5  ALTER TABLE ‘Room_reservations_Reservation_room‘
6    ADD FOREIGN KEY (‘reservations‘) REFERENCES ‘Reservation‘ (‘oid‘);
7  ALTER TABLE ‘Room_reservations_Reservation_room‘
8    ADD FOREIGN KEY (‘room‘) REFERENCES ‘Room‘ (‘oid‘);
```

End of Referential Constraints of Tables Updated by 'reserve'

In the next two sections we will illustrate how we decompose the transformation on each method into two: one that implements its precondition, to be implemented as a guard (Appendix E.12.2), and the other that implements its postcondition, to be implemented as an atomic transaction (Appendix E.12.3).

E.12.2 Implementing Guards as Stored Functions

The precondition of method *reserve* requires that the current number of allocations—characterised through the cardinality of the set-valued attribute `allocations`—is below a specific upper bound. We will generate a stored function—which has no effect on the database—which queries whether this guard holds at the current state; if not, we will block *reserve* and thus make it unavailable.

Stored Function Implementing the Guard of 'reserve'

```
1  CREATE FUNCTION ‘Hotel_reserve_guard‘
2          (‘this?‘ INTEGER, ‘dates?‘ CHAR(30), ‘m?‘ INTEGER) RETURNS BOOLEAN
3  BEGIN
4        DROP TEMPORARY TABLE IF EXISTS ‘this.allocations‘;
5        CREATE TEMPORARY TABLE ‘this.allocations‘ AS
6          SELECT ‘allocations‘ FROM ‘Allocation_host_Hotel_allocations‘
7                               WHERE ‘host‘ = ‘this‘;

8        RETURN SELECT COUNT(‘oid‘) FROM (SELECT ‘allocations‘
```

```
9                                       FROM 'this.allocations'
10                                      WHERE TRUE) AS allocations
11              WHERE TRUE) < 100;
12  END $$
```

End of Stored Function Implementing the Guard of 'reserve'

We first observe that names of all variables have been kept from the BOOSTER model domain, e.g. the input `dates?` at line 2, as well as the caching variable `this.allocations` at line 4. Lines 4 to 7 above exemplifies our caching mechanism (whose correctness is established in Property E.7.1 on page 257): we store the value of the set-valued attribute `allocations` in a temporary table. Lines 8 to 11 demonstrates the value of our caching mechanism: we will be able to obtain the value of `allocations`—until the next necessary re-evaluation of its value—by applying an efficient `SELECT` query, i.e. with its `WHERE` condition being `TRUE`. The more frequently multi-valued attributes such as `allocations` appear in long paths in the guard of its enclosing method, the more obvious we will see the improvement on performance. In the next section we will consider how we generate state-changing updates from the specification of method *reserve*.

E.12.3 Implementing Substitutions as Stored Procedures

We can immediately observe the value of our automated transformation by comparing the specification of method `reserve` (Appendix A.2) and all fragments of its implementation below—repeatedly writing queries that are consistent with the chosen implementation and caching strategies is destined to be tedious and error-prone. We will discuss this generated implementation in fragments.

We observe again that names of all variables have been kept from the BOOSTER model domain, e.g. the input and output parameters `dates?` and `r!` at line 2, as well as caching variable `m?.reservations` at line 25. We have declarations of caching variables (Lines 4 to 6) and meta-variables for implementing loops (Lines 8 to 10). Specifically, Line 10 declares a cursor over the set-valued input `dates?`, and Lines 8 and 9 declare, respectively, the bound variable and the variant of the loop—which has been shown to ensure its termination in Property E.8.1 on page 258—that implements the `ALL` iterator in method `reserve`.

Queries Implementing 'reserve': Declarations

```
1  CREATE PROCEDURE 'Hotel_reserve'
2      (IN 'this?' INTEGER, IN 'dates?' CHAR(30), IN 'm?' INTEGER, OUT 'r!' INTEGER)
3  BEGIN
4    DECLARE 'r!.status' CHAR(30);
5    DECLARE 'r!.host' INTEGER;
6    DECLARE 'r!.room' INTEGER;
7    -- loop-specific declarations
8    DECLARE 'x' Date;
9    DECLARE 'x_variant' INTEGER;
10   DECLARE 'x_cursor' CURSOR FOR SELECT * FROM 'dates?' WHERE TRUE;
```

End of Queries Implementing 'reserve': Declarations

Line 11 creates a new instance of `Reservation` by inserting an empty row—for output `r!`—into the appropriate class table. Line 12 assigns the corresponding, unique identifier to

`r!` for queries in later fragments to refer to. We will see in a later fragment how we initialise values of this empty output.

Queries Implementing 'reserve': Creating an Empty Output

```
11  INSERT INTO `Reservation` () VALUE ();
12  SET `r!` = last_insert_id ();
```

End of Queries Implementing 'reserve': Creating an Empty Output

From Lines 13 to 46 we update values of all declared cache variables. Each pair of `DROP TEMPORARY TABLE` and `CREATE TEMPORARY TABLE` queries update the value of a cache variable for some multi-valued properties. This is similar to what we discussed for attribute `allocations` in Appendix E.12.2. We update the caching variables for other types of properties through a `SELECT INTO` query. For example, at Lines 18 to 19, we cache the value of attribute `host` as possessed by the reservation `r!`. This would mean that any later path with `r!.host` as its prefix will be able to use its value directly without performing a—potentially costly—re-evaluation.

Queries Implementing 'reserve': Updating Caching Variables

```
13  DROP TEMPORARY TABLE IF EXISTS `r!.dates`;
14  CREATE TEMPORARY TABLE `r!.dates` AS
15    SELECT `dates` FROM `Reservation_dates`
16    WHERE `oid` = `r!`;

17  SELECT `status` INTO `r!.status` FROM `Reservation` WHERE `oid` = `r!`;

18  SELECT `host` INTO `r!.host` FROM `Reservation_host_Hotel_reservations`
19                               WHERE `reservations` = `r!`;

20  DROP TEMPORARY TABLE IF EXISTS `this.reservations`;
21  CREATE TEMPORARY TABLE `this.reservations` AS
22    SELECT `reservations` FROM `Reservation_host_Hotel_reservations`
23                          WHERE `host` = `this`;

24  SELECT `room` INTO `r!.room` FROM `Room_reservations_Reservation_room`
25                               WHERE `reservations` = `r!`;

26  DROP TEMPORARY TABLE IF EXISTS `m?.reservations`;
27  CREATE TEMPORARY TABLE `m?.reservations` AS
28    SELECT `reservations` FROM `Room_reservations_Reservation_room`
29                          WHERE `room` = `m?`;

30  DROP TEMPORARY TABLE IF EXISTS `r!.dates`;
31  CREATE TEMPORARY TABLE `r!.dates` AS
32    SELECT `dates` FROM `Reservation_dates`
33                   WHERE `oid` = `r!`;

34  SELECT `status` INTO `r!.status` FROM `Reservation` WHERE `oid` = `r!`;

35  SELECT `host` INTO `r!.host` FROM `Reservation_host_Hotel_reservations`
36                               WHERE `reservations` = `r!`;
```

```
37    DROP TEMPORARY TABLE IF EXISTS `this.reservations`;
38    CREATE TEMPORARY TABLE `this.reservations` AS
39      SELECT `reservations` FROM `Reservation_host_Hotel_reservations`
40                            WHERE `host` = `this`;

41    SELECT `room` INTO `r!.room` FROM `Room_reservations_Reservation_room`
42                                 WHERE (`reservations`) = (`r!`);

43    DROP TEMPORARY TABLE IF EXISTS `m?.reservations`;
44    CREATE TEMPORARY TABLE `m?.reservations` AS
45      SELECT `reservations` FROM `Room_reservations_Reservation_room`
46                            WHERE `room` = `m?`;
```

End of Queries Implementing 'reserve': Updating Caching Variables

Lines 47 to 59 instantiate our loop pattern in Section 6.5.5, for which the guarantee for termination was established in Property E.8.1 on page 258. We activate the declared cursor (Line 47) and and fetch its first available value (Line 48). We also calculate the size of the data set that the cursor will iterate over and use it as the variant of the loop defined in Lines 50 to 59. The exit condition is characterised through the decreasing—via Line 57—value of `x_cursor`; the bound variable `x` is updated with its value in each iteration at Line 56. We also update values of caching variables, just as long paths might occur in contexts of either iterators or quantifiers. For example, in each iteration of the loop, from Lines 50 to 54 we re-cache the value of the set-valued path `r!.dates`, in case there are other paths which contain it as a prefix. At Line 55 we perform the update of `r!.dates := r!.dates \/ { d }`, defined as the first substitution in the specification of method `reserve`.

Queries Implementing 'reserve': Terminating Loop

```
47    OPEN `x_cursor`;
48    FETCH `x_cursor` INTO `x`;
49    SELECT COUNT(*) INTO `x_variant` FROM `dates?` WHERE TRUE;

50    WHILE (`x_variant`) > (0) DO
51      DROP TEMPORARY TABLE IF EXISTS `r!.dates`;
52      CREATE TEMPORARY TABLE `r!.dates` AS
53        SELECT `dates` FROM `Reservation_dates`
54                       WHERE `oid` = `r!`;

55      INSERT INTO `Reservation_dates` (`oid`, `dates`) VALUE (`r!`, `x`);

56      FETCH `x_cursor` INTO `x`;
57      SET `x_variant` = `x_variant` - 1;
58    END WHILE;

59    CLOSE `x_cursor`;
```

End of Queries Implementing 'reserve': Terminating Loop

Finally, we transform the rest of the updates as specified in `reserve`. Line 60 implements the update `r!.status := unconfirmed`. As we have emphasised, each substitution as specified in `reserve` has been annotated with a comment that indicates which basic

assignment pattern it refers to in one of the tables in Section E.6. For example, the four generated query statements—that are located in Lines 61 to 70, Line 71, Lines 72 to 80, and Lines 81 to 86—correspond exactly to the rules specified for Pattern #21 in Table E.3; here we intend to update the *sequence*-to-1 association between classes `Hotel` and `Reservation`. Similarly, the two generated query statements—that are located in Lines 87 to 95 and Lines 96 to 101—correspond to Pattern #23 in Table E.3.

Queries Implementing 'reserve': Performing Updates

```
60  UPDATE `Reservation` SET `status` = 'unconfirmed' WHERE (`oid`) = (`r!`);

61  UPDATE `Reservation_host_Hotel_reservations`
62  SET `index` = `index` - 1
63  WHERE `host` = SELECT `host` FROM `Reservation_host_Hotel_reservations`
64                               WHERE `reservations` = `r!`
65                                     AND
66                                     `index` > (SELECT COUNT(`oid`)
67                                                FROM (SELECT `reservations`
68                                                      FROM `this.reservations`
69                                                      WHERE TRUE) AS reservations
70                                                WHERE TRUE) + 1;

71  DELETE FROM `Reservation_host_Hotel_reservations` WHERE `reservations` = `r!`;

72  UPDATE `Reservation_host_Hotel_reservations`
73  SET `index` = `index` + 1
74  WHERE `host` = `this`
75        AND
76        `index` >= (SELECT COUNT(`oid`)
77                    FROM (SELECT `reservations`
78                          FROM `this.reservations`
79                          WHERE TRUE) AS reservations
80                       WHERE TRUE) + 1;

81  INSERT INTO `Reservation_host_Hotel_reservations` (`reservations`, `host`, `index`)
82    VALUE (`r!`, `this`, (SELECT COUNT(`oid`)
83                          FROM (SELECT `reservations`
84                                FROM `this.reservations`
85                                WHERE TRUE) AS reservations
86                          WHERE TRUE) + 1);

87  UPDATE `Room_reservations_Reservation_room`
88  SET `index` = (`index`) + (1)
89  WHERE `room` = `m?`
90        AND
91        `index` >= (SELECT COUNT(`oid`)
92                    FROM (SELECT `reservations`
93                          FROM `m?.reservations`
94                          WHERE TRUE) AS reservations
95                    WHERE TRUE) + 1;

96  INSERT INTO `Room_reservations_Reservation_room` (`reservations`, `room`, `index`)
97    VALUE (`r!`, `m?`, (SELECT COUNT(`oid`)
```

```
98                          FROM (SELECT ‘reservations‘
99                                FROM ‘m?.reservations‘
100                               WHERE TRUE) AS reservations
101                         WHERE TRUE) + 1);

102 END $$
```

End of Queries Implementing 'reserve': Performing Updates