FORMAL DEVELOPMENT OF A
TRAIN CONTROL SYSTEM USING EVENT-B

A thesis submitted to attain the degree of

DOCTOR OF SCIENCES of ETH ZURICH

(Dr. sc. ETH Zurich)

presented by

ANDREAS FÜRST

MSc ETH CS, ETH Zurich

born on 28.06.1981

citizen of Gunzgen SO

accepted on the recommendation of

Prof. Dr. David Basin, examiner
Prof. Dr. Michael Butler, co-examiner
Dr. Thai Son Hoang, co-examiner
Prof. Dr. Peter Müller, co-examiner

2015
Acknowledgements

First of all, I would like to thank my supervisor David Basin for the opportunity to do my PhD studies in his research group and for all his scientific advises during this time.

I would like to thank my co-supervisor Thai Son Hoang for all his support and the many discussions on the different topics of my work. My thanks go also to Michael Butler and Peter Müller for their willingness to act as co-examiners and their effort to evaluate my work.

I like to thank Jean-Raymond Abrial for his inspirational lectures, which introduced me to the world of formal methods and Event-B and led me to do research in this field.

Since practical relevance of my work has always been important to me, I am grateful that I was able to do my research in collaboration with an industrial partner. In this regard, I would like to thank Kunihiko Miyazaki and Naoto Sato for the many interesting discussions and their hospitality during my visits in Japan.

Further, I thank all the members of the Institute of Information Security for the pleasant working atmosphere. Special thanks go to my office mate and fellow sufferer Michael Schläpfer for his encouragement and companionship in and outside the office.

Most importantly, I like to thank my family and friends for their unconditional moral support and for reminding me that there is more in life than research.
Abstract

Developing error free software is a difficult task. The main technique used in industry to increase the quality of software is to find possible errors through testing. The confidence of the software’s correctness depends on the number of test cases that are checked. For every non-trivial piece of code, testing is always limited to a range of use cases and is never complete. For safety critical software as used in railway systems, where a small error can be life threatening, simple testing is not enough. Internationally binding regulations require for software in railway systems a safety integrity level (SIL), which cannot be established with testing alone. The corresponding standard proposes different methods to develop software that establishes SIL4, the highest of the defined levels. One of the highly recommended techniques for development is Event-B.

Event-B is a formal method that evolved from the B-method. The Rodin platform is the tool support to develop models in the Event-B language and to prove that required safety properties are fulfilled by the model. Event-B has mainly been used in academia for rather small examples. There is a community that developed several plug-ins for the Rodin platform to enlarge its functionality and to simplify the modelling task. ETH Zurich and Hitachi Yokohama Research Laboratory worked together in a research project to evaluate whether Event-B is applicable to the development process of complex and large-scale systems. As a case study, we developed a train control system in Event-B similar to Communication Based Train Control (CBTC) or European Train Control System Level 3 (ETCS L3). The train control system is developed at the system level, i.e., we model the train controller together with its environment.

Developing a system in Event-B follows two key ideas. First, we introduce the system’s details gradually into the model by stepwise refinement. Second, every safety property of the system is proved at a refinement level as abstract as possible which results in simpler proofs due to fewer details. We realised during the development of our case study that using Event-B’s stepwise refinement, as the only abstraction technique is not sufficient for complex systems. As an additional way of abstraction, we introduced Abstract Data Types (ADTs) in Event-B. The novelty of our approach is the combination of refinement and instantiation techniques to manage the complexity of systems under development. With ADTs, we model system components at an abstract level, specifying just their necessary properties, and we postpone the introduction of their concrete definitions to later development steps. As the ADTs become more concrete via instantiation, behavioural details of systems are expanded via refinement in a consistent manner according to the ADTs’ transformation.

To successfully use Event-B for developing real systems, the models must be translated to program code that can be compiled and runs on hardware. We developed a Rodin plug-in to generate C code from Event-B models that is correct-by-construction. In our approach, the correctness of the generated code is guaranteed by the combined use of well-definedness restrictions, refinement, and assertions.
By enforcing the well-definedness of the translated model, we prevent runtime errors that originate from semantic differences between the target language and Event-B, such as different interpretations of the range of integer values. Using refinement, we show that the generated code correctly implements the original Event-B model. We provide a simple yet powerful scheduling language that allows one to specify an execution sequence of the model’s guarded events where assertions are used to express properties established by the event execution sequence, which are necessary for well-definedness and refinement proofs.

Since refinement in Event-B only preserves safety properties, we have no guarantees about the liveness of our controller. Our generated code could do nothing and would still fulfil all safety properties. To obtain convincing evidence that the controller behaves as expected, we simulate the system. We developed a plug-in that provides an extended version of ProB’s simulation framework for the Rodin platform. Instead of simulating the Event-B model of the entire system, our version only simulates the events that represent the environment. For the controller part we run the generated code. The synchronisation between Pro-B’s simulation of the Event-B model and the running C code is handled by the plug-in. We investigate in the difficulties that arise when sharing data between the simulated environment and the controller and discuss how we solved them in our case study. Furthermore, we address timing issues that only appear when we split the Event-B model of the overall system into environment and controller and we thereby break Event-B’s implicit atomicity of events.

ten Eigenschaften und vertagen die Angabe der genauen Definitionen auf einen späteren Zeitpunkt in der Entwicklung. Mit dem Konkretisieren der ADTs durch Instanziierung werden auch die Details über das Systemverhalten mittels Verfeinerung erweitert. Die Erweiterung verläuft dabei konsistent zur Transformation der ADTs.


## Contents

1. Introduction 1
   1.1. Contributions ........................................ 4
   1.2. Outline ............................................. 5

2. Background 7
   2.1. The Event-B Modelling Method ............................. 7
   2.2. Case Study ........................................ 9
      2.2.1. Terminology ...................................... 9
      2.2.2. Conventional Interlocking System .................. 11
      2.2.3. Limitations of Conventional Interlocking Systems ... 13
      2.2.4. A New Generation of Train Control Systems .......... 14
   2.3. Related Work ....................................... 15

3. Requirements 17
   3.1. Topology ........................................... 18
      3.1.1. Sections ......................................... 18
      3.1.2. Network ......................................... 19
      3.1.3. Position ......................................... 20
   3.2. Operation ........................................... 20
      3.2.1. Interlocking System ................................ 21
      3.2.2. Trains ........................................... 21
      3.2.3. CBTC Equipment ................................... 22
      3.2.4. Zone Protection Range .............................. 24
      3.2.5. Computation of the LoA ............................ 24
   3.3. Safety ............................................... 26
   3.4. Summary ............................................. 26

4. Design of the Train Control System 27
   4.1. Logical Sections .................................... 28
   4.2. Layout of a Train .................................... 29
   4.3. Entering and Leaving Trains ............................ 30
   4.4. Radio Communication .................................. 31
   4.5. Emergency Situations .................................. 31
   4.6. Future Work ......................................... 32
      4.6.1. Ghost Trains .................................... 32
      4.6.2. Speed Profile .................................... 33
      4.6.3. Train Integrity ................................... 33
8.4. Conclusion .................................................. 124

9. Conclusion and Future Work .................................... 127

A. Formal Description of the Test Topology ......................... 135
   A.1. Section .................................................. 135
   A.2. Block .................................................. 135
   A.3. Train .................................................. 135
   A.4. Track .................................................. 135
   A.5. Route .................................................. 136
   A.6. Points .................................................. 136
   A.7. Conflicts .............................................. 137

B. Further Code Generation Examples ............................... 138
   B.1. Simple Search ......................................... 139
   B.2. Binary Search in a Sorted Array ......................... 141
   B.3. Minimum of an Array of Natural Numbers ................ 145
   B.4. Array Partitioning .................................... 149
   B.5. Simple Sorting ....................................... 155
   B.6. Array Reversing ...................................... 160
   B.7. Numerical Program Computing the Square Root .......... 163
   B.8. The Common Minimum of Two Sorted Arrays ............ 165
1. Introduction

Software developers experience from time to time that developing error free software is a non-trivial task. An established technique to find possible errors is testing. Testing is also the main provision used in industry to increase the quality of software. The confidence of the software’s correctness thereby depends on the number of test cases that are checked. For every non-trivial piece of code, testing is always limited to a range of use cases and is never complete. Even small errors in the control software of a safety critical system can become life threatening. Hence, testing as the single method to assure the absence of errors is unacceptable in a safety critical environment.

Railway systems are safety critical environments with a high potential for damage if a control system fails. To reduce the risk of safety relevant failures, internationally binding regulations for railway systems require that control software establishes the Safety Integrity Level 4 (SIL 4). This is the highest of five levels defined in the European Standard EN 50128 [23]. The standard proposes different methods to develop software that establishes SIL4. One of the highly recommended techniques for development is Event-B [3].

Event-B is a formal method that evolved from the B-method [1]. The Rodin platform [5] is the tool support to develop Event-B models and to prove required safety properties. So far, Event-B has mainly been used in academia to develop small models. In June 2011, Hitachi Yokohama Research Laboratory (Hitachi) and ETH Zurich (ETH) started a research project to analyse the applicability of formal methods, in particular Event-B, to develop large and complex systems as they are found in industry.

As a case study, Hitachi provided us with specifications for a train control system. The core functionality of the specified controller is based on real specifications and follows those of Communication Based Train Control (CBTC) or European Train Control System Level 3 (ETCS L3). Our goal was to develop the case study from the requirements to a concrete Event-B model and generate code for the controller that can run on hardware. The system should be developed in a generic way, such that the controller is applicable to a large variety of railway topologies. We decided to developed the train control system at the system level, i.e., we modelled the train controller together with its environment. This allows us to specify and prove safety properties of the overall system, which is stronger compared to proving safety properties of the controller software. The challenge of modelling an entire system including the environment is the increased complexity.

In our case study we focused on a proper development approach as proposed in [3] by first writing clear requirements before we started modelling the system to
generate code in the end. The formulation of clear and unambiguous requirements is an essential part of formally modelling a system. The main challenge in writing clear requirements from implementation oriented specifications is to identify the main functionality of the system and to abstract away irrelevant details.

Once the requirements are settled we can start modelling the system. Modelling in Event-B follows two key ideas. First, we introduce the system’s details gradually into the model by stepwise refinement. Second, every safety property of the system is proved at a refinement level as abstract as possible which results in simpler proofs due to fewer details. The different refinement levels are connected by a refinement relation, which guarantees that the proved safety properties are preserved. Due to this tight connection of the levels, design decisions have high impact on the further refinements. This is a general issue of Event-B’s top-down approach using refinement. A bad design decision in the beginning may take effect several refinement steps later and becomes challenging to repair. Because there are not many examples of complex Event-B models in the literature we had to make our design decisions based on trial and error, which is time consuming and sometimes frustrating.

The most challenging part when modelling a system in Event-B is to find the right level of abstraction to start with. Starting at a too concrete level usually results in difficult proofs. Abstracting away some details that are later introduced by refinement usually solves the problem. We experienced that for large and complex systems this strategy does not work anymore. Due to the system’s complexity, even the specification of abstract properties requires many details and results in difficult manual proofs. Hence, refinement as the only form of abstraction is not sufficient for complex systems. What we need is an additional way to abstract away details of the system.

We searched for different forms of abstraction and discovered the notion of Abstract Data Types (ADTs) [34]. An ADT is a mathematical model of a class of data structures. It is typically defined by a set of operations that can be performed on the ADT, along with a specification of their effect. We borrowed this idea and developed an approach to introducing ADTs in Event-B. We use contexts, which represent the static part of an Event-B model, to define ADTs in Event-B. Besides finding a way to introduce ADTs in Event-B we had to find suitable ADTs to describe the environment of the train control system. The danger of using ADTs is that one can easily define data structures and operations that are not implementable. Since the ADTs are axiomatically defined in Event-B contexts, the definition can include contradictions. Hence, we must check for every newly defined ADT whether we can instantiate it before we use it in the development. We use generic instantiation to transform ADTs to concrete data structures.

Once we refined our model to a concrete level, where all details are included and all required safety properties are proved, we are ready for the next step. We modelled the train control system with the goal to produce software that implements the modelled controller and runs on hardware. Event-B models can-
not run directly on hardware for different reasons. First, they still include non-
determinism such as non-deterministic assignments. Second, the different events
of an Event-B model have no distinct execution order. In Event-B, the execution
order of events is chosen non-deterministically. Third, the standard environment
of Event-B models does not reflect the real environment in which the software will
run. For example, while the range of valid integer numbers is infinite in Event-B,
this is certainly not the case for the integer range that can be represented by real
data.

While there exist several plug-ins to generate code from Event-B models [21,
36, 43], none of them guarantees the correctness of the generated code. We think
that this is an important property that a code generator for Event-B must fulfil.
Otherwise the benefit of formally modelling the system becomes questionable. We
developed an approach to generating code from Event-B models that is correct-
by-construction. Our approach follows three main ideas. First, we restrict the
Event-B model to a subset of Event-B’s language that is well-defined with re-
spect to the target language. The restriction reflects the available datatypes and
the valid operations in the target language. Furthermore, the restriction removes
remaining non-determinism in the events. Second, we provide a simple yet pow-
nerful scheduling language to define the execution order on the translated events
for the generated code. The scheduling language provides loops, branches, and
sequential composition. Third, we justify the correctness of the generated code
using refinement. For this purpose, we generate an Event-B machine with the
same behaviour as the generated code and prove that this machine refines the
original one. This refinement proof implies that also the generated code refines
the original machine. Hence, the generated code fulfils the same safety properties
as the Event-B model. There is the possibility of timing issues that we must
address. These issues originate from splitting the Event-B model of the overall
system into environment and controller and thereby breaking Event-B’s implicit
atomicity of events.

Event-B’s refinement relation guarantees the preservation of safety properties.
Liveness properties, however, can be destroyed by refinement. Although we put
our main focus on safety properties when we develop the train control system, we
want to check some liveness properties. In particular, we wanted to see whether
the generated code behaves according to our expectations. Note that restricting
a model too much during refinement may result in a system that does nothing,
which is still safe but not very useful. There exist approaches to define and prove
liveness properties in Event-B [30, 40]. These are mainly theoretical approaches
and are not applicable to our system that is separated into an Event-B model and
C code. Furthermore, we do not have formally specified liveness properties for
our system. We therefore follow a different approach that allows us to informally
check the behaviour of our system. We experienced in the past, that simulation
is a useful way to check that an Event-B model has not accidentally been re-
stricted too much. Since our system’s controller is represented by C code while
the environment is an Event-B model, we cannot use existing simulation plug-ins.
We developed a prototype plug-in based on ProB’s simulation framework that performs co-simulation. In our approach we extend ProB’s simulator such that it only simulates the environment part of our system and runs the generated code of the controller in parallel. With a concrete network topology that we specified for this purpose we instantiated the model of the environment and were able to show that the controller indeed behaves as desired. This concludes our development process to formally model complex systems in Event-B, which allows us to generate code that runs on hardware and is correct-by-construction.

1.1. Contributions

In the following, we describe our contributions that we present in this thesis.

- We formulate a set of clear and unambiguous requirements to formally model a train control system. We extracted the requirements from the specifications of a real system. We show that it is possible to describe the main functionality of a complex system with a manageable number of requirements if we abstract away unnecessary implementation details.

- We developed the train control system in Event-B based on the described requirements. We modelled the controller together with its environment in order to prove important safety properties at the system level. Following this approach, we proved that our controller prevents derailments and collisions of trains. We share our experience in modelling a train control systems by describing some design decisions that led to a successful development.

- We introduce the notion of Abstract Data Types (ADTs) in Event-B to cope with the complexity of large systems. Our approach provides an additional form of abstraction in Event-B that complements the technique of refinement. We use Event-B contexts to formalise ADTs and use generic instantiation to further concretise and thereby implement the ADTs. We show that systems specified with ADTs are more abstract and hence easier to verify than systems developed directly without them. With ADTs, we model system components at an abstract level and only specify the details that are necessary to prove the system’s safety properties. The introduction of the ADTs concrete definition is postponed to later development steps.

- We support the user to transform ADTs to concrete data structures by providing a plug-in for generic instantiation. The usage of this plug-in is not limited to the implementation of ADTs but can be used for any instantiation step during modelling. In our development, e.g., we used the plug-in also to instantiate the abstract network topology with a concrete test topology for simulation.

- We provide an approach to generating C code from an Event-B model that is correct-by-construction. In contrast to other code generation solutions for
Event-B, we prove that our generated code fulfils the same safety properties as the original model does. We specify a subset of the Event-B language that reflects the available data types and the allowed data type operations of the target language. We utilise Event-B’s notion of well-definedness to check whether a model complies with these restrictions. We provide a scheduling language to define the desired execution order of the translated events. A schedule is an arrangement of events combined with control flow statements such as loops and branches. We translate the events according to a set of translation rules that we defined and combine the resulting code snippets with control flow statements according to the defined schedule. To justify the correctness of the translation, we generate an Event-B machine that has the same behaviour as the generated code. We then prove that this generated machine is a refinement of the original machine, which implies that also the generated code is a refinement of the original machine.

- We developed a plug-in that supports users to perform the different steps of our code generation approach. The plug-in checks whether a model is sufficiently restricted to generate code for the target language. It further provides an intuitive editor to define schedules. In our code generation approach we use refinement to prove the correctness of the scheduled machine. We discovered that many of the generated proof obligations are superfluous. To simplify the proving step, we implemented in our plug-in an improved proof obligation generator that generates only the necessary proof obligations. We developed a strategy based on assigning pre- and postconditions to the schedule which generates a sufficient set of proof obligations. We prove the sufficiency of the generated proof obligations by showing that they imply the set of proof obligations for refinement.

- We use simulation to check our expectations about the system’s liveness. We developed a prototype plug-in based on ProB’s simulation framework that supports co-simulation. In contrast to existing approaches to simulating Event-B models, our plug-in simulates only the environment part of the Event-B model and runs the code that we generated for the controller in parallel. We specified a concrete test topology to simulate the developed train control system. To interact with the environment and control its activity we developed a graphical interface of the concrete topology that the plug-in links to the Event-B model of the environment. We investigate in the difficulties with shared data between the simulated model of the environment and the executed controller code.

1.2. Outline

In Chapter 2 we give a brief introduction to the Event-B modelling language and describe some modelling techniques that we used to develop the train control
system. We describe the basics of railway operations and analyse the limitations of conventional interlocking systems. We explain the concepts that are specified in CBTC and ETCS L3 to overcome these limitations.

We present in Chapter 3 the requirements that we extracted from the provided specifications. We group them into requirements that must be fulfilled by the environment, i.e. assumptions, and requirements that the controller must fulfil, i.e. system properties that we must prove in the Event-B model.

In Chapter 4 we describe some design decisions that we made during modelling and explain the reasons that led us to them. We provide useful information to model similar train control systems in the future. We describe some limitations of our model as well as limitations of the specified system in general and present our ideas to overcome these limitations.

We present our approach to introducing Abstract Data Types (ADTs) in Event-B in Chapter 5. We use a stack as an example to explain the idea of ADTs and to show how we apply this idea to Event-B using generic instantiation. We discuss the need for ADTs in our case study and present the different ADTs that we developed to model the train control system. We provide statistics to evaluate our approach.

Chapter 6 describes our approach to generating code for Event-B models. We provide an overview of the main concept of our approach and use the “cars on a bridge” example from [3] to describe the different steps that are involved to generate code. We discuss possible timing and atomicity issues caused by separating the controller from the environment and compare our work to other existing code generation approaches.

We analyse in Chapter 7 why the last step in our code generation approach, where we prove refinement, becomes difficult for large models. We present our solution to simplify the refinement proof, which is to filter out irrelevant proof obligations and to generate only the necessary ones. We show how our plug-in uses the schedule to determine the necessary proof obligations and we prove that this set is sufficient to guarantee the correctness of the generated code.

In Chapter 8 we describe how we use simulation to informally check liveness properties of our system. We present the concept of co-simulation and the design of the plug-in that we developed for this purpose. We describe the necessary preparation steps to simulate the modelled environment and evaluate the generated code of our developed train control system in terms of liveness.

We conclude our results and present some future work in Chapter 9. We discuss our contributions for the different steps that are involved when formally developing a system all the way from requirements to C code.
2. Background

This chapter gives first a brief overview of the Event-B modelling method. We restrict our explanations to the modelling techniques that we used in our development. In the second part of this chapter, we describe the train control system that we used as a case study. To highlight the novelty of this system we discuss the main functionality and the limitations of conventional interlocking systems.

2.1. The Event-B Modelling Method

Event-B [3] is a formal method, which is well suited to develop control intensive software. Event-B represents a further evolution of the classical B-method [1], which has been simplified and focused around the notion of events. Event-B has a semantics based on transition systems and simulation between such systems. We will not describe in detail Event-B’s semantics here; full details are provided in [3]. Instead, we will describe those modelling concepts that are important for our subsequent presentation.

Event-B models are organized in terms of two constructs: contexts and machines.

Contexts  Contexts specify the static part of a model and may contain carrier sets, constants, axioms, and theorems. Carrier sets are similar to types. Axioms constrain carrier sets and constants, whereas theorems express properties derivable from axioms. The role of a context is to isolate the parameters of a formal model (carrier sets and constants) and their properties, which are intended to hold for all instances.

Machines  Machines specify behavioural properties of Event-B models. Machines may contain variables, invariants, theorems, and events. Variables $v$ define the state of a machine.¹ They are constrained by invariants $I(v)$. Theorems are properties derivable from the invariants. Possible state changes are described by events. An event $e$ can be represented by the term

$$e \triangleq \text{any } t \text{ where } G(t,v) \text{ then } S(t,v) \text{ end}$$

¹When referring to variables $v$, parameters $t$, carrier sets $s$, or constants $c$, we usually allow for multiple variables, parameters, carrier sets, or constants, and we abuse the notation and treat them as sets of the corresponding objects.
where \( t \) is the event’s parameters, \( G(t, v) \) is the event’s guard (the conjunction of one or more predicates), and \( S(t, v) \) is the event’s action. The guard states the condition under which an event may occur, and the action describes how the state variables evolve when the event occurs. The event’s action is composed of one or more assignments of the form \( x := E(t, v) \), where \( x \) is a variable in \( v \). Assignments may also be non-deterministic. In this case, \( x \) becomes either an element of a set \( x :\in E(t, v) \), or it is assigned with a value that satisfies a predicate \( x :| P(x', t, v) \).

All assignments of an action \( S(t, v) \) occur simultaneously. A dedicated event without any parameters or guard is used for initialisation.

**Refinement**  
Refinement provides a means to gradually introduce details about the system’s dynamic behaviour into formal models [3]. A machine \( CM \) can refine another machine \( AM \). We call \( AM \) the abstract machine and \( CM \) the concrete machine. The abstract machine’s states are related to the concrete machine’s states by a gluing invariant \( J(v, w) \), where \( v \) are the variables of the abstract machine and \( w \) are the variables of the concrete machine. A special case of refinement (called superposition refinement) is when \( v \) is kept in the refinement, i.e., \( v \subseteq w \). Intuitively, any behaviour of \( CM \) can be simulated by a behaviour of \( AM \) with respect to the gluing invariant \( J(v, w) \).

Refinement can be reasoned about on a per-event basis. Each event \( e \) of the abstract machine is refined by one or more concrete events \( f \). Simplifying somewhat, we can say that \( f \) refines \( e \) if \( f \)’s guard is stronger than \( e \)’s guard (guard strengthening), and the gluing invariant \( J(v, w) \) establish a simulation of \( f \) by \( e \) (simulation).

**Instantiation**  
Instantiation is a technique for reusing models by providing values for their parameters. Since an Event-B model is parameterised by the carrier sets and constants, instantiation in Event-B [6, 41] amounts to instantiating the contexts.

Suppose we have a generic development, as depicted in Figure 2.1, with machines \( M_1, \ldots, M_n \) building a chain of refinements with carrier sets \( s \) and constants \( c \), constrained by axioms \( A(s, c) \) defined in contexts \( C_1, \ldots, C_n \). Suppose too that we want to reuse the development within another context \( D_1, \ldots, D_n \), specified by (concrete) carrier sets \( t \) and constants \( d \), constrained by axioms \( B(t, d) \). Let \( T(t) \), which must be an Event-B type expression, and \( E(t, d) \) be the instantiated values for \( s \) and \( c \) respectively. Given that the instantiation is correct, i.e.,

\[
B(t, d) \Rightarrow A(T(t), E(t, d))
\]

the instantiated development \( N_1, \ldots, N_n \) where \( s \) and \( c \) are replaced by their corresponding instantiated values is correct-by-construction.

For more details on instantiation in Event-B and its tool support see [6] and [41]. All instantiation steps described in this thesis were performed using the
generic instantiation plug-in [26] for the Rodin platform [5]. The plug-in is jointly developed by Hitachi and ETH. An example of generic instantiation can be found in Section 5.2.2 for implementing ADTs.

2.2. Case Study

At the beginning of our research project, Hitachi provided specifications of a train control system as an industrial size case study. The purpose of this case study was to give Hitachi a better understanding of formally developing systems in Event-B and to demonstrate that Event-B is applicable for developing large and complex systems.

The core functionality of the train control system follows that of Communication Based Train Control (CBTC) [32] and the European Train Control System Level 3 (ETCS L3) [24]. The special feature of CBTC and ETCS L3 is that trains can follow each other in braking distance. To understand the need for this new feature we first have to understand how train movements are organised and what the purpose of conventional interlocking systems is.

2.2.1. Terminology

There are many railway companies all over the world, which built their networks and interlocking systems separately. As a result, there exist different terminologies in the railway field. In this thesis, we will follow the terminology of the International Union of Railways (UIC) [19] where possible.

Track – A track consists of two rails that are connected by sleepers. Trains are moving on tracks.

Section – A track can be partitioned into sections. Every section of the main tracks is equipped with a track clear detection device to detect whether the section is free or occupied by one or more trains. This is realised by either a track circuit or axle counters.

Track circuit – A track circuit applies an electrical circuit to the rails. The rails are electrically isolated from each other. The control circuit is closed as long as
the section is free. When a train enters the section, its iron wheels and axles connect the two rails and bypass the circuit. This is detected as an occupation of the section.

**Axle-counter** – An axle-counter detects the presence of a wheel that passes by and its moving direction. With an axle-counter at both ends of a section, the associated logic unit counts all wheels that enter and leave the section. By comparing the number of entering and leaving wheels, the unit can tell whether the section is occupied or free.

**Crossing** – A crossing is an X-shaped junction of two tracks at the same level. It is not possible to change from one track to the other at a crossing. To prevent collisions, only one track may be used at a time. A crossing is not to be confused with a level crossing, which is an intersection of a track and a road.

**Points** – Points are a Y-shaped junction of a single track on one side with two other tracks on the other side. Points can dynamically change their position such that the single track is connected with either one of the other tracks.

**Entry Points** – A station usually consists of several tracks and multiple points connecting them. The outermost points on every side of a station are called entry points and build the borders of the station.

**Cab** – The place where the train driver controls the train is called driver’s cab or just cab.

**Cab signalling** – For trains driving at higher speeds, it is no longer guaranteed that a driver is able to visually recognise a trackside signal. Therefore commands are transmitted to the train and shown on a display in the cab. This practise is called cab signalling.

**Route** – A route is a set path for a train on the network. Within the route the train is protected against collisions with other trains. The points in the route are set and locked to prevent derailments. Routes are defined in terms of sections.

**Movement Authority** – A Movement Authority (MA) is similar to a route. It is established on top of a route and is assigned to exactly one train. In contrast to routes, MAs can be defined in terms of exact positions within sections. Furthermore, they can dynamically change as the associated train moves forward.

**Trackside Logic Controller** – A Trackside Logic Controller (TLC) is a physical device near the tracks that is in contact with the trains over radio and assigns MAs to them. It computes and manages the MAs for a number of trains in its area. In addition to its connection to the trains, it is also in contact with the
interlocking system. The TLC is the core of our case study for which we develop the controller software.

**On-Board Unit** – In CBTC and ETCS L3, every train is equipped with an On-Board Unit (OBU). This device establishes the radio connection to the TLC, computes the braking curve based on the received MA and the train’s physical properties, controls the train’s speed according to the speed limits and the computed braking curve, and applies the brakes if necessary. It provides driving information for the train driver on a graphical interface in the cab.

### 2.2.2. Conventional Interlocking System

When we refer to conventional interlocking systems, this includes all different kinds and generations of interlocking systems: mechanical, electro-mechanical, electrical, and electronic ones. Although built in different decades or even centuries and by many different manufacturers, they all have the same purpose and follow the same principles. An interlocking system is responsible to set and lock points and to establish routes. This increases safety in railway operation by reducing the risk of derailments and collisions to a minimum [42].

Collisions are often classified into collisions between trains and collisions of trains with other obstacles on the tracks. With the exception of secured level crossings, interlocking systems only prevent collisions between trains. Collisions between trains are further classified into head-on collisions, rear-end collisions, and flank collisions. The different scenarios are depicted in Figure 2.2.

Derailments are also classified with respect to their cause. Derailments can happen as the result of a collision, due to ignoring speed restrictions (especially in curves and on points), and when points are switched while a train is passing them. The interlocking system prevents derailments implicitly by preventing collisions and explicitly by locking points in their position. Derailments due to speeding are prevented by train protection systems. In ETCS L3, the OBU supervises the speed limits.

Conventional interlocking systems prevent collisions and derailments by establishing routes on the tracks of a network. A route has a direction and trains on a route are only allowed to move in the route’s direction. Routes are a logical layer between trains and tracks. With a few exceptions, a route is established for a single train and is released after this train has left the route. Train separation, realised by allowing only one train per established route, is the main principle to prevent collisions. Furthermore, all points within a route are locked by the interlocking system as long as the route remains established.

**Establishing a Route.** Before a route is established, the tracks corresponding to this route are checked for clearance to prevent collisions with other trains ahead. This check prevents the train from causing rear-end collisions with stationary trains. As mentioned before, there are a few exceptions where a route is not ex-
clusively for one train. When trains are shunting or a train enters an occupied station track to join another train we must allow more than one train on a route. In this case, the interlocking system does not check whether the tracks are free. The increased risk of collisions is mitigated by speed restrictions. The responsibility to prevent collisions is thereby transferred from the interlocking system to the train operators.

After checking that the tracks are free (if necessary), all points are set to the correct position according to the route that is established and are locked in this position. An untimely switching of the points while the train is passing them would cause a derailment. If points cannot be set to the required position, the establishment of the route is aborted.

Free tracks ahead do not guarantee that there will be no train in the future. It could be the case that a train is arriving from the opposite direction, but has not arrived yet. In this case, however, a route was established for this train, which has at least one common section with the route to establish. Routes are only useful for train separation if they do not overlap.

If two routes include the same points, but require them to be set to different positions, the simultaneous establishment of both routes is implicitly prevented by the points. Conflicting routes that only share sections must be excluded explicitly by the interlocking system. The principle of non-overlapping routes prevents head-on and flank collisions.

**Preserving a Route.** Once a route is established, the light signal at the beginning of the route turns from red to green to allow the train to enter the route. After the train has passed the light signal it turns back to red. The red light protects

Figure 2.2.: From top to bottom: head-on collision, rear-end collision, flank collision
the train on the route from following trains and possible rear-end collisions. The route remains established as long as the authorised train is moving on this route.

**Releasing a Route.** As soon as the train has left the route, the route is released and the corresponding tracks can be used to establish routes for other trains. Releasing the route also unlocks the points such that they can be set to new positions if required.

### 2.2.3. Limitations of Conventional Interlocking Systems

Every section of the main tracks is equipped with a track clear detection device. This device detects whether the corresponding section is free or occupied by one or more trains. As the device cannot detect what exact part of a section is occupied, sections build the smallest entities of the network.

Unfortunately, sections can be very long. The track between two stations is often one section that is several kilometres long. As a result, only one train can move between two stations at a time. The longer a section is, the lower is the network’s throughput. To increase the throughput, there are two solutions. Either the trains drive faster or the sections are shortened.

Let us assume the tracks are in a condition that allows higher speeds. It is still not the case that all trains are also able to drive faster. Throughout Europe, the network is usually used by mixed traffic: passenger and freight trains move on the same tracks. In general, freight trains are assembled of older rolling stock with lower speed limits than passenger trains. Therefore, increasing the speed to increase the throughput is limited by the slowest train [37].

Another issue with higher speeds is that at some point a train driver cannot safely recognise the aspects of the light signals. In Switzerland, e.g., national regulations require cab signalling instead of trackside signals for tracks with speed limits above 160 km/h [25]. Other countries have similar regulations while the speed limit may vary from 160 km/h up to 220 km/h [42].

The other solution to increase the throughput is to shorten the sections. Long sections between stations are thereby split into multiple smaller ones. The more sections a track is divided into, the more trains can pass it since they can follow each other in a shorter distance. Therefore, it seems preferable to have as short and as many sections as possible. The disadvantage is that every section must be equipped with a track clear detection device and at least one light signal. If trains can move in both directions on this section two light signals are required.

Trackside equipment is not only expensive to buy and install, its maintenance is expensive as well. Therefore, railway companies try to reduce existing trackside elements rather than installing new equipment. Furthermore, every new section has to be integrated into the interlocking system’s logic, which must be revalidated afterwards.

To increase safety, interlocking systems are expanded with train protection systems that force a train to stop if it passes a red signal. Often, especially
in connection with higher speeds, braking after passing a red signal is too late to prevent an accident. Therefore, different systems have been developed that stop a train before it passes the red signal. Again, these systems require a lot of trackside equipment and in addition, every train has to be equipped with an additional on-board computer. Furthermore, almost every country has its own train protection system and internationally operating trains must be equipped with all the different on-board computers.

### 2.2.4. A New Generation of Train Control Systems

Railway companies are interested in new types of train control systems to overcome the limitations of conventional interlocking systems, to reduce the variety of national train protection systems, and to increase the network’s throughput without installing more trackside equipment.

As a case study, we develop the controller part of such a new system. The system is intended to keep all trains in a railway network a safe distance apart to prevent collisions. The network consists of tracks divided into sections, and points connecting these tracks. We develop the train control system on top of a conventional interlocking systems. The interlocking system switches the points to connect different tracks together, and results in a dynamically changing track layout. Instead of light signals, the train control system uses radio communication to send the trains the permission to move or stop.

![Diagram of train control system](image)

**Figure 2.3.** Train control system with the interlocking system as its environment

A key feature of the system is that it does not need trackside equipment as conventional interlocking systems do. In contrast, our system determines information about the occupation of sections from the trains’ position and length. An overview of the interacting system components is given in Figure 2.3. The trains themselves determine their positions and send them to the train control system by radio. Based on information about the occupied parts of the network, the controller calculates for each train the area in which it can safely move without collisions. This area is called the Movement Authority (MA) and represents the permission for a train to move as long as it does not leave this area. The calculated MAs are then directly sent to the trains where an On-Board Unit (OBU) interprets them to calculate the location where the permission to drive ends, called
the Limit of Authority (LoA). LoAs replace the light signals of conventional interlocking systems located along the tracks. To prevent driving over the LoA, the OBU continuously determines a speed limit and applies the train’s emergency brakes if necessary. This process corresponds to the main functionality of CBTC and ETCS L3.

A conventional interlocking system releases its routes as soon as the corresponding train has passed the tracks. For a following train, a new route must be established. In our control system we want to use the same route for the MAs of following trains. Therefore, we need an adapted interlocking system that lets the routes established until the controller decides that it is not used anymore.

A further advantage of this new generation of train control systems is that the information about occupied areas within the network is more fine-grained than in conventional interlocking systems. As a result, trains can follow each other in a closer distance, which increases the overall throughput of trains through the railway network.

The most important properties of the train control system are the prevention of collisions and derailments. Collision-freeness between trains is guaranteed by the overall system and relies on two conditions: (1) The trains are always within their assigned movement authorities, and (2) the controller ensures that the MAs issued to the trains do not overlap. In fact, (1) is only implementable if the MAs issued by the controller are never reduced at the front of the trains. Non-derailment is ensured by condition (1) mentioned before, and the condition (3) stating that the controller only grants MAs over the active network.

### 2.3. Related Work

Bjørner gives in [13] a comprehensive overview of formal techniques and tools used for developing software for transportation systems. Beside techniques like model checking and model-based test case generation, he mentions approaches using refinement. We discuss the most relevant approaches below.

The development of Metro line 14 in Paris [2, 12] is one of the better-known industrial applications of formal methods. In particular, the safety critical part of the software was developed using the (classical) B Method [1]. The formal reasoning there was only at the software level, i.e., reasoning about the correctness of the software in isolation. In contrast, we model not only the train control system, but also its environment, including the trains and their movement behaviour. Hence, we can reason at the system level covering the overall structure of the system, its components, and their relationship [4].

In [29], Haxthausen and Peleska present the formal development and verification of a distributed railway control system using the RAISE formal method. Their approach is similar to our work as they also use stepwise refinement and ADTs to cope with their system’s complexity. However, their system is overly simplified at some points, which reduced the development challenges that we found to be the
most difficult in our work. First, they only consider simple network topologies without loops. Second, they develop a system where sections are either fully occupied or free. Third, their trains can occupy at most two sections. Although they claim that their system can be easily adapted for trains occupying more than two sections, from our experience, this generalisation is challenging. Moreover, in their proof they require that if any two events are enabled in a valid state, executing one of the events and therefore changing the state cannot disable the other event’s guard. This is a strong property that is cumbersome to verify, as one must prove it for all pairs of events. Our model does not require this property in order to guarantee the system’s safety.

Platzer and Quesel verify parts of a similar train control system in [38] using their own verification tool KeYmaera. While we developed the functionality of the controller, their development focuses on the OBU. In their development, the controller belongs to the environment of the OBU and they assume that the controller does not issue MAs that are physically impossible for the trains. Our development fulfils this assumption by guaranteeing that the MAs are never reduced.
3. Requirements

Together with Hitachi we wrote a requirements document for the train control system based on specifications provided by Hitachi. These specifications describe a simplified version of a real train control system that works similar to CBTC or ETCS L3.

The simplified version does not include the detailed functionality of the on-board unit installed on every train, which is responsible for the communication with the train controller, the measurement of the train’s speed and position, the computation of a breaking curve corresponding to the movement authority granted by the train controller, the correct notification of the train driver about the granted movement authority and the application of the train’s emergency brakes if the train driver does not obey speed limits or the limits of the granted movement authority.

Furthermore, the simplified version abstracts away the implementation of the communication protocol between controller and trains. Although the specification allowed to assume a reliable radio communication, we modelled our system based on communication channels that provide integrity of transmitted messages but have the potential of losing them. The purpose of using a weaker communication model was to prove that our system does prevent train collisions even in the event of a complete communication breakdown where all messages get lost.

The specifications were very much implementation oriented and included details that are unrelated to the desired functionality, like the maximal number of trains that a controller can handle. We compared the provided specification to existing interlocking principles [42] and brought them in line with the specifications of CBTC [32] and ETCS L3 [24] to extract the main functionality of the train control system to develop.

We formulated the following requirements based on the specifications and validated them during several discussions between ETH and Hitachi. These discussions were very important such that both sides got the same understanding of the system’s functionality and the surrounding environment. Furthermore, we agreed on the safety properties that should be guaranteed by the modelled system. This first step of the development process is crucial since developing a formal model is only possible with clear and unambiguous requirements. This important fact has also been pointed out in [3].

Every requirement in the following list has a number and is marked with either ENV or REQ. Requirements of type ENV must be fulfilled by the environment. From the controller’s point of view, these are the assumption we make about the
environment’s behaviour. The requirements of type REQ describe the properties that must be fulfilled by the developed controller.

3.1. Topology

We stated earlier, that the network consists of tracks that are connected by points. For our model we use a more abstract view that uses only sections and represents their connections. It is not important to know to which track a section belongs. Furthermore, points are just connections of sections that dynamically change.

3.1.1. Sections

We use logical sections in our model, which represent the real sections for a particular moving direction. That means, two trains travelling on the same section but in different directions move on two different logical sections. The reasons for using logical sections are explained in Chapter 4.

![Diagram of logical sections]

<table>
<thead>
<tr>
<th>ENV 1</th>
<th>A physical section is the smallest entity in the topology.</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENV 2</td>
<td>A physical section has a defined nominal direction and a fixed length.</td>
</tr>
<tr>
<td>ENV 3</td>
<td>Every physical section is represented by two distinct logical sections, one for the nominal direction and one for the reverse direction. Both logical sections have the same length as the physical section.</td>
</tr>
</tbody>
</table>
The two ends of a logical section are named *beginning (B)* and *end (E)* when facing the logical section’s direction.

### 3.1.2. Network

We define the railway network like a directed graph. The sections are the graph’s nodes and the joints that connect the sections are its edges. A directed edge points from the end (E) of a logical section to the beginning (B) of another logical section. A node can have at most two incoming and one outgoing edge or two outgoing and one incoming edge.

A railway network is a set of logical sections that can be connected with each other.

The beginning of a logical section can be connected with at most two ends. The end of a logical section can be connected with at most two beginnings.

Although the purpose of our train control system is to allow more than one train in a section, there are situations where only one train per section or even group of sections is allowed. We call such an access restricting group of sections a block group. An example where we need a block group is that of points. The two branching sections are too close for simultaneous use. Two trains moving on these sections, one on each branch, could collide although they are located on different sections. An even more obvious application for block groups are crossings where two actually independent sections do overlap.

In train stations, the specifications require that a train can move backwards within the platform area. In general, we do not allow trains to move backward, as the space behind a train could already be used for the MA of another train. We solved this issue by providing the possibility to define sections in the platform area as reverse sections. By our definition, a reverse section must belong to a block group. As a result, the space from the beginning of this section to the train’s rear remains free. Hence, it is safe for the train to move backwards.

One or more logical sections can be defined as a block group.
Only one train is allowed to be in the entire set of logical sections of a block group at a time.

A block group containing only one logical section can be defined as a reverse section.

Within a reverse section the train is allowed to move backwards.

Neighbouring logical sections where two trains could collide, although they are in different sections, build together a block group. (e.g. around points or a crossing)

### 3.1.3. Position

The trains and the controller use positions to exchange information about their location and computed LoAs, respectively. For the safety of the system, it is indispensable that both side use the same notion. To prevent ambiguities, we require a notion in which every position is uniquely defined.

The beginning of every logical section is a unique reference point for positions.

A position is defined as the distance to a reference point. The representation of a position includes the used reference point and the distance to it.

A position is always described using the closest reference point in backward direction.

### 3.2. Operation

In our model we still use a conventional interlocking system to set the points and establish routes. It belongs to the controller’s environment. Again, we abstract
away from the specifications and use the term active network instead of routes. The active network is that part of the network that is secured by the interlocking system.

Using again the comparison to a graph, the active network is a subgraph of the railway network. Every node has at most one incoming and one outgoing edge. Hence, the set of edges in the active network corresponds to an injective function from nodes to nodes. Since the graph’s nodes represent the logical sections, a physical section is represented by two nodes. To prevent head-on collisions, only one of this two nodes can be in the active network at a time.

### 3.2.1. Interlocking System

![Diagram showing active and unset areas of the network.]

| ENV 15 | The interlocking system declares on which part of the network trains are allowed to move. The corresponding subset of the network is called active network. |
| ENV 16 | The active network can include the nominal or the reverse logical section of a physical section, but not both. |
| ENV 17 | In the active network, the beginning of a logical section is connected with at most one end of another section. The end of a logical section is connected with at most one beginning of another section. A logical section can be connected with itself. |

### 3.2.2. Trains

In addition to its head and rear, we also define the train’s orientation that corresponds to the direction when moving forward. This is useful to state the property that all trains must move in the same direction within a section and that a train can only move along the logical section’s direction.
ENV 18  Trains are moving on the active network.

ENV 19  The two ends of a train are called **head** and **rear**. The train’s orientation is from rear to head.

ENV 20  The orientation of a train is aligned with the direction of the logical sections on which the train moves.

To enable an unambiguous communication between trains and controller we define that a train’s position corresponds to its head position. This is reasonable since the train’s on-board unit, which determines the position, is located in the train’s head.

The MA of a train is limited by two positions: one behind and one in front of the train. Since a train moves only forward (except in reverse sections), only the position ahead is relevant for the train. This position is called Limit of Authority (LoA).

ENV 21  A train’s position is defined as the position of its head.

REQ 22  Every train has a MA assigned that defines the area in which a train can freely move without colliding with other trains. The front end of this MA is called LoA.

### 3.2.3. CBTC Equipment

Beside determining the position of the train, the on-board unit is also responsible for the communication to the controller. A radio channel is used to send updates on the train’s position and to receive updated LoAs.
On every train there is an OBU that communicates with the TLC over a reliable channel.

The OBU repeatedly determines the train’s position and sends it together with the train’s length to the TLC.

Every OBU repeatedly receives the LoA from the TLC.

The OBU enforces that the train can only move in the allowed direction, which is forward in general, and that it cannot move further than its assigned LoA. According to the provided specifications, the OBU knows the topology of the active network and allows the train to move backwards if its rear is in a reverse section. When moving backwards, the OBU enforces that the train’s rear cannot move further than the beginning of the reverse section.

A train can move forward (with the train’s direction) up to the LoA.

If a train’s rear is in a reverse section, the train can move backwards (against the train’s direction) up to the beginning of this reverse section.

The TLC and the OBU know the topology of the active network.

The OBU is responsible that the train does not move over the LoA granted by the TLC.

The OBU is responsible that the train does not move over the beginning of a reverse section when moving backward.
In emergency situations the train controller system can send a special message to the train’s OBU, which immediately applies the train’s emergency brakes.

<table>
<thead>
<tr>
<th>ENV 31</th>
<th>The TLC can send an emergency stop message to any OBU. If an OBU receives an emergency stop message the train’s emergency brakes are immediately applied.</th>
</tr>
</thead>
</table>

### 3.2.4. Zone Protection Range

In Japan, station platforms are equipped with emergency buttons. If such a button is pressed, the trains in the corresponding area are automatically stopped. We call the set of affected sections that belong to such an area a Zone Protection Range (ZPR).

<table>
<thead>
<tr>
<th>ENV 32</th>
<th>A Zone Protection Range (ZPR) is a predefined area in the network that can be closed for train movements in case of an emergency.</th>
</tr>
</thead>
</table>

| REQ 33 | If a train’s MA overlaps with an activated ZPR, the TLC sends an emergency stop message to the train’s OBU |

### 3.2.5. Computation of the LoA

The main task of the controller is to compute the MAs for trains in a safe way and to communicate the resulting LoAs to the corresponding trains. This computation is based on the position information of the trains that the controller receives.

The specifications define the notion of pathways, which are similar to routes in conventional interlocking systems. The controller sets for every train a pathway that acts as an intermediate step for computing a train’s MA. A train’s pathway marks the sections which can be added to its MA. To prevent derailments of trains, only sections of the active network can be marked by a pathway. The sections marked by a pathways must belong to the active network to prevent derailment of trains.

| ENV 34 | The TLC receives the position of every train in the active network sent by the train’s OBUs. |
The TLC sets the current pathway for every train in the active network.

The controller does not extend the MAs of trains that are located in activated ZPRs for safety reasons. These trains received an emergency stop message and updating their MAs could be misleading for the train drivers.

The TLC computes the LoA for every train in the active network whose MA does not overlap with an active ZPR and sends the LoA to the corresponding OBU.

We assumed that a train’s OBU enforces that the train cannot move further than the assigned LoA. This assumption is only valid if the OBU does not receive updated LoAs that are closer than the former ones. Hence, the controller must not reduce a train’s MAs at the front end.

The controller is responsible that only one train can access the sections of a block group at a time. Hence, only one MA can use the sections in a block group. The controller must prevent collisions between trains by keeping their MAs apart.

The purpose of ZPRs is to prevent that trains enter the corresponding sections in case of an emergency. Therefore, the controller must not extend MAs into sections that belong to an activated ZPR.

The TLC only extends the LoA for a train in its direction, never reduces it.

The TLC only extends the LoA into a section of a block group if there is no other train in a section of this block group.

The TLC never extends the LoA into the MA of another train.

The TLC never extends the LoA into an activated ZPR.
3.3. Safety

The following two requirements are the main safety properties that the train control system must fulfil. These requirements are fulfilled if all the former requirements are fulfilled. In fact, we prove the following two requirements in the Event-B model as theorems.

| REQ 41 | There must be no collision between any two (different) trains in the system. |
| REQ 42 | Every train in the system must stay on the tracks of the network. |

3.4. Summary

We defined a set of requirements that covers the main functionality of our train control system. We defined 42 requirements of which 30 are assumptions on the environment and 12 are requirements that the controller must fulfill.
4. Design of the Train Control System

In this chapter we describe some design decisions we made during modelling the train control system in Event-B. This chapter provides useful information for future developments in the railway domain using Event-B.

In Section 4.1 we discuss the reasons for introducing logical sections in our model. It appears counterintuitive why the representation of one physical section by two logical sections should reduce the number of events and proofs. We show that using logical sections simplifies the description of how trains move around in the network.

The representation of a train within the network is simple in conventional interlocking systems. A section is either free or fully occupied by a train, even if the train only occupies part of the section. We show in Section 4.2 that a train’s representation becomes more difficult if we use exact positioning within the sections. We need additional information about the train’s layout to prevent ambiguity.

In Section 4.3 we explain why the actions of the controller and the environment that are involved when a train enters or leaves the network must be executed in an atomic way. At the abstract level, where we try to keep the model as simple as possible, the easiest way to guarantee the atomic execution of actions is to put them all into one event. We can split this event in a later refinement step and restore the broken atomicity by modelling a handshaking protocol. This splitting step, however, should not be postponed for too long, as additional actions complicate the modelling of the handshaking protocol.

A highlight of our development is that the system fulfils all safety properties without requiring reliable communication channels between controller and trains. Although the specifications declared that we can assume a reliable radio connection, our developed system is immune to message losses. Section 4.4 describes how our system remains safe even in case of a complete communication blackout between trains and controller.

No train control system can prevent collisions between trains and other objects which are not under its control. Car drivers who ignore red lights at level crossings are a classic example. In terms of safety, the best a train control system can do, if it becomes aware of an uncontrolled object on the tracks, is to revoke all affected Movement Authorities. We justify in Section 4.5 why we think that this emergency procedure, although being important, should not be part of the controller’s core functionality.
In Section 4.6 we discuss some limitations of our current model and propose solutions that we consider as future work. We also explain why our train control system is not widely applicable in existing railway networks for the time being.

4.1. Logical Sections

We did not use logical sections in our model right from the beginning. It was an idea that came up during modelling to simplify the events and reduce the number of events that describe the trains’ movements. Let us assume we have a track that is divided into three sections $s_1$-$s_3$. To identify the two ends of every section we define for each section a beginning ($B$) and an end ($E$), as depicted in Figure 4.1. Let us further assume that a train moves from section $s_1$ to section $s_2$. As long as we were modelling at the abstract level without exact positions it was enough to mention the sections (ignoring $B$ and $E$) to describe the trains’ movements.

![Figure 4.1.: Orientation of sections within a track](image)

When we introduce exact positions within sections we have to model our train movement more precisely. For the same train as before, we must say that it leaves $s_1$ at $E$ and enters $s_2$ at $B$. For a real train it is not possible to leave $s_1$ at $E$ and enter $s_2$ at $E$. In general, we need the additional information about the orientation of the neighbouring sections to restrict the movements of trains to the physically possible.

We can assume that in a track all sections have the same orientation and, therefore, all neighbouring sections are connected such that the $B$ end of one section meets the $E$ end of another one. In our model, two tracks are always connected by points. In this case, it can happen that two $B$ ends or two $E$ ends are connected together. There exist track layouts where we cannot avoid such connections. Such an example is depicted in Figure 4.2.

![Figure 4.2.: Situation where we cannot avoid connecting two $E$ ends](image)

To describe the exact position of a train within a section we introduce the following metric. We defined that the $B$ end should have position 0 and the
position is increasing towards the $E$ end, which has the position equal to the section’s length. Depending on the moving direction of a train, moving forward means to move to a greater position within the section when it moves from $B$ to $E$ or to a smaller position when it moves from $E$ to $B$. As a result we have two different events for every moving action.

It gets especially cumbersome if the train moves over points and the two tracks have a different orientation. Let us assume that a train moves in Figure 4.2 from section $s_1$ to the loop section $s_3$ and enters this section on the left side. After entering $s_3$ its head position is decreasing while the rear position is increasing when moving forward. As a result we need additional events that account for different orientations of the head and rear section.

To simplify the movement events we introduce logical sections. A section on which a train can move in both directions is represented by two logical sections, one for each direction. If the interlocking system establishes a route, the logical section for the corresponding direction is activated. We require that the two logical sections for both directions of a physical section must not be in the active network at the same time (ENV 16). This property can easily be checked when the layout of the active network is changed.

The interlocking system activates logical sections according to routes. Hence, in the active network two logical sections are always connected such that a $B$ end meets an $E$ end. Furthermore, the trains are only allowed to move in the direction of the logical sections. These two properties are actually very useful as they already prevent head-on collisions.

Since the logical sections are all pointing in the same direction and we require that the trains move with the logical section’s direction, the head and rear positions do always increase when moving forward. As a result, we need fewer events to model the train’s movement. Note that the number of proof obligations that must be discharged during modelling is proportional to the number of events.

A further simplification of the movement events comes from using logical sections. When a train reaches the end of its current head section, it will move to the next section in the network. In contrast to the physical section, the logical section is directed like the former head section. Thus its new head position will always become 0, regardless of the alignment of the physical section. This simplifies the moving events and reduces its number too.

4.2. Layout of a Train

We use in our model the variable connection to describe the layout of trains. The set of occupied sections alone provides not enough information to clearly identify the layout of a train. We need the additional information about the sequence in which the train occupies these sections.

Let us assume we have a simple track that is connected to itself to build a circular network. The track consists of only one section $s$. A train is located in
this network that, consequently, occupies section $s$. Its head and rear are both in
section $s$ too. Let us further assume that the length of section $s$ is 100 and that
the train’s head position is 75 and its rear position 25.

Without the variable $\text{connection}$, we cannot tell whether this train is moving
clockwise or counterclockwise. The reason is, that the above information describes
two different train layouts as depicted in Figure 4.3. The missing information to
distinguish the two layouts is whether the train stretches across the section’s
border. In this case, the train would actually occupy section $s$ twice. Hence,
for the left layout we have $\text{connection} = \{s \leftrightarrow s\}$ while for the right layout $\text{connection} = \emptyset$ holds.

![Figure 4.3.: Two different train layouts](image)

The variable $\text{connection}$ is also useful to specify that a train must be in one
piece. Having only the information about the occupied sections, it would be pos-
sible that the train occupies different parts of the network that are not connected.

### 4.3. Entering and Leaving Trains

If a train enters or leaves the network controlled by a Trackside Logic Controller
(TLC) it must be activated or deactivated in the controller. At the abstract level,
we model the activation and deactivation of a train such that the sections that
it occupies are simultaneously activated or deactivated in the network. This is
important to guarantee the safety property that a train is always within the active
network and to prevent collisions between trains.

If we activate a section before activating the train that enters, the controller
(unaware of the entering train) could use the activated section for another train’s
Movement Authority (MA) and cause a collision. If we activate the entering train
before the corresponding sections, the train is not within the active network,
which the model considers as a derailment. The non-derailment property of the
model would become unprovable. The same pseudo-derailment would happen if
we deactivate sections before deactivating the corresponding train that occupies
them. If we deactivate the train before the corresponding sections are deactivated, there is again the risk that the section is used for another MA.

Activating or deactivating sections in the same event as activating or deactivating the occupying trains at the abstract level simplifies the modelling as well as the proofs. When we want to generate code or simulate our model, however, we have to separate these events. While activating and deactivating the sections belongs to the environment, activating and deactivating the trains is part of the controller.

The decomposition of the events for entering and leaving trains towards the end of the refinement chain was very cumbersome. We had to implement a handshaking protocol that simulates the simultaneous activation of sections and trains. It is questionable whether the simpler proofs at the beginning of the modelling justify the additional effort at the end.

4.4. Radio Communication

We prove in our model of the train controller, that it is possible to guarantee strong safety properties with an unreliable radio connection. The radio connection not being reliable affects only liveness, not the safety of our system.

Our proof that no two trains ever collide holds under the following assumptions. The On-Board Unit (OBU) is always capable to stop at the Limit of Authority (LoA). This assumption is only valid if we do not decrease the LoA but only increase it. This means that an updated LoA is always further away from the train than before. The minute a train receives an updated LoA, its braking curve is still computed such that it can stop at the former LoA that was closer to the train’s head. Therefore, it is no problem to also stop at the new LoA. If the train does not receive any update of the LoA, it will come to a stop at the old LoA although the track ahead is clear and locked by the system. This does not result in a safety issue; only liveness is affected.

In the other direction, the train sends the train controller position updates. When the controller receives an updated position of a train, its local information about track occupation is updated. This frees tracks behind the train that can be used for other trains’ MAs. If such a position update message gets lost, the tracks behind the moving train are not freed. Again, this results only in liveness issues but does not affect safety.

We see that the loss of messages always results in liveness issues but never in safety issues. The necessary assumption for this is that the MA is only increased at the train’s head and decreased at its rear.

4.5. Emergency Situations

Trains continuously move on the network within their assigned MAs. As long as trains are the only objects that occupy the tracks, our model is valid and we
can guarantee the absence of collisions. It is not possible that a train suddenly appears in front of another train. In real life, however, it is possible that the tracks ahead of a train must unexpectedly be closed for train operation. Reasons are, e.g., an accident where a person on the station platform falls on the tracks or a car is stuck on a level crossing.

It might be the case that this part of the track was already assigned to the MA of an arriving train. It must be possible to immediately stop the train in such situations by cancelling its MA and sending an emergency break message. We earlier assumed that the MA of a train is never decreased. This assumption now turns out to be too restrictive.

However, when we allow the MAs to be reduced we cannot guarantee anymore the non-collision property. Therefore, we decided to split the operation of the train control system into different modes. In normal mode our assumption about not decreasing the MA holds. Therefore, we can guarantee the absence of collisions between trains. In an emergency situation, when we have to cancel part of an MA we switch to emergency mode. In this mode we assume that MAs can be reduced. The activation of the emergency mode should be triggered by the zone protection ranges.

When we reduce a train’s MA, the OBU cannot guarantee that the train stops before the reduced LoA. Hence, we do not know whether it enters the track that has been cancelled due to an emergency. As a consequence, the cancelled track must remain unavailable until the emergency has been handled. In our model we do not deal with the emergency mode, as we cannot prove the non-collision property anyway. We model only the normal mode, which is the desired behaviour of the system.

4.6. Future Work

4.6.1. Ghost Trains

When a train is deactivated the corresponding sections that it occupies are deactivated as well. This prevents that the controller can use these still occupied sections for the MAs of other trains. However, our model does not prevent that after deactivation, the same sections are activated again without activation of the corresponding train. The train becomes a ghost train because it occupies part of the active network without the controller’s knowledge. This is a dangerous situation.

Conventional interlocking systems do not have this problem. They prevent the establishment of ordinary routes if the corresponding sections are not free based on real-time track information. The adapted interlocking system that we use in our train control system, however, is not equipped with track clear detection devices. It gets the information about occupied sections from the controller, which determines this information based on the positions and lengths of the active trains.
Hence, neither the controller nor the underlying interlocking system can detect ghost trains until they are activated again.

To prevent collisions, we need a countermeasure that prevents ghost trains. A solution could be to explicitly define special sections where trains can enter and leave the network and equip these sections with traditional track clear detection devices. Sections where a train can change its direction, e.g. in a station, must also be defined as special sections. Note that we model a train that changes its direction as a train that is deactivated and activated again with its reversed layout.

4.6.2. Speed Profile

In our system we did not consider different speed limitations on the tracks. For a safe operation, the train controller must also transmit a speed profile together with the MA. A speed profile is a diagram that shows the permitted speed in relation to the way ahead. This is especially important when a train moves across points in the diverging direction. Driving too fast on points causes derailment. The speed profile can also include temporary speed limitation due to construction sites. In our model we did not include this information since it is not used to determine the length of the MAs. It is additional information that the controller can look up in a static table.

4.6.3. Train Integrity

The update of the train position is sent by the OBU located at the train’s head. The relevant part of the train for releasing the track behind, however, is the train’s rear. This means that we assume that the train always checks its integrity, meaning that it has not been separated. For passenger trains, this is in general no problem since they are usually equipped with a redundant communication bus. Train separation would be noticed by the OBU immediately.

For freight trains, however, this is a non-trivial issue. In Europe, freight wagons are not equipped with any communication bus and the owners of the wagons have no intension to equip them any soon. Also ideas of putting a device at the rear of a train that communicates with the train’s head are not reasonable. Train operators refuse to be responsible to install these devices. Note that after any rearrangement of the freight train, the device has to be put at the rear of the last wagon.

For the time being, checking the integrity of freight trains is an open-end question. This is also the reason, why European Train Control System Level 3 (ETCS L3) will not be widely introduced any soon. ETCS L3 would be possible to install on isolated networks that are only used by passenger trains. Metros are typical closed systems with an isolated network and with passenger traffic only. Hence, a train control system like Communication Based Train Control (CBTC) is applicable.
5. Abstract Data Types in Event-B

5.1. Introduction

When developing large, complex systems, refinement alone is often insufficient. Machines containing sufficient details to state and prove relevant safety properties may lead to proofs of unmanageable complexity. We observed this limitation while developing the train control system by refinement in Event-B. To specify and reason about the system’s collision-freeness and non-derailment properties, we needed to provide detailed models, for example formalising the network on which the trains operate and the trains’ position and movement. As a consequence, we had to state numerous complex invariants, which resulted in many complicated manual proofs. This motivated an alternative approach to abstract away additional details from the system model to reduce the complexity and increase the automation of the resulting proofs.

Approach  To increase system abstraction and reuse during modelling, we introduce Abstract Data Types (ADTs) [34] in Event-B. An ADT is a mathematical model of a class of data structures. It is typically defined by a set of operations that can be performed on the ADT, along with a specification of their effect. By using Event-B contexts to formalise ADTs, we can subsequently utilise the ADTs to model the system’s dynamic behaviour in the machines. We use generic instantiation [6] to further concretise and thereby implement the ADTs. As the ADTs evolve, the machines are also refined accordingly.

We evaluate our approach by using it in our case study and measuring how the use of ADTs reduces the size and increases the automation of the development.

Contribution  Our contribution is to introduce ADTs into a refinement formalism and show how they can be used to manage development complexity. In this thesis, we use Event-B [3] to illustrate our approach, but other refinement formalisms such as ASMs [14], Action Systems [7], or TLA [33] could be used instead. We show that reasoning using ADTs can be done purely based on the properties of the ADTs’ operations, regardless of how they are implemented. As a result, systems specified with ADTs are more abstract and hence easier to verify than systems developed directly without them. In fact, ADTs encapsulate part of the system’s dynamic behaviour in the static context of Event-B. This is novel as traditionally Event-B contexts are only used to specify static parameters of a system’s model and all dynamic behaviour is modelled as a transition system in the Event-B machines. Furthermore, our use of generic instantiation is novel as
this technique has so far only been applied to reuse developments, for example in [41]. In contrast, we use generic instantiation to gradually introduce details into the formal models similar to refinement.

The way we use ADTs allows them to be used alongside refinement. Hence, one can combine these two different abstraction techniques during development and apply whichever fits better at a particular development stage and results in simpler proofs. In contrast to development strategies that use refinement or ADTs exclusively, our approach is more flexible and thereby appears better suited for developing large-scale industrial systems.

Structure The rest of this chapter is structured as follows. We motivate and present our approach in Section 5.2. We evaluate it on our industrial case study in Section 5.3. Finally, we discuss related work in Section 5.4 and conclude in Section 5.5.

5.2. Abstract Data Types in Event-B

We now describe how to specify and implement ADTs in Event-B. Our approach is based on refinement and generic instantiation. An ADT is typically defined in terms of a set of operations that can be performed on the ADT, along with a specification of the operations’ effect. Let us start with the standard example: a stack is a last-in-first-out data type that contains a collection of elements and is characterised by three operations:

- **push**: takes a stack \( S \) and an item \( e \) and returns a new stack, where \( e \) is added to the top of \( S \).
- **pop**: takes a (non-empty) stack \( S \) and returns a new stack, where \( S \)’s top element is removed.
- **top**: takes a (non-empty) stack \( S \) and returns \( S \)’s top element.

A special stack is the *empty* stack that contains no elements. Three important constraints on the stack operations are: Given a stack \( S \) and an element \( e \), \( \text{push}(S, e) \neq \text{empty} \), \( \text{pop}(	ext{push}(S, e)) = S \), and \( \text{top}(	ext{push}(S, e)) = e \).

ADTs and their operations can be modelled using carrier sets, constants, and axioms in Event-B. Instantiation can be used to “implement” an ADT using other ADTs. The instantiation proofs ensure that the implementations satisfy their specification.

5.2.1. Specifying ADTs in Event-B

Each ADT \( A \) is modelled as follows:

- A carrier set \( A\_TYPE \) defines the type of the objects in \( A \).
• An associated set constant \( A \subseteq \text{A_TYPE} \) represents all valid objects in \( A \).\(^1\)

• Each operation is modelled as a constant.

• The constraints on the operations are specified using axioms.

Consider the ADT \textbf{Stack} for elements of type \textit{ELEM}. It can be modelled in Event-B as follows.

\begin{verbatim}
sets : STACK_TYPE

constants : STACK, empty, push, pop, top

axioms :
  axm0_1 : STACK \subseteq STACK_TYPE
  axm0_2 : empty \in STACK
  axm0_3 : push \in STACK \times ELEM \rightarrow STACK
  axm0_4 : pop \in STACK \setminus \{empty\} \rightarrow STACK
  axm0_5 : top \in STACK \setminus \{empty\} \rightarrow ELEM
  axm0_6 : \forall S, e \cdot S \in STACK \land e \in ELEM \Rightarrow \text{push}(S \mapsto e) \neq empty
  axm0_7 : \forall S, e \cdot S \in STACK \land e \in ELEM \Rightarrow \text{pop}(\text{push}(S \mapsto e)) = S
  axm0_8 : \forall S, e \cdot S \in STACK \land e \in ELEM \Rightarrow \text{top}((\text{push}(S \mapsto e))) = e
\end{verbatim}

In the above, the notation \( S \times T \) corresponds to the \textit{Cartesian product} of \( S \) and \( T \), \( f \in S \rightarrow T \) specifies that \( f \) is a \textit{total function} from \( S \) to \( T \), and \( a \mapsto b \) denotes the \textit{ordered pair} \((a, b)\). The axioms \texttt{axm0\_6–axm0\_8} specify the relationship between the \textit{pop}, \textit{top}, and \textit{push} operations and the constant stack \textit{empty}. Note that there is no need to fully specify an ADT. In subsequent examples, we will only define as many axioms as are needed to prove the stated properties.

\subsection*{5.2.2. Implementing ADTs by Instantiation}

A possible implementation of \textbf{Stack} is one where a stack is represented as an array. Specifically, a stack is represented by a pair \((f, n)\), where \( n \) is the stack’s size and \( f \) is an array of size \( n \) representing its content. Similar to ADTs, we use operations to describe the concrete data type \textbf{Array}.

• \textit{append}: takes an array \( A \) and an element \( e \) and returns a new array where \( e \) is appended to the end of \( A \).

• \textit{front}: takes an array \( A \) and returns a new array where \( A \)’s last element is removed.

\(^1\)Note that we do not currently support the definition of parametrised ADTs, which would allow one to specify a generic ADT for stack independent of its elements’ type.
• last: takes an array \( A \) and returns \( A \)'s last element.

The concrete data type \textbf{Array} is specified in Event-B as follows.

\begin{center}
\begin{tabular}{|l|}
\hline
\textbf{constants} : ARRAY, append, front, last \\
\hline
\end{tabular}
\end{center}

\begin{center}
\begin{tabular}{|l|}
\hline
\textbf{axioms :}
\hline
\end{tabular}
\end{center}

\begin{center}
\begin{tabular}{|l|}
\hline
axm1\_1 : ARRAY = \{ f \mapsto n \mid n \in \mathbb{N} \land f \in 0..n-1 \mapsto ELEM \} \\
axm1\_2 : append = (\lambda (f \mapsto n) \mapsto e \cdot f \mapsto n \in ARRAY \land e \in ELEM \\
| \ (f \&= \{n \mapsto e\}) \mapsto n + 1) \\
axm1\_3 : front = (\lambda f \mapsto n \cdot f \mapsto n \in ARRAY \land n \neq 0 \\
| \ (\{(n - 1) \&= f\} \mapsto n - 1)) \\
axm1\_4 : last = (\lambda f \mapsto n \cdot f \mapsto n \in ARRAY \land n \neq 0 \mid f(n-1)) \\
\hline
\end{tabular}
\end{center}

In the above, \( f \&= \{x \mapsto E\} \) denotes the function identical to \( f \) except for the entry at \( x \) that is assigned \( E \), and \( \{x\} \&= f \) denotes the function where the entry for \( x \) is removed from \( f \). Note that all the constants are concretely defined using lambda abstractions. The term \((\lambda X \cdot P \mid E)\) denotes a lambda abstraction where the pattern expression \( X \) specifies the domain of the function. More specifically, \((\lambda X \cdot P \mid E)\) is defined to be the set comprehension \( \{x \cdot P \mid X \mapsto E\} \), where \( x \) is the list of variables that appears in \( X \).

To prove that \textbf{Array} implements \textbf{Stack}, we instantiate \textbf{Stack} as follows.

\begin{center}
\begin{align*}
STACK\_TYPE & \leftarrow \mathbb{P}(\mathbb{Z} \times ELEM) \times \mathbb{Z} \\
STACK & \leftarrow ARRAY \\
push & \leftarrow append \\
pop & \leftarrow front \\
top & \leftarrow last \\
empty & \leftarrow \emptyset \mapsto 0 \\
\end{align*}
\end{center}

We use the generic instantiation tool [26] to generate a new Event-B model where the \textbf{Stack} context is replaced by the \textbf{Array} context. The process of instantiation works as follows. We start at the developed model that uses \textbf{Stack} with carrier sets \( s \) and constants \( c \), depicted on the left in Figure 5.1. In the tool, we define the substitution rules for the carrier sets and constants according to the above table. We use the terms \( s \leftarrow T(t) \) and \( c \leftarrow E(t, d) \) to describe the substitution rules for carrier sets \( s \) and constants \( c \), respectively. The terms \( T(t) \) and \( E(t, d) \) represent a set of type expressions and a set of general expression, respectively, using \textbf{Array}'s carrier sets \( t \) and constants \( d \).

The tool then generates a copy of the refinement chain \( M_1, \ldots, M_x \) where all appearances of \textbf{Stack}'s carrier sets and constants are replaced according to the substitution rules. Hence, \( M_i(s, c) \) becomes \( M_i(T(t), E(t, d)) \) on the right side in Figure 5.1. The generated machines do not see contexts \( C_1, \ldots, C_n \) anymore.
but see the new context Inst, which is generated by the tool as an extension of Array represented by $D_1, \ldots, D_m$.

![Diagram](image)

Figure 5.1.: Generic instantiation in Event-B

We must prove that the instantiated abstract axioms $\text{axm0}_1 - \text{axm0}_8$ are derivable from the concrete axioms $\text{axm1}_1 - \text{axm1}_4$. The following theorems $\text{thm0}_1 - \text{thm0}_8$ are the instantiated version of the corresponding abstract axioms, where the carrier sets and constants are syntactically replaced according to the specified instantiation.

```
thm0_1 : ARRAY \subseteq \mathcal{P}(\mathbb{Z} \times \text{ELEM}) \times \mathbb{Z}
thm0_2 : \emptyset \mapsto 0 \in \text{ARRAY}
thm0_3 : \text{append} \in \text{ARRAY} \times \text{ELEM} \rightarrow \text{ARRAY}
thm0_4 : \text{front} \in \text{ARRAY} \setminus \{\emptyset \mapsto 0\} \rightarrow \text{ARRAY}
thm0_5 : \text{last} \in \text{ARRAY} \setminus \{\emptyset \mapsto 0\} \rightarrow \text{ELEM}
thm0_6 : \forall S, e : S \in \text{ARRAY} \land e \in \text{ELEM} \Rightarrow \text{append}(S \mapsto e) \neq \emptyset \mapsto 0
thm0_7 : \forall S, e : S \in \text{ARRAY} \land e \in \text{ELEM} \Rightarrow \text{front(append}(S \mapsto e)) = S
thm0_8 : \forall S, e : S \in \text{ARRAY} \land e \in \text{ELEM} \Rightarrow \text{last(append}(S \mapsto e)) = e
```

The tool automatically generates these theorems within the new context Inst. If all these theorems can be proved, then the axioms of Array are stronger than the axioms of Stack. Hence, by using generic instantiation we reduce the number of possible values for the carrier sets and constants in the context seen by the machines. We can say, the context Array represents a subset of context Stack. Since we proved refinement using Stack, the refinement proofs do also hold for the subset Array. This result has been shown in [6].

The proofs that $\text{thm0}_1 - \text{thm0}_8$ are derivable from $\text{axm1}_1 - \text{axm1}_4$ can be constructed by expanding the definitions of the concrete constants, i.e., ARRAY, append, front, and last, accordingly. For instance, the proof of $\text{thm0}_2$ gives rise to the following sub-goals:

\[ 0 \in \mathbb{N}, \text{ and} \]
\[ \emptyset \in 0 \ldots -1 \rightarrow \text{ELEM} , \]
which are trivial to prove. The proof of \texttt{thm0_6} eventually requires proving that \( n + 1 \neq 0 \) for any natural number \( n \), which is again trivial. The Rodin platform archive of the development can be found online.\(^2\)

5.3. Developing the Train Control System Using ADTs

In this section, we illustrate our approach on the case study. We first explain the difficulties in developing such a complex system without ADTs. We then present part of the development where we used ADTs. Finally, we evaluate our approach by giving an overview of the entire development together with statistics that document our approach’s effectiveness.

5.3.1. The Need for Abstraction

Our first challenge in developing the train control system is formalising the trains in the network. Figure 5.2 depicts a train occupying a part of the network. It illustrates a sequence of sections with fully occupied sections in the middle and partially occupied sections at each end of the train.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{train.png}
\caption{A train occupying a sequence of sections}
\end{figure}

In our first attempt at modelling this system, we used different variables to denote how trains occupy the network. Let \( ids \) be the set of active trains in the network. We modelled the different aspects of the trains, such as their head, rear, middle, connections, etc., by total functions as follows. For clarity, we omit from the presentation other aspects of the trains, such as the head- and rear-position within a section.

\begin{center}
\textbf{variables: } \texttt{ids, head, rear, middle, connection,…}
\end{center}

\(^2\)URL: http://sourceforge.net/projects/gen-inst/files/Examples/
invariants:
inv0_1: head ∈ ids → SECTION
inv0_2: rear ∈ ids → SECTION
inv0_3: middle ∈ ids → ℙ(SECTION)
inv0_4: connection ∈ ids → (SECTION → SECTION)
inv0_5: ∀t.t ∈ ids ⇒ head(t) ∈ middle(t)
inv0_6: ∀t.t ∈ ids ⇒ rear(t) ∈ middle(t)
inv0_7: ∀t.t ∈ ids ∧ connection(t) = ∅ ⇒ head(t) = rear(t)
inv0_8: ∀t,s.t ∈ ids ∧ s ∈ ran(connection(t)) ⇒
head(t) := s ∈ cl(connection(t))
inv0_9: ∀t,s.t ∈ ids ∧ s ∈ dom(connection(t)) ⇒
s := (head(t) := s ∈ cl(connection(t)))

In the invariants above, \( f : S \mapsto T \) specifies that \( f \) is a partial injective function from \( S \) to \( T \), and \( cl \) denotes the irreflexive transitive closure as defined in [3]. The invariants \( inv0_5 \)–\( inv0_9 \) specify several properties of trains. For example, \( inv0_7 \) specifies that if a train occupies only a single section, its head and rear be in the same section. Furthermore, \( inv0_8 \) and \( inv0_9 \) state that every train is connected from its head and its rear.

To motivate the need for additional abstraction during modelling, we focus on the events \( \text{train\_extend} \) and \( \text{train\_reduce} \).

train\_extend:
any t, s where
\( t \in \text{ids} \)
\( s \notin \text{dom}(\text{connection}(t)) \)
\( \text{head}(t) \notin \text{ran}(\text{connection}(t)) \)
then
\( \text{head}(t) := s \)
\( \text{middle}(t) := (\text{middle}(t) \cup \{\text{head}(t)\}) \setminus \{\text{rear}(t)\} \)
\( \text{connection}(t) := \text{connection}(t) \cup \{s \mapsto \text{head}(t)\} \)
end

train\_reduce:
any t where
\( t \in \text{ids} \)
\( \text{rear}(t) \in \text{ran}(\text{connection}(t)) \)
then
\( \text{rear}(t) := \text{connection}(t)^{-1}(\text{rear}(t)) \)
\( \text{middle}(t) := \text{middle}(t) \setminus \{\text{connection}(t)^{-1}(\text{rear}(t))\} \)
\( \text{connection}(t) := \text{connection}(t) \setminus \{\text{connection}(t)^{-1}(\text{rear}(t)) \mapsto \text{rear}(t)\} \)
end

In the above, \( \text{dom}(r) \) and \( \text{ran}(r) \) represent the domain and range of a relation \( r \), respectively, and \( r^{-1} \) denotes the inverse relation of \( r \).

The event \( \text{train\_extend} \) extends a train, denoted by \( t \), to a section, denoted by \( s \). Namely, \( \text{train\_extend} \) prepends \( s \) to the train’s head and \( s \) becomes the new
head. This event is used whenever the train reaches the end of the current head section and moves to the beginning of the next section in front of it. The event’s guard ensures that $t$’s connection remains a partial injective function ($\text{inv0}_4$). When updating $\text{middle}(t)$, we remove $\text{rear}(t)$ to guarantee that when the train occupies only one section ($i.e., \text{connection}(t) = \emptyset$, and hence $\text{head}(t) = \text{rear}(t)$ by $\text{inv0}_7$), the train’s middle is still empty afterwards.

The event $\text{train\_reduce}$ reduces a train $t$ by removing its rear. The event’s guard guarantees that the train spans at least two sections. Its action updates $t$’s rear, middle, and connection accordingly.

Proving that the events $\text{train\_extend}$ and $\text{train\_reduce}$ maintain the invariants, $e.g., \text{inv0}_5$ and $\text{inv0}_6$, requires additional invariants that relate the different aspects of trains, $i.e.,$ head, rear, middle, and connection. Two examples of this are the following invariants, $\text{inv0}_10$ and $\text{inv0}_11$.

\[
\begin{align*}
\text{inv0}_10 : \forall t. \text{middle}(t) = \text{dom} (\text{connection}(t)) \setminus \{\text{head}(t)\} \\
\text{inv0}_11 : \forall t. \text{middle}(t) = \text{ran} (\text{connection}(t)) \setminus \{\text{rear}(t)\}
\end{align*}
\]

In the following, we discuss several problems with the current representation of trains in the model and indicate how further modelling abstractions could help to overcome these difficulties.

**Encapsulation** The invariants $\text{inv0}_5$–$\text{inv0}_11$ describe that the trains’ layouts change independently of each other. As a result, the preservation of the invariants should be proven on a per train basis and by hiding the rest of the model. In Event-B, however, invariants are global and all other parts of the system are taken into account during their proofs, which substantially increases their complexity. This suggests that some encapsulation for the trains’ models will help simplify our proofs.

**High-level Properties of Low-level Details** An attempt to specify and prove properties such as collision-freeness (REQ 41) at a concrete level, like that described above, leads to complicated models and difficult proofs. In particular, expressing relationships between sequences, such as “containment” ($e.g.,$ a train is always within its MA) and “being disjoint” ($e.g.,$ the movement authorities of two different trains do not overlap) using information about the sequences’ head, rear, middle and connections, is far from trivial. This suggests that we should start modelling the system at an even more abstract level by omitting the detailed aspects of the sequences.

**Modelling Using Transitive Closure** Establishing invariants involving transitive closure, $i.e., \text{inv0}_8$ and $\text{inv0}_9$, is extremely complicated. In the end, we did not manage to prove all of the resulting proof obligations. One reason for this is that additional invariants were missing. However, this suggests that the decision
to use transitive closure for modelling was not a good one. In particular, if we can model at a more abstract level, we can delay the decision of which data type is used to model trains until we have more complete requirements for the data type. In fact, the formal model described in Section 5.3.2 does not use transitive closure.

**Reuse** In addition to the above problems, another motivation for using ADTs in our development is that modelling the trains’ movement authorities is similar to modelling the trains themselves. In fact, both trains and their MAs can be modelled using the same ADT, which allows the reuse of proofs.

### 5.3.2. Development using Abstract Data Types

In our development of the train control system, we develop the ADTs used to model the trains and MAs, as follows.

- We start with **Region**, an ADT that represents regions within the network.
- We subsequently instantiate **Region** with **Sequence**, an ADT representing sequences of sections.
- Finally, we instantiate **Sequence** with **ArbArray**, a specific data type corresponding to arbitrary-based arrays.

Another entity of the system that we model using ADTs is the active network that is controlled by the interlocking system.

- We start with the ADT **Network** representing all possible layouts of the active network.
- We subsequently instantiate **Network** with **Graph**, an ADT that represents network layouts more detailed in terms of graphs. The nodes of a graph correspond to sections and a graph’s edges correspond to connections between sections.

In the following, we describe the ADTs with their usage in our formal model in the order that we defined them.

**Network**

The ADT **Network** abstractly represents the interlocking system, which controls the active network on which the trains move around. **Network** has two operations.

- **enlarge**: takes a network $N$, a relation between sections $g$ (which must be a partial injective function), a set of sections $s$, and returns a new network where $g$ and $s$ become active, i.e., trains can operate on these connections and sections.
• \textit{contract}: takes a network \( N \), a relation between sections \( g \) (which must be a partial injective function), a set of sections \( s \), and returns a new network where \( g \) and \( s \) become inactive.

In Event-B, \textbf{Network} is modelled as follows.

\begin{itemize}
  \item \textbf{sets}: \texttt{NETWORK\_TYPE}
  \item \textbf{constants}: \texttt{NETWORK, enlarge, contract}
  \item \textbf{axioms}:
    \begin{align*}
    & \texttt{axm0\_1}: \texttt{NETWORK} \subseteq \texttt{NETWORK\_TYPE} \\
    & \texttt{axm0\_2}: \texttt{enlarge} \in \left( \times \left( \texttt{NETWORK} \leftrightarrow \texttt{SECTION} \right) \times \mathcal{P}\left( \texttt{SECTION} \right) \right) \rightarrow \texttt{NETWORK} \\
    & \texttt{axm0\_3}: \texttt{contract} \in \left( \times \left( \texttt{SECTION} \leftrightarrow \texttt{SECTION} \right) \times \mathcal{P}\left( \texttt{SECTION} \right) \right) \rightarrow \texttt{NETWORK}
    \end{align*}
\end{itemize}

The axioms \texttt{axm0\_1–axm0\_3} specify “typing” information of the operations. Additional axioms stating the consistency of \textbf{Network}’s operations are related to \textbf{Region} introduced in Section 5.3.2.

Using \textbf{Network}, the interlocking system can be abstractly modelled as follows, where \texttt{network} represents the currently active network.

\begin{itemize}
  \item \textbf{variables}: \texttt{network}
  \item \textbf{invariants}:
    \begin{align*}
    & \texttt{inv0\_1}: \texttt{network} \in \texttt{NETWORK}
    \end{align*}
\end{itemize}

\begin{itemize}
  \item \texttt{network\_enlarge}:
    \begin{align*}
    & \text{any } g, s \text{ where} \\
    & \texttt{network} \leftrightarrow g \mapsto s \in \text{dom}(\texttt{enlarge}) \\
    & \text{then} \\
    & \texttt{network} := \texttt{enlarge}(\texttt{network} \leftrightarrow g \mapsto s) \\
    & \text{end}
    \end{align*}
  \item \texttt{network\_contract}:
    \begin{align*}
    & \text{any } g, s \text{ where} \\
    & \texttt{network} \leftrightarrow g \mapsto s \in \text{dom}(\texttt{contract}) \\
    & \text{then} \\
    & \texttt{network} := \texttt{contract}(\texttt{network} \leftrightarrow g \mapsto s) \\
    & \text{end}
    \end{align*}
\end{itemize}
Region

Abstracting away the details of sequences, such as head, rear, middle, and connections, we start our modelling with an ADT corresponding to network regions. **Region** focuses on relationships between regions such as “contained” and “disjoint” and includes operations such as “extend”.

- **contained**: a binary relation associating a region $R_1$ with every region $R_2$ that contains $R_1$.
- **disjoint**: a binary relation associating two regions $R_1$ and $R_2$ with each other if they do not overlap.
- **extend**: takes a region $R$ and a section $s$, and returns a new region where $s$ is added to $R$.

Note that there are other operations of **Region** that we omit for clarity. **Region** is formalised as follows.

<table>
<thead>
<tr>
<th>sets</th>
<th>REGION_TYPE</th>
</tr>
</thead>
<tbody>
<tr>
<td>constants</td>
<td>REGION, contained, disjoint, extend</td>
</tr>
<tr>
<td>axioms</td>
<td></td>
</tr>
<tr>
<td>axm1_1</td>
<td>REGION ⊆ REGION_TYPE</td>
</tr>
<tr>
<td>axm1_2</td>
<td>contained ∈ REGION ↔ REGION</td>
</tr>
<tr>
<td>axm1_3</td>
<td>disjoint ∈ REGION ↔ REGION</td>
</tr>
<tr>
<td>axm1_4</td>
<td>extend ∈ REGION × SECTION → REGION</td>
</tr>
</tbody>
</table>

Constraints on the operations of the region ADT are modelled as axioms. For example, **contained** is transitive (axm1_5), **disjoint** is symmetric (axm1_6), and **extend** is strengthening with respect to **contained** (axm1_7). Note that in the following, we use $R_1 \in R_2$ to denote $R_1 \mapsto R_2 \in contained$, and $R_1 \not\in R_2$ to denote $R_1 \mapsto R_2 \in disjoint$.

| axm1_5        | $\forall R_1, R_2, R_3. R_1 \in R_2 \land R_2 \in R_3 \Rightarrow R_1 \in R_3$ |
| axm1_6        | $\forall R_1, R_2. R_2 \not\in R_1 \Rightarrow R_2 \not\in R_1$ |
| axm1_7        | $\forall R, s. R \mapsto s \in\! dom(extend) \Rightarrow R \in\! extend(R \mapsto s)$ |
| axm1_8        | $\forall R_1, R_2, R_3. R_1 \in R_2 \land R_2 \not\in R_3 \Rightarrow R_1 \not\in R_3$ |

To express the non-derailment property (REQ 42), we must link the two ADTs **Region** and **Network**. The following “inside” operation captures the relationship between them.

- **inside**: a binary relation associating a region $R$ with a network $N$ if $R$ is inside $N$. In the following, $R \supset N$ denotes $R \mapsto N \in inside$. 


constants: inside

\[
\begin{align*}
\text{axm1}_9 & : \text{inside} \in \text{REGION} \leftrightarrow \text{NETWORK} \\
\text{axm1}_{10} & : \forall R_1, R_2 \cdot R_1 \subseteq R_2 \land R_2 \supseteq N \implies R_1 \subseteq N \\
\text{axm1}_{11} & : \forall R, N, g, s \cdot R \subseteq N \land N \ni g \implies s \in \text{dom}(	ext{enlarge}) \implies \\
& \quad R \supseteq \text{enlarge}(N \ni g \implies s)
\end{align*}
\]

The current states of the active trains and their associated movement authorities are represented by two mappings \((\text{train} \quad \text{and} \quad \text{ma})\) from train identifiers \(\text{ids}\) to \(\text{REGION}\), the set of possible regions. The invariant \(\text{inv1}_3\) states that the trains always stay within their movement authorities. The invariant \(\text{inv1}_4\) states that the movement authorities of any two trains are disjoint. The invariant \(\text{inv1}_5\) states that the movement authorities are always inside the active network.

variables: ids, train, ma

invariants:
\[
\begin{align*}
\text{inv1}_1 & : \text{train} \in \text{ids} \rightarrow \text{REGION} \\
\text{inv1}_2 & : \text{ma} \in \text{ids} \rightarrow \text{REGION} \\
\text{inv1}_3 & : \forall t \cdot t \in \text{ids} \implies \text{train}(t) \subseteq \text{ma}(t) \\
\text{inv1}_4 & : \forall t_1, t_2 \cdot t_1 \in \text{ids} \land t_2 \in \text{ids} \land t_1 \neq t_2 \implies \text{ma}(t_1) \not\supseteq \text{ma}(t_2) \\
\text{inv1}_5 & : \forall t \cdot t \in \text{ids} \implies \text{ma}(t) \supseteq \text{network}
\end{align*}
\]

Properties such as collision-freeness (REQ 41) and non-derailment (REQ 42) are theorems derivable from the invariants \(\text{inv1}_3, \text{inv1}_4, \text{and} \text{inv1}_5\), and the properties relating \(\subseteq, \not\supseteq, \text{and} \supseteq\), e.g., \(\text{axm1}_6, \text{axm1}_8, \text{and} \text{axm1}_{10}\).

\[
\begin{align*}
\text{thm1}_1 & : \forall t_1, t_2 \cdot t_1 \in \text{ids} \land t_2 \in \text{ids} \land t_1 \neq t_2 \implies \text{train}(t_1) \not\supseteq \text{train}(t_2) \\
\text{thm1}_2 & : \forall t \cdot t \in \text{ids} \implies \text{train}(t) \supseteq \text{network}
\end{align*}
\]

For example, the proof of \(\text{thm1}_1\) is as follows. For all \(t_1\) and \(t_2\), we prove that if \(t_1 \in \text{ids} \land t_2 \in \text{ids} \land t_1 \neq t_2\) then \(\text{train}(t_1) \not\supseteq \text{train}(t_2)\).

\[
\begin{align*}
\text{train}(t_1) \not\supseteq \text{train}(t_2) \\
\Leftrightarrow \quad \{\text{axm1}_8\} \\
\text{train}(t_1) \subseteq \text{ma}(t_1) \land \text{ma}(t_1) \not\supseteq \text{train}(t_2) \\
\Leftrightarrow \quad \{\text{inv1}_3\} \\
\text{ma}(t_1) \not\supseteq \text{train}(t_2) \\
\Leftrightarrow \quad \{\text{axm1}_6\} \\
\text{train}(t_2) \not\supseteq \text{ma}(t_1) \\
\Leftrightarrow \quad \{\text{axm1}_8\} \\
\text{train}(t_2) \subseteq \text{ma}(t_2) \land \text{ma}(t_2) \not\supseteq \text{ma}(t_1) \\
\Leftrightarrow \quad \{\text{inv1}_3 \text{ and} \text{ inv1}_4\} \\
\checkmark
\end{align*}
\]
This proof corresponds to the informal reasoning mentioned in Section 2.2.4. It illustrates that by introducing the ADT \textbf{Region} we can abstractly state and prove central system properties such as collision-freeness and non-derailment early in the development. This leads to simple and understandable proofs.

The event \texttt{network\_enlarge} maintains \texttt{inv1\_5} trivially (with \texttt{axm1\_11}). For the event \texttt{network\_contract}, we must strengthen the guard accordingly.

\begin{verbatim}
network_contract :
  any g, s where
  network \mapsto g \mapsto s \in \text{dom(contract)}
  \forall t.t \in \text{ids} \Rightarrow ma(t) \cap contract(network \mapsto g \mapsto s)
  then
    network := contract(network \mapsto g \mapsto s)
  end
\end{verbatim}

The event \texttt{train\_extend} can be specified abstractly as follows. The last predicate of its guard ensures that the extended train never exceeds its assigned MA, and hence maintains \texttt{inv1\_3}.

\begin{verbatim}
train_extend :
  any t, s where
  t \in \text{dom(train)}
  train(t) \mapsto s \in \text{dom(extend)}
  extend(train(t) \mapsto s) \subseteq ma(t)
  then
    train(t) := extend(train(t) \mapsto s)
  end
\end{verbatim}

\section*{Sequence}

The model at this stage is abstract in two ways: (1) its dynamic behaviour is not fully described by the machine and (2) it uses the ADTs \textbf{Region} and \textbf{Network}, which are not fully “implemented”. For (2), we utilise generic instantiation to introduce more details on how \textbf{Region} and \textbf{Network} and their operations are realised. Similar to refinement, this realisation can be split into multiple instantiation steps.

In our development, we first replace \textbf{Region} by \textbf{Sequence}, which includes the following operations. Again, we omit part of the ADT’s specification for clarity.

- \texttt{prepend}: takes a sequence \( S \) and a section \( s \) and returns a new sequence where \( s \) is added to the head of \( S \).
- \texttt{head}: takes a sequence \( S \) and returns the head section of \( S \).
- \texttt{rear}: takes a sequence \( S \) and returns the rear section of \( S \).
- \texttt{middle}: takes a sequence \( S \) and returns the middle sections of \( S \).
• **connection**: takes a sequence $S$ and returns the connection between sections of $S$.

```
sets : SEQUENCE_TYPE
```

```
constants : SEQUENCE, prepend, head, rear, middle, connection
```

```
axioms :
axm2_1 : SEQUENCE ⊆ SEQUENCE_TYPE
axm2_2 : prepend ∈ SEQUENCE × SECTION → SEQUENCE
axm2_3 : head ∈ SEQUENCE → SECTION
axm2_4 : rear ∈ SEQUENCE → SECTION
axm2_5 : middle ∈ SEQUENCE → P(SECTION)
axm2_6 : connection ∈ SEQUENCE → (SECTION ↔ SECTION)
axm2_7 : ∀S·S ∈ SEQUENCE ⇒ head(S) ≠ middle(S)
axm2_8 : ∀S·S ∈ SEQUENCE ⇒ rear(S) ≠ middle(S)
```

We prove that **Sequence** is a valid representation of **Region** with the following instantiation.

```
REGION_TYPE ← SEQUENCE_TYPE
REGION ← SEQUENCE
extend ← prepend
```

We replace (instantiate) the operations *contained* and *disjoint* using *head*, *rear*, *middle*, and *connection*. For example, *contained* is instantiated as follows.

```
S_1 ⊢ S_2 | S_1 ∈ SEQUENCE ∧ S_2 ∈ SEQUENCE ∧
  connection(S_1) ⊆ connection(S_2) ∧
  middle(S_1) ⊆ middle(S_2) ∧
  (head(S_1) ∈ middle(S_2) ∨
   (head(S_1) = head(S_2) ∧ head_range(S_1) ⊆ head_range(S_2)) ∨
   (head(S_1) = rear(S_2) ∧ head_range(S_1) ⊆ rear_range(S_2))
  ) ∧
  (rear(S_1) ∈ middle(S_2) ∨
   (rear(S_1) = rear(S_2) ∧ rear_range(S_1) ⊆ rear_range(S_2)) ∨
   (rear(S_1) = head(S_2) ∧ rear_range(S_1) ⊆ head_range(S_2))
  )
```

Note that we omitted the definitions of *head_range* and *rear_range* for the sake of simplicity. These constants represent functions that return the occupied area within the head and rear section of a given sequence.

The abstract constant *disjoint* is instantiated by the following set comprehension.
The constants \textit{contained} and \textit{disjoint} are instantiated by rather large expressions. Replacing all appearances of these constants within the machines of the instantiated model would blow them up and make them unreadable. Furthermore, guards including this kind of set comprehension are not translatable to code and must be refined anyway. We therefore apply the following trick.

The ADT \texttt{Sequence} still includes two constants \textit{contained} and \textit{disjoint}. Hence, the substitution rules become simple.

\[
\begin{align*}
\text{contained} & \leftarrow \text{contained} \\
\text{disjoint} & \leftarrow \text{disjoint}
\end{align*}
\]

However, instead of constraining the constants as in \texttt{Region} we provide in \texttt{Sequence} concrete definitions of the constants, which correspond to the intended instantiation.

\[
\begin{align*}
\text{contained} &= \{ S1 \leftarrow S2 \mid S1 \in \textit{SEQUENCE} \wedge \ldots \} \\
\text{disjoint} &= \{ S1 \leftarrow S2 \mid S1 \in \textit{SEQUENCE} \wedge \ldots \}
\end{align*}
\]

The proof obligations with respect to instantiation are the same as before. The difference is that the machines still refer to the constants \textit{contained} and \textit{disjoint}. During several refinement steps, we gradually refine the events by replacing the guards including \textit{contained} and \textit{disjoint} with guards referring to \textit{middle}, \textit{head}, etc. only. This technique allows us to better structure the proof obligations related to refining the events’ guards by spreading them over different refinement steps.

For the event \texttt{train\_extend}, the refinement removes the reference to \textit{contained} (\(\in\)) in the guard \texttt{extend(train(t) \leftarrow s) \in ma(t)}.

\begin{verbatim}
train\_extend:
any t, s where
  t \in \text{dom}(train)
  head(train(t)) \neq head(ma(t))
  \ldots// other guards related to head/rear positions
then
  train(t) := prepend(train(t) \leftarrow s)
end
\end{verbatim}
Once the machine includes no references to these constants anymore the context of the ADT can be cleaned up. For this, we simply instantiate \texttt{Sequence} again with a version of it not including the constants \textit{contained} and \textit{disjoint} and with the following substitution rules.

\begin{align*}
\textit{contained} & \leftarrow \{ S1 \mapsto S2 \mid S1 \in \textit{SEQUENCE} \land \ldots \} \\
\textit{disjoint} & \leftarrow \{ S1 \mapsto S2 \mid S1 \in \textit{SEQUENCE} \land \ldots \}
\end{align*}

The machine will not change since there are no references to the constants anymore. The proof obligations related to instantiation will be trivial as the theorems to prove are of the form $x = x$, where $x$ is the definition of \textit{contained} and \textit{disjoint}, respectively.

At this point, \texttt{Sequence} is still abstract. In particular, we have not given the exact definition for sequences and we still rely on operators such as \texttt{head}, \texttt{rear}, \texttt{middle}, and \texttt{connection} and the relationships between them.

\textbf{Graph}

Similar to how \texttt{Region} is replaced with \texttt{Sequence}, we also transform \texttt{Network} into the ADT \texttt{Graph}, where sections represent nodes and connections between sections are modelled as edges. \texttt{Graph} has the following operations:

- \textit{edges}: take a graph $G$ and returns its edges (a partial injective function between sections).
- \textit{nodes}: take a graph $G$ and returns its nodes (a set of \textit{SEQUENCE}).
- \textit{add}: takes a network $G$, a relation between sections $g$ (a partial injective function), a set of sections $s$, and returns a new graph where $G$ and $s$ are added.
- \textit{remove}: takes a network $G$, a relation between sections $g$ (a partial injective function), a set of sections $s$, and returns a new graph where $g$ and $s$ are removed.

The formalisation of \texttt{Graph} is as follows. The axioms \texttt{axm3\_6} and \texttt{axm3\_7} specify that a graph’s edges connect only the nodes of the graph. The axioms \texttt{axm3\_8--axm3\_11} express how a graph’s edges and nodes change with the operations \texttt{add} and \texttt{remove}.

\begin{center}
\begin{tabular}{|c|}
\hline
\textbf{sets} : & \texttt{GRAPH\_TYPE} \\
\hline
\textbf{constants} : & \texttt{GRAPH, edges, nodes, add, remove} \\
\hline
\end{tabular}
\end{center}
**axioms:**

- **axm3.1:** $\text{GRAPH} \subseteq \text{GRAPH} \_\text{TYPE}$
- **axm3.2:** $\text{edges} \in \text{GRAPH} \rightarrow \langle \text{SECTION} \leftrightarrow \text{SECTION} \rangle$
- **axm3.3:** $\text{nodes} \in \text{GRAPH} \rightarrow \mathcal{P}(\text{SECTION})$
- **axm3.4:** $\text{add} \in \left( \begin{array}{c} \text{GRAPH} \\ \times \langle \text{SECTION} \leftrightarrow \text{SECTION} \rangle \\ \times \mathcal{P}(\text{SECTION}) \end{array} \right) \rightarrow \text{GRAPH}$
- **axm3.5:** $\text{remove} \in \left( \begin{array}{c} \text{GRAPH} \\ \times \langle \text{SECTION} \leftrightarrow \text{SECTION} \rangle \\ \times \mathcal{P}(\text{SECTION}) \end{array} \right) \rightarrow \text{GRAPH}$
- **axm3.6:** $\forall G \cdot G \in \text{GRAPH} \Rightarrow \text{dom}(\text{edges}(G)) \subseteq \text{nodes}(G)$
- **axm3.7:** $\forall G \cdot G \in \text{GRAPH} \Rightarrow \text{ran}(\text{edges}(G)) \subseteq \text{nodes}(G)$
- **axm3.8:** $\forall G, g, s \cdot G \rightarrow g \rightarrow s \in \text{dom}(\text{add}) \Rightarrow$
  \hspace{1cm} $\text{edges}(\text{add}(G \rightarrow g \rightarrow s)) = \text{edges}(G) \cup g$
- **axm3.9:** $\forall G, g, s \cdot G \rightarrow g \rightarrow s \in \text{dom}(\text{add}) \Rightarrow$
  \hspace{1cm} $\text{nodes}(\text{add}(G \rightarrow g \rightarrow s)) = \text{nodes}(G) \cup s$
- **axm3.10:** $\forall G, g, s \cdot G \rightarrow g \rightarrow s \in \text{dom}(\text{remove}) \Rightarrow$
  \hspace{1cm} $\text{edges}(\text{remove}(G \rightarrow g \rightarrow s)) = \text{edges}(G) \setminus g$
- **axm3.11:** $\forall G, g, s \cdot G \rightarrow g \rightarrow s \in \text{dom}(\text{remove}) \Rightarrow$
  \hspace{1cm} $\text{nodes}(\text{remove}(G \rightarrow g \rightarrow s)) = \text{nodes}(G) \setminus s$

**Graph** is a trivial realisation of **Network**, where the operations *add* and *remove* implement *enlarge* and *contract*.

\[
\begin{align*}
\text{NETWORK} \_\text{TYPE} & \leftarrow \text{GRAPH} \_\text{TYPE} \\
\text{NETWORK} & \leftarrow \text{GRAPH} \\
enlarge & \leftarrow \text{add} \\
contract & \leftarrow \text{remove}
\end{align*}
\]

The operation *inside* that connects **Network** and **Region** is realised by linking the operations of **Graph** and **Sequence** as follows.

\[
\text{inside} = \left\{ S \mapsto G \mid \begin{array}{l}
S \in \text{SEQUENCE} \land G \in \text{GRAPH} \land \\
\text{connection}(S) \subseteq \text{edges}(G) \land \\
\text{members}(S) \subseteq \text{nodes}(G)
\end{array} \right\}
\]

A sequence $S$ is within a graph $G$ if $S$’s connections are edges of $G$, and the members (head, middle, and rear) of $S$ are nodes of $G$. This realisation of *inside* leads to the following refinement of the event **network_contract**.

```plaintext
network_contract :
  any g, s where
  network \mapsto g \mapsto s \in \text{dom}(contract) \\
  \forall t \cdot t \in \text{ids} \Rightarrow \text{connection}(\text{ma}(t)) \cap g = \emptyset \\
  \forall t \cdot t \in \text{ids} \Rightarrow \text{members}(\text{ma}(t)) \cap s = \emptyset \\
  \text{then}
  \text{network} := \text{contract}(\text{network} \mapsto g \mapsto s) \\
  \text{end}
```
The event **network_contract** specifies that when the interlocking system disables a part of the active network, represented by \((g, s)\), there is no overlap between this part and the MA of any active train.

**ArbArray**

The model based on **Sequence** is abstract. To ensure that the model of the software controller is implementable, we must give a concrete representation for **Sequence**. In our development, we use **ArbArray** an arbitrarily-based array data type to implement **Sequence**. An arbitrarily-based array is an array that starts from an arbitrary index, in contrast to the common zero-based array that always starts from 0. More formally, each arbitrarily-based array can be represented by a tuple \(a \mapsto b \mapsto f\), where \(a\) and \(b\) are the starting and ending indices and \(f\) represents the array’s content. The operations of the arbitrarily-based array such as **head**, **rear**, **middle**, and **connection** are defined accordingly. For example, the **head** operation is defined as follows.

\[
\text{head} = (\lambda \ a \mapsto b \mapsto f \cdot a \mapsto b \mapsto f \in \text{ARRAY} \ | \ f(a))
\]

The advantage of using arbitrarily-based arrays compared to standard (zero-based) arrays is that there is no need to shift indices when extending or reducing the arrays. For example, the **prepend** operation is defined as follows.

\[
\text{prepend} = \left(\begin{array}{c}
\lambda \ (a \mapsto b \mapsto f) \mapsto s \cdot \\
\quad a \mapsto b \mapsto f \in \text{ARRAY} \land \\
\quad s \in \text{SECTION} \land \\
\quad \ldots \\
\quad (a - 1) \mapsto b \mapsto (f \leftarrow \{a - 1 \mapsto s\})
\end{array}\right)
\]

This simplifies the proof that **Sequence** is correctly implemented by the data type **ArbArray**. Arbitrarily-based arrays do not correspond to the common understanding of arrays in programming languages. To generate reasonable code, we would need to instantiate **Sequence** with a zero-based array. This is possible and we did some of the slightly more difficult proofs in the model. However, by further refining our model the representation of the trains’ sequences became obsolete. The trains’ head and rear sections provide enough information to compute MAs. The arbitrarily-based arrays only serve as an intermediate representation for the proofs.

**5.3.3. Development Summary and Comparison**

In our development of the train control system, the transformation of **Region** into **Sequence** is carried out in several instantiation steps. The benefit of having
steps with small changes in the ADTs is that the machines that are specified using ADTs can also be incrementally transformed in small steps. This also serves to decompose the proof of correctness of the systems into small, simple instantiation and refinement steps.

Our development contains five different stages (numbered 0–4), connected by instantiation relationships, where Stages 1–4 start as an instantiation of the previous stage. Each stage contains several refinement steps, developing the system’s main functionality.

**Stage 0** We formalise the system at the most abstract, generic level, using the ADTs **Region** and **Network**. In the refinement steps, we gradually introduce the active network, the active trains, the trains’ movement authorities, the movement authorities calculated by the controller, and the relationships between them.

**Stage 1–3** We carry out the transformation from the ADTs **Region** and **Network** to **Sequence** and **Graph** in three different instantiations. In **Stage 1** we instantiate the operation *contained* (the other operations such as *inside* and *disjoint* remain axiomatically defined as before). In **Stage 2** we instantiate the operation *inside* between **Region** and **Network**. Finally, in **Stage 3** we instantiate the operation *disjoint*. The refinement steps in these stages have two purposes: (1) they transform the events to use the new data types, and (2) they introduce the system’s design details, including notions like *train ahead*, *train behind*, and *last train within a section*.

**Stage 4** We instantiate **Sequence** by **ArbArray**, the arbitrarily-based array data type. We also incrementally introduce details on the calculation of the trains’ MAs.

We present statistics for our development in Table 5.1 and compare the development of the train control system with and without ADTs. Table 5.2a shows the proof statistics for our first attempt where we did not use ADTs. After 14 refinement steps and 45 difficult manual proofs, we stopped our development with numerous undischarged proof obligations remaining due to missing invariants. We would have needed additional invariants that are complex to express and would lead to even more complex proofs. Considering the proof effort needed up to this point, and the additional effort anticipated to complete the development, we concluded that continuing this way would be infeasible. Hence, it was necessary to adapt our development strategy and incorporate additional abstraction techniques to simplify the proofs.

Overall, our model using ADTs contains 67 refinements. The last machine contains 25 variables (6 that model the environment, 15 that model the state of the controller, and 4 that model the communication channels) and 30 events (11 that model the behaviour of the environment, 13 that specify how the controller
calculate MAs, and 6 that model the communication between the trains and the controller). There are 107 invariants and 56 theorems within these 67 refinements.

The large number of refinements in our development originates from two facts. First, the final machine contains 25 variables. We introduced these variables gradually into our model to keep the number of proof obligations per machine within manageable limits. Second, the definitions of the abstract constants \textit{contained} and \textit{disjoint} are quite complex. Replacing all appearance of these operations in the machine at once would result in a huge number of proof obligations for one single machine. To structure the proofs, we spread the replacement of the abstract operations over several refinement steps, only updating a couple of events in each step. Technically, the number of refinement steps could be reduced resulting in more proof obligations per step. However, this is undesirable and against the fundamental idea of stepwise refinement in Event-B.

Table 5.2b shows the proof statistics for our development using ADTs. We distinguish between proofs related to instantiations and proofs related to refinements. Overall, 14% of the proofs are related to instantiations, and the other 86% are related to refinements. As expected, proofs for the machines at the more abstract and generic levels are more automated. Most of the manual proofs originating from instantiation (in particular of Stage 4) have a similar structure that includes manually expanding the instantiation definitions. These proof steps could be automated with a dedicated proof strategy, which would further increase the amount of proof automation. Overall, the instantiation proofs have a better automation rate (82%) than the refinement proofs (62%).

The two developments are not directly comparable, since in the first attempt we had to give up on the proofs and the development has not been finished. Furthermore, we revised our refinement strategy after introducing ADTs, due to the possibility of using this way of abstraction in addition to refinement. The safety property stating the absence of derailments, e.g., is already defined in Stage 0 of the development using ADTs, while it was not yet introduced in our first attempt without ADTs.

In general, the number of proof obligations increases when using ADTs. This is due to the additional instantiation steps and their related proof obligations. However, thanks to the additional abstraction technique, the proofs related to the stated safety properties become easier. This is evident, since we were not able to continue the development without ADTs because of the many too difficult proofs. With ADTs, on the other hand, we were able to discharge all proof obligations of the entire development.

The number of refinement steps as well as the total number of discharged proof obligations indicates that the size and complexity of our case study is significantly higher than typical academic examples. Moreover, the level of detail in our models, which stems from realistic requirements, supports our claims about the relevance of our approach for large and complex real-world systems.
Table 5.1: Statistics

<table>
<thead>
<tr>
<th>14 Refinements</th>
<th>Obligations</th>
<th>Auto.</th>
<th>Manual</th>
<th>Undischarged</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>666</td>
<td>497 (75%)</td>
<td>45 (7%)</td>
<td>124 (18%)</td>
</tr>
</tbody>
</table>

(a) Development without ADTs

<table>
<thead>
<tr>
<th>Stage 0</th>
<th>Stage 1</th>
<th>Stage 2</th>
<th>Stage 3</th>
<th>Stage 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 Refinements</td>
<td>Instantiation</td>
<td>165</td>
<td>175</td>
<td>174</td>
</tr>
<tr>
<td>Instantiation</td>
<td>34</td>
<td>493</td>
<td>441</td>
<td>90</td>
</tr>
<tr>
<td>632</td>
<td>24</td>
<td>161</td>
<td>172</td>
<td>57</td>
</tr>
<tr>
<td>10</td>
<td>139</td>
<td>4</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>1365</td>
<td>1264</td>
<td>320</td>
<td>84</td>
<td>796</td>
</tr>
</tbody>
</table>

(b) Proofs statistics using ADTs

5.4. Related Work

In our approach to introducing ADTs into formal development, generic instantiation [6] is the key technique for realising the ADTs. This differs from [41] where instantiation provides a means to reuse formal models in combination with a composition technique. In particular, to guarantee the correctness of an instantiated model, carrier sets (which are assumed to be non-empty and maximal) must be instantiated by type expressions. This has been overlooked in [6] and [41].

Part of our approach was previously published in [11] and [28]. In [11], our main motivation for using ADTs was to encapsulate data and to split the development process into two parts that can be handled by a domain expert and a formal methods expert, respectively. In [28] and this chapter, we focus more on the need for alternative forms of abstraction when developing large, complex systems. We not only use ADTs to abstract away implementation details for the domain expert, we also use them as an integral part of our development from the start to abstractly specify the system’s properties and to simplify the proofs.

The development of the Theory Plug-in [17] for Rodin allows users to extend the mathematical language of Event-B, for example, by defining new data types. Theorems about new data types can be stated and later used by a dedicated tactic associated with the plug-in. There is also a clear distinction between the theory
modules (capturing data structures and their properties) and the Event-B models using the newly defined data structures.

The main difference between the Theory Plug-in and our approach is that the data types in the Theory Plug-in are concrete. One must give the complete definitions for the data types and prove theorems about them before using these data types for modelling. This bottom-up approach is in contrast with our top-down approach where the choice of implementations for ADTs can be delayed, and we can have different implementations for the ADTs. For example, instead of implementing the sequence ADT using arbitrarily-based arrays, we can use standard, zero-based arrays for the same purpose. In fact, we experimented with both implementations and decided to use arbitrarily-based arrays as they resulted in simpler proofs.

Our approach of using ADTs in Event-B is similar to work on algebraic specifications [39]. In this domain, a specification contains a collection of sorts, operations, and axioms constraining the operations. Specifications can be enriched by additional sorts, operations, and axioms. Furthermore, to develop programs from specifications, the specifications are transformed by a sequence of small “refinement” steps. During these steps, the operations are coded until the specification becomes a concrete description of a program. For each such refinement step, one must prove that the operations’ code satisfies the axioms constraining them. An algebraic specification therefore corresponds to an Event-B context, while refinement in algebraic specifications is similar to generic instantiation in Event-B.

In contrast to algebraic specifications [22], where the entire functionality of a system is modelled as ADTs (in the form of many-sorted algebras) [39], we use ADTs to abstract only part of our system’s functionality. Modelling every aspect of a complex system like our example as an algebraic specification would be very challenging. In addition to the data types, the transition systems must also be encoded as ADTs in the specification. This would require a large number of axioms to describe the transitions.

5.5. Conclusion

In this chapter we presented an approach to building formal models using ADTs and refinement. The ADTs allow us to hide details that are unimportant for proving abstract properties. One can therefore focus on building and refining abstract models of the system’s core functionality. The way we introduce ADTs in our approach allows us to utilise generic instantiation. This handles both the instantiation of an ADT by the chosen data structure as well as the generation of the required proof obligations to guarantee that the chosen structure is a valid instance of the ADT. As a large-scale case study we have successfully applied our approach to the development of a realistic train control system. We identified the limitations of only using refinement for this system and used ADTs to overcome this problem.

6.1. Introduction

Once a system’s model is sufficiently detailed, we can extract the controller part of the model. To run on given hardware, the controller part of the model must afterwards be translated into a sequential program. We identify three main challenges for this translation. First, the Event-B model must be restricted to a well-defined subset in order to generate code for a particular programming language. Well-definedness for the sublanguage thereby reflects the available data types of the target language. For example, arithmetic operations that are valid in Event-B, but not well-defined for a target language, might result in overflows at runtime because of the different domains of the integer type. Second, Event-B’s semantics is such that the event that is executed next is chosen non-deterministically from the set of enabled events. This non-determinism must be replaced by a schedule that defines an execution order on the events. As the scheduling language becomes more sophisticated, one can generate more efficient program code. Finally, it is evident that the translation must preserve the safety properties of the Event-B model.

There has been extensive related work on code generation for Event-B [3, 15, 16, 21, 36, 43]. The different approaches have limitations including restricted scheduling languages [21, 36, 43], ignoring the differences between the mathematical notation of Event-B and the target languages [3, 15, 16, 21, 43], and missing formal justification of the approach’s correctness [21, 36, 43]. More details on the limitations of the existing approaches are provided in Section 6.9.

To overcome these limitations, we present an approach to generating code from Event-B models that focuses on the translation’s correctness. We therefore concentrate on a single target language, namely C.

6.1.1. Event Execution in Event-B

An Event-B model is a specification of a non-deterministic program. The execution of an Event-B model is similar to that of Action Systems [8]. After executing the special initialisation event, any enabled event, i.e. an event with guard predicates that evaluate to true in the current state, is executed as long as there are enabled events. Any event execution sequence that follows this description is a correct run of the model. A standard event execution sequence that can be trans-
lated to a deterministic programming language like C is the following. The terms \( \text{grd}(\text{evt}_i) \) and \( \text{act}(\text{evt}_i) \) represent the conjunction of the guards and the sequential composition of the actions of event \( \text{evt}_i \), respectively.

\[
\text{act}(\text{init});
\text{while}(\text{true}){
\text{if}(\text{grd}(\text{evt}_1)){
\text{act}(\text{evt}_1)}
\}
\text{if}(\text{grd}(\text{evt}_2)){
\text{act}(\text{evt}_2)}
\}
\ldots
\text{if}(\text{grd}(\text{evt}_n)){
\text{act}(\text{evt}_n)}
\}
\]

Figure 6.1.: Possible execution order

The events are translated to branches where the conjunction of the guards becomes the branch condition and the actions become a sequence of assignments within the body. All these branches are then sequentially composed within a loop. The final program is the sequential composition of the initialisation actions with the loop. The choice for the loop condition depends on the type of program. A controller is usually designed to run forever and continuously reading sensor values and setting actuators. In this case, the loop condition should be \text{true}. Other models like the search algorithms in Appendix B include an event \text{final} that represents the successful termination of the program. In this case, the loop condition becomes the negation of the final event’s guard, resulting in the continuous execution of events until the condition for a successful termination is met.

6.1.2. Code Efficiency

The above execution order is a correct deterministic implementation of the non-deterministic Event-B model. However, it is not very efficient and we would like to give in our approach the possibility to arrange the execution of the events resulting in more efficient code. Therefore, we provide a flexible scheduling language to specify the desired execution order of the events. Our scheduling language basically provides the standard control flow statements like loops, branches and sequential composition.

Let us assume the following two events from the example in Appendix B.3.
A standard execution sequence of the translated events using branches would look as follows.

\[
\ldots
\begin{align*}
  &\text{if}(p < q \&\& f[p] > f[q])\{ \\
  &\quad \text{act}(\text{inc}) \\
  &\}\ \\
  &\text{if}(p < q \&\& f[p] \leq f[q])\{ \\
  &\quad \text{act}(\text{dec}) \\
  &\}\ \\
  &\ldots
\end{align*}
\]

Figure 6.2.: Standard execution sequence

We see that the two events have a common guard \( p < q \). If this guard evaluates to \( false \) in the first branch, the program stops the evaluation of the branch condition, jumps over the branch’s body and continues with checking the branch condition of the second branch. Since the values of \( p \) and \( q \) have not changed, the check will also be negative and is therefore useless. We can optimise this code by combining the two events where the common guard becomes the branch condition of a surrounding branch.

\[
\ldots
\begin{align*}
  &\text{if}(p < q)\{ \\
  &\quad \text{if}(f[p] > f[q])\{ \\
  &\quad\quad \text{act}(\text{inc}) \\
  &\quad\}\ \\
  &\quad \text{if}(f[p] \leq f[q])\{ \\
  &\quad\quad \text{act}(\text{dec}) \\
  &\quad\}\ \\
  &\}\ \\
  &\ldots
\end{align*}
\]

Figure 6.3.: First optimisation

With the additional branch, we can remove the extracted guard from the branch conditions of the original branches. The guards that remain in the branch conditions are negations of each other. Hence, there is no need to check the second
branch condition if the first was negative and we can further optimise the code. We combine the two inner branches in a single one with an else part.

```c
if(p<q){
  if(f(p)>f(q)){
    act(inc)
  }else{
    act(dec)
  }
}
```

Figure 6.4.: Second optimisation

As a result, we have no guards left to check before we execute the events’ actions. All the guards are guaranteed by the control flow statements. In general, there can be remaining guards that must be checked before executing the actions of the corresponding event. In this case, an additional branch around the actions with the remaining guards as branch condition would still be required.

To simplify the implementation of our code generation plug-in, we decided that all the guards must be guaranteed by the user-defined control flow statements and that only the events’ actions are translated by the tool. Hence, the tool does not have to figure out which guards are not yet guaranteed by the control flow statements. As a consequence of this simplification, it is the user’s task to specify additional branches for remaining guards, if necessary.

6.1.3. Correctness

We stated earlier that every event execution sequence is a correct implementation of the Event-B model. Therefore, a specific sequence defined by a schedule should also result in a correct implementation. This is the case if we translate the events including their guards in form of branch statements. In our approach, however, we only translate the actions and require that the guards are guaranteed by the control flow statements.

We cannot depend on the user and assume that the defined schedule guarantees the guards of the executed events. Instead, we must prove that the generated code is a correct implementation of the Event-B model. The obvious way to prove a correct implementation is to show that the generated code is a refinement of the original Event-B model. Unfortunately, there is no notion of refinement between Event-B models and C code.

To overcome this limitation, we generate an Event-B machine as described in Section 6.6, which has the equivalent behaviour as the generated code. We justify the equivalence based on their structural similarity. Because this machine includes program flow information originating from the schedule we call it the scheduled Event-B model. We can prove now that this model is a refinement of the original one. Due to the structural similarity between the scheduled model
and the generated code we can justify the following claim. If we can prove that the scheduled model is a refinement of the original Event-B model, then the generated C code is a refinement of it too and, therefore, a correct implementation.

### 6.1.4. Overview

Our code generation approach has four steps. Figure 6.5 depicts the different entities involved and their relationships together with the corresponding step in which they are provided by the user or the code generator. The “final Event-B model” represents the final refinement of an Event-B development and is the starting point for our code generation approach. First, we restrict the Event-B model via refinement to ensure that the variables are of suitable types and operations on them are well-defined. Second, we use a special scheduling language to specify a schedule for the restricted Event-B model, which describes the intended execution order on the events. Third, we execute our code generator with the schedule as input. Based on the schedule, the code generator translates the restricted model into a sequential program and thereby generates source code. In the same step, our code generator also generates a scheduled Event-B model with the equivalent behaviour as the sequential program. Finally, we prove that this scheduled model refines the restricted Event-B model, which implies the validity of the user-provided schedule.

The correctness of our translation relies on (i) the use of partial functions and well-definedness to ensure that the operations on the data types provided by the target language are valid, (ii) assertions that are annotated in the schedule and subsequently translated into invariants of the scheduled Event-B model, and (iii) the proof that the scheduled model refines the restricted Event-B model, which relies on the automatically generated invariants. The refinement proof together with the structural similarity between scheduled model and source code justifies the correctness of the generated code in our approach.

Overall, our contribution is an approach to code generation from Event-B models that guarantees that generated programs correctly implement their Event-B specifications and therefore will not incur runtime errors such as arithmetic over-
flows. The novelty of our approach is the use of well-definedness in the restriction step to prevent runtime errors, a flexible scheduling language with assertions for specifying scheduling information during the second step, and the use of refinement in the fourth step of our approach to prove the generated program code’s correctness. Based on our approach, we implemented a plug-in for the Rodin platform [5] and successfully generated code for industrial-scale case studies including an elevator control system and a train control system, both with strong safety properties. We illustrate our approach using a comparatively simple academic example from [3].

Structure.

We briefly overview the “cars on a bridge” case study from [3] in Section 6.2. We use this example to illustrate the four steps of our approach to generating code from Event-B models in Sections 6.3-6.6. In Section 6.7, we discuss possible timing issues and provide evidence for the general applicability of our approach in Section 6.8. In Section 6.9, we compare our approach with the existing code generation tools for Event-B. Part of this chapter has already been published in [27].

6.2. Running Example

In this section, we describe the “cars on a bridge” example taken from [3, Chapter 2] that we use as a running example to illustrate our approach to code generation. We use this example instead of the train control system for the sake of clarity. It is smaller, less complex and, therefore, better suited to explain our approach.

The system’s main functionality is to control the cars on a bridge between an island and the mainland. Due to the bridge’s width, only traffic in one direction is allowed at a time. The system is equipped with four sensors to detect the presence of cars entering and leaving the bridge. The system controls the two traffic lights located at both ends of the bridge. Moreover, the maximum number of cars allowed on the island is limited. The Event-B model is gradually developed in four machines. The last refinement includes environment events modelling the movement of cars that triggers the sensors and controller events setting the traffic lights accordingly. For the purpose of illustrating our approach to generating code, we focus on the following events of the last refinement.
The events ML\_out1, ML\_out2, and IL\_tl\_green are controller events and the event ML\_OUT\_DEP is an environment event. We omit other events for clarity. The constant $d$ represents the maximum number of cars allowed on the island. The variables $a$, $b$, $c$, $ml\_pass$, $il\_pass$ are controller variables and the variable $A$ is an environment variable. Other variables are shared variables representing the sensors (from the environment to the controller), i.e., $ml\_out\_10$, $il\_out\_10$, $ml\_in\_10$, $il\_in\_10$, $ML\_OUT\_SR$, and $IL\_OUT\_SR$, or the actuators (from the controller to the environment), i.e., $ml\_tl$ and $il\_tl$. The interested reader can find the exact meaning of the variables in [3, Chapter 2].

### 6.3. Well-definedness Restrictions

The final model of an Event-B development may still include parts that are not well-defined with respect to the target language. Using refinement, we restrict these remaining parts and thereby obtain a restricted model that is well-defined. Our plug-in (described in Chapter 7) checks that the model is restricted before generating source code for it. In the following, we describe the semantic differences between Event-B and C and describe our approach to establishing well-definedness.
6.3.1. Basic Types

The two basic types that our code generator supports are 32-bit integers and booleans. While the boolean type in C is equivalent to type BOOL in Event-B, the integer types have different ranges. We therefore define in Event-B the range of the C integer type as a constant C_INT \( (C\_INT = -2147483648..2147483647) \) and require that every integer variable belongs to this set. C_INT can be seen as a restricted data type in the model and we say that a variable is of type C_INT whenever it belongs to the set described by the constant C_INT, i.e., variable \( \in C\_INT \).

6.3.2. Arrays

We support one- and two-dimensional arrays for both basic types. Arrays are represented by total functions in Event-B. If a variable or constant is not of a basic type, then it must be of one of the array types in the table below, where \( k \) and \( l \) are natural numbers smaller than the maximum value in C_INT.

<table>
<thead>
<tr>
<th>Array Type</th>
<th>Declaration</th>
</tr>
</thead>
<tbody>
<tr>
<td>e</td>
<td>( e \in 0..k \rightarrow C_INT )</td>
</tr>
<tr>
<td>f</td>
<td>( f \in 0..k \rightarrow BOOL )</td>
</tr>
<tr>
<td>g</td>
<td>( g \in 0..k \times 0..l \rightarrow C_INT )</td>
</tr>
<tr>
<td>h</td>
<td>( h \in 0..k \times 0..l \rightarrow BOOL )</td>
</tr>
</tbody>
</table>

The restriction on \( k \) and \( l \) guarantees that the size of a generated array is always positive and at most the maximum number in C_INT. However, the maximal allowed size of an array depends on the target system and its memory management. Hence, we cannot guarantee that the memory allocation at the beginning of the running program will succeed.

6.3.3. Arithmetic Operators

Careless use of arithmetic operators is the source of integer overflows in software. Since the integer type in Event-B does not have a lower or upper bound, the addition of two positive integer numbers always results in a positive integer number. In a C program, however, this result might be larger than the maximum integer number and cause a runtime error or be mapped to a negative number. Either way, the outcome of the computation is different from that in the Event-B model. Due to the restriction of integer variables to the type C_INT, assignments of the form \( x := x + y \) are already checked for well-definedness when proving the preservation of the invariant \( x \in C\_INT \). The intermediate results of multiple arithmetic operations and arithmetic operations in predicates, however, are not
checked. To enforce the well-definedness of all arithmetic operations, we introduce special operators that are adapted to the integer type C\_INT and we restrict the use of each arithmetic operator to just these.

Table 6.2.: Allowed arithmetic operators

\[
\begin{align*}
c_{\text{plus}} &= \{\lambda a \mapsto b \cdot a \in \text{C\_INT} \land b \in \text{C\_INT} \land a + b \in \text{C\_INT} \mid a + b\} \\
c_{\text{minus}} &= \{\lambda a \mapsto b \cdot a \in \text{C\_INT} \land b \in \text{C\_INT} \land a - b \in \text{C\_INT} \mid a - b\} \\
c_{\text{mul}} &= \{\lambda a \mapsto b \cdot a \in \text{C\_INT} \land b \in \text{C\_INT} \land a \cdot b \in \text{C\_INT} \mid a \cdot b\} \\
c_{\text{div}} &= \{\lambda a \mapsto b \cdot a \in \text{C\_INT} \land b \in \text{C\_INT} \land b \neq 0 \land a \div b \in \text{C\_INT} \mid a \div b\} \\
c_{\text{mod}} &= \{\lambda a \mapsto b \cdot a \in \text{C\_INT} \land b \in \text{C\_INT} \land 0 \leq a \land 0 < b \mid a \mod b\}
\end{align*}
\]

The result of an integer division in Event-B is always rounded towards zero as in the C99 standard. In C89 and C90, however, it is implementation dependant whether the result of an integer division is rounded towards zero or towards minus infinity. This difference is important when the integer division result is a negative number. When using a compiler compliant to C89 or C90, the definition of the $c_{\text{div}}$ operation must be adapted to prevent negative results and a possibly inconsistent translation. No such action is required for the modulo operator since the domain of Event-B’s modulo operation is already restricted to natural numbers.

Due to the use of lambda expressions in the operator’s definition, arithmetic operations change from infix notation to function applications in the model. We keep this style in the translation to source code and define macros to replace the function calls during compilation by the standard operators.

Table 6.3.: Definition of the allowed arithmetic operators

\[
\begin{align*}
#define \ c_{\text{plus}}(x,y) \ (x+y) \\
#define \ c_{\text{minus}}(x,y) \ (x-y) \\
#define \ c_{\text{mul}}(x,y) \ (x*y) \\
#define \ c_{\text{div}}(x,y) \ (x/y) \\
#define \ c_{\text{mod}}(x,y) \ (x\%y)
\end{align*}
\]

6.3.4. Events

For the translation of events to source code, an event’s parameters must be fixed to specific values. Theoretically, an event parameter that is fixed to a single value is not that useful as any occurrence of the parameter in guards and actions could just be replaced by its fixed value. For practical reasons, we support event parameters as local storage for computation results. If the result of a computation is used in more than one action, it is more efficient to do the computation only once and temporarily store the result.
A core concept of our approach is that the guards of the events are not translated, but their evaluation to true is guaranteed by the control flow statements of the schedule or more precisely by the specified branch conditions, loop conditions, and assertions. The only guards that are translated are those that specify the value of an event parameter. We require in the restricted Event-B model that for every event parameter there is exactly one guard of the form parameter = ..., where the right-hand side of the equation must be an expression of type BOOL or C_INT.

Since an event’s actions denote parallel assignments, the order of the actions does not matter in Event-B. This changes when we translate the actions into a sequence of single assignments. As a result, the right-hand side of the assignments in the source code cannot refer to the before-values of the variables. To overcome this issue, we restrict the actions of the event so that the right-hand side of an assignment does not refer to variables that already occurred on the left-hand side of a previous assignment. If this restriction is not guaranteed, the developer must either rearrange the actions where possible or introduce parameters as auxiliary variables to store the before-value of the conflicting variables. This task could be automated in a future version of our plug-in.

6.3.5. Predicates and Expressions

We restrict predicates and expressions to a subset of the Event-B syntax for which we provide the translation mappings presented in Tables 6.11 and 6.12 in Section 6.5. In developments with arrays, there are often events with guards that contain quantifiers to express predicates on arrays. We therefore developed patterns for translating quantified predicates. In our approach, quantified predicates are translated to a function call of a dedicated function that evaluates the quantified predicate.

6.3.6. Assignments

In Table 6.14 in Section 6.5 we present the allowed assignments for updating variables in an event’s action. The right-hand side of an assignment to a variable of type BOOL or C_INT must be an expression of the corresponding type. The update of arrays is slightly more difficult. We provide different translation rules for updating arrays at one or more positions and for overwriting an array with a set of index-value pairs. The bound variable used in the set comprehension is translated to the iteration variable of a for-loop.

6.3.7. Example

Returning to the “cars on a bridge” example, we first restrict the context of the development, i.e., the values for the constant $d$ to $C\_INT$. There are two options that we can take.
1. We apply generic instantiation [28] to give \( d \) a concrete value (say \( d = 20 \)) and prove that \( d \in C\_INT \) as a theorem.

2. We add an axiom, i.e., \( d \in C\_INT \) to further constrain \( d \). In this way, \( d \) is left undefined and the user must define its concrete value within the program code. It is then the user’s task to ensure that the concrete value satisfies the axioms.

In terms of safety guaranties, the first option is preferable as we prove that the values chosen for the constants imply the specified axioms. Hence, the current version of our plug-in follows this approach. The second option provides more flexibility as the definition of the constants’ values can be written into a header file. The constants represent the parameters of the system and can easily be changed without generating new code. However, we have no practical way yet to enforce that the values in the header file are checked with respect to the model’s axioms.

We also restrict the variables of the machine of the development. More precisely, integer variables of the machine (e.g., \( a, b, c \)) must be restricted to \( C\_INT \). This can be done by proving the corresponding conditions, i.e., \( a \in C\_INT, b \in C\_INT, \) and \( c \in C\_INT \) as invariants or theorems of the machine. In our example, we can prove these conditions as theorems derivable from the restriction of \( d \) to \( C\_INT \), the fact that all variables are natural numbers, and the invariant \( a + b + c \leq d \). Moreover, we replace all occurrences of arithmetic operators in the events’ actions by their well-defined version. For example, events **ML\_out1** and **ML\_out2** are restricted as follows.

```plaintext
ML\_out1:
  when
    ml\_out\_10 = TRUE
    a + b + 1 < d
  then
    a := c\_plus(a \to 1)
    ml\_pass := 1
    ml\_out\_10 := FALSE
  end

ML\_out2:
  when
    ml\_out\_10 = TRUE
    a + b + 1 = d
  then
    a := c\_plus(a \to 1)
    ml\_tl := red
    ml\_pass := 1
    ml\_out\_10 := FALSE
  end
```

Note that arithmetic operations used in event guards need not be restricted, except for those used to define parameters, since only parameter definitions are translated.

### 6.4. Scheduling the Model

To specify the execution order on the events of the restricted model, we provide the following scheduling language in our plug-in.
### Table 6.4: Scheduling language

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;schedule&gt;</code></td>
<td><code>::= &lt;sequence&gt;</code></td>
</tr>
<tr>
<td><code>&lt;sequence&gt;</code></td>
<td>`::= &lt;sequence&gt; ;{a} &lt;sequence&gt;</td>
</tr>
<tr>
<td><code>&lt;block&gt;</code></td>
<td>`::= &lt;event&gt;</td>
</tr>
<tr>
<td><code>&lt;event&gt;</code></td>
<td><code>::= e</code></td>
</tr>
<tr>
<td><code>&lt;branch&gt;</code></td>
<td>`::= if(c) &lt;body&gt; else &lt;body&gt; fi</td>
</tr>
<tr>
<td><code>&lt;loop&gt;</code></td>
<td><code>::= do(c) &lt;body&gt; od</code></td>
</tr>
<tr>
<td><code>&lt;body&gt;</code></td>
<td>`::= &quot;&quot;</td>
</tr>
</tbody>
</table>

The symbols `a` and `c` represent a list of assertions and a loop or branch condition, respectively. The symbol `e` represents a reference to an event of the restricted model. The difference between a body and a sequence is that the body can be empty. For convenience, we can omit `{a}` if there are no assertions required between two sequentially composed sequences. Furthermore, our scheduling language provides two versions for branch blocks. In contrast to the long version the short one has no “else” part.

#### 6.4.1. Initialisation

Our train control system is designed to run forever or at least as long as the controller is powered up. The same applies to the controller of the “cars on a bridge” example. This means, the controller’s schedule, which represents one run, is executed many times. The initialisation of the controller should happen only once, prior to the first run. Therefore, we do not include the initialisation event in the schedule. Additional code that our plug-in generates makes sure that the generated code of the initialisation event is executed first. Afterwards, it repeatedly executes the generated code of the scheduled events.

#### 6.4.2. Example

The first part of our schedule is as follows. The numbers are automatically generated in the editor of our plug-in.

Note that arithmetic operations used in the branches must be restricted. Moreover, the assertion `{ml_out_10 = FALSE}` before the branch at position 7 allows us to avoid checking this condition in the branch.

#### 6.5. Translation to Source Code

In this section we present the translation of the schedule and the Event-B terms and events. We use $T(x)$ to represent the translation of a string $x$ that complies with Event-B syntax or with our scheduling language defined in Section 6.4.
6.5.1. Scheduling Elements

The translation of the schedule is straightforward using if-else statements and while-loops. Event blocks are references to events in the model. The translation of events is described in Section 6.5.5. Note that we do not translate assertions, which are only used for the proof of correctness. Branch and loop conditions are predicates that are translated according to Section 6.5.6.

<table>
<thead>
<tr>
<th>Schedule</th>
<th>C</th>
</tr>
</thead>
</table>
| <sequence1> ;{a} <sequence2> | $T(<sequence1>)$  
| | $T(<sequence2>)$ |
| if(c) <body> else <body> fi | if ($T(c)$) {
| | $T(<body>)$ |
| | } else {
| | $T(<body>)$ |
| | } |
| do(c) <body> od | while ($T(c)$) {
| | $T(<body>)$ |
| | } |

6.5.2. Basic Types

In our approach, we currently support two types for variables: 32-bit integers and booleans. Type declarations are translated straightforward.
6.5.3. Arrays

In Event-B, we define arrays as functions with a domain of the form $0..k$ for one-dimensional arrays and $0..k \times 0..l$ for two-dimensional arrays. We use the integer expressions $k$ and $l$ to specify the size in the arrays’ translated declarations.

6.5.4. Arithmetic Operators

As described in Section 6.3.3, we use special functions for arithmetic operations. The function applications are translated to C with only small syntactic adaptations. Macros in the generated C code transfer the function applications to standard C operators.

6.5.5. Events

The events’ guards (only parameter definitions) are predicates and are translated according to Table 6.11 in Section 6.5.6. The event’s actions are assignments and translated according to Section 6.5.7. The structure of the translated events is as follows.
Table 6.10.: Translation of events

<table>
<thead>
<tr>
<th>Event-B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ANY s, t</strong></td>
<td>{ int ( s = T(E_i(v,c)) );</td>
</tr>
<tr>
<td><strong>WHERE</strong></td>
<td>bool ( t = T(E_b(v,c,s)) );</td>
</tr>
<tr>
<td>( s = E_i(v,c) )</td>
<td>( vi = T(E_i(v,c,s,t)) );</td>
</tr>
<tr>
<td>( s \in C_{\text{INT}} ) (theorem)</td>
<td>( vb = T(E_b(v,c,s,t)) );</td>
</tr>
<tr>
<td>( t = E_b(v,c,s) )</td>
<td>}</td>
</tr>
<tr>
<td>( t \in \text{BOOL} ) (theorem)</td>
<td></td>
</tr>
<tr>
<td><strong>THEN</strong></td>
<td></td>
</tr>
<tr>
<td>( vi := E_i(v,c,s,t) )</td>
<td></td>
</tr>
<tr>
<td>( vb := E_b(v,c,s,t) )</td>
<td></td>
</tr>
<tr>
<td><strong>END</strong></td>
<td></td>
</tr>
</tbody>
</table>

6.5.6. Predicates and Expressions

The translation of the basic predicates and expressions is straightforward and similar to the translation mappings of the other approaches. For non-basic predicates and expressions we use the associated abstract syntax tree (AST) (that is generated by Rodin’s parser) to determine the translation sequence of the subterms. We enforce that the evaluation order in the generated code is equivalent to that in Event-B by generating parenthesis around the translated nodes of the AST.

Noteworthy is the possibility in our approach to translate quantified predicates, which are useful to express conditions in connection with arrays.

Table 6.11.: Translation of predicates

<table>
<thead>
<tr>
<th>Event-B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \top )</td>
<td>true</td>
</tr>
<tr>
<td>( \bot )</td>
<td>false</td>
</tr>
<tr>
<td>( x \land y )</td>
<td>( T(x) \land T(y) )</td>
</tr>
<tr>
<td>( x \lor y )</td>
<td>( T(x) \lor T(y) )</td>
</tr>
<tr>
<td>( x \Rightarrow y )</td>
<td>( !T(x) \lor T(y) )</td>
</tr>
<tr>
<td>( x \Leftrightarrow y )</td>
<td>( (!T(x) \lor T(y)) \land (T(y) \lor !T(x)) )</td>
</tr>
<tr>
<td>( \neg x )</td>
<td>( !T(x) )</td>
</tr>
<tr>
<td>( a = b )</td>
<td>( T(a) == T(b) )</td>
</tr>
<tr>
<td>( a \neq b )</td>
<td>( T(a) != T(b) )</td>
</tr>
<tr>
<td>( a &lt; b )</td>
<td>( T(a) &lt; T(b) )</td>
</tr>
<tr>
<td>( a \leq b )</td>
<td>( T(a) &lt;= T(b) )</td>
</tr>
<tr>
<td>( a &gt; b )</td>
<td>( T(a) &gt; T(b) )</td>
</tr>
<tr>
<td>( a \geq b )</td>
<td>( T(a) &gt;= T(b) )</td>
</tr>
</tbody>
</table>
Table 6.12.: Translation of expressions

<table>
<thead>
<tr>
<th>Event-B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>identifier</td>
<td>identifier</td>
</tr>
<tr>
<td>TRUE</td>
<td>true</td>
</tr>
<tr>
<td>FALSE</td>
<td>false</td>
</tr>
<tr>
<td>( f(a) )</td>
<td>( f[T(a)] )</td>
</tr>
<tr>
<td>( f(a \rightarrow b) )</td>
<td>( f[T(a)][T(b)] )</td>
</tr>
<tr>
<td>( c_operator(a \rightarrow b) )</td>
<td>( c_operator(T(a), T(b)) )</td>
</tr>
</tbody>
</table>

Table 6.13.: Translation of quantified predicates

<table>
<thead>
<tr>
<th>Event-B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \forall i \cdot i \in c_upto(j \rightarrow k) \Rightarrow P(v, c, i) )</td>
<td>eval_uid()</td>
</tr>
<tr>
<td>bool eval_uid() {</td>
<td></td>
</tr>
<tr>
<td>for(int i=T(j); i&lt;=T(k); i++) {</td>
<td></td>
</tr>
<tr>
<td>if (!T(P(v, c, i)))</td>
<td></td>
</tr>
<tr>
<td>return false;</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td>return true;</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
</tbody>
</table>

| \( \exists i \cdot i \in c\_upto(j \rightarrow k) \land P(v, c, i) \) | eval_uid()                      |
| bool eval_uid() { |
| for(int i=T(j); i<=T(k); i++) { |
| if (T(P(v, c, i))) |
| return true; |
| } |
| return false; |
| } |

To restrict the range of the iteration to \( C\_INT \) we define a special operator similar to the arithmetic operations. Instead of using Event-B’s range operator \( (j \ldots k) \) the following operator must be used.

\[
c\_upto = \{ \lambda a \mapsto b \cdot a \in C\_INT \land b \in C\_INT \mid a \ldots b \}\]

6.5.7. Assignments

For the translation of assignments we provide different patterns for updating arrays. We can either update an array at one or more fixed positions or we can iterate through the array and use a predicate to evaluate at runtime which positions are updated. Furthermore, it is possible to initialise an entire array to a fixed value.
Table 6.14.: Translation of assignments

<table>
<thead>
<tr>
<th>Event-B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{vint} := \text{intexpr}$</td>
<td>$\text{vint} = T(\text{intexpr})$;</td>
</tr>
<tr>
<td>$\text{vbool} := \text{boolexpr}$</td>
<td>$\text{vbool} = T(\text{boolexpr})$;</td>
</tr>
<tr>
<td>$f(a) := b$</td>
<td>$f[a] = T(b)$;</td>
</tr>
<tr>
<td>$g(a \mapsto b) := r$</td>
<td>$g[a][b] = T(r)$;</td>
</tr>
<tr>
<td>$f := f \iff {a1 \mapsto b1} \iff \ldots \iff {am \mapsto bm}$</td>
<td>$f[T(a1)] = T(b1);$;</td>
</tr>
<tr>
<td></td>
<td>$\ldots$</td>
</tr>
</tbody>
</table>
| $f := f \iff \{i \cdot i \in c\_upto(j \mapsto k) \land P(v,c,i) | for(int i=T(j); i<=T(k); i++){ 
| \quad E_1(v,c,i) \mapsto E_2(v,c,i)\}$ | $f[T(E_1(v,c,i))] = T(E_2(v,c,i));$ 
|                               | }                           |
| $f := c\_upto(a \mapsto b) \times \{c\}$ | for(int i=T(a); i<=T(b); i++){ 
|                               | $f[i] = T(c);$               |
|                               | }                           |

6.5.8. Example.

The C code generated corresponding to the above snippet of our schedule is as follows.

```c
if (ml_out_10 == true){
    if (c_plus(c_plus(a,b),1) < d){
        a = c_plus(a,1);
        ml_pass = 1;
        ml_out_10 = false;
    }
    else{
        a = c_plus(a,1);
        ml_tl = red;
        ml_pass = 1;
        ml_out_10 = false;
    }
}
if ((il_tl == red && (0 < b && (a == 0 &&
    (ml_pass == 1 && IL_OUT_SR == on))))) {
    il_tl = green;
    ml_tl = red;
    il_pass = 0;
}
```

73
6.6. Proving the Correctness of the Scheduled Model

As mentioned in Section 6.1, we do not translate the guards when generating code for the events. Hence, we must prove that the chosen schedule is valid, meaning that all guards are guaranteed to hold by the control flow statements in the translated code. We do this by generating an Event-B machine (i.e. the scheduled Event-B model) that has the same behaviour as the generated code and prove that this machine is a refinement of the restricted machine. If we can successfully prove this refinement relation, then the generated C code is a refinement of the restricted machine too, due to the structural similarity between scheduled machine and generated code. Hence, the generated C code is a correct implementation of the Event-B model.

The generation process of this scheduled machine consists of four steps. First we generate an empty Event-B machine and declare it to be a refinement of the restricted machine, i.e. the machine that we used as basis for code generation. Second, we generate for every event block an event in the scheduled machine and declare it to refine the corresponding event in the restricted machine. Third, we generate events in the scheduled machine that simulate the behaviour of the control flow statements that we declared in the schedule and have been translated to source code. In the fourth step, we generate invariants to state which predicates hold at the different positions within the program flow. We describe the steps in more detail in the following sections.

6.6.1. Step 1 – Machine Body

We generate an empty machine and declare it as a refinement of the restricted machine. This will afterwards trigger the built-in proof obligation generator of the Rodin platform to generate all proof obligation necessary to prove refinement. We copy all variables from the restricted machine to the scheduled machine and add a new variable \( pc \) of type \( C\_INT \) to encode the program flow. The variable \( pc \) represents the program counter and its value corresponds to the numbers on the left side of the schedule. We specify that the machine sees the same context as the restricted machine does.

6.6.2. Step 2 – Translated Events

According to the translation rule for events in Section 6.5.5, the guards of an event (with the exception of parameter initialisations) are not translated. Hence, the generated events in the scheduled machine do not have guards either (except for the parameter initialisations). To prove refinement, we declare the generated events to be the refinements of their corresponding events within the restricted machine. Without any additional information we would not be able to prove refinement, since all proof obligations regarding guard strengthening would fail.

We need the additional information at what position in the program flow the actions of an event are executed and which predicates are guaranteed to hold at
this position. The latter is provided by invariants as described in Step 4. To
declare the position of the event execution within the program flow, we add a

guard of the form \( pc = i \) to the event, where \( i \) corresponds to the line number
of the event block in the schedule. Furthermore, we add the action \( pc := pc + 1 \)
to the event to simulate the increment of the program counter after executing the
event.

The initialisation event is copied from the restricted machine with the additional
action \( pc := 0 \) to initialise the new variable \( pc \).

**Example.** The event block at position 2 in our example’s schedule refers to event
\( ML\_out1 \). Figure 6.6 depicts the corresponding line in the schedule (a) together
with the details of the restricted event \( ML\_out1 \) (b) and shows the generated
code for this event (c) next to the generated event in the scheduled machine (d).
6.6.3. Step 3 – Control Flow Structures

Since there are no control flow statements in Event-B, we have to encode them in the model as follows. In the scheduled machine, we generate new events that simulate the update of the program counter according to the schedule. For every branch and loop block in the schedule, we generate a set of events that simulate the behaviour of the corresponding block. The simulated behaviour is identical to the behaviour of the control flow statements in the generated code.

Branch Block. For every branch block, we generate the following set of events. The events differ slightly depending on whether the branch has an “else” part or not. The symbols $s$, $m$ and $e$ represent the block’s start, middle and end position, respectively and correspond to the line numbers within schedule. The branch condition is represented by $bc$.

<table>
<thead>
<tr>
<th>long form</th>
<th>short form</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>s:</strong> ( \text{if}(bc) )</td>
<td><strong>s:</strong> ( \text{if}(bc) )</td>
</tr>
<tr>
<td>( \vdots )</td>
<td>( \vdots )</td>
</tr>
<tr>
<td><strong>m:</strong> \text{else}</td>
<td><strong>e:</strong> \text{fi}</td>
</tr>
<tr>
<td>( \vdots )</td>
<td>( \vdots )</td>
</tr>
<tr>
<td><strong>e:</strong> \text{fi}</td>
<td></td>
</tr>
</tbody>
</table>

Figure 6.7: Branch block in the schedule

```
if _true :
  when
    pc = s
  bc
  then
    pc := pc + 1
  end

if _false (long form) :
  when
    pc = s
    \neg bc
  then
    pc := m + 1
  end

if _false (short form) :
  when
    pc = s
    \neg bc
  then
    pc := e
  end

if _exit (only long form) :
  when
    pc = m
  then
    pc := e
  end

if _end :
  when
    pc = e
  then
    pc := pc + 1
  end
```

Loop Block. For a loop block we generate the following three events. The symbols $s$ and $e$ represent the block’s start and end position, respectively and correspond to the line numbers within the schedule. The loop condition is represented by $lc$. 

76
6.6.4. Step 4 – Invariants

We generate invariants in the machine to express what predicates hold at certain positions in the schedule. All invariants are of the form $pc = i \Rightarrow p_1 \land p_2 \land \cdots \land p_n$, where $i$ is a position in the schedule and $p_1 \land p_2 \land \cdots \land p_n$ is the conjunction of the predicates that hold at this position. Note that every predicate $p_i$ in a list of predicates $(p_1, p_2, \ldots, p_n)$ holds iff the conjunction of these predicates $p_1 \land p_2 \land \cdots \land p_n$ holds.

The information about the predicates that hold at the different positions is extracted from the defined schedule. We associate to every block in the schedule two lists of predicates and call them the block’s preconditions and postconditions. The preconditions are the predicates that are guaranteed to hold at the beginning of the block. The postconditions are the predicates that must be established at the end of the block. These pre- and postconditions can be computed for every block in the schedule following a simple algorithm as described in Section 7.2.

The algorithm traverses through the schedule and thereby collects the branch and loop conditions to determine the pre- and postconditions for every block. To calculate the preconditions, the algorithm uses a depth-first path through the abstract syntax tree (AST) representing the schedule. According to our scheduling language, event blocks and empty bodies (""") build the leaves of the AST where the algorithm starts backtracking. At nodes where two sequences are sequentially composed, the algorithm user the collected preconditions when entering the left subtree and uses the associated user-defined assertions when entering the right subtree.

To calculate the postconditions, the algorithm traverses a second time through the AST. This time, it uses a depth-first path where it visits the right subtree of a sequential composition before its left one. The algorithm uses the collected post-
conditions when entering the right subtree and uses the associated user-defined assertions when entering the left subtree.

Using our algorithm, we compute the preconditions in a forward direction to discover what predicates are guaranteed when reaching an event block. We then try to prove that these predicates imply the removed guards of the event. With the weakest precondition calculus [20], in contrast, we would start at the event block where the event’s guards are the weakest precondition for the event block. We then calculate the weakest precondition in a backward direction until we reach the beginning of the schedule body. If the weakest preconditions of the schedule body is not $\top$, we know that the guards are not guaranteed by this schedule. The advantage of our approach is the easy computation of the guaranteed preconditions which can be automated.

Based on the computed pre- and postconditions, we generate for every block in the schedule a set of invariants. In the following, the predicates $\text{pre}_1, \ldots, \text{pre}_n$ represent a block’s preconditions and $\text{post}_1, \ldots, \text{post}_m$ a block’s postconditions.

**Event Block**

An event block is represented in the schedule as a single line at position $s$. This position represents the beginning of the event block, i.e. before the event’s actions are executed. Hence, the block’s preconditions hold at position $s$ and we generate the following invariant.

$$pc = s \Rightarrow \text{pre}_1 \land \text{pre}_2 \land \cdots \land \text{pre}_n$$

Note that we do not need to generate an invariant regarding the block’s postconditions. The postconditions must be established after the execution of the event’s actions, which corresponds to position $s + 1$. This position already belongs to another block and the invariant regarding this position will be generated in this context.

**Loop Block**

A loop block is represented by the keywords `do` and `od`, which are associated with the positions $s$ and $e$, respectively. The position $s$ represents the beginning
of the loop block where its preconditions hold. We therefore generate the following invariant.

$$pc = s \Rightarrow \text{pre}_1 \land \text{pre}_2 \land \cdots \land \text{pre}_n$$

In our approach, the conjunction of the preconditions is also the loop invariant that must hold after executing the loop’s body. Hence, the preconditions must also hold at position $e$ where the program will jump back to $s$ to start over the loop. We generate the following invariant associated with position $e$.

$$pc = e \Rightarrow \text{pre}_1 \land \text{pre}_2 \land \cdots \land \text{pre}_n$$

Note that we do not need to generate an invariant regarding the block’s postconditions. The postconditions must be established if the loop condition does not hold and we leave the loop block, which corresponds to position $e + 1$. This position already belongs to another block and the invariant regarding this position will be generated in this context.

Loop Invariant. Defining the loop invariant to be the conjunction of the preconditions is a valid choice for our approach. The purpose of the control flow statements is to arrange the events and to guarantee their guards. If an event is inside a loop block, its guards should be guaranteed by the loop block’s preconditions together with the loop condition. These preconditions must also hold after the execution of the loop body because we defined them as loop invariant. If the event within the body changes the variables such that one or more preconditions become false, then most likely also the event’s guard will be disabled. This indicates that the event is not meant to be executed multiple times in a row. Hence, putting this event into a loop block is probably not a good idea.

In terms of correctness, this is no issue. If the preconditions are not re-established after the loop body, we will not be able to prove refinement and are informed about the possible incorrectness of the generated code. In our examples, as well as in the train control system, we did not face any limitations due to this design of the loop invariant. As an advantage, it is not necessary to apply elaborate techniques to discover the loop invariant based on the variable transformations inside the loop body.
A branch block is represented by the keywords if, else, and fi, which are associated with the positions s, m, and e, respectively. In the short form of the branch block there is no else keyword and no position m accordingly. The position s represents the beginning of the branch block where its preconditions hold. We therefore generate the following invariant.

\[
 pc = s \Rightarrow pre_1 \land pre_2 \land \cdots \land pre_n
\]

The position e represents the end of the branch block where block’s postconditions must be established. We therefore generate the following invariant.

\[
 pc = e \Rightarrow post_1 \land post_2 \land \cdots \land post_m
\]

In the long form of the branch condition we have the keyword else at position m. This position will be reached after executing the “then” part of the branch block where the program will directly jump to block’s end at position e. Hence, also at position m the block’s postconditions must be established.

\[
 pc = m \Rightarrow post_1 \land post_2 \land \cdots \land post_m
\]

With these four steps, we have generated an Event-B machine that has the same behaviour as the generated code. We do not formally prove the behavioural equivalence. Hence, only the scheduled machine is formally proved to be a correct refinement of the original Event-B model. Due to the simple scheduling language and the common understanding of the behaviour of branches and loops, the structural similarity between the scheduled machine and the generated code is easily observable. As a result, we justify the correctness of our code generation approach based on the refinement proof of the scheduled model and the structural similarity between the scheduled model and the generated C code.

To formally prove the equivalence of the scheduled machine and the C code, we would need to define and compare the semantics of the used subsets of both
languages. More precisely, we would need to define for both sublanguages their operational semantics and show the equivalence with a bisimulation relation between these two semantics.

A different approach to increase the confidence about the generated code’s correctness would be the application of code verification techniques. Instead of generating a scheduled model and proving refinement we would annotate the original guards of the events as assertions to the generated code at the corresponding positions. There exist tools such as VCC [18] to proof the correctness of the specified assertions in a C program. VCC uses Boogie [10] to generate the verification condition of the program and tries to prove this condition with help of an SMT solver. The verification condition is generated using the weakest precondition calculus [20].

If a program becomes larger and more complex, the verification condition becomes larger too. Since proving the verification condition is a Boolean Satisfiability Problem (SAT), which is a NP-complete problem, VCC cannot guarantee the successful verification of complex program code. Furthermore, we have to use different tools for the verification of programs in different target languages, provided that such tools exist. With our approach, the justification of the generated code’s correctness happens within the Event-B framework and is independent of the chosen target language.

Since C code must be compiled to run on hardware, the same question about equivalence appears between C code and assembler. A solution could be to use a verified compiler, if available for the target hardware. Since the choice of hardware for the train control system as well as the preferred version of the C language was unknown and out of the project’s scope, we did not further investigate in formally proving the behavioural equivalence.

6.6.5. Example

Based on the (automatically generated) program counter associated with the statements, the scheduled Event-B model corresponding to the above schedule snippet of our running example is as follows. Most events are required for modelling the program flow of the schedule determined by the program counter $pc$.

```plaintext
if _ml_out_10_true_:
    when
        pc = 0
        ml_out_10 = TRUE
    then
        pc := pc + 1
    end

if _ml_out_10_false_:
    when
        pc = 0
        ¬(ml_out_10 = TRUE)
    then
        pc := 6
    end
```
if ml_out_true :
  when
      pc = 1
      c_plus(c_plus(a → b) → 1) < d
    then
      pc := pc + 1
  end

if ml_out_false :
  when
      pc = 1
      ¬(c_plus(c_plus(a → b) → 1) < d)
    then
      pc := 4
  end

ML_out1 :
  when
      pc = 2
    then
      a := c_plus(a → 1)
      ml_pass := 1
      ml_out_10 := FALSE
      pc := pc + 1
  end

if ml_out_exit :
  when
      pc = 3
    then
      pc := 5
  end

ML_out2 :
  when
      pc = 4
    then
      a := c_plus(a → 1)
      ml_tl := red
      ml_pass := 1
      ml_out_10 := FALSE
      pc := pc + 1
  end

if ml_out_end :
  when
      pc = 5
    then
      pc := pc + 1
  end

if ml_out_10_end :
  when
      pc = 6
    then
      pc := pc + 1
  end

if il_tl_green_true :
  when
      pc = 7
      il_tl = red ∧ 0 < b ∧ a = 0 ∧ ml_pass = 1 ∧ IL_OUT_SR = on
    then
      pc := pc + 1
  end
if _il_tl_green_false :
    when
        pc = 7
    ¬(il_tl = red ∧ 0 < b ∧ a = 0 ∧ ml_pass = 1 ∧ IL_OUT_SR = on)
    then
        pc := 9
    end

IL_tl_green :
    when
        pc = 8
    then
        il_tl := green
        ml_tl := red
        il_pass := 0
        pc := pc + 1
    end

if _il_tl_green_end :
    when
        pc = 9
    then
        pc := pc + 1
    end

We generate the following invariants for every position in the schedule to capture
the effect of the control flow statements and the user-defined assertions.

invariants :
    if _ml_out_10_Pre :
        pc = 0 ⇒ ⊤
    if _ml_out_Pre :
        pc = 1 ⇒ ml_out_10 = TRUE
    ml_out1_Pre :
        pc = 2 ⇒ ml_out_10 = TRUE ∧
                        c_plus(c_plus(a → b) → 1) < d
    if _ml_out_Post :
        pc = 3 ⇒ ml_out_10 = FALSE
    ml_out2_Pre :
        pc = 4 ⇒ ml_out_10 = TRUE ∧
                        ¬(c_plus(c_plus(a → b) → 1) < d)
    if _ml_out_Post :
        pc = 5 ⇒ ml_out_10 = FALSE
    if _ml_out_10_Post :
        pc = 6 ⇒ ml_out_10 = FALSE
    if _il_tl_green_Pre :
        pc = 7 ⇒ ml_out_10 = FALSE
    IL_tl_green_Pre :
        pc = 8 ⇒ ml_out_10 = FALSE ∧
                    (il_tl = red ∧ 0 < b ∧ a = 0 ∧
                        ml_pass = 1 ∧ IL_OUT_SR = on)
    if _il_tl_green_Post : pc = 9 ⇒ ⊤

Notice how the invariants take into account the effect of the nested branches, e.g.
when pc = 2, and of assertions, e.g. when pc = 8. Proving that the scheduled
Event-B model refines the restricted Event-B model is straightforward with these
invariants, except for the problem regarding shared variables.

6.6.6. Shared Variables and Atomicity

Our schedule imposes an atomicity assumption, captured by the scheduled Event-
B model, representing the semantics of the program code. The atomicity is in-
dicated by the values of the program counter pc. For example, we assume that
the evaluation of conditions in branches and loops are atomic. Moreover, we also assume that the assignments of the original events (which are translated as sequential updates) are executed atomically.

However, we break the atomicity assumption between checking the guards and executing the actions of the original events. In particular, the evaluation of the event guards is often distributed to different branch and loop conditions. For example, the guard of IL\_tl\_green is partially checked by the branch condition at $pc = 7$ and partially guaranteed (assertion $ml\_out\_10 = \text{FALSE}$) by the program flow before that. Since this atomicity assumption differs from the atomicity assumption of the restricted Event-B model, where the evaluation of an event’s guards and the execution of its actions are assumed to be atomic, inconsistency can arise. This is in particular the case when shared variables are used in the event’s guard.

We use the term shared variable for variables which are accessed by controller and environment events. Examples for such shared variables are sensors and actuators. Note that the notion of shared variables is also used in model decomposition where shared variables belong to different machines and require some synchronisation events for further refinement. Our controller and environment events, however, are located in the same machine, hence, no synchronisation measures are necessary.

Since we schedule only controller events, the environment events in the scheduled Event-B model can update the shared variables at any time. This is reflected by unprovable invariant preservation proof obligations of the scheduled Event-B model. In our example, variable $ml\_out\_10$ is assumed to be shared between the controller and the environment. More specifically, the environment can change the value of $ml\_out\_10$ with its event ML\_OUT\_DEP as follows.

```
ML\_OUT\_DEP:
when
  ML\_OUT\_SR = on
  ml\_tl = green
then
  ML\_OUT\_SR := off
  ml\_out\_10 := TRUE
  A := A + 1
end
```

Proving that ML\_OUT\_DEP maintains invariants like IL\_tl\_green\_Pre will fail, since our model does not prevent the occurrence of ML\_OUT\_DEP between checking the branch condition at $pc = 7$ and executing IL\_tl\_green’s action at $pc = 8$. To remedy the situation, we add a guard, e.g., $pc = 0$ to ML\_OUT\_DEP to prevent the environment event from occurring during the controller’s execution. The meaning of this guard is an assumption on the overall system such that when a car leaves the corresponding sensor, then the controller has finished processing the last message and is ready to process the next message. A simi-
lar assumption regarding the speed of the controller has also been made for the original development in [3].

6.7. Timing

In the “cars on a bridge” example, sensor values represent the state of the environment. The sensors provide real-time information about the environment that the controller can access. The value of such a sensor can be steady or volatile. For example, the state of a motor that is switched on and off exclusively by the controller does not change unless the controller does initiate this change. In this case the motor’s state is steady during controller execution. Different is the situation in the “cars on a bridge” example, where a sensor detects the presence of a car in front of the traffic light. This value changes independently of the controller’s execution. Hence, it is volatile.

The problem with volatile sensor values is, that the controller could miss their changes. If a car arrives at the sensor, the sensor’s value changes from 0 to 1. As soon as the entire car has passed the sensor, its value changes back to 0. If the controller does not read the sensor while its value is 1, the car enters the island without being counted by the controller.

The time that the value of the sensor is 1 depends on the car’s length and speed. Let us assume that the minimum length of a car is 2.5 m (this is the length of a Smart) and that the cars approach the traffic light with a maximum speed of $50 \frac{km}{h}$. Hence, the minimum time that the sensor detects (and indicates) a passing car is

$$t = \frac{s}{v} = \frac{2.5 \, m}{50 \, \frac{km}{h}} = 180 \, ms .$$

If the controller runs at least once every 180 ms it will not miss any passing cars. In this case, the assumption in [3] that the controller is fast enough is valid. Note, that we have to check our assumptions about the cars’ length and speed.

If we cannot guarantee that the controller is fast enough, we must return to the original development and perform further refinements. One possibility without relying on the controller’s speed would be to transfer some controller logic to the environment. In the above example, we could realise the event that turns the traffic light back to red within the environment. When the traffic light is green and a car has passed by (detected by the sensor), the light does automatically change to red without any interaction of the controller. Hence, the controller can take as much time as it wants, there is no other car passing the traffic light before the controller sets it actively to green again.

This behaviour was already installed in mechanical interlocking systems at the beginnings of railway controlling. Before a train was allowed to leave the station, the signalman had to set the signal lever such that the signal showed a green aspect. As soon as the head of the train had passed the signal it automatically fell back to the stop position, without the interaction of the signalmen. With this
it was prevented that another train could follow the correct one by accident due to not setting back the signal lever sufficiently early.

### 6.8. Experience

Before we applied our approach to the train control system, we tested it with a variety of smaller examples from [3]. The different steps of generating code for these examples are summarised in Appendix B. In addition to the “cars on a bridge” example and the small examples just mentioned, we generated code for a more sophisticated case studies: an elevator control system. The elevator control system is a simplification of real elevator systems and was used in a lecture on Event-B at ETH.

In the following table we present some statistics to compare the developments of the elevator system and the “cars on a bridge” example with the train control system.

<table>
<thead>
<tr>
<th></th>
<th>train control system</th>
<th>elevator control system</th>
<th>“cars on a bridge” controller</th>
</tr>
</thead>
<tbody>
<tr>
<td>controller variables</td>
<td>5</td>
<td>2</td>
<td>15</td>
</tr>
<tr>
<td>shared variables</td>
<td>8</td>
<td>12</td>
<td>21</td>
</tr>
<tr>
<td>controller events</td>
<td>8</td>
<td>30</td>
<td>34</td>
</tr>
<tr>
<td>refinement steps</td>
<td>3</td>
<td>3</td>
<td>105</td>
</tr>
<tr>
<td>schedule lines</td>
<td>18</td>
<td>88</td>
<td>90</td>
</tr>
<tr>
<td>invariants encoding the schedule</td>
<td>14</td>
<td>67</td>
<td>69</td>
</tr>
<tr>
<td>events encoding the schedule</td>
<td>18</td>
<td>99</td>
<td>92</td>
</tr>
<tr>
<td>POs for refinement</td>
<td>454</td>
<td>9075</td>
<td>8757</td>
</tr>
<tr>
<td>lines of C code</td>
<td>127</td>
<td>312</td>
<td>373</td>
</tr>
</tbody>
</table>

The numbers in Table 6.15 show that the elevator and train control system are comparable in terms of the size of their schedules and the number of proof obligations required to prove refinement. This may be surprising since the train control system is substantially more complex than the elevator control system and required considerably more effort to develop, which is reflected by the large number of refinement steps. This is because the train control system’s development includes several refinement steps that already account for the later restriction step and translation to code. Hence, both control systems are refined to a level where they are close to their final implementation and the translation to program code becomes straightforward. With the restriction of a model as the first step of our code generation approach, we force the developer to refine the model to a concrete level and therefore keep the translation effort in check. For this reason
our approach works equally well for small academic examples and large, complex industrial systems.

The number of proof obligations generated to prove that the scheduled Event-B model refines the restricted model is rather high. For every invariant/event pair, we have a proof obligation of the form $pc = M \vdash pc = N \Rightarrow \ldots$, where $N$ and $M$ are numbers. However, most of them are trivial as $N \neq M$ holds. The number of relevant proof obligations (i.e. $N = M$) is less than 180 for all three developments and at least 77% are automatically discharged. In fact, our plug-in can generate these relevant proof obligations directly from the schedule rather than having Rodin generating them together with all the irrelevant ones. The remaining task is to prove that this set of proof obligations indeed implies those proof obligations generated to prove the refinement relation. This proof can be done once at the meta-level and is valid for all translations. We present in Chapter 7 how the plug-in generates the necessary proof obligations and provide the meta-proof for refinement.

6.9. Related Work

Merging Rules. Merging rules are introduced in [3] as a mechanism for synthesising sequential programs from Event-B models. There are two rules for creating branches and loops that constitute patterns for developing sequential programs. As a result, the form of the programs is limited and not every program can be synthesised from its Event-B model. For example, a sequential statement is only possible after a loop.

B2C Tool. The B2C tool [43] was developed to generate code for a specific Event-B model of an instruction set architecture. As a result, the plug-in supports only the translation of the Event-B syntax used in this particular model. The most significant shortcoming is that it does not support contexts and therefore cannot be used when constants and sets are used in a machine.

With the B2C tool, there is no possibility to specify a desired execution order of the events. For every event in the model, a C function is created that checks the guards before the actions are executed. In a function named “iterate”, all these event functions are combined in a sequence of function calls equivalent to the event ordering in the Event-B machine. As soon as a function call is successful (the actions were executed) the iterate function returns. In the main function, first the initialisation function is called and then a while loop calls the iterate function as long as there is no deadlock in the system. The disadvantage of leaving the iterate function after a successful action execution of an event is that events at the bottom of the iteration sequence might never be executed.

EB2ALL Tool. EB2ALL [36] is a set of tools for generating code for different target languages. Currently there are four plug-ins included for translations to
C, C++, C#, and Java respectively. The EB2ALL tool is based on the B2C tool. The authors of the tool argue that its correctness is justified by an observable equivalence between the Event-B model and the generated code together with some meta-proofs. It is not specified what notion of observational equivalence is intended and no details on the meta-proof are provided. Furthermore, they state that the generated code usually must be altered manually after generation and that correctness is maintained if the manually added code is also verified in some way. Again, formal details are lacking.

In EB2ALL, the default scheduling is the same as in B2C. The tool provides an optimisation by automatically grouping events that have common guards. This is done by analysing the refinement relation of the events. Two events that both refine the same abstract event have the guards of the abstract event in common. Within the iterate function, these common guards are translated into an if-statement surrounding the function calls of the corresponding events. The intention is that if the guards of the abstract event evaluate to false, then there is no need to check the guards of the refining events. Unfortunately, this approach only works if the guards of the abstract event are all deterministic and translatable. Furthermore, this only produces more efficient code when many events refine one single abstract event. Otherwise, the overhead of additional if-statements may outweigh any efficiency gains.

**Tasking Event-B.** Tasking Event-B [21] is a tool developed for code generation from Event-B models into code with a special focus on concurrent processes. Currently, the tool supports translations to C, Ada and Java. As in EB2ALL, expressions with multiple arithmetic operators are supported. Currently it is not checked whether arithmetic operators maintain the lower and upper bound of the target data type; hence runtime overflows are possible.

Tasking Event-B is the most mature among the existing tools for code generation with respect to scheduling. It is the only tool that provides a scheduling language for user defined scheduling of events. Unfortunately, the language is very restrictive. The bodies of loops and branches are limited to single events. Hence, there is no support for schedules that include structures such as nested branches or a sequence of events within a loop.

**Scheduling Patterns.** We are unaware of any tool support for the scheduling patterns introduced in [15, 16]. The attractiveness of the work is the proof of correctness for the patterns done using set transformers. However, this reasoning must be done manually. Furthermore, our scheduling language is more expressive than the scheduling language defined in [15]. For example, nested branches are not possible in [15].

**Classical B.** Our scheduling language has features similar to (classical) B [1], for example conditional statements, sequential statements and loops. In B, the last
model of a refinement chain is a special construction, the IMPLEMENTATION, from which program code can be generated. Variables of the IMPLEMENTATION must be either concrete variables\textsuperscript{\textit{a}} (i.e., of some implementable data type) or variables of some predefined libraries. Updates of the variables must be well-defined, which is captured by the preconditions of the corresponding assignments. However, loop and branch conditions do not have preconditions to enforce their well-definedness. As a consequence, they are restricted to predicates over simple expressions (e.g., no arithmetic operations are allowed). In our approach, we check all conditions for well-definedness; hence they can contain any expressions. Furthermore, our approach allows us to state assertions between two sequential blocks. As a result, proof obligations can be generated for each block separately. In B, the effects of the sequentially composed statements are combined together, which often results in complicated proof obligations.

6.10. Conclusion

We presented in this chapter our approach to generating program code from Event-B models. Our approach is correct-by-construction and relies on reasoning about well-definedness, assertions, and refinement. Although we presented only the translation to C source code, our approach is also applicable to other languages by adapting the notion of well-definedness and the restriction step to the corresponding target language.

We justify the correctness of the generated source code with help of a scheduled Event-B model that has the equivalent behaviour as the generated code. We base our justification on the observability of this equivalence and provide a formal proof of correctness for the scheduled Event-B model instead. To justify the correctness of the generated code with a formal proof, we would need to elaborate the operational semantics of the Event-B language as well as the used subset of the C programming language and show the equivalence of the scheduled model and the source code with a bisimulation.

In our opinion, the structural similarity provides enough evidence to justify the correctness of the generated source code at this stage. Extending the supported subset of the Event-B language or the scheduling language might change this view. Therefore, we consider a formal proof of the equivalence as future work.
7. Code Generation Plug-in

Our code generation approach consists of different steps described in the former chapter. We developed a plug-in for the Rodin platform to support users in performing these steps. The plug-in provides an editor to define schedules. We extended Rodin’s structural editor that is also used for editing machines and contexts.

The first version of our plug-in generated in addition to the source code also the scheduled machine. This was necessary in the beginning to prove the validity of the schedule. The scheduled machine is thereby defined as a refinement of the original machine. The structure of the scheduled machine is chosen such that the modelled behaviour is identical to the behaviour of the generated code. As a consequence, the generated code does also refine the original machine if the scheduled machine refines the original machine.

What is left to the user is the refinement proof of the scheduled machine. The scheduled machine contains as many invariants as the schedule has lines. To prove refinement, Rodin generates proof obligations in the order of number of invariants times number of events. Hence, the number of proof obligations quickly reach unmanageable dimensions.

We solved the problem of too many proof obligations with a second version of our plug-in. Our investigation showed that the majority of the generated proof obligations include contradicting hypotheses and thus are easily proved. We analysed the origin of these superfluous proof obligations and found a way to identify them. There is no value in generating proof obligations that include contradicting hypotheses. Hence, instead of using Rodin’s proof obligation generator to prove refinement, our plug-in implements a proprietary proof obligation generator that generates only the necessary proof obligations.

The proof obligation generator is based on pre- and postconditions that are assigned to every block within the schedule. All the necessary proof obligations are deducible from these pre- and postconditions together with the schedule’s structure. Therefore, we can generate them right after defining the schedule and before generating code.

We show in this chapter that our set of proof obligations is indeed sufficient to prove the validity of the schedule. In particular, we prove that the set of proof obligations generated by our plug-in implies the set of proof obligations that Rodin would generate to prove that the scheduled machine refines the original one.

We show in the following that both sets of proof obligations are deducible from the schedule’s structure. Hence, we can perform our proof at a meta-level. This results in a meta-proof that holds for our approach rather than for a certain
schedule or model. This completes our justification that our generated code is correct-by-construction provided the user is able to discharge the proof obligation generated by our plug-in.

7.1. Scheduling Editor

Our code generation plug-in comes with an editor for the Rodin platform where a user can enter the different blocks of a schedule together with their relationships and store the entered information in a file. Rodin provides for plug-ins an interface to create editors for new file types. The same interface was used for the structured editor for machines and contexts.

The user enters the different blocks of a schedule and links them according to the desired structure of the schedule. For sequentially composed blocks, the user has the possibility to define assertions. Although these assertions are relevant for both of the composed blocks, we want to specify them only once. We therefore decided for our editor that the assertions between two sequentially composed blocks are always associated with the second block.

For the generation of the invariants in the scheduled machine, we need to calculate the pre- and postconditions of every block. This information is still needed in our optimised version of the plug-in where we generate the relevant proof obligations directly from the schedule instead of generating a scheduled machine. Since these pre- and postconditions can automatically be computed, we do not want to pollute the editor view with this information. Instead, we store only the user-defined information in the editable schedule file and generate a separate file which includes the computed pre- and postconditions.

Following Rodin’s file system for machines and contexts, the editor stores the schedule as an unchecked schedule file. After storing the file, the static checker in Rodin checks the file according to the rules defined in our plug-in. The checker generates a new file: the checked version of the schedule file. It first copies the original schedule including the specified assertions and afterwards generates for all blocks the list of pre- and postconditions according to the algorithm in Section 7.2.

In the further text we use the following notation for the different blocks of a schedule.

7.1.1. Event Block

An event block consists only of the corresponding event’s name and the pre- and postconditions \((pre_1, \ldots, pre_n\) and \(post_1, \ldots, post_m\)).
7.1.2. Branch Block

The “then” part of a branch block is a sequence of blocks \((blk_{t1}, \ldots, blk_{tx})\). The “else” part of a branch block is also a sequence of blocks \((blk_{e1}, \ldots, blk_{ey})\). Between two adjacent blocks, there can be a list of assertions \((asrt_{t2}, \ldots, asrt_{tx} \text{ and } asrt_{e2}, \ldots, asrt_{ey})\). Assertions are associated with the subsequent block, hence, the list of assertions between \(blk_{i-1}\) and \(blk_i\) is named \(asrt_i\). The branch condition located in parentheses after the if keyword is named \(bc\). Furthermore, a branch block consists of a list of pre- and postconditions \((pre_1, \ldots, pre_n \text{ and } post_1, \ldots, post_m)\).

\[
\begin{align*}
\text{pre}_1 \\
\vdots \\
\text{pre}_n \\
\text{event name} \\
\text{post}_1 \\
\vdots \\
\text{post}_m \\
\end{align*}
\]

7.1.3. Loop Block

The “body” of a loop block consists of a sequence of blocks \((blk_{1}, \ldots, blk_{x})\). Again, between two adjacent blocks, there can be a list of assertions \((asrt_{2}, \ldots, asrt_{x})\). The list of assertions between \(blk_{i-1}\) and \(blk_i\) is associated with the subsequent block \(blk_i\) and therefore named \(asrt_i\). The loop condition located in parentheses after the do keyword is named \(lc\). As the other blocks, a loop block consists of a list of pre- and postconditions \((pre_1, \ldots, pre_n \text{ and } post_1, \ldots, post_m)\). In our approach, we use the conjunction of the preconditions as the loop invariant.

\[
\begin{align*}
\text{pre}_1 \\
\vdots \\
\text{pre}_n \\
\text{if}(bc) \\
\quad blk_{t1}; \{asrt_{t2}\} \\
\quad \vdots \\
\quad blk_{tx-1}; \{asrt_{tx}\} \\
\quad blk_{tx} \\
\text{else} \\
\quad blk_{e1}; \{asrt_{e2}\} \\
\quad \vdots \\
\quad blk_{ey-1}; \{asrt_{ey}\} \\
\quad blk_{ey} \\
\text{fi} \\
\text{post}_1 \\
\vdots \\
\text{post}_m \\
\end{align*}
\]
7.1.4. Schedule Body

The body of a schedule is a sequence of blocks \((blk_1, \ldots, blk_x)\). Between two adjacent blocks, there can be assertions \((asrt_2, \ldots, asrt_x)\). The list of assertions between \(blk_{i-1}\) and \(blk_i\) is associated with the subsequent block \(blk_i\) and named \(asrt_i\). In contrast to blocks, the schedule body has no pre- or postconditions.

7.1.5. Further Notations

We abbreviate the list of all preconditions of a block \((pre_1, \ldots, pre_n)\) as \(pre\). Similarly, we abbreviate the list of all postconditions of a block \((post_1, \ldots, post_m)\) as \(post\). If it is not clear from the context, we can write \(blk_{i/pre}\) to refer to all preconditions of block \(i\) and \(blk_{j/pre_k}\) to refer to the \(k\)-th precondition of block \(j\). The same applies to postconditions.

7.2. Automatic Computation of Pre- and Postconditions

To ease the task of the user when writing the schedule for a model, we developed an algorithm that automatically generates the list of pre- and postconditions for the schedule blocks. With this, the user has only to provide explicit assertions. Such explicit assertions might be necessary for blocks that are sequentially composed. In such situations the algorithm cannot easily compute the guaranteed predicates at this position.

We describe our algorithm in an object-oriented style where the blocks are objects of a certain class corresponding to the block type, i.e. branch block, loop block, event block, and schedule body. We define for every class the two methods \(pre\) and \(post\) to compute the pre- and postconditions, respectively. While the
methods for the block types have a parameter to hand over a list of pre- or postconditions, the schedule body does not accept any argument.

7.2.1. Definitions

- **cond**: a list of conditions
- **[]**: the empty list
- **cond + c**: the list of conditions containing the elements of \(\text{cond}\) plus condition \(c\)
- **!c**: the negation of condition \(c\)
- **m.pre(cond)**: calling the method \(\text{pre}\) of block (object) \(m\) and passing a list of conditions \(\text{cond}\) as argument
- **pre**: list of preconditions of a block
- **post**: list of postconditions of a block

7.2.2. Algorithm Overview

The static checker for schedule files executes the following algorithm consisting of two phases. In each phase, the algorithm traverses through the abstract syntax tree (AST) of the defined schedule on a depth-first path. Event blocks and empty bodies are the terminals of our scheduling language. Hence, they become the leaves in the AST where the paths end and our algorithm uses backtracking to continue traversing.

**Phase 1**: We compute the list of preconditions for every block in the checked file given the optional, explicitly specified assertions in the unchecked file. We start with the schedule body (i.e. the root of the AST) by calling its \(\text{pre}\) method and traverse through the AST in a forward direction. This means, at a node where two sequences are sequentially composed, we first enter the left subtree and afterwards the right subtree.

**Phase 2**: We compute the postconditions for every block based on the checked file where the before computed preconditions are already present. We start again with the schedule body by calling its \(\text{post}\) method. This time, however, we traverse backwards. This means, at a node where two sequences are sequentially composed, we first enter the right subtree and afterwards the left subtree.

7.2.3. Phase 1 – Computing Preconditions

In the following we describe the method \(\text{pre}\) for every class.

**Schedule Body**

The \(\text{pre}\) method of the schedule body does not accept any argument as input. It calls the \(\text{pre}\) method of the first block in its body and passes an empty list of
preconditions as argument, meaning no preconditions are guaranteed at this point. After returning from the called method it calls the remaining compositionally combined blocks in its body. This time the user-defined assertions are passed as argument for the method’s parameter. After returning from the last block’s pre method, the method itself returns to its caller.

```plaintext
pre() {
    blk1.pre([]);
    for i ∈ 2 .. x {
        blk_i.pre(asrt_i);
    }
}
```

**Event Block**

The event block’s pre method assigns the preconditions received as parameter to the block’s list of preconditions and returns afterward.

```plaintext
pre(cond) {
    pre = cond;
}
```

**Loop Block**

The loop block’s pre method assigns the preconditions received as parameter to the block’s list of preconditions. Afterwards it calls the pre method of the first block inside the loop body and hands over the received list of precondition extended with the loop condition lc as an additional list entry. After returning from the called method it calls the pre method of the remaining compositionally combined blocks in its body. This time the user-defined assertions are passed as argument for the method’s parameter. After returning from the last block’s pre method, the method itself returns to its caller.

```plaintext
pre(cond) {
    pre = cond;
    blk1.pre(pre + lc);
    for i ∈ 2 .. x {
        blk_i.pre(asrt_i);
    }
}
```

**Branch Block**

The branch block’s pre method assigns the preconditions received as parameter to the block’s list of preconditions. Afterwards it calls the pre method of the first block inside the “then” part and hands over the received list of precondition extended with the branch condition bc as an additional list entry. After returning from the called method it calls the remaining compositionally combined blocks
in the “then” part. For these blocks, the user-defined assertions are passed as argument for the method’s parameter.

Afterwards the method calls the \texttt{pre} method of the first block inside the “else” part. This time it hands over the received list of precondition extended with the negated branch condition $!bc$ as an additional list entry. For the remaining blocks in the “else” part it proceeds as described for the “then” part. After returning from the last block’s \texttt{pre} method, the method itself returns to its caller.

\begin{verbatim}
pre(cond) {
    pre = cond;
    blk_{t1}.pre(pre + bc);
    for i \in 2..x {
        blk_{t_i}.pre(asrt_{t_i});
    }
    blk_{e1}.pre(pre + !bc);
    for j \in 2..y {
        blk_{e_j}.pre(asrt_{e_j});
    }
}
\end{verbatim}

\subsection*{7.2.4. Phase 2 – Computing Postconditions}

In the following we describe the method \texttt{post} for every class.

\textbf{Schedule Body}

The \texttt{post} method of the schedule body does not accept any argument as input. It calls the \texttt{post} method of the last block in its body and passes an empty list of postconditions as argument, meaning no postconditions need to be established by this block. After returning from the called method, it calls the remaining compositionally combined blocks in its body. This time the list of preconditions of the subsequent block (which are equivalent to the user-defined assertions) are passed as argument for the method’s parameter. After returning from the penultimate block’s \texttt{post} method, the method itself returns to its caller.

\begin{verbatim}
post() {
    blk_{x}.post([]);
    for i \in 1..x - 1 {
        blk_{i}.post(blk_{i+1}/pre);
    }
}
\end{verbatim}

\textbf{Event Block}

The event block’s \texttt{post} method assigns the postconditions received as parameter to the block’s list of postconditions and returns afterward.
post(cond) {
  post = cond;
}

Loop Block

The loop block’s post method assigns the postconditions received as parameter to the block’s list of postconditions. Afterwards it calls the post method of the last block in the loop body and hands over its list of precondition (i.e. the loop invariant). After returning from the called method it calls the post method of the remaining compositionally combined blocks in its body. In these calls the list of preconditions of the subsequent block are passed as argument for the method’s parameter. After returning from the penultimate block’s post method, the method itself returns to its caller.

post(cond) {
  post = cond;
  blk_x.post(pre);
  for i ∈ 1..x – 1 {
    blk_i.post(blk_{i+1}/pre);
  }
}

Branch Block

The branch block’s post method assigns the postconditions received as parameter to the block’s list of postconditions. Afterwards it calls the post method of the last block inside the “then” part and hands over the same list of precondition as it received. After returning from the called method it calls the remaining compositionally combined blocks in the “then” part. For these blocks the list of preconditions of the subsequent blocks are passed as argument for the method’s parameter. Afterwards the method calls the post method of the last block inside the “else” part. Again it hands over the received list of postconditions. For the remaining blocks in the “else” part it proceeds as described for the “then” part. After returning from the penultimate block’s post method, the method itself returns to its caller.
7.3. Proof Obligation Generator for Schedule

The following pages describe the proof obligations (PO) that must be generated to guarantee that the chosen schedule is indeed a valid one. Or more precisely, that the control flow statements defined by the schedule guarantee the guards of the events at their particular positions. For the representation of the POs as well as for the meta-proof we use sequent calculus as described in [3].

Although the set of necessary POs has been identified by analysing the refinement POs related to the scheduled machine, it is interesting to see that these POs can also be inferred using Hoare logic [31]. We therefore show for every block the relationship between the generated POs and Hoare logic with help of a derivation tree.

7.3.1. Definitions

- \( L \) well-definedness (WD)-operator translating an expression into the corresponding WD predicate
- \( s \) list of sets seen by the restricted model
- \( c \) list of constants seen by the restricted model
- \( A \) list of all axioms seen by the restricted model
- \( I \) list of all invariants seen by the restricted model
- \( v \) list of all variables in the restricted model
- \( v' \) values of variables \( v \) after an assignment
- \( g \) \( i \)th guard of an abstract event
- \( BAP \) before-after predicate of an event for all variables \( v \)

7.3.2. Proof Obligations for Well Definedness

For every user-defined assertion, loop condition or branch condition in the schedule, a well-definedness (WD) proof obligation (PO) must be generated. To generate these proof obligations, we use the WD-operator \( L \), which is implemented in the Rodin platform as a simplified version of the WD-operator defined in [35].
As for generating WD-POs for the guards of an event, the ordering of different assertions within a list has an effect on the generated PO as the previously stated assertions appear as hypotheses in the PO. We defined that assertions are associated with the subsequent block. Hence, as a logical consequence, we also associate the resulting WD-POs with the subsequent block.

For every assertion \( a_i \) within a list of assertions associated with block \( M (\text{asrt}_m = \{a_1 \ldots a_n\}) \), the following WD-PO is generated.

\[
\text{M/ai/WD (well-definedness for assertion)} \\
A, I, a_1, a_2, \ldots, a_{i-1}, \mathcal{L}(a_1), \ldots, \mathcal{L}(a_{i-1}) \vdash \mathcal{L}(a_i)
\]

For the loop condition \( lc \) of a loop block \( L \) with a list of preconditions \((pre_1, \ldots, pre_n)\), the following WD-PO is generated.

\[
\text{L/lc/WD (well-definedness of loop condition)} \\
A, I, pre_1, pre_2, \ldots, pre_n \vdash \mathcal{L}(lc)
\]

For the branch condition \( bc \) of a branch block \( B \) with a list of preconditions \((pre_1 \ldots pre_n)\), the following WD-PO is generated.

\[
\text{B/bc/WD (well-definedness of branch condition)} \\
A, I, pre_1, pre_2, \ldots, pre_n \vdash \mathcal{L}(bc)
\]

For every parameter \( p_i \) in an event referred to by an event block \( E \) with the parameter initialisation \( \text{pinit}_i \) of the form \( p_i = E(s, c, v, \text{par}_1, \ldots, \text{par}_{i-1}) \) and a list of preconditions \((pre_1 \ldots pre_n)\), the following WD-PO is generated.

\[
\text{E/pi/WD (well-definedness of parameter initialisation)} \\
A, I, pre_1, pre_2, \ldots, pre_n, \text{pinit}_1, \ldots, \text{pinit}_{i-1} \vdash \mathcal{L}(\text{pinit}_i)
\]

### 7.3.3. Proof Obligations for Loop Blocks

To guarantee the postconditions of a loop block \( L \), the following PO is generated.

\[
\text{L/POST (establishing postconditions of L when exit)} \\
pre_1, \ldots, pre_n, \neg lc \vdash post_1 \land \cdots \land post_m
\]

There is no need to check that the preconditions of the first block within the loop body are established when the loop condition is true. This fact is guaranteed by our precondition generation algorithm. The preconditions of the first inner block are defined to be the preconditions of the loop block conjoined with the loop condition.
The following derivation tree shows that the same proof obligation is generated using Hoare logic.

\[
\begin{align*}
\{\text{pre} \land \text{lc}\} \text{ Body } \{\text{pre}\} \\
\{\text{pre}\} \text{ do(\text{lc}) Body od } \{\text{pre} \land \neg \text{lc}\} \quad \text{pre} \land \neg \text{lc} \Rightarrow \text{post} \\
\{\text{pre}\} \text{ do(\text{lc}) Body od } \{\text{post}\}
\end{align*}
\]

The body of a loop block is either a sequence of blocks, a single block or empty. Although an empty loop body is not very useful, it is still correct without any additional PO as the following derivation tree shows.

\[
\begin{align*}
\text{pre} \land \text{lc} \Rightarrow \text{pre} & \quad \{\text{pre}\} \text{ skip } \{\text{pre}\} \\
\{\text{pre} \land \text{lc}\} \text{ skip } \{\text{pre}\} & \quad \{\text{pre}\} \text{ do(\text{lc}) Body od } \{\text{pre}\} \\
\{\text{pre}\} \text{ do(\text{lc}) Body od } \{\text{pre} \land \neg \text{lc}\} & \quad \text{pre} \land \neg \text{lc} \Rightarrow \text{post} \\
\{\text{pre}\} \text{ do(\text{lc}) Body od } \{\text{post}\}
\end{align*}
\]

In case of a single block, the proof obligation for the remaining term is generated by the block itself. Note that the pre- and postconditions of the term correspond exactly to those of the block as generated by our algorithm. If the loop body is a sequence, the remaining term can be further deduced as follows.

\[
\begin{align*}
\{\text{pre} \land \text{lc}\} \text{ S } \{\text{R}\} & \quad \{\text{R}\} \text{ T } \{\text{pre}\} \\
\{\text{pre} \land \text{lc}\} \text{ S;T } \{\text{pre}\} & \quad \{\text{pre}\} \text{ do(\text{lc}) Body od } \{\text{pre}\} \\
\{\text{pre}\} \text{ do(\text{lc}) Body od } \{\text{pre} \land \neg \text{lc}\} & \quad \text{pre} \land \neg \text{lc} \Rightarrow \text{post} \\
\{\text{pre}\} \text{ do(\text{lc}) Body od } \{\text{post}\}
\end{align*}
\]

Note that the midcondition \( R \) corresponds to the user-defined assertions \( \text{asrt}_T \) between S and T. The sequences S and T are further expanded which results in the following terms to prove, each of them corresponding to one of the sequentially composed blocks in the loop body. Again, the proof obligations are generated by these blocks themselves and the pre- and postconditions of the terms correspond to the ones generated by our algorithm.

\[
\begin{align*}
\{\text{pre} \land \text{lc}\} \text{ blk}_1 \{\text{asrt}_2\} & \quad \{\text{asrt}_2\} \text{ blk}_2 \{\text{asrt}_3\} \quad \ldots \quad \{\text{asrt}_x\} \text{ blk}_x \{\text{pre}\} \\
\vdots
\{\text{pre} \land \text{lc}\} \text{ S } \{\text{R}\} & \quad \{\text{R}\} \text{ T } \{\text{pre}\}
\end{align*}
\]

**Loop Termination.** We do not generate any proof obligations regarding loop termination. This is not necessary to prove refinement. A loop which runs forever does not affect the safety properties of a model. However, it might be useful in terms of liveness to show that a loop terminates. This liveness property can be proven in the Event-B model using a variant. Marking an event as convergent provokes the generation of additional proof obligations to prove that the event
decreases the variant with every execution. To prove loop termination, we have to mark all events within the loop’s body as convergent with respect to the same variant. If the loop condition is chosen such that it becomes false as the variant goes below some limit, the loop will terminate.

7.3.4. Proof Obligations for Branch Blocks

To guarantee the postconditions of a branch block B if it has an empty “then” part, the following PO is generated.

\[ B/\text{then}/\text{POST} \] (establishing postconditions of B)
\[ pre_1, \ldots, pre_n, bc \vdash post_1 \land \cdots \land post_m \]

To guarantee the postconditions of the branch block B if it has an empty “else” part, the following PO is generated.

\[ B/\text{else}/\text{POST} \] (establishing postconditions of B)
\[ pre_1, \ldots, pre_n, \neg bc \vdash post_1 \land \cdots \land post_m \]

The justification using Hoare logic is given by the following derivation tree.

\[
\begin{array}{c}
\{pre \land bc\} \text{Then} \{post\} & \{pre \land \neg bc\} \text{Else} \{post\} \\
\{post\} & \{post\} \\
\{pre \land bc\} \text{Skip} \{post\} & \{pre \land \neg bc\} \text{Skip} \{post\}
\end{array}
\]

Again, the two bodies (i.e. Then and Else) can be empty, a single block or a sequence. In case of a single block or a sequence, the derivation tree is extended in the same way as for the loop block above. If a body is empty, the PO that we generate is also inferable from the derivation tree. For simplicity, we show the derivation tree where both bodies are empty at the same time.

\[
\begin{array}{c}
\{pre \land bc\} \Rightarrow \text{post} & \{post\} \text{Skip} \{post\} \\
\{pre \land \neg bc\} \Rightarrow \text{post} & \{post\} \text{Skip} \{post\}
\end{array}
\]

\[
\begin{array}{c}
\{pre \land bc\} \text{Skip} \{post\} & \{pre \land \neg bc\} \text{Skip} \{post\} \\
\{pre \land bc\} \text{Then} \{post\} & \{pre \land \neg bc\} \text{Else} \{post\}
\end{array}
\]

\[
\{post\} \]

\[
\{pre \land bc\} \text{If}(bc) \text{Then else Else fi} \{post\}
\]

There is no difference in terms of PO generation for the short form of a branch block. After all, this is just a regular branch block where Then is empty.

7.3.5. Proof Obligation for Event Blocks

To prove that the preconditions of the event block E guarantee the guards \((grd_1, \ldots, grd_z)\) of the associated event \(evt\), the following PO is generated.

\[ E/\text{GRD} \] (establishing guards of \(evt\))
\[ pre_1, \ldots, pre_n, pinit_1, \ldots, pinit_p \vdash grd_1 \land \cdots \land grd_z \]
To guarantee the postconditions of an event block $E$, the following PO is generated. The primed postcondition $post'_i$ is a copy of the unprimed postcondition $post_i$ where all appearing variables have been replaced with their primed version.

$$E/POST\text{(establishing postconditions of E after executing the referred event)}$$

$$pre_1, \ldots, pre_n, pinit_1, \ldots, pinit_p, BAP \vdash post'_1 \land \cdots \land post'_m$$

In our translation of events, we do not translate the events’ guards except for parameter initialisations. Instead, we require that the control flow statements guarantee the guards of an event before its actions are executed. This property is not easy to express with standard Hoare logic. Hence, we do not present a derivation tree for the two PO’s above.

### 7.4. Generated Scheduled Machine

Before we provide the meta-proof that shows that the generated POs imply the refinement relation between scheduled machine and restricted machine we recall the different components in the generated machine.

#### 7.4.1. Seen Context

The scheduled machine sees the same context as the restricted machine. Hence, the seen sets $s$, constants $c$, and axioms $A$ remain the same.

#### 7.4.2. Variables

All variables of the restricted machine $v$ are also variables of the scheduled machine. The scheduled machine has one additional variable $pc$ that is used to describe the schedule’s program flow in form of a program counter.

#### 7.4.3. Invariants

The invariants of a scheduled machine represent the pre- and postconditions that hold at certain positions within the sequential program. Every block of the schedule has pre- and postconditions associated that where automatically computed by our algorithm. Furthermore, we assign line numbers to the schedule and use them together with the program counter $pc$ to argue about the program flow. Every block has one or more positions that correspond to a line number in the schedule. For every line number we generate exactly one invariant as described by the following sections.
Event Block

For every event block E at position s within the schedule, there exists an invariant of the following form.

\[ pc = s \Rightarrow E/pre_1 \land \cdots \land E/pre_n \]

Loop Block

For every loop block L that starts at position s and ends at position e, the following two invariants exist.

\[ pc = s \Rightarrow L/pre_1 \land \cdots \land L/pre_n \]
\[ pc = e \Rightarrow L/pre_1 \land \cdots \land L/pre_n \]

Branch Block

For every branch block B that starts at position s and ends at position e, the following two invariants exist.

\[ pc = s \Rightarrow B/pre_1 \land \cdots \land B/pre_n \]
\[ pc = e \Rightarrow B/post_1 \land \cdots \land B/post_m \]

If the branch block B is written in the long form, there exists a third invariant related to the block’s middle position m.

\[ pc = m \Rightarrow B/post_1 \land \cdots \land B/post_m \]

7.4.4. Program Counter

For the typing of the program counter pc we generate the following invariant.

\[ pc \in \mathbb{C}_{INT} \]

In the following, we make the reasonable assumption that the scheduler has less than \( MAX\_INT \) (2147483647) lines. Hence, every position in the scheduler (0...e) is associated with a natural number that is strictly less than \( MAX\_INT \). The events of the scheduled machine update the variable pc in two different ways. They assign a new number to pc, either a position within the schedule (0...e) or the position right after the schedule (e+1),

\[ pc := x, \text{ where } x \in 0..e + 1 \]

or they increment pc by one.

\[ pc := pc + 1 \]

In both cases, pc remains in the range defined by \( C\_INT \). When pc reaches the position e+1, the program flow defined by the schedule ends.
7.4.5. Events

In the following sections we describe the different events that we generate for the scheduled machine with respect to the blocks defined in the schedule. As a special case we first have a closer look at the initialisation event that is not part of the schedule.

Initialisation

The initialisation event includes the same actions to initialise the variables as the abstract initialisation in the restricted machine. In addition to this, it reflects the setup of the schedule by initialising the new variable $pc$ to 0.

\[
\text{init : begin}
\begin{align*}
  &\text{act}_1 \\
  &\text{act}_2 \\
  &\text{...} \\
  &\text{pc := 0}
\end{align*}
\text{end}
\]

Event Block

For every event block in the schedule one event is generated that refines the event in the restricted machine to which the event block refers. The initialisation of the parameters as well as the actions $act_1 \ldots act_x$ are copied from the abstract event. Additionally, $pc$ is incremented by 1.

\[
\text{evt : refines evt}
\begin{align*}
  \text{any } &\text{p}_1, \ldots, \text{p}_p \text{ where} \\
  &\text{pc = s} \\
  &\text{pinit}_1 \\
  &\text{...} \\
  &\text{pinit}_p \\
  \text{then} \\
  &\text{act}_1 \\
  &\text{...} \\
  &\text{act}_x \\
  &\text{pc := pc + 1}
\end{align*}
\text{end}
\]

Branch Block

For every branch block four events are generated if it contains an “else” part. For the short form, without an “else” part, only three events are generated.
if \_true :  
when  
  pc = s  
  bc  
then  
  pc := pc + 1  
end

if \_false (long form) :  
when  
  pc = s  
  \neg bc  
then  
  pc := m + 1  
end

if \_false (short form) :  
when  
  pc = s  
  \neg bc  
then  
  pc := e  
end

if \_exit (only long form) :  
when  
  pc = m  
then  
  pc := e  
end

if \_end :  
when  
  pc = e  
then  
  pc := pc + 1  
end

7.5. Meta-proof

To guarantee that the generated scheduled machine is a valid refinement of the restricted machine, Event-B generates proof obligations (POs). Instead of proving the large number of POs – for the train controller there are almost 9000 POs – we generate only a small set of POs (described in Section 7.3). With the following meta-proof we show that these POs are sufficient to prove refinement between scheduled and restricted machine by induction over the schedule’s structure.

In Section 6.6, we defined for every block type the events and invariants that are generated in the scheduled machine. With this information, we can derive the refinement proof obligations that are generated related to this events and invariants. Hence, we know what refinement proof obligations need to be proved for every block type. Furthermore, we defined in Section 7.3 for every block type the POs that we generate and the user must prove. In the following meta-proof, we proof for every block type that the associated POs generated by our tool imply the refinement POs that associated to this block type. With other words, our meta-proof shows that if the user proves the POs generated by our tool, he would
also be able to prove the refinement POs and therefore verified the correctness of
the generated code.

Since the goal of our meta-proof is to show that all refinement POs are guaran-
teed, we structure it according to the different types of proof obligations associated
with refinement. For every PO type, we analyse the refinement POs to prove and
show that they are guaranteed by the POs generated by our plug-in.

7.5.1. Definitions

We use the following abbreviations in addition to the already specified ones.

- \( J \) invariants of the generated scheduled machine
- \( J_b \) invariants of the generated scheduled machine declared so far
- \( H \) guards of the event referred to by an event block

7.5.2. Well-definedness of Invariants (WD)

Every invariant is checked whether it is well-defined by generating a WD-PO

\[ A, I, J_b \vdash L(pc = x \Rightarrow cond_1 \land cond_2 \land \cdots \land cond_n) \]

where \( x \) is a natural number and \( cond_1, cond_2, \ldots, cond_n \) are either the pre- or
postconditions that hold at this position \( x \).

The above PO can be decomposed into several simpler POs.

\[
\begin{align*}
A, I, J_b &\vdash L(pc = x) \\
A, I, J_b &\vdash pc = x \Rightarrow L(cond_1) \\
A, I, J_b &\vdash pc = x \land cond_1 \Rightarrow L(cond_2) \\
\cdots \\
A, I, J_b &\vdash pc = x \land cond_1 \land \cdots \land cond_{n-1} \Rightarrow L(cond_n)
\end{align*}
\]

PO (1) is easily proved, as \( L(pc = x) \) equals \( \top \) for any \( x \). According to the
algorithm for computing the pre- and postconditions of the blocks we know the
following.

In case \( cond_1, cond_2, \ldots, cond_n \) represent the postconditions of a block, they
are equivalent to a (possibly empty) list of assertions \( a_1, a_2, \ldots, a_n \). Hence, the
above POs are directly discharged using the well-definedness POs for assertions
generated for the schedule file (\( M/ai/WD \)).

If the conditions \( cond_1, cond_2, \ldots, cond_n \) represent the preconditions of a block,
they are equivalent to either

1. a (possibly empty) list of assertions \( a_1, a_2, \ldots, a_n \),
2. the preconditions of the outer block plus a branch condition \( bc \),
3. the preconditions of the outer block plus a negated branch condition \( \neg bc \), or
4. the preconditions of the outer block plus a loop condition \( lc \).
**Case 1.** The above POs are directly discharged using the well-definedness POs for assertions generated for the schedule file (M/ai/WD).

**Case 2.** The POs (2), (3), etc. are identical with the POs for the preconditions of the outer block. The remaining PO (4) is directly discharged using the well-definedness PO for branch blocks generated for the schedule file (B/bc/WD).

**Case 3.** This case is similar to case 2 and can be proved with help of the following property of the well-definedness operator $L$.

$$L(\text{predicate}) \equiv L(\neg \text{predicate})$$

**Case 4.** The POs (2), (3), etc. are identical with the POs for the preconditions of the outer block. PO (4) is directly discharged using the well-definedness PO for loop blocks generated for the schedule file (L/lc/WD).

### 7.5.3. Well-definedness of Guards and Actions (WD)

To guarantee the well-definedness of the guards and actions of the generated events in the scheduled machine, we analyse the different events corresponding to the scheduling blocks.

**Event Block.** The actions of the generated event $(\text{act}_1, \text{act}_2, \ldots)$ are identical with the actions of the abstract event in the restricted machine. Since the well-definedness of these actions is already proved with respect to the abstract guards and since the concrete guards imply the abstract ones (guard strengthening), there are no POs generated for these actions. The well-definedness of the additional action at the end of the event $pc := pc + 1$ is trivially proved.

The only guards that are generated are the ones that define the initialisation of the event’s parameters and the guard that checks whether the current program counter is equal to the block’s starting position $pc = s$. For these guards, we have to prove the following POs.

$$A, I, J \vdash L(pc = s) \quad (5)$$

$$A, I, J, pc = s \vdash L(pinit_1) \quad (6)$$

$$A, I, J, pc = s \land pinit_1 \vdash L(pinit_2) \quad (7)$$

$$\ldots$$

PO (5) is easily proved, as $L(pc = s)$ equals $\top$ for any $s$. To prove PO (6), (7) and the following we use the POs generated for the schedule (E/pi/WD)

$$A, I, pre_1, pre_2, \ldots, pre_n \vdash L(pinit_1)$$

$$A, I, pre_1, pre_2, \ldots, pre_n, pinit_1 \vdash L(pinit_2)$$
together with the following invariant that belongs to $J$ and guarantees that the block’s preconditions hold whenever the current program counter is equal to the block’s starting position $pc = s$.

\[ pc = s \Rightarrow pre_1 \land pre_2 \land \cdots \land pre_n \]

**Branch Block.** The well-definedness POs for the guards $pc = s$, $pc = m$, and $pc = e$ as well as for the actions of the different events are trivially proved. The remaining WD-POs for the branch condition and its negation are as follows.

\[
\begin{align*}
A, I, J, pc = s & \vdash L(bc) \\
A, I, J, pc = s & \vdash L(\neg bc)
\end{align*}
\]

PO (8) is proved with help of the invariant associated with the block’s start position (included in $J$)

\[ pc = s \Rightarrow pre_1 \land pre_2 \land \cdots \land pre_n \]

together with the following WD-PO generated for the schedule file ($B/bc/WD$).

\[ A, I, pre_1, pre_2, \ldots, pre_n \vdash L(bc) \]

PO (9) is proved with help of PO (8) and the following property of the WD operator.

\[ L(predicate) \equiv L(\neg predicate) \]

**Loop Block.** The well-definedness POs for the guards $pc = s$ and $pc = e$ as well as for the actions of the different events are trivially proved. The WD-POs for the loop condition and its negation are as follows.

\[
\begin{align*}
A, I, J, pc = s & \vdash L(lc) \\
A, I, J, pc = s & \vdash L(\neg lc)
\end{align*}
\]

PO (10) is proved with help of the invariant associated with the block’s start position (included in $J$)

\[ pc = s \Rightarrow pre_1 \land pre_2 \land \cdots \land pre_n \]

together with the following WD-PO generated for the schedule file ($L/lc/WD$).

\[ A, I, pre_1, pre_2, \ldots, pre_n \vdash L(lc) \]

PO (11) is proved with help of PO (10) and the following property of the WD operator.

\[ L(predicate) \equiv L(\neg predicate) \]
7.5.4. Invariant Establishment (INV)

We have to prove that all invariants are established by the initialisation of the machine. The initialisation event sets \( pc \) to 0. Hence, all invariants for positions different than 0 are trivially proved. The only invariant that we have to check, is the one for \( (pc = 0) \) which corresponds to the preconditions of the first block in the schedule body. According to our algorithm the first block in the schedule body receives an empty list of preconditions, hence we have the invariant \( pc = 0 \Rightarrow \top \). Therefore, the PO looks as follows and is trivially proved.

\[
A \vdash 0 = 0 \Rightarrow \top
\]

7.5.5. Invariant Preservation (INV)

For every combination of invariant \( (pc = z \Rightarrow cond_1 \land \cdots \land cond_n) \) and event (with guards \( H \) and before-after-predicate \( BAP \)) an INV-PO is generated for checking the preservation of the invariant after executing the event’s actions. Note the primed conditions that include the primed versions of the variables.

\[
A, J, H, BAP \vdash pc' = z \Rightarrow cond'_1 \land \cdots \land cond'_n
\]

Every event in the scheduled model has a guard that specifies which position in the schedule it represents \( (pc = x) \) and an action that updates \( pc \) to a new value \( y \). Most of the generated POs are trivially proved, namely, when \( z \) is different from \( y \). In fact, there remains only one interesting PO per event that is not trivially discharged since there is exactly one invariant for every position \( y \) in the schedule.

In order to prove the POs at the meta-level, we must know for every event how the invariant for position \( y \) looks like. More precisely, we must know what conditions \( (cond_1, \ldots, cond_n) \) are stated on the right-hand side of the implication within the invariant.

\[
pc = y \Rightarrow cond_1 \land \cdots \land cond_n
\]

When we analyse the events and in particular the update of \( pc \), we see that there are three different kinds of positions that \( y \) can stand for.

1. The position \( y \) points to the beginning of an inner block. This is the case for the events \( \text{if}_\text{true} \) (if the “then” part is not empty), \( \text{if}_\text{false} \) (long form), and \( \text{do}_\text{true} \).

2. The position \( y \) points to a keyword within the block. This is the case for the events \( \text{if}_\text{false} \) (short form), \( \text{if}_\text{exit} \), \( \text{do}_\text{return} \), and \( \text{if}_\text{true} \) (if the “then” part is empty).

3. The position \( y \) points to the next position after the end of the block. This is the case for the events \( \text{evt} \), \( \text{if}_\text{end} \), and \( \text{do}_\text{false} \).

We continue with an analysis of the invariant associated with the position \( y \) for the different cases.
Case 1 – Beginning of an inner block  The beginning of a block at position \( s \) is always associated with the block’s preconditions. From the algorithm to compute the preconditions we know that the preconditions of the first inner block \( I \) at position \( y \) are the preconditions of the outer block \( A \) plus a loop or (possibly negated) branch condition. Hence, the invariants for \( y \) in the different events are as follows.

\[
\begin{align*}
\text{event} & \quad \text{invariant} \\
\text{if\_true (with "then" part)} & \quad pc = y \Rightarrow A/pre_1 \land \cdots \land A/pre_n \land bc \\
\text{if\_false (long form)} & \quad pc = y \Rightarrow A/pre_1 \land \cdots \land A/pre_n \land \neg bc \\
\text{do\_true (non-empty body)} & \quad pc = y \Rightarrow A/pre_1 \land \cdots \land A/pre_n \land lc
\end{align*}
\]

Case 2 – Keyword of block  All keywords of the different blocks are associated with a position in the schedule and therefore have a corresponding invariant. In the case of a branch block with an empty “then” part the position \( y \) refers to the else or the fi keyword. In both cases the invariant states the postconditions of the block. If a loop block has an empty body, the position \( y \) points to the od keyword where the preconditions of the block must hold. The invariants for \( y \) in the different events are as follows.

\[
\begin{align*}
\text{event} & \quad \text{invariant} \\
\text{if\_false (short form)} & \quad pc = y \Rightarrow post_1 \land \cdots \land post_m \\
\text{if\_exit} & \quad pc = y \Rightarrow post_1 \land \cdots \land post_m \\
\text{do\_return} & \quad pc = y \Rightarrow pre_1 \land \cdots \land pre_n \\
\text{if\_true (without "then" part)} & \quad pc = y \Rightarrow post_1 \land \cdots \land post_m \\
\text{do\_true (empty body)} & \quad pc = y \Rightarrow pre_1 \land \cdots \land pre_n
\end{align*}
\]

Case 3 – Next position after the end  This case is a bit more difficult to analyse since we have to consider the surrounding of a block \( A \). To determine the invariant associated with the next position after the end of the block, we consider all possible locations of block \( A \) within a schedule.

- Block \( A \) is sequentially composed with another block \( C \). Therefore, \( y \) is the start position of block \( C \) and the invariant for position \( y \) states \( C \)’s preconditions. Since \( A \) and \( C \) are sequentially composed, \( C \)’s preconditions are equivalent to the assertions connected to this composition. Furthermore, following the algorithm for computing the postconditions, these assertions are also equivalent to \( A \)’s postconditions. Hence the invariant at position \( y \) states \( A \)’s postconditions.

- Block \( A \) is located at the end of the “then” part of a branch block \( B \). Hence, the position \( y \) points to the else keyword (if there is an “else” part) or to the fi keyword (if there is no “else” part) of block \( B \). In both cases the invariant for \( y \) states the postconditions of \( B \). According to the algorithm, these postconditions become also the postconditions of block \( A \). Hence, the invariant at position \( y \) states \( A \)’s postconditions.
• Block $A$ is located at the end of the “else” part of a branch block $B$. Hence, the position $y$ points to the $\text{fi}$ keyword of block $B$. Following the same argumentation as above, we know that the invariant at position $y$ states $A$’s postconditions.

• Block $A$ is located at the end of a loop block $L$. Hence, the position $y$ points to the $\text{od}$ keyword of block $L$. The invariant for $y$ states therefore the preconditions of block $L$. According to the algorithm, the postconditions of the last block within the loop body are set to the loop block’s preconditions. Hence, the invariant at position $y$ states $A$’s postconditions.

• Block $A$ is the last item within the schedule body. Hence, there is no invariant for position $y$ and, as a consequent, no PO to prove.

We conclude that, if there exists an invariant for position $y$, it states the block’s postconditions. Hence, the invariants for $y$ in the different events are as follows.

\[
\begin{array}{ll}
\text{event} & \text{invariant} \\
\text{evt} & pc = y \Rightarrow post_1 \land \cdots \land post_m \\
\text{if} \_\text{end} & pc = y \Rightarrow post_1 \land \cdots \land post_m \\
\text{do} \_\text{false} & pc = y \Rightarrow post_1 \land \cdots \land post_m \\
\end{array}
\]

Having analysed the relevant invariants, we return to the POs. We recall that for every event we only have to consider the PO generated for the invariant $pc = y \Rightarrow \ldots$ as all other POs are trivially discharged.

\[
A, I, J, H, BAP \vdash pc' = y \Rightarrow cond_1' \land \cdots \land cond_n'
\]

We can simplify the POs as $A$ and $I$ in the hypotheses are not needed. Furthermore, the only useful invariant in $J$ is the one that holds for $pc = x$. The right-hand side of the PO can be simplified too by replacing the implication. The resulting PO looks then as follows.

\[
pc = x \Rightarrow \ldots, H, BAP \vdash cond_1' \land \cdots \land cond_n'
\]

In the following, we will further simplify the POs by replacing $H$ and $BAP$ by the concrete expressions of the events. Furthermore, in all POs, except the ones associated with an event block, we can replace the primed conditions with their unprimed version. This is valid because only $pc$ is updated and the conditions do not refer to $pc$, hence, the primed and unprimed versions are identical.
Event Block.

\begin{verbatim}
    evt: refines evt
    any p_1, ..., p_p where
        pc = s
        pinit_1
        ...
        pinit_p
    then
        act_1
        ...
        act_x
        pc := pc + 1
    end
\end{verbatim}

At position $s$ the preconditions of the event block hold. The invariant of the non-trivial PO states the postconditions of the event block. Hence, the PO to prove is as follows.

\[
    pc = s \Rightarrow \text{pre}_1 \land \cdots \land \text{pre}_n \\
    pc = s \land \text{pinit}_1 \land \cdots \land \text{pinit}_p \\
    BAP \\
    \vdash \text{post}'_1 \land \cdots \land \text{post}'_m
\]

We prove PO (12) with help of the PO $E/POST$ that we generated for the schedule file.

\[
    \text{pre}_1, \ldots, \text{pre}_n, \text{pinit}_1, \ldots, \text{pinit}_p, BAP \vdash \text{post}'_1 \land \cdots \land \text{post}'_m
\]

Branch Block. We prove invariant preservation for all five possible events that are generated for a branch block $B$.

\begin{verbatim}
    if _true :
        when
            pc = s
            bc
        then
            pc := pc + 1
        end
\end{verbatim}

At position $s$ the preconditions of the branch block hold. The invariant for the next position varies depending on whether the branch block has a “then” part or not.
If the “then” part of the block is not empty, according to our analysis, the invariant of the PO states the block’s precondition plus the branch condition. Hence, the PO to prove looks as follows and is easily proved without help of any other PO.

\[
\begin{align*}
  pc &= s \Rightarrow pre_1 \land \cdots \land pre_n \\
  pc &= s \land bc \\
  BAP \\
  \vdash &\quad pre_1 \land \cdots \land pre_n \land bc
\end{align*}
\] (13)

If the “then” part of the block is empty, according to our analysis, the invariant of the PO states the block’s postconditions. Hence, the PO to prove looks as follows.

\[
\begin{align*}
  pc &= s \Rightarrow pre_1 \land \cdots \land pre_n \\
  pc &= s \land bc \\
  BAP \\
  \vdash &\quad post_1 \land \cdots \land post_m
\end{align*}
\] (14)

We prove PO (14) with help of the PO \texttt{B/then/POST} that is generated for the schedule file if there is no “then” part.

\[
\begin{aligned}
  pre_1, \ldots, pre_n, bc &\vdash post_1 \land \cdots \land post_m
\end{aligned}
\]

If a branch block is written in the long form, it has an “else” part. According to our analysis, the invariant of the PO states the block’s precondition plus the negated branch condition. Hence, the PO to prove looks as follows and is easily proved without help of any other PO.

\[
\begin{align*}
  pc &= s \Rightarrow pre_1 \land \cdots \land pre_n \\
  pc &= s \land \neg bc \\
  BAP \\
  \vdash &\quad pre_1 \land \cdots \land pre_n \land \neg bc
\end{align*}
\] (15)
If a branch block is written in the short form, it has no “else” part and, according to our analysis, the invariant of the PO states the block’s postconditions. Hence, the PO to prove looks as follows.

\[ pc = s \Rightarrow \text{pre}_1 \land \cdots \land \text{pre}_n \]
\[ pc = s \land \neg bc \]
\[ BAP \]
\[ \vdash \]
\[ \text{post}_1 \land \cdots \land \text{post}_m \]  

We prove PO (16) with help of the PO \texttt{B/else/POST} that is generated for the schedule file if there is no “else” part.

\[ \text{pre}_1, \ldots, \text{pre}_n, \neg bc \vdash \text{post}_1 \land \cdots \land \text{post}_m \]

At the middle position \( m \), where the \texttt{else} keyword is located as well as at the end position \( e \), where the \texttt{fi} keyword is located the postconditions of the branch block hold. Hence the PO to prove looks as follows and is easily proved.

\[ pc = m \Rightarrow \text{post}_1 \land \cdots \land \text{post}_m \]
\[ pc = m \]
\[ BAP \]
\[ \vdash \]
\[ \text{post}_1 \land \cdots \land \text{post}_m \]
At the end position $e$, the postconditions of the branch block hold. According to our analysis, the invariant of the PO states the block’s postconditions too. Hence the PO to prove looks as follows and is easily proved.

\[
\begin{align*}
    pc &= e \Rightarrow post_1 \land \cdots \land post_m \\
    pc &= e \\
    BAP \\
    \vdash \\
    post_1 \land \cdots \land post_m
\end{align*}
\]

(18)

**Loop Block.** We prove invariant preservation for all three events that are generated for a loop block $L$.

At position $s$ of the loop block the block’s preconditions hold. If the loop body is not empty, the invariant of the PO states the precondition of the block plus the loop condition. Hence, the PO to prove looks as follows and is easily proved without help of any other PO.

\[
\begin{align*}
    pc &= s \Rightarrow pre_1 \land \cdots \land pre_n \\
    pc &= s \land lc \\
    BAP \\
    \vdash \\
    pre_1 \land \cdots \land pre_n \land lc
\end{align*}
\]

(19)
In case of an empty body loop the invariant of the PO states the preconditions of the block. The PO looks as follows and is easily proved.

$$pc = s \Rightarrow pre_1 \land \cdots \land pre_n$$
$$pc = s \land lc$$
$$BAP$$
$$\vdash$$
$$pre_1 \land \cdots \land pre_n$$

```
do_false:
  when
    pc = s
     ¬lc
  then
    pc := e + 1
  end
```

According to our analysis, the invariant of the PO states the block’s postconditions. Hence, the PO to prove looks as follows.

$$pc = s \Rightarrow pre_1 \land \cdots \land pre_n$$
$$pc = s \land ¬lc$$
$$BAP$$
$$\vdash$$
$$post_1 \land \cdots \land post_m$$

We prove PO (21) with help of the PO \texttt{L/else/POST} that is generated for the schedule file.

$$pre_1, \ldots, pre_n, ¬lc \vdash post_1 \land \cdots \land post_m$$

```
do return:
  when
    pc = e
  then
    pc := s
  end
```
At the end position \( e \), where the `od` keyword is located as well as at the start position \( s \) of the loop block the block’s preconditions hold. Hence the PO to prove looks as follows and is easily proved.

\[
\begin{align*}
pc = e & \Rightarrow \text{pre}_1 \land \cdots \land \text{pre}_n \\
\neg pc = e & BAP \\
\vdash \quad & \text{pre}_1 \land \cdots \land \text{pre}_n 
\end{align*}
\]

\( (22) \)

### 7.5.6. Guard Strengthening (GRD)

Only the events corresponding to event blocks are refined events. For all the other events we do not need to prove any additional POs. For the refined events, we have only to prove guard strengthening. There are no POs with respect to action simulation and equivalence since the abstract actions do not change in the refined event and the additional action that updates \( pc \) is new.

For the event that corresponds to an event block with start position \( s \) we have the following POs to prove.

\[
\begin{align*}
A, I, J, H & \vdash grd_1 \\
\vdots \\
A, I, J, H & \vdash grd_z 
\end{align*}
\]

The only useful invariant in \( J \) is the one for position \( s \), which states the precondition of the event block.

\[
\begin{align*}
\neg pc = s & \Rightarrow \text{pre}_1 \land \cdots \land \text{pre}_n 
\end{align*}
\]
The guards $H$ include the predicate $pc = s$ and the parameter initialisations. Hence, for every guard $grd_i$ of the abstract event, we have a simplified PO of the following form.

\[
\begin{align*}
pc = s &\Rightarrow pre_1 \land \cdots \land pre_n \\
pc = s \land pinit_1 \land \cdots \land pinit_p &\vdash grd_i
\end{align*}
\]  \hspace{1cm} (23)

We can prove PO (23) for every guard $grd_i$ with help of the PO $E/GRD$ generated for the scheduler.

\[
pre_1, \ldots, pre_n, init_1, \ldots, init_p \vdash grd_1 \land \cdots \land grd_z
\]

7.6. Conclusion

Our plug-in provides an editor where the user can write a schedule for a model to generate code from. It automatically generates pre- and postconditions for the different blocks of the schedule based on the schedule’s structure. The plug-in further generates proof obligations related to the schedule and the original model. The user must discharge these POs in order to prove that generating code following the schedule results in code that “refines” the original model. We proved at the meta-level that the POs that our plug-in generates for the schedule file are indeed sufficient to prove refinement.
8. Simulation

Refinement in Event-B preserves safety properties. Liveness, however, is not guaranteed by default. This means that by refining our model we can end up with a model that has no functionality at all. We recall that a system that does nothing is always safe. There are different methods to get an impression how liveness properties are fulfilled by a system. In model checking, liveness properties are treated together with safety properties using a temporal logic like LTL [9]. Event-B does not provide a temporal logic per se. There is research [30, 40] that addresses the integration of a temporal logic to Event-B to reason about the liveness of a model. The question is always how refinement can preserve liveness and what additional properties are required for liveness preservation.

In our case study to develop a train controller, our focus was on safety not liveness. However, we wanted to see whether our model behaves as intended. We do not have specifically formulated liveness properties but want to see whether the controller assigns Movement Authorities to the trains and if they are moving within these MAs. From our own experience, simulation has been useful as an alternative way to detect flaws in the design regarding liveness.

Inspired by the simulation of an elevator system that was developed for an Event-B course, we specified the functionality of a simulation framework for our train control system. Among the existing simulation plug-ins for the Rodin platform, ProB was the most promising for our intention. Besides bringing a model checker into the world of Event-B, ProB also supports the simulation of Event-B models. The ProB framework for simulation allows the user to create a graphical view of the modelled system in which the model’s state is represented and actions can be triggered.

For our purpose, we had to extend the functionality of ProB’s simulation framework. Our goal was to only simulate the environment events of our model in Event-B and run the generated C code of the controller in parallel. The communication between controller and environment should be established by shared variables and handled by the simulation framework. Hitachi developed a prototype plug-in for Rodin based on the ProB plug-in with the additional functionality to interact with C code. In the following we briefly describe the developed plug-in and the challenges of co-simulation.

In order to only simulate environment events, the simulator must know which part of the model belongs to the environment. To simplify this task we created a plug-in that is customised to our train control system. Hence, the simulation plug-in knows exactly the environment’s variables and events. It includes the
graphical representation of our test topology for the network, which is depicted in Figure 8.1.

Furthermore, the plug-in links the graphical representation of the topology with the state of the model and assigns environment events to different buttons. When starting the simulation, one can either use these buttons to let the environment proceed in a stepwise manner or let the simulator choose an execution order of the environment events. The control of the simulation is transferred to the controller after every execution of an environment event by calling the compiled C code that is included in the plug-in as a dynamic link library (dll). The state of the environment is updated as soon as the controller has finished its execution.

### 8.1. Instantiation of the Topology

During development we did refine the environment only as far as necessary. There was still non-determinism in the environment events when we generated code for the controller part. Keeping the environment abstract means fewer assumptions on it and therefore our controller is applicable to more concrete topologies. One of our goals was to keep the system generic and prove its correctness at this level.

For the simulation of the environment we need a specific topology. As mentioned before, we used a test topology for our case study. To prove that the specific topology is compliant with the generic model we must show that the topology’s concrete context is an instance of the abstract one. As described in
Section 6.3.4, we prove that a specific model instantiates an abstract one as follows. The assumptions stated as axioms in the abstract model become theorems in the concrete model and must be proved to hold.

The specific topology is defined as a context using only data types that are allowed by our code generation plug-in. Note that the controller only runs with such concrete contexts. The built-in theorem provers of Rodin are not very useful to prove the theorems in the concrete context. ProB’s model checker, however, was able to do the proofs in a blink of an eye. For this task, a model checker is better suited than a theorem prover. This is a nice example where two techniques are combined to get the best result.

8.2. Synchronisation

When developing a controller at the system level, one has also to model the controller’s environment. In Event-B we use a top down approach for modelling, meaning that we start at a very abstract level. As a result, we abstract away details like communication between controller and environment. Therefore, the two sides exchange information by shared variables that both can access. In general, both sides can read and write to the shared variables, which works fine in the Event-B model. Although the controller and the environment are modelled within the same machine by a set of events each, only one event, and therefore one side, does access a shared variable at a time. In a sense, the controller and the environment are synchronised.

This is not the usual behaviour in real systems. In general, we cannot schedule the environment, which means it is not guaranteed that the shared variables do not change during execution of a controller event. Our solution in the railway controller is that we only work on a copy of the shared variables that we call a snapshot. This snapshot remains unchanged during the execution of controller events and the controller always works on consistent data. The disadvantage of taking a snapshot is that the data might be out-dated.

For the train control system, out-dated snapshots are no problem. Its environment comes to a stable and safe state if it gets no updates from the controller and the controller works conservative if it works on an out-dated state of the environment. We already justified these system properties in terms of unreliable radio transmission in Section 4.4.

In our simulation plug-in, the framework takes the snapshot of the current environment state. Afterwards it calls the controller code to run and thereby hands over the data of the snapshot. While the controller runs the environment might change. The controller, however, has no notification about this change. After the controller has run its code, the simulation framework is noticed and the controller hands over the computed values which the framework applies to the environment.
8.3. Evaluation

We used the simulation plug-in to informally validate the following liveness properties of our train control system.

- Before extending the MA of any train, the controller reduces the MA of every train that moved forward and reported its new position since the last run of the controller.

- The MA of every train is extended as far as possible. In general, this is the end of the activated network or the rear of the train ahead.

- If the trains do not update their positions, the controller reaches a state where it does not update any MA. After receiving new position updates from the trains the controller also updates the MAs again.

The simulation plug-in for our test topology includes two active trains that can move on the network. The MAs of the trains are initialised to the same area as the trains initially occupy. The active network consists of the sections occupied by the trains. We observed the following behaviour after starting the simulation.

As expected, the controller did not increase any of the MAs. Furthermore, we were not able to move the trains, as they already reached their assigned LoAs. Once we activated more sections of the network, the controller started to extend the MAs. As a result, we were able to let our trains move forward. From time to time we applied the event that represents a train sending its position to the controller. In its next run, the controller successfully processed the updated position of the train and adapted the corresponding MA by reducing them at the rear end. To be precise, we did not see whether the controller updated the MA since this is a controller internal variable. What we saw was that the LoA of the following train was updated which implies that the MA of the train ahead must have been updated before.

The simulation showed us that the developed train control software behaves according to our expectations. Although this is no formal verification of the system’s liveness, it is good enough to give us confidence that we did not build a system that does nothing.

Unfortunately, the plug-in does not allow the simulation of entering and leaving trains. All trains must be activated when initialising the simulation. As the test topology is circle free, the movements of the trains ended as soon as they reached the opposing end of the network. Hence, we were able to simulate only short train journeys.

8.4. Conclusion

With the simulation plug-in we have a useful tool to check the behaviour of the developed controller. Simulation thereby acts as an informal way to justify the
expected liveness properties of a system. Our plug-in supports co-simulation of the
environment and the controller. While the Event-B model is used to simulate the
environment, we use the generated C code for the controller. When we define the
schedule to generate code, we can easily influence the behaviour of the controller.
To check that we did not unintentionally change the controller’s behaviour it
is reasonable to observe the running code rather than the controller’s Event-B
model.
9. Conclusion and Future Work

We presented in this thesis different new techniques for developing large and complex systems in Event-B and for generating sequential code that runs on hardware. We showed the different steps from requirements to C code on a large case study. We modelled a train control system together with its environment to investigate the limitations of the standard Event-B framework. After developing techniques to overcome these limitations we used the case study to evaluate our approaches.

Requirements. We started with presenting a selection of requirements that we extracted from implementation oriented specifications. The goal was to keep the requirements as simple and clear as possible. A lot of effort was spent for writing these requirements. Clear and unambiguous requirements are essential for formal modelling in Event-B. We restricted the requirements to the core functionality of the train control system, since we rated the quality of the requirements higher than their quantity.

Design of the Train Control System. Based on the defined requirements we modelled the train control system in Event-B. We were surprised that modelling the environment was more challenging than modelling the controller. On second thoughts this is obvious. Our system builds on the functionality of classical interlocking system. This interlocking system that dynamically changes the active network, the trains with their On-Board Units, and the communication interfaces all belong to the environment. To prove the required safety properties of the overall system we had to refine the environment to a certain concrete level, although it is not the target of our development. In addition, simulating the system forced us to refine the environment even further to a very concrete level.

Instead of presenting the actual Event-B model which includes about one hundred machines, we discussed in Chapter 4 a selection of interesting topics from an engineer’s point of view. After all, our goal was to model the system such that it is possible to generate code from the model and simulate the system with a concrete topology. Hence, we kept this goal at the back of our minds when we made design decisions.

The described design of different parts of the train control system should serve as a lead for future developments of train control systems. In our development we postponed the splitting of global events into their controller and environment parts to the very end. In a future development of a similar system we would pay
more attention to do this step at an earlier stage hoping for less modelling effort and fewer proofs.

**Abstract Data Types.** For large systems like our train control system, stepwise refinement alone is not sufficient to cope with the model’s complexity. We searched for different ways of abstraction in and found the notion of Abstract Data Types (ADTs). We analysed this abstraction technique and introduced the concept in Event-B. With ADTs we provide another form of abstraction in addition to refinement. Using ADTs does not only simplify the modelling of complex systems but also results in easier proofs.

Our approach to introducing ADTs in Event-B builds on the concept of generic instantiation. We built a plug-in for the Rodin platform to perform generic instantiation. This plug-in allows a user to apply ADTs in Event-B models in a convenient way.

As future work we would like to overcome some of the current limitations of our approach. As mentioned in Chapter 5, we cannot presently specify parameterised ADTs. To handle this, we would need to extend the semantics of Event-B contexts and adapt the generic instantiation technique accordingly. An alternative solution is to incorporate the concept of data type instantiation within the Theory plug-in.

**Code Generation.** We developed an approach to generating C code from Event-B models. We thereby focused on the approach’s correctness rather than the number of included features. Hence we defined a simple yet powerful scheduling language to specify the desired execution order of the translated events. We support the translation of a subset of Event-B’s language that was sufficient to generate code for the train controller. In contrast to other code generation tools for Event-B our approach generates code that is correct-by-construction. Our approach guarantees that the safety properties of the Event-B model are also fulfilled by the generated code. This closes the gap in terms of verification between Event-B models and generated code.

As future work we would like to consider loop termination and liveness properties in general. The challenge here is to integrate standard loop variant reasoning into the scheduled Event-B model. Other future work that suggests itself is the adaptation of our code generation approach to different programming languages.

**Code Generation Plug-in.** We presented the different steps of our code generation approach in Chapter 6. We developed a plug-in for the Rodin platform that supports the user to perform these steps. It automatically translates the Event-B model to C code according to the user-defined schedule. The first version of our plug-in also generated the scheduled Event-B model to prove the validity of the user-defined schedule based on refinement proofs.

We developed an improved version of the code generation plug-in that generates a small set of proof obligations after the user has defined the schedule. We proved
that the generated proof obligations are sufficient to prove the validity of the schedule. At a meta-level we proved that the generated proof obligations imply the proof obligations that Rodin generates to prove refinement of the scheduled model. With this, we significantly reduce the number of proof obligations to prove the validity of the schedule.

With our current version of the plug-in it is not possible to generate code if the schedule’s validity is not proved. Checking the schedule’s validity first is an important improvement in terms of safety. The correctness of the generated code, which means that it fulfils the same safety properties as the original Event-B model, is only guaranteed if the code was generated with help of a valid schedule.

The structural editor that we provide in our plug-in uses a flat structure for the design of schedules. This means that the different event, loop and branch blocks are defined side-by-side at the same level. The nesting of different blocks is realised by using references to blocks. To check the schedule’s layout, the plug-in provides a pretty-print view that shows the resulting nested structure of the schedule. For a future version of the plug-in we would like to update the editor to a textual version such that the user can specify the blocks’ relations directly using indentation.

Simulation. Towards the end of our project, we felt the need to get an impression of the generated controller’s performance. We experienced in earlier projects, that simulation is a useful method to informally validate liveness properties of a system. While our research partners at Hitachi developed the prototype plug-in for co-simulation based on ProB, we prepared the Event-B models of the train control system’s environment for the later simulation. During modelling of our system we left the environment abstract on purpose. The more abstract we keep the environment, the more applicable is our developed controller.

To use the developed plug-in, we had to refine the environment to a more concrete level to reduce non-determinism. For example, we had to refine the abstract event to activate an arbitrary part of the network to two concrete events: one to activate tracks and one to activate points. We refined many of the abstract environment events in a similar way.

The plug-in includes the compiled code of the controller; the layout of the graphical simulation interface is specified in a file attached to the environment’s model. Hence, we can use the plug-in to simulate the controller together with with different network topologies. However, for a change in the controller a new instance of the plug-in must be generated that includes the new code. If the interface of the controller changes, the plug-in must be redeveloped.

For the simulation of a future development we would choose a different approach. Utilising ProB for the simulation plug-in appears reasonable as it already provides a simulation framework for Event-B models. However, it can only handle the non-determinism within a model to a certain extend and the environment must be refined to a concrete level. With such a concrete environment at hand, we are only one step away from the possibility to generate code for the environment.
Hence, we would translate the environment model to program code and link it to a graphical interface. We successfully used this approach already for simulating the elevator system.
Bibliography


A. Formal Description of the Test Topology

A.1. Section

\[ \text{max\_section} = 20 \]
\[ \text{section\_length} = \{ \]
\[ 0 \mapsto 10, 1 \mapsto 7, 2 \mapsto 5, 3 \mapsto 10, 4 \mapsto 5, 5 \mapsto 5, 6 \mapsto 10, 7 \mapsto 5, 8 \mapsto 3, 9 \mapsto 3, \]
\[ 10 \mapsto 7, 11 \mapsto 10, 12 \mapsto 10, 13 \mapsto 7, 14 \mapsto 5, 15 \mapsto 5, 16 \mapsto 10, 17 \mapsto 5, \]
\[ 18 \mapsto 7, 19 \mapsto 10 \} \]

A.2. Block

\[ \text{max\_block} = 6 \]
\[ \text{max\_block\_per\_section} = 1 \]
\[ \text{section\_2\_block\_array} = \{ \]
\[ 0 \mapsto 0 \mapsto 6, 1 \mapsto 0 \mapsto 6, 2 \mapsto 0 \mapsto 0, 3 \mapsto 0 \mapsto 3, \]
\[ 4 \mapsto 0 \mapsto 2, 5 \mapsto 0 \mapsto 0, 6 \mapsto 0 \mapsto 4, 7 \mapsto 0 \mapsto 6, \]
\[ 8 \mapsto 0 \mapsto 1, 9 \mapsto 0 \mapsto 2, 10 \mapsto 0 \mapsto 6, 11 \mapsto 0 \mapsto 6, \]
\[ 12 \mapsto 0 \mapsto 6, 13 \mapsto 0 \mapsto 6, 14 \mapsto 0 \mapsto 1, 15 \mapsto 0 \mapsto 6, \]
\[ 16 \mapsto 0 \mapsto 5, 17 \mapsto 0 \mapsto 0, 18 \mapsto 0 \mapsto 6, 19 \mapsto 0 \mapsto 6 \} \]

A.3. Train

\[ \text{max\_train} = 3 \]

A.4. Track

This context is only relevant for simulation. The TLC works independently of this context.

\[ \text{track\_id} = 0 \ldots 7 \]
\[ \text{track\_sections} = \{ \]
\[ 0 \mapsto \{0, 1\}, \]
\[ 1 \mapsto \{2, 3, 4\}, \]
\[ 2 \mapsto \{5, 6, 7\}, \]
3 ⇔ \{8, 9\},
4 ⇔ \{10, 11\},
5 ⇔ \{12, 13, 14\},
6 ⇔ \{15, 16, 17\},
7 ⇔ \{18, 19\} \}

\textit{track}\_\textit{graph} = \{
0 ⇔ \{0 ⇔ 1\},
1 ⇔ \{2 ⇔ 3, 3 ⇔ 4\},
2 ⇔ \{5 ⇔ 6, 6 ⇔ 7\},
3 ⇔ \{8 ⇔ 9\},
4 ⇔ \{10 ⇔ 11\},
5 ⇔ \{12 ⇔ 13, 13 ⇔ 14\},
6 ⇔ \{15 ⇔ 16, 16 ⇔ 17\},
7 ⇔ \{18 ⇔ 19\} \}

\textbf{A.5. Route}

This context is only relevant for simulation. The TLC works independently of this context.

\textit{route}\_\textit{id} = 0 .. 2
\textit{route}\_\textit{sections} = \{
0 ⇔ \{0, 1, 2, 3, 4, 10, 11\},
1 ⇔ \{0, 1, 5, 6, 7, 8, 9, 10, 11\},
2 ⇔ \{12, 13, 14, 15, 16, 17, 18, 19\} \}

\textit{route}\_\textit{graph} = \{
0 ⇔ \{0 ⇔ 1, 1 ⇔ 2, 2 ⇔ 3, 3 ⇔ 4, 4 ⇔ 10, 10 ⇔ 11\},
1 ⇔ \{0 ⇔ 1, 1 ⇔ 5, 5 ⇔ 6, 6 ⇔ 7, 7 ⇔ 8, 8 ⇔ 9, 9 ⇔ 10, 10 ⇔ 11\},
2 ⇔ \{12 ⇔ 13, 13 ⇔ 14, 14 ⇔ 15, 15 ⇔ 16, 16 ⇔ 17, 17 ⇔ 18, 18 ⇔ 19\} \}

\textbf{A.6. Points}

This context is only relevant for simulation. The TLC works independently of this context.

\textit{points}\_\textit{id} = 0 .. 2
\textit{points}\_\textit{graph} = \{
0 ⇔ \{1 ⇔ 2, 1 ⇔ 5, 17 ⇔ 18\},
1 ⇔ \{7 ⇔ 8, 14 ⇔ 15\},
2 ⇔ \{4 ⇔ 10, 9 ⇔ 19\} \}
A.7. Conflicts

This context is only relevant for simulation. The TLC works independently of this context.

\[ conflicts = \{ \\
    0 \mapsto 19, 19 \mapsto 0, \\
    1 \mapsto 18, 18 \mapsto 1, \\
    5 \mapsto 17, 17 \mapsto 5, \\
    6 \mapsto 16, 16 \mapsto 6, \\
    7 \mapsto 15, 15 \mapsto 7 \} \]
B. Further Code Generation Examples

This is a collection of small examples (most of them adopted from [3]) to demonstrate how we provide a schedule to build a sequential program. In every example the following parts are stated.

(a) All the events of the final Event-B model of the original development. The overview of the variables and the invariants of the machine are omitted.

(b) The events are restricted, meaning that all variables and constants are of type BOOL or C_INT. Furthermore, parallel assignments are rearranged or an auxiliary event parameter is introduced if needed.

(c) A possible schedule for the given events.

(d) The computed invariants (pre- or postconditions) extracted from the schedule.

(e) The schedule is encoded into the restricted events and additional events are introduced to model the program flow of the schedule.

(f) The resulting program code translated from the schedule and the restricted events. We omit the surrounding code and declarations that would be necessary for runnable code.

(g) The program code is flattened by substituting the if-then-else and while statements with if statements and gotos. This representation acts as an intermediate comparison step between program code and scheduled events.
B.1. Simple Search

(a) Original Events

\[
\text{init :}
\begin{align*}
\text{begin} \\
r &:= 1 \\
\text{end}
\end{align*}
\]

\[
\text{final :}
\begin{align*}
\text{when} & \quad f(r) = v \\
\text{then} & \quad \text{SKIP} \\
\text{end}
\end{align*}
\]

\[
\text{progress :}
\begin{align*}
\text{when} & \quad f(r) \neq v \\
\text{then} & \quad r := r + 1 \\
\text{end}
\end{align*}
\]

(b) Restricted Events

\[
\text{init :}
\begin{align*}
\text{begin} \\
r &:= 1 \\
\text{end}
\end{align*}
\]

\[
\text{final :}
\begin{align*}
\text{when} & \quad f(r) = v \\
\text{then} & \quad \text{SKIP} \\
\text{end}
\end{align*}
\]

\[
\text{progress :}
\begin{align*}
\text{when} & \quad f(r) \neq v \\
\text{then} & \quad r := c\_plus(r \mapsto 1) \\
\text{end}
\end{align*}
\]

(c) Schedule

\[
\begin{align*}
\text{init;} \\
0: & \quad \textbf{do} \quad (f(r) \neq v) \\
1: & \quad \text{progress} \\
2: & \quad \textbf{od}; \\
3: & \quad \{ f(r) = v \} \\
\end{align*}
\]

\[
\text{final}
\]
init:
begin
  r := 1
  pc := 0
end
d0l_true:
when
  pc = 0
  f(r) ≠ v
then
  pc := pc + 1
end
d0l_false:
when
  pc = 0
  ¬(f(r) ≠ v)
then
  pc := 3
end

progress:
when
  pc = 1
then
  r := c_plus(r ⊕ 1)
  pc := pc + 1
end
d0l_return:
when
  pc = 2
then
  pc := 0
end
final:
when
  pc = 3
then
  pc := pc + 1
end

\( pc = 0 \Rightarrow \top \)
\( pc = 1 \Rightarrow f(r) ≠ v \)
\( pc = 2 \Rightarrow \top \)
\( pc = 3 \Rightarrow f(r) = v \)

\( r=1; \) \( r=1; \)
while(f[r]!=v) { pc0: if(!(f[r]!=v)) goto pc3;
  r=c_plus(r,1);
  pc1: r=c_plus(r,1);
pc2: goto pc0;
pc3: ;
(f) Program Code  (g) Flattened Code
B.2. Binary Search in a Sorted Array

\[
\begin{aligned}
\text{init} : & \quad \begin{aligned}
\text{begin} \\
p & := 1 \\
q & := n \\
r & := (1 + n) \div 2 \\
\text{end}
\end{aligned} \\
\text{inc} : & \quad \begin{aligned}
\text{when} & \quad f(r) < v \\
\text{then} & \quad p := r + 1 \\
r & := (r + 1 + q) \div 2 \\
\text{end}
\end{aligned} \\
\text{dec} : & \quad \begin{aligned}
\text{when} & \quad v < f(r) \\
\text{then} & \quad q := r - 1 \\
r & := (p + r - 1) \div 2 \\
\text{end}
\end{aligned} \\
\text{final} : & \quad \begin{aligned}
\text{when} & \quad f(r) = v \\
\text{then} & \quad \text{SKIP} \\
\text{end}
\end{aligned}
\end{aligned}
\]

(a) Original Events
init:
begin
  p := 1
  q := n
  r := c_div(c_plus(1 → n) → 2)
end

inc:
when
  f(r) < v
then
  p := c_plus(r → 1)
  r := c_div(c_plus(c_plus(r → 1) → q) → 2)
end

dec:
when
  v < f(r)
then
  q := c_minus(r → 1)
  r := c_div(c_minus(c_plus(p → r) → 1) → 2)
end

final:
when
  f(r) = v
then
  SKIP
end

(b) Restricted Events

init;
0: do(f(r) ≠ v)  
1: if(f(r) < v)  
2:   inc  
3: else  
4:   dec  
5: fi  
6: od;  
7: {f(r) = v}  
8: final

(c) Schedule
init:
begin
  \( p := 1 \)
  \( q := n \)
  \( r := \text{c\_div}(\text{c\_plus}(1 \mapsto n) \mapsto 2) \)
  \( pc := 0 \)
end

dol1\_true:
when
  \( pc = 0 \)
  \( f(r) \neq v \)
then
  \( pc := pc + 1 \)
end

dol1\_false:
when
  \( pc = 0 \)
  \( f(r) < v \)
then
  \( pc := pc + 1 \)
end

if2\_true:
when
  \( pc = 1 \)
  \( f(r) < v \)
then
  \( pc := pc + 1 \)
end

if2\_false:
when
  \( pc = 1 \)
  \( f(r) \neq v \)
then
  \( pc := 4 \)
end

inc:
when
  \( pc = 2 \)
then
  \( p := \text{c\_plus}(r \mapsto 1) \)
  \( r := \text{c\_div}(\text{c\_plus}(\text{c\_plus}(r \mapsto 1) \mapsto q) \mapsto 2) \)
  \( pc := pc + 1 \)
end

if2\_exit:
when
  \( pc = 3 \)
then
  \( pc := 6 \)
end

dec:
when
  \( pc = 4 \)
then
  \( q := \text{c\_minus}(r \mapsto 1) \)
  \( r := \text{c\_div}(\text{c\_minus}(\text{c\_plus}(p \mapsto r) \mapsto 1) \mapsto 2) \)
  \( pc := pc + 1 \)
end

if2\_end:
when
  \( pc = 5 \)
then
  \( pc := pc + 1 \)
end

(d) Scheduled Events
(d) Scheduled Events - cont.

\[
\begin{align*}
&pc = 0 \Rightarrow \top \\
&pc = 1 \Rightarrow f(r) \neq v \\
&pc = 2 \Rightarrow f(r) \neq v \land f(r) < v \\
&pc = 3 \Rightarrow \top \\
&pc = 4 \Rightarrow f(r) \neq v \land \neg(f(r) < v) \\
&pc = 5 \Rightarrow \top \\
&pc = 6 \Rightarrow \top \\
&pc = 7 \Rightarrow f(r) = v
\end{align*}
\]

(e) Invariants

\[
\begin{align*}
p &= 1; \\
q &= n; \\
r &= \text{c\_div}(\text{c\_plus}(1,n),2); \\
p &= 1; \\
q &= n; \\
r &= \text{c\_div}(\text{c\_plus}(1,n),2);
\end{align*}
\]

while(f[r]!=v) { 
  if(f[r]<v) { 
    p = \text{c\_plus}(r,1); 
    r = \text{c\_div}(\text{c\_plus}(\text{c\_plus}(r,1),q),2); 
  } else { 
    q = \text{c\_minus}(r,1); 
    r = \text{c\_div}(\text{c\_minus}(\text{c\_plus}(p,r),1),2); 
  } 
}

(f) Program Code

pc0: if(!f[r]!=v) goto pc7; pc1: if(!f[r]<v) goto pc4; pc2: p = \text{c\_plus}(r,1); r = \text{c\_div}(\text{c\_plus}(\text{c\_plus}(r,1),q),2); pc3: goto pc6; pc4: q = \text{c\_minus}(r,1); r = \text{c\_div}(\text{c\_minus}(\text{c\_plus}(p,r),1),2); pc5: goto pc6; pc6: goto pc0; pc7: ;

(g) Flattened Code
B.3. Minimum of an Array of Natural Numbers

(a) Original Events

\begin{align*}
\text{init :} & \quad \text{begin} \\
& \hspace{1cm} p := 1 \\
& \hspace{1cm} q := n \\
& \hspace{1cm} m := \mathbb{N} \\
& \hspace{1cm} \text{end} \\
\text{inc :} & \quad \text{begin} \\
& \hspace{1cm} \text{when} \quad p < q \\
& \hspace{1.5cm} f(p) > f(q) \\
& \hspace{1cm} \text{then} \quad p := p + 1 \\
& \hspace{1cm} \text{end} \\
\text{dec :} & \quad \text{begin} \\
& \hspace{1cm} \text{when} \quad p < q \\
& \hspace{1.5cm} f(p) \leq f(q) \\
& \hspace{1cm} \text{then} \quad q := q - 1 \\
& \hspace{1cm} \text{end} \\
\text{minimum :} & \quad \text{begin} \\
& \hspace{1cm} \text{when} \quad p = q \\
& \hspace{1.5cm} \text{then} \quad m := f(p) \\
& \hspace{1cm} \text{end}
\end{align*}

(b) Restricted Events

\begin{align*}
\text{init :} & \quad \text{begin} \\
& \hspace{1cm} p := 1 \\
& \hspace{1cm} q := n \\
& \hspace{1cm} m := 0 \\
& \hspace{1cm} \text{end} \\
\text{inc :} & \quad \text{begin} \\
& \hspace{1cm} \text{when} \quad p < q \\
& \hspace{1.5cm} f(p) > f(q) \\
& \hspace{1cm} \text{then} \quad p := c\_plus(p \rightarrow 1) \\
& \hspace{1cm} \text{end}
\end{align*}
(b) Restricted Events - cont.

init;
0: \textbf{do}(p < q)
1: \textbf{if}(f(p) > f(q))
2: \quad \textbf{inc}
3: \textbf{else}
4: \quad \textbf{dec}
5: \textbf{fi}
6: \textbf{od};
7: \{p = q\}
8: \textbf{minimum}

(c) Schedule

After the do-loop, \(\neg(p < q)\) holds. Together with the invariant \(p \leq q\) we have that after the loop \(p = q\) holds and therefore the assertion is satisfied.
<table>
<thead>
<tr>
<th>Event</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>init:</td>
<td>begin ( p := 1 ) ( q := n ) ( m := 0 ) ( pc := 0 ) end</td>
</tr>
<tr>
<td>do1_true:</td>
<td>when ( pc = 0 ) ( p &lt; q ) then ( pc := pc + 1 ) end</td>
</tr>
<tr>
<td>do1_false:</td>
<td>when ( pc = 0 ) ( \neg(p &lt; q) ) then ( pc := 7 ) end</td>
</tr>
<tr>
<td>if2_true:</td>
<td>when ( pc = 1 ) ( f(p) &gt; f(q) ) then ( pc := pc + 1 ) end</td>
</tr>
<tr>
<td>if2_false:</td>
<td>when ( pc = 1 ) ( \neg(f(p) &gt; f(q)) ) then ( pc := 4 ) end</td>
</tr>
<tr>
<td>if2_exit:</td>
<td>when ( pc = 3 ) then ( pc := 5 ) end</td>
</tr>
<tr>
<td>if2_end:</td>
<td>when ( pc = 5 ) then ( pc := pc + 1 ) end</td>
</tr>
<tr>
<td>inc:</td>
<td>when ( pc = 2 ) then ( p := c_{\text{plus}}(p \mapsto 1) ) ( pc := pc + 1 ) end</td>
</tr>
<tr>
<td>dec:</td>
<td>when ( pc = 4 ) then ( q := c_{\text{minus}}(q \mapsto 1) ) ( pc := pc + 1 ) end</td>
</tr>
<tr>
<td>do1_return:</td>
<td>when ( pc = 6 ) then ( pc := 0 ) end</td>
</tr>
<tr>
<td>minimum:</td>
<td>when ( pc = 7 ) then ( m := f(p) ) ( pc := pc + 1 ) end</td>
</tr>
</tbody>
</table>

(d) Scheduled Events
\[pc = 0 \Rightarrow \top\]
\[pc = 1 \Rightarrow p < q\]
\[pc = 2 \Rightarrow p < q \land f(p) > f(q)\]
\[pc = 3 \Rightarrow \top\]
\[pc = 4 \Rightarrow p < q \land \neg(f(p) > f(q))\]
\[pc = 5 \Rightarrow \top\]
\[pc = 6 \Rightarrow \top\]
\[pc = 7 \Rightarrow p = q\]

(e) Invariants

\[p = 1;\]
\[q = n;\]
\[m = 0;\]

while(p<q) {
    if(f[p]>f[q]) {
        p = c_plus(p,1);
    } else {
        q = c_minus(q,1);
    }
}
\[m = f[p];\]

(f) Program Code

\[pc0: \text{if}(!{(p<q)}) \text{goto pc7};\]
\[pc1: \text{if}(!{(f[p]>f[q])}) \text{goto pc4};\]
\[pc2: p = c\_plus(p,1);\]
\[pc3: \text{goto pc5};\]
\[pc4: q = c\_minus(q,1);\]
\[pc5: ;\]
\[pc6: \text{goto pc0};\]
\[pc7: m = f[p];\]

(g) Flattened Code
B.4. Array Partitioning

```
init:
begin
  j := 0
  k := 0
  g := f
end

final:
when
  j = n
then
  SKIP
end
```

```
progress_1:
when
  j ≠ n
  g(j + 1) > x
then
  j := j + 1
end

progress_2:
when
  j ≠ n
  g(j + 1) ≤ x
  k = j
then
  k := k + 1
  j := j + 1
end

progress_3:
when
  j ≠ n
  g(j + 1) ≤ x
  k ≠ j
then
  k := k + 1
  j := j + 1
  g := g \Leftrightarrow \{k + 1 \mapsto g(j + 1)\} \Leftrightarrow \{j + 1 \mapsto g(k + 1)\}
end
```

(a) Original Events

When translating event `progress_3` we face the problem of sequential composition of parallel assignments. As a solution, we use event parameter `h` for the swapping in `g`. Furthermore we replace `f` and `g` by `a` and `b` to transform the original arrays of the development into arrays that start from 0. The gluing invariant/axiom that we use is as follows.

```
invariants:
  b = \{0 \mapsto 0\} \cup g
```

```
axioms:
  a = \{0 \mapsto 0\} \cup f
```

With data refinement, the variable `f` disappears. We use context extension and subsequently generic instantiation for the constant `g` to disappear.
init :
begin
\begin{align*}
j &:= 0 \\ k &:= 0 \\ b &:= \{ i : i \in c_{upto}(0 \rightarrow n) \land T \mid i \mapsto a(i) \}\end{align*}
end

\begin{tabular}{|l|}
\hline
\textbf{progress\_1} : \\
\textbf{when} \\
\textbf{when} \\
\textbf{when} \\
\textbf{when} \\
\textbf{when} \\
\textbf{when} \\
\textbf{when} \\
\hline
\end{tabular}

\begin{tabular}{|l|}
\hline
\textbf{progress\_2} : \\
\textbf{when} \\
\textbf{when} \\
\textbf{when} \\
\textbf{when} \\
\textbf{when} \\
\textbf{when} \\
\hline
\end{tabular}

\begin{tabular}{|l|}
\hline
\textbf{final} : \\
\textbf{when} \\
\textbf{then} \\
\hline
\end{tabular}

(b) Restricted Events
progress_3 :
any h where
    j \neq n
    b(j + 1) \leq x
    k \neq j
    h = b(c_{plus}(k \mapsto 1))
then
    b := b \Leftarrow \{c_{plus}(k \mapsto 1) \mapsto b(c_{plus}(j \mapsto 1))\} \Leftarrow \{c_{plus}(j \mapsto 1) \mapsto h\}
    k := c_{plus}(k \mapsto 1)
    j := c_{plus}(j \mapsto 1)
end

(b) Restricted Events - cont.

init;
0: do(j \neq n)
1: if(b(c_{plus}(j \mapsto 1)) > x)
2: progress_1
3: else
4: if(k = j)
5: progress_2
6: else
7: progress_3
8: fi
9: fi
10: od;
   \{j = n\}
11: final

(c) Schedule
init:
begin
  \( j := 0 \)
  \( k := 0 \)
  \( b := \{ i \in c_{\text{upto}}(0 \rightarrow n) \land (\exists i \in a(i)) \} \)
  \( pc := 0 \)
end

\[
\text{do1}_\text{true}:
\begin{align*}
  &\text{when} \quad pc = 0 \quad j \neq n \\
  &\text{then} \quad pc := pc + 1 
\end{align*}
\]

\[
\text{do1}_\text{false}:
\begin{align*}
  &\text{when} \quad pc = 0 \quad \neg(j \neq n) \\
  &\text{then} \quad pc := 11 
\end{align*}
\]

\[
\text{if2}_\text{true}:
\begin{align*}
  &\text{when} \quad pc = 1 \\
  &\text{then} \quad b(c_{\text{plus}}(j \rightarrow 1)) > x \\
  &\text{then} \quad pc := pc + 1 
\end{align*}
\]

\[
\text{if2}_\text{false}:
\begin{align*}
  &\text{when} \quad pc = 1 \\
  &\text{then} \quad \neg(b(c_{\text{plus}}(j \rightarrow 1)) > x) \\
  &\text{then} \quad pc := 4 
\end{align*}
\]

\[
\text{progress}_1:
\begin{align*}
  &\text{when} \quad pc = 2 \\
  &\text{then} \quad j := c_{\text{plus}}(j \rightarrow 1) \\
  &\text{then} \quad pc := pc + 1 
\end{align*}
\]

\[
\text{if2}_\text{exit}:
\begin{align*}
  &\text{when} \quad pc = 3 \\
  &\text{then} \quad pc := 9 
\end{align*}
\]

\[
\text{if3}_\text{true}:
\begin{align*}
  &\text{when} \quad pc = 4 \\
  &\text{then} \quad k = j \\
  &\text{then} \quad pc := pc + 1 
\end{align*}
\]

\[
\text{if3}_\text{false}:
\begin{align*}
  &\text{when} \quad pc = 4 \\
  &\text{then} \quad \neg(k = j) \\
  &\text{then} \quad pc := 7 
\end{align*}
\]

(d) Scheduled Events
progress_2:
when
  pc = 5
then
  k := c_plus(k \mapsto 1)
  j := c_plus(j \mapsto 1)
  pc := pc + 1
end

if3_exit:
when
  pc = 6
then
  pc := 8
end

progress_3:
any h where
  pc = 7
  h = b(c_plus(k \mapsto 1))
then
  b := b \in \{c_plus(k \mapsto 1) \mapsto b(c_plus(j \mapsto 1))\} \iff \{c_plus(j \mapsto 1) \mapsto h\}
  k := c_plus(k \mapsto 1)
  j := c_plus(j \mapsto 1)
  pc := pc + 1
end

if3_end:
when
  pc = 8
then
  pc := pc + 1
end

if2_end:
when
  pc = 9
then
  pc := pc + 1
end

dol_return:
when
  pc = 10
then
  pc := 0
end

final:
when
  pc = 11
then
  SKIP
end

(d) Scheduled Events - cont.
\( pc = 0 \Rightarrow \top \)
\( pc = 1 \Rightarrow j \neq n \)
\( pc = 2 \Rightarrow j \neq n \land b(c_{\text{plus}}(j \rightarrow 1)) > x \)
\( pc = 3 \Rightarrow \top \)
\( pc = 4 \Rightarrow j \neq n \land (b(c_{\text{plus}}(j \rightarrow 1)) > x) \)
\( pc = 5 \Rightarrow j \neq n \land (b(c_{\text{plus}}(j \rightarrow 1)) > x) \land k = j \)
\( pc = 6 \Rightarrow \top \)
\( pc = 7 \Rightarrow j \neq n \land (b(c_{\text{plus}}(j \rightarrow 1)) > x) \land \neg (k = j) \)
\( pc = 8 \Rightarrow \top \)
\( pc = 9 \Rightarrow \top \)
\( pc = 10 \Rightarrow \top \)
\( pc = 11 \Rightarrow j = n \)

(e) Invariants

\[
\begin{align*}
&j=0; \\
&k=0; \\
&\text{for(int } i=0; \ i<=n; \ i++) \\
&\quad \text{if(true) } \\
&\quad \quad b[i] = a[i]; \\
&\text{} \\
\end{align*}
\]

(f) Program Code

\[
\begin{align*}
&j=0; \\
&k=0; \\
&\text{for(int } i=0; \ i<=n; \ i++) \\
&\quad \text{if(true) } \\
&\quad \quad b[i] = a[i]; \\
&\text{} \\
\end{align*}
\]

\[
\begin{align*}
&\text{while(}j! = n\text{) } \\
&\quad \text{if(}b[c_{\text{plus}}(j,1)] > x\text{) } \\
&\quad \quad j = c_{\text{plus}}(j,1); \\
&\quad \text{else } \\
&\quad \quad \text{if(}k==j\text{) } \\
&\quad \quad \quad k = c_{\text{plus}}(k,1); \\
&\quad \quad \quad j = c_{\text{plus}}(j,1); \\
&\quad \quad \text{else } \\
&\quad \quad \quad \{ \\
&\quad \quad \quad \quad \text{int } h = b[c_{\text{plus}}(k,1)]; \\
&\quad \quad \quad \quad b[c_{\text{plus}}(k,1)] = b[c_{\text{plus}}(j,1)]; \\
&\quad \quad \quad \quad b[c_{\text{plus}}(j,1)] = h; \\
&\quad \quad \quad \quad k = c_{\text{plus}}(k,1); \\
&\quad \quad \quad \quad j = c_{\text{plus}}(j,1); \\
&\quad \quad \quad \} \\
&\quad \} \\
&\text{pc0: if(}!(j! = n)\text{) goto pc11; } \\
&\text{pc1: if(}!(b[c_{\text{plus}}(j,1)] > x)\text{) goto pc4; } \\
&\text{pc2: } j = c_{\text{plus}}(j,1); \\
&\text{pc3: goto pc9; } \\
&\text{pc4: if(}!(k==j)\text{) goto pc7; } \\
&\text{pc5: } k = c_{\text{plus}}(k,1); \\
&\text{pc6: goto pc8; } \\
&\text{pc7: } \\
&\text{pc8: ; } \\
&\text{pc9: ; } \\
&\text{pc10: goto pc0; } \\
&\text{pc11: ; } \\
\end{align*}
\]

(g) Flattened Code
B.5. Simple Sorting

\begin{align*}
\text{init :} & \quad \text{final :} & \quad \text{progress :} \\
\text{begin} & \quad \text{when} & \quad \text{when} \\
g \leftarrow f & \quad k = n & \quad k \neq n, \quad j = n \\
k \leftarrow 1 & \quad \text{then} & \quad \text{then} \\
l \leftarrow 1 & \quad \text{skip} & \quad g \leftarrow \{k \mapsto g(k)\} \cup \{l \mapsto g(l)\} \\
j \leftarrow 1 & \quad \text{end} & \quad k \leftarrow k + 1 \\
\end{align*}

\begin{align*}
\text{prog1 :} & \quad \text{prog2 :} \\
\text{when} & \quad \text{when} \\
k \neq n & \quad k \neq n \\
j \neq n & \quad j \neq n \\
g(l) \leq g(j + 1) & \quad g(j + 1) < g(l) \\
\text{then} & \quad \text{then} \\
j \leftarrow j + 1 & \quad j \leftarrow j + 1 \\
\text{end} & \quad l \leftarrow j + 1 \\
\end{align*}

(a) Original Events

Again, we replace the arrays $f$ and $g$ with the arrays $a$ and $b$ that start from 0.
init:
begin
    \( b := \{ i \cdot i \in c\_upto(0 \mapsto n) \land T \mid i \mapsto a(i) \} \)
    \( k := 1 \)
    \( l := 1 \)
    \( j := 1 \)
end

progress:
any \( h \) where
\( k \neq n \)
\( j = n \)
\( h = b(k) \)
then
    \( b := b \leftarrow \{ k \mapsto b(l) \} \leftarrow \{ l \mapsto h \} \)
    \( j := c\_plus(k \mapsto 1) \)
    \( l := c\_plus(k \mapsto 1) \)
    \( k := c\_plus(k \mapsto 1) \)
end

prog1:
when
\( k \neq n \)
\( j \neq n \)
\( b(l) \leq b(c\_plus(j \mapsto 1)) \)
then
    \( j := c\_plus(j \mapsto 1) \)
end

prog2:
when
\( k \neq n \)
\( j \neq n \)
\( b(c\_plus(j \mapsto 1)) < b(l) \)
then
    \( l := c\_plus(j \mapsto 1) \)
    \( j := c\_plus(j \mapsto 1) \)
end

(b) Restricted Events
init;

0: do($k \neq n$)
1: do($j \neq n$)
2: if($b(l) \leq b(c\_plus(j \rightarrow 1))$
3: prog_1
4: else
5: prog_2
6: fi
7: od;
8: \{ $k \neq n, j = n$ \}
9: progress
10: \{ $k = n$ \}
11: final

(c) Schedule

\[
\text{init : begin } \\
\quad b := \{ i \cdot i \in c\_upto(0 \mapsto n) \land T | i \mapsto a(i) \} \\
\quad k := 1 \\
\quad l := 1 \\
\quad j := 1 \\
\quad pc := 0 \\
\text{end}
\]

\[
\text{do1\_true : begin } \\
\quad \text{when } pc = 0 \\
\quad \text{when } k \neq n \\
\quad \text{then } pc := pc + 1 \\
\text{end}
\]

\[
\text{do1\_false : begin } \\
\quad \text{when } pc = 0 \\
\quad \text{when } k \neq n \\
\quad \text{then } pc := 10 \\
\text{end}
\]

\[
\text{do2\_true : begin } \\
\quad \text{when } pc = 1 \\
\quad \text{when } j \neq n \\
\quad \text{then } pc := pc + 1 \\
\text{end}
\]

\[
\text{do2\_false : begin } \\
\quad \text{when } pc = 1 \\
\quad \text{when } j \neq n \\
\quad \text{then } pc := 7 \\
\text{end}
\]

(d) Scheduled Events
if3_true:
when
  pc = 2
  b(l) ≤ b(c_plus(j → 1))
then
  pc := pc + 1
end

if3_false:
when
  pc = 2
  ¬(b(l) ≤ b(c_plus(j → 1)))
then
  pc := 5
end

prog1:
when
  pc = 3
then
  j := c_plus(j → 1)
  pc := pc + 1
end

if3_exit:
when
  pc = 4
then
  pc := 6
end

prog2:
when
  pc = 5
then
  l := c_plus(j → 1)
  j := c_plus(j → 1)
  pc := pc + 1
end

if3_end:
when
  pc = 6
then
  pc := pc + 1
end

d02_return:
when
  pc = 7
then
  pc := 1
end

do1_return:
when
  pc = 9
then
  pc := 0
end

progress:
any h where
  pc = 8
  h = b(k)
then
  b := b ∪ {k → b(l)} ∪ {l → h}
  j := c_plus(k → 1)
  l := c_plus(k → 1)
  k := c_plus(k → 1)
  pc := pc + 1
end

final:
when
  pc = 10
then
  pc := pc + 1
end

(d) Scheduled Events - cont.
\[ pc = 0 \Rightarrow \top \]
\[ pc = 1 \Rightarrow k \neq n \]
\[ pc = 2 \Rightarrow k \neq n \land j \neq n \]
\[ pc = 3 \Rightarrow k \neq n \land j \neq n \land b(l) \leq b(c \_plus(j \mapsto 1)) \]
\[ pc = 4 \Rightarrow k \neq n \]
\[ pc = 5 \Rightarrow k \neq n \land j \neq n \land \neg(b(l) \leq b(c \_plus(j \mapsto 1))) \]
\[ pc = 6 \Rightarrow k \neq n \]
\[ pc = 7 \Rightarrow k \neq n \]
\[ pc = 8 \Rightarrow k \neq n \land j = n \]
\[ pc = 9 \Rightarrow \top \]
\[ pc = 10 \Rightarrow k = n \]

(e) Invariants

```java
for(int i=0; i<=n; i++) {
    if(true)
        b[i]=a[i];
}
k=1;
l=1;
j=1;
while(k!=n) {
    while(j!=n) {
        if(b[l]<=b[c\_plus(j,1)]) {
            j=c\_plus(j,1);
        } else {
            l=c\_plus(j,1);
            j=c\_plus(j,1);
        }
    }
    int h=b[k];
    b[k]=b[l];
    b[l]=h;
    j=c\_plus(k,1);
    l=c\_plus(k,1);
    k=c\_plus(k,1);
}
}
```

(f) Program Code

```java
for(int i=0; i<=n; i++) {
    if(true)
        b[i]=a[i];
}
k=1;
l=1;
j=1;
while(k!=n) {
    if(!(k!=n)) goto pc10;
    while(j!=n) {
        if(!(j!=n)) goto pc7;
        if(!(b[l]<=b[c\_plus(j,1)])) goto pc5;
        j=c\_plus(j,1);
        goto pc6;
    }
    int h=b[k];
    b[k]=b[l];
    b[l]=h;
    j=c\_plus(k,1);
    l=c\_plus(k,1);
    k=c\_plus(k,1);
    goto pc0;
}
```

(g) Flattened Program
B.6. Array Reversing

(a) Original Events

| init | begin | \( i := 1 \) | \( j := n \) | \( g := f \) | end |
| final | begin | when \( j \leq i \) | then | SKIP | end |
| progress | when \( i < j \) | then | \( g := g \leftarrow \{i \rightarrow g(j)\} \leftarrow \{j \rightarrow g(i)\} \) | \( i := i + 1 \) | \( j := j - 1 \) | end |

(b) Restricted Events

| init | begin | \( i := 1 \) | \( j := n \) | \( b := \{k \cdot k \in c\_upto(0 \mapsto n) \land T \mid k \mapsto a(k)\} \) | end |
| progress | when | \( j \leq i \) | then | SKIP | end |
| any | \( h \) | where \( i < j \) | \( h = b(i) \) | then | \( b := b \leftarrow \{i \mapsto b(j)\} \leftarrow \{j \mapsto h\} \) | \( i := c\_plus(i \mapsto 1) \) | \( j := c\_minus(j \mapsto 1) \) | end |
init;
0 \textbf{do}(i < j) \\
1 \textbf{progress} \\
2 \textbf{od}; \\
{ j \leq i} \\
3 \textbf{final} \\

(c) Schedule

\begin{align*}
\text{init :} \\
\text{begin} \\
i := 1 \\
j := n \\
b := \{k \cdot k \in c\_upto(0 \mapsto n) \land T \mid k \mapsto a(k)\} \\
pc := 0 \\
\text{end}
\end{align*}

\begin{align*}
\text{dol\_true :} \\
\text{when} \\
\text{pc} = 0 \\
\text{i} < j \\
\text{then} \\
\text{pc} := \text{pc} + 1 \\
\text{end}
\end{align*}

\begin{align*}
\text{dol\_alse :} \\
\text{when} \\
\text{pc} = 0 \\
\neg(i < j) \\
\text{then} \\
\text{pc} := 3 \\
\text{end}
\end{align*}

\begin{align*}
\text{progress :} \\
\text{any} \ h \ \text{where} \\
\text{pc} = 1 \\
h = b(i) \\
\text{then} \\
b := b \cup \{i \mapsto b(j)\} \cup \{j \mapsto h\} \\
i := c\_\text{plus}(i \mapsto 1) \\
j := c\_\text{minus}(j \mapsto 1) \\
\text{pc} := \text{pc} + 1 \\
\text{end}
\end{align*}

\begin{align*}
\text{dol\_return :} \\
\text{when} \\
\text{pc} = 2 \\
\text{then} \\
\text{pc} := 0 \\
\text{end}
\end{align*}

\begin{align*}
\text{final :} \\
\text{when} \\
\text{pc} = 3 \\
\text{then} \\
\text{pc} := \text{pc} + 1 \\
\text{end}
\end{align*}

(d) Scheduled Events
\( pc = 0 \Rightarrow T \)
\( pc = 1 \Rightarrow i < j \)
\( pc = 2 \Rightarrow T \)
\( pc = 3 \Rightarrow j \leq i \)

(e) Invariants

\[
i = 1; \\
j = n; \\
\text{for(int } k = 0; k \leq n; k++\{ \\
\quad \text{if(true) } \\
\quad \quad b[k] = a[k]; \\
\}\}
\]

while\((i < j)\) {
\{
\quad \text{int } h = b[i]; \\
\quad b[i] = b[j]; \\
\quad b[j] = h; \\
\quad i = \text{c plus}(i,1); \\
\quad j = \text{c minus}(j,1); \\
\}\}

(f) Program

\[
i = 1; \\
j = n; \\
\text{for(int } k = 0; k \leq n; k++\{ \\
\quad \text{if(true) } \\
\quad \quad b[k] = a[k]; \\
\}\}
\]

while\((i < j)\) {
\{
\quad \text{int } h = ga[i]; \\
\quad ga[i] = ga[j]; \\
\quad ga[j] = h; \\
\quad i = \text{c plus}(i,1); \\
\quad j = \text{c minus}(j,1); \\
\}\}

(g) Flattened Program

\[
i = 1; \\
j = n; \\
\text{for(int } k = 0; k \leq n; k++\{ \\
\quad \text{if(true) } \\
\quad \quad ga[k] = fa[k]; \\
\}\}
\]

while\((i < j)\) {
\{
\quad \text{int } h = ga[i]; \\
\quad ga[i] = ga[j]; \\
\quad ga[j] = h; \\
\quad i = \text{c plus}(i,1); \\
\quad j = \text{c minus}(j,1); \\
\}\}

pc0: if(!\((i < j)\)) goto pc3;
pc1: {
\quad \text{int } h = ga[i]; \\
\quad ga[i] = ga[j]; \\
\quad ga[j] = h; \\
\quad i = \text{c plus}(i,1); \\
\quad j = \text{c minus}(j,1); \\
\}\}

pc2: goto pc0;
pc3: ;
B.7. Numerical Program Computing the Square Root

<table>
<thead>
<tr>
<th>init:</th>
<th>final:</th>
<th>progress:</th>
</tr>
</thead>
<tbody>
<tr>
<td>begin</td>
<td>when</td>
<td>when</td>
</tr>
<tr>
<td></td>
<td>$n &lt; a$</td>
<td>$a \leq n$</td>
</tr>
<tr>
<td></td>
<td>then</td>
<td>then</td>
</tr>
<tr>
<td></td>
<td>SKIP</td>
<td>$r := r + 1$</td>
</tr>
<tr>
<td>end</td>
<td></td>
<td>$a := a + b$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$b := b + 2$</td>
</tr>
</tbody>
</table>

(a) Original Events

<table>
<thead>
<tr>
<th>init:</th>
<th>final:</th>
<th>progress:</th>
</tr>
</thead>
<tbody>
<tr>
<td>begin</td>
<td>when</td>
<td>when</td>
</tr>
<tr>
<td></td>
<td>$n &lt; a$</td>
<td>$a \leq n$</td>
</tr>
<tr>
<td></td>
<td>then</td>
<td>then</td>
</tr>
<tr>
<td></td>
<td>SKIP</td>
<td>$r := c_\text{plus}(r \mapsto 1)$</td>
</tr>
<tr>
<td>end</td>
<td></td>
<td>$a := c_\text{plus}(a \mapsto b)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$b := c_\text{plus}(b \mapsto 2)$</td>
</tr>
</tbody>
</table>

(b) Restricted Events

In this example, the order of the actions in event progress is such that no auxiliary variable is necessary.

```plaintext
init;
0: do($a \leq n$)
1: progress
2: od;
   { $n < a$ }
3: final
```

(c) Schedule
\( pc = 0 \Rightarrow \top \)
\( pc = 1 \Rightarrow a \leq n \)
\( pc = 2 \Rightarrow \top \)
\( pc = 3 \Rightarrow n < a \)

(d) Invariants

\[
\text{init} : \\
\text{begin} \\
r := 0 \\
a := 1 \\
b := 3 \\
ec{pc} := 0 \\
\text{end}
\]

\[
\text{dol\_true} : \\
\text{when} \\
ec{pc} = 0 \\
\text{then} \\
ec{pc} := \ec{pc} + 1 \\
\text{end}
\]

\[
\text{dol\_false} : \\
\text{when} \\
ec{pc} = 0 \\
\neg (a \leq n) \\
\text{then} \\
ec{pc} := 3 \\
\text{end}
\]

(e) Scheduled Events

```
r=0; 
a=1; 
b=3; 
while(a<=n) { 
    pc0: if(!(a<=n)) goto pc3; 
    r=c\_plus(r,1); 
    a=c\_plus(a,b); 
    b=c\_plus(b,2); 
} 

pc1: r=c\_plus(r,1); 
    a=cp(a,b); 
    b=c\_plus(b,2); 

pc2: goto pc0; 
    pc3: ;
```

(f) Program

```
r=0; 
a=1; 
b=3; 
while(a<=n) { 
    pc0: if(!(a<=n)) goto pc3; 
    r=c\_plus(r,1); 
    a=c\_plus(a,b); 
    b=c\_plus(b,2); 
} 

pc1: r=c\_plus(r,1); 
    a=cp(a,b); 
    b=c\_plus(b,2); 

pc2: goto pc0; 
    pc3: ;
```

(g) Flattened Program
### B.8. The Common Minimum of Two Sorted Arrays

<table>
<thead>
<tr>
<th>init :</th>
<th>progressF :</th>
</tr>
</thead>
<tbody>
<tr>
<td>begin</td>
<td>when</td>
</tr>
<tr>
<td>$m \in \mathbb{Z}$</td>
<td>$f(i) &lt; g(j)$</td>
</tr>
<tr>
<td>$i := 0$</td>
<td>then</td>
</tr>
<tr>
<td>$j := 0$</td>
<td>$i := i + 1$</td>
</tr>
<tr>
<td>end</td>
<td>end</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>progressG :</th>
<th>minimum :</th>
</tr>
</thead>
<tbody>
<tr>
<td>when</td>
<td>when</td>
</tr>
<tr>
<td>$f(i) &gt; g(j)$</td>
<td>$f(i) = g(j)$</td>
</tr>
<tr>
<td>then</td>
<td>then</td>
</tr>
<tr>
<td>$j := j + 1$</td>
<td>$m := f(i)$</td>
</tr>
<tr>
<td>end</td>
<td>end</td>
</tr>
</tbody>
</table>

(a) Original Events
init:
begin
    m := 0
    i := 0
    j := 0
end

progressF :
when
    f(i) < g(j)
then
    i := c_plus(i \rightarrow 1)
end

progressG :
when
    f(i) > g(j)
then
    j := c_plus(j \rightarrow 1)
end

minimum :
when
    f(i) = g(j)
then
    m := f(i)
end

(b) Restricted Events

init;
0: do(f(i) \neq g(j))
1: if(f(i) < g(j))
2:   progressF
3: else
4:   progressG
5: fi
6: od;
   \{f(i) = g(j)\}
7: minimum

(c) Schedule
init:
begin
  $m := 0$
  $i := 0$
  $j := 0$
  $pc := 0$
end

$$\text{do1}$$ _true:
when
  $pc = 0$
  $f(i) \neq g(j)$
then
  $pc := pc + 1$
end

$$\text{do1}$$ _false:
when
  $pc = 0$
  $\neg (f(i) \neq g(j))$
then
  $pc := 7$
end

if2_true:
when
  $pc = 1$
  $f(i) < g(j)$
then
  $pc := pc + 1$
end

if2_false:
when
  $pc = 1$
  $\neg ((i) < g(j))$
then
  $pc := 4$
end

progressF:
when
  $pc = 2$
then
  $i := c_\text{plus}(i \mapsto 1)$
  $pc := pc + 1$
end

progressG:
when
  $pc = 4$
then
  $j := c_\text{plus}(j \mapsto 1)$
  $pc := pc + 1$
end

if2_exit:
when
  $pc = 3$
then
  $pc := 5$
end

if2_end:
when
  $pc = 5$
then
  $pc := pc + 1$
end

$$\text{do1}$$ _return:
when
  $pc = 6$
then
  $pc := 0$
end

minimum:
when
  $pc = 7$
then
  $m := f(i)$
  $pc := pc + 1$
end

(c) Scheduled Events
\[ \text{Invariants} \]

\begin{align*}
\text{pc } = 0 & \Rightarrow \top \\
\text{pc } = 1 & \Rightarrow f(i) \neq g(j) \\
\text{pc } = 2 & \Rightarrow f(i) \neq g(j) \land f(i) < g(j) \\
\text{pc } = 3 & \Rightarrow \top \\
\text{pc } = 4 & \Rightarrow f(i) \neq g(j) \land \neg(f(i) < g(j)) \\
\text{pc } = 5 & \Rightarrow \top \\
\text{pc } = 6 & \Rightarrow \top \\
\text{pc } = 7 & \Rightarrow f(i) = g(j) \\
\end{align*}

\text{(e) Invariants}

\begin{verbatim}
\text{m=0; i=0; j=0; pc0: if(!(f[i]!=g[j])) goto pc7; pc1: if(!(f[i]<g[j])) goto pc4; pc2: i=c_plus(i,1); pc3: goto pc5; pc4: j=c_plus(j,1); pc5: ; pc6: goto pc0; pc7: m=f[i];}
\end{verbatim}

\text{(f) Program}

\text{(g) Flattened Program}