UNIVERSITY OF SOUTHAMPTON

Developing Verified Sequential Programs with Event-B

by

Mohammadsadegh Dalvandi

A thesis submitted in partial fulfillment for the degree of Doctor of Philosophy

in the

Faculty of Physical Sciences and Engineering

Electronics and Computer Science

April 2018

UNIVERSITY OF SOUTHAMPTON

ABSTRACT

FACULTY OF PHYSICAL SCIENCES AND ENGINEERING
ELECTRONICS AND COMPUTER SCIENCE

Doctor of Philosophy

by Mohammadsadegh Dalvandi

The constructive approach to software correctness aims at formal modelling of the intended behaviour and structure of a system in different levels of abstraction and verifying properties of models. The target of analytical approach is to verify properties of the final program code. A high level look at these two approaches suggests that the constructive and analytical approaches should complement each other well. The aim of this thesis is to build a link between Event-B (constructive approach) and Dafny (analytical approach) for developing sequential verified programs. The first contribution of this thesis is a tool supported method for transforming Event-B models to simple Dafny code contracts (in the form of method pre- and post-conditions). Transformation of Event-B formal models to Dafny method declarations and code contracts is enabled by a set of transformation rules. Using this set of transformation rules, one can generate code contracts from Event-B models but not implementations. The generated code contracts must be seen as an interface that can be implemented. If there is an implementation that satisfies the generated contracts then it is considered to be a correct implementation of the abstract Event-B model. A tool for automatic transformation of Event-B models to simple Dafny code contracts is presented. The second contribution of this thesis is an approach for derivation of algorithmic structure in Event-B refinement. To facilitate this, we augment Event-B with a scheduling language that allows modeller to explicitly define the control flow between Event-B events in each refinement level. The scheduling language supports both non-deterministic (choices and iterations) and deterministic (conditionals and loops) control structures and treat Event-B events as its atoms. We provide a set of schedule refinement rules for refining an abstract scheduling language to a concrete program structure. We also provide a set of rules allowing the elimination of event guards at the concrete level. The final contribution of this thesis is a method for transforming scheduled Event-B models to Dafny code and contracts. We formulate the transformation of a scheduled Event-B model to Dafny program constructs and show how the actions of an atomic event can be sequentialised in the final program. We introduce an approach for generation of Dafny contracts in the form of assertions in order to verify the correctness of the sequentialisation.

Contents

List of Figures

List of Tables

Listings

Acknowledgements

First and foremost, I would like to express my sincere gratitude to my supervisor Professor Michael Butler. I believe his vast knowledge, experience and patience are the qualities that make him an exceptional supervisor. It has been of great privilege and honour working under his supervision.

I would also like to thank my second supervisor Dr Abdolbaghi Rezazadeh for his continued support during my PhD studies. His advice in the different aspects of my PhD research was of invaluable help.

I am also thankful to Dr Gennaro Parlato who was my internal examiner during the early stages of my PhD. I would also like to thank all my friends and colleagues in the Electronic and Software Systems (ESS) research group.

Finally I am forever indebted to my parents and wife Narges for their unconditional love, support and encouragement throughout my research.

This work would have not been possible for me without the help and support of these people.

I gratefully acknowledge that this work was partially supported by generous funding from Microsoft Research and University of Southampton.

Chapter 1

Introduction

The increasing demand for more functionalities and features in software systems makes them more complex and consequently more error-prone. Given the vital role that software systems play in our everyday life and our high level of dependency on safety and business critical software systems, their correct functioning is crucial. Failure in safety or business critical systems can be catastrophic and may result in loss of lives and money.

A *failure* occurs when the system no longer complies with the description of its expected behaviour (i.e. specification). Failures happen because of *errors* in the system. An error is a state in the system that is liable to lead to a failure [Lap95]. By removing the errors from a system we can prevent its failure. Once a software system is built, it may be tested in order to discover the possible errors. Testing process may find a number of errors in a system however it cannot prove that a system is error-free (faultless).

One way to build a faultless software system is to employ rigorous engineering disciplines in order to prove the correctness of a software system in different phases of development. Formal methods are amongst the effective ways that provide software engineering with mathematical techniques to prove the correctness of a system.

Formal methods in software engineering are mathematical-based techniques that offer a systematic approach in which some of a software system properties can be specified, developed and verified. The essence of a formal method is a sound and well-defined mathematical basis which is defined by a formal specification language. Defining concepts like consistency and correctness in a precise way is facilitated by the use of this mathematical basis. It also provides the means to verify the properties of a system without running it in order to observe its behaviours [Win90]. The effectiveness of formal methods has been demonstrated through various successful industrial projects [CW96, WLBF09, Abr07].

Two major approaches to software correctness can be distinguished based on their target phases in the development cycle: the *constructive approach* (or top-down) and the *analytical approach* (or bottom-up). The constructive approach can be employed from the early stages of the development. It aims at formal modelling of the intended behaviour and structure of a system in different levels of abstraction and verifying the properties of the models. The analytical approach focuses on the code level and its target is to verify the properties of the final program code [Dij68].

When a system is developed using the constructive approach, the first step is to provide a high level abstract specification of the system. This abstract model of the system focuses on specifying the ultimate goal of the system rather than how this can be achieved. This abstract specification then can be augmented with more details in a number of successive steps called *refinement*. This approach provides a way in which different properties of a system can be modelled and proved in different levels of abstraction. Each level is proved to be consistent with respect to the higher level. Splitting the proof effort between different refinement levels helps to tackle the proof complexity. However, the difficulty of derivation of a correct implementation in a programming language from the mathematical model of a system built using a constructive method is a challenge.

When the analytical approach is employed for developing verified software, the development starts at code level with lots of details that are needed for implementation in a programming language. In this approach, the formal specification of the software is provided separately and the implementation is verified against the specification. The presence of a large amount of detail in a low-level programming language makes the reasoning difficult in comparison with the reasoning about the abstract mathematical model of a software in the constructive method. Despite the difficulty of formal reasoning about a program written in a low-level language at code level, it is a common style of programming to start the development of a software with many details.

Leino in [LY12] has identified some issues with the common style of programming and suggested the stepwise refinement offered by the constructive approach as an alternative. Some of the identified issues are as follows: First, detecting specification errors will be delayed until the software is completely implemented and the testing phase is started when it is very costly. Second, algorithms are usually understood by programmers in terms of pseudo-code but they are not recorded as part of the program text and cannot be compared with the implementation. Third, software evolves over time by introduction of optimizations. This is usually done by replacing old code with the new optimized one. This replacement makes the understanding of the software harder for the developer and the later examination of the code. In stepwise refinement, intermediate stages of development (various levels of abstraction of the program model) are recorded and preserved in a format that is understandable to the human and also can be analysed by tools. Stepwise refinement also makes the complexity manageable by hiding details. By following this approach, specification errors can be detected earlier and by going back

to and changing the original description, errors can be removed and also the necessary constraints that have not been evident before can be applied. Another benefit of using the constructive approach is that the specification is recorded in the program in various levels of abstraction. This makes the program more understandable to the human and also allows the verification of the program with regards to its high-level specification if adequate tools are provided.

1.1 Motivation: Linking Verification Methods

Various formal methods communities [CW96, HMLS09, LAB+06, WB07] have suggested that no single formal method can cover all aspects of a verification problem, therefore engineering bridges between complementary verification tools to enable their effective interoperability may increase the verification capabilities of verification tools.

As mentioned before, we distinguish two formal approaches to software correctness, namely constructive and analytical approaches where the former aims at verification of high-level properties of abstract model of the software and the latter focuses on verification of properties at code level. A wide range of verification tools exist to support both approaches provided by formal methods' communities worldwide. A high level look at these two approaches suggests that the constructive and analytical approaches should complement each other well. Nevertheless, our understanding and experience of how these approaches can be combined at a large scale is very limited. This represents an opportunity to combine these approaches in a way that complement each other effectively.

We have chosen Event-B [Abr10] and Dafny [Lei10] as examples of constructive and analytical approaches respectively. Event-B is a formal approach for modelling and verifying software systems. An Event-B model is built through a number of successive refinement steps starting from an abstract representation of the system towards a concrete level. Event-B is supported by an open platform called Rodin [ABH+10]. Dafny is a programming language and verifier. Dafny pursues the idea of *design-by-contract* [Mey02] in which the specification annotations are embedded in the language. A Dafny program can be specified formally with the help of its rich set of built-in specification constructs. Given a program code and its formal specification, the Dafny tool can verify the program against its specification.

The main motivation of this research is to increase interoperability and effectiveness of different formal approaches to software correctness by building a link between constructive (Event-B) and analytical (Dafny) approaches. Our aim is to develop a combined method to employ the abstraction and refinement power of Event-B with the static code analysis of Dafny in the process of constructing correct software. This method is beneficial for both Event-B and Dafny users. It makes the abstraction and refinement

of Event-B available for generating Dafny code and contracts which are correct with respect to a higher level of abstract specification in Event-B and provides a framework in which Event-B models can be implemented and verified in a programming language.

1.2 Contributions

As previously stated, the aim of this work is devising a method to develop verified software by employing a combination of constructive and analytical approaches. The idea is to start with an abstract specification of the system under development in Event-B and refine it towards a concrete implementation in Dafny in a way that the implementation satisfies the abstract specification. The verification task is split over different refinement levels in Event-B and Dafny where high-level specification and algorithmic structure are verified in Event-B and low-level code-oriented properties (e.g. correctness of sequentially composed assignment statements) are verified at Dafny level.

This thesis introduces two methods for linking Event-B and Dafny. The first method transforms an Event-B model to Dafny method declarations and pre- and post-conditions without actual implementation of the model. By contrast, the second method results in derivation of the algorithmic structure and the implementation of the model in Dafny. The rest of this section outlines the two aforementioned methods and contributions of this thesis.

- **First Contribution: Event-B Model to Simple Code Contracts** The first contribution of this thesis is introducing an approach for transforming Event-B models to simple Dafny code contracts (i.e. method's pre- and post-conditions) in a way that any implementation that satisfies the generated code contracts is considered to be a correct implementation of the abstract Event-B specification. The aim of this approach is to generate Dafny code contracts from Event-B models rather than implementation. Providing a Dafny implementation that satisfies the generated contracts will remain as a future task for the modeller. We have developed a tool for automatic transformation of Event-B models to simple Dafny contracts. The approach and the tool are validated through a number of case studies.

- **Second Contribution: Scheduled Event-B** Transformation of an abstract model to simple Dafny code contracts will work fine for simple algorithms (e.g. manipulation of simple abstract datatypes) as there is usually an implementation that can be verified against the generated code contracts in Dafny with minimal effort (in terms of providing new assertions such as complicated loop invariants, variants, and etc.). However, if there is no implementation that can satisfy the generated code contracts without providing further complex assertions, then the

aforementioned approach is not considered to be efficient since the gap between the abstract Event-B specification and the concrete implementation is significant. One way to lessen this gap is to take advantage of refinement in Event-B in order to derive and verify the implementation in a series of refinement steps before transforming the model to Dafny code and contracts.

Developing an algorithm in Event-B usually starts with one or more atomic event specifying different desirable outcomes of the algorithm (e.g. success or failure of a search algorithm). Further refinements introduce more events to the model in order to decompose atomicity of the abstract events and add more details on how the goal of the algorithm is going to be achieved. In each refinement level, execution of a group of events in a specific order which is enforced by event guards, results in achieving the goal. The final (concrete) refinement level usually specifies all the necessary deterministic steps that are needed to be taken in order to fulfil the main objective of the algorithm. The concrete level is much closer to the final implementation than the abstract specification. However, because Event-B lacks explicit control structure, it is still difficult to transform the model to the final code.

The second contribution of this thesis is called Scheduled Event-B (SEB) which addresses the problem of control flow in Event-B by augmenting it with a scheduling language in order to make the algorithmic structure explicit. The scheduling language allows the modeller to make the control flow and algorithmic structure explicit as soon as the development is started in abstract level. The proposed scheduling language contains familiar control constructs like (deterministic/non-deterministic) loops and branches. We have introduced a number of refinement patterns in order to make it easier for the modeller to refine the abstract control flow to a more concrete one. The final refinement level will lead to a model with deterministic algorithmic structure and events which is very similar to the final program code and can be easily transformed to implementation. We have used SEB to develop couple of examples including the Schorr-Waite graph marking algorithm.

- **Third Contribution: Scheduled Event-B to Dafny** An Event-B model comprises of atomic guarded events with a number of actions (assignments) which model the state changes in the system. Scheduled Event-B imposes control flow on atomic events rather than individual actions. However, in a sequential programming language we cannot execute a number of assignments atomically hence we need to sequentialise assignments in the final code properly. The third contribution of this thesis is to formulate the transformation of a Scheduled Event-B model to Dafny implementation. This includes the sequentialisation of events' actions and translation of control structures to their Dafny counterparts. The transformation also leads to the generation of a number of Dafny assertion for verification of the sequentialisation correctness at the code level.

To sum up, the overall contribution of this thesis is the development of a method for developing correct sequential programs by linking high level specifications in Event-B and low level program and contracts in Dafny. This thesis links Event-B and Dafny and experiments with the combined method in two ways:

- First, by providing an approach for direct mapping between Event-B specifications and Dafny method code contracts (i.e. pre- and post-conditions) in a way that satisfying the Dafny contracts by any implementation implies that the implementation is a correct refinement of the abstract Event-B model.
- Second, by augmenting Event-B with a scheduling language that allows derivation of algorithmic structure in a stepwise manner and transforming the model to code and contracts in Dafny in a way that most of the verification is performed in Event-B refinement step and only sequentialising event actions is verified in the Dafny level.

1.3 Publications

Parts of this thesis has been already published in the following papers:

- Mohammadsadegh Dalvandi, Michael Butler, and Abdolbaghi Rezazadeh. From Event-B models to Dafny code contracts. In Mehdi Dastani and Marjan Sirjani, editors, *Fundamentals of Software Engineering*, volume 9392 of *LNCS*, pages 308–315. Springer International Publishing, 2015
- Mohammadsadegh Dalvandi, Michael J. Butler, and Abdolbaghi Rezazadeh. Transforming Event-B models to Dafny contracts. *ECEASST*, 72, 2015
- Mohammadsadegh Dalvandi, Michael Butler, and Abdolbaghi Rezazadeh. Derivation of algorithmic control structures in Event-B refinement. *Science of Computer Programming*, 148(Supplement C):49 – 65, 2017. Special issue on Automated Verification of Critical Systems (AVoCS 2015)

1.4 Report Organisation

This thesis is organised in 9 chapters. Chapter 2 provides background information on Event-B and Dafny. Chapter 3 explains the methodology, transformation rules, and the formal basis of our work for transforming Event-B models to simple Dafny code contracts. Chapter 4 illustrates three examples of Event-B models and their transformation to simple Dafny code contracts. The tool support for simple contract generation is discussed in Chapter 5. In Chapter 6 Scheduled Event-B (scheduling language and refinement

patterns) is discussed. Chapter 7 presents two case studies (a sorting algorithm and the Schorr-Waite algorithm) and their development in Scheduled Event-B. The final step of the development, generation of Dafny executable code and contracts required for verification is presented in Chapter 8. Finally, Chapter 9 concludes this thesis.

Chapter 2

Background

This chapter provides background information that is necessary for understanding this work. In Section 2.1, we provide a brief overview of Guarded Command Language. In Section 2.2, we introduce the Event-B formal method and its tool support. In Section 2.3 Hoare logic and some of its inference rules are discussed. Section 2.4 introduces refinement calculus. The concept of design by contract is briefly presented in Section 2.5. Section 2.6 provides preliminary information on the Dafny language and verifier. Finally, in Section 2.7 an overview on the related work to development of verified software with model-based formal methods and linking different formal approaches is presented.

2.1 Guarded Command Language

Guarded command language (GCL) was presented by Dijkstra in 1970s [Dij75]. The language is based on the concept of the *guarded commands*. It provides choice and iteration constructs and allows non-determinism in the program. By using GCL a program can be constructed without necessarily determining the computation or a unique final state. The syntax of the language is as follows:

⟨*guarded command*⟩ ::= ⟨*guard*⟩→⟨*guarded list*⟩

⟨*guard*⟩ ::= ⟨*boolean expression*⟩

⟨*guarded list*⟩ ::= ⟨*statement*⟩ {; ⟨*statement*⟩}

⟨*guarded command set*⟩ ::= ⟨*guarded command*⟩ { □ ⟨*guarded command*⟩}

⟨*alternative construct*⟩ ::= `if` ⟨*guarded command set*⟩ `fi`

⟨*repetitive construct*⟩ ::= `do` ⟨*guarded command set*⟩ `od`

⟨*statement*⟩ ::= ⟨*alternative statement*⟩
| ⟨*repetitive construct*⟩
| `other statements`

where `other statements` is an assignment or a procedure call. The semicolon represents sequential composition. If the guarded list is executed then its statements will be executed from left to right. A guarded command is a component of a guarded command set. Using alternative and repetitive constructs, a statement can be constructed out of a guarded command set. If an alternative construct is executed and there is no guarded command in the guarded command set which its guard is true then the program aborts otherwise an arbitrary guarded command with a true guard will be executed. If a repetitive construct is executed and there is no guarded command with a true guard in the guarded command set then the loop terminates. Otherwise one of the guarded commands with a true guard will be arbitrary chosen for execution in each iteration.

2.1.1 Weakest Pre-condition

Dijkstra used the concept of *weakest pre-condition* to define the semantics of the language. The notation *wp(S,R)* (where S and R are an statement and a condition on the state of the system, respectively) denotes the weakest pre-condition for the initial state such that execution of S is guaranteed to be terminated in a state satisfying the post-condition R [Dij75, Dij76]. Some of the weakest pre-condition rules are as follows:

- **Skip**:

$$wp(skip, R) = R$$

- **Assignment**:

$$wp(x := E, R) = R[E/x]$$

- **Sequence**:

$$wp(S_1 ; S_2, R) = wp(S_1, wp(S_2, R))$$

- **Alternative**

$$\begin{aligned} wp(\textbf{if}\, E_1 \mapsto S_1 \,\square\, ... \,\square\, E_n \mapsto S_n\, \textbf{fi}, R) = \quad & (E_1 \vee ... \vee E_n) \\ & \wedge (E_1 \Rightarrow wp(S_1, R)) \\ & ... \\ & \wedge (E_n \Rightarrow wp(S_n, R)) \end{aligned}$$

Weakest-preconditions are a special kind of a more general concept called *Predicate Transformer*. A predicate transformer is a total function on the state of the program which relates a post-condition to a pre-condition in a way that if the program is executed under that pre-condition, it is guaranteed to terminates in a state satisfying the post-condition. In this thesis we write a predicate transformer as follows:

$$S.q$$

where S is a statement and q is the post-condition.

Later in this thesis we use guarded command language and predicate transformers to define the semantics of Event-B, Dafny and our scheduling language.

2.2 Event-B

Event-B is a formal modelling language for system level modelling based on set theory and predicate logic for specifying, modelling and reasoning about systems, introduced by Abrial [Abr10]. Although Event-B is evolved from B-method, they are used for different purposes. The purpose of the B Method [Abr05] is the development of correct by construction software, while the Event-B aims at the system modelling which may include hardware, software and environment of operation [JLPC12]. Event-B is greatly inspired by the notion of Action Systems [BKS88] and Guarded Commands [Dij75, Dij76]. The structure of an Event-B specification is similar to those in Action Systems where the behaviour of the system is specified by guarded actions.

In the rest of this section the Event-B method is described. The structure of Event-B models, stepwise refinement, and proof obligations are discussed in more details.

2.2.1 Structure and Notation

A model in Event-B consists of two main part: *contexts* and *machines*. The static part (types and constants) of a model is placed in a context and is specified using carrier sets, constants and axioms. The dynamic part (variables and events) is specified in a

machine by means of variables, invariants and events. A context, say C, can be seen by one or more machines and it can be extended by another context D. A machine, say M, can be refined by another machine N. In this case, it is said that N is refinement of M and M is an abstraction of N. If machine M can see context C, then N can see context C too. Figure 2.1 illustrates the relation between machines and contexts in an Event-B development.

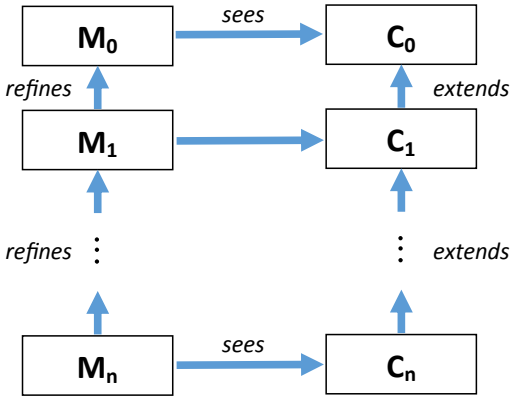

FIGURE 2.1: Machine Refinement and Context Extension

In the above figure machine M_0 is the most abstract machine and M_n is the most concrete machine in the development.

2.2.1.1 Context Structure

To describe the static part of a model there are four elements in the structure of a context: *carrier sets*, *constants*, *axioms*, and *theorems*. Carrier sets are represented by their name and they are disjoint from each other. Constants are defined using axioms. Axioms are predicates that express properties of sets and constants. Theorems in contexts should be proved from axioms.

2.2.1.2 Machine Structure

A machine in Event-B consists of three main elements: (1) a set of *variables*, which defines the states of a model (2) a set of *invariants*, which is a set of conditions on state variables that must hold whenever a variable is changed by an event and (3) a number of *events* which model the state change in the system. Each event may have a number of assignments called *actions*. Each event may also have a number of *guards*. Guards are predicates that are implicitly conjoined forming the necessary overall enabling condition of an event. Note that an event might not have any guards, in this case that event will always be enabled. An event may have a number of parameters. Event parameters are

considered to be local to the event. Variables and invariants are denoted by v and $I(v)$ respectively. An event may have one of the following forms:

$$\boxed{\text{Evt} \triangleq \textbf{when } P(v) \textbf{ then } S(v) \textbf{ end}}$$
$$\boxed{\text{Evt} \triangleq \textbf{any } t \textbf{ when } P(t,v) \textbf{ then } S(t,v) \textbf{ end}}$$

The difference between above events is that the first event does not have any parameters while the second one has one or more parameters represented by t. $P(v)$ and $P(t,v)$ denote guards and $S(v)$ and $S(t,v)$ denote actions.

As mentioned above, actions are assignments describing the associated state change with an event. An assignment may have one of the following types: (1) deterministic assignment, (2) non-deterministic assignment of a value from a set, or (3)non-deterministic assignment of a value satisfying a predicate. There may also be no action in an event which in that case it is shown by `skip`. Table 2.1 shows these three types and a simple example for each.

Type	**Assignment**	**Example**
deterministic	$x := E(t,v)$	$x := x + 1$
non-deterministic	$x :\in E(t,v)$	$x :\in A \cup \{y\}$
non-deterministic	$x :\mid Q(t,v,x')$	$x,y :\mid x' > y \wedge y' > x' + z$

TABLE 2.1: Assignment Types in Event-B

In the Table 2.1 [AH07] x is a variable, $E(t,v)$ is an expression, and $Q(t,v,x')$ is a predicate.

There also maybe *theorems* in a context or a machine. Axioms, invariants, and guards can be marked as theorems. This means that the validity of the theorem should be proved by axioms, invariants, and guards which appeared before the theorem [AH07].

Events may be labelled as *convergent* which means that the event requires a *variant*. A variant is a natural number or finite set expression which must be decreased by all *convergent* events. This is to prove that events *do not diverge* and a chosen set of events are enabled only a finite number of time before a terminating event occurs [AH07].

2.2.2 Refinement in Event-B

Modelling a complex system in Event-B largely benefits from refinement as modelling a system in different levels of abstraction helps to tackle the complexity of design. Refinement is a stepwise process of building a large system starting from an abstract level towards a concrete level [But09, But13]. This is done by a series of steps in which, new details of functionality are added to the model in each step. The abstract level represents

key features and the main purpose of the system. The abstract model does not include implementation details and *how* the goal of the system is going to be achieved. Instead, it focuses on *what* is the goal of the system.

It is important to prove the correctness of refinement steps in Event-B. This is discussed in the next section in detail.

Refining an Event-B model may involve context extension and machine refinement. When a context is extended, new sets, constants, and axioms are added and sets and constants in abstract context will be kept in the extension. Context extension has no associated proof obligations.

All abstract events must be refined by one or more concrete events. New events can be added in a refined machine. All new events refine a dummy skip event in the abstract machine. The new events must not diverge. This means that they should not run for ever.

Each refinement may involve introducing new variables to the model. This usually results in extending abstract events or adding new events to the model. It is also possible to replace abstract variables by newly defined concrete variables.

Refinement of a machine may consist of refining existing events, adding new events, and adding new variables and invariants. Refining an existing abstract event may have one of the following two forms: (1) Extending the abstract event with new parameters, guards, and actions (*horizontal refinement*) or (2) Modifying parameters, guards, and actions of the abstract event (*vertical refinement*). In the former, abstract parameters, guards, and actions do not change. In the latter however, replacing and adding new parameters, guards, and actions are allowed. In both cases guards of concrete event should be stronger than guards of abstract event.

Concrete variables are connected to abstract variables through *gluing invariants*. A gluing invariant associates the state of the concrete machine with that of its abstraction. All invariants of a concrete model including gluing invariants should preserve for all events.

In Event-B, events may be seen as transition relations from one state to another. If S is the state space of the the abstract model, then abstract event a_i is a relation from S to S. Similarly, if T is the state space of the concrete model then the concrete event c_i is a relation from T to T. If v is the set of abstract state variables and w is the set of concrete state variables then:

$$S = \{v \mid I(v)\}$$

$$T = \{w \mid \exists\, v \,.\, (I(v) \;\wedge\; J(v, w)\}$$

where $I(v)$ and $J(v, w)$ denote the abstract and concrete model invariants, respectively. Formally, the data refinement relation R between a_i and c_i is as follows:

$$R = \{w \mapsto v \,|\, I(v) \ \wedge \ J(v, w)\}$$

R formalises the gluing invariants between abstract and concrete states. In this thesis, we use $\sqsubseteq_R$ to denote data refinement through relation R and $a_i \sqsubseteq_R c_i$ is read as "c_i is a data refinement of a_i".

2.2.3 Before-After Predicates

In Event-B there is a predicate associated with every event action called *before-after predicate*. A before-after predicate (BA) denotes the relationship between the value of the variable which is updated by the action right before and right after the execution of that action.

A before-after predicate of an action can be simply obtained by priming the left-hand side variable of the assignment and substituting the assignment symbol (:=) with equality symbol (=). For example, assume that $x := x + 1$ is an event action. The associated before-after predicate with this action is as follows:

$$x' = x + 1$$

where x' represents the value of the variable after the execution of the action and x represents the value of the variable before the execution of the action. Before-after predicates are used in the construction of the proof obligations (See Section 2.2.4).

2.2.4 Proof Obligations in Event-B

For proving different aspects of a model a number of proof obligations are defined [Abr10]. In order to verify the model all proof obligations should be discharged. This can be done with the help of automatic provers or interactively. The different kinds of proof obligations are discussed in this section.

- **Invariant Preservation (INV)**: This proof obligation ensures that each invariant in a machine is preserved by all events in that machine.
- **Feasibility (FIS)**: This is to ensure that each non-deterministic action is feasible, i.e. there is an after state satisfying the corresponding before-after predicate.

- **Guard Strengthening (GRD)**: This proof obligation is to guarantee that the guards of a concrete event are stronger than the guards of its abstract event. This means that whenever the concrete event is enabled, the abstract event is enabled as well.
- **Simulation (SIM)**: Proving this proof obligation guarantees that each action in an abstract event is correctly simulated (i.e. the actions preserve the gluing invariant) in the corresponding refined event. This means that execution of a concrete event is not contradictory to its abstract event.
- **Numeric Variant (NAT)**: This proof obligation ensures that under the guards of each convergent event a proposed numeric invariant is indeed a natural number.
- **Finite Set Variant (FIN)**: Similar to the previous one, this proof obligation ensures that under the guards of each convergent event a proposed set variant is a finite set.
- **Variant (VAR)**: This proof obligation guarantees that each convergent event decreases a proposed variant expression.
- **Theorem (THM)**: This proof obligation is to guarantee that a theorem (defined in a context or machine) is provable.
- **Well-definedness (WD)**: This proves that guards, actions, invariants, axioms, theorems, and variants are *well-defined*. A well-defined expression is an expression which its definition assigns it a unique interpretation or value. If an expression is not well-defined, it is ambiguous.

2.2.5 Merging Rules

Event-B is used to model various kinds of systems (including hardware, software and environment of operation). When a sequential program is developed in Event-B, once the development is finished, events need to be merged in order to obtain the final sequential program [Abr03]. An Event-B machine can be treated as a sequential program by constructing a loop whose body is non-deterministic choice of events (E_1,E_2,...,E_n):

$$\textbf{do } E_1 \square E_2 \square \ldots \square E_n \textbf{ od}$$

To construct the final deterministic sequential program from events, a number of *merging rules* are needed. Figure 2.2 illustrates two merging rules defined by Abrial [Abr10].

The first rule (M_IF) is defining a conditional statement. The second one (M_WHILE) is for defining a loop statement. These rules say that if there are two events like the left hand side of the $\rightsquigarrow$ symbol, then they can be merged to form a *pseudo-event* like

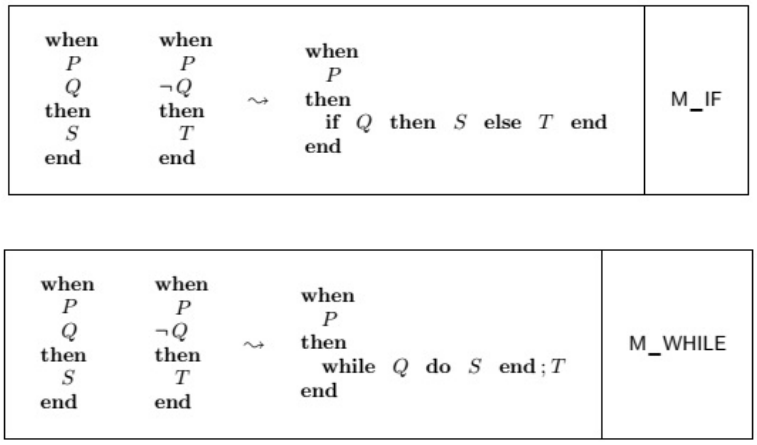

FIGURE 2.2: IF and WHILE Merging Rules (taken from [Abr10])

the one in the right hand side of the symbol. The antecedent events of both rules are same. To avoid confusion, there are some incompatible side conditions which make them distinct. The second rule requires that the first antecedent event appears as convergent *at one refinement level below that of second one*. In this way by providing a variant, loop termination can be proved. Also the first event must keep the common condition P[Abr10].

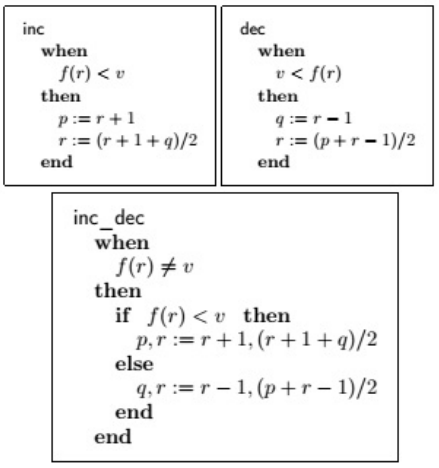

FIGURE 2.3: An Example of Event Merging (taken from [Abr10])

Figure 2.3 shows an example of event merging. Two events (*inc* and *dec*) are merged to form *inc_dec* event. To merge these two events, the first merging rule is applied.

2.2.6 Rodin Platform

Modelling in Event-B is facilitated by a tool called Rodin [ABH+10, ABHV06]. Rodin is an open source software which is built on top of the Eclipse IDE. It is an extensible and adaptable modelling tool. Various useful plug-ins have been developed in order to perform different tasks in Rodin. The ProB animator [LB08], the decomposition

plug-in [SPHB11], the code generator plug-in [EB11], and many other plug-ins are good examples of Rodin extensibility.

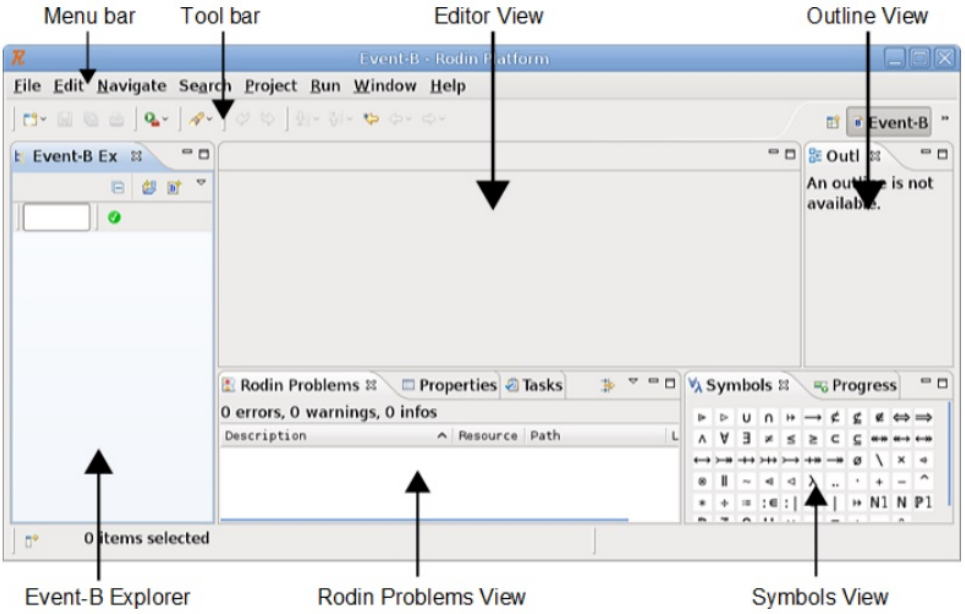

FIGURE 2.4: An Screenshot of Rodin GUI (taken from [JLPC12])

Rodin provides feed back about model text by detecting possible syntactic errors and produces error messages. One of the most important tasks which Rodin performs automatically is generating proof obligations. As mentioned in 2.2.4 proof obligations define what should be proved about the model. By trying to discharge proof obligations, developers can find possible errors in their design. Rodin applies automated provers to proof obligations in order to discharge them automatically. If a proof obligation is not discharged by automated tools, then Rodin provides facilities for interactive proving.

Figure 2.4 is a screenshot of Rodin platform taken from . A comprehensive and very useful guide on Rodin tool can be found in [ABHV06] and [JLPC12].

2.2.7 Theory Extension

To extend the built-in mathematical language of the Rodin tool, a feature called the *Theory Plug-in* has been developed to make the theory extension possible. A theory, which is a new kind of Event-B component, can be defined independently from a particular model and it is the means by which the mathematical language and mechanical provers may be extended [BM13]. Figure 2.5 illustrates the overall structure of Event-B theories.

A theory has a name and a number of type parameters. There may be a number of operators in a theory. A predicate operator defines a property on one or more expressions. A predicate operator may be infix or prefix. An expression operator is to form an expression from a number of expressions. New datatypes can be defined. A new datatype

theory *name*
type parameters $T_1, \ldots, T_n$

{ ⟨ *Predicate Operator Definition* ⟩
| ⟨ *Expression Operator Definition* ⟩
| ⟨ *Data Type Definition* ⟩
| ⟨ *Rewrite Rule* ⟩
| ⟨ *Inference Rule* ⟩ }

FIGURE 2.5: Overall Structure of Event-B Theories (taken from [BM13])

declaration defines a new type constructor together with constructor and destructor functions for elements of the type. A rewrite rule is used to rewrite an expression in order to facilitate the proof. An inference rule is a list of hypothesis and a consequent which is parametrised by one or more variables.

Independent theories can be reuse in different modelling projects. For example sequences that are not part of the core mathematical language of Event-B are introduced by the theory of sequences.

2.3 Hoare Logic

Hoare Logic is a formal approach for reasoning about correctness of programs introduced in late 1960's by C.A.R Hoare. In his famous paper "An Axiomatic Basis for Computer Programming"[Hoa69], Hoare states that for any given computer program all of its properties and all the consequences of its execution can be found out from its text *by means of purely deductive reasoning*. One of the important properties of a program is that whether the program serves its intended purpose. The intended behaviour of a program can be specified using general assertions about the values of the variables after the execution of the program. These assertions are called post-conditions. There are many cases in which the program can satisfy its post-conditions only if the variables have some specific values (conditions) before the program is initiated. These conditions can be specified with the same type of general assertions that are used to specify the values of the variables after the execution of a program. Assertions that specify the necessary values of the variables before execution of a program are called pre-conditions. If a pre-condition of a program is violated (i.e the pre-condition is not true) the outcome of the program is undefined. From a pre-condition P, a program C and a post-condition Q a *Hoare triple* can be formed as follows:

$$\{P\}\ C\ \{Q\} \tag{2.1}$$

The above triple is interpreted as follows: "if P is true just before the execution of program C then Q is true immediately after execution of C".

2.3.1 Rules of Hoare Logic

There are a number of axioms and inference rules for Hoare logic. This section intends to describe some of the important axioms and inference rules of Hoare logic. The axioms of Hoare logic are specified by *schemas* which can be instantiated. The inference rules of Hoare logic have the following form:

$$\frac{\vdash S_1, \ldots, \vdash S_n}{\vdash S} \tag{2.2}$$

In the above inference rule schema, $\vdash S$ is the conclusion and $\vdash S_1, \ldots, \vdash S_n$ are the hypotheses of the rule.

2.3.1.1 The Assignment Axiom

The Hoare assignment axiom has the following form:

$$\vdash \{P[E/V]\}\ V := E\ \{P\} \tag{2.3}$$

In above axiom V is any variable, E is any expression, P is any statement and $P[E/V]$ is the result of substituting E for all occurrences of V in the P. This axiom states that for any variable V and expression E, statement P is true after execution of command $V{:=}E$ if $P[E/V]$ is true before the execution. To illustrate this, assume there is a command as follows:

$$\mathrm{X} := \mathrm{X} + 1$$

For the statement $X = n + 1$ to be true after execution of the command, the following statement must be true before execution:

$$\mathrm{X} + 1 = \mathrm{n} + 1$$

So the following Hoare triple holds:

$$\{X = n\}\ X := X + 1\ \{X = n + 1\}$$

It is important to note that the Hoare assignment axiom is backwards from what you may expect. Using this axiom a precondition may be found based on the post-condition P and it is not used to discover a post-condition from the pre-conditions. There is a

forwards version of assignment axiom which is often called Floyd's assignment axiom [ST07]. The axiom has the following form:

$$\vdash \left\{P\right\}\ V := E\ \left\{\ \exists\, v.\ (V = E[v/V]) \wedge P[v/V]\right\} \tag{2.4}$$

In the Floyd assignment axiom the existentially quantified variable v is the value of V before execution of the assignment $V{:=}E$. The post-condition states that the value of V must be equal to the value of E evaluated in the state before execution of the program therefore because P was true before the execution it still holds for the value of v after execution.

2.3.1.2 Pre-condition Strengthening

The pre-condition strengthening rule allows a pre-condition to be replaced with a stronger one and has the following form:

$$\frac{\vdash P' \Rightarrow P,\ \vdash \{P\}\, C\, \{Q\}}{\vdash \{P'\}\, C\, \{Q\}} \tag{2.5}$$

2.3.1.3 Post-condition Weakening

Similar to the previous rule, the post-condition weakening rule allows to replace a post-condition with a weaker one and has the following form:

$$\frac{\vdash \{P\}\, C\, \{Q\},\ \vdash Q \Rightarrow Q'}{\vdash \{P\}\, C\, \{Q'\}} \tag{2.6}$$

2.3.1.4 The Composition Rule

If there are programs C_1 and C_2 and the post-condition of the former (Q) is the pre-condition of the latter and these programs are executed sequentially (C_1;C_2) then the rule of composition with the following form can be applied to them:

$$\frac{\vdash \{P\}\, C_1\, \{Q\},\ \vdash \{Q\}\, C_2\, \{R\}}{\vdash \{P\}\, C_1\, ; C_2\, \{R\}} \tag{2.7}$$

2.3.1.5 The Conditional Rule

The conditional rule of Hoare logic has the following form:

$$\frac{\vdash \{B \wedge P\}\, C_1\, \{Q\},\ \vdash \{\neg B \wedge P\}\, C_2\, \{Q\}}{\vdash \{P\}\, \texttt{IF}\ B\ \texttt{THEN}\ C_1\ \texttt{ELSE}\ C_2\, \{Q\}} \tag{2.8}$$

2.3.1.6 The WHILE Rule

If an assertion P is true before and after execution of program C then it is called an *invariant* of the program. If we have a while loop with test condition B and P is an invariant of the body of the loop, then the `WHILE` rule states that the P is an invariant for the whole loop. An invariant which holds for the whole loop is called a *loop invariant*.

$$\frac{\vdash \{P \wedge B\}\, C\, \{P\}}{\vdash \{P\}\, \texttt{WHILE}\ B\ \texttt{DO}\ C\ \{P \wedge \neg B\}} \tag{2.9}$$

From the above rule it can be seen that the loop condition must be false when the loop is terminated. The above rule is enough for partial correctness. Partial correctness means that provided that the algorithm is terminated the post-conditions are established. Total correctness, however, requires the proof of termination as well. The following is the WHILE rule for total correctness:

$$\frac{\vdash \{P \wedge B \wedge (E = n)\}\, C\, \{P \wedge (E{<}n)\},\ \vdash P \wedge B \Rightarrow E \geq n}{\vdash \{P\}\, \texttt{WHILE}\ B\ \texttt{DO}\ C\ \{P \wedge \neg B\}} \tag{2.10}$$

In the above rule E is a *loop variant*. A variant is an integer-valued expression that decreases on each iteration of the loop. Variable n is an auxiliary variable specifying the fact that E decreases. The rule has an extra hypothesis that ensures the non-negativity of the variant.

2.4 Refinement Calculus

The refinement calculus, introduced by Back, Morgan, and Morris independently [BW98, Bac88, Mor87, MRG88], is a formal framework based on higher order logic and lattice theory. It unifies the stepwise refinement and program transformation together with invariant-based approaches in order to use them as a tool for derivation of correct programs in a stepwise manner [Bac88]. Basically, the calculus provides a framework to answer the following two questions formally: 1) is a program correct with regard to a given specification, and 2) how a program can be refined (improved) while the correctness is preserved [BW98].

The refinement calculus extends the guarded command language [Dij75] of Dijkstra with the notion of *contracts* which is a generalisation of programs and specifications where refinement is a relation between contracts. In the refinement calculus specifications

are considered as program statements, which are not necessarily executable. However specifications define some properties on the program state which should be satisfied [Bac88].

A refinement relation between two programs S and S' is considered as correct if S' satisfies all the specification that S satisfies. If S is a program that is correctly refined by program S' then $S \sqsubseteq S'$ denotes the refinement relation. In formal words, if $S \sqsubseteq S'$ and $\{p\}\, S\, \{q\}$ holds then $\{p\}\, S'\, \{q\}$ also holds where p is a pre-condition and q is a post-condition for S and S'.

Using the refinement calculus, program derivation starts with a high-level (and probably non-executable) specification. This specification is then refined by program constructs where there still may be some parts that are not implemented. This process continues until all the unimplemented specifications are replaced by executable program text. The refinement calculus provides laws for replacing non-deterministic specification statements with deterministic program structures (sequential composition, if-statements and while-statements). In this thesis we use the refinement calculus mainly in Chapter 6 to define our schedule refinement rules.

2.5 Design by Contract

Design by Contract is a design approach for developing verified software introduced by Meyer [Mey02]. The approach extends program text with specification constructs. Method pre- and post-conditions, loop invariants, and assertions are among the specification constructs and are referred to as *contracts*. A method is usually annotated by pre- and post-conditions to specify its intended behaviour. The idea is that if a method p is called in a state that its precondition *pre(p)* is true then after execution of procedure p, the postcondition *post(p)* will be true [AFPdS11b]. This idea is implemented by some programming languages by embedding contracts in their code. Examples of these programming language and approaches are Eiffel [Mey92], Dafny [Lei10], JML [LBR06], VCC [DMS+09], Spec# [BLS05] and Spark [Bar97].

2.6 Dafny

The purpose of this section is to introduce the Dafny language [Lei10] and its static program verifier. Dafny is an imperative, class-based language [Lei10] which allows both strong and weak typed variables. Dafny implements the verification method of Hoare logic where a program can be specified with pre- and post-conditions. In the Dafny language, pre and post-conditions are influenced by the Eiffel language [Mey92] and the idea of *design-by-contract* [Mey02]. Dafny is an object-oriented programming

language with generic classes and allows creation of objects which gives rise to pointers [LM10]. Despite the fact that Dafny is a class-based language, it does not support subclasses and inheritance. However, there is a built-in **object** type that is a supertype of all class types. Dafny supports inductive datatypes and has its own specification constructs. Standard pre- and post-conditions, framing construct (See Section 2.6.1.8) and termination metrics (See Section 2.6.1.6) are included in the specifications. In this thesis we call these specification constructs code contracts. The language also offers recursive functions, sets, sequences and some other features to support specification. Dafny allows the definition of **ghost** variables. A ghost variable is a variable that is used by the Dafny verifier and ignored at run time. A ghost variable is used for specification purposes only and does not appear in any part of the implementation. Specifications and ghost variables are omitted by the compiler and are used just during the verification process.

The Dafny verifier attempts to verify different parts of a program locally (modular verification) and infer the correctness of the whole system from those locally verified parts. The Dafny verifier translates a Dafny program to an immediate verification language known as Boogie 2 [Lei08]. This is done in a way that the correctness of the generated Boogie program implies the correctness of the Dafny program. First-order verification conditions then are generated by the Boogie tool and are then passed to an SMT-solver which is called Z3 [DMB08].

The Dafny compiler generates code compatible with .NET platform via intermediate C# programs. The support for compilation and interfacing with .NET code is minimal.

The rest of this chapter is an overview to the Dafny language and some of its basic structures. Also few examples are discussed in more detail.

2.6.1 Basics of Dafny

2.6.1.1 Methods

A method in Dafny is a piece of imperative, executable code and is one of the basic units of a Dafny program. In Listing 2.1 an example of a simple Dafny program is presented. In this example method *C* has two arguments and returns an integer value.

```
module A{
        class B{

                var x: int;

                function isPositive (a:int) : bool
                {
                        if a >= 0 then true else false
                }
```

```
                method C(b:int, c:int) returns (d:int)
                requires b >= c;
                modifies this;
                ensures isPositive(d);
                {
                        x := b - c;
                        d := x;
                        assert x == d;
                }

                method testing()
                modifies this;
                {
                        var m := C(3,3);
                        assert m >= 0;
                        assert m == 0;
                }

        }
}
```

Listing 2.1: A Simple Example of a Dafny Program

Types of variables b, c and d are integer. Types are required for each parameter and return value. Types come after a colon following the name of parameters or return values. There can be single or multiple parameters or return values for each method.

In Dafny, assignment operator is ":= "while "== "is used for equality. To return a value from a method, the value is assigned to one of the return values. Return statements also can be used if one wants to return a value before the end of the code block is reached.

2.6.1.2 Pre- and Post-conditions

The verification power of Dafny originates from its annotations. Program behaviour can be annotated in Dafny using specification constructs such as pre- and post-conditions. The verifier then tries to prove that the code behaviour satisfies its annotations. This approach leads to producing correct code not only in terms of its syntax but also in terms of its behaviour.

There are several ways to annotate a program in Dafny. Methods' pre- and post-conditions are among them. Dafny uses the `ensures` keyword for post-condition declaration. A post-condition is always a boolean expression. The reason return values are given a name is now clear since in this way they can be easily referred to in a method's post-conditions. Each method can have more than one post-condition which can either be joined with boolean *and* (&&) operator or be defined separately using the `ensures` keyword. To declare a pre-condition the `requires` keyword is used. Like post-conditions multiple pre-conditions are allowed in the same style. Pre- and post-conditions are placed after method declarations and before method body.

2.6.1.3 Assertions

Assertions are another useful annotations in Dafny. They can be placed somewhere in the body of a method and state that what they are asserting is true when the program control reaches there. In Dafny the `assert` keyword is used for assertions. The purpose of assertions is to check that what you are expecting from your program in different points is true or not. Consider the method *testing* from the Listing 2.1 where m is a local variable. In this case, the return value of method C is assigned to variable m. There are two assertions in the method *testing*. Dafny can trivially prove the first one but it reports assertion violation for the second one. From the body of the method C it can easily be inferred that if both input values are equal, then the subtraction of them would be zero. The point here is that Dafny does not reason about the other methods by looking at their bodies. Dafny forgets about every other methods' body in the program except for the one which it is trying to prove. The only thing that Dafny knows about the other methods in the program, is their pre- and post-conditions. With this approach Dafny can be used to reason about each method in isolation and this leads to simplification of verification.

Now, if we add the following post-condition to method C then Dafny will be able to prove the second assertion in method *testing*:

```
ensures d == b - c
```

2.6.1.4 Functions

Functions in Dafny have very similar concept to mathematical functions. A Dafny function cannot write to memory and consists of just one expression. Functions are required to have only one unnamed return value. In Listing 2.1 a function (*isPositive()*) which takes a single integer and returns a boolean value can be seen.

This function evaluates the value of a and if it is positive the function returns `true`, otherwise `false`. The most important fact about functions is that they can be used directly in annotations. The other point is that unlike methods, Dafny does not forget about a function's body. It is also important to bear in mind that functions can only appear in annotations and they will not be part of the compiled code. Functions are just there to provide help for specification and verification. Yet someone might find it useful to have a function in a real code. If this is the case, then a function can be defined as *function method*.

2.6.1.5 Loop Invariants

Loop invariants are discussed in 2.3.1.6. A loop invariant is an expression that holds upon entering a loop and after each execution of the loop body. Consider the following method:

```
method test2(n: nat)
{
        var i := 0;
        while(i<n)
        {
                i := i + 1;
        }
        assert i == n;
}
```

From the body of the loop it is obvious that after exiting the loop, the value of variable i is equal to n and what is asserted in the method should be proved. However Dafny is unable to prove the assertion since it lacks a suitable loop invariant. The following loop invariant is needed here to make verification possible for Dafny:

```
method test2(n: nat)
{
        var i := 0;
        while(i<n)
        invariant 0<=i<=n
        {
                i := i + 1;
        }
        assert i == n;
}
```

The important challenge with loop invariants is to identify an invariant which is preserved by the loop and also is suitable for the loop post-condition.

2.6.1.6 Termination

Loop termination can be proved by Dafny. For this purpose Dafny uses `decreases` annotations. In some cases Dafny guesses the suitable decrease expression for the loop. If the loop condition is a comparison in the form of A >B then Dafny makes a guess like:

```
decreases A - B
```

If the verifier cannot verify the guessed decrease expression then a correct decreases expression should be made explicit by the programmer.

Dafny proves termination in two situations: loops and function recursion. In each of these situations the `decreases` annotation should be either explicitly expressed by the user or correctly guessed by Dafny. The `decreases` annotation is an expression which is

required to decrease in each iteration of a loop or recursive call. Dafny verifies two things about `decreases` annotation. First it verifies whether the expression is getting smaller and second that it is bounded. Dafny assumes that the bound of integer expressions are zero. The following example illustrates the use of `decreases` annotation:

```
while(0<i)
invariant 0 <= i;
decreases i;
{
        i := i - 1;
}
```

This annotation is used in functions in a similar way to the loops for proving termination of recursive calls.

2.6.1.7 Predicates

A predicate is a function which returns a boolean value and it has its own construct and reserved keyword `predicate`. It has the following form:

```
predicate abc(a: T)
{
...
}
```

The benefit of using predicates is that the code will be shorter and more readable as there is no need to write a long property many times.

2.6.1.8 Framing

In a Dafny program a method can freely read whatever it wants from the memory by dereferencing a variable or expression. But this is not the same for modifying the memory. A method is just allowed to modify those parts of the program that are specified by the `modifies` clause. Unlike the methods, functions are not allowed to modify the memory. Furthermore they are not allowed to read anything from the memory unless it is specified by the `reads` clause. Specifying that which parts of a program's states are allowed to be changed by a method or read by a function is called framing.

Framing is specified in object granularity for the purpose of simplification. Hence the `modifies` and `reads` clauses indicate sets of objects that a method can modify or a function can read. One important point about the aforementioned sets of objects is that in many cases they should be dynamically updated since the set of possible objects that a program can modifies may change dynamically. To address this issue Dafny uses the concept of *dynamic framing*. In this approach Dafny uses a set-valued ghost variable to specify the frame. To clarify the use of framing, consider the following example:

```
predicate sorted(a: array<int>)
requires a != null
reads a
{
        forall j,k:: 0<=j<k<a.length ==> a[j]<=a[k]
}
```

As it is shown, this is a predicate which determines whether array a is sorted or not. It is important to provide a sufficient `reads` clause for the predicates. In this case `reads` a is sufficient and should be provided, otherwise the verifier will complain and returns an error.

2.6.1.9 Modules and Refinement

Modules provide a way to group related parts of a program together in Dafny. An example of a module can be seen in Listing 2.1 where the class B is defined inside of the module A. A module can import other modules or can be imported by other modules with the `import` keyword. Modules are used for code re-use and it is also possible to separate implementation from interface by abstracting over modules. One module can be put inside another module in a nested fashion.

Dafny has some support for refinement which makes "programming in stages"possible; even though this feature is still experimental. With this feature a module can refine another module with the `refines` keyword. An example of refinement in Dafny is as follows:

```
module m0{
        method test_a (a: int) returns (b : bool)
        ensures a > 0 ==> b == true

        method test_b (a: int) returns (b : bool)
}

module m1 refines m0{
        method test_a...
        ensures a < 0 ==> b == false
        {
                if(a > 0)
                {
                        b := true;
                }
                else
                {
                        b := false;
                }
        }
}
```

In the example, module *m1* refines module *m0*. In this refinement, method *test_a* is refined and a new post condition and several lines of code are added to it. The "..."

in the method declaration, informs the verifier that it should take whatever is in the module that it is refining; in this case taking the signature and the postcondition of the method, from *m0*.

2.6.2 Dafny IDE

The Dafny program verifier is supported by an IDE [LW14] which is an extension of the Microsoft Visual Studio. The IDE provides *design-time feedback*. This means that the IDE runs the verifier in the background hence the user does not need to invoke the verifier.

Information such as variable types, loop invariants, and so on are made available by the Dafny IDE through *hover text*. The user can hover the mouse cursor over the program text to access to provided information. The tool also provides lots of verification error messages with useful information. This helps the user to debug the program easier.

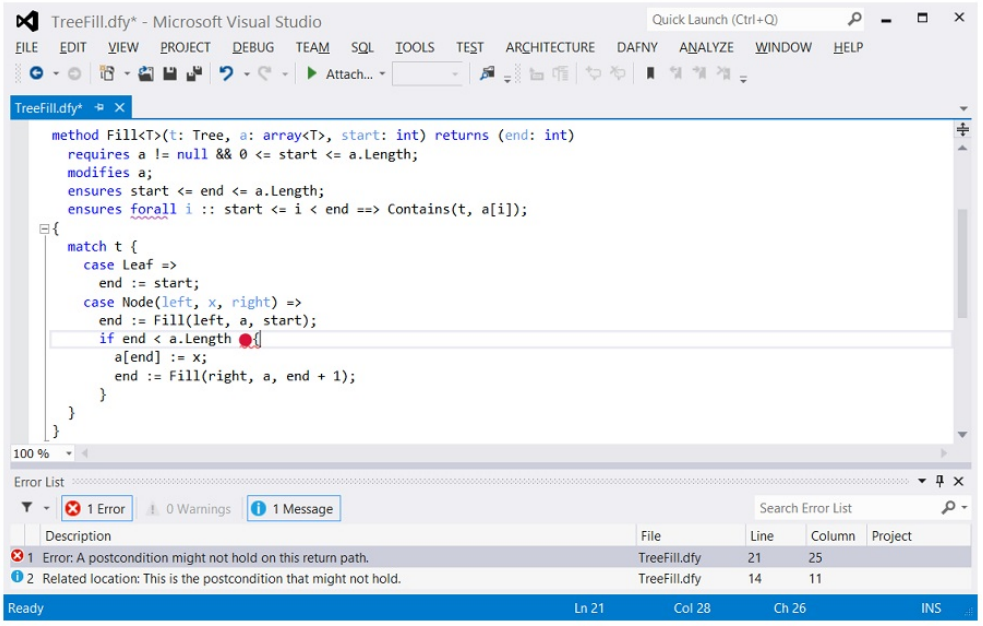

FIGURE 2.6: A Screenshot of Dafny IDE (taken from [LW14])

There is a web-based version[1] of the Dafny verifier which can be run within a web browser. Also it is possible to run the verifier from command line. A screenshot of Dafny IDE can be seen in 2.6. More details about the IDE can be found in [LW14].

[1] http://rise4fun.com/dafny

2.7 Related Work to Development of Verified Programs Using Formal Modelling Languages

In this section we briefly review the related work to development of verified code with formal modelling languages. In particular, we discuss some of the work on code generation, linking constructive and analytical approaches, and control flow and algorithmic structure in Event-B.

Many research has been performed in order to generate executable program code from high-level formal models. However, there is very little research on generating code contracts from Event-B models. To the best of our knowledge, no research has been carried out in order to generate verifiable Dafny code from Event-B models.

EventB2Dafny [CLR12] is a Rodin plug-in for translating Event-B proof obligations to Dafny code. The aim of *EventB2Dafny* is to use the Dafny verifier as an external theorem prover for proving Event-B proof obligations. In order to do this, Event-B structures are formalised in Dafny as datatypes and their corresponding operators are defined as functions. The *EventB2Dafny* translates proof obligations to Dafny code with the aid of these formalisations of Event-B constructs. Constants are modelled with the use of 0-ary integer functions and axioms as boolean functions with a post-condition that introduces the axiom. Translation of theorems is similar to translation of axioms except that they are checked by the `assert` clause. Invariants are also modelled with boolean functions and can be asserted or assumed. As in Event-B all variables, constants, and carrier sets are implicitly non-null, this should be made explicit in Dafny. To do this, a non-nullness function that ensures that all constants and carrier sets are non-null, will be in the generated code. The plug-in supports the full Event-B syntax. We used the tool in order to translate proof obligations of our case studies in to Dafny code. But as the syntax of the Dafny language has been changed and new constructs are introduced to the language, the generated code by the plugin is not verifiable with the current version of Dafny verifier.

In [CRW13, RCn14] an approach for translating Even-B models to JML-specified [LBR06, BCC+05] Java code is presented. A Rodin plug-in called *EventB2JML* has been developed to automate the translation from Event-B models to Java specified code. The plug-in translates each Event-B machine to a JML-specified Java class implementation. Later, the implementation can be verified against the specification using existing JML verification tools. The plug-in translates an Event-B machine into a class. EventB2JML translates each Event-B event to a *guard* method containing the translation of the guards of the event and *run* method which contains translation of actions of the event. Only when the *guard* method returns true, can the *run* method be executed. EventB2JML implements Event-B models by producing a Java thread implementation for each event

and does not impose any control flow on events. Also at the code level invariants need to be verified again using a static verifier.

Méry and Monahan in [MM13] proposed a transformation technique from an Event-B specification to an executable algorithm. The approach is mostly based on the work of Méry on refinement-based development of algorithmic systems [Mér09]. The transformation has two steps: (1) transforming an Event-B specification to a concrete recursive algorithm and (2) transforming from the recursive algorithm to an optimized iterative version. In this approach, first, the problem should be stated as method contracts in the form of Spec# contracts. Then an Event-B machine called `PREPOST` with the same contract as the method's contracts is created. Then a machine called `PROCESS` refines `PREPOST` to generate a concrete specification that satisfies the contracts. Control information (in the form of events' guards) is added in this refinement. A recursive algorithm is then generated from `PROCESS` machine. The recursive algorithm is transformed to an iterative one which is annotated with method's contract. The generated code then should be verified by the Spec# static program verifier.

Abrial in [Abr03] proposed a method for developing sequential programs in an event-based approach. The approach is based on two *merging rules* for defining conditional statements and loops. Merging rules may be effective in many cases for derivation of sequential programs from Event-B models, however, they do not help with making the control flow in a model explicit. Due to this, using merging rules for automated code generation from Event-B models is not straightforward.

Hallerstede in [Hal09] presented the development and verification of the iterative Quicksort algorithm in Event-B and reported the benefits of using Event-B techniques like anticipated events and witnesses in developing sequential programs. Hallerstede and Leuschel in [HL12] report their experience in sequential program verification using Event-B. They presented the development and verification of models of two algorithms: the factorial algorithm and the Quicksort algorithm. They used Abrial's merging rules for deriving the final program structure and used program counters for modelling sequentiality. They noted that adopting different refinement strategies can lead to different program structures for the same problem. This means that the refinement strategy and the program structure are coupled, and this limits the practical application of Event-B for developing sequential programs.

Making the control flow for Event-B models explicit has been explored by others as well. Hallerstede in [Hal10] introduced a method for specifying structured models in Event-B. The notation that is used by the method drives the generation of necessary proof obligations and also specifies the structure of a model. The method has a corresponding graphical notation in order to help the modeller to understand the structure of the model more easily. The notation allows the modeller to make the control flow explicit however it does not support concrete branches and loops. Illiasov in [Ili10] extended

Event-B with expressions called *flows* specifying the ordering between events. A flow allows the modeller to specify sequential composition, loop and choice. The approach uses the refinement of traces as a definition of the refinement relation and it relies only on theorem proving. The aforementioned approaches do not provide rules for introducing program structures in stepwise refinement. Boström in [Bos10] proposed a method for introducing control flow constructs to Event-B. The method provides a set transformers based semantics for Event-B and a scheduling language for describing the control flow in a model. The scheduling language has control constructs like branch and loop however it lacks explicit conditions for those constructs and does not introduce any rules for stepwise refinement of the schedule. Schneider et al. in [STW10] proposed a combined method by using CSP to provide explicit ordering for an Event-B model. Fürst et al. in [FHB+14] presented an approach for generating sequential code from Event-B model. In order to facilitate code generation a scheduling language for specifying event ordering is provided. The approach uses explicit program counters and auxiliary events for proving the correctness of the scheduled model. The auxiliary events and program counters can be problematic since for a model with complex control structure we may end up with a large number of control events and invariants which are necessary for proof. Edmunds and Butler in [EB11] proposed an extension to Event-B in order to facilitate code generation. The control flow in the model should be made explicit prior to code generation using a scheduling language. The approach does not support incremental introduction of control flow. Fathabadi et al. in [FBR14] introduced event refinement structure (ERS). ERS is a graphical notation for explicit representation of the relationship between abstract events and new events in a refinement level. It also provides a way for imposing explicit sequencing between events. ERS is designed for distributed systems and does not target sequential programs.

Research for generating code from formal models is not limited to Event-B. There is much research in the literature focusing on derivation of executable code from formal models in languages like VDM [AP11] and Z [SA92].

VDM is supported by a tool called VDMTools which supports automatic C++ and Java code generation [FLS+08]. The code generator supports about 95% of the VDM language. Overture [LBF+10] is an open source platform for supporting the construction and analysis of VDM formal models. Jørgensen et al in [JLC15] introduced a code generation platform for VDM in the Overture tool. The platform transforms a model to an Intermediate Representation (IR) in order to make the task of code generation easier. Since the IR is independent form any programming language, it can be later used for generation of code in different languages.

Rafsanjani and Colwill in [RC93] and Fukagawa et al in [FHY94] presented an approach for structural mapping from Object-Z [DRS95] (an object-oriented extension to Z) to C++ code. The main purpose of the approach was to aide the developer in obtaining

code. In [NH12] an approach for animating Object-Z specifications using C++ is presented. This approach extends the aforementioned structural mapping by covering a larger subset of the Object-Z language. In [RZ06] an approach for mapping a subset of Object-Z specifications to skeletal Java classes is introduced. The approach tries to take advantage of code contracts in order to ensure the correctness of the generated code. However the contracts are encoded in the program code as condition of branches and no static analysis is performed. *Circus* [WC02, WC01] is a unified language based on Z, CSP and refinement calculus for specifying, designing and programming concurrent programs introduced by Woodcock and Cavalcanti. It combines Z and CSP in a way that is suitable for refinement. A number of tools have been developed to provide support for refinement and code generation for Circus. Sennett in [Sen92] presents a notation to demonstrate how a program written in Ada satisfies a specification written in Z. Hamie in [Ham04] explains an approach for translating OCL to JML-specified Java code. Nielsen and et al. in [NLL12] discuss the combination of VDM with executable code.

Chapter 3

Transforming Event-B Models to Simple Dafny Code Contracts

In the previous chapter we discussed Event-B and Dafny in detail. Event-B in its original form does not have any support for the implementation phase. On the other hand, Dafny has very little support for abstraction and refinement. In this chapter we present a tool-supported development method by linking two verification technologies: Rodin and Dafny. Our method provides a way for transforming Event-B models to simple Dafny code contracts (method pre- and post-conditions). Our combined methodology is beneficial for both Event-B and Dafny users. It makes the abstraction and refinement of Event-B available for generating Dafny specifications which are correct with respect to a higher level of abstract specification in Event-B and provides a framework in which Event-B models can be implemented and verified in a programming language.

Transformation of Event-B formal models to annotated Dafny method declarations is achieved through a set of transformation rules. Using the transformation rules, one can generate code contracts from Event-B models but not implementations. The generated code contracts must be seen as an interface that should be implemented by the user. The user provided implementation can be verified later against the generated annotations (pre- and post-conditions) using an automatic verifier to prove its correctness with regards to the high level Event-B specification. The transformation rules are validated by being applied to a number of case studies including a map and a stack abstract data type (See Chapter 4). We have developed a tool that is based on these transformation rules for generating Dafny code contracts from Event-B models (See Chapter 5).

The rest of this chapter is structured as follows: Section 3.1 demonstrates the core idea of this chapter through an example, Section 3.2 explains the translation of an Event-B model (including generic types, model variables, and invariants) to a Dafny class. Section 3.3 explains the transformation of events to Dafny method contracts and its proof of correctness. Finally, Section 3.4 summarises the chapter.

3.1 Event-B to Dafny: The Core Idea

This section illustrates the core idea of transforming an Event-B model to Dafny code contracts through an example. It also discusses some of the issues and restrictions associated with the transformation.

3.1.1 A Demonstration

Assume that we want to model the enqueue operation of a fixed-size queue abstract data type. The queue has the generic type $ELEM$. There are two possible outcomes for the enqueue operation: first, the queue is full and consequently no new element can be added or second, the queue is not full and new elements can be added. The Event-B model presented in Listing 3.1 specifies the queue and its enqueue operation. The generic type $ELEM$ is specified in the context ($queueContext$). The queue itself is modelled using sequences[1]. The aforementioned possible outcomes are specified by events $enqueue1$ and $enqueue2$ for the first and second cases, respectively.

CONTEXT $queueContext$
SETS $ELEM$

MACHINE $queueMachine$ SEES $queueContext$
VARIABLES q
INVARIANTS
  inv1: $queue \in seq(ELEM)$
  inv2: $seqSize(queue) \leq 5$
EVENTS
Event $INITIALISATION$
then
  act1: $queue := emptySeq$
End

Event $enqueue1$
any e
where
  grd1: $seqSize(queue) = 5$
  grd2: $e \in ELEM$
then
  skip
End

Event $enqueue2$
any e
where
  grd1: $seqSize(queue) < 5$
  grd2: $e \in ELEM$
then
  act1: $queue := seqAppend(queue, e)$

[1] Sequences are not part of the core Event-B mathematical language. They are defined by theory extension using Rodin Theory plug-in.

```
End
```

Listing 3.1: Abstract Specification of a Queue ADT

The same enqueue operation can be implemented in Dafny in a single *enqueue* method. The expected behaviour of the enqueue operation should be specified in method's pre- and post-conditions. Method declaration and specification of the operation can be seen in Listing 3.2.

```
1 class queue<ELEM>{
2         var queue:seq<ELEM>;
3
4         method enqueue(e: ELEM)
5         ensures |old(queue)|<5 ==> queue == old(queue) + [e]
6         ensures |old(queue)|==5 ==> queue == old(queue)
7 }
```

Listing 3.2: Dafny Method Contracts for the Enqueue Operation

In the above code variable *queue* is a sequence with the generic type $ELEM$ and is a field in class *queue* (Line 1-2). The two different expected outcomes are defined as post-conditions (Lines 5-6). Here we omit the implementation of *enqueue* method. Any implementation that can satisfy the post-conditions can be seen as a correct implementation of the operation *enqueue*

A high level look at both the Event-B model and Dafny code shows that there similarities between the two. For instance, the possible outcomes are modelled in Event-B as two events and in Dafny they are specified as two post-conditions. However a direct mapping between the two is not straight forward.

3.1.2 Mapping Restrictions

In the previous section we presented an Event-B model specifying the enqueue operation. We also provided a Dafny method and its contracts (method pre- and post-conditions) for the same operation. Although there are similarities between the two, transformation of an Event-B model to a Dafny method contracts is not a one-to-one mapping between events and methods, so to make it possible, clear transformation rules are required. The transformation rules that are presented in the rest of this chapter deal with the following restrictions:

- The mathematical language of Event-B is richer than Dafny. There are constructs in the Event-B language that have no correspondence in Dafny (e.g. relations). Also the Event-B language can be extended with new constructs using the Theory Plug-in (See 2.2.7). Due to the difference between the mathematical languages of Event-B and Dafny, we can only transform a subset of the Event-B language to Dafny.

- Variable typing in Event-B is implicit. The implicit typing should be made explicit in the Dafny program.
- Event-B events can have parameters. However, there is no distinction between input and output parameters. In order to transform an event to a method we need to make the role of each parameter explicit.
- Different cases of a same operation are defined by multiple events where each event represents a distinct case. However, we usually want to be able to implement different cases of an operation within a single method. To achieve this we need to group the relevant events to a specific operation together. Event-B does not have any syntax for grouping events. This issue should be addressed.
- Guards and actions of an event are transformed to method pre- and post-condition. While some of the guards are used in the pre-conditions, some others contribute to the generation of the post-conditions. Since there is no distinction between event guards in Event-B, we need to define clear rules for categorisation of the event guards.

The rest of this chapter will present the rules and extensions that are needed to overcome the above limitations.

3.2 Transforming Event-B Machines to Dafny Classes

This section outlines the rules required for transforming an Event-B model to Dafny code contracts. The first translation rule is that an Event-B machine is translated to a Dafny class with the same name as the machine. Variables of the machine are transformed to Dafny class variables. Carrier sets defined in the context of the model are mapped to the class generic types. These transformations are almost one-to-one. Transformation of events to method contracts is discussed in the next sections.

3.2.1 Mapping Event-B Constructs to Dafny

To be able to transform an Event-B model, it should be refined to a level that there are only those Event-B mathematical operators in the model that have a counterpart in Dafny. This is essential for reducing the syntactic gap between Event-B and Dafny. When the aforementioned point in the refinement process is reached then a machine and its elements (e.g. variables, invariants,...) are translated to a Dafny class. Tables 3.1, 3.2, and 3.3 summarize the mapping between constructs of the core Event-B language and Dafny.

Operator	**Event-B**	**Dafny**
Sum	m + n	m + n
Difference	m - n	m - n
Product	m × n	m * n
Quotient	m / n	m / n
Remainder	m `mod` n	m % n

TABLE 3.1: Arithmetic Operators in Event-B and Dafny

Predicate	**Event-B**	**Dafny**
True	true	true
False	false	false
Conjunction	P ∧ Q	P && Q
Disjunction	P ∨ Q	P ‖ Q
Implication	P ⇒ Q	P==>Q
Equivalence	P ⇔ Q	P<==>Q
Negation	¬P	!P
Universal Quantification	∀z.P⇒Q	forall z :: P ==>Q
Existential Quantification	∃z.P⇒Q	exists z :: P ==>Q
Equality	P = Q	P == Q
Inequality	P ≠ Q	P != Q

TABLE 3.2: Predicates in Event-B and Dafny

Set Operator/Predicate	**Event-B**	**Dafny**
Empty Set	∅	{}
Set Comprehension	{z.P| F}	set z: T | P :: F
Union	A ∪ B	A + B
Intersection	A ∩ B	A * B
Difference	A ∖ B	A - B
Membership	a ∈ A	a in A
Non-Membership	a ∉ A	a !in A
Subset	A ⊆ B	A <= B
Not a Subset	A ⊈ B	!(A <= B)
Proper Subset	A ⊂ B	A <B
Not a Proper Subset	A ⊄ B	!(A <B)

TABLE 3.3: Set Operator/Predicates in Event-B and Dafny

In our transformation, we assume that basic Event-B types (i.e. integer, boolean and natural numbers) correspond to the Dafny primitive types. In addition to the core Event-B constructs presented in the above table, extensions to the core language may also be transformed to Dafny if the Dafny language has counterparts for them. An example for this is sequences that can be added to the Event-B language by Theory Plug-in.

3.2.2 Generic Types

Generic types in an Event-B model are defined using carrier sets in a context. In Dafny, generics are declared in angle brackets after the name of a class. The following example shows how generic types are defined in Dafny:

`class` *class_name*<T_1, T_2,...,T_n>{*...class body...*}

When a machine is translated to a Dafny class, all carrier sets which are defined in the context that is seen by that machine are translated as Dafny generics. Note that it is assumed that the context of the model only contains carrier sets.

3.2.3 Variables

In an Event-B model variables are declared in the *variables* part of a machine. Variable types are specified using *typing invariants* separately in *invariants* part of the machine. All machine variables are translated as class variables in Dafny. The type of variables in Dafny are inferred from the Event-B typing invariants.

3.2.4 Invariants

In Event-B there is no explicit distinction between different invariants. However, Event-B invariants can be categorised based on the property that they specify as follows:

- **Typing invariants:** A typing invariant is an invariant that declares the type of a variable. It has the general form of $v \in T$ where v is a machine variable and T is a primitive type or a generic type (carrier set defined in the model context) or a data type defined using theory plug-in (e.g. sequence).
- **Gluing invariants:** A gluing invariant relates the concrete variables to abstract variables. If an invariant is referring to both abstract and concrete invariants, then it is a gluing invariant.
- **Model invariants:** A model invariant expresses the properties of the model and only refers to concrete variables and is not identified as a typing invariant.

Typing invariants are used for variable declarations and preservation of typing invariants are checked implicitly by the Dafny type system. Gluing invariants are not translated to Dafny because we assume that the machine that is being translated is a data refinement of the abstract machine and none of the abstract variables are present in the refined

machine. Preservation of gluing invariants must be proved in Event-B. Only model invariants are translated to Dafny. The conjunction of all model invariants are placed in a Dafny predicate called *Invariants()*. It is explained later that how predicate *Invariants()* is used in pre-condition generation.

3.3 Transforming Events to Annotated Method Declarations

In the previous section we discussed how the declaration elements of an Event-B machine should be translated to Dafny class members. In this section we present the way in which machine events are transformed to Dafny method declarations and their contracts.

3.3.1 Event to Hoare Triple

As mentioned before, Dafny uses Hoare logic as the basis for verification. Here a simple transformation of an Event-B event to a Hoare triple is presented. This simple transformation is refined and used later for generation of Dafny code contracts. In Hoare logic a program is specified using a Hoare triple. Each Hoare triple has three parts, a precondition P, a program statement or series of statements S, and a postcondition Q:

$$\{P\}\ S\ \{Q\} \tag{3.1}$$

Each Event-B event has the following general form:

$$\text{Evt} \triangleq \textbf{any}\ \text{x}\ \textbf{where}\ \text{G(x,v)}\ \textbf{then}\ \text{v} := \text{E(x,v)}\ \textbf{end}$$

where v is the list of machine variables, x is the list of event parameters, *G(x,v)* is the conjunction of guards of the event and $v := E(x,v)$ is a set of assignments. The above event is enabled exactly when guard G is satisfied for some x:

$$\exists\, x.G(x,v) \tag{3.2}$$

If predicate 3.2 is true then event *Evt* can be executed. The following before-after predicate denotes the relationship between the value of the variables before and after execution of the event:

$$v = E(x, old(v))^2 \tag{3.3}$$

where $old(v)$ refers to the value of v before the execution of the event. Based on predicates 3.2 and 3.3 the following Hoare triple can be formed:

$$\{\exists x.G(x, v)\}\ impl\ \{v = E(x, old(v))\} \tag{3.4}$$

where *impl* is a placeholder for the (yet to be constructed) correct implementation of the method. The above Hoare triple is formed by treating the event guard as pre-condition. Our decision to treat guards as pre-conditions is based on the fact that the Dafny method contracts only support pre- and post-conditions. If a guard is not true it blocks execution of the event, however if a pre-condition does not hold it would not prevent the code from being executed and only releases the code from its obligations (i.e. satisfying the post-conditions upon termination). Because of this, we have to assume that the caller of the Dafny method ensures that the method pre-conditions are satisfied before execution of the method.

3.3.2 Method Constructor Statement

Recall the example given in Section 3.1.1. In that example two events *enqueue*1 and *enqueue*2 were used to model different cases of the *enqueue* operation. This illustrates the Event-B modelling style where different cases of some conceptual operation are represented as separate events. In contrast, in a programming language, it is usually preferred to have a single method implementing an operation. When we are transforming an Event-B model to Dafny the modeller should be able to group related events together. Event-B does not have any construct for event grouping. In addition to this, event parameters may have different implicit roles (e.g. input or output). However, Event-B does not have any support for making the role of an event parameter explicit. To make it possible for the modeller to group events together and make the role of event parameters explicit, we have introduced a new element to Event-B machines called the *method constructor statement*. A method constructor statement has the following form:

$$\texttt{method}\ mtd\_name(in_1,\ in_2,...)\ \texttt{returns}\ (out_1,\ out_2,...)\ \{Evt_1,\ Evt_2,...\}$$

Each method constructor statement has four parts: the name of the target method (mtd_name), a comma separated list of the method's input arguments ($in_1,\ in_2,...$), a

[2] The Event-B convention for before-after predicates is to use a primed variable to represent the value of that variable after the exectution of the event and unprimed variable to represent the value before the execution. To avoid confusion, here we use *old* operator to refer to the value of the variable before the execution and use unprimed variable to refer to the value after execution.

comma separated list of the method's output arguments (out_1, out_2,...) and a comma-separated list of event names placed between braces (Evt_1, Evt_2,...). The list of the input and output arguments can be empty. The listed events in the method constructor are expected to have parameters with the same names as the input and output parameters of the method constructor. An event may have other parameters that are not listed as input or output arguments. Events listed in a method constructor should represent different cases of a single operation. As an example, to transform Event-B events of the *enqueue* operation given in the queue example (See Section 3.1.1) we use the following method constructor statement:

$$\texttt{method}\ enqueue(e)\ \texttt{returns}\ ()\ \{enqueue1, enqueue2\}$$

An alternative way of grouping methods and making the role of parameters explicit is to use a naming convention for events and parameters. However there are couple of drawbacks to this approach. First, to group the events of an existing model, the modeller should change the name of events and parameters. This is not always easy because an event might have been further refined by another machine. If this is the case then the refined model should also be amended to comply with the new event and parameter names. Second, following a certain naming convention in a model requires more work on part of the modeller than the proposed approach (method constructor). Finally, a naming convention can potentially reduce the readability of an Event-B model.

3.3.3 Method Contract Generation

To transform Event-B events to Dafny method contracts, method constructor statements are used. Each defined method constructor statement gives rise to generation of a method declaration and its pre- and post-conditions. Pre- and post-conditions of each method are generated from the listed Event-B events. No implementation for the method is generated. Any method implementation that satisfies the generated contracts (method pre- and post-conditions) would be considered as a correct implementation of the Event-B events listed in the method constructor statement.

For generating pre- and post-conditions based on a method constructor statement two situations can be considered: (1) only one event is listed in the method constructor statement (2) more than one event is listed in the method constructor statement. Generation of method contracts for both of the mentioned scenarios are discussed in following sections.

3.3.3.1 Method Contract Generation From One Listed Event

Assume there is a method constructor with only one listed event as follows:

$$\texttt{method}\, EVT(x)\, \texttt{returns}(y)\, \{Evt\} \tag{3.5}$$

Each Event-B event (*Evt*) may have a number of parameters. Event parameters can be categorised with respect to the method constructor that the event is listed in as follows:

- **Input parameter (x)**: the parameter has input behaviour (receives a value from the environment of the machine) and is defined as an input parameter in the method constructor
- **Output parameter (y)**: the parameter has output behaviour (returns a value to the environment of the machine) and is defined as an output parameter in the method constructor
- **Internal parameter (z)**: the parameter is a local variable to the event and is not defined as input/output parameter in the method constructor

Input and output parameters are listed in the method constructor and they are used as a method's input or return arguments. Parameters that are not listed are treated as internal parameters. It is explained later how internal parameters are dealt with. Event *Evt* can be represented as follows:

Evt $\triangleq$ **any** x, y, z **where** P(x, y, z, v) **then** v := E(x, y, z, v) **end**

A number of pre-conditions should be defined for each method in Dafny to specify the conditions that must be true before a method is called. Pre-conditions are generated from invariants and event guards. As was mentioned previously, the conjunction of model invariants is translated to a Dafny predicate called *Invariants*. Predicate *Invariants* should be a pre-condition for all generated Dafny methods. The reason for this is that from the Event-B model, it is expected that invariants are true before execution of each event therefore it can be expected that invariants are true before execution of each method as well.

With respect to method constructor 3.5, we can categorise event guards. The category of a guard is either recognised based on its syntactic form or the variables it refers to. Different guard categories are as follows:

- **Typing guards (GT):** A typing guard declare the type of an event's input/output parameter. A typing guard only refers to a single input/output parameter. It has the general form of $p \in T$ where p is the input/output parameter and T is either a primitive type or a generic type (carrier set defined in the model context).

- **Output guards (GO):** An output guard determines the value of output parameters. The guard refers to output parameters and may also refer to other parameters and state variables as well.
- **Internal guards (GI):** An internal guard is a guard that refers to an internal parameter and does not refer to an output parameter but may refer to input parameters.
- **Method guards (GP):** A method guard is a guard that only refers to input parameters and class machine variables and not to internal or output parameters.

Each guard should fall into one of the above categories. For generating pre-conditions from event guards we only consider method guards. Note that a method guard only refers to machine variables and input parameters. We elaborate the guard of *Evt* into these categories as follows:

```
Event Evt
any x, y, z
where
    GT(x)
    GT(y)
    GP(x, v)
    GI(x, z, v)
    GO(x, y, z, v)
then
    v := E(x, y, z, v)
End
```

Recall predicate 3.2. Event *Evt* is enabled exactly when the predicate 3.2 is true. This predicate can be represented w.r.t above event as follows:

$$\exists\, x, y, z.GT(x) \wedge GT(y) \wedge GP(x, v) \wedge GI(x, z, v) \wedge GO(x, y, z, v) \tag{3.6}$$

Predicate 3.6 can be used for generation of the method pre-condition. We want to have a pre-condition that if it holds, guarantees that the event is enabled. Predicate 3.6 is complicated so it can be simplified. We know that the value of an input parameter (x) is determined by the environment of the event (the caller of the method, in a programming language), hence, it should be treated as a constant which its value is sent to the event (method) by the environment (caller) of the event (method). We decompose predicate 3.6 as follows:

$$GT(x) \wedge GP(x, v) \tag{3.7}$$

$$GT(x) \wedge GP(x,v) \Rightarrow \exists y,z.GT(y) \wedge GI(x,z,v) \wedge GO(x,y,z,v) \tag{3.8}$$

Predicates 3.7 and 3.8 together imply 3.6. Predicate 3.7 is used as a pre-condition for the method. This means that it is the method caller responsibility to make sure that predicate 3.7 holds before the execution of the method. Predicate 3.8 represents constraints on output and internal parameters. This can become a feasibility proof obligation that is verified within Event-B (See 3.3.4.1). Thus the event is enabled when the method pre-condition holds. We also include model invariants ($I(v)$) in pre-conditions of the method. Theoretically, this is not needed because once a class is instantiated and initialised, if all the methods (including initialisation method) satisfy their post-conditions then we know that invariants are preserved (this is proved at the Event-B level) and there is no need to force the caller of a method to verify that the invariants are preserved when an specific method is being called. However, for practical reasons we incorporate model invariants in the pre-condition because the Dafny verifier tries to prove some properties like index out of bounds by default and in some cases it needs some context information (invariants) to verify them. The conjunction of predicate 3.7 and the model invariants forms the pre-condition generated based on the given method constructor (3.5):

$$I(v) \wedge GT(x) \wedge GP(x,v) \tag{3.9}$$

In Dafny, the desirable value of the variables after execution of the method and the method return value (value of the output arguments) are specified in the method post-conditions. In Event-B, the event output guards specify the desirable value of the output parameters. Also the execution of the event establishes the before-after predicates of actions. Output guards and before-after predicates of actions depend on internal variables (z) so we incorporate z as an existentially quantified variable constrained by internal guards. Also, if a parameter is listed in a method constructor as an output parameter it should be treated as a free variable that its value is determined by the body of the method. Based on this, the post-condition for the method defined by method constructor 3.5 is as follows:

$$GT(y) \wedge (\exists z.GI(x,z,v) \wedge GO(x,y,z,v) \wedge v = E(x,y,z,old(v))) \tag{3.10}$$

The method constructor 3.5 gives rise to the generation of a Dafny method declaration and contracts using predicates 3.9 and 3.10:

`method` Evt (x : T) `returns`(y : R)
`requires` $I(v) \wedge GP(x,v)$
`ensures` $\exists z.GI(x,z,v) \wedge GO(x,y,z,v) \wedge v = E(x,y,z,old(v))$

T and R in the above method declaration are the type of input and output parameters. They are determined by typing guards.

3.3.3.2 Method Contract Generation From n Listed Events

Assume there is a method constructor with n listed events as follows:

$$\texttt{method}\ EVT(x)\ \texttt{returns}(y)\ \{Evt_1,..,Evt_n\} \tag{3.11}$$

We assume that all listed events in the above method constructor have x and y as their parameters. Also each listed event may or may not have a number of internal parameters $(z_1...z_n)$.

With respect to method constructor 3.11, we can categorise event guards as follows:

- **Typing guards (GT):** A typing guard declares the type of an event's input/output parameter. A typing guard only refers to a single input/output parameter. It has the general form of $p \in T$ where p is the input/output parameter and T is either a primitive type or a generic type (carrier set defined in the model context).
- **Method guards (GP):** A method guard is a guard that is being shared between all listed events in a method constructor and do not refer to output or internal parameters.
- **Output guards (GO):** An output guard determines the value of output parameters. The guard refers to output parameters and may also refer to other parameters as well.
- **Internal guards (GI):** An internal guard is a guard that refers to an internal parameter and does not refer to an output parameter but may refer to input parameters.
- **Case guards (GC):** Case guards are the guards that make the enabling condition of each listed event distinct from other listed events and only refer to input parameters and machine variables.

The above categorisation generalises the case where a method has a single event presented in the previous section. If there is only one listed event in a method constructor then GC will be empty. With regards to the above categorisation, listed events in method constructor 3.11 can be represented as follows:

Event Evt_1		**Event** Evt_n
any x, y, z_1		**any** x, y, z_n
where		**where**
$GT(x)$		$GT(x)$
$GT(y)$		$GT(y)$
$GP(x, v)$	...	$GP(x, v)$
$GC_1(x, v)$		$GC_n(x, v)$
$GI_1(x, z_1, v)$		$GI_n(x, z_n, v)$
$GO_1(x, y, z_1, v)$		$GO_n(x, y, z_n, v)$
then		**then**
$v := E_1(x, y, z_1, v)$		$v := E_n(x, y, z_n, v)$
End		**End**

We assume that all of the events that define a method have the same method guards and typing guards. They may differ in their case, local and output guards. From the above representation of the listed events and invariants of the model the following predicate can be generated and used as pre-condition for the Hoare triple related to method constructor 3.11:

$$I(v) \wedge GT(x) \wedge GP(x, v) \tag{3.12}$$

The reason for including invariants in pre-conditions was explained in the previous section. For generating post-conditions from events we use case guards, internal guards, output guards and before-after predicates of event actions. Each of the listed events in the method constructor gives rise to a predicate that specifies the behaviour of the event when its case guards are true. This predicate has the following form (where $i \in 1..n$):

$$\begin{aligned} &GT(y) \wedge \\ &(GC_{\mathrm{i}}(x, v) \Rightarrow (\exists\, z_{\mathrm{i}}.GI_{\mathrm{i}}(x, z_{\mathrm{i}}, v) \wedge GO_{\mathrm{i}}(x, y, z_{\mathrm{i}}, v) \wedge v = E_{\mathrm{i}}(x, y, z_{\mathrm{i}}, old(v)))) \end{aligned} \tag{3.13}$$

The above predicate specifies the post-condition of the method when the pre-conditions and $\mathrm{GC_i}$ hold. As mentioned before, a case guard makes the enabling condition of its respective listed event distinct from the other listed events. Case guards determine which event is enabled in the current state and therefore what is the expected outcome of the method. Based on method constructor 3.11 and predicates 3.12 and 3.13, then the following method declaration and contracts can be generated:

method Evt (x : T) **returns**(y : R)
requires $I(v) \wedge GP(x, v)$
ensures $GC_1[v/old(v)] \Rightarrow \exists\, z_1 :: GI_1[v/old(v)]$ && $GO_1[v/old(v)]$ && $v == E_1$
⋮
ensures $GC_{\mathrm{n}}[v/old(v)] \Rightarrow \exists\, z_{\mathrm{n}} :: GI_{\mathrm{n}}[v/old(v)]$ && $GO_{\mathrm{n}}[v/old(v)]$ && $v == E_{\mathrm{n}}$

T and R in the above method declaration are the type of input and output parameters. They are determined by typing guards. If there is only one listed event in a method constructor, then all non-typing and all non-output guards would be common guards and the set of case guards would be empty. The correctness of the typing guards is guaranteed by the Dafny type system.

3.3.4 Proof Obligations

To ensure that the translation is sound and the generated code contracts are implementable a number of proof obligations should be discharged. These proof obligations are discussed in this section.

3.3.4.1 Internal and Output Parameter Feasibility

We should make sure that the specification of an internal or output parameter is feasible and there exists a value that satisfies internal and output guards:

$$GT(x), GP(x, v), GC_i(x, v) \vdash \exists\, y, z_i.\, GT(y) \;\wedge\; GI_i(x, z_i, v) \;\wedge\; GO_i(x, y, z_i, v)$$

We split the above to have two separate proof obligations for internal parameter feasibility and output feasibility. For internal parameter feasibility, if there are n listed events in a method constructor and GC_i and GI_i (where $i \in 1..n$) are conjunction of case guards and conjunction of internal guards of ith listed event respectively, then we can generate n proof obligations with the following form:

$$GT(x), GP(x, v), GC_i(x, v) \vdash \exists\, z_i.\, GI_i(x, z_i, v)$$

Similar to the internal parameter feasibility proof obligation, if there are n listed events in a method constructor and GC_i, GI_i and GO_i (where $i \in 1..n$) are conjunction of case guards, conjunction of internal guards and conjunction of output guards of ith listed event respectively, then we can generate n proof obligations with the following form:

$$GT(x), GP(x, v), GC_i(x, v),\, GI_i(x, z_i, v) \vdash \exists\, y.\, GT(y) \;\wedge\; GO_i(x, y, z_i, v)$$

3.3.4.2 Disjointness

For pragmatic reasons we chose to generate a separate Dafny post-condition for each event listed in a method constructor. Since separate post-conditions are implicitly conjoined, the case need to be disjoint. This means that we remove any non-determinism arising from overlapping event guards prior to translation to Dafny contracts.

If there is a situation where more than one of the cases are available then the generated Dafny specification would not be implementable. This is due to the fact that there is no implementation that its execution under an initial condition can establish two different assertions about the program state upon termination. To avoid unimplementable specification, the modeller must make sure that case guards of all listed events are disjoint.

To prove the disjointness of the case guards a number of proof obligations must be discharged. If there are n events listed in the method constructor and GC_i where $i \in 1..n$ is the conjunction of all case guards of ith event then n sequent can be generated with the following form:

$$GT(x), GP(x,v), GC_i(x,v)$$

$$\vdash$$

$$\neg GC_1(x,v) \land ... \land \neg GC_{\text{i-1}}(x,v) \land \neg GC_{\text{i+1}}(x,v) \land ... \land \neg GC_n(x,v)$$

The size and number of proof obligations can be reduced by simplifying the above sequent:

$$GT(x), GP(x,v), GC_i(x,v) \vdash \neg GC_{\text{i+1}}(x,v) \land ... \land \neg GC_{\text{n}}(x,v)$$

3.3.4.3 Completeness

When there is more than one event listed in a method constructor, case guards are one of the forming components of each generated post-condition. There might be situations in which the generated post-conditions do not specify the intended behaviour of the method for all possible values specified by the method pre-conditions. This problem can be referred to as an incompleteness issue i.e. the specification is not complete. If there are n events listed in the method constructor and GC_i where $i \in 1..n$ is the conjunction of all case guards of ith event and GP is the conjunction of method guards, to avoid incompleteness issue, the following proof obligation should be proved:

$$GT(x), GP(x,v) \vdash GC_1(x,v) \lor ... \lor GC_n(x,v)$$

3.3.5 Semantics

We discussed in the previous sections how we can define a method with the help of method constructor statements. In this section we provide a semantics for the definition of a method. We use a subset of the guarded command language presented in Back's work in [BW98] and weakest pre-condition predicate transformers to give a common

semantics to events and definition of a method in Event-B and Dafny. First we introduce the following language which is a subset of the guarded command language defined in [BW98]:

$$S ::= \quad v := E(v) \;\mid\; v :\mid R(v, v') \;\mid\; \{g\} \;\mid\; [g] \;\mid\; S_1 \,;\, S_2 \;\mid\; S_1 \,\square\, S_2$$

$v := E(v)$ is a deterministic assignment which changes the value of v to $E(v)$. $v :\mid R(v, v')$ is a non-deterministic assignment which assigns a value v' to v such that the predicate $R(v, v')$ is satisfied. $\{g\}$ is an assertion where g is a predicate. An assertion is a requirement of the current state and if does not hold the command aborts. If g holds the assertion has no effect on the state and can be considered as `skip`. $[g]$ is a guard (assumption) where g is a predicate. Similar to assertions, guards do not change the state. If a guard holds it interprets as `skip` and if does not hold it blocks the execution. $S_1 \,;\, S_2$ is sequential composition and $S_1 \,\square\, S_2$ is a demonic choice between S_1 and S_2. The following weakest pre-condition predicate transformers can be given for above commands based on [BW98]:

$wp(v := E(v), Q) = Q[E(v)/v]$ (assignment)

$wp(v :\mid R(v, v'), Q) = \forall\, v'.\, R(v, v') \Rightarrow Q[v'/v]$ (non-deterministic assignment)

$wp(\{g\}, Q) = g \wedge Q$ (assertion)

$wp([g], Q) = g \Rightarrow Q$ (guard)

$wp(S_1 \,;\, S_2, Q) \;=\; wp(S_1, wp(S_2, Q))$ (sequential composition)

$wp(S_1 \,\square\, S_2, Q) \;=\; wp(S_1, Q) \wedge wp(S_2, Q)$ (external choice)

Previously, we described that we define a method by a group of events $(Evt_1 ... Evt_n)$. We interpret the definition of a method with n events as follows:

$$\{grd(Evt_1) \vee ... \vee grd(Evt_n)\} \,;\, (Evt_1 \,\square\, ... \,\square\, Evt_n) \tag{3.14}$$

where $grd(Evt_i)$ is the conjunction of guards of event Evt_i. The above definition states that there is an obligation on the caller to call the method when at least one of the events is enabled. If no event in the group is enabled then the behaviour of the method is undefined.

Now assume that we have the following n Event-B events:

$$Evt_1 \triangleq \textbf{where } P(v) \wedge G_1(v) \textbf{ then } v := E_1(v) \textbf{ end}$$

$$\vdots$$

$$Evt_n \triangleq \textbf{where } P(v) \wedge G_n(v) \textbf{ then } v := E_n(v) \textbf{ end}$$

An event Evt_i (where $i \in 1..n$) can be presented in the guarded command language as follows:

$$[P(v) \wedge G_i(v)]\,;\, v := E_i(v) \tag{3.15}$$

If $P(v)$ is the common guard of the grouped events and $G_i(v)$ is the conjunction of all other guards of event Evt_i then we can have:

$$\begin{aligned}
&\{grd(Evt_1) \vee ... \vee grd(Evt_n)\}\,;\,(Evt_1 \,\square\, ... \,\square\, Evt_n)\\
&=\\
&\{(P(v) \wedge G_1(v)) \vee ... \vee (P(v) \wedge G_n(v))\}\,;\,(Evt_1 \,\square\, ... \,\square\, Evt_n)\\
&=\\
&\{P(v) \wedge (G_1(v) \vee ... \vee G_n(v))\}\,;\,(Evt_1 \,\square\, ... \,\square\, Evt_n)\\
&= \{\text{completeness property (see Section 3.3.4.3)}\}\\
&\{P(v)\}\,;\,(Evt_1 \,\square\, ... \,\square\, Evt_n)
\end{aligned} \tag{3.16}$$

Now we give a similar definition to the generated contracts in the language introduced earlier in this section. Using the rules defined in the previous sections, a group of n events $Evt_1...Evt_n$ is transformed to Dafny method pre- and post-conditions (pre-condition $P(v)$ and post-condition $G_1(v) \Rightarrow v = E_1(v) \wedge ... \wedge G_n(v) \Rightarrow v = E_n(v)$). If there is an implementation *impl* that satisfies the generated contracts then the following Hoare triple holds:

$$\{P(v)\}\ impl\ \{G_1(v) \Rightarrow v = E_1(v) \wedge ... \wedge G_n(v) \Rightarrow v = E_n(v)\}$$

The above Hoare triple tells us how *impl* changes state v. The above generated contracts are represented in the guarded command language as follows:

$$\{P(v)\}\,;\, v :|\, G_1(v) \Rightarrow v' = E_1(v) \wedge ... \wedge G_n(v) \Rightarrow v' = E_n(v) \tag{3.17}$$

where v' is the value of v after execution of *impl*.

3.3.6 Refinement Proof

The previous section provided a common semantics for the grouped events in Event-B and implementation *impl* satisfying the generated contracts. The given semantics for the grouped events is:

$$\{P(v)\}\,;(Evt_1 \,\square\, ... \,\square\, Evt_n) \tag{3.18}$$

where $Evt_1, ..., Evt_n$ are Event-B events in the following form:

$$Evt_1 \triangleq \textbf{where } P(v) \wedge G_1(v) \textbf{ then } v := E_1(v) \textbf{ end}$$

$$\vdots$$

$$Evt_n \triangleq \textbf{where } P(v) \wedge G_n(v) \textbf{ then } v := E_n(v) \textbf{ end}$$

The given semantics for the implementation satisfying the generated contracts is:

$$\{P(v)\}\,;v :|\, G_1(v) \Rightarrow v' = E_1(v) \wedge ... \wedge G_n(v) \Rightarrow v' = E_n(v) \tag{3.19}$$

where v' is the value of v after execution of *impl*.

In this section we prove that the provided semantics for the grouped events (3.18) and its implementation (3.19) are equal and therefore the implementation is considered to be a correct refinement of the abstract grouped events:

$$\{P(v)\};(Evt_1 \,\square\, ... \,\square\, Evt_n) \;=\; \{P(v)\};v :|\, G_1(v) \Rightarrow v' = E_1(v) \wedge ... \wedge G_n(v) \Rightarrow v' = E_n(v) \tag{3.20}$$

To prove the above, we use the following rule from [BW98]:

$$(v :|\, R_1) \,\square\, ... \,\square\, (v :|\, R_n) \;=\; v :|\, R_1 \vee ... \vee R_n \tag{3.21}$$

and the following rule:

$$[G]\,; v := E(v) \;=\; v :|\, G \wedge v' = E(v) \tag{3.22}$$

Proof. Soundness of rule (3.22) can be proved as follows:

$[G]\,; v := E(v)$

$=$ {Definition of assignment}

$$
\begin{aligned}
&[G]\,;\, v :|\, v' = E(v) \\
&= \{\text{Definition of assumption through relational assignment}\} \\
&(v :|\, G \wedge v' = v)\,;\,(v :|\, v' = E(v)) \\
&= \{\text{Composition of relational assignments}\} \\
&v :|\, G \wedge v' = E(v)
\end{aligned}
$$

□

And finally we show (3.20) holds:

Proof.

$$\{P(v)\}\,;\,(Evt_1 \,\square\, ... \,\square\, Evt_n)$$

$= \{\text{definition of } Evt_i\}$

$$\{P(v)\}\,;\,([P(v) \wedge G_1(v)]\,;\, v \coloneqq E_1(v) \,\square\, ... \,\square\, [P(v) \wedge G_n(v)]\,;\, v \coloneqq E_n(v))$$

$= \{\text{common guard}\}$

$$\{P(v)\}\,;\,[P(v)]\,;\,([G_1(v)]\,;\, v \coloneqq E_1(v) \,\square\, ... \,\square\, [G_n(v)]\,;\, v \coloneqq E_n(v))$$

$= \{\{q\}\,;\,[q] = \{q\}\}$

$$\{P(v)\}\,;\,([G_1(v)]\,;\, v \coloneqq E_1(v) \,\square\, ... \,\square\, [G_n(v)]\,;\, v \coloneqq E_n(v))$$

$= \{\text{by (3.22)}\}$

$$\{P(v)\}\,;\,(v :|\, G_1(v) \wedge v' = E_1(v)) \;\square\; ... \;\square\; (v :|\, G_n(v) \wedge v' = E_n(v))$$

$= \{\text{by (3.21)}\}$

$$\{P(v)\}\,;\,(v :|\, (G_1(v) \wedge v' = E_1(v)) \vee ... \vee (G_n(v) \wedge v' = E_n(v)))$$

$= \{\text{converting to CNF}[3]\}$

$$\{P(v)\}\,;\,(v :|\, \bigwedge_{0 \le k < 2^n} (\bigvee_{0 \le i < n} \varphi_{k,i}))$$

where

[3] Conjunctive Normal Form

$$\varphi_{k,i} = \begin{cases} G_{i+1}(v), & \text{if } (k/2^i) \mod 2 = 0 \\ v' = E_{i+1}(v), & \text{if } (k/2^i) \mod 2 = 1 \end{cases}$$

=

$$\{P(v)\} ; (v :| \bigwedge_{\forall\, m\,.\, 0\le m<n \wedge k=2^m} (\bigvee_{0\le i<n} \varphi_{k,i}) \wedge \bigwedge_{\forall\, m\,.\, 0\le m<n \wedge k\neq 2^m} (\bigvee_{0\le i<n} \varphi_{k,i}))$$

Note that if $k = 2^m$ then:

$$\bigvee \varphi_{k,i} = \neg(\bigvee_{0\le i<n \wedge i\neq m} G_{i+1}(v)) \Rightarrow v' = E_{m+1}(v) = G_{m+1}(v) \Rightarrow v' = E_{m+1}(v)$$

=

$$\{P(v)\};(v :| \bigwedge_{\forall\, m\,.\, 0\le m<n \wedge k=2^m} (G_{m+1}(v) \Rightarrow v' = E_{m+1}(v)) \wedge \bigwedge_{\forall\, m\,.\, 0\le m<n \wedge k\neq 2^m} (\bigvee_{0\le i<n} \varphi_{k,i}))$$

$\Rightarrow$ {by lemma 3.1 (See below)}

$$\{P(v)\} ; (v :| \bigwedge_{\forall\, m\,.\, 0\le m<n \wedge k=2^m} (G_{m+1}(v) \Rightarrow v' = E_{m+1}(v)))$$

=

$$\{P(v)\} ; (v :| (G_1(v) \Rightarrow v' = E_1(v)) \wedge ... \wedge (G_n(v) \Rightarrow v' = E_n(v)))$$

□

Lemma 3.1. *If*

$$\bigvee_{i\in I} A_i = true$$

and C_i is either A_i or B_i then

$$\bigwedge_{i\in I} A_i \Rightarrow B_i \Rightarrow \bigvee_{i\in I} C_i$$

Proof. Suppose

$$\bigwedge_{i\in I} A_i \Rightarrow B_i$$

we show

$$\bigvee_{i \in I} C_i$$

We know

$$\bigvee_{i \in I} A_i$$

holds so $\exists\, i_0$ such that A_{i_0} holds. By assumption, we know that B_{i_0} also holds. Therefore C_{i_0} holds in either case. Therefore

$$\bigvee_{i \in I} C_i$$

holds. □

3.4 Summary

We have presented a method for transforming Event-B models to simple Dafny code contracts (i.e. method pre- and post-conditions). This approach provides a framework in which Event-B models can be implemented correctly in a sequential programming language. Our method provides a way for merging Event-B events in order to generate contracts for a single method which implements different cases. We have also proved that if the generated contracts are satisfied by an implementation in Dafny then it correctly refines the abstract Event-B events.

Chapter 4

Case Studies For Simple Dafny Code Contracts

In Chapter 3 a method for transforming Event-B models to simple Dafny code contracts was introduced. This chapter presents a number of case studies for the purpose of illustration and validation of the given method. Our case studies are a map abstract datatype (Section 4.1) and a stack abstract datatype (Section 4.2).

4.1 Map Abstract Datatype

A map (also called associative array) is an abstract data type which associates a collection of unique keys to a collection of values. Here we first present the Event-B model of the map ADT. After that, the necessary constructor statements are provided and then transformation of the model to Dafny code contracts is illustrated. Finally we present the Dafny implementation of the methods which are constructed by hand.

4.1.1 Event-B Model of the Map ADT

In the abstract level, the map is modelled as a partial function which links a key to a value. Types $KEYS$ and $VALUES$ are defined in the context as sets ($c0$, not shown here). Variable *map* is the only variable in this level and initialised with the empty map. The abstract specification of the map is as follows:

Machine $m0$ Sees $c0$
Variables map
Invariants $map \in KEYS \rightarrow\!\!\!\!\!\rightarrow VALUES$
Initialisation $map := \varnothing$

Event Add
any k, v
where
  grd1: $k \in KEYS$
  grd2: $v \in VALUES$
then
  act1: $map(k) := v$
End

Event $Remove$
any k
where
  grd1: $k \in dom(map)$
then
  act1: $map := \{k\} \lhd map$
End

There are two events in the abstract level: *Add* and *Remove*. Event *Add* has two parameters k and v. These parameters receive their values from the environment of the model. If k is a new key in the *map* then it will be added to the map by the event but if it already exists in the map the event updates the associated value to k. Event *Add* performs both add and update operations. Event *Remove* receives a key k from the environment of the model. The key must already exist in the map (this is enforced by guard *grd1*). The event removes k and its associated value from the map.

As mentioned in the previous chapter the model should be refined to a level that that uses only operators and datatypes that have counterparts in Dafny. Due to this, the abstract model of the event is refined. Machine *m1* is a refinement of the abstract model:

Machine $m1$ refines $m0$ Sees $c0$
Variables $keys, values$
Invariants
$keys \in seq(KEYS) \land values \in seq(VALUES) \land$
$seqSize(keys) = seqSize(values) \land$
$(\forall i, j.i \in 0..seqSize(keys) - 1 \land j \in 0..seqSize(keys) - 1 \land i \neq j \Rightarrow keys(i) \neq keys(j))$
Initialisation $keys := \varnothing, values := \varnothing$

Event $Add1$
refines Add
any k, v
where
  grd1: $k \in KEYS$
  grd2: $v \in VALUES$
  grd3: $k \notin ran(keys)$
then
  act1: $keys := seqPrepend(keys, k)$
  act2: $values := seqPrepend(values, v)$
End

Event $Add2$
refines Add
any k, v, i
where
  grd1: $k \in KEYS$
  grd2: $v \in VALUES$
  grd3: $i \in 0..seqSize(keys)$
  grd4: $seqElemAccess(keys, i) = k$
then
  act1: $values := seqElemUpdate(values, i, v)$
End

```
Event Remove
refines Remove
any k, i
where
 grd1: k ∈ KEYS
 grd2: k ∈ ran(keys)
 grd3: i ∈ 0..seqSize(keys)
 grd4: seqElemAccess(keys, i) = k
then
 act1: keys ≔ seqSliceToN(keys, i) seqConcat seqSliceFromN(keys, i + 1)
 act1: values ≔ seqSliceToN(values, i) seqConcat seqSliceFromN(values, i+1)
End
```

The abstract model (machine *m0*) is refined by machine *m1*. The abstract map is data refined by two sequences where one sequence stores keys and the other stores values. Event *Add* is refined by two events *Add1* and *Add2* to deal with two different cases. Event *Add1* will prepend a new key k to the *keys* sequence and value v to the *values* sequence. Event *Add2* modifies the value associated with an existing key k, in sequence of values. Event *Remove* is refined in this level to be able to remove an existing key k and its associated value from both sequences. Now, machine *m1* has only those Event-B constructs that have a Dafny counterpart, hence it can be transformed to Dafny contracts. Note that sequences are not part of the core Event-B language. The sequence operators and their Dafny counterparts are presented in Table 4.1. They are added through theory extension using the *Theory plug-in*. As explained in the previous chapter, machine variables are translated as class fields. Also model invariants are translated to a Dafny predicate called *Invariants*. The transformation of variables and invariants are not shown here.

Operator	**Event-B**	**Dafny**
Sequence	$seq(TYPE)$	$seq < TYPE >$
Empty Sequence	$\varnothing$	$[]$
Sequence Size	$seqSize(s)$	$\|s\|$
Element Access	$seqElemAccess(s, i)$	$s[i]$
Element Update	$seqElemUpdate(s, i, a)$	$s[i := a]$
Prepend	$seqPrepend(s, a)$	$[a] + s$
Slicing	$seqSliceToN(s, i)$	$s[..i]$
Slicing	$seqSliceFromN(s, i)$	$s[i..]$
Concatenation	$s1\ seqConcat\ s2$	$s1 + s2$

TABLE 4.1: Sequence Operators in Event-B and Dafny

4.1.2 Model Preparation and Transformation

As discussed in the previous chapter, to be able to transform events to Dafny contracts a number of method constructor statements should be provided. In the following subsections we explain how two methods, *Add* and *Remove*, are specified from the given model.

4.1.2.1 Method *Add*

Here we want to merge events *Add1* and *Add2* to be able to generate method *Add*. This method should deal with adding a new key and value or updating the associated value to an existing key. Therefore, we specify a constructor statement as follows:

$$\texttt{method}\ Add(k,\ v)\ \texttt{returns}\ ()\ \{Add1,\ Add2\}$$

According to the guard categorisation that we introduced in the previous chapter, the guards of events *Add1* and *Add2* must be categorised:

```
Event Add1
refines Add
any k, v
where
   grd1: k ∈ KEYS  ←Typing
   grd2: v ∈ VALUES  ←Typing
   grd3: k ∉ ran(keys)  ←Case
then
   act1: keys := seqPrepend(keys, k)
   act2: values := seqPrepend(values, v)
End
```

```
Event Add2
refines Add
any k, v, i
where
   grd1: k ∈ KEYS  ←Typing
   grd2: v ∈ VALUES  ←Typing
   grd3: i ∈ 0..seqSize(keys)  ←Internal
   grd4: seqElemAccess(keys, i) = k  ←Internal
then
   act1: values := seqElemUpdate(values, i, v)
End
```

Based on the guard categorisation, a case guard only refers to machine variables and input parameters. There is no guard falling into this category in event *Add2*. The determination of the guard classification is done using our tool (based on the definitions given in Section 3.3.3.2) not by the user. The tool is discussed in Chapter 5.

As can be seen there is no method guard and output guard in these events. Also event *Add1* does not have any internal guards (because there is no internal parameter) and event *Add2* does not have any case guard. As discussed in Section 3.3.4 a number of proof obligations must be proved based on the given constructor statement. Because event *Add1* does not have any output or internal parameter then there will not be any output or internal parameter feasibility proof obligations. Event *Add2* does not have output parameters nor output guards, but it has one internal parameter and two internal guards. This gives rise to generation of an internal parameter feasibility proof obligation as follows:

$$k \in KEYS \land v \in VALUES \vdash \exists i.i \in 0..seqSize(keys) - 1 \land seqElemAccess(keys, i) = k$$

The above sequent says that for every value of k and v there exists an i which is the index of the position that k is stored in the sequence of keys. This is not provable. A counter-example for this PO is a key that is not already in the sequence of the keys. To be able to prove the internal parameter feasibility proof obligation a new guard ($grd5$) must be added to the event:

Event $Add1$
refines Add
any k, v
where
  grd1: $k \in KEYS$ ←Typing
  grd2: $v \in VALUES$ ←Typing
  grd3: $k \notin ran(keys)$ ←Case
then
  act1: $keys := seqPrepend(keys, k)$
  act2: $values := seqPrepend(values, v)$
End

Event $Add2$
refines Add
any k, v, i
where
  grd1: $k \in KEYS$ ←Typing
  grd2: $v \in VALUES$ ←Typing
  grd3: $i \in 0..seqSize(keys)$ ←Internal
  grd4: $seqElemAccess(keys, i) = k$ ←Internal
  grd5: $k \in ran(keys)$ ←Case
then
  act1: $values := seqElemUpdate(values, i, v)$
End

Now the internal parameter feasibility PO is as follows:

$$k \in KEYS \land v \in VALUES, k \in ran(keys)$$

$$\vdash$$

$$\exists i.i \in 0..seqSize(keys) - 1 \land seqElemAccess(keys, i) = k$$

The above PO can be proved easily. Two more proof obligations (disjointness and completeness POs) should be proved for this transformation. Disjointness and completeness has the following forms:

- Disjointness:

$$k \in KEYS \land v \in VALUES, k \notin ran(keys) \vdash \neg k \in ran(keys)$$

- Completeness:

$$k \in KEYS \land v \in VALUES \vdash k \notin ran(keys) \lor k \in ran(keys)$$

Although the $Add2$ event new guard ($grd5$) is not required to prove refinement, it is necessary for discharging the above POs. Now that all essential POs have been discharged, the transformation can be performed. As explained before, each event in the method constructor gives rise to generation of an **ensures** clause. Figure 4.1 illustrates the transformation of events *Add1* and *Add2* to post-conditions of method *Add* in Dafny. The contract is generated automatically by our tool.

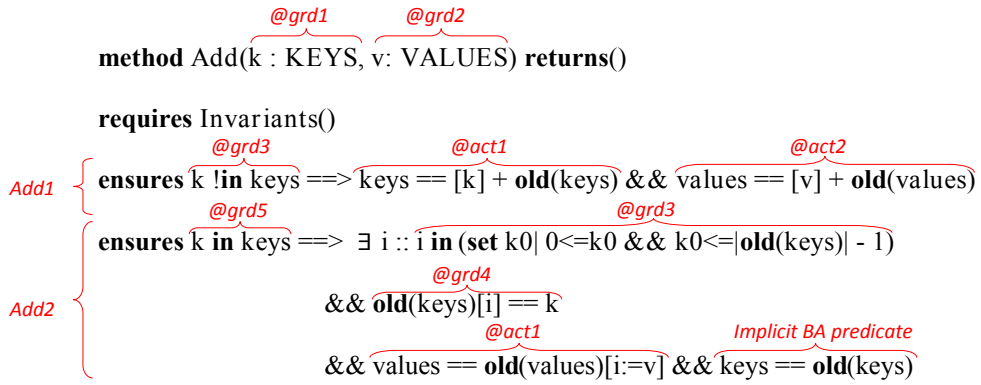

FIGURE 4.1: Method Contracts Generated from Events *Add1* and *Add2*

Note that event *Add2* does not change the value of the sequence *keys*. This is reflected by *keys == old(keys)* in the second `ensures` clause. Adding a predicate that denotes that the method does not change the value of a variable is done for all variables that remain unchanged by an event. *Invariant*() is a predicate including all model invariants. This predicate is generated automatically by the tool.

The expected implementation for methods whose contracts are generated from more than one event is typically in the form of `if...else` where the case guards of each event are the conditions for each branch. We also expect the existence of a loop in the implementation to search the *keys* sequence to find the index of an existing key.

4.1.2.2 Method *Remove*

In the previous subsection, we illustrated the transformation of events *Add1* and *Add2* to method *Add*. Here we presents the transformation of event *Remove* to a method with the same name. To do this we have to provide a method constructor:

$$\texttt{method}\ Remove(k)\ \texttt{returns}\ ()\ \{Remove\}$$

With respect to the above method constructor, the event guards of *Remove* event can be categorised as follows:

```
Event Remove
refines Remove
any k, i
where
 grd1: k ∈ KEYS  ←Typing
 grd2: k ∈ ran(keys)  ←Method
 grd3: i ∈ 0..seqSize(keys)  ←Internal
 grd4: seqElemAccess(keys, i) = k  ←Internal
then
 act1: keys := seqSliceToN(keys, i) seqConcat seqSliceFromN(keys, i + 1)
 act1: values := seqSliceToN(values, i) seqConcat seqSliceFromN(values, i+1)
End
```

Because there is only one listed event in the method constructor, there is no case guard. Also because there is no output parameter then there is no output guard. Hence, there is only one proof obligation to be discharged which is the internal parameter feasibility PO. This proof obligation has the following form and can be discharged easily:

$$k \in KEYS, k \in ran(keys) \vdash \exists i.i \in 0..seqSize(keys) - 1 \wedge seqElemAccess(keys, i) = k$$

Now the event is ready to be transformed to a method contract. Figure 4.2 illustrates the generated contract based on the given method constructor:

```
                                  @grd1
method Remove(k : KEYS) returns()
                  @grd2
requires k in keys

requires Invariants()
                                  @grd3                                  @grd4
ensures ∃i :: i in (set k0| 0<=k0 && k0<=|old(keys)| - 1) && old(keys)[i] == k
                                  @act1
&& keys == old(keys)[..i] + old(keys)[i + 1..]
                                  @act2
&& values == old(values)[..i] + old(values)[i + 1..]
```

FIGURE 4.2: Method Contracts Generated from Event *Remove*

The method guard of event *Remove* is translated as a pre-condition of the method, and the internal guards and before-after predicates of the event form the post-condition.

4.2 Stack Abstract Datatype

A stack is an abstract data type which is a collection of elements. A stack has two main operations called *push* and *pop*. *Push* adds an element to the top of the stack and *pop* removes the top element from the stack. In this section an Event-B model of a stack is presented and its transformation to Dafny contracts is explained.

4.2.1 Event-B Model of the Stack ADT

Here we modelled a stack in Event-B using a sequence. The stack has the generic type of *ELEM* which is defined in the context *c0* (not shown here). The stack model is as follows:

```
Machine StackADT Sees c0
Variables stack
Invariants stack ∈ seq(ELEM)
Initialisation stack := ∅

Event push
any e
where
  grd1: e ∈ ELEM
then
  act1: stack := seqPrepend(stack, e)
End

Event pop
any e
where
  grd1: e ∈ ELEM
  grd2: seqSize(stack) ≠ 0
  grd3: e = seqElemAccess(stack, 0)
then
  act1: stack := seqSliceFromN(stack, 1)
End

Event empty
any b
where
  grd1: b ∈ BOOL
  grd2: seqSize(stack) = 0
  grd3: b = TRUE
then
  skip
End

Event nonEmpty
any b
where
  grd1: b ∈ BOOL
  grd2: seqSize(stack) ≠ 0
  grd3: b = FALSE
then
  skip
End
```

There are four events in the model: *push*, *pop*, *empty* and *nonEmpty*. As the names suggest events *push* and *pop* are dealing with push and pop operations. Events *empty* and *nonEmpty* do not change the state of the model. Event *empty* is enabled whenever the stack is empty and *nonEmpty* is enabled whenever the stack is not empty. They both have boolean parameter b as an output.

4.2.2 Model Preparation and Transformation

Similar to the previous example, machine variables are translated to class variables. The type of the *stack* variable is inferred from the only invariant that the model has.

4.2.2.1 Push Method

Event *push* should be transformed to a method with the same name. The following method constructor is provided for this purpose:

`method` $Push(e)$ `returns` () $\{push\}$

Event *Push* has only one guard which is a typing guard:

```
Event push
any e
where
  grd1: e ∈ ELEM ←Typing
then
  act1: stack := seqPrepend(stack, e)
End
```

Because the event does not have any method, case, internal and output guard there is no proof obligation to be discharged. So the event can be transformed to a method contract:

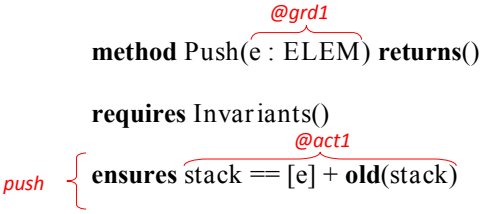

FIGURE 4.3: Method Contracts Generated from Event *Push*

The above method can be implemented in Dafny as follows:

```
method Push(e : ELEM) returns()
modifies this;
ensures stack == [e] + old(stack);
{
     stack := [e] + stack;
}
```

FIGURE 4.4: Implementation of Method *Push*

In the above implementation, *stack* is a class variable and declared as a sequence of integers (`seq<int>`). The part of the code that is placed in the grey box is constructed by hand and the rest is generated automatically by our tool.

4.2.2.2 Pop Method

Transformation of event *pop* to a Dafny method is facilitated by providing a method constructor:

$$\texttt{method}\ Pop()\ \texttt{returns}\ (e)\ \{pop\}$$

With respect to the above method constructor, guards of the event are categorised as follows:

```
Event pop
any e
where
   grd1: e ∈ ELEM  ←Typing
   grd2: seqSize(stack) ≠ 0  ←Method
   grd3: e = seqElemAccess(stack, 0) ←Output
then
   act1: stack := seqSliceFromN(stack, 1)
End
```

Event *pop* has one method guard and one output guard. Based on these a proof obligation must be discharged to prove the feasibility of the output parameter:

$$e \in ELEM, stack \neq emptySeq \vdash \exists\, e.e = seqElemAccess(stack, 0)$$

The above proof obligation can be discharged easily because we know from the hypothesis that *stack* is not empty so a value exists in index 0 of *stack*. The generated method is as follows:

```
                                          @grd1
method Pop() returns(e : ELEM)
                                  @grd2
requires |stack| != 0

requires Invariants()
                                  @act1                    @grd3
pop {  ensures stack == old(stack)[1..] && e == old(stack)[0]
```

FIGURE 4.5: Method Contracts Generated from Event *Pop*

Method *Pop* is implemented and verified in Dafny:

```
method Pop() returns(e : ELEM)
requires stack != [];
modifies this;
ensures stack == old(stack)[1..] && e == old(stack)[0];
{
     e := stack[0];
     stack := stack[1..];
}
```

FIGURE 4.6: Implementation of Method *Pop*

The part of the code that is placed in the grey box is constructed by hand and the rest is generated automatically by our tool.

4.2.2.3 isEmpty Method

The purpose of the *isEmpty* method is to return true if the stack is empty and false if it is not empty. To generate this method we merge events *empty* and *nonEmpty*:

$$\texttt{method}\ isEmpty()\ \texttt{returns}\ (b)\ \{empty,\ nonEmpty\}$$

Guards of event *empty* and *nonEmpty* are categorised with respect to the above method constructor:

```
Event empty
any b
where
   grd1: b ∈ BOOL ←Typing
   grd2: seqSize(stack) = 0 ←Case
   grd3: b = TRUE ←Output
then
   skip
End
```

```
Event nonEmpty
any b
where
   grd1: b ∈ BOOL ←Typing
   grd2: seqSize(stack) ≠ 0 ←Case
   grd3: b = FALSE ←Output
then
   skip
End
```

Before we can transform the above events to *isEmpty* method a number of proof obligations must be discharged:

- Event *empty* Output feasibility:

$$b \in BOOL, seqSize(stack) = 0 \vdash \exists\, b.b = TRUE$$

- Event *nonEmpty* Output feasibility:

$$b \in BOOL, seqSize(stack) \neq 0 \vdash \exists\, b.b = FALSE$$

- Disjointness:

$$b \in BOOL, seqSize(stack) = 0 \vdash \neg seqSize(stack) \neq 0$$

- Completeness:

$$b \in BOOL \vdash seqSize(stack) = 0 \vee seqSize(stack) \neq 0$$

All of the above proof obligations are trivially discharged. The following method can be generated based on the given method constructor:

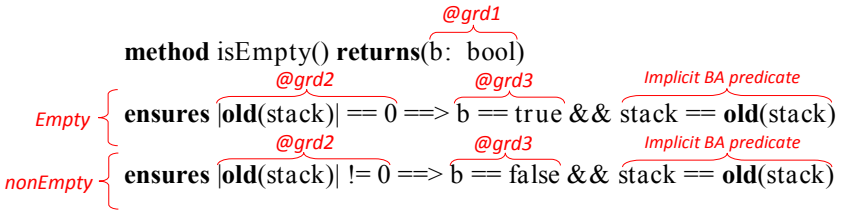

FIGURE 4.7: Method Contracts Generated from Events *Empty* and *nonEmpty*

The above method is implemented and verified in Dafny:

```
method isEmpty() returns(b : bool)
ensures |old(stack)| == 0 ==> stack == old(stack) && b ==  true
ensures |old(stack)| != 0 ==> stack == old(stack) && b ==  false
{
      if(|stack| == 0)
      {
            b := true;
      }
      else if(|stack| != 0)
      {
            b := false;
      }
}
```

FIGURE 4.8: Implementation of Method *isEmpty*

4.3 Summary

In this chapter we presented Event-B models of a map ADT and a stack ADT. We discussed the transformation of the events to Dafny method declaration and contracts based on the rules and extensions provided in Section 3. Method constructor statements are proved to be an effective means for grouping the events and also making event parameter roles explicit. The guard categorisation presented in Section 3.3.3.2 made it possible to differentiate guards and use them in pre- or post-conditions. The extra proof obligations introduced in Section 3.3.4 helped us to discover a new guard for *Add*2 event in the map model. Although the guard was not required for the Event-B refinement verification, it was necessary for generation of implementable Dafny contracts. The generated proof obligations were discharged automatically in Rodin. Our decision to generate a separate `ensures` clause for each listed event in a method constructor makes the generated contracts more readable.

We managed to implement the generated methods manually and verified the implementation against the generated contracts using the Dafny verifier. Implementation of the map methods (*Add* and *Remove*) involved a search algorithm to find the position of an existing key in the *keys* sequence. Verification of the implementation required further assertions in the form of loop invariants (search was implemented using loops). This leaves an important part of the verification for the code level.

Chapter 5

Tool Support for Simple Contract Generation

We introduced a method for transforming Event-B models to simple Dafny code contracts in Chapter 3. This chapter presents a tool in the form of a Rodin plug-in for automatic generation of simple code contracts from Event-B models. The rest of this chapter is organised as follows: Section 5.1 provides an overview of the tool, Section 5.2 illustrates the plug-in interface, Section 5.3 explains the proof obligation generator which generates the necessary POs needed for a sound transformation, Section 5.4 explains how the transformation is done and finally, Section 5.6 discuss some of the tool limitations.

5.1 Tool Overview

Rodin platform is an Eclipse-based platform which allows extensions by means of new plugins. Developing a new plugin for Rodin is done using Eclipse facilities for plug-in development. The Contract Generator tool is also developed as a Rodin plugin. The tool has a simple interface that allows definition of new method constructors in an Event-B machine. An Event-B machine containing the method constructors is the main input of the tool.

FIGURE 5.1: Contract Generator Tool Workflow

The tool directly works with event ASTs in order to analyse the input model. After initial analysis (e.g. guard categorisation) the tool transforms the Event-B model to an intermediate representation in form of a tree. The tree is analysed and further elaborated in order to include all information required for the final translation. Once the final tree is built, the translator is invoked and translates the tree to the final program and contract text.

5.2 User Interface

The plug-in extends the Rodin database by adding a new element to machines for storing constructor statements. The structural Event-B editor has been extended with a new section for constructor statements. Section 5.2 illustrates the area that constructor statements can be added or edited.

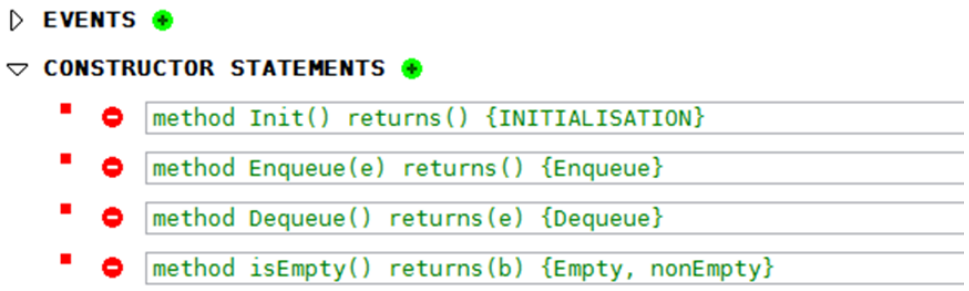

FIGURE 5.2: Constructor Statements Section

In the above figure there are four constructor statements defined in the machine as examples.

As discussed in Section 3.3.4 a number of proof obligations must be discharged before transformation of a model to Dafny contracts. The tool generates these proof obligations automatically based on each constructor statement (See Section 5.3). Newly generated proof obligations can be found in the Event-B explorer under the name of the associated machine. Figure 5.3 illustrates this.

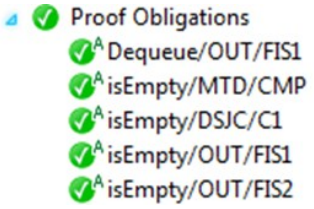

FIGURE 5.3: Contract Generation Proof Obligations

Finally, the user can invoke the tool for transformation of a machine by right clicking on a machine name and then selecting "Generate Dafny Contracts" under "Contract Generation" menu (Figure 5.5).

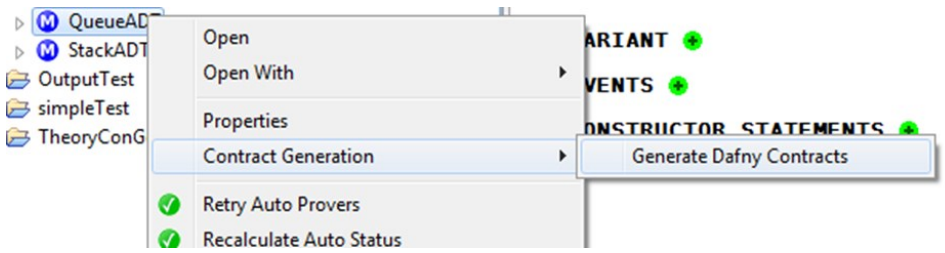

FIGURE 5.4: Invocation of Contract Generator

5.3 Proof Obligation Generator

We have extended the Rodin proof obligation generator to generate the POs discussed in Section 3.3.4. Whenever a new constructor statement is added to a machine, the proof obligation generator is invoked automatically. The plug-in first categorises event guards with respect to the constructor statement. This will happen based on the categorisation introduced in Section 3.3.3.2. The PO generator checks the listed events and input/output parameters of the constructor statement and generates four different POs depending on guards of the listed events:

- If there are n events with internal parameters listed in a constructor statement, then the tool automatically generates n proof obligations except for the initialisation constructor statement. The name of each internal parameter feasibility proof obligation is "*method_name*/INT/FISi" where *method_name* is the name of the target method in Dafny and i is an index to distinct different POs.
- If there are n events with output parameters listed in a constructor statement, then the tool automatically generates n proof obligations for each constructor statement except for initialisation constructor statement. The name of each output feasibility proof obligation is "*method_name*/OUT/FISi" where *method_name* is the name of the target method in Dafny and i is an index to distinct different POs.
- If there is more than one event listed in the constructor statement, then the tool automatically generates one completeness proof obligation. The name of each completeness proof obligation is "*method_name*/MTD/CMP" where *method_name* is the name of the target method in Dafny. No proof obligation is generated for the *Init()* method.

- If there are n events listed in a constructor statement then the tool automatically generates *n-1* disjointness proof obligations. The name of each disjointness PO is "*method_name*/DISJ/Ci" where *method_name* is the name of the target method in Dafny and i is an index to distinct different POs.

5.4 Transformation

After invocation of the contract generator by the user, the transformation of an Event-B model to simple Dafny code contracts takes place in two phases. The tool first builds an abstract syntax tree (AST) with respect to the defined constructor statements and then the AST is translated to Dafny syntax by a translator.

The tool stores everything using a tree. For example, consider $a = b + 3$ as a guard in an Event-B model which should be used for contract generation, the tool stores it in the following tree:

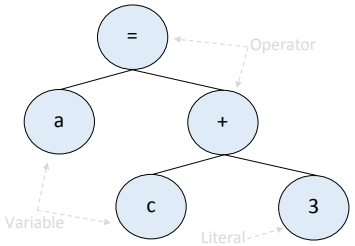

FIGURE 5.5: A Tree Representing Predicate $a = b + 3$

Each node in the tree stores various information such as type of the node, content of the node (e.g. variable names), an array list of the children of the node and etc. When the tool starts building a tree based on a machine and its constructor statements it first generates a *class node* as the root node of the tree. The class node has a number of children: class name, class generics, and class body. The class body node may have three children: class variables, class invariants, and class methods. Figure 5.6 illustrates the structure of an AST built by the tool.

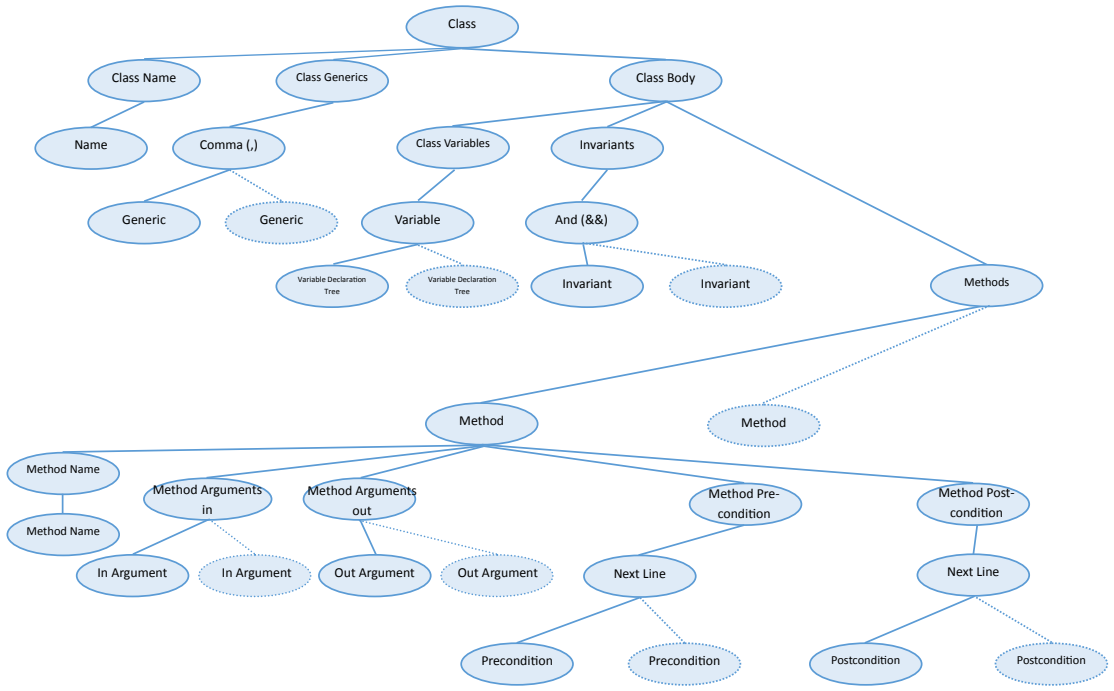

FIGURE 5.6: The Structure of an AST Built by the Tool

The tool builds separate trees for different parts (e.g. method declarations, pre-conditions, post-conditions, etc) of the final program and links them together to build the final tree. When the tree is built it is translated to the target language (here Dafny) by a translator. The translator traverses the tree recursively and translates it to the Dafny language and finally writes it to a file.

5.5 Example: Map ADT

We successfully validated the tool by applying it to a number of examples. In this section we present the results of applying the tool to the model of Map ADT introduced in the previous section. The context $c0$ contains two carrier sets $KEYS$ and $VALUES$ (not shown here). Also we omit the model gluing invariants as well.

`Machine` $m1$ `refines` $m0$ `Sees` $c0$
`Variables` $keys, values$
`Invariants`
$keys \in seq(KEYS) \land values \in seq(VALUES) \land$
$seqSize(keys) = seqSize(values) \land$
$(\forall i, j.i \in 0..seqSize(keys) - 1 \land j \in 0..seqSize(keys) - 1 \land i \neq j \Rightarrow keys(i) \neq keys(j))$
`Initialisation` $keys := \varnothing, values := \varnothing$

Event $Add1$
refines Add
any k, v
where
  grd1: $k \in KEYS$
  grd2: $v \in VALUES$
  grd3: $k \notin ran(keys)$
then
  act1: $keys := seqPrepend(keys, k)$
  act2: $values := seqPrepend(values, v)$
End

Event $Add2$
refines Add
any k, v, i
where
  grd1: $k \in KEYS$
  grd2: $v \in VALUES$
  grd3: $i \in 0..seqSize(keys)$
  grd4: $seqElemAccess(keys, i) = k$
  grd5: $k \in ran(keys)$
then
  act1: $values := seqElemUpdate(values, i, v)$
End

Event $Remove$
refines $Remove$
any k, i
where
  grd1: $k \in KEYS$
  grd2: $k \in ran(keys)$
  grd3: $i \in 0..seqSize(keys)$
  grd4: $seqElemAccess(keys, i) = k$
then
  act1: $keys := seqSliceToN(keys, i)\ seqConcat\ seqSliceFromN(keys, i+1)$
  act1: $values := seqSliceToN(values, i)\ seqConcat\ seqSliceFromN(values, i+1)$
End

We introduce the following method constructors to the model:

method $Init()$ returns() $\{INITIALISATION\}$

method $Add(k,v)$ returns() $\{Add1, Add2\}$

method $Remove(k)$ returns() $\{Remove\}$

The contract generator tool generates the following code based on the above model and method constructors:

```
class MapADT<KEYS,VALUES>
{
var keys : seq<KEYS>;
var values : seq<VALUES>;

predicate Invariants()
reads this;
{
|keys| == |values| && (forall i,j :: i in (set k0| 0<=k0 && k0<=|keys| - 1)
&& j in (set k0| 0<=k0 && k0<=|keys| - 1) && i != j ==> keys[i] != keys[j])
}

method Init() returns()
modifies this;
ensures keys == [] && values == [];

method Add(k : KEYS,v : VALUES) returns()
```

```
requires Invariants();
modifies this;
ensures k !in old(keys) ==> keys == [k] + old(keys)
        && values == [v] + old(values);
ensures k in old(keys) ==>
        exists i :: i in (set k0| 0<=k0 && k0<=|old(keys)| - 1)
        && keys[i] == k && values == old(values)[i:=v];


method Remove(k : KEYS) returns()
requires Invariants();
requires k in keys;
modifies this;
ensures exists i :: i in (set k0| 0<=k0 && k0<=|old(keys)| - 1)
        && keys[i] == k
        && keys == old(keys)[..i] + old(keys)[i + 1..]
        && values == old(values)[..i] + old(values)[i + 1..];
}
```

5.6 Limitations

The presented tool in this chapter has a number of limitations:

1. Translation rules are hard-coded in the source code. If a new operator is defined using the theory plug-in the user has to add the new translation rule in the source code. This issue can be solved by providing a way that user-defined translation rules are allowed.

2. The tool does not perform any static check on the defined constructor statements. If an invalid constructor statement (e.g. a constructor statement with wrong parameters or events list) is defined the tool would not generate anything.

5.7 Summary

This section presented a proof-of-concept tool for transforming Event-B models to Dafny code contracts. We extended Event-B machines to include method constructor statements. The Rodin proof obligation generator was also extended to accommodate the generation of the proof obligations introduced in Section 3. We successfully managed to transform Event-B models discussed in Section 4 using this tool. The source code of the tool is available on GitHub[1].

[1]Contract generator plug-in GitHub repository: https://github.com/dalvandi/congen

Chapter 6

Scheduled Event-B: Derivation of Algorithmic Control Structures in Event-B Refinement

6.1 Introduction

In Chapter 3 we introduced an approach for generating Dafny code contracts from Event-B models. The proposed approach generates Dafny method pre- and post-conditions from a group of atomic Event-B events in a way that any implementation that satisfies the generated pre- and post-conditions is considered to be a correct implementation of the Event-B abstract model.

The above approach works fine for those Event-B specifications where there is an implementation that can be verified against the generated pre- and post-conditions with no or minimal efforts in terms of providing more assertions (e.g. complex loop invariants) at code level. Simple abstract data types (ADTs) such as stacks and queues are examples of such specifications. We presented a number of Event-B specifications and generated Dafny code contracts as examples in Chapter 4.

One common characteristic amongst the aforementioned specifications is that each operation (e.g. *add* and *remove* in map ADT) is performed by only one event and in one step and if there is more than one event for an operation then they are used to specify different cases of that specific operation. For instance, in the Map ADT example, the *add* operation was specified by two events, each of them representing a distinct case: adding a new key and value to the map, or updating the value associated with an existing key. This characteristic implies that in such specifications, the ordering between different events in the model (e.g. *Add* and *Remove* events in Map ADT model) is not

important, because each of them specifies a distinct outcome whose goal is achieved by a single execution of the appropriate event.

In contrast, developing a sequential algorithm to a concrete level requires a strict ordering between model events. A typical development of a sequential algorithm in Event-B starts with an abstract model of the system with a small number of events specifying the ultimate goal of the system. The development continues by refining the abstract model and adding new variables and events in order to model the functional behaviour of the system. In each refinement level, the model is nothing more than a machine with a number of variables, invariants and events and possibly a context which is seen by the machine. Each event can be executed only when it is enabled (i.e. its guard is true). The ordering required for the algorithm should be encoded in the event guards. If there is more than one enabled event, then the next event to be executed is chosen non-deterministically. This non-determinism and implicit control flow has several shortcomings for developing verified programs with Event-B. First, it makes the program derivation specifically for sequential programs difficult (because of the implicit control flow). Second, making the decision about the program structure is left for the concrete level which might not be easy. Third, for a programmer who wants to employ a formal approach like Event-B, it may be easier to have a code-like structure expressing the algorithmic structure of each refinement and its relation to the abstract level. This code-like (program constructs) structure is missing in Event-B.

To overcome the above shortcomings, we have augmented Event-B models with an explicit scheduling language and provided a number of rules for helping the modeller to refine the abstract program structure to a concrete level. In this chapter, we introduce a scheduling language to support the incremental derivation of algorithmic control structure for events as part of the Event-B refinement process. We introduce operators for sequential composition, choice and iteration. Our scheduling language supports non-deterministic choice and iteration of event groups for use at the intermediate levels of refinement. Event groups allow us to model different cases at both the specification and implementation level. Closer to the implementation level, non-deterministic choices are replaced by deterministic if-statements and non-deterministic iterations by deterministic while-statements. We present a number of refinement rules to assist the modeller on how to refine the abstract schedule towards the concrete level.

Our approach is similar to the refinement calculus [BW98, Mor88, Mor87]. The refinement calculus extends the guarded command language [Dij75] of Dijkstra with a *specification statement*. A simple specification statement comprises two predicates *pre* and *post* over the program variable v. This means that if the initial state satisfies *pre*, the execution of the statement terminates in a state satisfying *post*. A specification in the refinement calculus is comprised of executable statements (called *code*) and specification statements. The refinement calculus provides laws for replacing non-deterministic specification statements with deterministic program structures (sequential composition,

if-statements and while-statements). In our approach we introduce non-deterministic choice and iteration before if-statements and while-statements. This allows us to retain guards with events and use the guards to keep the reasoning (including data refinement) localised to events or pairs of corresponding abstract and refined events. This style of reasoning corresponds directly to Event-B refinement and allows us to mix program refinement and data refinement by using an existing tool (Rodin) to represent and prove data refinements. After introducing the deterministic program structure, the control guards in a schedule provide derived guards for events that allow the elimination of the event guards and thus they may be removed from a final implementation.

The rest of the chapter is organised as follows: In Section 6.2 the abstract scheduling language is presented. The use of the language is illustrated through an example in Section 6.2.1. In Section 6.2.3 the abstract schedule refinement rules are introduced. Section 6.3 introduces concrete scheduling language and refinement rules. Section 6.4 introduces guard propagation rules and elimination conditions. Finally, Section 6.5 summarises the chapter.

6.2 Abstract Scheduling Language

We augmented Event-B with a scheduling language to be able to make the structured program associated with each level of refinement explicit. In our approach, in addition to the standard Event-B machine components (i.e. variables, invariants, variants, and events), each model has an associated *schedule*. A schedule allows us to define a structured program with guarded events as its atoms. A schedule associated with an abstract model comprises non-deterministic control constructs since the model events usually have non-deterministic actions and/or abstract data types. The syntax of our proposed abstract scheduling language is shown using Extended Backus-Naur Form (EBNF) [Wir96] in Figure 6.1.

$$
\begin{array}{lcl}
\langle \mathit{Schedule} \rangle & ::= & \mathit{Event} \\
& | & \langle \mathit{Schedule} \rangle \ ; \ \langle \mathit{Schedule} \rangle \\
& | & \langle \mathit{Schedule} \rangle \ \square \ \langle \mathit{Schedule} \rangle \\
& | & \langle \mathit{Schedule} \rangle^{*}
\end{array}
$$

FIGURE 6.1: The Abstract Scheduling Language

The simplest form of a schedule is a single event. *Event* denotes an event in the schedule. A schedule may contain one or more Event-B event. A sequential order can be imposed by the sequential composition operator (;). Non-deterministic choice ($S_1 \ \square \ S_2$) and iteration (S^*) are the abstract control structures. Iteration is required to be finite. This is enforced by proving convergence. The aforementioned control structures allow us to retain the event structure (guards and actions together) so that data refinement

reasoning is localised to pairs of corresponding abstract and refining events using the standard definition of the Event-B refinement.

6.2.1 Example: A Binary Search Algorithm

Here we present a simple example (a binary search algorithm) to illustrate how a sequential program can be developed in Event-B and how we can make the control flow explicit using the abstract scheduling language introduced in the previous section.

The algorithm was originally modelled in [Abr10] with minor differences. Assume that we have a sorted array of integers f and a target value v where v exists in f. The goal of the algorithm is to find the position of v in f. If the size of the array is n, the context of our model will have the following axioms:

axm1: $f \in 0..n\text{-}1 \rightarrow \mathbb{Z}$

axm2: $v \in ran(f)$

axm3: $\forall x, y.\, x \in 0..n-1 \wedge y \in 0..n-1 \wedge x \leqslant y \Rightarrow f(x) \leqslant f(y)$

The array is specified using a total function ($axm1$), v is in the array ($axm2$) and the array is sorted ($axm3$). These axioms can be considered as pre-conditions of our algorithm. The abstract specification is very simple. It has only one event (*found*) apart from the initialisation event. We use a result variable r to specify the desired outcome of the algorithm:

```
Machine m0 Sees c0
Variables r
Invariants r ∈ 0..n − 1
Initialisation r := 0
```

```
Event found
any e
where
   grd1:  f(e) = v
then
   act1:  r := e
End
```

The *found* event finds the position of the v in *one shot* and specifies *what* should be achieved by the algorithm which is yet to be developed. One can capture the control flow between events of the model using the proposed scheduling language:

$$initialisation\,;\, found$$

This means that the execution of the initialisation event followed by *found* event achieves the goal of the algorithm. There is no detail in the abstract specification about how the goal is going to be achieved. The future refinement steps will address this. In the next refinement we need a new event *search* in order to find the position of v in the array.

We specify that the execution of event *search* for a finite number of iterations should result in finding the position of v in the array. This can be captured by the scheduling language as follows:

$$initialisation\,;\, search^*\,;\, found$$

The associated Event-B model with the above schedule is as follows:

`Machine` $m1$ `refines` $m0$ `Sees` $c0$
`Variables` r, k
`Invariants` $k \in 0..n-1$
`Initialisation` $r := 0, k :\in 0..n-1$
	`Event` $search$
	`where`
	`grd1:` $f(k) \neq v$
	`then`
	`act1:` $k :\in 0..n-1$
	`End`
		`Event` $found$
		`refines` $found$
		`where`
		`grd1:` $f(k) = v$
		`then`
		`act1:` $r := k$
		`End`

where k is a new variable used in the computation to find the position of v in f. The *search* event is responsible for finding the index of v in f. Each execution of the *search* event assigns an integer between 0 and $n-1$ to k non-deterministically. Once search finds a value for k that satisfies $f(k) = v$, then *search* is disabled and $found$ is enabled. Note that at this stage *search* is an anticipated event whose convergence is proved in the next refinement. The above schedule is a refinement of the abstract schedule just like the events in the schedule that refine the abstract ones. We use the $\sqsubseteq_R$ symbol to indicate the data refinement (See Section 2.2.2) and $S_0 \sqsubseteq_R S_1$ is read as "*S_0 is refined by S_1*":

$$initialisation\,;\, found$$

$$\sqsubseteq_R$$

$$initialisation\,;\, search^*\,;\, found$$

The above schedule specifies that before the final goal of the algorithm is achieved, event *search* may be executed a number of times. This indicates the existence of a loop in the final program. Although now we have some information on *how* the goal is going to be achieved, still the model contains non-deterministic assignments and does not implement a binary search algorithm. The next refinement replaces this with a binary search. The schedule refinement for this level is as follows:

$$initialisation\,;\, search^*\,;\, found$$

$$\sqsubseteq_R$$

$$initialisation\,;\, (search\_inc \,\square\, search\_dec)^*\,;\, found$$

In this refinement two new variables i and j are defined such that the index of v is between i and j ($v \in f[i..j]$, specified in the invariant). The *search* event is refined by two separate events, *search_inc* and *search_dec* which bring i and j closer together. Events *search_inc* and *search_dec* are marked as convergent and we introduce a variant $(j - i)$ to prove the termination:

Machine $m2$ **refines** $m1$ **Sees** $c0$
Variables r, k, i, j
Invariants $i \in 0..n-1 \wedge j \in 0..n-1 \wedge k \in i..j \wedge v \in f[i..j]$
Variant $j - i$
Initialisation $r := 0, k :\in 0..n-1, i := 0, j := n-1$

Event *search_inc*
refines *search*
where
grd1: $f(k) < v$
then
act1: $k :\in k+1..j$
act2: $i := k+1$
End

Event *search_dec*
refines *search*
where
grd1: $f(k) > v$
then
act1: $k :\in i..k-1$
act2: $j := k-1$
End

Event $found$
refines $found$
where
grd1: $f(k) = v$
then
act1: $r := k$
End

The model still has non-deterministic assignments (the value of variable k is assigned non-deterministically). The next refinement will replace the non-deterministic actions with deterministic ones:

Machine $m3$ **refines** $m2$ **Sees** $c0$
Variables r, k, i, j
Initialisation $r := 0, k := (n-1)/2, i := 0, j = n-1$

Event *search_inc*
refines *search*
where
grd1: $f(k) < v$
then
act1: $k := (k+j+1)/2$
act2: $i := k+1$
End

Event *search_dec*
refines *search*
where
grd1: $f(k) > v$
then
act1: $k := (i+k-1)/2$
act2: $j := k-1$
End

Event $found$
refines $found$
where
grd1: $f(k) = v$
then
act1: $r := k$
End

This refinement adds no new event to the model and does not change its algorithmic structure so the associated schedule remains the same. It is worth noting that so far this derivation involved abstract program refinement (introduction of *search*) prior to data refinement (introduction of i and j). However there is still a gap between the non-deterministic schedule that was introduced in the last refinement and the concrete algorithmic structure containing `if` and `while` rather than $\square$ and $*$, that is needed for the final program code. Before we introduce the concrete schedule language and refinement, first we formalise the introduction and refinement of the abstract schedules in the next section.

6.2.2 Abstract Schedule Semantics

Earlier in this section we presented the abstract scheduling language (See Figure 6.1). The language is similar to the guarded command language introduced in [BW98] and inherits its rules. We intend to provide a number of specific rules for refining a schedule along with the Event-B normal refinement. We base our approach on Back's refinement calculus. In addition to our scheduling constructs, we will use other constructs from Back's language (i.e. `skip`, assumption, assertion and strong iteration) in our proofs. These constructs are not part of our scheduling language and are not intended to be used by modellers.

Here we provide predicate transformers for the constructs of our scheduling language and other constructs used in the proof of our refinement rules.

Event has the following general form:

$$\textbf{Event } \mathit{Event} \textbf{ where } G(v) \textbf{ then } v :| R(v, v')$$

In Section 3.3.5 we showed that an event can be represented using the guarded command language. We represent the above event as follows:

$$[G(v)]\,;\, v :| R(v, v') \tag{6.1}$$

We present the predicate transformer of a statement S and an arbitrary predicate (post-condition) q as $S.q$. We have the following predicate transformer for *Event* with the above definition [BW98]:

$$([G(v)]\,;\, v :| R(v, v')).q \;=\; G(v) \Rightarrow (\forall\, v'.R(v, v') \Rightarrow q[v'/v]) \tag{6.2}$$

Sequential composition and choice[1] predicate transformers are as follows [BW98]:

$$(S_1\,;\, S_2).q = S_1.(S_2.q) \tag{6.3}$$

$$(S_1 \,\square\, S_2).q = S_1.q \wedge S_2.q \tag{6.4}$$

As mentioned before, we use some other constructs from the refinement calculus in our proof. These constructs are `skip`, $\{g\}$ (assertion), $[g]$ (guard) and S^ω (strong iteration). We gave informal definition of assertions and guards (assumptions) in Section 3.3.5. `skip`

[1] Choice operator in our language corresponds to demonic choice in refinement calculus.

does not change the state at all. The `skip`, assertion and guard predicate transformers are as follows [BW98]:

$$\texttt{skip}.q = q \tag{6.5}$$

$$\{g\}.q = g \wedge q \tag{6.6}$$

$$[g].q = g \Rightarrow q \tag{6.7}$$

There are two important iteration operators in refinement calculus: weak iteration (S^*) and strong iteration (S^ω). Weak iteration S^* means that S is executed a finite number of times. Strong iteration S^ω is similar, but S can be executed indefinitely. Weak and strong iterations are modelled in refinement calculus by greatest and least fixpoint, respectively [BW98]:

$$S^\omega = (\mu X \bullet S\,;X \,\square\, \texttt{skip}) \tag{6.8}$$

$$S^* = (\nu X \bullet S\,;X \,\square\, \texttt{skip}) \tag{6.9}$$

The following predicate transformers for strong and weak iteration can be given [BW98]:

$$S^\omega.q = (\mu\, x \bullet S.x \;\wedge\; q) \tag{6.10}$$

$$S^*.q = (\nu\, x \bullet S.x \;\wedge\; q) \tag{6.11}$$

As mentioned earlier, the iteration (S^*) in our language is required to terminate after a finite number of iterations. The termination should be proved at the Event-B level by providing variants and proving that events in the iteration decrease the variant. If the termination events in S is proved then we assume $S^* = S^\omega$.

6.2.3 Abstract Schedule Refinement

In the previous section we gave a predicate transformer semantics to our abstract scheduling language. Our language is a subset of the guarded command language given in [BW98] with events as its atoms. We defined Event-B events using basic guarded command language constructs. The semantics of our language is the same as the guarded command language. Giving the same semantics as refinement calculus to our scheduling

language allows us to use refinement laws introduced in [BW98]. In the rest of this section we provide a number of refinement rules and prove their soundness using refinement calculus laws.

The Event-B refinement is performed at event granularity where each event of a refinement either refines some abstract event or refines `skip`. Correctness of the Event-B refinement is verified through a number of proof obligations. Using Event-B and our scheduling language we can develop structured programs in a stepwise refinement. To accomplish this, here we introduce a number of new rules for *abstract schedule refinement*. Abstract schedule refinement is about elaborating the schedule in tandem with event refinement and the introduction of new events.

Before we present the schedule refinement rules, we need to set a few conventions and rules. First, we use lower case letters to represent individual events and upper case letters to represent *event groups*. We assume that individual events are in the form of (6.1). An event group is a non-deterministic choice between a list of n events:

$$E = e_1 \Box e_2 \Box ... \Box e_{\mathrm{n}}$$

Second, if event group E is refined by event group E' of the same length, written $E \sqsubseteq E'$, then there is a pairwise refinement relation between events in E and E' as follows:

$$e_1 \sqsubseteq e'_1, e_2 \sqsubseteq e'_2, ..., e_{\mathrm{n}} \sqsubseteq e'_n$$

and $\sqsubseteq$ has the same meaning as refinement calculus.

Third, we can show that if event e_1 is refined by event e_2 in the Event-B level correctly, then it is a correct refinement in refinement calculus setting as well (i.e. $e_1 \sqsubseteq e_2$ holds). To prove this we need to show that the following holds:

$$wp(e_1, Q) \Rightarrow wp(e_2, Q)$$

Assume that e_1 and e_2 have the following form in Event-B:

$$\textbf{Event } e_1 \textbf{ where } G_1(v) \textbf{ then } v :| R_1(v, v')$$

$$\textbf{Event } e_2 \textbf{ where } G_2(v) \textbf{ then } v :| R_2(v, v')$$

The above events can be represented in the guarded command language as follows:

$$[G_1(v)]\,; v :| R_1(v, v')$$

$$[G_2(v)]\,;\, v :|\, R_2(v, v')$$

Based on the weakest precondition predicate transformers given in Section 3.3.5 we have:

$$(G_1(v) \Rightarrow \forall\, v'.\, R_1(v, v') \Rightarrow Q[v'/v]) \Rightarrow (G_2(v) \Rightarrow \forall\, v'.\, R_2(v, v') \Rightarrow Q[v'/v])$$

Event-B proof obligations guarantee that guards of the concrete event are stronger than the guards of abstract event (Guard strengthening PO) and actions of concrete event simulate the actions of the abstract event [Abr10]. Based on this, if all Event-B proof obligations are discharged, we have:

$$\begin{aligned}
&wp(e_1, Q)\\
&= \{\text{definition of event } e_1\}\\
&wp([G_1(v)]\,;\, v :|R_1(v, v'), Q)\\
&= \{\text{weakest precondition of event } e_1\}\\
&G_1(v) \Rightarrow (\forall\, v'.\, R_1(v, v') \Rightarrow Q[v'/v])\\
&\Rightarrow \{\text{guard strengthening proof obligation}\}\\
&G_2(v) \Rightarrow (\forall\, v'.\, R_1(v, v') \Rightarrow Q[v'/v])\\
&\Rightarrow \{\text{simulation proof obligation}\}\\
&G_2(v) \Rightarrow (\forall\, v'.\, R_2(v, v') \Rightarrow Q[v'/v])\\
&= \{\text{weakest precondition}\}\\
&wp([G_2(v)\,;\, v :|R_2(v, v')], Q)\\
&= \{\text{definition of } e_2\}\\
&wp(e_2, Q)
\end{aligned}$$

Fourth, recall from Section 2.2.2 that we write $E \sqsubseteq_R E'$ to denote data refinement of E by E' through gluing relation R. Here we assume that events are correctly data refined in Event-B. Using $\sqsubseteq$ indicates that no data refinement is performed.

Finally, we know from refinement calculus [BW98] that the sequential composition (;), non-deterministic choice ($\Box$), and iteration ($*$) operators are monotonic with respect to refinement, i.e., if $S_1 \sqsubseteq S_1'$ and $S_2 \sqsubseteq S_2'$ then:

$$S_1\,;\, S_2 \sqsubseteq S_1'\,;\, S_2'$$

$$S_1 \,\Box\, S_2 \sqsubseteq S_1' \,\Box\, S_2'$$

$$S_1^* \sqsubseteq (S_1')^*$$

We also use the following rule from [BW98]:

$$S = \texttt{skip} \,;\, S = S \,;\, \texttt{skip}$$

and the following rule:

$$S^* = (\texttt{skip} \,\square\, S)^*$$

The soundness of the above rule is proved using iteration decomposition lemma given in [BW98].

We introduce the following general rules for algorithmic and schedule refinement:

- **Rule 1 (Case Split)**

  $e \sqsubseteq_R (e'_1 \,\square\, ... \,\square\, e'_m)$ (provided $e \sqsubseteq_R e'_1$,..., $e \sqsubseteq_R e'_m$)

  In a typical Event-B refinement an abstract event may be refined by one or more events in the concrete level. If an event is refined by more than one event, each of the refined events is typically used to represent a different case of the abstract event. An abstract schedule with one event (e) can be replaced with a concrete schedule with a group of m events ($e'_1 \,\square\, ... \,\square\, e'_m$).

  Proof. The case split refinement rule follows from the monotonicity property of non-deterministic choice operator and $e = e \,\square\, ... \,\square\, e$. □

  We also need to prove that the enabledness condition is preserved by the case events. This means that whenever abstract event (e) is enabled at least one of the concrete events ($e'_1, ..., e'_m$) should be enabled:

  $$grd(e) \Rightarrow grd(e'_1) \vee ... \vee grd(e'_m)$$

  where $grd(e)$ represents the guards of event e. We will discuss in Section 6.4.1 that if the enabledness condition is not satisfied then we would not be able to eliminate event guards in the final program.

- **Rule 2 (Loop Introduction)**

  $E \sqsubseteq_R F^* \,;\, E'$ (provided $\texttt{skip} \sqsubseteq_R F$ and $E \sqsubseteq_R E'$)

  Here the abstract schedule E is replaced by the schedule $F^* \,;\, E'$ which specifies that some event from F is continually executed until an event in E' is enabled. In the later refinements F^* should be refined by a concrete loop.

Proof. This refinement rule can be proved as follows:

$$\begin{aligned} E &\sqsubseteq \texttt{skip} \,;\, E \\ &\sqsubseteq \texttt{skip}^* \,;\, E \\ &\sqsubseteq_R \{monotonicity\, of\ *\ and\ ;\} \\ &F^* \,;\, E' \end{aligned}$$

□

Since we have to prove loop termination, F should be either labelled as *convergent* or *anticipated*. If it is labelled as *convergent* then a variant should be provided in order to prove the termination of each event in F. The proof of termination can be postponed for any of the F events until future refinements by marking them as *anticipated*. Proving the convergence ensures the termination of the loop.

Similar to the previous rule, enabledness condition should be satisfied:

$$grd(E) \Rightarrow grd(F) \vee grd(E')$$

If E is an event group then $grd(E)$ represents the disjunction of the guards of the events in E.

- **Rule 3 (Loop Body Extension)**

  $E^* \sqsubseteq_R (F \,\square\, E')^*$ (provided $\texttt{skip} \sqsubseteq_R F$ and $E \sqsubseteq_R E'$)

  The body of a loop can be extended by an event group in the concrete level. The body should be later refined by a concrete schedule.

  Proof. The soundness of loop body extension refinement rule is proved as follows:

$$\begin{aligned} E^* &= (\texttt{skip} \,\square\, E)^* \\ &\sqsubseteq_R \{monotonicity\, of\ \square\} \\ &(F \,\square\, E')^* \end{aligned}$$

  therefore:

$$E^* \sqsubseteq_R (F \,\square\, E')^*$$

  □

  Also like previous rule, F should be labelled as *convergent* or *anticipated* and enabledness condition should be proved.

Nested control structures can be derived by repeated application of these refinement rules.

6.3 Refinement to Concrete Program Structures

The scheduling language and refinement rules introduced in the previous section provide a way for specifying the control flow between events and refining the control flow. However, due to the fact that the schedule itself contains non-deterministic choices and iterations there is still a gap between the schedule and the concrete algorithmic structure of the final program code. To bridge the gap we extend the abstract scheduling language introduced in Section 6.2 with deterministic branches and loops (Figure 6.2).

$\langle Schedule \rangle ::= Event$
$\quad | \quad \langle Schedule \rangle \; ; \; \langle Schedule \rangle$
$\quad | \quad \langle Schedule \rangle \; \Box \; \langle Schedule \rangle$
$\quad | \quad \langle Schedule \rangle^*$
$\quad | \quad \textbf{if}(\langle Cond \rangle)\{\langle Schedule \rangle\}, \{\textbf{elseif}(\langle Cond \rangle)\{\langle Schedule \rangle\}\}, [\textbf{else}\{\langle Schedule \rangle\}]$
$\quad | \quad \textbf{while}(\langle Cond \rangle)\{\langle Schedule \rangle\}$

$\langle Cond \rangle ::= Predicate$

FIGURE 6.2: The Scheduling Language

The extension adds deterministic `if..else` branches and `while` loops with explicit conditions ($Cond$) to the language. The branch and loop conditions should be valid Event-B predicates as defined in [Abr10]. Non-deterministic choices and iterations can be refined to deterministic branches and loops, respectively. Deterministic branches and loops can be defined using non-deterministic choices and iterations and guards [BW98]:

$$\textbf{while}(Cond)\{S\} \triangleq ([Cond] \; ; \; S)^\omega \; ; \; [\neg Cond]$$

$$\textbf{if}(Cond)\{S_1\}\textbf{else}\{S_2\} \triangleq [Cond] \; ; \; S_1 \; \Box \; [\neg Cond] \; ; \; S_2$$

The while-loop is defined using *strong iteration* operator. Since we ensure termination by proving convergence we assume $S^\omega = S^*$. So we have the following definition for while-loop:

$$\textbf{while}(cond)\{S\} \triangleq ([cond] \; ; \; S)^* \; ; \; [\neg cond]$$

In order to be able to keep data and algorithmic refinements separate from each other we postpone the refinement of abstract control structures (i.e. choice and iteration) to concrete structures until the data refinement in Event-B is done. The following two schedule refinement rules will allow us to refine an abstract schedule with non-deterministic control structures to a concrete level:

- **Rule 4 (Concrete Loop)** $S^* \sqsubseteq \textbf{while}(cond)\{S\}$

  An abstract loop can be refined by a deterministic while-loop. The modeller should explicitly determine the concrete loop condition ($cond$).

  Proof. The refinement of an abstract iteration to a concrete while-loop can be proved by induction:

  $$\begin{aligned} S &\sqsubseteq \texttt{skip} \,;\, S \\ &\sqsubseteq \{\text{guards refine } \texttt{skip}\} \\ &[cond] \,;\, S \end{aligned}$$

  Since * is monotonic:

  $$\begin{aligned} S^* &\sqsubseteq ([cond] \,;\, S)^* \\ &\sqsubseteq ([cond] \,;\, S)^* \,;\, \texttt{skip} \\ &\sqsubseteq \{\text{guards refine } \texttt{skip}\} \\ &([cond] \,;\, S)^* \,;\, [\neg cond] \\ &= \{\text{definition of while-loop}\} \\ &\textbf{while}(cond)\{S\} \end{aligned}$$

  □

- **Rule 5 (Concrete Branch)**

  $S_0 \,\square\, S_1 \,\square\, ... \,\square\, S_m$

  $\sqsubseteq$

  $\textbf{if}(cond1)\{S_1\ \}\textbf{elseif}(cond2)\{S_2\}...\textbf{else}\{S_m\}$

  Non-deterministic choice between two or more schedules can be refined to an *if..else* structure. The modeller should explicitly provide branch conditions.

  Proof. Here, for simplification, we present a proof for a simple case when there are only two event groups involved in the non-deterministic choice:

  $$\begin{aligned} S_1 \,\square\, S_2 &\sqsubseteq (\texttt{skip} \,;\, S_1) \,\square\, (\texttt{skip} \,;\, S_2) \\ &\sqsubseteq \{\text{guards refine } \texttt{skip} \text{ and monotonicity of } \square\} \\ &([cond] \,;\, S_1) \,\square\, ([\neg cond] \,;\, S_2) \\ &= \{\text{definition of branch}\} \\ &\textbf{if}(cond)\{S_1\}\textbf{else}\{S_2\} \end{aligned}$$

  □

  The proof can be generalised to the case of m branches.

6.3.1 Refining the Abstract Schedule of the Search Example

Remember the example from Section 6.2. We refined the abstract specification to a level that the model was completely deterministic and no further data refinement was required. Now that our scheduling language has deterministic loops and branches, we can refine its non-deterministic schedule to a deterministic one. This is done in three steps. We first refine the loop body to a deterministic branch:

$$search\_inc \,\square\, search\_dec$$

$$\sqsubseteq$$

$$\textbf{if}(f(k)<v)\{search\_inc\}\ \textbf{else}\{search\_dec\}$$

The body of the loop is replaced by its refinement and finally the next step is to replace the non-deterministic loop with a deterministic while-loop:

$$initialisation\,;(\textbf{if}(f(k)<v)\{search\_inc\}\ \textbf{else}\{search\_dec\})^*\,;found$$

$$\sqsubseteq$$

$$initialisation;$$

$$\textbf{while}(f(k)\neq v)\{\textbf{if}(f(k)<v)\{search\_inc\}\ \textbf{else}\{search\_dec\}\};$$

$$found$$

The final schedule represents the program structure at code level. In Section 6.4 we discuss how the guards of the events are removed by verifying that they follow from the explicit guards of the schedules.

6.4 Guard Propagation Rules and Elimination Conditions

When the algorithmic structure in an Event-B model is made explicit by our scheduling language we want to be able to eliminate the guards within the events as we expect them to follow from the explicit control guards. To reason about this, it would be enough to prove that right before the execution of the actions, the corresponding guards are true. This is to say that, conditions and ordering imposed by the schedule can replace the event guards. Here we want to define a condition called *guard elimination condition* for each event in the schedule in a way that if it holds then event guards can be eliminated safely.

Before we can define the guard elimination condition we have to make it clear how the schedule guards and ordering are related to the scheduled events. To do this, we introduce a new concept called *derived guards*. A *derived guard* is a condition (assertion) that is guaranteed by the schedule to hold before the execution of an event. To illustrate this, assume that we have a simple schedule with only one event evt:

$$\textbf{Event } evt \textbf{ where } G(v) \textbf{ then } v :| R(v, v')$$

We say dg is a derived guard for event evt which is guaranteed to hold when control reaches there:

$$\{dg\} \,;\, evt$$

In the above schedule derived guard dg is obtained from forward propagation of the relevant schedule guards (i.e. loop or branch conditions). Forward propagation rules are discussed in the Refinement Calculus [BW98] and here we provide the following rules for obtaining derived guards for scheduled events:

- **Rule 6 (Branch)** If dg_s is a derived guard for a conditional construct (`if..else`) and it has n branches then the guards can be propagated as follows:

$$\{dg_s\} \,;\, \textbf{if}(cond_1)\{\, S_1 \,\} \textbf{ elseif}(cond_2)\{\, S_2 \,\} \ldots \textbf{ else}\{\, S_{\mathrm{n}} \,\}$$
$$=$$
$$\{dg_s\} \,;\, \textbf{if}(cond_1)\{\, \{dg_1\} \,;\, S_1 \,\} \textbf{ elseif}(cond_2)\{\, \{dg_2\} \,;\, S_2 \,\} \ldots \textbf{ else}\{\, \{dg_{\mathrm{n}}\} \,;\, S_{\mathrm{n}} \,\}$$

  where

$$dg_{\mathrm{i}} \triangleq dg_{\mathrm{s}} \wedge (\neg cond_1 \wedge \ldots \wedge \neg cond_{\mathrm{i\text{-}1}}) \wedge cond_{\mathrm{i}}$$
$$dg_{\mathrm{n}} \triangleq dg_s \wedge \neg cond_1 \wedge \ldots \wedge \neg cond_{\mathrm{n\text{-}1}}$$

  and $i \in 1..n-1$. Here dg_i holds right before S_i and dg_n holds right before S_n. Depending on the structure of S_i and S_n, the derived guards may be forward propagated further.

- **Rule 7 (Loop)** A loop condition can be propagated as follows:

$$\textbf{while}(cond)\{\, S_1 \,\} \,;\, S_2$$
$$=$$
$$\textbf{while}(cond)\{\, \{cond\} \,;\, S_1 \,\} \,;\, \{\neg cond\} \,;\, S_2$$

For further propagation of derived guards we also have the following three rules:

- **Rule 8** If dg is a derived guard for a schedule and the schedule has two sub-schedules connected via a sequential composition operator:

  $$\{dg\} ; (S_1 ; S_2)$$

  then dg is only a derived guard for the first sub-schedule:

  $$\{dg\} ; S_1 ; S_2$$

- **Rule 9** A schedule S preserves the condition p if:

  $$\{p\} ; S \sqsubseteq S ; \{p\}$$

  The program operators preserve conditions if their operands do:

  - $S_1 ; S_2$ preserves p provided that S_1 and S_2 preserve p.
  - **if**($cond$)**{**S_1**}else{**S_2**}** preserves p provided that if $cond$ holds then S_1 and if $\neg cond$ the S_2 preserve p.
  - **while**($cond$)**{**S**}** preserves p provided that if $cond$ holds then S preserves p.

  In addition to the above rules, for an atomic event evt proving that the event preserves p is the same as proving invariant preservation by treating p as invariant. Invariant preservation proof obligation are discussed in details in [Abr10].

Now that we have clear rules for obtaining derived guards we can define the guard elimination condition. We represent event evt using the language introduced in 3.3.5:

$$[G(v)] ; v :|R(v, v') \tag{6.12}$$

If dg is a derived guard for event evt then we have

$$\{dg\} ; [G(v)] ; v :|R(v, v')$$

We can eliminate event guard $G(v)$ using derived guard dg:

$$\{dg\} ; [G(v)] ; v :|R(v, v') = \{dg\} ; v :|R(v, v')$$

$$\text{provided that } dg \Rightarrow G(v)$$

Based on the above, if there is a schedule with n events, each event will have one guard elimination condition in the following general form:

$$dg_{\mathrm{i}} \Rightarrow G_{\mathrm{i}}$$

where $i \in 1..n$ and dg_i and G_i represent the derived guard and event guards of the ith event, respectively. Proving the guard elimination condition for each event allows us to remove event guards safely.

By propagating schedule conditions (derived guards) to events we should be able to remove the event original guards. If the event guards are eliminated successfully (i.e. the execution of actions under the propagated guards preserve the model invariants) then the ordering imposed by the schedule guarantees the preservation of the model properties and establishment of the post-conditions (i.e. successful elimination of the $FINAL$ event guards).

Recall the concrete schedule and model of the binary search algorithm in Section 6.3.1:

$$initialisation;$$

$$\textbf{while}(f(k) \neq v)\{\textbf{if}(f(k)<v)\{search\_inc\}\ \textbf{else}\{search\_dec\}\};$$

$$found$$

By applying forward propagation rules, the following guards (shown in brackets before each event) can be derived:

$\{f(v) \neq v \wedge f(k) < v\}$
Event $search\_inc$
where
  grd1: $f(k) < v$
then
  act1: $k := (k+j+1)/2$
  act2: $i := k + 1$
End

$\{f(v) \neq v \wedge \neg(f(k) < v)\}$
Event $search\_dec$
where
  grd1: $f(k) > v$
then
  act1: $k := (i+k-1)/2$
  act2: $j := k - 1$
End

$\{\neg(f(v) \neq v)\}$
Event $found$
where
  grd1: $f(k) = v$
then
  act1: $r := k$
End

Proving that the original event guards can be removed safely is trivial.

6.4.1 Enabledness and Guard Elimination Condition

Now that guard elimination condition is introduced we can show that if the enabledness condition presented in Section 6.2.3 for abstract schedule refinement rules is not satisfied, we will not be able to eliminate event guards in the final level. We illustrate this with

an example. Recall the search algorithm introduced earlier. Assume that we strengthen the guards of events *search_in* and *search_dec* by adding a new guard $i < j$:

```
Event search_inc
where
   grd1:  f(k) < v
   grd2:  i < j
then
   act1:  k := (k + j + 1)/2
   act2:  i := k + 1
End
```

```
Event search_dec
where
   grd1:  f(k) > v
   grd2:  i < j
then
   act1:  k := (i + k − 1)/2
   act2:  j := k − 1
End
```

While the new events with stronger guards still satisfy the model properties, the enabledness condition cannot be satisfied:

$$f(k) \neq v \Rightarrow (f(k) < v \wedge i < j) \vee (f(k) > v \wedge i < j)$$

The explicit schedule guards presented in Section 6.3.1 cannot satisfy guard elimination condition for any of the search events:

- GEC for *search_inc*: $f(v) \neq v \wedge f(k) < v \Rightarrow f(k) < v \wedge i < j$
- GEC for *search_dec*: $f(v) \neq v \wedge f(k) > v \Rightarrow f(k) > v \wedge i < j$

Even if we strengthen the explicit schedule conditions by adding $i < j$ to the branch condition we will not be able to satisfy guard elimination condition for event *search_dec*. This simple example shows that if the enabledness is not preserved it is most likely that we will not be able to eliminate some of the event guards.

6.5 Summary

Event-B has a rich mathematical language and powerful tool support (Rodin) that can be used effectively for specification and verification of sequential programs. However, Event-B lacks the explicit control flow structure and therefore events ordering should be specified implicitly as event guards. The lack of the explicit ordering makes the transformation of the concrete model to a program code difficult. In this chapter we introduced an approach that makes the control flow explicit by augmenting Event-B with a *scheduling language*.

The scheduling language and the introduced refinement rules allow the modeller to derive the program structure in an stepwise manner. The scheduling language allows the modeller to introduce iterations and non-deterministic choices in the abstract level. It also has familiar control constructs like if-statements and while-loops that will replace the non-deterministic structures in the concrete level. We presented a number of rules

which will assist the modeller in refining an abstract schedule towards a concrete one. We also discussed how guard elimination conditions can guarantee that the events in the schedule will follow the explicit control flow and how we can remove original event guards.

Chapter 7

Case Studies: Simple Sort and Schorr-Waite Marking Algorithms

7.1 Introduction

In the previous chapter we introduced an approach for derivation of algorithmic structure in Event-B refinement. This chapter will present two case studies based on that approach: 1) A simple sorting algorithm 2) Schorr-Waite graph marking algorithm. We will discuss the modelling and derivation of the algorithmic structure for both algorithms based on the approach presented previously. The case studies validate our approach for developing correct sequential algorithms in Event-B.

7.2 Case Study 1: Simple Sort Algorithm

The first example we present in this chapter is derivation and verification of a sorting algorithm using Scheduled Event-B. This example is important specifically because it showcases the introduction of nested loops. The rest of this section discusses the modelling and verification of the algorithm.

7.2.1 Abstract Specification

The goal of the algorithm is to sort any non-empty array of natural numbers. The input array is specified in the context of the model as follows:

CONTEXT $c0$
CONSTANTS n, f

AXIOMS
  axm1: $0 < n$
  axm2: $f \in 0..n-1 \rightarrowtail \mathbb{N}$

In the context $c0$ constant n represents the size of array f. Machine $m0$ defines the abstract specification of the algorithm. Event *sorted* sorts the array in one shot and assigns the sorted array to the result variable r:

MACHINE $m0$ SEES $c0$
VARIABLES r
INVARIANTS
  inv1: $r \in 0..n-1 \rightarrowtail \mathbb{N}$
EVENTS
Event $INITIALISATION$
then
  act1: $r := f$
End

Event *sorted*
any s
where
  grd1: $s \in 0..n-1 \rightarrowtail \mathbb{N}$
  grd2: $ran(s) = ran(f)$
  grd3: $\forall i, j.i \in 0..n-1 \land j \in 0..n-1 \land i < j \Rightarrow s(i) \leqslant s(j)$
then
  act1: $r := s$
End

The abstract schedule is:

$$initialisation\,;sorted$$

The above schedule states that after the model is initialised a single execution of event *sorted* will produce the desirable result which is a sorted version of the input array.

7.2.2 First Refinement

In order to derive the implementation from the abstract specification we have to refine it. In the first refinement we introduce a new event called *sorting*. The new event specifies the way that the array is sorted. The *sorting* event is supposed to be executed repeatedly until the array is sorted. The abstract schedule can be refined as follows:

$$initialisation\,;sorted$$

$$\sqsubseteq_R$$

$$initialisation\,;sorting^*\,;sorted$$

Here we used the loop introduction rule introduced in Section 6.2.3. As can be seen, the refined *sorted* event is a refinement of the abstract *sorted* event and the *sorting* event refines *skip*. The enabledness condition for this schedule refinement can be discharged trivially.

Two new variables are introduced in this refinement. First, variable *sor* which is a copy of array f which can be modified and sorted by event *sorting*, and second, variable i which is the index to the value in the array which is currently being evaluated and each execution of the event will increase it by one. We labelled *sorting* event as *convergenet* and by providing variant $n - i$ we managed to prove the termination of the abstract loop introduced in this schedule refinement. The Event-B model for this refinement is as follows:

MACHINE $m1$ refines $m0$ SEES $c0$
VARIABLES sor, i
INVARIANTS
  inv1: $i \in 0..n-1$
  inv2: $sor \in 0..n-1 \rightarrow \mathbb{N}$
  inv3: $ran(sor) = ran(f)$
  inv4: $\forall k, l.k \in 0..i \wedge l \in 0..i \wedge k < l \Rightarrow sor(k) \leqslant sor(l)$
  inv5: $\forall k, l.k \in 0..i-1 \wedge l \in i..n-1 \Rightarrow sor(k) \leqslant sor(l)$
EVENTS
Event $INITIALISATION$
then
  act1: $i := 0$
  act2: $sor := f$
End

Event *sorting*
any m
where
  grd1: $i \neq n-1$
  grd2: $m \in i..n-1$
  grd3: $sor(m) = min(sor[i..n-1])$
then
  act1: $sor := sor \Leftarrow \{i \mapsto sor(m)\} \Leftarrow \{m \mapsto sor(i)\}$
  act2: $i := i+1$
End

Event *sorted* refines *sorted*
where
  grd1: $i = n-1$
with
  s: $s = sor$
then
  act1: $r := sor$
End

Invariant $inv4$ states that the array is sorted up to ith element. Invariant $inv5$ states that all elements up to index i are smaller than elements with indices greater than i. We have provided a witness for event *sorted* and removed parameter s from it.

7.2.3 Second Refinement

The second refinement adds more details about how the algorithm sorts the array. We introduce variable x which is a value between i and $n-1$. It will help us to remove parameter m from *sorting* event and model the steps needed to calculate that value. The schedule is refined as follows:

$$initialisation\,;sorting^*\,;sorted$$

$$\sqsubseteq_R$$

$$initialisation\,;(sorting\_in^*\,;sorting)^*\,;sorted$$

Here again we used the loop introduction rule in schedule refinement. The result is a nested loop where the new event *sorting_in* represents the inner loop. Since there is not enough detail to prove the termination of the inner loop we label *sorting_in* as *anticipated*. The *sorting_in* event indicates that the index of the smallest value between indices i and $n-1$ (i.e. variable x) should be calculated. However, at this level we do not provide the details on how x is calculated. The only thing that we know is that *sorting_in* is executed repeatedly until x satisfies guard $grd2$ of the *sorting* event:

```
Event sorting_in
then
   act1:  x :∈ i..n − 1
End

Event sorting refines sorting
where
   grd1:  i ≠ n − 1
   grd2:  sor(x) = min(sor[i..n − 1])
then
   act1:  sor := sor ⩤ {i ↦ sor(x)} ⩤ {x ↦ sor(i)}
   act2:  i := i + 1
   act3:  x := i + 1
End
```

7.2.4 Third Refinement

The third refinement provides all the details about the steps that are needed to be taken in order to find the smallest value between indices i and $n-1$. In this level we refine *sorting_in* by two events *sorting_in_1* and *sorting_in_2*. The schedule is refined as follows:

$$initialisation\,;(sorting\_in^*\,;sorting)^*\,;sorted$$

$$\sqsubseteq_R$$

$$initialisation\,;((sorting\_in\_1\,\square\,sorting\_in\_2)^*\,;sorting)^*\,;sorted$$

Variable j is a counter for the inner loop. The model should guarantee that x is always the index of the smallest value between i and j. The model invariants and events are as follows:

```
INVARIANTS
   inv1:  j ∈ 0..n − 1
   inv2:  x ∈ i..j
   inv3:  sor(x) = min(sor[i..j])

Event sorting_in_1 refines sorting_in
where
   grd1:  i ≠ n − 1
   grd2:  j ≠ n − 1
   grd3:  sor(x) ⩽ sor(j + 1)
then
   act1:  j := j + 1
End

Event sorting_in_2 refines sorting_in
where
   grd1:  i ≠ n − 1
   grd2:  j ≠ n − 1
   grd3:  sor(x) > sor(j + 1)
then
   act1:  j := j + 1
   act2:  x := j + 1
End

Event sorting refines sorting
where
   grd1:  i ≠ n − 1
   grd2:  j = n − 1
then
   act1:  sor := sor ⩤ {i ↦ sor(x)} ⩤ {x ↦ sor(i)}
   act2:  i := i + 1
   act3:  x := i + 1
   act4:  j := i + 1
End
```

Event *sorting_in* is refined by two events *sorting_in_1* and *sorting_in_2*. This follows the normal Event-B refinement which was described in rule 1 (see 6.2.3). Now that the concrete level has no non-deterministic action we can refine the abstract constructs in the schedule to their concrete counterparts. We do this in three steps: first, we refine the non-deterministic choice to a concrete branch, second, we refine the non-deterministic inner loop to a concrete `while` loop and finally we refine the outer non-deterministic loop:

$$
\begin{array}{l}
initialisation\,;((sorting\_in\_1 \,\square\, sorting\_in\_2)^*\,;sorting)^*\,;sorted \\
\sqsubseteq \\
initialisation\,;((\\
\quad if(sor(x) \leqslant sor(j+1))\{sorting\_in\_1\}else\{sorting\_in\_2\} \\
)*\,;sorting)*\,;sorted \\
\sqsubseteq \\
initialisation\,;(\\
\quad while(j \neq n-1)\{ \\
\quad\quad if(sor(x) \leqslant sor(j+1))\{sorting\_in\_1\}else\{sorting\_in\_2\} \\
\quad \} \\
;sorting)^*\,;sorted \\
\sqsubseteq \\
initialisation; \\
\quad while(i \neq n-1)\{ \\
\quad\quad while(j \neq n-1)\{ \\
\quad\quad\quad if(sor(x) \leqslant sor(j+1))\{sorting\_in\_1\}else\{sorting\_in\_2\} \\
\quad\quad \}; \\
\quad\quad sorting \\
\quad \}; \\
sorted
\end{array}
$$

Here we broke the schedule refinement into three steps to show how it is done, however it is possible to perform all the three steps at once. The final (concrete) schedule has no non-deterministic loop or branch so no further refinement is required to obtain final algorithmic structure. Now that all the abstract non-deterministic constructs are refined to their concrete deterministic counterparts, the guard elimination conditions can be generated to prove the correctness of the schedule.

7.2.5 Guard Elimination Conditions

In Section 6.4 we introduced a number of rules for propagation of derived guards in order to be able to eliminate actual event guards. Here we show how derived guards are used in guard elimination.

- **Event sorting_in_1** Using the rules presented in Section 6.4, the derived guard for *sorting_in_1* is as follows:

$$i \neq n-1 \wedge j \neq n-1 \wedge sor(x) \leqslant sor(j+1)$$

  The above condition is guaranteed to hold by the schedule right before execution of event *sorting_in_1*. Since we can prove that this condition is stronger than the event guards, we can eliminate event guards in the final algorithm structure safely:

$$i \neq n-1 \wedge j \neq n-1 \wedge sor(x) \leqslant sor(j+1) \Rightarrow i \neq n-1 \wedge j \neq n-1 \wedge sor(x) \leqslant sor(j+1)$$

- **Event sorting_in_2** Similar to event *sorting_in_1*, here we can obtain the following derived guard:

  $$i \neq n-1 \wedge j \neq n-1 \wedge \neg(sor(x) \leqslant sor(j+1))$$

  and this derived guard is stronger than event guards:

  $$i \neq n-1 \wedge j \neq n-1 \wedge \neg(sor(x) \leqslant sor(j+1)) \Rightarrow i \neq n-1 \wedge j \neq n-1 \wedge sor(j+1) < sor(x)$$

- **Event sorting** The following derived guard is obtained:

  $$i \neq n-1 \wedge j = n-1$$

  As discussed in Section 6.4 rule 6 has a side condition which should be satisfied before we can use it. In this case we have to prove that both events inside the inner loop preserve $i \neq n-1$. It is trivial to prove this, since neither of the events inside the inner loop update the value of i. The above derived guard is stronger than the event guards so we can eliminate event guards safely:

  $$i \neq n-1 \wedge j = n-1 \Rightarrow i \neq n-1 \wedge j = n-1$$

- **Event sorted** Finally using rule 2 we can obtain the following derived guard for event *sorted* and we can safely eliminate its guard:

  $$i = n-1$$

By proving that all the events guards can be eliminated safely, we can conclude that the schedule is correct.

7.3 Case Study 2: Schorr-Waite Marking Algorithm

The Schorr-Waite [SW67] algorithm is a graph marking algorithm named after its inventors. The importance of the algorithm is due to its low memory consumption [Bub07]. It marks all reachable nodes from a *top* node in a given graph. Instead of logging the path to the current node from the top node, the algorithm simply reverses the traversed edges and use them for backtracking when there is no reachable unmarked node left. It was originally proposed for dealing with the problem of garbage collection when the available space is low. The algorithm has attracted substantial attention in the literature and become an interesting problem and testbed for program verification.

Apart from the importance of the algorithm in the literature, we had another motive in choosing this algorithm: the algorithm has already been modelled for binary graphs in an event-based approach and proved using Atelier B [Cle] by Abrial in [Abr03] without any explicit control structure. Abrial applied some *merging rules* to obtain the program structure from the model. This gives us a good mean to measure our approach against standard Event-B approach. Second, Leino in [Lei10] presented a full implementation and contracts for the algorithm in the Dafny language. This also allows us to compare the constructive approach against the analytical one.

The rest of this section outlines the development of the algorithm using our approach.

7.3.1 The Algorithm

The Schorr-Waite algorithm is a depth first graph traversal algorithm (illustrated in Figure 7.1) and can be specified informally as follows:

- The algorithm traverses a given graph from a given *top* node
- The graph is traversed depth-first
- The algorithm marks all reachable nodes from the given *top* node
- The backtracking structure (path to the top node) is stored in the graph by reversing the traversed edges
- Each time the algorithm backtracks it reverses the edge to its original direction

The algorithm has four main functional properties that a correct implementation should guarantee for any given unmarked graph (with a set of finite nodes) and any arbitrary *top* node:

1. The algorithm will mark all reachable nodes from the *top* node.
2. The algorithm will only mark reachable nodes from the *top* node.
3. The algorithm will terminate.
4. Upon the termination, the graph structure is the same as its original structure.

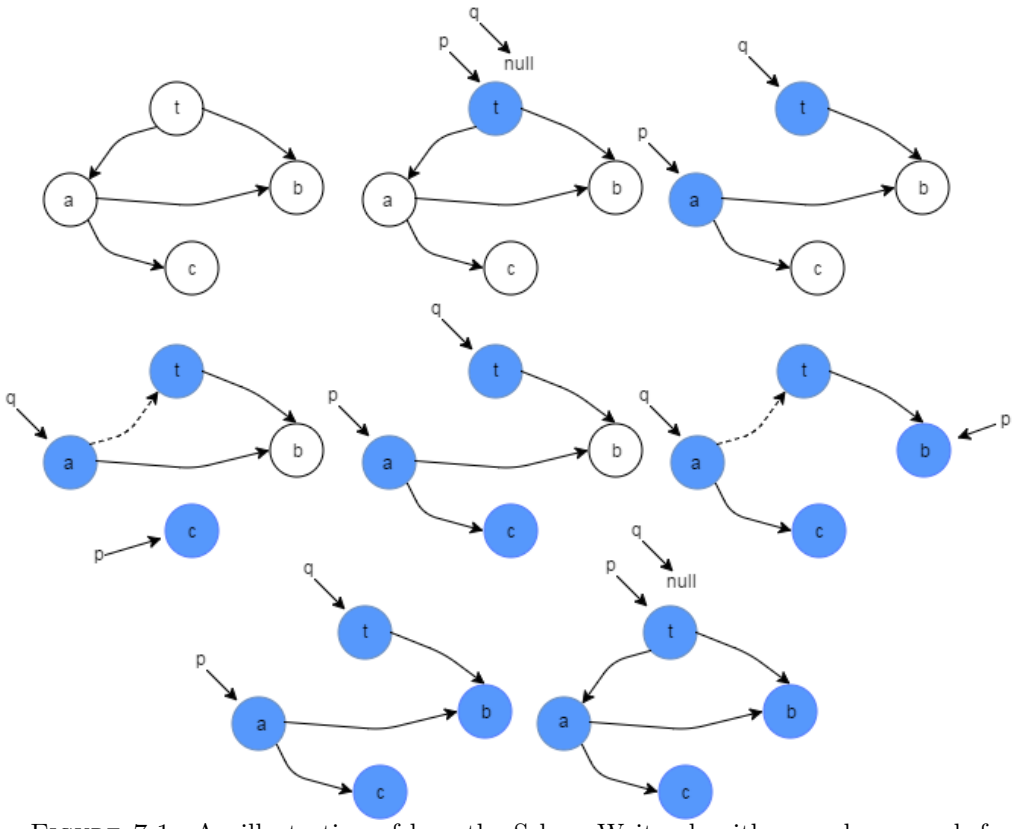

FIGURE 7.1: An illustration of how the Schorr-Waite algorithm marks a graph from node t. p and q are two auxiliary pointers to current and previous nodes, respectively. Dashed arrows represent the reversed edges used in the backtracking phase. The most recent traversed edge is represented using p and q and no explicit arrow is shown. The rest of the traversed edges are represented within the graph.

7.3.2 The Development of the Algorithm

As mentioned before, the algorithm has previously been modelled by Abrial [Abr03] for a binary graph to show how a sequential program can be constructed using an event-based approach, however the model was not refined to a concrete level. We borrowed the abstract specification of the algorithm from Abrial's work, modified it a bit, and developed and refined our version (where each node may have an arbitrary number of children) to a concrete level based on it.

Here we explain the model details briefly and concentrate on the refinement of the algorithmic structure using our scheduling language and rules[1].

7.3.2.1 Abstract Specification

In the abstract level the graph is specified using a constant binary relation (g) on a set N of nodes. To find all reachable nodes in the graph from the top node t it would be

[1]The full Event-B model of the algorithm can be found at: http://users.ecs.soton.ac.uk/md5g11/SchorrWaite/

enough to calculate the image of $\{t\}$ under transitive closure of g. Therefore the abstract model need have only one event (*marked*) whose execution will mark all the reachable nodes in one shot:

CONTEXT $c0$
SETS N
CONSTANTS g, c, t
AXIOMS
  axm1: $g \in N \leftrightarrow N$
  axm2: $c \in N \leftrightarrow N$
  axm3: $t \in N$
  axm4: $\forall s.(s \subseteq N \Rightarrow s \subseteq c[s])$
  axm5: $\forall s, x, y.(s \subseteq N \land x \mapsto y \in g \land x \in c[s] \Rightarrow y \in c[s])$
  axm6: $\forall s.(s \subseteq N \land g[s] \subseteq s \Rightarrow c[s] \subseteq s)$
  axm7: $finite(N)$

MACHINE $m0$ SEES $c0$
VARIABLES r
INVARIANTS
  inv1: $r \subseteq N$
EVENTS
Event $INITIALISATION$
then
  act1: $r :\in \mathbb{P}(N)$
End

Event $marked$
then
  act1: $r := c[\{t\}]$
End

In the context, g represents the graph and c is the transitive closure of g. Axioms $axm4$, $axm5$, and $axm6$ specify the properties of transitive closure (c) needed for specifying the desirable outcome of the algorithm. Machine $m0$ has an event called *marked* which calculates all reachable nodes from top node t using the image of the transitive closure under t in one step. The associated schedule with this specification is simply:

$$initialisation\,;marked$$

7.3.2.2 First Refinement

From this refinement we gradually introduce the algorithm and refine it to a concrete level. Here a new event (*marking*) is introduced to compute the reachable nodes and mark them (add them to set b) gradually by its repetition, the *marking* event selects the next unmarked reachable node from the set of marked nodes (b) non-deterministically:

INVARIANTS
  inv1: $b \subseteq N$
  inv2: $t \in b$
  inv1: $b \subseteq c[\{t\}]$

EVENTS
Event $INITIALISATION$
then
  act1: $r :\in \mathbb{P}(N)$
  act2: $b := \{t\}$
End

Event $marking$
any y
where
  grd1: $y \in g[b] \backslash b$
then
  act1: $b := b \cup \{y\}$
End

Event $marked$ refines $marked$
where
  grd1: $g[b] \subseteq b$
then
  act1: $r := b$
End

Event *marked* will be enabled when all the reachable nodes from marked nodes are marked. By applying rule 2 from Section 6.2.3 we can refine the abstract schedule:

$$initialisation\,;marked$$

$$\sqsubseteq_R$$

$$initialisation\,;marking^*\,;marked$$

Event *marking* is marked as *convergent* and by providing variant $card(N \backslash b)$ we can prove that it will terminate.

7.3.2.3 Second Refinement

This refinement specifies an important feature of the algorithm: backtracking. The backtracking structure is specified using an injective function *f*. Function f allows the algorithm to backtrack to the previous node when it cannot traverse further. Also a pointer to the current node p and a flag n are introduced. If n is true then all reachable nodes are marked. Two new events *backtracking* and *termination* are added to the model. The *backtracking* event changes the current node to the previous node if all the children of p are marked and $p \neq t$ and *termination* changes the flag n to *TRUE* if $p = t$ and all of the t children are marked or it does not have any. The *marking* event is also refined to update backtracking structure when it traverses from current node to its child:

INVARIANTS
  inv1: $p \in b$

inv2: $f \in (b \cup \{p\}) \backslash \{t\} \nrightarrow (b \cup \{t\}) \backslash \{p\}$
inv3: $n \in BOOL$
⋮

EVENTS
Event *marking* refines *marking*
any y
where
grd1: $g[\{p\}] \not\subseteq b$
grd2: $n = FALSE$
grd3: $y \in g[\{p\}] \backslash b$
then
act1: $b \coloneqq b \cup \{y\}$
act2: $p \coloneqq y$
act3: $f \coloneqq f \cup \{y \mapsto p\}$
act4: $path \coloneqq path \cup \{y\}$
End

Event *backtrack*
where
grd1: $g[\{p\}] \subseteq b$
grd2: $n = FALSE$
grd3: $p \neq t$
then
act1: $p \coloneqq f(p)$
act2: $f \coloneqq \{p\} \lhd f$
act3: $path \coloneqq path \backslash \{p\}$
End

Event *termination*
where
grd1: $g[\{p\}] \subseteq b$
grd2: $n = FALSE$
grd3: $p = t$
then
act1: $n \coloneqq TRUE$
End

Event *marked* refines *marked*
where
grd1: $n = TRUE$
then
act1: $r \coloneqq b$
End

In this level we mark *backtrack* as *convergent* and provide a variant (*path*). It can be trivially proved that the event terminates. We mark termination event as *anticipated* and will prove its termination in the next refinement. By applying rule 3 (loop body extension, see 6.2.3) we will have the following schedule refinement:

$$initialisation\,;\, marking^*\,;\, marked$$

$$\sqsubseteq_R$$

$$initialisation\,;\, (termination \,\square\, backtracking \,\square\, marking)^*\,;\, marked$$

7.3.2.4 Third and Fourth Refinements

Third refinement adds a boolean attribute *marked* to each node. This is specified as a total function from N to $BOOL$. The algorithm will set marked to true if the node is reached.

Fourth refinement changes the representation of the graph to a total function from nodes to a sequence of nodes. The sequence represents the children of each node:

CONTEXT $c1$ extends $c0$
CONSTANTS $graph$
AXIOMS
axm1: $graph \in N \rightarrow (\mathbb{Z} \rightarrowtail\!\!\!\!\!\!\rightarrow N)$
axm2: $\forall x.x \in N \Rightarrow ran(graph(x)) = ran(\{x\} \lhd g)$
axm3: $\forall x.x \in N \Rightarrow dom(graph(x)) = 0..card(ran(\{x\} \lhd g)) - 1$
axm4: $\forall x.x \in dom(graph) \Rightarrow finite(graph(x))$
axm5: $finite(graph)$

Function $graph$ is the new representation of the graph. Axioms $axm2$ and $axm3$ specify the relation between g and $graph$. In the third and forth refinements we only perform data refinement and replace g with $graph$ and b with $marked$:

EVENTS
Event $marking$ refines $marking$
any y
where
grd1: $\exists x.x \in ran(graph(p)) \land marked(x) = FALSE$
grd2: $n = FALSE$
grd3: $y \in ran(graph(p))$
grd4: $marked(y) = FALSE$
then
act1: $marked(y) \coloneqq TRUE$
act2: $p \coloneqq y$
act3: $f \coloneqq f \cup \{y \mapsto p\}$
act4: $path \coloneqq path \cup \{y\}$
End

Event $backtrack$ refines $backtrack$
where
grd1: $(p \in dom(graph) \land \forall nod.nod \in ran(graph(p)) \Rightarrow marked(nod) = TRUE) \lor p \notin dom(graph)$
grd2: $n = FALSE$
grd3: $p \neq t$
then
act1: $p \coloneqq f(p)$
act2: $f \coloneqq \{p\} \rhd f$
act3: $path \coloneqq path \backslash \{p\}$
End

Event $termination$ refines $termination$
where
grd1: $(p \in dom(graph) \land \forall nod.nod \in ran(graph(p)) \Rightarrow marked(nod) = TRUE) \lor p \notin dom(graph)$

grd2: $n = FALSE$
grd3: $p = t$
then
act1: $n := TRUE$
End

Event $marked$ refines $marked$
any m
where
grd1: $n = TRUE$
grd2: $m = \{nod.nod \in N \wedge marked(nod) = TRUE | nod\}$
then
act1: $r := m$
End

If in a level only data refinement is performed then the schedule will remain the same as the abstract one:

initialisation; (termination □ backtracking □ marking); marked*

7.3.2.5 Fifth Refinement

Until now the selection of the next child for marking was done non-deterministically. Now that we have the children of the current node in an ordered list, the selection of the next node can be done deterministically. To facilitate this, we add a new attribute (chV) to N to count the number of visited children of each node. This is specified by a total function from N to $\mathbb{Z}$. The *marking* event is refined to choose chVth child of p as the next node if it is not already marked and increases the chV of p by 1. A new event *nextChild* is also introduced to increase the chV of p by 1 in case that chVth child is already marked. The algorithm now backtracks if chV of p is equal to the number of p children if $p \neq t$. If chV of p is equal to the number of p children and also $p = t$ then the flag n is set to $TRUE$.

INVARIANTS
inv1: $chV \in N \rightarrow \mathbb{Z}$
inv2: $\forall nod, ch.nod \in dom(graph) \wedge ch \in graph(nod)[0..chV(nod) - 1] \Rightarrow marked(ch) = TRUE$
⋮

EVENTS
Event $marking$ refines $marking$
any y
where
grd1: $graph(p) \neq \emptyset$
grd2: $n = FALSE$
grd3: $chV(p) < card(graph(p))$
grd4: $marked(y) = FALSE$
lgrd: $y = (graph(p))(chV(p))$
then

act1: $marked(y) \coloneqq TRUE$
act2: $p \coloneqq y$
act3: $f \coloneqq f \cup \{y \mapsto p\}$
act4: $path \coloneqq path \cup \{y\}$
act5: $chV(p) \coloneqq chV(p) + 1$
End

Event *nextChild*
where
grd1: $graph(p) \neq \varnothing$
grd2: $n = FALSE$
grd3: $chV(p) < card(graph(p))$
grd4: $marked((graph(p))(chV(p))) = TRUE$
then
act1: $chV(p) \coloneqq chV(p) + 1$
End

Event *backtrack* refines *backtrack*
where
grd1: $(graph(p) \neq \varnothing \wedge chV(p) = card(graph(p))) \vee graph(p) = \varnothing$
grd2: $n = FALSE$
grd3: $p \neq t$
then
act1: $p \coloneqq f(p)$
act2: $f \coloneqq \{p\} \lhd f$
act3: $path \coloneqq path \backslash \{p\}$
End

Event *termination* refines *termination*
where
grd1: $(graph(p) \neq \varnothing \wedge chV(p) = card(graph(p))) \vee graph(p) = \varnothing$
grd2: $n = FALSE$
grd3: $p = t$
then
act1: $n \coloneqq TRUE$
End

Event *marked* refines *marked*
any m
where
grd1: $n = TRUE$
grd2: $m = \{nod.nod \in N \wedge marked(nod) = TRUE | nod\}$
then
act1: $r \coloneqq m$
End

As you can see the selection of the next node is now performed deterministically. By applying rule 3 (loop body extension), the schedule from last level can be refined:

$$initialisation\,;(termination \,\square\, backtracking \,\square\, marking)^*\,;marked$$

$$\sqsubseteq_R$$

$$initialisation\,;(nextChild \,\square\, termination \,\square\, backtracking \,\square\, marking)^*\,;marked$$

7.3.2.6 Sixth Refinement

This is another data refinement. In this level we introduce constant node *null*. *null* is not in the graph. We also introduce a new pointer q which points to the parent of the p if $p \neq t$. If $p{=}t$ then $q{=}null$. The *marked* and *backtracking* events are extended to update q where appropriate. We omit the model details here. The corresponding schedule to this level is the same as previous one:

initialisation;(nextChild $\Box$ *termination* $\Box$ *backtracking* $\Box$ *marking)*;marked*

7.3.2.7 Seventh Refinement

In this refinement finally we store the backtracking structure within the graph itself and remove it from the algorithm. We add a new attribute to N called cg for storing the list of a node's children. Events *marking* and *backtracking* are refined to reverse the traversed or backtracked edges:

INVARIANTS
- inv1: $cg \in N \rightarrow (\mathbb{Z} \twoheadrightarrow N)$
- inv2: $dom(graph) = dom(cg)$
- inv3: $cg(p) = graph(p)$
- inv4: $\forall x.x \in dom(graph) \Rightarrow dom(graph(x)) = dom(cg(x))$
- $\vdots$

EVENTS
Event *marking* refines *marking*
any z
where
- lv1: $z = cg(p)(chV(p))$
- grd1: $cg(p) \neq \varnothing$
- grd2: $n = FALSE$
- grd3: $chV(p) < card(cg(p))$
- grd4: $marked((cg(p))(chV(p))) = FALSE$

then
- act1: $marked(z) := TRUE$
- act2: $p := z$
- act3: $q := p$
- act4: $path := path \cup \{z\}$
- act5: $chV(p) := chV(p) + 1$
- act6: $cg(p) := (\{chV(p)\} \lhd cg(p)) \cup \{chV(p) \mapsto q\}$

End

Event *nextChild*
where
- grd1: $cg(p) \neq \varnothing$
- grd2: $n = FALSE$
- grd3: $chV(p) < card(cg(p))$
- grd4: $marked((cg(p))(chV(p))) = TRUE$

then
- act1: $chV(p) := chV(p) + 1$

End

Event *backtrack* **refines** *backtrack*
where
grd1: $(cg(p) \neq \varnothing \wedge chV(p) = card(cg(p))) \vee cg(p) = \varnothing$
grd2: $n = FALSE$
grd3: $p \neq t$
then
act1: $p := q$
act2: $q := cg(q)(chV(q) - 1)$
act3: $path := path \backslash \{p\}$
act4: $cg(q) := (\{chV(q) - 1\} \lhd cg(q)) \cup \{chV(q) - 1 \mapsto p\}$
End

Event *termination* **refines** *termination*
where
grd1: $(cg(p) \neq \varnothing \wedge chV(p) = card(cg(p))) \vee cg(p) = \varnothing$
grd2: $n = FALSE$
grd3: $p = t$
then
act1: $n := TRUE$
End

Event *marked* **refines** *marked*
any m
where
grd1: $n = TRUE$
grd2: $m = \{nod.nod \in N \wedge marked(nod) = TRUE | nod\}$
then
act1: $r := m$
End

Now that we have the concrete data structure and the actions of events are deterministic, non-deterministic choices or loops in the schedule can be refined to a concrete branch or loop. By applying rules 4 and 5 given in Section 6.3 we will have the final concrete schedule:

initialisation;(nextChild $\square$ *termination* $\square$ *backtracking* $\square$ *marking)*;marked*

$\sqsubseteq$

initialisation;

$while(n = FALSE)\{$

$if(chV(p) = card(cg(p)))\{$

$if(p = t)\{termination\}else\{backtrack\}$

$else\{$

$if(marked(p) = TRUE)\{nextChild\}else\{marking\}$

$\}$

};

marked

To ensure the correctness of the above program structure we have to apply forward propagation rules and eliminate original event guards. The following five conditions (one for each guarded event in the schedule) guarantees the safe elimination of the guards:

1. GEC for **termination** event:

   $I \wedge n = FALSE \wedge chV(p) = card(cg(p)) \wedge p = t \Rightarrow grd1 \wedge grd2 \wedge grd3$

2. GEC for **backtracking** event:

   $I \wedge n = FALSE \wedge chV(p) = card(cg(p)) \wedge \neg(p = t) \Rightarrow grd1 \wedge grd2 \wedge grd3$

3. GEC for **nextChild** event:

   $I \wedge n = FALSE \wedge \neg(chV(p) = card(cg(p))) \wedge marked(p) = TRUE \Rightarrow grd1 \wedge grd2 \wedge grd3 \wedge grd4$

4. GEC for **marking** event:

   $I \wedge n = FALSE \wedge \neg(chV(p) = card(cg(p))) \wedge \neg(marked(p) = TRUE) \Rightarrow grd1 \wedge grd2 \wedge grd3 \wedge grd4$

5. GEC for **marked** event:

   $I \wedge \neg(n = FALSE) \Rightarrow grd1$

In the above conditions I represents the model invariants and the right hand side of each implication ($grd1 \wedge grd2 \wedge ...$) is referring to conjunction of guards of the respective event. All the above conditions can be trivially proved.

7.4 Summary

In this chapter we presented the development of two algorithms using Event-B and our scheduling language. We illustrated the use of our scheduling language and refinement rules for derivation of the final algorithmic structures in Event-B refinement. The abstract scheduling language allowed us to capture the structure of the algorithms from the very first step. The refinement rules allowed us to systematically derive the concrete program structure from the abstract schedule. The Event-B model and proof of the Schorr-Waite algorithm presented in this chapter is very similar to the Abrial's model and proof presented in [Abr03]. The main difference between the two developments of the algorithm is the program derivation method that is employed. In the Abrial's

work two *merging rules* are used in order to derive the program structure (conditionals and loops) from the model. In contrast, we used our scheduling language in order to derive the program structure in successive steps. Our abstract scheduling language (non-deterministic choice and iteration) allowed us to capture the control flow between events easier. It also enabled us to refine the abstract schedule to a concrete level and establish a relation between the schedules in different levels of abstraction.

Chapter 8

Transforming Scheduled Event-B to Code and Contract

8.1 Introduction

In Chapter 3 we introduced an approach for transforming Event-B models to simple Dafny contracts (method's pre- and post-conditions), however that approach had limitations as discussed before. In Chapter 6 we augmented Event-B with a scheduling language to be able to perform algorithmic refinement in Event-B level to overcome the limitations of the previous approach. The proposed approach allows the modeller to impose explicit control flow on the execution of events with standard program constructs (i.e. branches and loops). A scheduled Event-B model eventually should be transformed to executable code in a low-level programming language. We provided a formal definition for `while` loops and `if` branches in Section 6.3. We assume that deterministic control structures (i.e. `if` branches and `while` loops) in Scheduled Event-B (SEB) and Dafny have the same definitions. Therefore the concrete algorithmic structure of SEB (i.e. `while` and `if`) can be transformed to Dafny directly. Although the control structures in SEB can be mapped to the Dafny control structures in a one to one mapping, the same does not apply to the events in a schedule. This is due to the fact that there is no ordering on the execution of actions of an event and they are considered to be executed simultaneously, i.e. an event is atomic. To be able to transform event actions to Dafny code, they should be sequentialised correctly. This sequentialisation can be performed and verified at the Event-B level. However it involves the overhead of introducing new events and invariants. To avoid this overhead, we have decided to perform sequentialisation of actions in a programming language (Dafny) which has a much more convenient structure for sequencing than Event-B. Dafny verification features will also allow us to verify the correctness of the sequentialisation of actions with respect to the Event-B event. For this purpose, in addition to the code, a number of Dafny contracts

should be generated in order to assist the verifier to verify the correctness of the event sequentialisation.

In the rest of this chapter we first discuss an approach for modelling structured data types and pointers in Event-B. Then we provide a number of rules for transforming a scheduled Event-B model to Dafny. We will also discuss the event sequentialisation and generation of assertions that will allow us to verify the correctness of the sequentialisation.

8.2 Modelling Structured Data Types in Event-B

The Schorr-Waite algorithm case study introduced in Chapter 7 involved structured data types and pointers. This section discusses the way that structured data types can be modelled in Event-B. The transformation rules required for Dafny code generation is introduced in the following sections.

The Event-B mathematical language does not support direct definition of structured data types. However, using the mathematical language of Event-B (e.g. carrier sets and functions), simple structured data types can be modelled conveniently. The approach introduced here is not new and is very similar to [EB06].

Here by *structured data type* we refer to a group of items of different types, very similar to C Structured Data types. A structured data type is represented by a carrier set in the context of a model and its fields can be declared using total functions where the domain is a subset of the carrier set which defines the structured data type and the co-domain is the field's type. The following example depicts the definition of node structured data type appeared in the Schorr-Waite case study:

CONTEXT C
SETS N

MACHINE M SEES C
VARIABLES $cg, chV, marked$
INVARIANTS
  inv1: $cg \in N \rightarrow (\mathbb{Z} \nrightarrow N)$
  inv2: $chV \in N \rightarrow \mathbb{Z}$
  inv3: $marked \in N \rightarrow \mathbb{BOOL}$

LISTING 8.1: Node structured data used in Event-B model of Schorr-Waite algorithm

In the above example, set N is the structured data type definition and variables cg, chV and $marked$ are the fields. Invariants $inv1$, $inv2$ and $inv3$ specify the type of each field. A general definition for a structured data type in Event-B can be given as follows:

CONTEXT C
SETS SD
CONSTANTS $null$
AXIOMS
  axm1: $null \in SD$

MACHINE M SEES C
VARIABLES $sd, f_1, f_2, ..., f_n$
INVARIANTS
  inv_frame: $sd \subseteq SD$
  inv_null: $null \notin sd$
  inv1: $f_1 \in sd \rightarrow TYPE_1$
  inv2: $f_2 \in sd \rightarrow TYPE_2$
  ⋮
  invn: $f_n \in sd \rightarrow TYPE_n$

LISTING 8.2: General definition of a structured data type in Event-B

In the context C, SD is the carrier set representing the structured data type and *null* is a constant representing the null value. In the machine M, variable sd is a subset of SD which contains all the non-null instances of the structured data type accessible to the machine (invariant *inv_frame*). Variables $f_1, ..., f_n$ represent the fields of the structured data type. Invariant *inv_null* ensures non-null pointer references. Invariants `inv1, ...,invn` declare the type of each field. The modeller should ensure that every new instance of the structured data type which is created by the machine is added to sd. It is straightforward to add new fields to an existing structured data type in later refinements by simply introducing new variables and invariants specifying the field's type.

Instances of the structured data type that are not created by the machine but can be accessed by it (like pointers passed to a method or global variables of a class) should be specified in the context and the data structure fields and sd set should be initialised accordingly. For example, p is a pointer to an instance of SD which is not created by the model:

CONTEXT C
SETS $SD, sd\_init, p, f_1\_init, ..., f_n\_init$
CONSTANTS $null$
AXIOMS
  axm1: $null \in SD$
  axm2: $sd\_init \subseteq SD$
  axm3: $null \notin sd\_init$
  axm4: $p \in sd\_init$
  axm_init_1: $f_1\_init \in sd\_init \rightarrow TYPE_1$
  ⋮
  axm_init_n: $f_n\_init \in sd\_init \rightarrow TYPE_n$
  axm_i_v_1: $f_1\_init(p) = E_1$
  ⋮
  axm_i_v_n: $f_n\_init(p) = E_n$

In the above context, sd_init is the initial set of instances, and $f_1\_init, ..., f_n\_init$ are the fields of the initial instances. These initial constants will be used in the $INITIALISATION$ event of the machine to initialise sd and other fields of the structured data type:

```
Event INITIALISATION
   THEN
      act0:  sd := sd_init
      act1:  f1 := f1_init
      act2:  f2 := f2_init
      ⋮
      actn:  fn := fn_init
```

If there is no existing instance of the structured data type that is accessed to or modified by the model then there is no need to have constants for SD fields in the context and sd variable and other field variables should be initialised with empty set.

A field of an instance may be updated using assignment operator in Event-B. For example, $f_1(p) \coloneqq x$ (where the type of x is $TYPE_1$) assigns x to field f_1 of instance p. Also the value of field (e.g. f_1) of an instance like p can be accessed by $f_1(p)$.

To summarise, in this section we did not introduce a new concept. We just provided an approach for modelling structured data types and pointers in Event-B using the existing Event-B mathematical language. In the transformation rules provided later in this chapter, we always assume that structured data types and pointers are modelled using the approach introduced here.

8.3 Transforming Models to Program Constructs

In the Chapter 6 we introduced Scheduled Event-B (SEB). SEB augments the Event-B language with a set of explicit control constructs (i.e. sequential compositions, choices, iterations, branches and loops) to allow the modeller to provide explicit control structure and refine it along with the normal Event-B refinement. A concrete SEB model should be transformed to a target language (in our case Dafny) to have executable code. Before the transformation can happen, a SEB model should be refined to a level that all abstract choices and iterations are replaced by concrete branches and loops, respectively. Also all non-deterministic assignments should be replaced by deterministic ones and all data structures should be replaced by a concrete counterpart in the target language.

In this section we introduce transformation rules required for generation of Dafny executable code from a concrete SEB model. We also provide some rules for generation of assertions required for verification of sequentialisation on event actions.

To formulate the transformation of SEB models to Dafny code and contracts, we define a function called $SEB2DFY$. The function accepts an Event-B model (M)(consisting of a machine and the context it sees) and a Schedule (S) and returns generated code and contracts:

$$\begin{aligned} SEB2DFY(M,S) \triangleq\ & SEB2DFY_{\text{sd}}(M) \\ & SEB2DFY_{\text{class}}(M,S) \end{aligned} \tag{8.1}$$

Functions $SEB2DFY_{\text{sd}}$ translates defined structured data types to a class in the target language, while $SEB2DFY_{\text{class}}$ defines a class including method declarations implementing the algorithm. They are discussed in the following sections. The input model M and schedule S are expected to be refined to a concrete level as explained earlier.

8.3.1 Transforming Structured Data Types to Dafny

Function $SEB2DFY_{\text{sd}}(M)$ transforms a structured data type defined in a SEB model into a Dafny class. If no structured data is defined then it does nothing. Consider the general definition of a structured data type given in Listing 8.2 and assume it is included in model M, then $SEB2DFY_{\text{sd}}(M)$ can be specified as follows:

$$\begin{aligned} SEB2DFY_{\text{sd}}(M) \triangleq\ & class\, SD\, \{ \\ & \quad SEB2DFY_{\text{var}}(f_1, inv1) \\ & \quad SEB2DFY_{\text{var}}(f_2, inv2) \\ & \quad \vdots \\ & \quad SEB2DFY_{\text{var}}(f_{\text{n}}, invn) \\ & \} \end{aligned} \tag{8.2}$$

We implement a structured data type as a class in Dafny. In the above rule, $SEB2DFY_{\text{var}}$ is a function which receives a variable and its associated typing invariant and generates equivalent Dafny variable declaration code:

$$SEB2DFY_{\text{var}}(f_1, inv1) \triangleq \texttt{var}\ f_1 : T; \tag{8.3}$$

where T is the variable type determined by the typing invariant ($inv1$).

8.3.2 Dafny Method Generation

The focus of SEB is on development and verification of sequential algorithms. SEB does not cover concepts like method calls or recursions. With this in mind, for the purpose of code generation, it would be an appropriate decision to map a SEB model to a class with a method implementing the algorithm based on the provided schedule S. Based on this decision, function $SEB2DFY_{\text{class}}$ will return a class with a method based on the model which was passed to it:

$$\begin{aligned} SEB2DFY_{\text{class}}(M,S) \triangleq\ & \texttt{class}\ mchn\{ \\ & SEB2DFY_{\text{mtd}}(M,S) \\ & \} \end{aligned} \tag{8.4}$$

Function $SEB2DFY_{\text{mtd}}(M,S)$ defines the way that the method should be generated:

$$\begin{aligned} SEB2DFY_{\text{mtd}}(M,S) \triangleq\ & \texttt{method}\ mchn(SEB2DFY_{\text{args}}(M)) \\ & SEB2DFY_{\text{pre}}(M) \\ & \{ \\ & SEB2DFY_{\text{var}}(v_1, inv_{v1}) \\ & SEB2DFY_{\text{var}}(v_2, inv_{v2}) \\ & \vdots \\ & SEB2DFY_{\text{var}}(v_{\text{n}}, inv_{vn}) \\ & SEB2DFY_{\text{alg}}(M,S) \\ & \} \end{aligned} \tag{8.5}$$

In the above class and method *mchn* is a placeholder for the name of the machine being transformed. If there is a value that the algorithm needs to receive in order to perform a specific task on it such as an unsorted array to be sorted, it is usually declared and specified in the model context using constants and axioms. In this case the constant is mapped to an input argument which is passed to the method and the axioms specifying it are transformed to method pre-conditions. Functions $SEB2DFY_{\text{args}}(M)$ and $SEB2DFY_{\text{pre}}(M)$ are used to generate method's input arguments and its necessary pre-conditions. Assume that we have a model containing machine *mchn* and a context with constants $a_1, ..., a_{\text{k}}$ where each constant is of type $T_1, ..., T_{\text{k}}$, respectively. Function $SEB2DFY_{\text{args}}(M)$ has the following definition:

$$SEB2DFY_{\text{args}}(M) \triangleq \ a_1 : T_1, ..., a_{\text{k}} : T_{\text{k}} \tag{8.6}$$

If model M has n axioms specifying input arguments then function $SEB2DFY_{\text{pre}}(M)$ has the following definition:

$$\begin{aligned} SEB2DFY_{\text{pre}}(M) \triangleq \ & \texttt{requires}\ SEB2DFY_{\text{pred}}(axm_1) \\ & \vdots \\ & \texttt{requires}\ SEB2DFY_{\text{pred}}(axm_{\text{n}}) \end{aligned} \tag{8.7}$$

Function $SEB2DFY_{\text{pred}}$ transforms an Event-B predicate to its Dafny equivalent. Finally, $SEB2DFY_{\text{alg}}(M, S)$ generates the implementation and necessary contracts. This function will be discussed in details in the rest of this chapter.

8.3.3 Algorithm Generation

An important step in transforming a SEB model to Dafny code is the generation of the code implementing the algorithm. Function $SEB2DFY_{\text{alg}}(M, S)$ formulates this step. Schedule S contains key information about the algorithmic structure of model M. A schedule is usually comprised of a number of sub-schedules (which are either a control structure or a single event) ordered using sequential composition operator:

$$S \triangleq S_1 \ ; ... \ ; S_{\text{n}}$$

If a schedule is comprised of a number of sub-schedules like the above, then function $SEB2DFY_{\text{alg}}(M, S)$ is defined as follows:

$$\begin{aligned} SEB2DFY_{\text{alg}}(M, S) \triangleq \ & SEB2DFY_{\text{alg}}(M, S_1) \\ & \vdots \\ & SEB2DFY_{\text{alg}}(M, S_{\text{n}}) \end{aligned} \tag{8.8}$$

As mentioned before, a sub-schedule may be a control structure (branch or loop) or an event. The general form of a branch sub-schedule is as follows:

$$S_{\text{i}} \triangleq \texttt{if(}c_1\texttt{)\{}\ s_1\ \texttt{\}}\ \texttt{elseif(}c_2\texttt{)\{}\ s_2\ \texttt{\}}\ ...\ \ \texttt{else\{}\ s_{\text{n}}\ \texttt{\}}$$

where $c_1, ..., c_{\text{n-1}}$ are branch conditions (in the form of Event-B predicates) and $s_1, ..., s_{\text{n}}$ are schedules. In this case the definition of $SEB2DFY_{\text{alg}}(M, S_{\text{i}})$ is as follows:

$$\begin{aligned} SEB2DFY_{\text{alg}}(M, S_{\text{i}}) \triangleq\ & \text{if}(SEB2DFY_{\text{pred}}(c_1))\{ \\ & \quad SEB2DFY_{\text{alg}}(M, s_1) \\ & \} \\ & \text{elseif}(SEB2DFY_{\text{pred}}(c_2))\{ \\ & \quad SEB2DFY_{\text{alg}}(M, s_2) \\ & \} \\ & \vdots \\ & \text{else}\{ \\ & \quad SEB2DFY_{\text{alg}}(M, s_{\text{n}}) \\ & \} \end{aligned} \tag{8.9}$$

If sub-schedule S_{j} is a loop then it has the following general form:

$$S_{\text{j}} \triangleq \texttt{while(}c\texttt{)\{ }s\texttt{ \}}$$

where c is the loop condition and s is a schedule representing the body of the loop. The definition of $SEB2DFY_{\text{alg}}(M, S_{\text{j}})$ is as follows:

$$\begin{aligned} SEB2DFY_{\text{alg}}(M, S_{\text{j}}) \triangleq\ & \text{while}(SEB2DFY_{\text{pred}}(c))\{ \\ & \quad SEB2DFY_{\text{alg}}(M, s) \\ & \} \end{aligned} \tag{8.10}$$

Now that we defined $SEB2DFY_{\text{alg}}$ for branches and loops, we need one more definition for the case that a (sub-)schedule is only a single event. This case will be discussed in the next section in details.

8.3.4 Events to Sequential Statements

The most basic component of a schedule is an event. Event-B events usually have a number of guards and actions. In Chapter 6 we showed that a schedule guarantees that event guards hold right before the execution of an event if the guard elimination condition for that event is satisfied. If all the guard elimination conditions are satisfied then we can eliminate all the event guards safely. Since event actions are assumed to be executed

simultaneously in Event-B, no ordering is assumed between them. Transformation of an event to code involves sequentialisation of event actions and imposing a suitable sequencing on execution of them using sequential composition. This section discusses the transformation of an event to code and contracts. The contracts will be used for proving the correctness of the sequentialisation.

8.3.4.1 Sequentialisation of an Event: Preparation Step

Event-B events are considered to be executed atomically. Because actions of an event are translated to assignments in the target language and they will be executed sequentially rather than simultaneously then a suitable order should be imposed on the actions in the target language. As mentioned above, transformation of an event to code involves sequentialisation of its actions. Sequentialisation of an event is the process of scheduling the execution of the actions of the event one after the other (as opposed to simultaneous execution) in a way that the same result is yielded, i.e. sequentialised actions change the state of the model in the same way that their simultaneous execution do.

The following simple example illustrates the problem. Assume that we have the following event which swap the value of two variables:

Event evt_1
where
  grd1: $g_1(v)$
then
  act1: $v_1 := v_2$
  act2: $v_2 := v_1$
End

The effect of the execution of the above event can be described using its before-after predicate:

$$v_1' == v_2 \wedge v_2' == v_1$$

where primed and unprimed variables denote the value of the variables after and before the execution of the event, respectively. To be able to execute the actions of the above event in a programming language sequentially and achieve the same result as the event, we need to introduce an auxiliary variable:

```
var temp := v1;
v1 := v2;
v2 := temp;
```

Here with the help of *temp* variable we managed to transform the actions of the event to a sequential program. In the above code, the ordering between assignments is important and any other ordering will result in a different effect on the state. To remove

the ordering constraint between the execution of the actions we can introduce another auxiliary variable to the program:

```
var temp1 := v1;
var temp2 := v2;
v1 := temp2;
v2 := temp1;
```

Now that the right-hand side of both actions (lines 3 and 4) are independent from each other, the ordering between them is irrelevant. A simple solution to sequentialising the actions of an event is to make the right-hand side of actions independent from the variables that are modified by the same event. In the preparation step, for practical reasons, we chose to introduce and formulate the required auxiliary variables in Event-B level using event parameters. Assume that we have the following event:

```
Event evt
where
   G
then
   act1:  v1 ≔ E1(v1, ..., vn)
   ⋮
   actn:  vn ≔ En(v1, ..., vn)
End
```

where G represents event guards. We introduce a parameter for each variable and then replace the variable in the right-hand side of the actions with its parameter counter part:

```
Event evt
any p1, ..., pn
where
   G
   lcg1:  p1 = v1
   ⋮
   lcgn:  pn = vn
then
   act1:  v1 ≔ E1[v1, ..., vn/p1, ..., pn]
   ⋮
   actn:  vn ≔ En[v1, ..., vn/p1, ..., pn]
End
```

In the above event, $E_i[v_1, ..., v_n/p_1, ..., p_n]$ means that variables $v_1, ..., v_n$ are replaced by $p_1, ..., p_n$ in expression E_i. While the above event is the same as the previous one, it can be sequentialised without imposing any specific ordering on the actions because the right-hand side of all actions are now independent from the variables that the event modifies.

By default, Event-B does not allow a variable to appear on the left-hand side of actions of an event more than once. However, when we are dealing with structured data types and pointers, a variable may appear on the left-hand side of more than one action in the form of an argument of a function defining a structured data type field. For instance, consider the following:

```
Event update_pointer
any x
where
   lcg1:  x = p
then
   act1:  p := q
   act2:  q := a(q)
   act3:  a(q) := a(x)
End
```

where p and q are pointers to instances of a structured data type (SD) which has field a that itself is a pointer to an instance of the same SD. Although in the above event right-hand side of each action is independent from the previous actions, but still direct sequentialisation of these actions would not result in a correct implementation. This is because `act2` is updating the value of pointer q while `act3` is updating field a of the instance that q is pointing to right before the execution of the event. As soon as `act2` executed in a programming language we lose the track of the instance that it is pointing to. Therefore, if `act3` is executed sequentially after `act2`, it updates another instance of the SD which was not intended.

To deal with this problem, we need to impose an ordering constraint based on the left-hand side of an action: if a pointer and also the fields of the instance that the pointer is pointing to, are updated by the event then actions performing the field updates should appear before the action updating the value of the pointer. For instance, the actions of the event in the previous example should be reordered as follows:

```
Event update_pointer
any y
where
   lcg1:  y = a(q)
then
   act3:  a(q) := a(p)
   act1:  p := q
   act2:  q := y

End
```

To sum up, the preparation step, which should be carried out by the modeller, has to make the right-hand side of actions independent from the variables updated by the event and impose an ordering constraints on the event actions. The ordering constraint should impose the following constraint on the actions: if the event modifies a pointer variable and there are actions modifying fields of an instance using that pointer variable, then actions modifying the fields should appear before the action that modifies the value of the pointer variable. After the preparation step is done, the general form of an event will be as follows:

```
Event evt
any p_1, ..., p_n
where
   G
   lcg1:  p_1 = v_1[(x_1)]
   ⋮
   lcgn:  p_n = v_n[(x_n)]
then
   act1:  v_1[(x_1)] := E_1[v_1[(x_1)], ..., v_n[(x_n)]/p_1, ..., p_n]
   act2:  v_2[(x_2)] := E_2[v_1[(x_1)], ..., v_n[(x_n)]/p_1, ..., p_n]
   ⋮
   actn:  v_n[(x_n)] := E_n[v_1[(x_1)], ..., v_n[(x_n)]/p_1, ..., p_n]
End
```

LISTING 8.3: General form of an event after preparation step

where (x_i) appears if the action is updating a field (v_i) of an instance and $x_i \in V \setminus \{v_1, ..., v_{i-1}\}$ and V is the set of all the model variables.

8.3.4.2 Sequentialisation of an Event: Prepared Event to Dafny

In Section 8.3.3 we presented the definition of $SEB2DFY_{\text{alg}}(M, S)$ when S is either a branch or a loop. This section deals with the case that $S \triangleq evt$ where evt represents a single event. $SEB2DFY_{\text{alg}}(M, evt)$ can only be called when the preparation step is finished. Assume that the event evt has the same definition as Listing 8.3 then:

$$
\begin{aligned}
SEB2DFY_{\text{alg}}(M, evt) \triangleq\ & SEB2DFY_{\text{ghost}}(M, evt) \\
& SEB2DFY_{\text{lcvar}}(p_1, p_1 = v_1) \\
& \vdots \\
& SEB2DFY_{\text{lcvar}}(p_n, p_n = v_n) \\
& SEB2DFY_{\text{act}}(v_1[(x_1)] := E_1[v_1, ..., v_n/p_1, ..., p_n]) \\
& SEB2DFY_{\text{act}}(v_2[(x_2)] := E_2[v_1, ..., v_n/p_1, ..., p_n]) \\
& \vdots \\
& SEB2DFY_{\text{act}}(v_n[(x_n)] := E_n[v_1, ..., v_n/p_1, ..., p_n]) \\
& SEB2DFY_{\text{post}}(M, evt)
\end{aligned}
\tag{8.11}
$$

Functions $SEB2DFY_{\text{ghost}}$ and $SEB2DFY_{\text{post}}$ which appeared on the first and last lines of the above definition, respectively, are used for contract generation purposes which will be discussed in the next section. Function $SEB2DFY_{\text{lcvar}}$ receives a parameter and a guard specifying the value of the parameter and transforms them to a local variable (p) and an assignment $(p = F(v))$ in the Dafny program:

$$SEB2DFY_{\text{lcvar}}(p, p = v) \triangleq \begin{array}{l} var\ p : T; \\ p := v; \end{array} \tag{8.12}$$

where T is the type of variable p which should be the same as type of v. Function $SEB2DFY_{\text{act}}$ receives an action (a) in the form of $v[(x)] := E(v, p)$ and has the following definition:

$$SEB2DFY_{\text{act}}(a) \triangleq SEB2DFY_{\text{varlhs}}(v[(x)]) := SEB2DFY_{\text{exp}}(E); \tag{8.13}$$

$SEB2DFY_{\text{exp}}$ transforms an Event-B expression to Dafny based on the translations presented in Section 3.2.1. $SEB2DFY_{\text{varlhs}}(v[(x)])$ is a function that transforms the left-hand side of an action to Dafny code:

$$SEB2DFY_{\text{varlhs}}(v[(x)]) \triangleq [x.]v \tag{8.14}$$

8.4 Generation of Code Contracts

In Section 8.3.4 we discussed the sequentialisation of an event in detail but we did not discuss how we can prove the correctness of the sequentialisation step. In order to be able to verify the sequentialisation, along with the translation of the actions of each event, we generate assertions representing the expected behaviour of the program based on before-after predicate of those actions.

Before we continue to explain our approach for verifying the correctness of event sequentialisation, we need to justify why this step is done at Dafny level. Although it is possible to sequentialise an event in Event-B to impose a sequential order on the execution of its actions and prove its correctness, it involves the overhead of adding a number of new events, guards, and program counters and also extending the scheduling language and refinement rules proposed in Chapter 6. Due to this, performing event sequentialisation in a programming language designed for development of sequential programs seems to be a more appropriate choice than trying to sequentialise actions in Event-B level which involves the aforementioned overhead.

In Section 8.3.4, we showed that how an event is translated to a number of of sequential statements in Dafny. A program (or part of a program) in Dafny may be specified (annotated) using code contracts (method's pre- and post-conditions and assertions).

The Dafny verifier checks an annotated program text against its specification in order to prove that the program behaves as intended. In order to prove that an event is correctly transformed to Dafny code (sequentialised correctly), a number of code contracts should be generated from the event in the form of assertions.

As it was discussed before, the way that the state is changed by an Event-B event can be expressed by a before-after predicate. By transforming an event's before-after predicate to Dafny contracts, we will be able to verify the correctness of event sequentialisation. Functions $SEB2DFY_{\text{ghost}}$ and $SEB2DFY_{\text{post}}$ will generate the necessary ghost variables and assertions for verification of sequentialisation.

8.4.1 Ghost Variable and Assertion Generation

In this section we explain how we should generate assertions from Event-B events in order to verify the correctness of the sequentialisation. Assume that we have a model with a set of variables $v = \{v_1, v_2, .., v_n\}$ and an event evt:

```
Event evt
where
   G(v)
then
   act1:   v1 := E1(v)

End
```

After the execution of the above event, the value of variable v_1 is changed in the following way:

$$v_1' = E_1(v) \tag{8.15}$$

where v_1' is the value of variable v_1 after execution of evt. A block of code implements event evt correctly, if it has the same behaviour as the event, i.e. its execution establishes (8.15). If we want to verify that a block of code sequentialises the event actions correctly, then we need to generate assertions like (8.15) in Dafny based on event before-after predicates.

The challenge here is how to refer to before and after values (unprimed and primed variables) in an assertion in Dafny. A variable in a Dafny assertion always refers to the current value of the variable. So to be able to transform a before-after predicate like (8.15) to an assertion, we need to have access to the value of variables before execution of the block of code implementing the event. To illustrate this, assume that in the event evt $E_1(v) = v_1 + 1$. The following is the before-after of the event:

$$v_1' = v_1 + 1 \tag{8.16}$$

We transform the event *evt* and the above before-after predicate to Dafny code and an assertion using a ghost variable for storing the before value (unprimed variable) of variable v_1:

```
ghost old_v1 := v1;
v1 := v1 + 1;
assert v1 == old_v1 + 1;
```

Here we use ghost variable old_v_1 to keep the before value of variable v_1. A ghost variable is a variable that is used by the Dafny verifier and ignored at run time.

In the previous example we did not deal with updating pointers and fields of an instance of a structured data type. Here we present an example involving pointer manipulation. Assume that we have the following event *evt_pointer*:

Event *evt_pointer*
where
  $G(v)$
then
  act1: $p := q$
  act2: $q := b(p)$
  act3: $b(q) := a(p)$
  act4: $a(p) := a(b(q))$

End

Listing 8.4: An event manipulating pointers

where p and q are pointers to instances of the same structured data type SD with fields a and b of type SD, respectively. After that the preparation step is taken place, we have the following:

Event *evt_pointer*
any l_p, l_q, l_b_q, l_a_p
where
  $G(v)$
  lcg1: $l\_p = p$
  lcg2: $l\_q = q$
  lcg3: $l\_b\_q = b(q)$
  lcg4: $l\_a\_p = a(p)$
then
  act1: $b(q) := l\_a\_p$
  act2: $a(p) := a(l\_b\_q)$
  act3: $p := l\_q$
  act4: $q := b(l\_p)$

End

Listing 8.5: The event from Listing 8.4 after preparation step

As discussed in the previous section, we introduce an auxiliary variable for each variable that is being updated by the event and make all the right-hand side expressions of all actions independent from the variables modified by the event. The generated code based on the above event will have the following form:

```
var l_p := p;
var l_q := q;
var l_b_q := q.b;
var l_a_p := p.a;
q.b := l_a_p;
p.a := l_b_q.a;
p := l_q;
q := l_p.b;
```

Now, we need a number of assertions to be able to prove the correctness of the above code with respect to the event, i.e. to prove that the code changes the state in the same way that the event does. Events given in Listing 8.4 and Listing 8.5 are the same, i.e. they have the same effect on the state. The effect of the execution of event *evt_pointer* can be expressed using the following before-after predicate:

$$p' = q \wedge q' = b(p) \wedge b'(q) = a(p) \wedge a'(p) = a(b(q))$$

We transform the above before-after predicate to the following ghost variables and assertions:

```
ghost var old_p := p;
ghost var old_q := q;
ghost var old_b_p := p.b;
ghost var old_a_p := p.a;
ghost var old_a_b_q := q.b.a;

var l_p := p;
var l_q := q;
var l_b_q := q.b;
var l_a_p := p.a;
q.b := l_a_p;
p.a := l_b_q.a;
p := l_q;
q := l_p.b;

assert p == old_q;
assert q == old_b_p;
assert old_q.b == old_a_p;
assert old_p.a == old_a_b_q;
```

Based on the unprimed variables appeared in the previous before-after predicate we generated five ghost variables. We treated a field of a structured data type as a single variable when we generated a ghost variable to keep its before data. For instance, to store the before value of $a(b(q))$ we generated variable *old_a_b_q*.

Function $SEB2DFY_{\text{ghost}}$ formulates the generation of the required ghost variables. It receives the model and a specific event, and works directly on the event before-after predicate and generates one ghost variable for every unprimed variable and initialise it with the value of the unprimed variables. For before values of fields of structured data types, the function generates a single variable and initialises it with the value of the field.

The four generated assertions in the above code are implicitly conjoined. For practical reasons, we decided to generate one `assert` statement per each action. The assertions are yielded by replacing unprimed variables in the before-after predicate with their ghost counterparts. Function $SEB2DFY_{\text{post}}$ formulates this.

8.4.2 Event Transformation Example

In this section we apply function $SEB2DFY_{alg}$ to event *marking* from the Schorr-Waite case study presented in Section 7.3. Before we transfer the event to code and contracts, we perform preparation step. Then we generate required ghost variables and assertions. Recall the *marking* event:

`Event` *marking* `refines` *marking*
`where`
  `grd1:` $cg(p) \neq \varnothing$
  `grd2:` $n = FALSE$
  `grd3:` $chV(p) < card(cg(p))$
  `grd4:` $marked((cg(p))(chV(p))) = FALSE$
`then`
  `act1:` $marked(cg(p)(chV(p))) := TRUE$
  `act2:` $p := cg(p)(chV(p))$
  `act3:` $q := p$
  `act4:` $chV(p) := chV(p) + 1$
  `act5:` $cg(p) := (\{chV(p)\} \lhd cg(p)) \cup \{chV(p) \mapsto q\}$
`End`

Listing 8.6: *marking* event before preparation step

Here we removed the auxiliary variable used in the event in Section 7.3 (replaced the variable by its value) and also removed the action updating *path* variable because *path* is an abstract variable and can be removed from the final refinement level. After preparation step the above event will have the following form:

```
Event marking
any l_marked_cg_p_chV_p, l_p, l_q, l_chV_p, l_cg_p
where
   G
   lcg1:  l_marked_cg_p_chV_p = marked(cg(p)(chV(p)))
   lcg2:  l_p = p
   lcg3:  l_q = q
   lcg4:  l_chV_p = chV(p)
   lcg5:  l_cg_p = cg(p)
then
   act1:  marked(cg(p)(chV(p))) := TRUE
   act4:  chV(p) := l_chV_p + 1
   act5:  cg(p) := ({l_chV_p} ⩤ l_cg_p) ∪ {l_chV_p ↦ l_q}
   act2:  p := l_cg_p(l_chV_p)
   act3:  q := l_p
end
```

Listing 8.7: *marking* event after preparation step

The above event and its auxiliary variables can be transformed to Dafny code:

```
var l_marked_cg_p_chV_p := p.cg[p.chV].marked;
var l_p := p
var l_q := q
var l_chV_p := p.chV
var l_cg_p := p.cg

p.cg[p.chV].marked := true;
p.chV := l_chV_p + 1;
p.cg := l_cg_p[l_chV_p := l_q];
p := l_cg_p[l_chV_p];
q := l_p;
```

Based on the *marking* event we have the following before-after predicate:

$$
\begin{aligned}
& marking'(\boxed{\text{cg(p)(chV(p))}}) = TRUE \wedge \\
& p' = \boxed{\text{cg(p)(chV(p))}} \wedge \\
& chV'(\boxed{\text{p}}) = \boxed{\text{chV(p)}} + 1 \wedge \\
& cg'(\boxed{\text{p}}) = (\{\boxed{\text{chV(p)}}\} \lhd \boxed{\text{cg(p)}}) \cup \{\boxed{\text{chV(p)}} \mapsto \boxed{\text{q}}\} \wedge \\
& q' = \boxed{\text{p}}
\end{aligned}
\tag{8.17}
$$

We boxed unprimed variables for which a ghost variable needs to be generated. The result of calling $SEB2DFY_{\text{ghost}}$ are as follows:

```
ghost var old_cg_p_chV_p := p.cg[p.chV];
ghost var old_p := p;
ghost var old_chV_p := p.chV;
ghost var old_cg_p := p.cg;
ghost var old_q := q;
```

With respect to the before-after predicate and the generated ghost variables, the function $SEB2DFT_{post}$ will result in generation of the following assertions:

```
assert old_cg_p_chV_p.marked == true;
assert p == old_cg_p_chV_p;
assert old_p.chV == old_chV_p + 1;
assert old_p.cg == old_cg_p[old_chV_p := old_q];
assert q == old_p;
```

The final output of $SEB2DFY_{alg}(M, marking)$ is as follows:

```
ghost var old_cg_p_chV_p := p.cg[p.chV];
ghost var old_p := p;
ghost var old_chV_p := p.chV;
ghost var old_cg_p := p.cg;
ghost var old_q := q;

var l_marked_cg_p_chV_p := p.cg[p.chV].marked;
var l_p := p
var l_q := q
var l_chV_p := p.chV
var l_cg_p := p.cg

p.cg[p.chV].marked := true;
p.chV := l_chV_p + 1;
p.cg := l_cg_p[l_chV_p := l_q];
p := l_cg_p[l_chV_p];
q := l_p;

assert old_cg_p_chV_p.marked == true;
assert p == old_cg_p_chV_p;
assert old_p.chV == old_chV_p + 1;
assert old_p.cg == old_cg_p[old_chV_p := old_q];
assert q == old_p;
```

8.5 Transforming Event-B Model of Schorr-Waite Algorithm to Dafny Code and Contracts

Schorr-Waite algorithm was discussed in details in the previous chapter. We showed how we can develop and verify the algorithm in different levels of abstraction using Scheduled Event-B. In the previous sections we discussed how a SEB model can be transformed to Dafny code. The following code is the result of transforming the concrete model of the algorithm to code:

```
1  class N{
2  var cg : seq<N>;
3  var marked : bool;
4  var chV:int;
5  }
6
7  class ShorrWaite{
8  ghost var NODES:set<N>;
9
```

```
10 method SchorrWaite(t:N)
11 requires t in NODES;
12 modifies NODES;
13 requires null !in NODES;
14 decreases *;
15 {
16 var p:N;
17 var q:N;
18 var n:bool;
19 n := false;
20 p := t;
21 q := null;
22 while(!n)
23 decreases *;
24 {
25         //frame condition
26         assume p in NODES && (q!=null ==> q in NODES);
27         assume forall x :: x in p.cg || (q!=null && x in q.cg) ==> x in NODES;
28         //invariants
29         assume p.chV >=0 && p.chV<=|p.cg|;
30         assume q!=null ==> q.chV >=0 && q.chV<=|q.cg|;
31         assume q!=null ==> q.chV > 0;
32         assume p != t ==> q!=null;
33
34         if(p.chV == |p.cg|)
35         {
36                 if(p==t)
37                 {
38                         //event: termination
39                         n := true;
40
41                         assert n == true;
42                 }
43                 else
44                 {
45                         //event: backtracking
46                         ghost var old_cg_q := q.cg;
47                         ghost var old_chV_q := q.chV;
48                         ghost var old_q := q;
49                         ghost var old_p := p;
50
51                         var l_q := q;
52                         var l_p := p;
53                         var l_cg_q := q.cg;
54
55                         q.cg := l_q.cg[l_q.chV-1:=l_p];
56                         p := l_q;
57                         q := l_cg_q[l_q.chV-1];
58
59                         assert old_q.cg == old_cg_q[old_chV_q-1 := old_p];
60                         assert p == old_q;
61                         assert q == old_cg_q[old_chV_q-1];
62                 }
63         }
64         else
65         {
66                 if(p.marked == true)
67                 {
68                         //event: nextChild
```

```
69                          ghost var old_chV_p := p.chV;
70                          ghost var old_p := p;
71
72                          var l_chV_p := p.chV;
73                          p.chV := l_chV_p + 1;
74
75                          assert old_p.chV == old_chV_p + 1;
76                  }
77                  else
78                  {
79                          //event: marking
80                          ghost var old_cg_p_chV_p := p.cg[p.chV];
81                          ghost var old_p := p;
82                          ghost var old_chV_p := p.chV;
83                          ghost var old_cg_p := p.cg;
84                          ghost var old_q := q;
85
86                          var l_marked_cg_p_chV_p := p.cg[p.chV].marked;
87                          var l_p := p;
88                          var l_q := q;
89                          var l_chV_p := p.chV;
90                          var l_cg_p := p.cg;
91
92                          p.cg[p.chV].marked := true;
93                          p.cg := l_cg_p[l_chV_p := l_q];
94                          p.chV := l_chV_p + 1;
95                          q := l_p;
96                          p := l_cg_p[l_chV_p];
97
98                          assert old_cg_p_chV_p.marked == true;
99                          assert old_p.cg == old_cg_p[old_chV_p := old_q];
100                         assert old_p.chV == old_chV_p + 1;
101                         assert p == old_cg_p_chV_p;
102                         assert q == old_p;
103                 }
104         }
105 }
106 }
107 }
```

The actual implementation of the algorithm and the contracts required for proving the event sequentialisation are together about 85 lines of code (excluding blank lines and comments). Note that we had to manually add some assumptions and contracts to the implementation in order to satisfy some properties like framing and absence of index out of bounds errors. Lines 8, 11-13, and 26-27 deal with the framing. Lines 29-32 are basically some of the model invariants that we had in our model which are needed to ensure that index out of bounds do not happen. Lines 14 and 23 inform the verifier to do not try to prove the termination of the algorithm because we have already done that in Event-B level. The rest of the code and contracts are obtained by applying $SEB2DFY$ functions to the prepared SEB model.

8.6 Event-B Approach vs Dafny Approach

The Schorr-Waite algorithm has been already specified and verified in Dafny by Leino [Lei10]. Despite some minor differences, the implementation presented in the previous section is similar to the Leino's implementation of the algorithm. However the main difference lies in the development and verification approach.

In the Scheduled Event-B approach the development started with a very abstract specification of the algorithm. We largely benefited from the rich mathematical language of Event-B in specifying the algorithm's high level properties. For instance, we used transitive closure to model graph marking in one shot. The abstract model was enriched through seven correctness preserving data and algorithmic refinement steps. We associated a schedule to each level of abstraction and refined the schedule to derive the final algorithmic structure. Finally, we used the rules introduced in this section to sequentialise the atomic events. We enjoyed the interactive proof offered by the Rodin platform. It provided us with a convenient way for discovering necessary model invariants and missing event guards.

In contrast, the Dafny approach starts with implementing the algorithm at the concrete level. Then the programmer should specify the desired outcome as the method post-conditions. The most difficult part of the specification at the code level in Dafny is the loop invariant discovery. In the Leino's implementation, there are 19 complicated loop invariants required for the verification. Leino concluded that the complexity of the loop invariants is because they are *so concrete* and a refinement-based approach for development is preferable [Lei10].

8.7 Summary

This chapter presented the final step required in constructing verified programs using Scheduled Event-B and Dafny. In this chapter we formulated the transformation of a Scheduled Event-B model to Dafny code and contracts. We discussed how an atomic Event-B event can be sequentialised in Dafny program and how we can generate code contracts from events in a way that the verification of the generated code and contracts implies the correctness of the implementation with regards to the abstract Event-B specification.

We also discussed an approach for modelling and verifying structured data types and pointer variables. Pointer manipulation in a program is often tricky because of its possible side effects. The fact that Event-B events are executed atomically and the limitations that are imposed on the event actions by the language make it easier to avoid common mistakes and side effects of pointer updates that may happen in a programming

language. Using the rules provided in this chapter for sequentialising events and also generation of assertions facilitate the correct transformation of models to code.

SEB2DFY functions which were presented in this chapter provide a basis for mechanising the transformation from Scheduled Event-B to Dafny. Sequentialisation of an event can become fully mechanised using the provided rules. The future step of this work is to build a tool to facilitate mechanised transformation.

Chapter 9

Conclusions

The main focus of this work is on developing verified sequential programs by using a combination of Event-B (constructive approach) and Dafny (analytical approach). In this work, we try to split the development and proof efforts between the two aforementioned methods. In doing so, we extended the Event-B language to accommodate new constructs needed for bridging the gap between the abstract model and concrete implementation and used Dafny to verify the code level properties.

In Chapter 3, we presented a method for transforming Event-B models to simple Dafny code contracts (i.e. method pre- and post-conditions). Our method provides a mechanism for grouping events together by extending the Event-B machines with a new construct called method constructor statement. Grouped events represent different cases of the same operation and are used to generate Dafny contracts. We proved that if the generated contracts are satisfied by an implementation in Dafny, the implementation also preserves the invariants of the abstract model. Therefore, there is no need to reprove invariant preservation in the Dafny level.

In Chapter 5, we presented a tool in the form of a Rodin plug-in which has been developed to facilitate automatic contract generation from an Event-B model. Given a machine and a number of method constructors the tool automatically generates relevant code contracts. The Rodin proof obligation generator is also extended to generate the new extra proof obligations (discussed in Section 3.3.4) that are needed to guarantee the soundness of the generated contracts. Two examples were provided in Chapter 4 to illustrate the transformation and validate our tool and transformation rules.

The aforementioned approach for transforming Event-B models to Dafny code contracts works fine for the Event-B model of algorithms that their implementations can satisfy the generated contracts without further assertions like loop invariants. However, if there is no implementation that can satisfy the contracts trivially then this approach is not efficient, since a considerable part of reasoning (reasoning about the algorithmic

structure) is left to the code level which is not always easy. One way to address this, is to derive and verify the algorithmic structure in refinement steps.

In Chapter 6, we presented an approach for derivation of algorithmic control structures in Event-B refinement. The introduced approach allows us to mix program refinement and data refinement using the existing Event-B tool support for representing and proving the data refinement. We augmented Event-B with a *scheduling language* language to make the control flow and algorithmic structure explicit. The scheduling language and the schedule refinement rules allow the modeller to derive the program structure in a stepwise manner. The scheduling language allows the modeller to introduce loops and non-deterministic choices in the abstract level. It also has familiar control constructs like if-statements and while-loops that will replace the non-deterministic structures in the concrete level. We presented a number of rules which will assist the modeller in refining an abstract schedule towards a concrete one. We also showed and proved how guard elimination conditions can guarantee that the events in the schedule will follow the explicit control flow and how we can remove original event guards.

In Chapter 7, we validated our approach of Chapter 6 by developing and verifying a simple sorting algorithm and the Schorr-Waite graph marking algorithm using Event-B and our scheduling language. The explicit control flow defined by the scheduling language allowed us to gain a better insight on the ordering of the events and the algorithmic structure from the abstract level and the given rules in Section 6.2.3 and Section 6.3 helped us in derivation of the final program structure in a stepwise manner.

In Chapter 8, we presented the final step required for generating executable code from Event-B and our scheduling language. We formulated the transformation of a scheduled Event-B model to Dafny code and contracts. We discussed how the actions of an atomic event can be sequentialised in a Dafny program. We introduced an approach for generation of Dafny contracts in the form of assertions in order to verify the correctness of the sequentialisation.

Scheduled Event-B does not support procedures (methods or functions) and procedure calls. The lack of a mechanism for defining procedures can potentially limit the use of the approach for developing medium and large size programs. It may also limit the re-usability of the developed small size programs.

Although the two case studies that were presented in Chapter 7 covered many aspects of developing a sequential program (including nested constructs and pointers) but still more examples are needed to be studied. More case studies can help us to gain better understanding on how efficient and effective the approach is. However, developing big algorithms with complex control structures can easily become very difficult to manage when a proper tool support is absent. A tool that enforces the correct use of the introduced refinement rules and generates the necessary proof obligations is essential for further validation of the approach discussed in Chapter 7 and Chapter 8.

9.1 Future Work

There is still room for improving the development of verified programs in Event-B. In future, the work presented in this thesis can be extended and improved in the following ways:

- The scheduling language presented in Chapter 6 can be extended to accommodate more complex program constructs such as procedure calls. It also can be extended to allow definition of more than one procedure for each model.
- Currently the concrete schedule conditions for conditionals and loops should be determined by the modeller manually. It should be possible to find new rules for discovering proper schedule conditions. Such rules can be used later for automation of condition discovery.
- To employ the full power of Event-B and the Rodin platform in developing verified programs the Rodin platform should be extended to support schedule refinement and automate the generation of extra proof obligations for enabledness and guard elimination conditions. One option is to extend the contract generator introduced in Chapter 5 to accommodate Scheduled Event-B as well.
- The transformation rules presented in Chapter 8 are specifically defined to translate scheduled Event-B to Dafny. The current set of the rules can be generalised and extended to facilitate the transformation of the models to other languages (e.g. JML-annotated Java code) as well. This might be achieved by first translating the models to a common intermediate language and then transforming the intermediate language to a program in the target language.
- The event sequentialisation rules can be extended and the process of the re-ordering and finding the auxiliary variables required for sequentialisation should be automated.
- The way that the structured data types and pointers are modelled in Event-B is effective. However our experience shows that the models containing structured data types are less readable and cumbersome. An alternative way for presenting those constructs should be investigated. One possible solution might be the use of theories and Theory Plug-in.
- The assertions generated from scheduled Event-B might not be satisfied by the Dafny verifier without encoding some of the invariants in the program text in the form of assumptions. In addition to that, some more auxiliary assumptions might be needed to satisfy the Dafny framing condition (See Section 8.5). Automatic generation of those assertions should be investigated.

References

[ABH+10] Jean-Raymond Abrial, Michael Butler, Stefan Hallerstede, ThaiSon Hoang, Farhad Mehta, and Laurent Voisin. Rodin: an open toolset for modelling and reasoning in Event-B. *International Journal on Software Tools for Technology Transfer*, 12(6):447–466, 2010.

[ABHV06] Jean-Raymond Abrial, Michael Butler, Stefan Hallerstede, and Laurent Voisin. An open extensible tool environment for Event-B. In Zhiming Liu and Jifeng He, editors, *Formal Methods and Software Engineering*, volume 4260 of *LNCS*, pages 588–605. Springer Berlin Heidelberg, 2006.

[Abr03] Jean-Raymond Abrial. Event based sequential program development: Application to constructing a pointer program. In *FME 2003: Formal Methods*, pages 51–74. Springer, 2003.

[Abr05] Jean-Raymond Abrial. *The B-book: assigning programs to meanings*. Cambridge University Press, 2005.

[Abr07] Jean-Raymond Abrial. Formal methods: Theory becoming practice. *Journal of Universal Computer Science*, 13(5):619–628, 2007.

[Abr10] Jean-Raymond Abrial. *Modeling in Event-B: system and software engineering*. Cambridge University Press, 2010.

[AFPdS11a] Jos Bacelar Almeida, Maria Joo Frade, Jorge Sousa Pinto, and Simo Melo de Sousa. An overview of formal methods tools and techniques. In *Rigorous Software Development*, pages 15–44. Springer, 2011.

[AFPdS11b] José Bacelar Almeida, Maria João Frade, Jorge Sousa Pinto, and Simao Melo de Sousa. *Rigorous software development: an introduction to program verification*. Springer, 2011.

[AH07] Jean-Raymond Abrial and Stefan Hallerstede. Refinement, decomposition, and instantiation of discrete models: Application to Event-B. *Fundamenta Informaticae*, 77(1):1–28, 2007.

[ALN+91] J. R. Abrial, M. K. O. Lee, D. S. Neilson, P. N. Scharbach, and I. H. Srensen. The B-method. In Sren Prehn and Hans Toetenel, editors, *VDM*

'91 Formal Software Development Methods, volume 552 of *LNCS*, pages 398–405. Springer Berlin Heidelberg, 1991.

[AP11] V. S. Alagar and K. Periyasamy. Vienna Development Method. In *Specification of Software Systems*, Texts in Computer Science, pages 405–459. Springer London, 2011.

[Bac88] Ralph-JR Back. A calculus of refinements for program derivations. *Acta Informatica*, 25(6):593–624, 1988.

[Bar97] John Barnes. *High integrity Ada: the SPARK approach*. Addison-Wesley Professional, 1997.

[BCC+05] Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joseph R. Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer*, 7(3):212–232, 2005.

[BKS88] R. J. R. Back and F. Kurki-Suonio. Distributed cooperation with Action Systems. *ACM Trans. Program. Lang. Syst.*, 10(4):513–554, October 1988.

[BLS05] Mike Barnett, K.RustanM. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In Gilles Barthe, Lilian Burdy, Marieke Huisman, Jean-Louis Lanet, and Traian Muntean, editors, *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, volume 3362 of *LNCS*, pages 49–69. Springer Berlin Heidelberg, 2005.

[BM13] Michael Butler and Issam Maamria. Practical theory extension in Event-B. In *Theories of Programming and Formal Methods*, pages 67–81. Springer, 2013.

[Bos10] Pontus Boström. Creating sequential programs from Event-B models. In *IFM*, pages 74–88. Springer, 2010.

[Bow01] Jonathan Peter Bowen. Z: A formal specification notation. In Marc Frappier and Henri Habrias, editors, *Software Specification Methods*, Formal Approaches to Computing and Information Technology FACIT, pages 3–19. Springer London, 2001.

[Bub07] Richard Bubel. The Schorr-Waite algorithm. In *Verification of Object-Oriented Software. The KeY Approach*, pages 569–587. Springer, 2007.

[But09] Michael Butler. Incremental design of distributed systems with Event-B. *Engineering Methods and Tools for Software Safety and Security*, 22:131, 2009.

[But13] Michael Butler. Mastering system analysis and design through abstraction and refinement. 2013.

[BW98] Ralph-Johan Back and Joakim Wright. *Refinement calculus: a systematic introduction*. springer Heidelberg, 1998.

[Cle] Clearsy. Atelier B, user and reference manuals, 2001.

[CLR12] N. Catano, K. R. M. Leino, and V. Rivera. The EventB2Dafny Rodin plug-in. In *Developing Tools as Plug-ins (TOPI), 2012 2nd Workshop on*, pages 49–54, 2012.

[CRW13] Néstor Cataño, Camilo Rueda, and Tim Wahls. A machine-checked proof for a translation of Event-B machines to JML. *arXiv preprint arXiv:1309.2339*, 2013.

[CW96] Edmund M. Clarke and Jeannette M. Wing. Formal methods: state of the art and future directions. *ACM Comput. Surv.*, 28(4):626–643, 1996.

[DBR15a] Mohammadsadegh Dalvandi, Michael Butler, and Abdolbaghi Rezazadeh. From Event-B models to Dafny code contracts. In Mehdi Dastani and Marjan Sirjani, editors, *Fundamentals of Software Engineering*, volume 9392 of *LNCS*, pages 308–315. Springer International Publishing, 2015.

[DBR15b] Mohammadsadegh Dalvandi, Michael J. Butler, and Abdolbaghi Rezazadeh. Transforming Event-B models to Dafny contracts. *ECEASST*, 72, 2015.

[DBR17] Mohammadsadegh Dalvandi, Michael Butler, and Abdolbaghi Rezazadeh. Derivation of algorithmic control structures in Event-B refinement. *Science of Computer Programming*, 148(Supplement C):49 – 65, 2017. Special issue on Automated Verification of Critical Systems (AVoCS 2015).

[Dij68] E.W. Dijkstra. A constructive approach to the problem of program correctness. *BIT Numerical Mathematics*, 8(3):174–186, 1968.

[Dij75] Edsger W Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, 1975.

[Dij76] Edsger W Dijkstra. *A discipline of programming*, volume 1. prentice-hall Englewood Cliffs, 1976.

[DMB08] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[DMS+09] M. Dahlweid, M. Moskal, T. Santen, S. Tobies, and W. Schulte. VCC: Contract-based modular verification of concurrent C. In *Software Engineering - Companion Volume, 2009. ICSE-Companion 2009. 31st International Conference on*, pages 429–430, May 2009.

[DRS95] Roger Duke, Gordon Rose, and Graeme Smith. Object-Z: A specification language advocated for the description of standards. *Computer Standards and Interfaces*, 17(56):511–533, 1995.

[EB06] Neil Evans and Michael Butler. A proposal for records in Event-B. In *FM 2006: Formal Methods*, pages 221–235. Springer, 2006.

[EB11] Andrew Edmunds and Michael Butler. Tasking Event-B: An extension to Event-B for generating concurrent code. *Programming Language Approaches to Concurrency and Communication-cEntric Software*, page 1, 2011.

[FBR14] Asieh Salehi Fathabadi, Michael Butler, and Abdolbaghi Rezazadeh. Language and tool support for event refinement structures in Event-B. *Formal Aspects of Computing*, pages 1–25, 2014.

[FHB+14] Andreas Fürst, Thai Son Hoang, David Basin, Krishnaji Desai, Naoto Sato, and Kunihiko Miyazaki. Code generation for Event-B. In *Integrated Formal Methods*, pages 323–338. Springer, 2014.

[FHY94] M. Fukagawa, T. Hikita, and H. Yamazaki. A mapping system from Object-Z to C++. In *Proceedings of 1st Asia-Pacific Software Engineering Conference*, pages 220–228, Dec 1994.

[FLS+08] John Fitzgerald, Peter Gorm Larsen, Shin Sahara, et al. VDMTools: advances in support for formal modeling in VDM. *ACM Sigplan Notices*, 43(2):3, 2008.

[Hal09] Stefan Hallerstede. Proving quicksort correct in event-b. *Electronic Notes in Theoretical Computer Science*, 259:47 – 65, 2009.

[Hal10] Stefan Hallerstede. Structured Event-B models and proofs. In *Abstract State Machines, Alloy, B and Z*, pages 273–286. Springer, 2010.

[Ham04] Ali Hamie. Translating the object constraint language into the Java modelling language. In *Proceedings of the 2004 ACM symposium on Applied computing*, pages 1531–1535. ACM, 2004.

[HL12] Stefan Hallerstede and Michael Leuschel. Experiments in program verification using Event-B. *Form. Asp. Comput.*, 24(1):97–125, 2012.

[HMLS09] CAR Hoare, Jayadev Misra, Gary T Leavens, and Natarajan Shankar. The verified software initiative: A manifesto. *ACM Comput. Surv*, 41(4):1–8, 2009.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, October 1969.

[Ili10] A Iliasov. On Event-B and control flow. In *TECHNICAL REPORT SERIES*, page 19. School of Computing Science Newcastle University, 2010.

[JLC15] Peter Würtz Vinther Jørgensen, Morten Larsen, and Luis Diogo Monteiro Duarte Couto. A code generation platform for VDM. *University of Newcastle-upon-tyne. Computing Science. Technical Report Series*, 2015.

[JLPC12] M Jastram, L Ladenberger, D Plagge, and J Clark. The Rodin handbook, 2012.

[LAB+06] Gary T Leavens, Jean-Raymond Abrial, Don Batory, Michael Butler, Alessandro Coglio, Kathi Fisler, Eric Hehner, Cliff Jones, Dale Miller, Simon Peyton-Jones, et al. Roadmap for enhanced languages and methods to aid verification. In *Proceedings of the 5th international conference on Generative programming and component engineering*, pages 221–236. ACM, 2006.

[Lap95] JC Laprie. Dependable computing and fault tolerance: Concepts and terminology. In *Fault-Tolerant Computing, 1995, Highlights from Twenty-Five Years., Twenty-Fifth International Symposium on*, page 2. IEEE, 1995.

[LB08] Michael Leuschel and Michael Butler. ProB: an automated analysis toolset for the B method. *International Journal on Software Tools for Technology Transfer*, 10(2):185–203, 2008.

[LBF+10] Peter Gorm Larsen, Nick Battle, Miguel Ferreira, John Fitzgerald, Kenneth Lausdahl, and Marcel Verhoef. The Overture initiative integrating tools for VDM. *SIGSOFT Softw. Eng. Notes*, 35(1):1–6, January 2010.

[LBR06] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. *SIGSOFT Softw. Eng. Notes*, 31(3):1–38, May 2006.

[Lei08] K Rustan M Leino. This is Boogie 2. *Manuscript KRML*, 178:131, 2008.

[Lei10] K Rustan M Leino. Dafny: An automatic program verifier for functional correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 348–370. Springer, 2010.

[LM10] K. Rustan M. Leino and Rosemary Monahan. Dafny meets the verification benchmarks challenge, 2010.

[LW14] K Rustan M Leino and Valentin Wüstholz. The Dafny integrated development environment. *arXiv preprint arXiv:1404.6602*, 2014.

[LY12] K. RustanM Leino and Kuat Yessenov. Stepwise refinement of heap-manipulating code in Chalice. *Formal Aspects of Computing*, 24(4-6):519–535, 2012.

[Mér09] Dominique Méry. Refinement-based guidelines for algorithmic systems. *Int. J. Software and Informatics*, 3(2-3):197–239, 2009.

[Mey92] Bertrand Meyer. *Eiffel: The Language*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992.

[Mey02] Bertrand Meyer. *Design by contract*. Prentice Hall, 2002.

[MM13] Dominique Mery and Rosemary Monahan. Transforming Event-B models into verified C# implementations. In *VPT 2013. First International Workshop on Verification and Program Transformation*, volume 16, 2013.

[Mor87] Joseph M Morris. A theoretical basis for stepwise refinement and the programming calculus. *Science of Computer programming*, 9(3):287–306, 1987.

[Mor88] Carroll Morgan. The specification statement. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 10(3):403–419, 1988.

[MRG88] Carroll Morgan, Ken Robinson, and Paul Gardiner. *On the refinement calculus*. Oxford University. Computing Laboratory. Programming Research Group, 1988.

[NH12] M. Najafi and H. Haghighi. An approach to animate Object-Z specifications using C++. *Scientia Iranica*, 19(6):1699 – 1721, 2012.

[NLL12] ClausBallegaard Nielsen, Kenneth Lausdahl, and PeterGorm Larsen. Combining VDM with executable code. In John Derrick, John Fitzgerald, Stefania Gnesi, Sarfraz Khurshid, Michael Leuschel, Steve Reeves, and Elvinia Riccobene, editors, *Abstract State Machines, Alloy, B, VDM, and Z*, volume 7316 of *LNCS*, pages 266–279. Springer Berlin Heidelberg, 2012.

[RC93] G-H Bagherzadeh Rafsanjani and S. J. Colwill. *From Object-Z to C++: A Structural Mapping*, pages 166–179. Springer London, London, 1993.

[RCn14] Víctor Rivera and Néstor Cataño. Translating Event-B to JML-specified Java programs. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, SAC '14, pages 1264–1271, New York, NY, USA, 2014. ACM.

[RZ06] S. Ramkarthik and Cui Zhang. Generating Java skeletal code with design contracts from specifications in a subset of Object-Z. In *5th IEEE/ACIS International Conference on Computer and Information Science and 1st IEEE/ACIS International Workshop on Component-Based Software Engineering,Software Architecture and Reuse (ICIS-COMSAR'06)*, pages 405–411, July 2006.

[SA92] J Michael Spivey and JR Abrial. *The Z notation*. Prentice Hall Hemel Hempstead, 1992.

[Sen92] C.T. Sennett. Demonstrating the compliance of Ada programs with Z specifications. In CliffB. Jones, RogerC. Shaw, and Tim Denvir, editors, *5th Refinement Workshop*, Workshops in Computing, pages 70–87. Springer London, 1992.

[SPHB11] Renato Silva, Carine Pascal, Thai Son Hoang, and Michael Butler. Decomposition tool for Event-B. *Software: Practice and Experience*, 41(2):199–208, 2011.

[ST07] P.H. Schmitt and I. Tonin. Verifying the mondex case study. In *Software Engineering and Formal Methods, 2007. SEFM 2007. Fifth IEEE International Conference on*, pages 47–58, Sept 2007.

[STW10] Steve Schneider, Helen Treharne, and Heike Wehrheim. A CSP approach to control in Event-B. In *Integrated formal methods*, pages 260–274. Springer, 2010.

[SW67] Herbert Schorr and William M Waite. An efficient machine-independent procedure for garbage collection in various list structures. *Communications of the ACM*, 10(8):501–506, 1967.

[WB07] Jim Woodcock and Richard Banach. The verification grand challenge. *J. UCS*, 13(5):661–668, 2007.

[WC01] Jim Woodcock and Ana Cavalcanti. A concurrent language for refinement. In *IWFM*, volume 1, page 5th, 2001.

[WC02] Jim Woodcock and Ana Cavalcanti. *The Semantics of Circus*, pages 184–203. Springer Berlin Heidelberg, Berlin, Heidelberg, 2002.

[Win90] J. M. Wing. A specifier's introduction to formal methods. *Computer*, 23(9):8–22, 1990.

[Wir96] Niklaus Wirth. Extended Backus-Naur Form (EBNF). *ISO/IEC*, 14977:2996, 1996.

[WLBF09] Jim Woodcock, Peter Gorm Larsen, Juan Bicarregui, and John Fitzgerald. Formal methods: Practice and experience. *ACM Computing Surveys (CSUR)*, 41(4):19, 2009.

[Wri09] Steve Wright. Automatic generation of C from Event-B. In *Workshop on integration of model-based formal methods and tools*, page 14. Citeseer, 2009.