System Modelling & Design
Using Event-B

Ken Robinson
Contents

1 Book Layout and Guide

2 System Modelling and Design
   2.1 Software Engineering, not Programming
   2.2 Mathematics, not Magic
   2.3 Background and Timeline

3 Contexts, Machines, State, Events, Proof and Refinement
   3.1 Machines
      3.1.1 CoffeeClub
         3.1.1.1 Piggybank Requirements
      3.1.2 Proof Obligations: Sequent representation
      3.1.3 Proof Obligations for CoffeeClub
      3.1.4 What you need to know to discharge POs
   3.2 Refinement
      3.2.1 Refinement machine
      3.2.2 Refinement rules
      3.2.3 Refinement of the CoffeeClub
      3.2.4 Context Members
      3.2.5 Refinement MemberShip
      3.2.6 Proof Obligations
      3.2.7 What are the new POs?
   3.3 Animation
      3.3.1 Animation Constraints

4 Refinement
   4.1 What is Refinement?
   4.2 An example: Square root
      4.2.1 Square Root Definition
      4.2.2 Refinement: How we make progress
      4.2.3 “Opening Up” an event
         4.2.3.1 Parameters
         4.2.3.2 Convergence and Variants
      4.2.4 Improving Improve
         4.2.4.1 Witness and the With clause
      4.2.5 Refining SquareRootR2
      4.2.6 Refining SquareRootR3
      4.2.7 An alternative refinement to SquareRootR4
         4.2.7.1 Exercise
## CONTENTS

4.3 Modelling a parametric argument ........................................... 33
  4.3.1 Remainder of development .............................................. 35
  4.3.2 Converting to programming code ..................................... 37

5 Invariants: Specifying Safety .................................................. 39
  5.1 Simple Traffic Lights ...................................................... 39
    5.1.1 Simplifying and Generalising ...................................... 42
    5.1.2 The Simple TwoWay machine ...................................... 42
  5.2 A Multiway Intersection .................................................. 44
    5.2.1 The Initial Traffic Light model .................................. 45

6 Event-B Semantics ............................................................... 51
  6.1 Semantics in Event B ...................................................... 51
    6.1.1 State Change ................................................................ 51
      6.1.1.1 Before-After Predicates ....................................... 52
    6.1.2 Substitution .................................................................. 52
      6.1.2.1 Other Forms of Substitution ................................... 53
    6.1.3 Contexts ..................................................................... 53
      6.1.3.1 Context Machines: Semantics & Proof Obligations .... 53
      6.1.3.2 Theorems ............................................................. 53
    6.1.4 Machines ..................................................................... 54
      6.1.4.1 Machine POs: Invariant and Theorems .................... 54
      6.1.4.2 Initialisation ......................................................... 54
    6.1.5 Events ........................................................................ 55
      6.1.5.1 Event: Proof Obligations ....................................... 55
      6.1.5.2 Event: Maintaining Invariant ................................. 55
    6.1.6 Machine Refinements ..................................................... 55
      6.1.6.1 Variables and Invariant ......................................... 56
      6.1.6.2 Proof Obligations ................................................ 56
    6.1.7 Refined Events ............................................................. 56
      6.1.7.1 Proof Obligations for Refined Events .................... 56
      6.1.7.2 The Variant and Convergent Events ....................... 56
  6.2 One Point Rule ................................................................. 57

7 Data Refinement: A Queue Model .............................................. 59
  7.1 Context for a queue ............................................................ 59
  7.2 A Queue machine .............................................................. 59
  7.3 A Context that defines an abstract Queue datatype ................. 62
  7.4 A more abstract model ....................................................... 64
  7.5 Changing the data representation ....................................... 67
  7.6 Further refinement ............................................................ 73

8 Lift System Modelling ............................................................... 77
  8.1 Basic Lift ............................................................................ 77
    8.1.1 Lift Context .................................................................. 78
    8.1.2 Basic Lift machine ......................................................... 79
  8.2 Adding Lift Doors .............................................................. 83
    8.2.1 Door Context .............................................................. 83
    8.2.2 Lift Plus Doors ............................................................ 83
    8.2.3 Adding Floor Doors ....................................................... 86
  8.3 Adding Buttons ................................................................. 88
    8.3.1 Buttons inside Lift ........................................................ 88
    8.3.2 Buttons Context .......................................................... 88
CONTENTS

8.3.3 Lift Buttons machine ......................................................... 89
8.3.4 Floor Buttons ................................................................. 94

9 Proof Obligations ................................................................. 95

10 Exercises ................................................................. 97
  10.1 Relations and Functions ......................................................... 97
    10.1.1 Exercises ................................................................. 98
    10.1.2 More on Sets ............................................................. 98
  10.2 A Simple Bank Machine ...................................................... 104
  10.3 Supermarket Model .......................................................... 105
    10.3.1 The Supermarket_ctx context ........................................... 105
    10.3.2 The Supermarket machine .............................................. 105
    10.3.3 Refinement of Supermarket Machine .................................. 107

11 Solutions ................................................................. 109
  11.1 Relations and Functions ..................................................... 109

A Models ................................................................. 117
  A.1 Coffee Club ............................................................... 117

B Proof Obligations .......................................................... 123

C Event-B Concise Summary ..................................................... 125
  C.1 Predicates ........................................................................ 126
  C.2 Sets .............................................................................. 126
    C.2.1 Set predicates ............................................................ 127
  C.3 BOOL and bool ................................................................. 127
  C.4 Numbers ....................................................................... 127
    C.4.1 Number predicates ......................................................... 127
  C.5 Relations ....................................................................... 128
    C.5.1 Iteration and Closure .................................................... 128
    C.5.2 Functions ................................................................. 129
  C.6 Models ....................................................................... 129
    C.6.1 Events ..................................................................... 129
    C.6.2 Actions .................................................................... 129

D Rodin ................................................................. 131
  D.1 The Rodin platform ........................................................ 131

Bibliography .............................................................. 133

Index .............................................................. 135
Chapter 1

Book Layout and Guide

This book contains Event-B examples designed to be used with the Rodin toolkit [8]. The source text for the models is embedded in the book text, for example:

\textbf{MACHINE} CoffeeClub

\textbf{VARIABLES} piggybank \hspace{1cm} This machine state is represented by the variable, \textit{piggybank}, denoting a supply of money for the coffee club.

\textbf{INVARINTS} \hspace{1cm} inv1: \hspace{1cm} \textit{piggybank} \in \mathbb{N} \hspace{1cm} \text{piggybank must be a natural number}

\textbf{END} CoffeeClub
Chapter 2

System Modelling and Design

This book is concerned with the verification of system design using system modelling. The modelling will be carried out in a rigorous way that allows us to quantitatively verify a design against the requirements.

It is important to understand that in order to explore a design we will be concerned with events: any parameters that are involved, when the event can occur (fire), and what state changes occur as a consequence of the event firing.

*Only one event fires at a time.*

Generally, designs will be presented through many layers of abstraction called refinements. Refinements allow us to introduce more details of the design and to expose new levels of a system with extra functionality.

2.1 Software Engineering, not Programming

While the modelling we will discuss is not restricted to software systems, there will be some parts of systems that will be implemented as software. We are concerned to emphasise an engineering approach to system design in general and to software system design in particular, in contrast to the more common code and test approach. Due to the discrete nature of digital computers, testing of software can be particularly weak due to the lack of ability to interpolate or extrapolate on test results. Traditional engineering disciplines generally work in continuous domains that allow interpolation, and maybe extrapolation. In civil engineering, for example, it will generally be the case that a beam that does not fail under a 100 tonne load will not fail under a 50 tonne load. There is no similar expectation for software. This is not an argument against testing, it is an argument against non-rigorous verification.

It's also worth noting that software is intrinsically unstable in the following sense. The physical implementation of a conventional engineering design can never be exact: components of a civil engineering structure or an electrical engineering device will never be precisely as specified in the design. What then generally happens is that the structure will distort slightly and reach a stable equilibrium configuration. As the reader with any software experience will be well aware, software —generally— does not behave in that way: if part of a software implementation is not exact —precisely meeting its requirements— then the software will most probably collapse. That is, it is unstable. Hence, *near enough is never good enough.* **Note:** any error recovery strategy must be explicitly added to the software; it is not provided automatically by the environment.

Engineering design should be rigorous, and given the above observations it would seem that this especially needs to be the case for software design. Ironically this is often —perhaps usually— not the case.
The approach adopted in this book will emphasise rigorous design, meaning that a design will be subjected to a rigorous, mathematical quantification of the required behaviour of a system and efforts will be made to ensure that the design does satisfy the requirements. Models will generally be developed in layers, each successive layer being a refinement of the predecessor layer. This is done to enable the correctness of the model to follow from the principle of *correctness by construction* [6].

### 2.2 Mathematics, not Magic

We will be using mathematics, as is perfectly normal for other engineering disciplines. Because of the discrete nature of the descriptions that we need to represent we will be using set theory and logic. Verification will involve the use of proof. To assist with proof we will use theorem provers. In many areas this type of design has been referred to as *formal methods*. We will not use that term; we prefer simply *mathematics*. In particular we wish to avoid any suggestion that *proof* equals *correct* or, even more extreme, that because our designs have been proved they can never fail. We recognise that any engineering design can fail, however the designer should be aware of the conditions under which it may fail.

Requirements need to be interpreted and then quantified in order to be able to reason about the realisation of the requirements. For any system, our objective will be to give a rigorous statement of our assumptions and rigorous arguments that our design satisfies the requirements for the system.

### 2.3 Background and Timeline

The following is an abbreviated timeline of important contributions to the understanding of computer programs.

- **1960** John Backus & Peter Naur [5], *Backus-Naur Form (BNF) for specifying syntax*
- **1967** Robert Floyd [9], *Assigning Meanings to Programs*
- **1969** CAR Hoare [12], *An Axiomatic Basis for Programming*
- **1976** Edsger Dijkstra [6], *Correct by Construction*
- **1980** Cliff Jones [13] [14], *VDM*
- **1983** Niklaus Wirth [24], *Stepwise Refinement*
- **1987** Ralph Back [3] [4], *A calculus of refinement for program for program derivation*
- **1987** Ian Hayes [11], *Z case studies*
- **1990** Carroll Morgan [15], *Specification statement*
- **1996** Jean-Raymond Abrial [1], *Classical B: Assigning Programs to Meanings*
- **1989** Michael Spivey [17] [18], *Z*
- **2010** Jean-Raymond Abrial [2], *System and Software Engineering*
Chapter 3

Contexts, Machines, State, Events, Proof and Refinement

This chapter will explore a very simple model in order to gain some familiarity with modelling using Event-B.

The model is very elementary, but it will introduce many aspects of modelling in Event-B including a very simple —and probably unexpected— instance of refinement. This will throw into relief one basic aspect of refinement.

The basic concepts of Event-B will be introduced in this chapter, and the reader is encouraged to install Rodin —the Event-B toolkit, see [8]— and copy the model developed in this chapter, as your first exercise.

Models in Event-B are described in terms of contexts and machines

Contexts define constants that are either numeric or sets. Within a context, constants are declared and their properties and relationships are defined by axioms and theorems. Axioms describe properties that cannot be derived from other axioms. Theorems describe properties that are expected to be able to be derived from the axioms.

Machines: define dynamic behaviour. A machine may see one or more contexts and have a state and events. The state is represented by variables, whose types and behaviour are defined by invariants and theorems. Events model “things that may happen” in the context of the machine. An event is represented by parameters, which are simply symbolic names for values, guards, which express the conditions of the state and parameters under which the event may fire; and actions, which describe the change of state that occurs when the event does fire. Multiple actions are evaluated concurrently. Only one event fires at any one time, and if more than one event is enabled —all guards satisfied (TRUE)— then the event that will fire is chosen nondeterministically.

3.1 Machines

It is important to understand that machines should not be automatically thought of as software programs —although they might be implemented by software. The machine models a state and the events represent behaviour that could occur: the conditions that must apply if an event is to fire; and the effect the event has on the state. All communication occurs through the state. As such, a machine gives a representation of possible behaviours of some system.
3.1.1 CoffeeClub

The elementary description of machines will be illustrated with a simple running example of a coffee club. We will introduce a machine that will be used to model some of the requirements of the coffee club.

3.1.1.1 Piggybank Requirements

For the coffee club we require a piggybank that stores money used by the coffee club.

REQ1: a money bank for storing and reclaiming finite, non-negative funds for a coffee club;
REQ2: an operation for adding money to the money bank;
REQ3: an operation for removing money from the money bank; cannot remove more than money bank contains.

MACHINE CoffeeClub

VARIABLES
piggybank The machine state is represented by the variable, piggybank, denoting the money bank for the coffee club.

INVARIANTS
inv1: piggybank ∈ \mathbb{N} REQ1: piggybank must not be negative

The invariants specify the properties that the variables (the state) must satisfy before and after every event, excepting the initialisation where the invariants must be satisfied after the initialisation.

Notation
\begin{align*}
\text{math} & : \text{ascii} \\
\in & : \text{set membership} \\
\mathbb{N} & : \text{NAT the set of natural numbers = non-negative integers}
\end{align*}

EVENTS

Events model what can happen in the machine; the conditions under which they can happen; and how the state of the machine is changed.

Initialisation

Initialisation is a distinguished event that occurs once only, before any other event. This event initialises the machine’s variables to a set of values that establishes the invariant. Remember that the variables do not have any value before initialisation.

THEN
act1: piggybank := 0 Could initialise piggybank to any natural number

END

FeedBank ≡ REQ2: adding to piggybank
ANY
amount amount to be added to piggybank

WHERE

grd1: amount ∈ N1 to prevent the event firing uselessly if amount = 0

THEN

act1: piggybank := piggybank + amount

END

Notation

math ascii
:= := "becomes equal to": x := e means assign to the variable x the value of the expression e

N1 NAT1 the set of non-zero natural numbers

RobBank ≜ REQ3: removing money from piggybank

ANY

amount

WHERE

grd1: amount ∈ 1..piggybank The amount must not exceed the contents of piggybank

THEN

act1: piggybank := piggybank – amount

END

Notation

math ascii
m..n m..n denotes the subrange or interval \{i | i ≥ m ∧ i ≤ n\}, where m and n are integers. If m > n then the set is empty.

< = less than or equal

≥ = greater than or equal

\{i | P\} \{i | P\} set comprehension: the set of all values of i that satisfy P

END CoffeeClub

3.1.2 Proof Obligations: Sequent representation

\[ \text{hypotheses} \vdash \text{goal} \]

Figure 3.1: Sequent representation of PO

As the specification of the model is expressed in mathematics it is possible to generate checks to show that the behaviour of the model is consistent with the formal constraints of the model. To achieve this
the Event-B workbench, Rodin, generates proof obligations (PO) that can be checked with a prover, or even verified visually. There are many classes of POs, (see Appendix B for a discussion of all types of POs).

Proof obligations will be represented by a sequent having the form shown in 3.1. The meaning of the PO shown in 3.1 is

\[ \text{the truth of the hypotheses leads to the truth of the goal.} \]

The symbol ‘\(\vdash\)’ is sometimes called stile or turnstile. Note:

1. If any of the hypotheses is ‘\(\bot\)’ then any goal is trivially established.
2. If the hypotheses are identically ‘\(\top\)’ then the hypotheses will be omitted.

<table>
<thead>
<tr>
<th>Notation</th>
<th>math</th>
<th>ascii</th>
</tr>
</thead>
<tbody>
<tr>
<td>‘(\top)’</td>
<td>true</td>
<td>Boolean true</td>
</tr>
<tr>
<td>‘(\bot)’</td>
<td>false</td>
<td>Boolean false</td>
</tr>
</tbody>
</table>

**Proof in Rodin**

When discharging a proof in Rodin, three views are available: hypotheses, goal and proof control that provides access to a range of provers and also tactics for manual proof.

### 3.1.3 Proof Obligations for CoffeeClub

CoffeeClub is a very simple model and the POs are correspondingly simple. It is very easy to see that the POs are satisfiable without resort to a theorem prover. As a consequence the POs are easily discharged automatically by the provers in the Rodin tool.

The following POs are generated for the above machine.

*Notice that all POs concerned with maintenance of INV1 are verifying that REQ1 is satisfied.*

**INITIALISATION/inv1/INV:**

\[ \vdash 0 \in \mathbb{N} \]

Assuming nothing, (\(\top\) or true), show that 0 is an element of the set of natural numbers.

Verifying that the initialisation, \(\text{piggybank} := 0\), establishes the invariant \(\text{piggybank} \in \mathbb{N}\).

**FeedBank/inv1/INV:**

\[
\begin{align*}
\text{piggybank} & \in \mathbb{N} \\
\text{amount} & \in \mathbb{N} \\
\vdash & \text{piggybank} + \text{amount} \in \mathbb{N}
\end{align*}
\]

Verifying that the actions of FeedBank,

\(\text{piggybank} := \text{piggybank} + \text{amount}\),

maintains the invariant,

\(\text{piggybank} \in \mathbb{N}\).

This is clearly true as both \(\text{piggybank}\) and \(\text{amount}\) are natural numbers.
RobBank/inv1/INV: $\text{piggybank} \in \mathbb{N} \quad \text{amount} \in 1..\text{piggybank} \quad \vdash \text{piggybank} - \text{amount} \in \mathbb{N}$

Similar to the preceding POs, but this time verifying that

$$\text{piggybank} \in \mathbb{N}$$

is maintained by the action

$$\text{piggybank} := \text{piggybank} - \text{amount}.$$ 

This is not quite so simple, but the guard

$$\text{amount} \leq \text{piggybank}$$

ensures the invariant is maintained.

### 3.1.4 What you need to know to discharge POs

The first mistake that many people make when faced with discharging a PO, is believing that there is some other information they need. While it may turn out that some extra information is required, it should be appreciated that the information presented as in the above POs is “complete”. Complete that is, excepting the axioms relating to numbers, predicates and logic. It is very important to understand that the consequent should be proveable from the given hypotheses; there is nothing else in the form of a hypothesis that should be required.

**Proof is syntactic**

Discharging a PO is essentially a syntactic exercise: the proof is concerned with symbols and their properties. Again, many coming to this for the first time may try to reason on the basis of what an event is doing to the state, or similar types of reasoning. Such reasoning is almost certainly useless, and counter productive.

If the PO cannot be discharged then there are many cases that must be considered, of which

- the invariants are too strong/weak
- the guards are too weak/strong;
- the actions are inappropriate/incomplete

are some possibilities. The problem may go back to the context, which might be wrong/incomplete, etc.

**But remember . . .**

the exercise is not primarily about discharging the proof obligations, it’s about determining whether the model is consistent with the requirements and internally consistent. The proof obligations might be discharged, but the model may not be what is required.
3.2 Refinement

Refinement is a process that is used to describe any or all of the following changes to a model:

- **extended functionality**: we add more functionality to the model, perhaps modelling the requirements for a system in *layers*;

- **more detail**: we give a finer-grained model of the events. This is often described as moving from the *abstract* to the *concrete*. This form of refinement tends to move from *what* towards *how*;

- **changing state model**: we change the way that the state is modelled, but also describe how the new state models the old state.

In all cases of refinement, the behaviour of the refined machine must be *consistent* with the behaviour of the machine being refined. It is important to appreciate that *consistent* does not mean *equivalent*: the behaviour of the refined machine does not have to be the same, but the behaviour must not contradict the behaviour of the machine being refined. As an example, machines may be—and frequently are—nondeterministic and the refined machine may remove some of the nondeterminism.

### 3.2.1 Refinement machine

The refinement machine consists of:

- **a refined state**: that is logically a new state. The refined state must contain a *refinement relation* that expresses how the refined state models the state being refined. The refined state may contain variables that are syntactically and semantically equivalent to variables in the state of the machine being refined. In that case, the *new* and *old* variables are implicitly related by an equivalence relation.

- **refined events**: that logically refine the events of the refined machine. The refined events are considered to *simulate* the behaviour of the events being refined, where the effects of the refined events are interpreted through the refinement relation.

- **new events**: that add new functionality to the model. The new events must not add behaviour that is inconsistent with the behaviour of the refined machine.

#### Concepts

| **refinement relation** | the refinement relation, which is expressed explicitly or implicitly in the invariant of a refinement, relates the state of the machine being refined to the state of the refinement machine. |

### 3.2.2 Refinement rules

As mentioned above, refinement requires *consistency*. This means that any behaviour of a refined event must be *acceptable behaviour* of the unrefined event in the unrefined model. An informal example of this is:

if at a restaurant you asked for a Pepsi or a Coke, then it would be acceptable for you to be given a Coke, but not acceptable for you to be given a Fanta.
The following rules apply to refinement:

**strengthen guards and invariants:** guards and invariants can be strengthened, provided overall functionality is not reduced;

**nondeterminism can be reduced:** where a model offers choice, then the choice can be reduced—but not increased—in the refinement;

**the state may be augmented by an orthogonal state:** new state variables, whose values do not affect the existing state, may be added.

Consistent with the above, a single event may be refined by multiple events, or conversely, multiple events may be refined a single event.

**New events** As well as refinements of the events of the refined machine, the refinement may introduce new events. The new events *must not* change variables inherited from the state of the refined machine. This is a restriction that recognises that a machine state can be modified only by the events of that machine, or their refinements.

### 3.2.3 Refinement of the CoffeeClub

At the moment the CoffeeClub simply describes a piggybank that models an amount (of money), and events that describe adding to —FeedBank— or taking from —RobBank— the amount modelled by piggybank. We will now model behaviour that describes club-like behaviour for members who want to be able to purchase cups of coffee. The new requirements are:

**REQ4:** a facility for members to join the coffee club; each member has a distinct membership id;

**REQ5:** members have an account that cannot go into debt;

**REQ6:** an operation that enables a member to add money to their account;

**REQ7:** money added to a members account is also added to the club money bank;

**REQ8:** an operation that sets the price for a cup of coffee;

**REQ9:** an operation that enables a member to buy a cup of coffee; the member’s account is reduced by the cost of a cup of coffee;

We will introduce variables members, accounts and coffeeprice and events that correspond to

- **a new member joining the club:** each member of the club is represented by a unique identifier that is arbitrarily chosen from an abstract set MEMBER;

- **a member adding money to their account:** each member has an account, to which they can add “money”;

- **a member buying a cup of coffee:** there will be a variable, coffeeprice, representing the cost of a cup of coffee, and each member can buy a cup of coffee provided they have enough money in their account.

The value of all money added to accounts is added to piggybank.
**Contexts**  Contexts are used in Event-B to define constant values such as *abstract sets, relations, functions*; properties of those constants, called *axioms* and *theorems* expressing properties of the constants that can be deduced from the axioms. The abstract sets are sometimes called *carrier sets*.

**Concepts**

<table>
<thead>
<tr>
<th>term</th>
<th>definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>axiom</td>
<td>an axiom is a property that is asserted; it cannot be proved</td>
</tr>
<tr>
<td>theorem</td>
<td>a theorem is a property that is implied by axioms or invariants; it must be proved</td>
</tr>
</tbody>
</table>

For this refinement we need to define an abstract set *MEMBER*, which we will use as the source of unique identifiers for members. The set is not given a specific size (cardinality), but it is declared to be *finite*, meaning that it does have a size (cardinality) that is a natural number. Sets are potentially infinite, unless declared otherwise. Note that in Event-B *infinity* is not a natural number.

### 3.2.4 Context Members

```
CONTEXT MembersContext
SETS
  MEMBER  REQ4
AXIOMS
  axm1: finite(MEMBER)
END
```

**Notation**

<table>
<thead>
<tr>
<th>math</th>
<th>ascii</th>
</tr>
</thead>
<tbody>
<tr>
<td>finite</td>
<td>finite</td>
</tr>
</tbody>
</table>

finite(S) is \(\top\) if the set \(S\) is finite. This does not require the set to have a specific size, but the set must have a size

### 3.2.5 Refinement MemberShip

The refinement *MemberShip* is clearly aimed at adding new functionality, rather than refining the current functionality. For that reason all events will be displayed in *extended mode*, a mode supported by Rodin.

It should be clear that the events *FeedBank* and *RobBank* are unchanged in the refinement, but *NewMember, SetPrice, BuyCoffee* and *Contribute* are new. To simplify the listing, *FeedBank* and *RobBank* will be omitted here. They can be found in the appendix. It is important to understand that *MemberShip* is what is known as an *algorithmic* refinement. All variables from *CoffeeClub* are included in the refinement and new variables may be added. *MemberShip* is an *extension* of *CoffeeClub*.

```
MACHINE MemberShip
REFINES
  CoffeeClub
SEES
  Members
```
3.2. REFINEMENT

**VARIABLES**
- piggybank
- members: REQ4: the set of current members
- accounts: REQ5: the member accounts
- coffeeprice: REQ8: the price of a cup of coffee

**IN/VARIANTS**
- inv1: \( \text{piggybank} \in \mathbb{N} \)
- inv2: \( \text{members} \subseteq \text{MEMBER} \)  
  REQ4: each member has unique id
- inv3: \( \text{accounts} \in \text{members} \rightarrow \mathbb{N} \)  
  REQ5: each member has an account
- inv4: \( \text{coffeeprice} \in \mathbb{N}^1 \)  
  REQ8: any price other than free!

**Notation**
- \( \subseteq \): subset \( \subseteq \) or equal =
- \( \subset \): strict subset: not equal
- \( \rightarrow \): denotes a total function. If \( f \in X \rightarrow Y \) and \( x \in X \), then \( f(x) \) is defined.
- \( \mapsto \): denotes a partial function. If \( f \in X \mapsto Y \) and \( x \in X \), then \( f(x) \) is not necessarily defined.

**EVENTS**

**Initialisation**: extended \( \equiv \)

**THEN**
- act2: \( \text{members} := \emptyset \) empty set of members
- act3: \( \text{accounts} := \emptyset \) empty set of accounts
- act4: \( \text{coffeeprice} \in \mathbb{N}^1 \) initial coffee price set to arbitrary non-zero value

**Concepts**

In extended mode, only the new parameters, guards and actions are displayed, that is, only the parts of an event that extend the event being refined.

**Notation**
- \( :\in :\): “becomes in”: \( x :\in e \) means assign to \( x \) any element of the set \( s \)
- \( \emptyset \): \( \{\} \) empty set

**SetPrice** \( \equiv \) REQ8

**ANY**
- amount

**WHERE**
- grd1: \( \text{amount} \in \mathbb{N}^1 \)

**THEN**
- act1: \( \text{coffeeprice} := \text{amount} \)

**END**
NewMember ≡ REQ4
ANY
  member
WHERE
  grd1:  member ∈ MEMBER \ members  choose an unused element of MEMBER
THEN
  act1: members := members ∪ {member}
  act2: accounts(member) := 0
END

Contribute ≡ REQ6
ANY
  amount
  member
WHERE
  grd1:  amount ∈ N1
  grd2:  member ∈ members
THEN
  act1: accounts(member) := accounts(member) + amount
  act2: piggybank := piggybank + amount  REQ7
END

BuyCoffee ≡ REQ9
ANY
  member
WHERE
  grd1:  member ∈ members
  grd2:  accounts(member) ≥ coffeeprice
THEN
  act1: accounts(member) := accounts(member) − coffeeprice
END

FeedBank : extended ≡
REFINES
  FeedBank
ANY
WHERE
THEN
3.2. REFINEMENT

RobBank : extended \equiv

REFINES
RobBank
ANY
WHERE
THEN
END

END Membership

3.2.6 Proof Obligations

INITIALISATION:inv1/INV: \[ \vdash 0 \in \mathbb{N} \]

INITIALISATION:inv3/INV: \[ \vdash \emptyset \in \emptyset \rightarrow \mathbb{N} \]

INITIALISATION:inv4/INV: \[ \text{coffeeprice}^\prime \in \mathbb{N}_1 \vdash \text{coffeeprice}^\prime \in \mathbb{N}_1 \]

INITIALISATION:act4/FIS: \[ \vdash \mathbb{N}_1 \neq \emptyset \]

SetPrice/inv4/INV: \[
\begin{align*}
\text{coffeeprice} & \in \mathbb{N}_1 \\
\text{amount} & \in \mathbb{N}_1
\end{align*} \vdash \text{amount} \in \mathbb{N}_1
\]

NewMember/inv3/INV: \[
\begin{align*}
\text{accounts} & \subseteq \text{members} \rightarrow \mathbb{N} \\
\text{member} & \in \text{MEMBER} \setminus \text{members}
\end{align*} \vdash \text{accounts} \left( \text{member} \mapsto 0 \right) \in \text{members} \cup \{ \text{member} \} \rightarrow \mathbb{N}
\]

Contribute/inv1/INV: \[
\begin{align*}
\text{piggybank} & \in \mathbb{N} \\
\text{amount} & \in \mathbb{N}_1 \\
\text{member} & \in \text{members}
\end{align*} \vdash \text{piggybank} + \text{amount} \in \mathbb{N}
\]

Contribute/inv3/INV: \[
\begin{align*}
\text{account} & \in \text{members} \rightarrow \mathbb{N} \\
\text{amount} & \in \mathbb{N}_1 \\
\text{member} & \in \text{members}
\end{align*} \vdash \text{accounts} \left( \text{member} \mapsto \text{accounts}(\text{member}) + \text{amount} \right) \in \text{members} \rightarrow \mathbb{N}
\]

Contribute/piggybank/EQL: \[
\begin{align*}
\text{amount} & \in \mathbb{N}_1 \\
\text{member} & \in \text{members}
\end{align*} \vdash \text{piggybank} = \text{piggybank} + \text{amount}
\]
Contribute/act2/WD:

\[ \text{amount} \in \mathbb{N} \quad \text{member} \in \text{members} \quad \land \quad \text{accounts} \in \text{MEMBER} \rightarrow \mathbb{Z} \]

BuyCoffee/grd2/WD:

\[ \text{member} \in \text{members} \quad \land \quad \text{accounts} \in \text{MEMBER} \rightarrow \mathbb{Z} \]

BuyCoffee/inv3/INV:

\[
\begin{align*}
\text{accounts} \in \text{members} \rightarrow \mathbb{N} \\
\text{member} \in \text{members} \\
\text{accounts}(\text{member}) \geq \text{coffeeprice} \\
\{ \text{member} \mapsto \text{accounts}(\text{member}) - \text{coffeeprice} \} \in \\
\text{members} \rightarrow \mathbb{N}
\end{align*}
\]

BuyCoffee/act1/WD:

\[ \text{member} \in \text{members} \quad \land \quad \text{accounts} \in \text{MEMBER} \rightarrow \mathbb{Z} \]

**Notation**

\[
\begin{align*}
\neq & \quad a \neq b = a \text{ is not equal to } b \\
\mapsto & \quad a \mapsto b \text{ (a maps to b)}
\end{align*}
\]

The proof obligations contain a surprise:
Contribute/piggybank/EQL on action piggybank := piggybank + amount cannot be discharged by the auto-prover.

This EQL PO requires a proof that piggybank is not changed, but of course, piggybank := piggybank + amount must change the value of the variable piggybank, unless amount is 0.

What is that all about?

Contribute appears in the refinement as a new event, but here it is changing the value of the variable piggybank, which is part of the state of CoffeeClub, the machine being refined.

In order to preserve consistency, any event of a refinement that modifies the state of the machine being refined must itself be a refinement of one or more events of the machine being refined.

**Solution**

The event FeedBank of CoffeeClub changes the value of the variable piggybank in a similar way to contribute, thus Contribute must be seen as a refinement of FeedBank and Contribute should be defined as follows.

Contribute \(\cong\)

\[
\begin{align*}
\text{REFINES} \\
\text{FeedBank} \\
\text{ANY} \\
\text{amount} \\
\text{member} \\
\text{WHERE} \\
\text{grd1:} & \quad \text{amount} \in \mathbb{N} \\
\text{grd2:} & \quad \text{member} \in \text{members}
\end{align*}
\]
3.2. REFINEMENT

THEN
act1:  piggybank := piggybank + amount
act2:  accounts(member) := accounts(member) + amount
END

or more concisely, using extension:

Contribute : extended \cong
REFINES
FeedBank
ANY
member
WHERE
grd3:  member \in members
THEN
act2:  accounts(member) := accounts(member) + amount
END

This removes the EQL PO, and there is an important lesson in this example. In most —if not all— cases the presence of EQL POs will probably indicate a bad refinement.

3.2.7 What are the new POs?

There are a number of INV POs, but the following are new:

INITIALISATION: act4/FIS: N1 \neq \emptyset

Contribute/act2/WD:
member \in \text{dom}(accounts) \land accounts \in MEMBER \rightarrow N1

BuyCoffee/grd2/WD:
member \in \text{dom}(accounts) \land accounts \in MEMBER \rightarrow N1

BuyCoffee/act1/WD:
member \in \text{dom}(accounts) \land accounts \in MEMBER \rightarrow N1

FIS is concerned with feasibility, deriving in this case from the initialisation

\text{coffeprice}' \in N1

This will be only feasible if N1 \neq \emptyset, which of course is trivially true.

WD is concerned with well-definedness. Such POs are concerned with showing that an expression is well defined. In this case they all derive from expression containing f(x), which will only be well-defined if x \in \text{dom}(f), in this case member \in \text{dom}(accounts). This is guaranteed by the guard member \in members and the invariant accounts \in member \rightarrow N.
3.3 Animation

The process of modelling that is being described here is concerned with ensuring that the model that is being developed is consistent across the development. There is one flaw:

the analysis of the informally presented requirements cannot be formalised.

Animation is a useful technique that provides for informal correlation of behaviour with the requirements. It should be appreciated that animation is not a substitute for rigorous verification using proof. Animation and rigorous modelling are complementary. In particular, animation provides a strategy for explaining your model to someone who does not understand Event-B. Animation is also useful to the modeller for obtaining a check of the behaviour of the events in the model against the informal statement of the requirements.

AnimB is a very good animation plugin for the Rodin platform. AnimB is interesting because it provides a number of different ways of animating, all of which can be mixed.

At each step in animation AnimB shows which events are enabled. Then the person running the animation has the following choices:

1. choose the event and the values of any parameters;
2. choose the event and let the animator choose the values of the parameters nondeterministically;
3. let the animator choose the event and the parameters nondeterministically.

3.3.1 Animation Constraints

Animators generally, and AnimB in particular impose a stronger finiteness constraint than the finiteness constraints imposed by Event-B. Thus AnimB minimum and maximum limits on the set \( \mathbb{N} \), and these can be set in the animator.

The context Members, as declares a finite set MEMBER, but AnimB requires an explicit limit on the set. That can be done by defining a finite set of elements of MEMBER. For this purpose the context has a section where AnimB values can be defined. In this case MEMBER is declared to be a set containing 3 members, \( \{m1, m2, m3\} \), as shown below.

```plaintext
context Members
sets
    MEMBER
axioms
```
3.3. ANIMATION

\[ \text{axm1: } \text{finite(MEMBER)} \]

\begin{verbatim}
\textbf{AnimB VALUES}
\textbf{MEMBER} \{m1, m2, m3\}  Define a set of 3 members
\textbf{END}
\end{verbatim}

The machine \textit{MemberShip} is as before and is able to be animated by AnimB.
Chapter 4

Refinement

The previous chapter explored a simple development that pursued refinement mainly as extension. In this chapter we will pursue refinement as a development path from an abstract specification through to a concrete model that is very close to implementation.

The development will also illustrate the strategy of commencing with a very concise, precise specification of the model in which we are interested.

4.1 What is Refinement?

Refinement is the process by which more detail, or even new layers, are added to a model. This must be done in such a way that the refined model is consistent with the model being refined. The word refinement has many shades of meaning some of which may be considered to be confusing for the understanding the intended purpose of refinement. The most appropriate dictionary meaning of refinement is

*improve (something) by making small changes, in particular make (an idea, theory or method) more subtle or accurate* (Oxford English Dictionary)

although even that has its problems as a model should be perfectly clear without refinement.

Cliff Jones (in [14] proposed the word *reification* 

*make (something abstract) more concrete or real* (Oxford English Dictionary)

as an alternative to refinement.

In any case the refinement of a model must be consistent with the model. This is enforced by the following constraints on a refinement of an event:

**guards must not be weaker:** may be stronger

**actions may not be more nondeterministic** may be more deterministic.

4.2 An example: Square root

Generally, we will not be using examples that are principally numeric computation, but for the current purpose the example of computing the “integer square root” of a natural number will provide a simple example that illustrates refinement quite effectively. It will also be used to show how parameters can be used to *steer* refinement, especially towards a concrete refinement.
4.2.1 Square Root Definition

We start with a definition of the square root function $SQRT$ that we define and eventually refine to a concrete model.

Let $num$ be the number whose integer square root we want to compute and $SQRT$ be the integer square root function. The integer square root of a natural number is the largest integer that is not greater than the real square root. We define $SQRT$ as follows:

\begin{align*}
num & \in \mathbb{N} \\
SQRT & \in \mathbb{N} \rightarrow \mathbb{N} \\
SQRT(num) \times SQRT(num) & \leq num \\
um & < (SQRT(num) + 1) \times (SQRT(num) + 1)
\end{align*}

A context will be used to define an arbitrary constant $num$, whose square root we wish to compute. The value of $num$ is any natural number.

**CONTEXT** SquareRootArg

**CONSTANTS**

$num$

**AXIOMS**

\text{axm1: } num \in \mathbb{N}

**END**

Next we will define a context that contains some number theoretic theorems that will be useful in the discharge of proof obligations:

**CONTEXT** Theories

**AXIOMS**

\text{thm1: } \forall n \cdot n \in \mathbb{N} \Rightarrow (\exists m \cdot m \in \mathbb{N} \land (n = 2 \cdot m \lor n = 2 \cdot m + 1))

\text{Every natural number is either even or odd}

\text{thm2: } \forall n \cdot n \in \mathbb{N} \Rightarrow n < (n + 1) \cdot (n + 1)

\text{Every natural number is less than the square of its successor}

\text{thm3: } \forall m, n \cdot m, n \in \mathbb{N} \land n \in \mathbb{N} \Rightarrow (m + n)/2 < n

\text{The mean of any pair of unequal natural numbers is less than the larger of the pair}

\text{thm4: } \forall m, n \cdot m, n \in \mathbb{N} \land n \in \mathbb{N} \Rightarrow (m + n)/2 \geq m

\text{The mean of any pair of natural numbers is greater than or equal to the smaller of the pair}

**END**

**Notation**

<table>
<thead>
<tr>
<th>math</th>
<th>ascii</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Rightarrow$</td>
<td>$\implies$</td>
<td>logical implication: $P \Rightarrow Q$, if $P$ is true then $Q$ is true, but $Q \Rightarrow P$ is not necessarily true</td>
</tr>
<tr>
<td>$\forall x \cdot P$</td>
<td>$\forall x . P$</td>
<td>Universal quantification</td>
</tr>
<tr>
<td>$\exists x \cdot P$</td>
<td>$\exists x . P$</td>
<td>Existential quantification</td>
</tr>
</tbody>
</table>
4.2. AN EXAMPLE: SQUARE ROOT

Concepts

∀ x ⋅ P  
for all x, P is true (⊤), where x is a list of variables, and P is a predicate. In many cases P will be an implication: ∀ x ⋅ Q ⇒ R

∃ x ⋅ P  
there exists x such that P is true (⊤), where x is a list of variables, and P is a predicate. In many cases P will be a conjunction: ∃ x ⋅ Q ∧ R

and now a context that captures the definition of the SQRT function given in 4.2.1

CONTEXT SquareRoot_ctx
EXTENDS Theories
CONSTANTS SQRT
AXIOMS
axml: SQRT ∈ N → N

∀ m, n ⋅ m ∈ N ∧ n ∈ N

axm2: ⇒

(m = SQRT(n) ⇔ m ⋅ m ≤ n ∧ (m + 1) ⋅ (m + 1) > n)

∀ m, n ∈ N

thm1: ⇒

SQRT(n) ⋅ SQRT(n) ⇔ n ∧ (SQRT(n) + 1) ⋅ (SQRT(n) + 1) > n

END

Concepts

extend extend a context by adding new Sets, Constants, Axioms and Theorems

Notation

math ascii
⇔ <=>

logical equivalence: P ⇔ Q = P ⇒ Q and Q ⇒ P
We model \( \sqrt{\text{id}} \) as follows:

**MACHINE** SquareRoot

**REQ:** \( \text{sqrt} = \text{SQRT}(\text{num}) \)

**SEES**
- SquareRoot_ctx
- SquareRootArg

**VARIABLES**
- \( \text{sqrt} \)
- \( \text{final} \)

**INVARIANTS**
- inv1: \( \text{sqrt} \in \mathbb{N} \)
- inv2: \( \text{final} \in \text{BOOL} \)
- inv3: \( \text{final} = \text{TRUE} \Rightarrow \text{sqrt} = \text{SQRT}(\text{num}) \)

**EVENTS**

**Initialisation** \( \cong \)

**THEN**
- act1: \( \text{sqrt} := \mathbb{N} \)
- act2: \( \text{final} := \text{FALSE} \)

**END**

**SquareRoot** \( \cong \)

**WHERE**
- grd1: \( \text{final} = \text{FALSE} \)

**THEN**
- act1: \( \text{sqrt} := \text{SQRT}(\text{num}) \)
- act2: \( \text{final} := \text{TRUE} \)

**END**

**END** SquareRoot

**Note:**

An alternative to the action \( \text{sqrt} := \text{SQRT}(\text{num}) \) is:

\[
\text{sqrt} \mid (\text{sqrt}' \in \mathbb{N}) \\
\land (\text{sqrt}' \ast \text{sqrt}' \leq \text{num}) \\
\land (\text{num} < (\text{sqrt}' + 1) \ast (\text{sqrt}' + 1))
\]

**Notation**

<table>
<thead>
<tr>
<th>math</th>
<th>ascii</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \mid )</td>
<td>( :</td>
</tr>
</tbody>
</table>

“becomes such that”: \( x \mid P \), where \( x \) is a variable and \( P \) is a predicate, means assign to \( x \) a value such that \( P(x) \) is \( \top \), where \( \top \) is Boolean true. Within \( P \), \( x \) represents the value of the variable \( x \) before the assignment, and \( x' \) represents the value of \( x \) after the assignment. Thus, \( x \mid x' = x + 1 \) assigns the value of \( x + 1 \) to the variable \( x \). Equivalent to \( x := x + 1 \).

The BOOL variable \( \text{final} \) is used to express the notion that the value of \( \text{sqrt} \) has no particular meaning unless \( \text{final} = \text{TRUE} \), and the event SquareRoot will only fire when \( \text{final} = \text{FALSE} \).
4.2. AN EXAMPLE: SQUARE ROOT

<table>
<thead>
<tr>
<th>Notation</th>
<th>math</th>
<th>ascii</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOOL</td>
<td>BOOL</td>
<td>BOOL  = {TRUE, FALSE}</td>
</tr>
<tr>
<td>bool</td>
<td>bool</td>
<td>bool = {\top, \bot}</td>
</tr>
</tbody>
</table>

Note: the type bool is not denotable in an Event-B model.

When we look at the Proof obligations for SquareRoot we see:

\[
\text{inv2/WD: } \sqrt{e} \in \mathbb{N} \mid final = \text{TRUE} \Rightarrow num \in \text{dom}(\text{SQRT}) \land \text{SQRT} \in \mathbb{Z} \to \mathbb{Z}
\]

\text{INITIALISATION/inv1/INV: } sqrt' \in \mathbb{N} \mid sqrt' \in \mathbb{N}

\text{INITIALISATION/inv2/INV: } sqrt' \in \mathbb{N} \mid \text{FALSE} = \text{TRUE} \Rightarrow sqrt' = \text{SQRT}(num)

\text{INITIALISATION/act1/FIS: } \neg \mathbb{N} \neq \emptyset

\text{SquareRoot/inv1/INV: } \sqrt{e} \in \mathbb{N} \mid final = \text{FALSE} \Rightarrow \text{SQRT}(num) \in \mathbb{N}

\text{SquareRoot/inv2/INV: } final = \text{TRUE} \Rightarrow sqrt = \text{SQRT}(num) \mid TRUE = \text{TRUE}

\text{SquareRoot/act1/WD: } final = \text{FALSE} \Rightarrow num \in \text{dom}(\text{SQRT}) \land \text{SQRT} \in \mathbb{Z} \to \mathbb{Z}

These are all easy —even trivial— to discharge.

This model trivially models the requirement as the action is literally the requirement. Of course we still have no idea how to compute the required value of SQRT.

4.2.2 Refinement: How we make progress

A simple way of motivating what to do next was presented by David Gries in Science of Programming\[10\]. In the current specification we have two predicates:

\[
\sqrt{e} \times \sqrt{e} \leq num
\]

\[
um < (\sqrt{e} + 1) \times (\sqrt{e} + 1)
\]

Each of them is easy to satisfy on their own, but the difficulty is satisfying them both at the same time. This suggests that we should use two variables and then try to bring them together. Thus we will use

\[
low \times low \leq num \quad \text{(4.5)}
\]

\[
um < high \times high \quad \text{(4.6)}
\]

\[
low < high \quad \text{(4.7)}
\]

and move low and high closer together until

\[
low + 1 = high \quad \text{(4.8)}
\]

at which point low will be the desired value of sqrt.
4.2.3 “Opening Up” an event

We need to “open up” the simple, but correct action \( \text{sqrt} := \text{SQRT}(\text{num}) \) and demonstrate one way of modelling the computation of \( \text{SQRT}(\text{num}) \). The type of refinement steps we show next will be repeated in many other examples, differing only in the detail of how we are modelling our requirements.

We introduce a new slave event \textit{Improve} that uses parameters constrained by guards to produce “better” values of \textit{low} and \textit{high}. This

\begin{verbatim}
MACHINE SquareRootR1
REFINES SquareRoot
SEES SquareRoot_ctx

VARIABLES
  sqrt
  final
  low
  high

INVARIANTS
  inv1: \( \text{low} \in \mathbb{N} \)
  inv2: \( \text{high} \in \mathbb{N} \)
  inv3: \( \text{low} + 1 \leq \text{high} \)
  inv4: \( \text{low} \ast \text{low} \leq \text{num} \)
  inv5: \( \text{num} < \text{high} \ast \text{high} \)
  thm1: \( \text{low} + 1 \neq \text{high} \Rightarrow \text{low} < (\text{low} + \text{high})/2 \)
  thm2: \( (\text{low} + \text{high})/2 < \text{high} \)

VARIANT high – low

EVENTS
  Initialisation : extended ≡
  THEN
    act3: \( \text{low} :| \text{low}' \in \mathbb{N} \wedge \text{low}' \ast \text{low}' \leq \text{num} \)
    act4: \( \text{high} :| \text{high}' \in \mathbb{N} \wedge \text{num} < \text{high}' \ast \text{high}' \)
  END

SquareRoot ≡
REFINES SquareRoot
WHERE
  grd1: \( \text{final} = \text{FALSE} \)
  grd2: \( \text{low} + 1 = \text{high} \)
  thm1: \( \text{low} \ast \text{low} \leq \text{num} \)
  thm2: \( \text{num} < \text{high} \ast \text{high} \)

THEN
  act1: \( \text{sqrt} := \text{low} \)
  act2: \( \text{final} := \text{TRUE} \)
END
\end{verbatim}
4.2. AN EXAMPLE: SQUARE ROOT

4.2.3.1 Parameters

Parameters represent arbitrary, nondeterministic values that control the “firing” of an event through the guards. In the above we have introduced parameters \( l \) and \( h \) to represent improvements in \( \text{low} \) and \( \text{high} \), respectively. The constraints on \( l \) and \( h \) are specified, but there is no guidance given as to how to choose such values. Notice that the specification allows for only one of \( l \) and \( h \) to be an improvement, but one must be an improvement, or the machine will deadlock. It should also be noticed that the events \( \text{SquareRoot} \) and \( \text{Improve} \) have complementary guards. This ensures that \( \text{Improve} \) will keep firing until \( \text{low} + 1 = \text{high} \), at which point \( \text{SquareRoot} \) will fire once. After that the machine is deadlocked.

4.2.3.2 Convergence and Variants

The event \( \text{Improve} \) has been given a status of convergent. The reason is that the original single event \( \text{SquareRoot} \) has been refined into an event that will fire only once, when its guard is satisfied, but we have introduced a new slave event \( \text{Improve} \) that could, in principle, fire forever. By giving it the status convergent we are signalling that the event converges, i.e. it will fire only some finite number of times. To prove convergence we are required to give a variant. A variant is an expression that may be of two possible forms:

1. an expression that yields an integer value, or
2. an expression that yields a finite set

The variant is subject to the following constraints:

**case 1:** whenever any convergent event occurs the value of the variant must be a natural number before the event and must strictly decrease across the event.

**case 2:** whenever any convergent event occurs the cardinality of the variant must strictly decrease across the event.

By decrease across the event we mean that the value on exit from the event is less than the value on entry to the event.
It is clear that if these constraints on the variant are satisfied then all convergent events must eventually terminate.

**Note:** There is only one variant for a machine and the above conditions apply to all convergent events. The variant produces the following POs for each convergent event:

- **VWD:** possible well-definedness PO;
- **NAT:** proof that a numeric variant always yields a natural number after the event;
- **VAR:** proof that the event reduces the value of a numeric-valued variant expression, or the cardinality of a set-valued variant.

Notice that the variant in the above machine could also have been a set-valued variant: \( \text{low} \ldots \text{high} \).

Failure of the PO generated for the variant may show that the machine is subject to **livelock**. Livelock occurs when ‘main events’, in this case the \( \text{SquareRoot} \) cannot fire because the slave events can fire indefinitely.

### 4.2.4 Improving Improve

We will now refine \( \text{Improve} \). The central idea is to refine either \( \text{low} \) or \( \text{high} \) by choosing a value that is strictly between \( \text{low} \) and \( \text{high} \); we know we can do that because \( \text{low} \) and \( \text{high} \) are separated by more than 1: \( \text{low} < \text{high} \) and \( \text{low} + 1 \neq \text{high} \). In the first refinement we will propose a new parameter \( m \) that replaces either \( l \) or \( h \). We will refine \( \text{Improve} \) in two ways: improving either \( \text{low} \) or \( \text{high} \).

**MACHINE** \( \text{SquareRootR2} \)

**REFINES**

\( \text{SquareRootR1} \)

**SEES**

\( \text{SquareRoot\_ctx} \)

**VARIABLES**

\( \text{sqrt} \)

\( \text{final} \)

\( \text{low} \)

\( \text{high} \)

**INVARIANTS**

\( \text{Improve1} \equiv \)

**REFINES**

\( \text{Improve} \)

**ANY**

\( m \)

**WHERE**

\( \text{grd1: low + 1 \neq high} \)

\( \text{grd2: m \in \mathbb{N}} \)

\( \text{grd3: low < m \land m < high} \)

\( \text{grd4: m \ast m \leq \text{num}} \quad m \text{ is a better value for low} \)

**WITH**

\( l: l = m \)

\( h: h = \text{high} \)

**THEN**
4.2. An Example: Square Root

act1: \( low := m \)

END

\textbf{Improve2} \( \cong \)

\textbf{REFINES}

\textbf{Improve}

\textbf{ANY}

\( m \)

\textbf{WHERE}

\( \text{grd1: } \) \( low + 1 \neq high \)

\( \text{grd2: } \) \( m \in \mathbb{N} \)

\( \text{grd3: } \) \( low < m \land m < high \)

\( \text{grd4: } \) \( m \times m > num \hspace{1cm} m \text{ is a better value for} \)

\( high \)

\textbf{WITH}

\( l: \) \( l = low \)

\( h: \) \( h = m \)

\textbf{THEN}

\( \text{act1: } \) \( high := m \)

END

\textbf{EVENTS}

\textbf{Initialisation: } \textit{extended} \( \cong \)

\textbf{THEN}

\textbf{END}

\textbf{SquareRoot: } \textit{extended} \( \cong \)

\textbf{REFINES}

\textbf{SquareRoot}

\textbf{WHERE}

\textbf{THEN}

\textbf{END}

\textbf{END SquareRootR2}

4.2.4.1 Witness and the With clause

The issue here is that we have replaced two parameters, \( l \) and \( h \), by a single parameter, \( m \), in each of the two refinements of \textbf{Improve}. Parameters \( l \) and \( h \) have disappeared. To enable the verification that \textbf{Improve1} and \textbf{Improve2} do refine \textbf{Improve} we have to give what is known as a \textit{witness} for \( l \) and \( h \). This will show how the new parameters \textit{simulate} the old.

4.2.5 Refining SquareRootR2

The previous refinement introduced the value \( m \) and defined it declaratively as simply a value —any value— strictly between \( low \) and \( high \). We can proceed with many different strategies for \( m \)
1. low + 1 and high − 1

2. a value midway between low and high

We will adopt for this refinement.

```
MACHINE SquareRootR3
REFINES
    SquareRootR2
SEES
    SquareRoot_ctx
VARIABLES
    sqrt
    final
    low
    high
INVARIANTS

EVENTS
Initialisation : extended ≜
THEN
END

SquareRoot : extended ≜
REFINES
    SquareRoot
WHERE
THEN
END

Improve1 ≜
REFINES
    Improve
ANY
    m
WHERE
    grd1: low + 1 ≠ high
    grd2: m = (low + high)/2
    grd3: m * m ≤ num  m is a better value for low
THEN
    act1: low := m
END

Improve2 ≜
REFINES
    Improve
ANY
    m
WHERE
4.2. AN EXAMPLE: SQUARE ROOT

\textbf{4.2.6 Refining SquareRootR3}

SquareRootR3 is still not completely \textit{concrete} as it depends on the \textit{abstract} parameter \(m\). But the value of \(m\) is clearly able to be computed from the values of the variable \(low\) and \(high\) and hence can be replaced by a variable, which we will name \(mid\). Thus, we will introduce a variable \(mid\) with the invariant

\[ mid \times mid > num \]

At the same time we will remove the nondeterministic initialisation of \(low\) and \(high\), making it easier to initialise \(mid\), and also producing a concrete machine, or algorithm. It is clear that initialisation of \(low\) to 0 and \(high\) to \(num + 1\) will satisfy the invariant, but it is also clear that neither will be very good approximations to the square root for very large values of \(num\). However, finding a better approximation will require computation and as the final algorithm is logarithmic it can be argued that 0 and \(num + 1\) are good enough.

\textbf{MACHINE SquareRootR4}
\textbf{REFINES}
SquareRootR3
\textbf{SEES}
SquareRoot_ctx
\textbf{VARIABLES}
sqrt
low
high
mid
\textbf{INvariants}
inv1: \( mid = (low + high)/2 \)
\textbf{EVENTS}
Initialisation \(\triangleq\)
THEN
act1: \( sqrt := 0 \)
act2: \( low := 0 \)
act3: \( high := num + 1 \)
act4: \( mid := (num + 1)/2 \)
END

\textbf{SquareRoot : extended} \(\triangleq\)
\textbf{REFINES}
SquareRoot
\textbf{WHERE}

\begin{align*}
\text{grd1:} \quad & low + 1 \neq high \\
\text{grd2:} \quad & m = (low + high)/2 \\
\text{grd3:} \quad & m \times m > num \quad m \text{ is a better value for} \\
\text{THEN} \quad & \text{act1:} \quad high := m \\
\text{END}
\end{align*}
THEN
END

Improve1 ⊆
REFINES Improve
ANY

WHERE
  grd1: \(low + 1 \neq high\)
  grd2: \(mid * mid \leq num\) \(mid\) is a better value for \(low\)

WITH
  m: \(m = mid\)
THEN
  act1: \(low := mid\)
  act2: \(mid := (mid + high) / 2\)
END

Improve2 ⊆
REFINES Improve
ANY

WHERE
  grd1: \(low + 1 \neq high\)
  grd2: \(mid * mid > num\) \(mid\) is a better value for \(high\)

WITH
  m: \(m = mid\)
THEN
  act1: \(high := mid\)
  act2: \(mid := (low + mid) / 2\)
END

END SquareRootR4

4.2.7 An alternative refinement to SquareRootR4

It is possible to refine directly from SquareRootR2 to SquareRootR4 bypassing the step in which the parameter \(m\) is equated to \((low + high)/2\) and going straight to the introduction of the variable \(mid\). That is not to say that either approach is better. In general there will be many different refinement sequences, each expressing different insights into how the refinements can proceed. In this case, we are pursuing a different path that avoids the introduction of the parameter \(m\). On the other hand the above refinement demonstrates how useful concepts can be introduced via parameters.
4.2.7.1 Exercise

Produce the alternative refinement step from SquareRootR3 to SquareRootR4. Name it SquareRootR4B.

4.3 Modelling a parametric argument

In the model above we modeled an arbitrary argument to SquareRoot as a constant \( num \) in the context to the SquareRoot machine. That is perfectly satisfactory as far as verifying the square root process, however it is not parametric.

The following is a repeat of the modelling of SquareRoot using a parameter.

**MACHINE SquareRoot**

**SEES**
- Theories

**VARIABLES**
- sqrt

**INVARIANTS**
- inv1: sqrt \( \in \mathbb{N} \)

**EVENTS**

**Initialisation**  

**THEN**

act1: sqrt := 0

**END**

**SquareRoot**  

**ANY**

**WHERE**

- grd1: num \( \in \mathbb{N} \)

**THEN**

act1: sqrt := \( SQRT(num) \)

**END**

**END SquareRoot**

The above is very similar to the earlier starting point for SquareRoot except that \( num \) is now a parameter.

When SquareRoot is refined it is clear that in order to be able to reference the value of the \( num \) parameter from different events the value \( num \) will have to be stored in a variable.

**MACHINE SquareRootR1**

**REFINES**

SquareRoot

**SEES**

SquareRoot_ctx

**VARIABLES**
sqrt
low
high
numv
active

**INVARIANTS**

inv1: \( numv \in \mathbb{N} \)
inv2: \( low \in \mathbb{N} \)
inv3: \( high \in \mathbb{N} \)
inv4: \( low + 1 \leq high \)
inv5: \( low \times low \leq numv \)
inv6: \( numv < high \times high \)
inv7: \( active \in \text{BOOL} \)
inv8: \( active = \text{FALSE} \Rightarrow sqrt = \text{SQRT}(numv) \)

**VARIANT**

\( high - low \)

**EVENTS**

**Initialisation :** *extended* \( \triangleq \)

**THEN**

act2: \( numv := 0 \)
act3: \( low := 0 \)
act4: \( high := 1 \)
act5: \( active := \text{FALSE} \)

**END**

**SquareRoot** \( \triangleq \)

**REFINES**

SquareRoot

**WHERE**

grd1: \( low + 1 = high \)
grd2: \( active = \text{TRUE} \)

**WITH**

num: \( num = numv \)

**THEN**

act1: \( sqrt := low \)
act2: \( active := \text{FALSE} \)

**END**

**Improve** \( \triangleq \)

**STATUS** convergent

**ANY**

\( l \)
\( h \)

**WHERE**
4.3. MODELLING A PARAMETRIC ARGUMENT

\begin{align*}
\text{grd1:} & \quad low + 1 \neq high \\
\text{grd2:} & \quad l \in \mathbb{N} \land low \leq l \land l \ast l \leq num \\
\text{grd3:} & \quad h \in \mathbb{N} \land h \leq high \land num < h \ast h \\
\text{grd4:} & \quad l + 1 \leq h \\
\text{grd5:} & \quad h - 1 < high - low \\
\text{grd6:} & \quad active = \text{TRUE}
\end{align*}

THEN

\begin{align*}
\text{act1:} & \quad low, high := l, h
\end{align*}

END

activate \trianglerighteq

\begin{align*}
\text{ANY} \\
\text{num} \\
\text{WHERE} \\
\text{grd1:} & \quad num \in \mathbb{N} \\
\text{grd2:} & \quad active = \text{FALSE}
\end{align*}

THEN

\begin{align*}
\text{act1:} & \quad numv := num \\
\text{act2:} & \quad low \mid low' \in \mathbb{N} \land low' \ast low' \leq num \\
\text{act3:} & \quad high \mid high' \in \mathbb{N} \land num < high' \ast high' \\
\text{act4:} & \quad active := \text{TRUE}
\end{align*}

END

END SquareRootR1

The BOOL variable, active is used in the above refinement to distinguish between the states when the events are actively searching for a square root (active = \text{TRUE}) and the quiescent state (active = \text{FALSE}) when a square root has been found.

4.3.1 Remainder of development

The remainder of the development follows the previous for the parameterless SquareRoot event, leading to the final refinement.

MACHINE SquareRootR4
REFINES SquareRootR3
SEES SquareRoot_ctx

VARIABLES
sqrt
numv
low
high
active
mid

INVariANTS
inv1: \quad mid = (low + high)/2
EVENTS
Initialisation $\triangleq$
THEN
act1: $numv := 0$
act2: $low := 0$
act3: $high := num + 1$
act4: $sqrt := 0$
act5: $active := FALSE$
act6: $mid := 0$
END

SquareRoot $\triangleq$
REFINES SquareRoot
WHERE
grd1: $low + 1 = high$
grd2: $active = TRUE$
THEN
act1: $sqrt := low$
act2: $active := FALSE$
END

Activate $\triangleq$
REFINES activate
ANY
num
WHERE
grd1: $num \in \mathbb{N}$
grd2: $active = FALSE$
THEN
act1: $numv := num$
act2: $low := 0$
act3: $high := num + 1$
act4: $mid := (num + 1)/2$
act5: $active := TRUE$
END

Improve1 $\triangleq$
REFINES Improve1
WHERE
grd1: $low + 1 \neq high$
grd2: $mid \times mid \leq numv$
WITH
m: $m = mid$
4.3. MODELLING A PARAMETRIC ARGUMENT

\[
\begin{align*}
\text{THEN} & \\
\text{act1:} & \quad low := mid \\
\text{act2:} & \quad mid := (mid + high)/2 \\
\text{END}
\end{align*}
\]

\textbf{Improve2} $\cong$

\textbf{REFINES}

\textbf{Improve2}

\textbf{WHERE}

\begin{align*}
\text{grd1:} & \quad low + 1 \neq high \\
\text{grd2:} & \quad numv < mid \ast mid \\
\text{grd3:} & \quad active = TRUE
\end{align*}

\textbf{WITH}

\begin{align*}
\text{m:} & \quad m = mid
\end{align*}

\textbf{THEN}

\begin{align*}
\text{act1:} & \quad high := mid \\
\text{act2:} & \quad mid := (low + mid)/2
\end{align*}

\text{END}

\textbf{END} \textbf{SquareRootR4}

4.3.2 Converting to programming code

The final refinement is easily seen to be translated to the following code.

\[
\begin{align*}
\text{low := 0;} \\
\text{high := num + 1;} \\
\text{while low + 1 \neq high} \\
\text{\quad mid := (low + high)/2} \\
\text{\quad if mid \ast mid \leq num} \\
\text{\quad \quad low := mid} \\
\text{\quad else high := mid} \\
\text{\quad} \\
\text{sqrt := low}
\end{align*}
\]
Chapter 5

Invariants: Specifying Safety

Use of invariants to formulate “safety” and as a means of ensuring “safety”
Use of theorems to provide a check on properties that are expected to be satisfied
Increasing familiarity with the set theory used by Event-B
Data refinement: this chapter contains the first example of refinement that significantly refines the data (variables) of the model

There is a danger that the invariant is seen merely as a mechanism for typing variables, somewhat similar to the type specifications in typed programming languages. The square root example should have shown that the invariant is more than that. The invariant can be used to specify the semantic relationship between variables, and in the square root example that relationship was critical to being able to demonstrate that the value finally produced in the variable $\text{sqrt}$ —when all the events complete—did indeed produce the required value of the square root. If the invariants were reduced to recording only type information, the model would still behave the same as the preceding model, but the PO would not provide confirmation.

This should highlight the requirement that developers of Event-B models should make maximum use of invariance and not behave in the way they might if writing a program.

*Invariants*

Invariants should be as strong as necessary, *but no stronger.*

Invariants are often described metaphorically as *safety constraints* and in the next example the invariant is literally an expression of safety.

Also, theorems provide very useful sanity checks to confirm those properties that are “obviously” true.

5.1 Simple Traffic Lights

We wish to explore the use of invariants to ensure safety for a traffic light controlled intersection. The discussion will move from:

* a simple 2-way intersection* consisting of *NorthSouth* and *EastWest* directions, and then moving to
* a generalised multiway intersection*
The simple two-way intersection consists of two directions: NorthSouth and EastWest. Each direction has two sets of identical traffic lights each displaying Red, Green and Amber lights. There are only two directions: for example there is no turn-right or turn-left direction.

Traffic light sequence
Throughout this chapter traffic lights change through the sequence: Red, Green, Amber, Red, .... Only one colour is shown at any time.

CONTEXT SimpleTwoWay0
SETS LIGHTS DIRECTION
CONSTANTS Red Green Amber NorthSouth EastWest
AXIOMS
  axm1: partition(LIGHTS, {Red}, {Green}, {Amber})
  axm2: partition(DIRECTION, {NorthSouth}, {EastWest})
END

<table>
<thead>
<tr>
<th>Notation</th>
<th>math</th>
<th>ascii</th>
</tr>
</thead>
<tbody>
<tr>
<td>partition</td>
<td>partition</td>
<td>( \cap )</td>
</tr>
<tr>
<td>( \cap )</td>
<td>/\</td>
<td>Set intersection: ( S \cap T ) is the elements of elements that are in both ( S ) and ( T )</td>
</tr>
</tbody>
</table>

\( \text{partition}(\text{LIGHTS}, \{\text{Red}\}, \{\text{Green}\}, \{\text{Amber}\}) \) is equivalent to

\[
\text{LIGHTS} = \{\text{Red}, \text{Green}, \text{Amber}\}
\]

\[
\text{Red} \neq \text{Green}
\]

\[
\text{Green} \neq \text{Amber}
\]

\[
\text{Amber} \neq \text{Red}
\]

Sets LIGHTS and DIRECTION are finite enumerated sets.

- \( \text{LIGHTS} \) has 3 distinct colours, and
- \( \text{DIRECTION} \) has 2 distinct directions.

Now consider a machine SimpleChangeLights of which only a skeleton will be shown.

MACHINE SimpleChangeLights
SEES SimpleTwoWay0
VARIABLES lights
INVARIAENTS
5.1. SIMPLE TRAFFIC LIGHTS

\[\text{inv1: } \text{lights} \in \text{DIRECTION} \rightarrow \text{LIGHTS}\]

\textit{END SimpleChangeLights}

At the moment \textit{lights} is simply declared as a total function from \textit{DIRECTION} to \textit{LIGHTS}, and we want to explore what else is necessary to ensure a safe set of traffic lights.

We want the following to be true:

- whenever the intersection is unsafe the invariant must be \textit{false};
- whenever the invariant is \textit{true} the intersection must be safe.

That is:

\[\neg(\text{safe}) \Rightarrow \neg(\text{invariant}) \quad (5.1)\]

and

\[\text{invariant} \Rightarrow \text{safe} \quad (5.2)\]

Note that instead of \( x = T \), we will simply write \( x \), and wherever we might write \( x = F \), we will simply write \( \neg x \), for example \textit{safe} and \textit{invariant} in the above.

\textbf{Notation}

\begin{tabular}{|c|c|}
\hline
\text{math} & \text{ascii} \\
\hline
\neg & \textit{not} \\
\hline
\end{tabular}

\(\neg P\) negates the predicate \(P\)

\textbf{5.2} will be recognised as the contrapositive of \textbf{5.1} that is

\[P \Rightarrow Q \equiv \neg Q \Rightarrow \neg P\]

so we have only one requirement for safety, not two.

The second invariant —the safety condition— could be

\[\text{lights}(\text{NorthSouth}) \in \{\text{Green, Amber}\} \Rightarrow \text{lights}(\text{EastWest}) = \text{Red}\]

or

\[\text{lights}(\text{EastWest}) \in \{\text{Green, Amber}\} \Rightarrow \text{lights}(\text{NorthSouth}) = \text{Red}\]

\begin{array}{|c|c|c|}
\hline
\text{Red} & \text{Green} & \text{Amber} \\
\hline
\text{Red} & \text{safe} & \text{safe} \\
\text{Green} & \text{safe} & \text{unsafe} & \text{unsafe} \\
\text{Amber} & \text{safe} & \text{unsafe} & \text{unsafe} \\
\hline
\end{array}

There are other invariants that adequately express safety for this two-way intersection:

\[\text{lights}(\text{NorthSouth}) = \text{Red} \lor \text{lights}(\text{EastWest}) = \text{Red}\]

\(\text{Red} \in \text{ran(lights)}\)

But these conditions do not generalise to intersections with more than two ways. Indeed the expression of the invariant that best generalises is the formulation given in the next section.
CHAPTER 5. INVARIANTS: SPECIFYING SAFETY

5.1.1 Simplifying and Generalising

Instead of referencing the directions and their conflicting directions by name we will define a constant function OTHERDIR that maps NorthSouth to EastWest and vice-versa. This prepares the context to deal with multiple directions, something we will do in the next section. The definition of OTHERDIR is given in a new context, SimpleTwoWay1, that is an extension of SimpleTwoWay0

CONTEXT SimpleTwoWay1
EXTENDS SimpleTwoWay0
CONSTANTS OTHERDIR
AXIOMS
axm3: \text{OTHERDIR} \in \text{DIRECTION} \rightarrow \text{DIRECTION}
axm4: \text{OTHERDIR}(\text{NorthSouth}) = \text{EastWest}
axm5: \text{OTHERDIR}(\text{EastWest}) = \text{NorthSouth}
\forall \text{dir} \in \text{DIRECTION} \Rightarrow \text{OTHERDIR}(\text{OTHERDIR}(\text{dir})) = \text{dir}
\text{thm1: OTHERDIR ; OTHERDIR} \subseteq \text{id}

END

The context SimpleTwoWay1 extends SimpleTwoWay0 with a relation, OTHERDIR, (actually a total function) that maps each of the directions, respectively, to the “other” direction. The behaviour of OTHERDIR is defined by axioms axm3, axm4, axm5.

The same behaviour could be defined using universal quantification as shown in thm1, however given the axiomatic definition this behaviour should now be provable, hence the use of a theorem.

Similarly, it should be clear that if OTHERDIR is sequentially composed with itself the result should be the identity relation on the set DIRECTION. Again, this is tested by proposing a theorem.

5.1.2 The Simple TwoWay machine

The machine has a three events that, respectively, change the light in a particular direction to Red, Green or Amber. The machine must ensure:

• safety;
• correct sequencing: Red, Green, Amber, Red, ...
MACHINE SimpleChangeLights
SEES
SimpleTwoWay1
VARIABLES
lights
INvariants
inv1: \( \text{lights} \in \text{DIRECTION} \rightarrow \text{LIGHTS} \)
inv2: \( \text{lights}(\text{NorthSouth}) \in \{\text{Green}, \text{Amber}\} \Rightarrow \text{lights}(\text{EastWest}) = \text{Red} \)
thm1: \( \forall \text{dir} \cdot \text{dir} \in \text{lights} \land \text{lights}(\text{dir}) \in \{\text{Green}, \text{Amber}\} \Rightarrow \text{lights}((\text{OTHERDIR})(\text{dir})) = \text{Red} \)

EVENTS
Initialisation \( \triangleq \)
THEN
act1: \( \text{lights} = \{\text{NorthSouth} \mapsto \text{Red}, \text{EastWest} \mapsto \text{Red}\} \)
END

ToAmber \( \triangleq \)
ANY
\( \text{dir} \)
WHERE
\( \text{grd1: } \text{lights}(\text{dir}) = \text{Green} \)  
Sequencing
\( \text{thm1: } \text{lights}((\text{OTHERDIR})(\text{dir})) = \text{Red} \)  
Safety

Theorem in place of guard
The above set of guards contains a theorem. We would expect a guard expressing the constraint that the colour of the lights in direction \( \text{OTHERDIR}(\text{dir}) \) should be \text{Red}, but, in this case, the lights in direction \( \text{dir} \) are \text{Green}, so we expect the colour in the other direction will be \text{Red}. Therefore, we use a theorem, rather than a guard. Using a theorem is stronger than using a guard. We are proposing a stronger test on what we believe to be the current state.
Theorems can also appear in place of invariants, or axioms.

THEN
act1: \( \text{lights}(\text{dir}) := \text{Amber} \)
END

ToGreen \( \triangleq \)
ANY
\( \text{dir} \)
WHERE
\( \text{grd1: } \text{lights}(\text{dir}) = \text{Red} \)  
Sequencing
\( \text{grd2: } \text{lights}((\text{OTHERDIR})(\text{dir})) = \text{Red} \)  
Safety
THEN
act1: \( \text{lights}(\text{dir}) := \text{Green} \)
END

ToRed \( \triangleq \)
ANY
\( \text{dir} \)
WHERE
CHAPTER 5. INVARIANTS: SPECIFYING SAFETY

44

CHAPTER 5. INVARIANTS: SPECIFYING SAFETY

grd1: \( \text{lights}(\text{dir}) = \text{Amber} \)  
Sequencing: Safety is preserved

THEN

act1: \( \text{lights}(\text{dir}) := \text{Red} \)

END

END SimpleChangeLights

5.2 A Multiway Intersection

The multiway intersection consists of:

DIRECTION: a finite set of directions that are not enumerated;

LIGHTS: the standard set of Red, Green and Amber lights;

CONFLICT: a relation identifying directions that conflict with one another.

CONTEXT MultiWayCtx

SETS
LIGHTS
DIRECTION

CONSTANTS
Red
Green
Amber
CONFLICT

AXIOMS

axm1: \( \text{partition}(\text{LIGHTS}, \{\text{Red}\}, \{\text{Green}\}, \{\text{Amber}\}) \)

axm2: \( \text{finite}(\text{DIRECTION}) \)

axm3: \( \text{CONFLICT} \in \text{DIRECTION} \leftrightarrow \text{DIRECTION} \)

axm4: \( \text{CONFLICT} \cap \text{id} = \emptyset \)

axm5: \( \text{CONFLICT}^{-1} = \text{CONFLICT} \)

thm1: \( \forall d \cdot d \in \text{DIRECTION} \Rightarrow d \notin \text{CONFLICT}[[d]] \)

thm2: \( \forall d1, d2 \cdot d1 \notin \text{CONFLICT}[[d2]] \Rightarrow d2 \notin \text{CONFLICT}[[d1]] \)

END

Notation

\( r^{-1} \)  \( r^{-} \) inverse of \( r \), that is \( r \cdot r^{-1} \subseteq \text{id} \). Note: Rodin, even in marked-up form, retains the \( \sim \) notation

\( \notin \)  \( / \) not an element of; non-membership

\( r[s] \)  \( r[s] \) Relational image: \( r[s] \) is the set of values related to all elements of \( s \) under the relation \( r \)
Notes on *CONFLICT*

**axm3:** CONFLICT is a relation that relates all pairs of directions for which a *safety* invariant applies:

\[ \forall d_1, d_2. d_1 \mapsto d_2 \in \text{CONFLICT} \quad \text{(5.3)} \]

\[ \text{LIGHTS}[(d_1)] \in \{\text{Green, Amber}\} \Rightarrow \text{LIGHTS}(d_2) = \text{Red}; \quad \text{(5.4)} \]

**axm4:** (irreflexive) no direction can conflict with itself;

**axm5:** (symmetry) conflicts are symmetric: \( d_1 \text{ conflicts with } d_2 \Rightarrow d_2 \text{ conflicts with } d_1 \);

**thm1:** no direction \( d \) can be in the set of directions that conflict with \( d \). This follows from **axm4**, since if it weren’t true then the direction would conflict with itself.

**thm2:** the contrapositive of symmetry: \( d_2 \text{ does not conflict with } d_1 \Rightarrow d_1 \text{ does not conflict with } d_2 \).

### 5.2.1 The Initial Traffic Light model

Our initial model may look strange, as we are going to consider an initial state that has only *Red* and *Green* lights, and only events for changing *Red* to *Green* and vice-versa.

This models the sense in which those events are the primary events and changing lights from *Green* to *Amber* is a further expression of a safety constraint. An intersection in which lights were suddenly changed between *Red* and *Green* would be far from *safe*, despite our safety invariant.

Also in the interest of safety we will introduce time intervals between light changes.

**MACHINE** MultiWay

**SEES**

MultiWayCtx

**VARIABLES**

\begin{align*}
\text{lights} \\
\text{inv1:} \quad & \text{lights} \in \text{DIRECTION} \rightarrow \{\text{Red, Green}\} \\
\text{inv2:} \quad & \forall d \cdot d \in \text{DIRECTION} \land \text{lights}(d) = \text{Green} \\
& \Rightarrow \text{lights}[\text{CONFLICT}([d])] \subseteq \{\text{Red}\} \\
\text{inv3:} \quad & \text{finite}(\text{lights})
\end{align*}

**EVENTS**

**Initialisation** \( \triangleleft \)

**THEN**

\[ \text{lights} : |\text{lights}' \in \text{DIRECTION} \rightarrow \{\text{Red, Green}\} \]

\& \( \forall d \cdot d \in \text{DIRECTION} \land \text{lights}'(d) = \text{Green} \)

\Rightarrow \text{lights}'[\text{CONFLICT}([d])] \subseteq \{\text{Red}\})

**END**

**RedToGreen** \( \triangleleft \)

**ANY**

\( \text{adir} \)

**WHERE**

\[ \text{grd1:} \quad \text{lights}(\text{adir}) = \text{Red} \]

**THEN**

\[ \text{act1:} \quad \text{lights} := \text{lights} \perp (\text{CONFLICT}([\text{adir}]) \times \{\text{Red}\}) \perp \{\text{adir} \mapsto \text{Green}\} \]

**END**
ToRed ≜
ANY  
adir
WHERE  
grd1:  lights(adir) = Green
THEN  
act1:  lights(adir) := Red
END
END MultiWay

Notation
math ascii
⇒  ←+ Override: \( r \Leftarrow s \) yields the relation \( r \) overridden by the relation \( s \). As far as possible \( r \Leftarrow s \) behaves like \( s \): \( r \Leftarrow s = \text{dom}(s) \Leftarrow r \cup s \)

While the above machine preserves the safety invariant the intersection is not safe as lights are changed instantly from Green to Red and from Red to Green.

The data refinement MultiWayR will address that problem by introducing Amber between Green and Red, and also introducing a delay between all transitions. What we are doing is opening up the state to reveal more detail.

Thus, \( \text{lights} \) is refined to \( x\text{lights} \), \( \text{(extra lights)} \), that introduces Amber.

MACHINE MultiWayR
REFINES MultiWay
SEES MultiWayCtx
VARIABLES xlights delay rdir togreen tored

INVARIENTS
5.2. A MULTIWAY INTERSECTION

inv1: \[ \text{xlights} \in \text{DIRECTION} \to \text{LIGHTS} \]

inv2: \[ \forall d \cdot d \in \text{DIRECTION} \land \text{xlights}[d] \subseteq \{\text{Green}, \text{Amber}\} \]
\[ \Rightarrow \text{xlights}[\text{CONFLICT}[d]] \subseteq \{\text{Red}\} \]

inv3: \[ rdir \in \text{DIRECTION} \]

inv4: \[ \text{togreen} \in \text{BOOL} \]

inv5: \[ \text{tored} \in \text{BOOL} \]

inv6: \[ \text{togreen} = \text{TRUE} \Rightarrow \text{tored} = \text{FALSE} \]

inv7: \[ \Rightarrow \text{CONFLICT}[\{rdir\}] \subseteq \text{lights} \]
\[ = \text{CONFLICT}[\{rdir\}] \subseteq \text{xlights} \]

inv8: \[ \text{delay} \subseteq \text{DIRECTION} \]

inv9: \[ \text{tored} = \text{TRUE} \Rightarrow (\text{xlights} \ominus \{rdir \mapsto \text{Red}\} = \text{lights} \ominus \{rdir \mapsto \text{Red}\}) \]

inv10: \[ \text{togreen} = \text{FALSE} \land \text{tored} = \text{FALSE} \Rightarrow \text{lights} = \text{xlights} \]

thm1: \[ \text{finite}(\text{xlights}) \]
\[ \forall d, b, a \cdot d \in \text{DIRECTION} \]
\[ \land b \in \text{LIGHTS} \land a \in \text{LIGHTS} \land a \neq b \]
\[ \land \text{xlights}(d) = b \]
\[ \Rightarrow \]
\[ \text{card}((\text{xlights} \ominus \{d \mapsto a\}) \triangleright \{b\}) = \text{card}(\text{xlights} \triangleright \{b\}) - 1 \]

changing light in direction d from b (= before) to a (= after) decreases number of colour b lights by 1

thm2: \[ \forall d, b, a \cdot d \in \text{DIRECTION} \]
\[ \land b \in \text{LIGHTS} \land a \in \text{LIGHTS} \land a \neq b \]
\[ \land \text{xlights}(d) = b \]
\[ \Rightarrow \]
\[ \text{card}((\text{xlights} \ominus \{d \mapsto a\}) \triangleright \{a\}) = \text{card}(\text{xlights} \triangleright \{a\}) + 1 \]

changing light in direction d from b (= before) to a (= after) increases number of colour a lights by 1

thm3: \[ \forall d, b, a, c \cdot d \in \text{DIRECTION} \]
\[ \land b \in \text{LIGHTS} \land a \in \text{LIGHTS} \land c \in \text{LIGHTS} \]
\[ \land \text{xlights}(d) = b \land c \neq a \land c \neq b \]
\[ \Rightarrow \]
\[ \text{card}((\text{xlights} \ominus \{d \mapsto a\}) \triangleright \{c\}) = \text{card}(\text{xlights} \triangleright \{c\}) \]

changing light in direction d from b (= before) to a (= after) does not change number of colour c for c \neq a, c \neq b

**Notation**

```
\text{math}  \text{ascii}
\ominus  \langle \rangle  \text{Domain subtraction: } s \ominus r \text{ is the subset of } r \text{ in which } s \text{ has been subtracted from the domain of } r 
\triangleright  \langle \rangle  \text{Range restriction: } r \triangleright s \text{ is the subset of } r \text{ in which the range is restricted to the set } s
```
EVENTS
Initialisation $\cong$
WITH
  lights': $\equiv$ lights' = xlights'
THEN
  xlights :|
  xlights' $\in$ DIRECTION $\rightarrow$ \{Red, Green\}
  act1: $\land \\forall d \in$ DIRECTION $\land$ xlights'(d) = Green
       $\Rightarrow$
       xlights'[CONFLICT[{d}]] $\subseteq$ \{Red\})
  act2: delay := $\emptyset$
  act3: togreen, tored := FALSE, FALSE
  act4: rdir $\in$ DIRECTION
END

RedToGreen $\cong$
REFINES
RedToGreen
WHERE
  grd1: togreen $=$ TRUE
  grd2: xlights(rdir) = Red
  grd3: xlights[CONFLICT[{rdir}]] $\subseteq$ \{Red\}
  grd4: rdir $\notin$ delay
WITH
  adir: adir = rdir
THEN
  act1: xlights(rdir) := Green
  act2: togreen := FALSE
END

RedToGreenInit $\cong$
ANY
  adir
WHERE
  grd1: togreen $=$ FALSE
  grd2: tored $=$ FALSE
  grd3: xlights(adir) = Red
THEN
  act1: rdir := adir
  act2: togreen := TRUE
END

GreenToAmber $\cong$
STATUS Convergent
ANY
5.2. A MULTIWAY INTERSECTION

\[\text{dir} \]

\textbf{WHERE}
\begin{align*}
\text{grd1: } & \text{ togreen } = \text{ TRUE} \\
\text{grd2: } & \text{ dir } \in \text{ CONFLICT}[\{rdir\}] \\
\text{grd3: } & \text{ xlights(dir) } = \text{ Green} \\
\text{grd4: } & \text{ dir } \notin \text{ delay} \\
\end{align*}

\textbf{THEN}
\begin{align*}
\text{act1: } & \text{ xlights(dir) } := \text{ Amber} \\
\text{act2: } & \text{ delay } := \text{ delay } \cup \{\text{dir}\} \\
\end{align*}

\textbf{END}

\textbf{AmberToRed} \doteq \textbf{AmberToRed}

\textbf{STATUS} \text{ ordinary} \\
\text{convergent \text{ ANY}}

\textbf{dir} \textbf{WHERE}
\begin{align*}
\text{grd1: } & \text{ togreen } = \text{ TRUE} \\
\text{grd2: } & \text{ dir } \in \text{ CONFLICT}[\{rdir\}] \\
\text{grd3: } & \text{ xlights(dir) } = \text{ Amber} \\
\text{grd4: } & \text{ dir } \notin \text{ delay} \\
\end{align*}

\textbf{THEN}
\begin{align*}
\text{act1: } & \text{ xlights(dir) } := \text{ Red} \\
\text{act2: } & \text{ delay } := \text{ delay } \cup \{\text{rdir}\} \\
\end{align*}

\textbf{END}

\textbf{Delay} \doteq

\textbf{STATUS} \text{ ordinary} \\
\text{convergent \text{ ANY}}

\textbf{dir} \textbf{WHERE}
\begin{align*}
\text{grd1: } & \text{ dir } \in \text{ delay} \\
\end{align*}

\textbf{THEN}
\begin{align*}
\text{act1: } & \text{ delay } := \text{ delay } \setminus \{\text{dir}\} \\
\end{align*}

\textbf{END}

\textbf{ToRed} \doteq

\textbf{REFINES}
\begin{align*}
\textbf{ToRed} \\
\textbf{WHERE}
\begin{align*}
\text{grd1: } & \text{ tored } = \text{ TRUE} \\
\text{grd2: } & \text{ xlights(rdir) } = \text{ Amber} \\
\text{grd3: } & \text{ rdir } \notin \text{ delay} \\
\end{align*}

\textbf{WITH}
\begin{align*}
\text{adir : } & \text{ adir } = \text{ rdir} \\
\end{align*}
THEN
act1: \( x\text{lights}(r\text{dir}) := \text{Red} \)
act2: \( t\text{ored} := \text{FALSE} \)
END

ToRedInit \( \cong \)
ANY
\( \text{adir} \)
WHERE
grd1: \( x\text{lights}(\text{adir}) = \text{Green} \)
grd2: \( \text{tored} = \text{FALSE} \)
grd3: \( \text{togreen} = \text{FALSE} \)
THEN
act1: \( r\text{dir} := \text{adir} \)
act2: \( \text{tored} := \text{TRUE} \)
END

ToAmber \( \cong \)
WHERE
grd1: \( \text{tored} = \text{TRUE} \)
grd2: \( x\text{lights}(r\text{dir}) = \text{Green} \)
grd3: \( r\text{dir} \notin \text{delay} \)
THEN
act1: \( x\text{lights}(r\text{dir}) := \text{Amber} \)
act2: \( \text{delay} := \text{delay} \cup \{r\text{dir}\} \)
END

VARIANT
\( 4 \ast \text{card}(x\text{lights} \triangleright \{\text{Green}\}) + 2 \ast \text{card}(x\text{lights} \triangleright \{\text{Amber}\}) + \text{card}(\text{delay}) \)

END MultiWayR
Chapter 6

Event-B Semantics

This chapter presents the semantics of Event-B.
The various Proof Obligations (PO) that result from those semantics.
An understanding of “what those POs mean”.
The roles of POs in verifying a refinement.
The classification of POs, which identify what a particular PO is “all about”.

6.1 Semantics in Event B

- Each construct in B is given a formal semantics.
- Additionally, machines must satisfy a set of constraints.

These rules provide for

- the verification of the consistency of a machine;
- the verification that the behaviour of a refinement machine is consistent with the behaviour of the machine it refines.

Note that it is not possible to prove that the behavior of the initial abstract machine is correct, that is, conforms with the written requirements.

6.1.1 State Change

There are three principal constructions — that Event B calls substitutions — for changing the state of a machine:

\[ x := e \] \[ x \text{ becomes equal to the value of } e \]

This rule may be used recursively to assign to any number of variables.

\[ x : P \] \[ x \text{ becomes such that it satisfies the before-after predicate } P \]

\[ x : \in s \] \[ x \text{ becomes in the set } s \]
All of the above, except apparently :∈, can be extended to multiple assignment: \( x, y := e_1, e_2 \) and \( x, y : | P \), and recursively to many variables. The variables must be distinct.

Note: all assignments can be written in the form: \( x, y : | P \).

### 6.1.1.1 Before-After Predicates

Before-after predicates contain primed and unprimed variables, for example

\[
x' = x + 1
\]

where the primed variables represent the after value of a variable and the unprimed variables the before value.

Thus,

\[
x : | x' = x + 1
\]

and

\[
x := x + 1
\]

are equivalent.

Similarly we can write

\[
x, y : | x' = x + 1 \land y' = y + 1
\]

or

\[
x, y := x + 1, y + 1.
\]

### 6.1.2 Substitution

We will frequently need to compute, for example in computing POs, the weakest predicate on the state before a state given a required predicate on the after state.

We can do this by substituting into the after state.

We will write

\[
[x, y := e_1, e_2]R
\]

to denote the concurrent substitution of \( e_1 \) and \( e_2 \) for \( x \) and \( y \) in \( R \), respectively.

For example,

\[
[x, y := y - 1, x + 1]x - y < x + y = (y - 1) - (x + 1) < (y - 1) + (x + 1)
\]

or

\[
y - x - 2 < y + x
\]

This gives the weakest constraint on the before state such that \( x, y := y - 1, x + 1 \) will give an after state satisfying \( x - y < x + y \).
6.1.2.1 Other Forms of Substitution

For each of the 3 change of state substitutions, substitution into a predicate takes the following form:

\[ v : \in S \quad \forall v' \cdot v' \in S \Rightarrow [v := v'] R \]

where:

1. \( v \) in general is a list of variables, and \( E \) a list of expressions;
2. \( P \) is a predicate containing both \( v \) and \( v' \), where \( v' \) represents the value of \( v \) after the action.

6.1.3 Contexts

<table>
<thead>
<tr>
<th>Sets</th>
<th>( S )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constants</td>
<td>( C )</td>
</tr>
<tr>
<td>Axioms</td>
<td>( A )</td>
</tr>
<tr>
<td>Theorems</td>
<td>( T_c )</td>
</tr>
</tbody>
</table>

Contexts are used to define abstract carrier sets (\( S \)) and constants (\( C \)).

Notice that \( S \) and \( C \) are essentially extensions of the “builtin” sets such as \( \mathbb{N}, \mathbb{N}_1, \mathbb{Z} \) etc and constants from those sets, but we will elide any explicit extension.

6.1.3.1 Context Machines: Semantics & Proof Obligations

The semantics of the sets and constants are specified in the axioms. The essential proof obligations is one of feasibility: show that sets and constants exist that will satisfy the axioms. That is:

\[ (\exists S, C \cdot A) \]

where sub-axioms \( a_1, a_2, \ldots, a_n \) are effectively conjuncted into a single \( A \). The POs can be recursively split into separate POs based on

\[ (\exists S, C \cdot a_1 \land a_2 \land \ldots \land a_n) \]

\[ \equiv \exists S, C \cdot a_1 \land (\exists S, C \cdot a_1 \Rightarrow a_2 \land \ldots \land a_n) \]

*This may require the sub-axioms to be ordered*

Of course, components of \( S, C \) that are not referenced in \( a_i \) can be eliminated from \( \exists S, C \cdot a_i \).

6.1.3.2 Theorems

Theorems describe properties that follow from the axioms, so the general PO for the theorems is

\[ (\forall S, C \cdot A \Rightarrow T_c) \]
The theorems will, in general, be broken in sub-theorems \( t_1, t_2, \ldots, t_n \), and since universal quantification distributes through conjunction this breaks into multiple POs:

\[
(\forall S, C. A \Rightarrow t_1), \ldots, (\forall S, C. A \Rightarrow t_n)
\]

Thus, separate proof obligations can be generated for each theorem, however since the sub-theorems are usually distributed through the axioms (or invariants or guards depending on the context of the theorem), theorems must be placed after any axioms on which it depends.

### 6.1.4 Machines

The form of a machine is:

- **Context**: \( S, C \)
- **Variables**: \( V \)
- **Invariant**: \( I \)
- **Theorems**: \( T_v \)
- **Variant**: \( Var \)
- **Events**: \( E \)

#### 6.1.4.1 Machine POs: Invariant and Theorems

**The invariant** as for the axioms for context machines, the invariant may raise feasibility proof obligations:

\[
(\exists S, C \cdot A) \Rightarrow (\exists V \cdot I)
\]

**The theorems** must follow from the set/constant axioms and the invariant:

\[
\forall S, C, V \cdot A \land I \Rightarrow T_v
\]

**Note:** where we have \( A \) we could also have \( A \land T_c \), but since \( A \Rightarrow C \) this does not gain any extra strengthening.

#### 6.1.4.2 Initialisation

Initialisation, which is a special part of the events, must establish a state in which the variables satisfy the invariant.

Let us represent the initialisation by a multiple substitution

\[
V := E(S, C)
\]

where \( E(S, C) \) emphasises that the initialising expressions can only reference sets and constants: \( E \) must not reference any variables, since all variables at this point are undefined.
Then the proof obligation for initialisation is
\[ \forall S, C \cdot A \Rightarrow [V := E(S, C)] I \]

### 6.1.5 Events

Events have the following form

\[
\begin{array}{ll}
\text{ANY} & x \\
\text{WHERE} & G \\
\text{THEN} & \text{Action}
\end{array}
\]

#### 6.1.5.1 Event: Proof Obligations

There may be feasibility POs: that there exist parameters \( P \) that will satisfy the guards \( G \)
\[ \forall S, C \cdot A \land \exists V, x \cdot I \land G \]

#### 6.1.5.2 Event: Maintaining Invariant

The event must maintain the invariant of the machine: essentially the invariant will be true before the event is scheduled and must remain true when the event terminates.
\[ \forall S, C, V, x \cdot A \land I \land G \Rightarrow [\text{Action}]I \]

### 6.1.6 Machine Refinements

The form of a refinement machine is

<table>
<thead>
<tr>
<th>Context</th>
<th>( S_r, C_r )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variables</td>
<td>( V_r )</td>
</tr>
<tr>
<td>Invariant</td>
<td>( I_r )</td>
</tr>
<tr>
<td>Theorems</td>
<td>( T^+_v )</td>
</tr>
<tr>
<td>Events</td>
<td>( E_r ) refines ( E )</td>
</tr>
<tr>
<td>Variant</td>
<td>( Var )</td>
</tr>
</tbody>
</table>

where \( E_r \) represents a refined event and \( E \) represents new normal events.
6.1.6.1 Variables and Invariant

The variable $V_r$ are in general a superset of the variables in the machine being refined.
The invariant is the invariant of the refined machine plus invariants for the new variables. In addition the invariant contains the refinement relation relating the state of the refined machine to the variables of the refining machine. This gives a simulation relation.
The proof obligations for the variables, invariant and theorems are similar to those for the machine given above. We will concentrate on the new proof obligations that arise from the refined events.

6.1.6.2 Proof Obligations

\[ \forall V_i, V \cdot I_r \Rightarrow I \]
the new invariant must not allow behaviour that was not part of the refined machine’s behaviour, excepting where the state of the refining machine is “orthogonal” to the refined machine.

6.1.7 Refined Events

Refined Events have the following form

\[
\text{ANY } x_r \\
\text{WHERE } G_r \\
\text{WITH } w : W \\
\text{THEN } \text{Action}_r
\]

6.1.7.1 Proof Obligations for Refined Events

\[ \text{guard refinement} \]
\[ \forall S, C, S_r, C_r, V, V_r, x, x_r \cdot A \land A_r \land I \land I_r \Rightarrow (G_r \Rightarrow G) \]
\[ \text{witness} \]
\[ \forall S, C, S_r, C_r, V, V_r, x, x_r \cdot A \land A_r \land I \land I_r \Rightarrow (\exists w \cdot W) \]
\[ \text{Simulation} \]
\[ \forall S, C, S_r, C_r, V, V_r, x, x_r \cdot A \land A_r \land I \land I_r \land W \land [\text{Action}_r]I_r \Rightarrow [\text{Action}]I \]
where $A_r$ denotes the refinement axioms.

6.1.7.2 The Variant and Convergent Events

The variant ($Var$) is an expression that denotes either a finite set or a natural number.
The purpose of the variant is to show that all convergent events must terminate. This is achieved by showing that the size of the set, or the natural number value is strictly decreasing.

\[ \text{Natural number variant} \]
\[ \forall S, C, S_r, C_r, V, V_r, x, x_r \cdot A \land I \land I_r \land W \Rightarrow Var \in \mathbb{N} \]
\[ \forall S, C, S_r, C_r, V, V_r, x, x_r \cdot A \land I \land I_r \land W \Rightarrow [\text{Action}_r]Var < Var \]
6.2. ONE POINT RULE

Set variant
\[ \forall S, C, S_r, V, V_r, x, x_r \cdot A \land I \land I_r \land W \Rightarrow \text{finite } Var \]
\[ \forall S, C, S_r, V, V_r, x, x_r \cdot A \land I \land I_r \land W \Rightarrow \text{card}([Action_r]Var) < \text{card}(Var) \]

6.2 One Point Rule

Consider \( \forall x \cdot x \in X \land x = e \Rightarrow P(x) \).

For any \( x \) in \( S \), \( x = e \) is either true or false. If it is false then the universal quantification is trivially true; if it is true then the quantification reduces to \( P(e) \). So

\[ (\forall x \cdot x \in X \land x = e \Rightarrow P(x)) = P(e) \]

By a similar argument,

\[ (\exists x \cdot x \in X \land x = e \land P(x)) = P(e) \]

Strictly, each should be conjuncted with \( \exists x \cdot x \in X \land x = e \).
Chapter 7

Data Refinement: A Queue Model

This model of a simple queue explores data-refinement to a greater depth than in previous models.

The model explores refinement to concrete machines that are closely related to class models in object-oriented design, and are refined far enough to be directly translatable to code.

Todo: Lots! this chapter needs much more commentary.

7.1 Context for a queue

CONTEXT QueueContext
SETS
  TOKEN
  ITEM
CONSTANTS
  QUEUE
AXIOMS
  axm1:  finite(TOKEN)
  axm2:  finite(ITEM)
  axm3:  QUEUE = \{ q | q ∈ N → TOKEN ∧ finite q ∧ dom q = 1 .. card(q) \}
  thm1:  ∅ ∈ QUEUE
END

7.2 A Queue machine

MACHINE QueueA
SEES
  QueueContext
VARIABLES
CHAPTER 7. DATA REFINEMENT: A QUEUE MODEL

queuetokens tokens for currently queued items
queue the queue of tokens
queueitems a function for fetching the item associated with a token
qsize current size of queue

INVARINTS
inv1: queuetokens ⊆ TOKEN
inv2: queue ∈ QUEUE
inv3: qsize ∈ N
inv4: queue ∈ 1..qsize ↦ queuetokens
    ∀i, j · i ∈ dom(queue) ∧ i ≠ j
thm1: queue(i) ≠ queue(j)
thm2: queuetokens = ran(queue)
inv5: queueitems ∈ queuetokens ↦ ITEM
thm3: card(queue) = qsize
thm4: queue⁻¹ ∈ queuetokens ↦ 1..qsize
thm5: queuetokens ≠ ∅ ⇒ qsize ≠ 0

Notation
math ascii
↦ ↠+>> Partial surjection: a surjective function is an onto relation which maps to all
elements of the range.
⇒ >> Total bijection: a total bijective function is a one-to-one and onto relation which
maps all elements of the domain.

EVENTS
Initialisation ≡
THEN
act1: queuetokens := ∅
act2: queue := ∅
act3: qsize := 0
act4: queueitems := ∅
END

Enqueue ≡
ANY item qid
WHERE
grd1: item ∈ ITEM
grd2: qid ∈ TOKEN \ queuetokens
THEN
7.2. A QUEUE MACHINE

act1: \text{queuetokens} := \text{queuetokens} \cup \{\text{qid}\}

act2: \text{queue}(\text{qsize} + 1) := \text{qid}

act3: \text{queueitems}(\text{qid}) := \text{item}

act4: \text{qsize} := \text{qsize} + 1

\text{END}

\text{Dequeue} \equiv
\text{WHERE}
\text{grd1: } 0 < \text{qsize}
\text{THEN}
act1: \text{queue} : \text{queue}' \in 1..\text{qsize} - 1 \mapsto \text{queuetokens} \setminus \{\text{queue}(1)\}
\land (\forall i \in 1..\text{qsize} - 1 \Rightarrow \text{queue}'(i) = \text{queue}(i + 1))
act2: \text{queueitems} := \{\text{queue}(1)\} \triangleq \text{queueitems}
act3: \text{queuetokens} := \text{queuetokens} \setminus \{\text{queue}(1)\}
act4: \text{qsize} := \text{qsize} - 1
\text{END}

\text{Unqueue} \equiv
\text{ANY}
\text{qid}
\text{WHERE}
\text{grd1: } \text{qid} \in \text{queuetokens}
\text{thm1: } \text{qsize} \neq 0
\text{THEN}
\text{queue} : \text{queue}' \in 1..(\text{qsize} - 1) \mapsto \text{queuetokens} \setminus \{\text{qid}\}
\land (\text{qsize} = 1 \Rightarrow \text{queue}' = \emptyset)
\land (\text{qsize} > 1 \Rightarrow
(\forall i \in 1..\text{queue}^{-1}(\text{qid}) - 1 \Rightarrow \text{queue}'(i) = \text{queue}(i))
\land (\forall j \in \text{queue}^{-1}(\text{qid}) + 1..\text{qsize} \Rightarrow \text{queue}'(j - 1) = \text{queue}(j)))
act2: \text{queueitems} := \{\text{qid}\} \triangleq \text{queueitems}
act3: \text{queuetokens} := \text{queuetokens} \setminus \{\text{qid}\}
act4: \text{qsize} := \text{qsize} - 1
\text{END}

\text{END QueueA}
7.3 A Context that defines an abstract Queue datatype

```plaintext
CONTEXT QueueType
Check extension|| EXTENDS
    QueueContext
CONSTANTS
    ENQUEUE
    DEQUEUE
    DELETE
AXIOMS
```
7.3. A CONTEXT THAT DEFINES AN ABSTRACT QUEUE DATATYPE

axm1: \( ENQUEUE \in QUEUE \times TOKEN \rightarrow QUEUE \)

axm2: \( \forall q, t \cdot q \in QUEUE \land t \in TOKEN \land t \notin ran(q) \Rightarrow \text{card}(ENQUEUE(q \mapsto t)) = \text{card}(q) + 1 \)

thm1: \( \forall q, t \cdot q \in QUEUE \land t \in TOKEN \land t \notin ran(q) \Rightarrow \text{dom}(ENQUEUE(q \mapsto t)) = 1 \ldots \text{card}(q) + 1 \)

axm3: \( \forall q, t, i \cdot q \in QUEUE \land t \notin ran(q) \Rightarrow (i \in \text{dom}(q) \Rightarrow ENQUEUE(q \mapsto t)(i) = q(i)) \land (i = \text{card}(q) + 1 \Rightarrow ENQUEUE(q \mapsto t)(i) = t) \)

axm4: \( DEQUEUE \in QUEUE \rightarrow QUEUE \)

axm5: \( \text{dom}(DEQUEUE) = QUEUE \setminus \{\emptyset\} \)

axm6: \( \forall q \cdot q \in QUEUE \land q \neq \emptyset \Rightarrow DEQUEUE(q) \in 1 \ldots \text{card}(q) - 1 \mapsto \text{ran}(q) \setminus \{q(1)\} \)

axm7: \( \forall q \cdot q \in \text{dom}(DEQUEUE) \Rightarrow \text{card}(DEQUEUE(q)) = \text{card}(q) - 1 \)

thm2: \( \forall q \cdot q \in \text{dom}(DEQUEUE) \Rightarrow \text{dom}(DEQUEUE(q)) = 1 \ldots \text{card}(q) - 1 \)

axm8: \( \forall q, i \cdot q \in QUEUE \land i \in \text{dom}(q) \Rightarrow \text{DEQUEUE}(q \mapsto i) = q(i + 1) \)

axm9: \( DELETE \in QUEUE \times \mathbb{N}_1 \rightarrow QUEUE \)

axm10: \( \forall q, i \cdot q \in QUEUE \land i \in \text{dom}(q) \Rightarrow \text{DELETE}(q \mapsto i) \in 1 \ldots \text{card}(q) - 1 \mapsto \text{ran}(q) \setminus \{q(i)\} \)

axm11: \( \forall q, i \cdot q \in QUEUE \land i \in \text{dom}(q) \Rightarrow q \mapsto i \in \text{dom}(DELETE) \)

axm12: \( \forall q, i \cdot q \mapsto i \in \text{dom}(DELETE) \Rightarrow \text{card}(DELETE(q \mapsto i)) = \text{card}(q) - 1 \)

thm3: \( \forall q, i \cdot q \mapsto i \in \text{dom}(DELETE) \Rightarrow \text{dom}(DELETE(q \mapsto i)) = 1 \ldots \text{card}(q) - 1 \)

axm13: \( \forall q, i \cdot (j < i \land j \in \text{dom}(q) \Rightarrow DELETE(q \mapsto i)(j) = q(j)) \land (j \geq i \land j + 1 \in \text{dom}(q) \Rightarrow DELETE(q \mapsto i)(j) = q(j + 1)) \)

END
CHAPTER 7. DATA REFINEMENT: A QUEUE MODEL

7.4 A more abstract model

Machine QueueB is a refinement of QueueA using the abstract “methods” defined in QueueType. In fact, QueueA could also be refined from QueueB, so the two machines are equivalent models.

MACHINE QueueB
REFINES QueueA
SEES QueueType

VARIABLES
queuetokens tokens for currently queued items
queue the queue of tokens
queueitems a function for fetching the item associated with a token
qsize current size of queue

INVARIANTS
inv1: queuetokens ⊆ TOKEN
inv2: queue ∈ QUEUE
inv3: qsize = card(queue)
inv4: queue ∈ 1..qsize ⇒ queuetokens
∀i, j ∈ dom(queue) ∧ j ∈ dom(queue) ∧ i ≠ j
thm1: ⇒ queue(i) ≠ queue(j)

thm2: queuetokens = ran(queue)
inv5: queueitems ∈ queuetokens → ITEM
thm3: queue⁻¹ ∈ queuetokens ⇒ 1..qsize
(∀qid. qid ∈ TOKEN \ queuetokens
thm4: ⇒ ENQUEUE(queue ↦ qid) = queue ≜ {qsize + 1 ↦ qid})
∀qid. qid ∈ queuetokens

thm5: ⇒ queue ↦ queue⁻¹(qid) ∈ dom(DELETE)
qsize ≠ 1
⇒
thm6: (∀qid, i. qid ∈ queuetokens ∧ i ∈ 1..(queue⁻¹(qid) − 1)
⇒ (DELETE(queue ↦ queue⁻¹(qid)))(i) = queue(i)
qsize ≠ 1
⇒
thm7: (∀qid, i. qid ∈ queuetokens ∧ i ∈ queue⁻¹(qid) + 1..qsize
⇒ (DELETE(queue ↦ queue⁻¹(qid)))(i − 1) = queue(i)
∀qid. qid ∈ queuetokens

thm8: ⇒ queue⁻¹(qid) ≤ qsize

EVENTS
Initialisation ≜
THEN
7.4. A MORE ABSTRACT MODEL

act1: $\text{queuetokens} := \emptyset$
act2: $\text{queue} := \emptyset$
act3: $\text{qsize} := 0$
act4: $\text{queueitems} := \emptyset$

END

Enqueue $\equiv$
REFINES
Enqueue

ANY
item
qid
WHERE
  grd1: item \in ITEM
  grd2: qid \in TOKEN \setminus \text{queuetokens}
THEN
  act1: $\text{queuetokens} := \text{queuetokens} \cup \{qid\}$
  act2: $\text{queue} := \text{ENQUEUE}(\text{queue} \mapsto qid)$
  act3: $\text{queueitems}(qid) := \text{item}$
  act4: $\text{qsize} := \text{qsize} + 1$
END

Dequeue $\equiv$
REFINES
Dequeue

WHERE
  grd1: $\text{qsize} \neq 0$
THEN
  act1: $\text{queue} := \text{DEQUEUE}(\text{queue})$
  act2: $\text{queueitems} := \{\text{queue}(1)\} \ominus \text{queueitems}$
  act3: $\text{queuetokens} := \text{queuetokens} \setminus \{\text{queue}(1)\}$
  act4: $\text{qsize} := \text{qsize} - 1$
END

Unqueue $\equiv$
REFINES
Unqueue

ANY
qid
WHERE
  grd1: qid \in \text{queuetokens}
THEN
act1:  queue := DELETE(queue \(\mapsto\) queue\(^{-1}\) (qid))
act2:  queueitems := \{qid\} \(\leftarrow\) queueitems
act3:  queuetokens := queuetokens \(\setminus\) \{qid\}
act4:  qsize := qsize \(-\) 1

end

end QueueB
7.5 Changing the data representation

In the following the monolithic queue of the preceding models by a “linked” queue. This models the well-known linked structures familiar in software design and implementation.

In order to be able to demonstrate how the new model simulates the monolithic model the following are required:

**Relational composition:** if \( r_1 \) and \( r_2 \) are two relations over the same set \( X \) then \( r_1; r_2 \) is the forward composition of the two relations.

**Todo:** picture needed

**Relational closure:** is the union of all possible compositions of a relation with itself: \( r; r; \ldots; r \). This turns out to be finite and there are two versions of closure:

**reflexive:** in which the closure contains \( r^0 \) by definition, and

**irreflexive:** in which \( r^0 \) may be present, but is not present by definition.

**Todo:** much more discussion required

The context **Iteration** defines axioms and theorems for iteration and (irreflexive) closure.
axm1: \( \text{iterate} \in (\text{TOKEN} \leftrightarrow \text{TOKEN}) \times \mathbb{N} \rightarrow (\text{TOKEN} \leftrightarrow \text{TOKEN}) \)
\[ \forall r \cdot r \in \text{TOKEN} \rightarrow \text{TOKEN} \]

axm2: \[
\Rightarrow \text{iterate}(r \mapsto 0) = \text{TOKEN} \circ \text{id} \]
\[ \forall r, n \cdot r \in \text{TOKEN} \leftrightarrow \text{TOKEN} \land n \in \mathbb{N}_1 \]

axm3: \[
\Rightarrow \text{iterate}(r \mapsto n) = \text{iterate}(r \mapsto n - 1); r \]
\[ \forall s \cdot s \subseteq \mathbb{N} \land 0 \in s \]

thm1: 
\( (\forall n \cdot n \in s \Rightarrow n + 1 \in s) \Rightarrow \mathbb{N} \subseteq s \)
\[ \forall r, n \cdot r \in \text{TOKEN} \leftrightarrow \text{TOKEN} \land n \in \mathbb{N}_1 \]

thm2: \[
\Rightarrow \text{dom}(\text{iterate}(r \mapsto n)) \subseteq \text{dom}(r) \]

thm3: \[
\Rightarrow \text{ran}(\text{iterate}(r \mapsto n)) \subseteq \text{ran}(r) \]

axm4: \( \text{idclos} \in (\text{TOKEN} \leftrightarrow \text{TOKEN}) \rightarrow (\text{TOKEN} \leftrightarrow \text{TOKEN}) \)
\[ \forall r \cdot r \in \text{TOKEN} \leftrightarrow \text{TOKEN} \]

axm5: \[
\Rightarrow \text{idclos}(r) = (\bigcup n \cdot n \in \mathbb{N}_1 | \text{iterate}(r \mapsto n)) \]
\[ \forall r \cdot r \in \text{TOKEN} \leftrightarrow \text{TOKEN} \]

thm4: \[
\Rightarrow \text{dom}(\text{idclos}(r)) \subseteq \text{dom}(r) \]

END
7.5. CHANGING THE DATA REPRESENTATION

MACHINE QueueR
REFINES QueueB
SEES
Iteration
VARIABLES
queuetokens tokens currently in queue
queueitems a function for fetching the item associated with a token
qsize current size of queue
qfirst first item, if any, in queue
qlast last item, if any, in queue
qnext link to next item, if any, in queue
INVARs
inv1: $q_{first} \in \text{TOKEN}$
inv2: $q_{last} \in \text{TOKEN}$
inv3: $q_{size} \neq 0 \Rightarrow q_{first} = \text{queue}(1)$
inv4: $q_{size} \neq 0 \Rightarrow q_{last} = \text{queue}(q_{size})$
inv5: $q_{next} \in \text{queuetokens} \mapsto \text{queuetokens}$
inv6: $\text{dom}(q_{next}) = \text{queuetokens} \setminus \{q_{last}\}$
inv7: $q_{next} \cap \text{id} = \emptyset$
inv8: $\text{ran}(q_{next}) = \text{queuetokens} \setminus \{q_{first}\}$
thm1: $q_{size} = 1 \Rightarrow q_{first} = q_{last}$

$\forall i \cdot i \in 1..q_{size} \wedge i < q_{size}$

$\Rightarrow$

$q_{next}(\text{queue}(i)) = \text{queue}(i + 1)$
thm2: $q_{size} \geq 1 \Rightarrow \text{iterate}(q_{next} \mapsto 0)[\{q_{first}\}] = \{\text{queue}(1)\}$

$q_{size} \geq 1 \Rightarrow (\forall n \cdot n \in 1..q_{size} - 1 \wedge \text{iterate}(q_{next} \mapsto n - 1)[\{q_{first}\}] = \{\text{queue}(n)\}$
thm3: $\Rightarrow$

$\text{iterate}(q_{next} \mapsto n)[\{q_{first}\}] = \{\text{queue}(n + 1)\}$
thm4: $q_{size} \geq 1 \Rightarrow (\forall n \cdot n \in 1..q_{size} - 1 \Rightarrow \text{iterate}(q_{next} \mapsto n - 1)[\{q_{first}\}] = \{\text{queue}(n)\})$
thm5: $q_{size} \geq 1 \Rightarrow \text{iclosure}(q_{next})[\{q_{first}\}] = \text{queuetokens}$

Notation
math ascii
$\mapsto \mapsto \partial \inj$ partial injective function; injections are one-to-one relations

EVENTS
Initialisation $\equiv$

THEN
act1: $\text{queuetokens} := \emptyset$
act2: $q_{size} := 0$
act3: $\text{queueitems} := \emptyset$
act4: $q_{first} \in \text{TOKEN}$
act5: $q_{last} \in \text{TOKEN}$
act6: $q_{next} := \emptyset$
Enqueue0 ≡
\text{REFINES Enqueue}
\text{ANY item qid}
\text{WHERE}
\hspace{1em} \text{grd1: item} \in \text{ITEM}
\hspace{1em} \text{grd2: qid} \in \text{TOKEN} \setminus \text{queuetokens}
\hspace{1em} \text{grd3: qsize} = 0
\text{THEN}
\hspace{1em} \text{act1: queuetokens} := \text{queuetokens} \cup \{\text{qid}\}
\hspace{1em} \text{act2: queueitems(qid)} := \text{item}
\hspace{1em} \text{act3: qsize} := \text{qsize} + 1
\hspace{1em} \text{act4: qfirst} := \text{qid}
\hspace{1em} \text{act5: qlast} := \text{qid}
\text{END}

Enqueue1 ≡
\text{REFINES Enqueue}
\text{ANY item qid}
\text{WHERE}
\hspace{1em} \text{grd1: item} \in \text{ITEM}
\hspace{1em} \text{grd2: qid} \in \text{TOKEN} \setminus \text{queuetokens}
\hspace{1em} \text{grd3: qsize} \neq 0
\text{THEN}
\hspace{1em} \text{act1: queuetokens} := \text{queuetokens} \cup \{\text{qid}\}
\hspace{1em} \text{act2: queueitems(qid)} := \text{item}
\hspace{1em} \text{act3: qsize} := \text{qsize} + 1
\hspace{1em} \text{act4: qnext(qlast)} := \text{qid}
\hspace{1em} \text{act5: qlast} := \text{qid}
\text{END}

Dequeue0 ≡
\text{REFINES Dequeue}
\text{WHERE}
\hspace{1em} \text{grd1: qsize} = 1
\text{THEN}
7.5. CHANGING THE DATA REPRESENTATION

\begin{align*}
\text{act1: } & qsize := qsize - 1 \\
\text{act2: } & \text{queuetokens} := \text{queuetokens} \setminus \{qfirst\} \\
\text{act3: } & \text{queueitems} := \{qfirst\} \rhd \text{queueitems} \\
\text{act4: } & \text{qnext} := \{qfirst\} \setminus \text{qnext} \\
\end{align*}

END

Dequeue1 \equiv

\begin{align*}
\text{REFINES} & \quad \text{Dequeue} \\
\text{WHERE} & \quad \text{grd1: } qsize > 1 \\
\text{THEN} & \quad \text{act1: } qsize := qsize - 1 \\
& \quad \text{act2: } \text{queuetokens} := \text{queuetokens} \setminus \{qfirst\} \\
& \quad \text{act3: } \text{queueitems} := \{qfirst\} \rhd \text{queueitems} \\
& \quad \text{act4: } qfirst := \text{qnext}(qfirst) \\
& \quad \text{act5: } \text{qnext} := \{qfirst\} \setminus \text{qnext} \\
\end{align*}

END

Unqueue0 \equiv

\begin{align*}
\text{REFINES} & \quad \text{Unqueue} \\
\text{ANY} & \quad \text{qid} \\
\text{WHERE} & \quad \text{grd1: } \text{qid} \in \text{queuetokens} \\
& \quad \text{grd2: } qsize = 1 \\
\text{THEN} & \quad \text{act1: } \text{queueitems} := \{\text{qid}\} \rhd \text{queueitems} \\
& \quad \text{act2: } \text{queuetokens} := \text{queuetokens} \setminus \{\text{qid}\} \\
& \quad \text{act3: } qsize := qsize - 1 \\
\end{align*}

END

Unqueue1 \equiv

\begin{align*}
\text{REFINES} & \quad \text{Unqueue} \\
\text{ANY} & \quad \text{qid} \\
\text{WHERE} & \quad \text{grd1: } \text{qid} \in \text{queuetokens} \\
& \quad \text{grd2: } qsize > 1 \\
& \quad \text{grd3: } \text{qid} = qfirst \\
\text{THEN} & \quad \text{act1: } \text{queueitems} := \{\text{qid}\} \rhd \text{queueitems} \\
& \quad \text{act2: } \text{queuetokens} := \text{queuetokens} \setminus \{\text{qid}\} \\
& \quad \text{act3: } qsize := qsize - 1 \\
\end{align*}
CHAPTER 7. DATA REFINEMENT: A QUEUE MODEL

act1: \( \text{queueitems} := \{\text{qid}\} \triangleq \text{queueitems} \)
act2: \( \text{queuetokens} := \text{queuetokens} \setminus \{\text{qid}\} \)
act3: \( \text{qsize} := \text{qsize} - 1 \)
act4: \( \text{qfirst} := \text{qnext}(\text{qid}) \)
act5: \( \text{qnext} := \{\text{qid}\} \triangleleft \text{qnext} \)

END

Unqueue2 \( \cong \)
REFINES
Unqueue
ANY
qid
WHERE
grd1: \( \text{qid} \in \text{queuetokens} \)
grd2: \( \text{qsize} > 1 \)
grd3: \( \text{qlast} = \text{qid} \)
THEN
act1: \( \text{queueitems} := \{\text{qid}\} \triangleq \text{queueitems} \)
act2: \( \text{queuetokens} := \text{queuetokens} \setminus \{\text{qid}\} \)
act3: \( \text{qsize} := \text{qsize} - 1 \)
act4: \( \text{qlast} := \text{qnext}\cdot\setminus(\text{qid}) \)
act5: \( \text{qnext} := \text{qnext} \triangleright \{\text{qid}\} \)

END

Unqueue3 \( \cong \)
REFINES
Unqueue
ANY
qid
WHERE
grd1: \( \text{qid} \in \text{queuetokens} \)
grd2: \( \text{qsize} > 1 \)
grd3: \( \text{qfirst} \neq \text{qid} \)
grd4: \( \text{qlast} \neq \text{qid} \)
THEN
act1: \( \text{queueitems} := \{\text{qid}\} \triangleq \text{queueitems} \)
act2: \( \text{queuetokens} := \text{queuetokens} \setminus \{\text{qid}\} \)
act3: \( \text{qsize} := \text{qsize} - 1 \)
act4: \( \text{qnext}(\text{qnext}\cdot\setminus(\text{qid})) := \text{qnext}(\text{qid}) \)

END

END QueueR
7.6 Further refinement

While the current refinement can be considered to be close to a concrete model that would map reason-
ably easily into a concrete implementation there is one construct that cannot be considered as concrete:
\( q_{\text{next}}^{-1} \) in Unqueue3. This models a backward pointer, but it is mathematics, and cannot be considered
as concrete.

QueueRR, a further refinement of QueueR produces a concrete modelling of \( q_{\text{next}}^{-1} \), which is easily
seen to be a loop that searches for the queue item that preceded \( queue(qid) \).

MACHINE QueueRR
REFINES QueueR
SEES Iteration

VARIABLES
queuetokens tokens currently in queue
queueitems a function that maps tokens to items
qsize current size of queue
qfirst first item, if any, in queue
qlast last item, if any, in queue
qnext link to next item, if any, in queue
deleting Unqueue deletion in progress
qprev concrete version of queue
qidv copy of qid

INvariants
inv1: deleting \( \in \) BOOL
inv2: qprev \( \in \) TOKEN
inv3: qidv \( \in \) TOKEN
inv4: deleting \( = \) TRUE \( \Rightarrow \) qidv \( \in \) queuetokens
inv5: deleting \( = \) TRUE \( \Rightarrow \) qidv \( \neq \) qfirst
inv6: deleting \( = \) TRUE \( \Rightarrow \) qsize \( > \) 1
inv7: deleting \( = \) TRUE \( \Rightarrow \) qprev \( \in \) dom(qnext)
        deleting \( = \) TRUE
        \( \Rightarrow \)
        qidv \( \in \) iclosure(qnext)[{qprev}]

EVENTS
Initialisation : extended \( \triangleq \)
THEN
act7: deleting := FALSE
act8: qprev \( \in \) TOKEN
act9: qidv \( \in \) TOKEN
END

Enqueue0 : extended \( \triangleq \)
CHAPTER 7. DATA REFINEMENT: A QUEUE MODEL

**Enqueue0**

**Refines**

**Enqueue0**

**Any**

**Where**

grd4: deleting = FALSE

**Then**

**End**

**Enquire1 : extended \( \equiv \)**

**Refines**

Enquire1

**Any**

**Where**

grd4: deleting = FALSE

**Then**

**End**

**Dequque0 : extended \( \equiv \)**

**Refines**

Dequque0

**Any**

**Where**

grd2: deleting = FALSE

**Then**

**End**

**Dequeue1 : extended \( \equiv \)**

**Refines**

Dequeue1

**Where**

grd2: deleting = FALSE

**Then**

**End**

**Unqueue0 : extended \( \equiv \)**

**Refines**

Unqueue0

**Any**

**Where**

grd3: deleting = FALSE
7.6. FURTHER REFINEMENT

THEN

END

\textbf{Unqueue1} : \textit{extended} \equiv
\begin{aligned}
&\text{REFINES} \\
&\quad \text{Unqueue1} \\
&\text{ANY} \\
&\text{WHERE} \\
&\text{THEN} \\
&\text{END}
\end{aligned}

\begin{aligned}
&\text{Unqueue2} \equiv \\
&\text{REFINES} \\
&\quad \text{Unqueue2} \\
&\text{WHERE} \\
&\quad \text{grd1: } \text{deleting} = \text{TRUE} \\
&\quad \text{grd2: } \text{qnext} (\text{qprev}) = \text{qidv} \\
&\quad \text{grd3: } \text{qlast} = \text{qidv} \\
&\text{WITH} \\
&\quad \text{qid: } \text{qid} = \text{qidv} \\
&\text{THEN} \\
&\quad \text{act1: } \text{queueitems} := \{\text{qidv}\} \interface\text{queueitems} \\
&\quad \text{act2: } \text{queuetokens} := \text{queuetokens} \setminus \{\text{qid}\} \\
&\quad \text{act3: } \text{qsize} := \text{qsize} - 1 \\
&\quad \text{act4: } \text{qlast} := \text{qprev} \\
&\quad \text{act5: } \text{qnext} := \text{qnext} \setminus \{\text{qidv}\} \\
&\quad \text{act6: } \text{deleting} := \text{FALSE} \\
&\text{END}
\end{aligned}

\begin{aligned}
&\text{Unqueue3} \equiv \\
&\text{REFINES} \\
&\quad \text{Unqueue3} \\
&\text{WHERE} \\
&\quad \text{grd1: } \text{deleting} = \text{TRUE} \\
&\quad \text{grd2: } \text{qnext} (\text{qprev}) = \text{qidv} \\
&\quad \text{grd3: } \text{qidv} \neq \text{qlast} \\
&\text{WITH} \\
&\quad \text{qid} \quad \text{qid} = \text{qidv} \\
&\text{THEN}
\end{aligned}
act1: queueitems := \{qidv\} \leftarrow queueitems
act2: queuetokens := queuetokens \setminus \{qidv\}
act3: qsize := qsize - 1
act4: qnext(qprev) := qnext(qidv)
act5: deleting := FALSE

**END**

UnqueueI ≡  Initialise for search

ANY
qid

WHERE
grd1: qid ∈ queuetokens
grd2: qsize > 1
grd3: qfirst \neq qid
grd4: deleting = FALSE

THEN
act1: qprev := qfirst
act2: qidv := qid
act3: deleting := TRUE

**END**

UnqueueS ≡  Search for predecessor

STATUS convergent

WHERE
grd1: deleting = TRUE
grd2: qnext(qprev) \neq qidv

THEN
act1: qprev := qnext(qprev)

**END**

VARIANT

\text{iclosure}(qnext)(\{qprev\})

**END** QueueRR
Chapter 8

Lift System Modelling

Using layered refinements to develop a model for a lift system.

To learn the lessons of separation of concerns, and hence separation of functionality.

In this chapter we will build a small model of a lift system. Abstractly, a lift can have many incarnations, although most people probably think of something like the arrangement that we will model: a transport mechanism with doors and buttons, etc. You might be interested in [21]. Such lifts actually exist.

Because of the common reaction to the mention of a lift system, there is a strong temptation to introduce too much detail too early and to produce a model that is very difficult to understand. This defeats an important goal of modelling: to produce a model that can be reasoned about both informally and formally.

We will develop the model of the lift system through a number of refinement layers.

8.1 Basic Lift

The first layer, modelled by the BasicLift machine, is concerned with the basic rules for lift movement.

Basic Lift Attributes The first step will be to define the basis lift attributes:

What they are: distinguished informally by name;

What they do: how they modify the behaviour of a lift;

What are the parameters: what are the principal controlling parameters of the events;

When they run: the conditions under which the basic lift events can happen.

We will not be concerned with how these lift events might be controlled. At this level the only control is imposed by the guards of the events. This will enable us to establish the conditions under which these lift events are legal.

Of course, as this model develops there will be different manifestations of the basic events with strengthened guards and possibly extra parameters and actions.

LIFTS: there will be some finite set of lifts, modelled here by the finite set LIFT.
CHAPTER 8. LIFT SYSTEM MODELLING

STATUS: lifts will have a status. We conceive of three:

- **IDLE**: the lift is *inactive*, but capable of becoming active.
- **STOPPED**: the lift is *active* and stopped with doors closed;
- **WAITING**: the lift is *active*, stopped and waiting with doors open.
- **MOVING**: the lift is *active* and moving;

FLOOR: there will be some finite set of floors for each lift. In this model it is assumed that all lifts operate over the same set of floors. We will model the floors as a subrange $0 .. MAXFLOOR$, where $MAXFLOOR$ is at least 1, giving distinct top and bottom floors.

8.1.1 Lift Context

```
CONTEXT Lift_ctx
SETS
  DIRECTION
  STATUS
  LIFT
CONSTANTS
  MAXFLOOR
  FLOOR
  UP
  DOWN
  IDLE
  STOPPED
  MOVING
  WAITING
  CHANGE

AXIOMS
  axm1:  MAXFLOOR ∈ N
  axm2:  FLOOR = 0 .. MAXFLOOR
  axm3:  finite LIFT
  axm4:  partition(DIRECTION, {UP}, {DOWN})
  axm5:  partition(STATUS, {IDLE}, {STOPPED}, {WAITING}, {MOVING})
  axm6:  CHANGE ∈ DIRECTION ↠ DIRECTION
  axm7:  CHANGE = {UP ↦ DOWN, DOWN ↦ UP}

thm1:  FLOOR ≠ ∅
thm2:  finite FLOOR
thm3:  finite STATUS
thm4:  finite DIRECTION
thm5:  finite CHANGE
```

END
8.1.2 Basic Lift machine

The BasicLift machine models basic lift movements, and establishes basic lift constraints.

- The behaviour is nondeterministic:
- there is no attempt to express any sort of lift control or scheduling
  A discipline of lift direction is established:
  - level 0: direction is UP
  - level MAXFLOOR: direction is DOWN
  - other levels: either direction is valid.
- There are no doors.

MACHINE BasicLift
SEES Lift_ctx

VARIABLES
  liftposition
  liftstatus
  liftdirection

INVARIANTS
  inv1: liftposition ∈ LIFT → FLOOR
  thm1: finite liftposition
  inv2: liftstatus ∈ LIFT → STATUS
  thm2: finite liftstatus
  inv3: liftdirection ∈ LIFT → DIRECTION
  thm3: finite liftdirection
  inv4: ∀l ∈ LIFT ∧ liftposition(l) = 0 ⇒ liftdirection(l) = UP
  inv5: ∀l ∈ LIFT ∧ liftposition(l) = MAXFLOOR ⇒ liftdirection(l) = DOWN
  thm4: ∀l ∈ LIFT ∧ liftdirection(l) = DOWN ⇒ liftposition(l) ≠ 0
  thm5: ∀l ∈ LIFT ∧ liftdirection(l) = UP ⇒ liftposition(l) ≠ MAXFLOOR

EVENTS
Initialisation ≜

THEN
  act1: liftposition := LIFT × {0}
  act2: liftdirection := LIFT × {UP}
  act3: liftstatus := LIFT × {IDLE}

END
CHAPTER 8. LIFT SYSTEM MODELLING

Notation
\[ A \times B \text{ is the set of all maplets, } a \mapsto b, \text{ in which } a \in A \text{ and } b \in B \]

**IdleLift** \[ \triangleq \] Idle lifts cannot move

\[ \text{ANY} \]
\[ \text{lift} \]
\[ \text{WHERE} \]
\[ \text{grd1: liftstatus(lift) \ STopped} \]
\[ \text{THEN} \]
\[ \text{act1: liftstatus(lift) := IDLE} \]
\[ \text{END} \]

**ActivateLift** \[ \triangleq \] Ready an Idle lift to enable moving

\[ \text{ANY} \]
\[ \text{lift} \]
\[ \text{WHERE} \]
\[ \text{grd1: liftstatus(lift) \ IDLE} \]
\[ \text{THEN} \]
\[ \text{liftstatus(lift) \ liftstatus'} \in \text{LIFT} \rightarrow \text{STATUS} \land \]
\[ \text{act1: (liftstatus' = liftstatus} \ LIFT \rightarrow \text{STOPPED}) \lor \]
\[ \text{(liftstatus' = liftstatus \ \{lift \mapsto \text{WAITING}\})} \]
\[ \text{END} \]

**StartLift** \[ \triangleq \] Models starting of a stopped lift, maintaining the previous direction

\[ \text{ANY} \]
\[ \text{lift} \]
\[ \text{WHERE} \]
\[ \text{grd1: liftstatus(lift) = \text{STOPPED}} \]
\[ \text{THEN} \]
\[ \text{act1: liftstatus(lift) := MOVING} \]
\[ \text{END} \]

**ChangeDir** \[ \triangleq \] Models the changing of direction of a STOPPED lift

\[ \text{ANY} \]
\[ \text{lift} \]
\[ \text{WHERE} \]
\[ \text{grd1: liftstatus(lift) = \text{STOPPED}} \]
\[ \text{grd2: liftposition(lift) \neq 0} \]
\[ \text{grd3: liftposition(lift) \neq \text{MAXFLOOR}} \]
\[ \text{THEN} \]
\[ \text{act1: liftdirection(lift) := CHANGE(liftdirection(lift))} \]
\[ \text{END} \]
8.1. BASIC LIFT

\[ \text{MoveUp} \triangleq \text{Models a lift moving up to the next floor and continuing to move} \]

\[ \text{ANY lift} \]

\[ \text{WHERE} \]
\[ \text{grd1: liftstatus(lift)} = \text{MOVING} \]
\[ \text{grd2: liftdirection(lift)} = \text{UP} \]
\[ \text{grd3: liftposition(lift)} \neq \text{MAXFLOOR} - 1 \]

\[ \text{THEN} \]
\[ \text{act1: liftposition(lift)} := \text{liftposition(lift)} + 1 \]

\[ \text{END} \]

\[ \text{MoveUpAndStop} \triangleq \text{Models a lift moving up to the next floor and stopping} \]

\[ \text{ANY lift} \]

\[ \text{WHERE} \]
\[ \text{grd1: liftstatus(lift)} = \text{MOVING} \]
\[ \text{grd2: liftdirection(lift)} = \text{UP} \]

\[ \text{THEN} \]
\[ \text{act1: liftposition(lift)} := \text{liftposition(lift)} + 1 \]
\[ \text{liftdirection : liftdirection}' \in \text{LIFT} \rightarrow \text{DIRECTION} \]
\[ \land (\text{liftposition(lift)} + 1 = \text{MAXFLOOR}) \]
\[ \Rightarrow \]
\[ \text{act2: liftdirection}' = \text{liftdirection} \Leftrightarrow \{\text{lift} \mapsto \text{DOWN}\} \]
\[ \land (\text{liftposition(lift)} + 1 \neq \text{MAXFLOOR}) \]
\[ \Rightarrow \]
\[ \text{act3: (liftstatus(lift)} := \text{STOPPED} \]

\[ \text{END} \]

\[ \text{MoveDown} \triangleq \text{Models a lift moving down to the next floor and continuing to move} \]

\[ \text{ANY lift} \]

\[ \text{WHERE} \]
\[ \text{grd1: liftstatus(lift)} = \text{MOVING} \]
\[ \text{grd2: liftdirection(lift)} = \text{DOWN} \]
\[ \text{grd3: liftposition(lift)} \neq 1 \]

\[ \text{THEN} \]
\[ \text{act1: liftposition(lift)} := \text{liftposition(lift)} - 1 \]

\[ \text{END} \]

\[ \text{MoveDownAndStop} \triangleq \text{Models a lift moving down to the next floor and stopping} \]

\[ \text{ANY} \]
CHAPTER 8. LIFT SYSTEM MODELLING

lift
WHERE
  grd1: liftstatus(lift) = MOVING
  grd2: liftdirection(lift) = DOWN
THEN
  act1: liftposition(lift) := liftposition(lift) − 1
       liftdirection(lift) : liftdirection′ ∈ LIFT → DIRECTION
       ∨ (liftposition(lift) = 1
           ⇒ liftdirection′ = liftdirection)
       ∨ (liftposition(lift) + 1 ≠ 1
           ⇒ liftdirection′ = liftdirection)
  act2: liftdirection′ = liftdirection ⇐ {lift ′= UP}
       ∨ (liftposition(lift) + 1 ≠ 1
           ⇒ liftdirection′ = liftdirection)
  act3: (liftstatus(lift) := STOPPED
END

END BasicLift

The above model behaves like a normal lift, but the behaviour is completely nondeterministic; there is no way of influencing the behaviour. For example, there is no way to ensure a particular lift:

- moves;
- moves in a particular direction;
- stops at a particular floor.
8.2 Adding Lift Doors

In the next layer we add lift doors, satisfying the following requirements:

Safety: a lift door may be open only if the lift is stopped;

Opening: while the lift movement is still nondeterministic we require that when a lift stops at a floor then the door must open.

8.2.1 Door Context

\( \text{CONTEXT Doors_ctx} \)

\( \text{SETS} \)

\( \text{DOORS} \)

\( \text{CONSTANTS} \)

\( \text{CLOSED} \)

\( \text{OPENING} \)

\( \text{OPEN} \)

\( \text{CLOSING} \)

\( \text{AXIOMS} \)

axm1: \( \text{partition}(\text{DOORS}, \{\text{CLOSED}\}, \{\text{OPENING}\}, \{\text{OPEN}\}, \{\text{CLOSING}\}) \)

\( \text{END} \)

8.2.2 Lift Plus Doors

\( \text{MACHINE LiftPlusDoors} \)

\( \text{REFINES} \)

BasicLift

\( \text{SEES} \)

Lift_ctx

Doors_ctx

\( \text{VARIABLES} \)

liftposition

liftstatus

liftdirection

liftdoorstatus

\( \text{INVARINANTS} \)

inv1: \( \text{liftdoorstatus} \in \text{LIFT} \rightarrow \text{DOORS} \)

thm1: \( \text{finite(liftdoorstatus)} \)

\( \forall l \cdot l \in \text{LIFT} \land \text{liftstatus}(l) \in \{\text{MOVING, IDLE}\} \)

inv2: \( \Rightarrow \)

\( \text{liftdoorstatus}(l) = \text{CLOSED} \)

\( \forall l \cdot l \in \text{LIFT} \land \text{liftdoorstatus}(l) \in \{\text{OPENING, OPEN}\} \)

thm2: \( \Rightarrow \)

\( \text{liftstatus}(l) = \text{STOPPED} \)

\( \text{EVENTS} \)

\( \text{INITIALISATION} : \text{extended} \equiv \)

\( \text{THEN} \)

act4: \( \text{liftdoorstatus} := \text{LIFT} \times \{\text{CLOSED}\} \)

\( \text{END} \)
OpenLiftDoor ≡ Open lift door: lift must be STOPPED

\text{ANY}
\text{lift}
\text{WHERE}
\text{grd1: } \text{liftstatus(lift)} = \text{STOPPED}
\text{grd2: } \text{liftdoorstatus(lift)} = \text{OPENING}
\text{THEN}
\text{act1: } \text{liftdoorstatus(lift)} := \text{OPEN}
\text{END}

CloseLiftDoor ≡

\text{ANY}
\text{lift}
\text{WHERE}
\text{grd1: } \text{liftdoorstatus(lift)} = \text{OPEN}
\text{THEN}
\text{act1: } \text{liftdoorstatus(lift)} := \text{CLOSED}
\text{END}

IdleLift : \text{extended} ≡ Idle lifts cannot move

\text{REFINES}
\text{IdleLift}
\text{WHERE}
\text{grd2: } \text{liftdoorstatus(lift)} = \text{CLOSED}
\text{END}

ActivateLift : \text{extended} ≡ Ready an Idle lift to enable moving

\text{REFINES}
\text{ActivateLift}
\text{THEN}
\text{liftdoorstatus} : | \text{liftdoorstatus}' \in LIFT \rightarrow DOORS \land
\text{act2:}
\{(\text{liftdoorstatus}' = \text{liftdoorstatus} \land \{\text{lift} \mapsto \text{CLOSED}\}) \lor
\text{liftdoorstatus}' = \text{liftdoorstatus} \land \{\text{lift} \mapsto \text{OPENING}\})\}
\text{END}

StartLift : \text{extended} ≡

\text{REFINES}
\text{StartLift}
\text{WHERE}
\text{grd2: } \text{liftdoorstatus(lift)} = \text{CLOSED}
\text{END}

ChangeDir : \text{extended} ≡
8.2. ADDING LIFT DOORS

ChangeDir
END

\textbf{MoveUp : extended } \triangleq \text{ Models a lift moving up to the next floor and continuing to move}

\textbf{MoveUp : extended } \triangleq \text{ Models a lift moving up to the next floor and stopping}

\textbf{MoveDown : extended } \triangleq \text{ Models a lift moving down to the next floor and continuing to move}

\textbf{MoveDownAndStop : extended } \triangleq \text{ Models a lift moving down to the next floor and stopping}
8.2.3 Adding Floor Doors

In this layer we add floor doors with the following requirements:

1. The floor door opens AFTER the lift door opens;
2. Floor doors may be OPEN only on the floor where a lift is stopped;
3. If a lift is MOVING then the floor door for that lift is CLOSED on all floors;
4. The floor door OPEN implies the lift door OPEN.

MACHINE LiftPlusFloorDoors
REFINES LiftPlusDoors
SEES Lift_ctx, Doors_ctx
VARIABLES liftposition, liftstatus, liftdirection, liftdoorstatus, floordoorstatus
INVARs:
inv1: \( floordoorstatus \in LIFT \to (FLOOR \to DOORS) \)
inv2: \( \forall l \cdot l \in LIFT \land liftdoorstatus(l) \neq OPEN \)
inv3: \( \forall l, f \cdot l \in LIFT \land f \in FLOOR \setminus \{ liftposition(l) \} \)
inv4: \( \forall l \cdot l \in LIFT \land liftdoorstatus(l)(liftposition(l)) \neq CLOSED \)

thm1: \( \text{finite}(floordoorstatus) \)
\[ \forall l \cdot l \in LIFT \land liftdoorstatus(l) \neq OPEN \]
\[ \Rightarrow \quad \text{floordoorstatus}(l)(\text{liftposition}(l)) = CLOSED \]
\[ \forall l, f \cdot l \in LIFT \land f \in FLOOR \setminus \{ \text{liftposition}(l) \} \]
\[ \Rightarrow \quad \text{floordoorstatus}(l)(f) = CLOSED \]
\[ \forall l, f \cdot l \in LIFT \land f \in FLOOR \land \text{liftstatus}(l) = MOVING \]
\[ \Rightarrow \quad \text{floordoorstatus}(l)(f) = CLOSED \]
\[ \forall l \cdot l \in LIFT \land \text{floordoorstatus}(l)(\text{liftposition}(l)) \neq CLOSED \]
\[ \Rightarrow \quad \text{liftstatus}(l) = STOPPED \]

INITIALISATION: extended \( \triangleq \)

THEN
act5: \( \text{floordoorstatus} := LIFT \times \{ FLOOR \times \{ CLOSED \} \} \)

END

OpenFloorDoor \( \triangleq \)
ANY
lift
8.2. ADDING LIFT DOORS

WHERE
\begin{align*}
  & \text{grd1:} \quad \text{liftstatus}(\text{lift}) = \text{STOPPED} \\
  & \text{grd2:} \quad \text{liftdoorstatus}(\text{lift}) = \text{OPEN} \\
  & \text{grd3:} \quad \text{floordoorstatus}(\text{lift})(\text{liftposition}(\text{lift})) = \text{OPENING}
\end{align*}

\text{THEN}
\begin{align*}
  & \text{act1:} \quad \text{floordoorstatus}(\text{lift}) := \text{floordoorstatus}(\text{lift}) \ominus \{\text{liftposition}(\text{lift}) \mapsto \text{OPEN}\}
\end{align*}

\text{END}

\text{CloseFloorDoor} \triangleq \text{any lift}
\begin{align*}
  & \text{grd1:} \quad \text{floordoorstatus}(\text{lift})(\text{liftposition}(\text{lift})) = \text{OPEN} \\
\text{THEN}
  & \text{act1:} \quad \text{floordoorstatus}(\text{lift}) := \text{floordoorstatus}(\text{lift}) \ominus \{\text{liftposition}(\text{lift}) \mapsto \text{CLOSED}\}
\end{align*}

\text{END}

\text{OpenLiftDoor} : \text{extended} \triangleq \text{refines OpenLiftDoor}
\begin{align*}
  & \text{WHERE} \\
\text{THEN}
  & \text{act2:} \quad \text{floordoorstatus}(\text{lift}) := \text{floordoorstatus}(\text{lift}) \ominus \{\text{liftposition}(\text{lift}) \mapsto \text{OPENING}\}
\end{align*}

\text{END}

\text{CloseLiftDoor} : \text{extended} \triangleq \text{refines CloseLiftDoor}
\begin{align*}
  & \text{WHERE} \\
\text{grd2:} \quad \text{floordoorstatus}(\text{lift})(\text{liftposition}(\text{lift})) = \text{CLOSED}
\end{align*}

\text{END}

\text{StartLift} : \text{extended} \triangleq \text{refines StartLift}
\begin{align*}
  & \text{WHERE} \\
\text{grd2:} \quad \text{liftdoorstatus}(\text{lift}) = \text{CLOSED}
\end{align*}

\text{END}

\text{ChangeDir} : \text{extended} \triangleq \text{refines ChangeDir}
\text{END}
MoveUp : extended ≜ Models a lift moving up to the next floor
  REFINES MoveUp
END

MoveUpAndStop : extended ≜ Models a lift moving up to the next floor and stopping
  REFINES MoveUpAndStop
END

MoveDown : extended ≜ Models a lift moving down to the next floor
  REFINES MoveDown
END

MoveDownAndStop : extended ≜ Models a lift moving down to the next floor and stopping
  REFINES MoveDownAndStop
END

END LiftPlusFloorDoors

8.3 Adding Buttons

We will now add buttons to enable lift passengers to signal their requests: both inside the lifts and on the floors of the building.

8.3.1 Buttons inside Lift

8.3.2 Buttons Context

CONTEXT
SETS
  BUTTONS
CONSTANTS
  ON
  OFF
AXIOMS
  axm1: partition(BUTTONS, {ON}, {OFF})
END
8.3.3 Lift Buttons machine

In this layer we model passenger requests for lifts to stop at particular floors, and the consequent scheduling of the lift to stop at those floors. The following scheduling discipline is established:

servicing of floor requests in direction of travel: a lift services all existing requests in its direction of travel;

idle if no requests: if a lift has no current requests it becomes idle.

To manage the scheduling a lift schedule is associated with each lift. The lift schedule is modelled by a sets of floors for which there are requests. The lift schedule is more general than the requests recorded by lift buttons, thus allowing the lift schedule to be used to schedule other requests, for example from floor buttons on each floor (outside the lifts) by which passengers request lifts for travel in a particular direction.

MACHINE LiftButtons
REFINES LiftPlusFloorDoors

SEES Lift_ctx
Doors_ctx
Buttons_ctx

VARIABLES
liftposition
liftstatus
liftdirection
liftdoorstatus
floordoorstatus
liftbuttons
liftschedule

INVIANTS
inv1: \( \text{liftbuttons} \in \text{LIFT} \rightarrow (\text{FLOOR} \rightarrow \text{BUTTONS}) \)
inv2: \( \text{liftschedule} \in \text{LIFT} \rightarrow \mathbb{P}(\text{FLOOR}) \)

thm1: \( \forall l \in \text{LIFT} \Rightarrow \text{finite}(\text{liftschedule}(l)) \)
inv3: \( \forall l, f \in \text{dom(\text{liftbuttons}(l))) \land f \in \text{dom(\text{liftbuttons}(l)))} \)
\( \Rightarrow (\text{liftbuttons}(l)(f) = \text{ON} \Rightarrow f \in \text{liftschedule}(l)) \)

inv4: \( \forall l \in \text{LIFT} \land \text{liftposition}(l) \in \text{liftschedule}(l) \)
\( \Rightarrow \text{liftstatus}(l) = \text{STOPPED} \)

thm2: \( \forall l \in \text{LIFT} \land \text{liftstatus}(l) = \text{MOVING} \)
\( \Rightarrow \text{liftposition}(l) \notin \text{liftschedule}(l) \)

EVENTS

INITIALISATION: extended \( \equiv \)
THEN
act6: \( \text{liftbuttons} := \text{LIFT} \times \{\text{FLOOR} \times \{\text{OFF}\}\} \)
act7: \( \text{liftschedule} := \text{LIFT} \times \{\emptyset\} \)
END
**SelectFloor** ≜

\[\text{ANY} \quad \text{lift} \quad \text{floor} \]

**WHERE**
- grd1: \( \text{floor} \in \text{FLOOR} \)
- grd2: \( \text{liftbuttons}(\text{lift})(\text{floor}) = \text{OFF} \)
- grd3: \( \text{liftposition}(\text{lift}) \neq \text{floor} \)

**THEN**
- act1: \( \text{liftbuttons}(\text{lift}) := \text{liftbuttons}(\text{lift}) \cup \{ \text{floor} \mapsto \text{ON} \} \)
- act2: \( \text{liftschedule}(\text{lift}) := \text{liftschedule}(\text{lift}) \cup \{ \text{floor} \} \)

**END**

**MoveUp**: \textit{extended} ≜ Models a lift moving up to the next floor

**REFINES**
- MoveUp

**WHERE**
- grd4: \( \text{liftschedule}(\text{lift}) \neq \emptyset \)
- grd5: \( \text{liftposition}(\text{lift}) < \text{max(} \text{liftschedule}(\text{lift}) \text{)} \)
- grd6: \( \text{liftposition}(\text{lift}) + 1 \notin \text{liftschedule}(\text{lift}) \)

**END**

**MoveUpAndStop**: \textit{extended} ≜ Models a lift moving up to the next floor and stopping

**REFINES**
- MoveUpAndStop

**ANY**
- \text{lift}

**WHERE**
- grd1: \( \text{liftstatus}(\text{lift}) = \text{MOVING} \)
- grd2: \( \text{liftdirection}(\text{lift}) = \text{UP} \)
- grd3: \( \text{liftposition}(\text{lift}) + 1 \in \text{liftschedule}(\text{lift}) \)

**THEN**
- act1: \( \text{liftposition}(\text{lift}) := \text{liftposition}(\text{lift}) + 1 \)
- \( \text{liftdirection} : | \text{liftdirection}' \in \text{LIFT} \Rightarrow \text{DIRECTION} \)
- \( \wedge (\text{liftposition}(\text{lift}) + 1 = \text{MAXFLOOR} \Rightarrow \text{liftdirection}' = \text{liftdirection}) \)
- \( \wedge (\text{liftposition}(\text{lift}) + 1 \neq \text{MAXFLOOR} \Rightarrow \text{liftdirection}' = \text{liftdirection}) \)
- act2: \( \text{liftstatus}(\text{lift}) := \text{STOPPED} \)
- act3: \( \text{liftdoorstatus}(\text{lift}) := \text{OPENING} \)

**END**
8.3. ADDING BUTTONS

**MoveDown**: extended $\equiv$ Models a lift moving down to the next floor

REFINES
MoveDown

WHERE

- $\text{grd4: } \text{liftschedule}(\text{lift}) \neq \emptyset$
- $\text{grd5: } \text{liftposition}(\text{lift}) > \text{min}(\text{liftschedule}(\text{lift}))$
- $\text{grd6: } \text{liftposition}(\text{lift}) - 1 \notin \text{liftschedule}(\text{lift})$

END

**MoveDownAndStop**: Models a lift moving up to the next floor and stopping

REFINES
MoveDownAndStop

ANY
lift

WHERE

- $\text{grd1: } \text{liftposition}(\text{lift}) = 1$
- $\text{grd2: } \text{liftposition}(\text{lift}) = \text{min}(\text{liftposition}(\text{lift}))$
- $\text{grd3: } \text{liftposition}(\text{lift}) + 1 \in \text{liftposition}(\text{lift})$

THEN

- $\text{act1: } \text{liftposition}(\text{lift}) := \text{liftposition}(\text{lift}) + 1$
- $\text{liftdirection'} \in \text{LIFT} \rightarrow \text{DIRECTION} \land (\text{liftposition}(\text{lift}) = 1 \Rightarrow \text{liftdirection'})$
- $\text{act2: } \text{liftposition}(\text{lift}) = \text{min}(\text{liftposition}(\text{lift}))$
- $\text{act3: } \text{liftposition}(\text{lift}) := \text{STOPPED}$
- $\text{act4: } \text{liftdoorstatus}(\text{lift}) := \text{OPENING}$

END

**ActivateLiftClosed**: Ready an Idle lift to enable moving, but leave doors CLOSED

REFINES
ActivateLift

ANY
lift

WHERE

- $\text{grd1: } \text{liftposition}(\text{lift}) = \text{IDLE}$
- $\text{grd2: } \text{liftposition}(\text{lift}) \neq \emptyset$
- $\text{grd3: } \text{liftposition}(\text{lift}) \notin \text{liftposition}(\text{lift})$

THEN

- $\text{act1: } \text{liftposition}(\text{lift}) := \text{STOPPED}$
- $\text{act2: } \text{liftdoorstatus}(\text{lift}) := \text{OPENING}$

END
ActivateLiftOpen \overset{\text{REFINES}}{=} \text{Ready an Idle lift to enable moving, but commence opening doors}\\
\text{ANY lift}\\
\text{WHERE}\\
grd1: \quad \text{liftstatus(lift)} = \text{IDLE}\\
grd2: \quad \text{liftschedule(lift)} \neq \emptyset\\
grd3: \quad \text{liftposition(lift)} \notin \text{liftschedule(lift)}\\
\text{THEN}\\
act1: \quad \text{liftstatus(lift)} := \text{STOPPED}\\
act2: \quad \text{liftdoorstatus} := \text{liftdoorstatus} \setminus \{\text{lift} \mapsto \text{OPENING}\}\\
\text{END}\\

ExtendLiftSchedule \overset{\text{REFINES}}{=} \text{Extend the lift schedule}\\
\text{ANY lift floor}\\
\text{WHERE}\\
grd1: \quad \text{lift} \in \text{LIFT}\\
grd2: \quad \text{floor} \in \text{FLOOR}\\
grd3: \quad \text{liftposition(lift)} \neq \text{floor}\\
\text{THEN}\\
act1: \quad \text{liftschedule(lift)} := \text{liftschedule(lift)} \cup \{\text{floor}\}\\
\text{END}\\

ContractLiftSchedule \overset{\text{REFINES}}{=} \text{Remove floor from lift schedule}\\
\text{ANY lift floor}\\
\text{WHERE}\\
grd1: \quad \text{lift} \in \text{LIFT}\\
grd2: \quad \text{floor} \in \text{FLOOR}\\
grd3: \quad \text{floor} \in \text{liftschedule(lift)}\\
grd4: \quad \text{liftbuttons(lift)(floor)} = \text{OFF}\\
\text{THEN}\\
act1: \quad \text{liftschedule(lift)} := \text{liftschedule(lift)} \setminus \{\text{floor}\}\\
\text{END}\\

OpenFloorDoor : \text{extended} \overset{\text{REFINES}}{=} \text{OpenFloorDoor}\\
\text{WHERE}\\
grd4: \quad \text{liftposition(lift)} \in \text{liftschedule(lift)}
8.3. ADDING BUTTONS

CloseFloorDoor : extended \equiv
REFINES
CloseFloorDoor
THEN
act2: \textit{liftschedule}(lift) := \textit{liftschedule}(lift) \setminus \textit{liftposition}(lift)
act3: \textit{liftbuttons}(lift) := \textit{liftbuttons}(lift) \leftarrow \{\textit{liftposition}(lift) \mapsto \text{OFF}\}
END

OpenLiftDoor : extended \equiv
REFINES
OpenLiftDoor
WHERE
grd3: \textit{liftposition}(lift) \in \textit{liftschedule}(lift)
END

CloseLiftDoor : extended \equiv
REFINES
CloseLiftDoor
END

IdleLift : extended \equiv \text{Idle lifts cannot move}
REFINES
IdleLift
WHERE
grd3: \textit{liftposition}(lift) \in \textit{liftposition}(lift)
END

StartLift : extended \equiv
REFINES
StartLift
WHERE
grd3: \textit{liftposition}(lift) \notin \textit{liftposition}(lift)
  liftdirection(lift) = DOW N
  \Rightarrow \textit{liftposition}(lift) > \textit{min}(\textit{liftposition}(lift))
  liftdirection(lift) = UP
  \Rightarrow \textit{liftposition}(lift) < \textit{max}(\textit{liftposition}(lift))
  liftposition(lift) \notin \textit{liftposition}(lift)
END
\textbf{ChangeDir} : \textit{extended} \triangleq \textbf{REFINES} \textbf{ChangeDir}
\textbf{WHERE}
\item \textbf{grd4:} \textit{liftschedule}(\textit{lift}) \neq \emptyset
\item \textit{liftdirection}(\textit{lift}) = \textit{UP}
\item \textbf{grd5:} \Rightarrow \textit{liftposition}(\textit{lift}) > \textit{max}(\textit{liftschedule}(\textit{lift}))
\item \textit{liftdirection}(\textit{lift}) = \textit{DOWN}
\item \textbf{grd6:} \Rightarrow \textit{liftposition}(\textit{lift}) < \textit{min}(\textit{liftschedule}(\textit{lift}))
\textbf{END}

\textbf{END \textbf{LiftButtons}}

\textbf{8.3.4 Floor Buttons}

The next layer refines \textit{LiftButtons} to model floor requests and their scheduling. This machine is left as an exercise for the reader.
Chapter 9

Proof Obligations

All proof obligations have a name and an abbreviation:

<table>
<thead>
<tr>
<th>Id</th>
<th>Name</th>
<th>Needed to discharge</th>
</tr>
</thead>
<tbody>
<tr>
<td>INV</td>
<td>Invariant</td>
<td></td>
</tr>
<tr>
<td>FIS</td>
<td>Feasibility</td>
<td></td>
</tr>
<tr>
<td>WD</td>
<td>Well-definedness</td>
<td></td>
</tr>
<tr>
<td>GRD</td>
<td>Guard</td>
<td></td>
</tr>
<tr>
<td>EQL</td>
<td>Equal</td>
<td></td>
</tr>
</tbody>
</table>
Chapter 10

Exercises

10.1 Relations and Functions

You need to gain familiarity with the various types of relations used in B, as these will dominate the models you will be building. There are a confusingly large number of arrows that you will need to master.

A relation is simply a set of pairings between two sets, for example between FRIENDS and their PHONE numbers. We might have a set of such pairings in the set phone, which we might declare as

\[\text{phone} \in \text{FRIENDS} \leftrightarrow \text{PHONE}\]

In Event-B a pair is denoted using the maps to symbol \(\mapsto\), for example

\[\text{phone} = \{\text{jim} \mapsto 0456123456, \text{lisa} \mapsto 0423456234, \text{jim} \mapsto 0293984321, \ldots\}\]

Notice that relations can be many to many, there can be many friends mapping to telephone numbers, but also each friend may have many telephone numbers.

Functions are many to one, meaning that there are many things, but each thing can map to only one value. You’ve met functions in mathematics and maybe other places.

There are two basic sort of functions:

**Partial functions:** \(f \in X \mapsto Y\), a function that may not be defined everywhere in \(X\), for example a function between friend and their partner.

**Total functions:** \(f \in X \rightarrow Y\), a function that is defined everywhere in \(X\), for example the function that maps each number to its square.

Having got that far we don’t leave it there. We further restrict functions as follows:

**Injective functions:** \(f \in X \mapsto Y\), or \(f \in X \rightarrow Y\), one to one functions, where \(f(x) = f(y)\) only if \(x = y\), for example a function from person and their licence number, assuming that each person is uniquely identified.

**Surjective function:** \(f \in X \mapsto Y\), or \(f \in X \rightarrow Y\), onto functions, where each value of \(Y\) is equal to \(f(x)\) for some value of \(x\) in \(X\).

**Bijective functions:** \(f \in X \mapsto Y\), one-to-one and onto functions.

It is important to recognise and use these relationships when developing a model.
10.1.1 Exercises

Investigate the relationships for the following:

1. the sibling relationship between people;
2. the brother and sister relationships between people;
3. the relationship between people and their cars;
4. the relationship between people and registration plates;
5. relationships in student enrolment at UNSW;
6. the relationship between coin denominations and their value;
7. the relationship that describes the coins you have in your pocket;
8. relationships concerning products on a supermarket shelf;
9. the relationship between courses and lecturers.

10.1.2 More on Sets

These exercises are intended to familiarise you with set concepts and the way EventB uses sets to model mathematical concepts. The tutorial also introduces EventB notation.

It is recommended that these exercise should be done in conjunction with the B Concise Summary. Also, while notation needs to be understood and this involves semantics, it is recommended that the reasoning about expressions should be conducted syntactically.

In this tutorial we also use single letters, which we will call jokers (from Classical-B), to represent arbitrary expressions and we utilise the notation of EventB proof theories (rules) for expressing properties. Thus when we say, “let \( S \) be a set”, \( S \) is an expression, which in this case must be a set expression, for example \( \text{members} \cup \{\text{newmember}\} \). You should not think of single letters as being variables.

A rule has the form \( P \Rightarrow Q \), stating that if we know \( P \) is true, then \( Q \) is true. For example,

\[
A \subseteq B \land a \in A \Rightarrow a \in B.
\]

Notice that while rules look like predicates, the elements of the rule are not typed, for example in the above rule \( A \) and \( B \) are both sets and their types must be compatible, otherwise \( A \subseteq B \) would not be defined. The rules are higher order logic, not first-order as used in EventB machines.

The use of the joker and proof rule notation allows us to say things about arbitrary expressions so long as they are well-typed.

**Simple sets**  The basis of EventB is simple sets. A set is an unordered collection of things, without multiplicity. The only property of sets is membership: we can evaluate \( x \in X \), “\( x \) is a member of \( X \)”.

Finite sets have cardinality, \( \text{card}(S) \), the number of elements in \( S \). Infinite sets do not have cardinality; EventB does not have an infinity.
10.1. RELATIONS AND FUNCTIONS

Powersets  From a simple set $S$ we can form the powerset of $S$, written $\mathcal{P}(S)$, which is the set of all subsets of $S$. We can define $\mathcal{P}(S)$ using set comprehension:

$$\mathcal{P}(S) = \{ s \mid s \subseteq S \}$$

We could also use a symmetric rule to express a property of powersets

$$p \in \mathcal{P}(P) \Rightarrow p \subseteq P$$

$$p \subseteq P \Rightarrow p \in \mathcal{P}(P)$$

Also,

$$S \in \mathcal{P}(S)$$

Products  Given two sets $S$ and $T$ we can form the product of $S$ and $T$, sometimes called the Cartesian product denoted $S \times T$. The product is the set of ordered pairs taken respectively from $S$ and $T$:

$$S \times T = \{ x, y \mid x \in S \land y \in T \}$$

A rule for products is

$$a \mapsto b \in A \times B \Rightarrow a \in A \land b \in B$$

The following sets are used in the exercises:

$$NAMES = \{ Jack, Jill \};$$

$$PHONE = \{ 123, 456, 789 \}$$

10. In this question you will be dealing with products, or sets of pairs. Instead of writing a pair as $(a, b)$, which is probably what you would normally do, write them as $a \mapsto b$, where $\mapsto$ is pronounced “maps to”.

j) What is $NAMES \times PHONE$?

k) What might it represent (model)?

l) What is $\text{card}(NAMES \times PHONE)$?

m) What is $\text{card}(NAMES \times \{ \})$?
	n) What is $\mathcal{P}(S)$?

o) Given $\text{card}(S) = N$, what is $\text{card}(\mathcal{P}(S))$?

p) What is $\text{card}(\mathcal{P}(NAMES \times PHONE))$?

q) What does $\text{card}(\mathcal{P}(NAMES \times PHONE))$ give you?

r) Is $NAMES \times PHONE$ a function?

s) Give a functional subset.

t) Give a total functional subset.

u) If a subset $S$ is described as a partial functional set, which of the following is correct?

   i. $S$ is not a total functional set.
   
   ii. $S$ might not be a total functional set

All of the following classes of functions may be total or not total.

v) Give an injective functional subset.

w) Give a surjective functional subset.

x) Give a (total) bijective functional subset.
Relations  Any subset of $X \times Y$ is called a (many-to-many) relation. The set of all relations between $X$ and $Y$ is denoted $X \leftrightarrow Y$. Since each relation is an element of $X \times Y$, it follows that $X \leftrightarrow Y = \mathcal{P}(X \times Y)$. This could be expressed by a rule:

$$\forall r \in X \leftrightarrow Y \Rightarrow r \in \mathcal{P}(X \times Y).$$

Domain and range  Given a relation $r$, where $r \in X \leftrightarrow Y$ then the domain of $r$, $\text{dom}(r)$, is the subset of $X$ for which a relation is defined. The range of $r$, $\text{ran}(r)$ is the subset of $Y$ onto which the $\text{dom}(r)$ is mapped. Here are some rules:

$$\forall r \in X \leftrightarrow Y \Rightarrow \text{dom}(r) \subseteq X \wedge \text{ran}(r) \subseteq Y,$$

$$\forall r \in X \leftrightarrow Y \wedge x \mapsto y \in r \Rightarrow x \in \text{dom}(r) \wedge y \in \text{ran}(r).$$

11. Given $\text{phonebook} \in \text{NAMES} \leftrightarrow \text{PHONE}$,

   a) Give some examples of $\text{phonebook}$.
   b) Give $\text{NAMES} \leftrightarrow \text{PHONE}$.
   c) What is $\text{card}(\text{NAMES} \leftrightarrow \text{PHONE})$?

Relational inverse  The relational inverse of $r$, $r^{-1}$, is the relation produced by inverting the mappings within $r$

$$\forall r \in X \leftrightarrow Y \Rightarrow y \mapsto x \in r \Rightarrow x \mapsto y \in r^{-1}.$$

Domain and range restriction  Domain (range) restriction restricts the domain (range) of a relation. $s \triangleleft r$ is the relation $r$ domain restricted to $s$. This gives a subset of the relation $r$ whose domain is a subset of $s$:

$$\forall r \in X \leftrightarrow Y \wedge s \subseteq X \Rightarrow s \triangleleft r \subseteq r \wedge \text{dom}(s \triangleleft r) \subseteq s,$$

$r \triangleright s$ is the relation $r$ range restricted to $s$. This gives a subset of the relation $r$ whose range is a subset of $s$:

$$\forall r \in X \leftrightarrow Y \wedge s \subseteq Y \Rightarrow r \triangleright s \subseteq r \wedge \text{ran}(r \triangleright s) \subseteq s.$$
10.1. RELATIONS AND FUNCTIONS

d) What is \{Jack\} \textless phonebook?

e) What is \{Jack\} \textless phonebook?

f) What is phonebook \textbb{123, 789}?

g) What is phonebook \textbb{123, 789}?

h) What is phonebook([(Jack)])?

**Functions**  Functions are many-to-one relations. $X \rightarrow Y$ is the set of all partial functions formed from $X$ and $Y$. A many-to-one relation is one where each element of the domain maps to only one value in the range, as illustrated by the following rule:

$$f \in X \rightarrow Y \land x \mapsto u \in f \land x \mapsto v \Rightarrow u = v$$

Partial functions are the most general form of function. For every $x$ in the domain of a function $f$ ($x \in \text{dom}(f)$) we can write $f(x)$ to obtain the value $x$ maps to under $f$, that is

$$f \in X \rightarrow Y \land x \mapsto y \Rightarrow f(x) = y$$

13. a) Give $\text{NAMES} \rightarrow \text{PHONE}$.

   b) What is $\text{card}($\text{NAMES} \rightarrow \text{PHONE}$)?

**Total functions**  $X \rightarrow Y$ is the set of all total functions formed from $X$ and $Y$. Total functions are (partial) functions with maximal domains:

$$f \in X \rightarrow Y \Rightarrow \text{dom}(f) = X$$

14. a) Give $\text{NAMES \rightarrow PHONE}$.

   b) What is $\text{card}($\text{NAMES \rightarrow PHONE}$)?

**Partial injective functions**  $X \rightarrow Y$ is the set of all partial, injective functions formed from $X$ and $Y$. An injective function is a one-to-one relation:

$$f \in X \rightarrow Y \land u \mapsto y \in f \land v \mapsto y \in f \Rightarrow u = v$$

15. a) Give $\text{NAMES \rightarrow PHONE}$.

   b) What is $\text{card}($\text{NAMES \rightarrow PHONE}$)?

**Total injective functions**  $X \rightarrow Y$ is the set of all total, injective functions formed from $X$ and $Y$. A total injective function is both total and injective:

$$f \in X \rightarrow Y \Rightarrow f \in X \rightarrow Y \land f \in X \rightarrow Y$$

16. a) Give $\text{NAMES \rightarrow PHONE}$.

   b) What is $\text{card}($\text{NAMES \rightarrow PHONE}$)?
Surjective functions \( X \mapsto Y \) is the set of all partial, surjective functions formed from \( X \) and \( Y \). A surjective function is a functional onto relations; a function whose range is maximal:

\[
f \in X \mapsto Y \Rightarrow \text{ran}(f) = Y
\]

17. a) Give \( \text{NAMES} \mapsto \text{PHONE} \).
    b) What is \( \text{card}(\text{NAMES} \mapsto \text{PHONE})? \)

Total surjective functions \( X \mapsto Y \) is the set of all total, surjective functions formed from \( X \) and \( Y \). A total surjective function is both total and surjective:

\[
f \in X \mapsto Y \Rightarrow f \in X \mapsto Y \land f \in X \mapsto Y
\]

18. a) Give \( \text{NAMES} \mapsto \text{PHONE} \).
    b) What is \( \text{card}(\text{NAMES} \mapsto \text{PHONE})? \)
    c) Why is a partial bijection unnecessary?

Bijective functions \( X \mapsto Y \) is the set of all (total) injective and surjective functions formed from \( X \) and \( Y \). A bijective function is total, injective and surjective.

\[
f \in X \mapsto Y \Rightarrow f \in f \mapsto Y \land f \in f \mapsto Y \land f \in f \mapsto Y
\]

19. a) Give \( \text{NAMES} \mapsto \text{PHONE} \).
    b) What is \( \text{card}(\text{NAMES} \mapsto \text{PHONE})? \)
    c) Why is a partial bijection unnecessary?
    d) Specify the drivers who are currently assigned.

20. Suppose \( \text{STUDENTS} \) is the set of all students that could be enrolled in a particular course. Students pass a course if they gain at least 50 marks in the final examination. Given a function \( \text{results} \in \text{STUDENTS} \mapsto \mathbb{N} \), that yields the examination result for a particular student, specify

    a) the set of students that pass;
    b) the set of students that fail.

21. If we were modelling a taxi fleet company we might have three variables, \( \text{drivers}, \text{taxis} \) and \( \text{assigned} \) constrained by

\[
\begin{align*}
\text{drivers} & \in \mathcal{P}(\text{DRIVERS}) \\
\text{taxis} & \in \mathcal{P}(\text{TAXIS}) \\
\text{assigned} & \in \text{drivers} \mapsto \text{taxis}
\end{align*}
\]

where \( \text{DRIVERS} \) is the set of possible drivers, \( \text{TAXIS} \) is the set of possible taxis, \( \text{drivers} \) is the set of drivers working for the company, \( \text{taxis} \) is the set of taxis owned by the company, and \( \text{assigned} \) is a function recording the assignment of drivers to taxis.

The arrow \( \text{inj} \) denotes a partial injective function. An injective function is a one-to-one function.

    a) Why is \( \text{assigned} \) a function?
    b) Why is \( \text{assigned} \) a partial function?
    c) Why is \( \text{assigned} \) an injective function?
    d) Specify the drivers who are currently assigned.
10.1. RELATIONS AND FUNCTIONS

e) Specify the drivers who are currently unassigned.
f) Specify the taxis that are currently assigned.
g) Specify the taxis that are currently unassigned.

22. Are the following rules correct or incorrect?
   a) \( f \in X \rightarrow Y \Rightarrow f \in X \rightarrow Y \)
   b) \( f \in X \rightarrow Y \Rightarrow f \in X \rightarrow Y \)
   c) \( f \in X \rightarrow Y \Rightarrow f \in X \rightarrow Y \)
   d) \( f \in X \rightarrow Y \Rightarrow f \in X \rightarrow Y \)
   e) \( f \in X \rightarrow Y \Rightarrow f \in X \rightarrow Y \)
   f) \( f \in X \rightarrow Y \Rightarrow f \in X \rightarrow Y \)
   g) \( f \in X \rightarrow Y \Rightarrow f \in X \rightarrow Y \)
   h) \( f \in X \rightarrow Y \Rightarrow f \in X \rightarrow Y \)
   i) \( f \in X \rightarrow Y \Rightarrow f \in X \rightarrow Y \)
   j) \( f \in X \rightarrow Y \Rightarrow \dom(f) \subset X \)
   k) \( f \in X \rightarrow Y \Rightarrow \ran(f) = Y \)
   l) \( f \in X \rightarrow Y \land x \in \dom(f) \Rightarrow f[\{x\}] = \{f(x)\} \)
   m) \( (r^{-1})^{-1} = r \)
   n) \( r \in X \leftrightarrow Y \Rightarrow \dom(r^{-1}) = \ran(r) \)
   o) \( r \in X \leftrightarrow Y \Rightarrow \ran(r^{-1}) = \dom(r) \)
   p) \( r \in X \leftrightarrow Y \Rightarrow \ran(r) \in Y \)
   q) \( r \in X \leftrightarrow Y \Rightarrow \ran(r) \subseteq Y \)
   r) \( r \in X \leftrightarrow Y \Rightarrow \ran(r) \in \mathcal{P}(Y) \)

23. \( \text{union}(S) \) is the generalised union of the elements of \( S \), that is, if \( S \) is a set of sets, then \( \text{union}(S) \) is the union of all of the sets that are contained in \( S \). What is \( \text{union}() \)?

24. \( \text{inter}(S) \) is the generalised intersection of the elements of \( S \), that is, if \( S \) is a set of sets, then \( \text{inter}(S) \) is the intersection of all of the sets that are contained in \( S \). What is \( \text{inter}() \)?

25. Two subsets of a set \( S \) are said to be disjoint if and only if they have no elements in common. Define a binary relation \( \text{disjoint} \) that holds between a pair of subsets of \( S \) exactly when they are disjoint.

26. A set of subsets of \( S \) is said to be pairwise disjoint if and only if every pair of distinct sets in it is disjoint (in the sense of (c)). A partition of a set \( S \) is a pairwise disjoint set of subsets of \( S \) whose generalised union is equal to \( S \).
   a) Define the set of all partitions of \( S \).
   b) Which of the subsets of \( \{a, b\} \) are partitions of \( \{a, b\} \)?
10.2 A Simple Bank Machine

The objective of this tutorial exercise is to develop EventB models. In all cases the resulting machines should be introduced into the Rodin Toolkit, analyzed, proof obligations generated, the autoprover run and any remaining undischarged proof obligations inspected carefully. In many cases it would be a good idea to animate the machine.

1. A simple bank Produce a model, consisting of SimpleBank_ctx and SimpleBank machines, of a very simple bank with the following requirements. Follow the English very carefully.

- **accounts** the bank customers are represented by accounts. Having obtained an account a customer may use the other operations supported by the bank.
- **balance** the bank needs to maintain a balance for all accounts.
- **NewAccount** an operation by which a customer obtains an account identifier. Account identifiers are allocated from a pool (set) of identifiers maintained by the bank.
- **Deposit** an operation to add an *amount* to an *account* balance.
- **WithDraw** an operation withdraw an *amount* from an *account*. Customers cannot withdraw more than the balance in their account.
- **Balance** an enquiry operation for a customer to obtain the *balance* in their *account*.
- **Holdings** an operation that returns the total sum of all the balances held by the bank. Clearly this should be a privileged operation not able to be run by anyone, but we will keep things simple
- **Transfer** an operation that transfers an *amount* of money from one bank account to another.

Note: the balance and all other money amounts can be represented by natural numbers.
10.3 Supermarket Model

The objective of this set of tutorial exercises is to develop a model of a simple supermarket.

10.3.1 The Supermarket_ctx context

This context models the “things” that you find in a supermarket.

```
CONTEXT Supermarket_ctx
SETS
  TROLLEY
  PRODUCT
CONSTANTS
  MAXPRICE
  SHELF
  PRICE
  Milk
  Cheese
  Cereal
  BASKET
AXIOMS
  axm1: MAXPRICE ∈ \mathbb{N}
  axm2: PRICE = 0 .. MAXPRICE
  axm3: SHELF = PRODUCT ↦ \mathbb{N}_1
  axm4: partition(PRODUCT, (Milk), (Cheese), (Cereal))
  axm5: BASKET = PRODUCT ↦ \mathbb{N}_1
END
```

Explain the sets and constants you see in the above machine.

10.3.2 The Supermarket machine

For the supermarket we want to model the products in the supermarket, the shelf containing the products, the trolleys available for customers, the customers with trolleys and products in those trolleys.

**Important:** all products on the shelves of the supermarket and in the trolleys must have a price.

Here is part of the Supermarket machine.

```
MACHINE Supermarket
SEES
  Supermarket_ctx
VARIABLES
  shelf
  trolleys
  products
  price
  customers
  reorderlevel
  reorder
  topay
  stock
```
CHAPTER 10. EXERCISES

INVARIENTS
inv1: \(shelf \in SHELF\)
inv2: \(trolleys \subseteq TROLLEY\)
inv3: \(products \subseteq PRODUCT\)
inv4: \(price \in products \rightarrow PRICE\)
inv5: \(products = \text{dom}(price)\)
inv6: \(\text{dom}(shelf) = products\)
inv7: \(customers \in trolleys \rightarrow BASKET\)
inv8: \(\forall t \in \text{dom}(customers) \Rightarrow \text{dom}(customers(t)) \subseteq products\)
inv9: \(\text{reorderlevel} \in products \rightarrow N_1\)
inv10: \(\text{reorder} \subseteq products\)
inv11: \(\text{topay} \in trolleys \rightarrow N\)
inv12: \(\text{stock} \in products \rightarrow N\)

EVENTS

INITIALISATION \(=\)

...  

END

...  

END Supermarket

The above machine is intended to model:

- products on the shelf of the supermarket
- products in customer trolleys
- total stock of products: note that \(stock\) includes all products that are still in the supermarket, either on the shelf, in customers’ trolleys or perhaps in reserve somewhere else in the supermarket.
- checkout
- stock alert when stock level drops below some minimum requirement

Complete the Initialisation and add the following events:

Setprice set the price of a product;
AddStock add some amount of product to the supermarket stock;
AddProductShelf add some amount of product to the shelf of the supermarket;
GetTrolley get a vacant trolley;
AddProductTrolley take some amount of product on shelf and add to trolley;
RemProductTrolley take some amount of product from trolley and return to shelf.
SetMinStock set the minimum amount of product to have in stock;
Checkout checkout product from trolley;
Pay pay for products in trolley;
ReturnTrolley return empty trolley;
ReStock indicate that stock of product has fallen below minimum stock level.

10.3.3 Refinement of Supermarket Machine

Refine the Supermarket machine, especially showing two methods of implementing Checkout: one allowing multiple product items to be processed and the other processing one product items at a time. Events that don’t change can be simply inherited using the mysterious first menu on the event line in Rodin.
Chapter 11

Solutions

11.1 Relations and Functions

1. the sibling relationship between people;
   sibling is clearly a many-to-many relation, that is it is simply a relation and can’t be further
   strengthened: sibling ∈ people ↔ people

2. the brother and sister relationships between people;
   brother and sister are similar to sibling, indeed each is a subset of sibling: brother ⊆ sibling,
   sister ⊆ sibling.

3. the relationship between people and their cars;
   People may have many cars, so again this is simply a relation.

4. the relationship between people and registration plates;
   Registrations are between a registration number and a person (or maybe an identified group of
   people), so this is functional and what’s more it is injective, that is one-to-one.

5. relationships in student enrolment at UNSW;
   The relation between a student identifier and a student is an injective function.

6. the relationship between coin denominations and their value;
   Again, an injective function.

7. the relationship that describes the coins you have in your pocket;
   You probably have many coins in your pocket, and possibly many of the same coin denominations,
   so the relation is a function between the coin denomination and the number you have in your
   pocket. Incidentally, this is known as a bag.

8. relationships concerning products on a supermarket shelf;
   Similar to the preceding question: the relation is functional, between products and the number of
   each product on the shelf.

9. the relationship between courses and lecturers.
   Generally, this will only be a relation.

10. In this question you will be dealing with products, or sets of pairs. Instead of writing a pair
    as (a, b), which is probably what you would normally do, write them as a → b, where → is
    pronounced “maps to”.

109
j) What is $NAMES \times PHONE$?

$NAMES \times PHONE = \{$

\begin{align*}
&Jack \mapsto 123, Jack \mapsto 456, Jack \mapsto 789, \\
&Jill \mapsto 123, Jill \mapsto 456, Jill \mapsto 789 \\
&\}
\end{align*}

k) What might it represent (model)? It might model entries in a phone book.

l) What is $\text{card}(NAMES \times PHONE)$? $6 = \text{card}(NAMES) \times \text{card}(PHONE) = 2 \times 3$

m) What is $\text{card}(NAMES \times \{\})$? $0$; $\text{card}(NAMES \times \{\}) = \{\}$

n) What is $\mathcal{P}(S)$? $\mathcal{P}(S)$, the powerset of $S$ is the set of all subsets of $S$.

o) Given $\text{card}(S) = N$, what is $\text{card}(\mathcal{P}(S))$? $\text{card}(\mathcal{P}(S)) = 2^N$

p) What is $\text{card}(\mathcal{P}(NAMES \times PHONE))$?

$\text{card}(\mathcal{P}(NAMES \times PHONE)) = 2^{\text{card}(NAMES \times PHONE)} = 2^6 = 64$

q) What does $\text{card}(\mathcal{P}(NAMES \times PHONE))$ give you? It gives you all possible mappings between elements of $NAMES$ and elements of $PHONE$, ie it gives you all possible configurations of your phone book.

r) Is $NAMES \times PHONE$ a function? No, it's many to many.

s) Give a functional subset. \{Jack $\mapsto 123$\}

t) Give a total functional subset. \{Jack $\mapsto 123$, Jill $\mapsto 123$\}

u) If a subset $S$ is described as a partial functional set, which of the following is correct?

i) $S$ is not a total functional set.

ii) $S$ might not be a total functional set

iii) is correct. A partial function may happen to be total. If $X \rightarrow Y$ is the set of all partial functions from $X$ to $Y$ and $X \rightarrow Y$ is the set of all total functions from $X$ to $Y$, then $X \rightarrow Y \subseteq X \rightarrow Y$

All of the following classes of functions may be total or not total.

v) Give a injective functional subset. \{Jack $\mapsto 123$, Jill $\mapsto 456$\}

w) Give a surjective functional subset. There is no such function, since any set of mappings from a set of 2 elements to a set of 3 elements could not be functional

x) Give a (total) bijective functional subset. No such function, for the same reason as in (u).

11. Any subset of $X \times Y$ is called a (many to many) relation. $X \leftrightarrow Y$ is the set of all relations formed from $X$ and $Y$. That is $X \leftrightarrow Y = \mathcal{P}(X \times Y)$.

Given phonebook $\in NAMES \leftrightarrow PHONE$,

a) Give some examples of phonebook. \{Jack $\mapsto 123$, \{Jack $\mapsto 123$, Jill $\mapsto 123$, Jack $\mapsto 456$, Jill $\mapsto 789$\\}

b) Give NAMES $\leftrightarrow PHONE$.

$$NAMES \leftrightarrow PHONE = \{$$

\begin{align*}
&\{\}, \\
&\{Jack \mapsto 123\}, \{Jack \mapsto 456\}, \{Jack \mapsto 789\}, \{Jill \mapsto 123\}, \{Jill \mapsto 456\}, \{Jill \mapsto 789\}, \\
&\{Jack \mapsto 123, Jack \mapsto 456\}, \{Jack \mapsto 123, Jack \mapsto 789\}, \{Jack \mapsto 456, Jack \mapsto 789\}, \\
&\{Jill \mapsto 123, Jill \mapsto 456\}, \{Jill \mapsto 123, Jill \mapsto 789\}, \{Jill \mapsto 456, Jill \mapsto 789\}, \\
&\{Jack \mapsto 123, Jill \mapsto 123\}, \{Jack \mapsto 123, Jill \mapsto 456\}, \{Jack \mapsto 123, Jill \mapsto 789\}, \\
&\{Jack \mapsto 456, Jill \mapsto 123\}, \{Jack \mapsto 456, Jill \mapsto 456\}, \{Jack \mapsto 456, Jill \mapsto 789\},
\end{align*}$$
c) What is $\text{card}(\text{NAMES} \leftrightarrow \text{PHONE})$? $\text{card}(\text{NAMES} \leftrightarrow \text{PHONE}) = \text{card}(\mathcal{P}((\text{NAMES} \times \text{PHONE}))) = 64$

12. $X \rightarrow Y$ is the set of all partial functions formed from $X$ and $Y$.

a) Give $\text{NAMES} \rightarrow \text{PHONE}$.

$$\text{NAMES} \rightarrow \text{PHONE} = \{\}$$
{}.
{Jack \mapsto 123}, \{Jack \mapsto 456\}, \{Jack \mapsto 789\}, \{Jill \mapsto 123\}, \{Jill \mapsto 456\}, \{Jill \mapsto 789\},
{Jack \mapsto 123, Jill \mapsto 123}, \{Jack \mapsto 123, Jill \mapsto 456\}, \{Jack \mapsto 123, Jill \mapsto 789\},
{Jack \mapsto 456, Jill \mapsto 123}, \{Jack \mapsto 456, Jill \mapsto 456\}, \{Jack \mapsto 456, Jill \mapsto 789\},
{Jack \mapsto 789, Jill \mapsto 123}, \{Jack \mapsto 789, Jill \mapsto 456\}, \{Jack \mapsto 789, Jill \mapsto 789\}
}

b) What is \(\text{card}(\text{NAMES} \mapsto \text{PHONE})\)? 16

13. \(X \mapsto Y\) is the set of all total functions formed from \(X\) and \(Y\).

a) Give \(\text{NAMES} \mapsto \text{PHONE}\).

\[
\text{NAMES} \mapsto \text{PHONE} = \{
{} ,
{Jack \mapsto 123}, \{Jack \mapsto 456\}, \{Jack \mapsto 789\}, \{Jill \mapsto 123\}, \{Jill \mapsto 456\}, \{Jill \mapsto 789\},
{Jack \mapsto 123, Jill \mapsto 123}, \{Jack \mapsto 123, Jill \mapsto 456\}, \{Jack \mapsto 123, Jill \mapsto 789\},
{Jack \mapsto 456, Jill \mapsto 123}, \{Jack \mapsto 456, Jill \mapsto 456\}, \{Jack \mapsto 456, Jill \mapsto 789\},
{Jack \mapsto 789, Jill \mapsto 123}, \{Jack \mapsto 789, Jill \mapsto 456\}, \{Jack \mapsto 789, Jill \mapsto 789\}
\}
\]

b) What is \(\text{card}(\text{NAMES} \mapsto \text{PHONE})\)? 9 = 3 \times 3

14. \(X \mapsto Y\) is the set of all partial, injective functions formed from \(X\) and \(Y\).

a) Give \(\text{NAMES} \mapsto \mapsto \text{PHONE}\).

\[
\text{NAMES} \mapsto \mapsto \text{PHONE} = \{
\{\} ,
{Jack \mapsto 123}, \{Jack \mapsto 456\}, \{Jack \mapsto 789\}, \{Jill \mapsto 123\}, \{Jill \mapsto 456\}, \{Jill \mapsto 789\},
{Jack \mapsto 123, Jill \mapsto 123}, \{Jack \mapsto 123, Jill \mapsto 456\}, \{Jack \mapsto 123, Jill \mapsto 789\},
{Jack \mapsto 456, Jill \mapsto 123}, \{Jack \mapsto 456, Jill \mapsto 456\}, \{Jack \mapsto 456, Jill \mapsto 789\},
{Jack \mapsto 789, Jill \mapsto 123}, \{Jack \mapsto 789, Jill \mapsto 456\}
\}
\]

b) What is \(\text{card}(\text{NAMES} \mapsto \mapsto \text{PHONE})\)? 13 = 6 + 6 + 1

15. \(X \mapsto Y\) is the set of all total, injective functions formed from \(X\) and \(Y\).

a) Give \(\text{NAMES} \mapsto \text{PHONE}\).

\[
\text{NAMES} \mapsto \text{PHONE} = \{
\{\} ,
{Jack \mapsto 123}, \{Jack \mapsto 456\}, \{Jack \mapsto 789\}, \{Jill \mapsto 123\}, \{Jill \mapsto 456\}, \{Jill \mapsto 789\},
{Jack \mapsto 123, Jill \mapsto 456}\}, \{Jack \mapsto 456, Jill \mapsto 789\},
{Jack \mapsto 789, Jill \mapsto 123}, \{Jack \mapsto 789, Jill \mapsto 456\}
\}
\]

b) What is \(\text{card}(\text{NAMES} \mapsto \text{PHONE})\)? 6

16. \(X \mapsto Y\) is the set of all partial, surjective functions formed from \(X\) and \(Y\).
11.1. RELATIONS AND FUNCTIONS

a) Give $\text{NAMES} \mapsto \text{PHONE}$.

$\text{NAMES} \mapsto \text{PHONE} = \{\}$

b) What is $\text{card}(\text{NAMES} \mapsto \text{PHONE})$? 0

17. $X \rightarrow Y$ is the set of all total, surjective functions formed from $X$ and $Y$.

a) Give $\text{NAMES} \mapsto \text{PHONE}$.

$\text{NAMES} \mapsto \text{PHONE} = \{\}$

b) What is $\text{card}(\text{NAMES} \mapsto \text{PHONE})$? 0

18. $X \rightarrow Y$ is the set of all (total) bijective functions formed from $X$ and $Y$.

a) Give $\text{NAMES} \rightarrow \text{PHONE}$.

$\text{NAMES} \rightarrow \text{PHONE} = \{\}$

b) What is $\text{card}(\text{NAMES} \rightarrow \text{PHONE})$? 0

c) Why is a partial bijection unnecessary? A partial bijection $X \mapsto Y$ can be represented by a total injections $Y \rightarrow X$.

19. This set exercises some important relational operators.

Given $\text{phone} = \{\text{Jack} \mapsto 123, \text{Jack} \mapsto 789, \text{Jill} \mapsto 456, \text{Jill} \mapsto 789\}$

a) What is $\text{dom}(\text{phone})$? $\{\text{Jack, Jill}\}$.

b) What is $\text{ran}(\text{phone})$? $\{123, 456, 789\}$.

c) What is $\text{phone} \leftarrow \{\text{Jack} \mapsto 123\}$? $\{\text{Jack} \mapsto 123, \text{Jill} \mapsto 456, \text{Jill} \mapsto 789\}$.

d) What is $\text{Jack} \in \text{phone}$? $\{\text{Jack} \mapsto 123, \text{Jack} \mapsto 789\}$.

e) What is $\text{Jill} \in \text{phone}$? $\{\text{Jill} \mapsto 456, \text{Jill} \mapsto 789\}$.

f) What is $\text{phone} \succ \{123, 789\}$? $\{\text{Jack} \mapsto 123, \text{Jack} \mapsto 789, \text{Jill} \mapsto 789\}$.

g) What is $\text{phone} \prec \{123, 789\}$? $\{\text{Jill} \mapsto 456\}$.

h) What is $\text{phone}[[\text{Jack}]]$? $\{123, 789\}$.

20. Students pass a subject if they gain at least 50 marks in the final examination. Given a function $\text{results} : \text{STUDENTS} \rightarrow \mathbb{N}$, that yields the examination result for a particular student, specify

a) the set of students that pass: $\text{dom}(\text{results} \succ \{n \mid n \in \mathbb{N} \land n \geq 50\})$

$\{s \mid s \in \text{dom}(\text{results}) \land \text{results}(s) \geq 50\}$

or

$\text{dom}(\text{results} \succ \{n \mid n \in \mathbb{N} \land n \geq 50\})$

b) the set of students that fail.

$\{s \mid s \in \text{dom}(\text{results}) \land \text{results}(s) < 50\}$

21. If we were modelling a taxi fleet company we might have three variables, $\text{drivers}$, $\text{taxis}$ and $\text{assigned}$ constrained by

$\text{drivers} : \mathbb{P} \text{DRIVERS}$

$\text{taxis} : \mathbb{P} \text{TAXIS}$

$\text{assigned} : \text{drivers} \rightarrow \rightarrow \text{taxis}$

where $\text{drivers}$ is the set of drivers working for the company, $\text{taxis}$ is the set of taxis owned by the company, and $\text{assigned}$ is a function recording the assignment of drivers to taxis.

The arrow $\rightarrow \rightarrow$ denotes a partial injective function. An injective function is a one-to-one function. Notice that the inverse of an injective function is also an injective function. In general, of course, the inverse of a function is not necessarily even a function.
a) Why is \textit{assigned} a function? It is a \textit{function} because a driver would be assigned to at most one taxi.

b) Why is \textit{assigned} a partial function?
   It is \textit{partial} because at any time not all drivers would necessarily be assigned to a taxi.

c) Why is \textit{assigned} an injective function?
   It is \textit{injective} because a taxi would be assigned to at most one driver.

d) Specify the drivers who are currently assigned.
   \text{dom}(\text{assigned})

e) Specify the drivers who are currently unassigned.
   \text{drivers} - \text{dom}(\text{assigned})

f) Specify the taxis that are currently assigned.
   \text{ran}(\text{assigned})

g) Specify the taxis that are currently unassigned.
   \text{taxis} - \text{ran}(\text{assigned})

22. Are the following rules correct or incorrect?

\begin{itemize}
  \item [a)] \( f \in X \rightarrow Y \Rightarrow f \in X \rightarrow Y \)
    Correct, a total function is a partial function.
  \item [b)] \( f \in X \rightarrow Y \Rightarrow f \in X \rightarrow Y \)
    Correct, a partial injection is a partial function.
  \item [c)] \( f \in X \rightarrow Y \Rightarrow f \in X \rightarrow Y \)
    Incorrect, a partial injection is not a total function.
  \item [d)] \( f \in X \rightarrow Y \Rightarrow f \in X \rightarrow Y \)
    Correct, a total injection is a total function.
  \item [e)] \( f \in X \rightarrow Y \Rightarrow f \in X \rightarrow Y \)
    Correct, a total injection is a partial function.
  \item [f)] \( f \in X \rightarrow Y \Rightarrow f \in X \rightarrow Y \)
    Correct, a partial surjection is a partial function.
  \item [g)] \( f \in X \rightarrow Y \Rightarrow f \in X \rightarrow Y \)
    Incorrect, a partial surjection is not a total function.
  \item [h)] \( f \in X \rightarrow Y \Rightarrow f \in X \rightarrow Y \)
    Correct, a total surjection is a total function.
  \item [i)] \( f \in X \rightarrow Y \Rightarrow f \in X \rightarrow Y \)
    Correct, a total surjection is a partial function.
  \item [j)] \( f \in X \rightarrow Y \Rightarrow \text{dom}(f) \subset X \)
    Incorrect.
  \item [k)] \( f \in X \rightarrow Y \Rightarrow \text{ran}(f) = Y \)
    Incorrect.
  \item [l)] \( f \in X \rightarrow Y \land x \in \text{dom}(f) \Rightarrow f([x]) = \{f(x)\} \)
    Correct.
  \item [m)] \((r^{-1})^{-1} = r\)
    Correct.
  \item [n)] \( r \in X \leftrightarrow Y \Rightarrow \text{dom}(r^{-1}) = \text{ran}(r) \)
    Correct.
  \item [o)] \( r \in X \leftrightarrow Y \Rightarrow \text{ran}(r^{-1}) = \text{dom}(r) \)
    Correct.
\end{itemize}
11.1. RELATIONS AND FUNCTIONS

p) \( r \in X \leftrightarrow Y \Rightarrow \text{ran}(r) \in Y \)
Incorrect.

q) \( r \in X \leftrightarrow Y \Rightarrow \text{ran}(r) \subseteq Y \)
Correct.

r) \( r \in X \leftrightarrow Y \Rightarrow \text{ran}(r) \in \mathcal{P}(Y) \)
Correct.

23. The generalized union of a set of subsets of \( X \) contains those elements of \( X \) that are in at least one of the subsets. Define a function \( \text{union} : \mathcal{P} \mathcal{P} X \rightarrow \mathcal{P} X \) that maps a set of subsets of \( X \) to its generalized union. What is \( \text{union} \emptyset \)?

a) \( \text{union}(U) = \{ x : X \mid \exists u \cdot (u \in U \land x \in u) \} \)

b) \( \text{union}(\emptyset) = \{ x : X \mid \forall u \cdot (u \in \emptyset \Rightarrow x \in u) \} \)
\( = \{ x : X \mid \text{false} \} \)
\( = \emptyset \)

To shed a bit more light on this, it is clear that \( \text{union}([x]) = \{ x \} \) for any \( x \in X \). But, \( \text{union}([x]) = \text{union}([x], \emptyset) = \text{union}([x]) \cup \text{union}(\emptyset) \). It follows that \( \text{union}(\emptyset) \) must be the empty set.

24. The generalized intersection of a set of subsets of \( X \) contains just those elements of \( X \) that are in all the subsets. Define a function \( \text{inter} \mathcal{P} \mathcal{P} X \rightarrow \mathcal{P} X \) that maps a set of subsets of \( X \) to its generalized intersection. What is \( \text{inter} \emptyset \)?

a) \( \text{inter}(U) = \{ x : X \mid \exists u \cdot (u \in U \Rightarrow x \in u) \} \)

b) \( \text{inter}(\emptyset) = \{ x : X \mid \forall u \cdot (u \in \emptyset \Rightarrow x \in u) \} \)
\( = \{ x : X \mid \top \} \)
\( = X \)

To shed a bit more light on \( \text{inter}(\emptyset) \), it is clear that \( \text{inter}([x]) = \{ x \} \). Now consider \( \text{inter}(P, [x]) \), where \( P \) is a list of sets. Then, \( \text{inter}(P, [x]) = \text{inter}(\{P\}) \cap [x] \). Now take the case where the list \( P \) is empty, \( [x] = \text{inter}(\emptyset) \cap \{ x \} \) for all \( x \in X \). Therefore, \( \text{inter}(\emptyset) \) must be \( X \).

25. Two subsets of a set \( X \) are said to be disjoint if and only if they have no elements in common. Define a binary relation \( \text{disjoint} \) that holds between a pair of subsets of \( X \) exactly when they are disjoint.

\( \text{disjoint}(u, w) = u \cap w = \emptyset \)

26. A set of subsets of \( X \) is said to be pairwise disjoint if and only if every pair of distinct sets in it is disjoint (in the sense of (c)). A partition of a set \( X \) is a pairwise disjoint set of subsets of \( X \) whose generalized union is equal to \( X \).

a) Define the set of all partitions of \( X \).
\( \text{partition}X = \{ w : w \in \mathcal{P}(\mathcal{P}(X)) \land \text{union}(w) = X \land \forall(u1, u2) \cdot (u1 \in w \land u2 \in w \Rightarrow u1 \neq u2) \land \text{disjoint}(u1, u2) \} \)

b) Which of the subsets of \( \{a, b\} \) are partitions of \( \{a, b\} \)?
\( \{\{a\}, \{b\}\}, \{\emptyset, \{a\}, \{b\}\}, \{\{a\}, \\emptyset, \{b\}\}, \{\{a, b\}\}, \{\emptyset, \{a, b\}\} \).
Appendix A

Models

A.1 Coffee Club

MACHINE CoffeeClub

VARIABLES
piggybank denotes a supply of money for the coffee club.

INVARIANTS
inv1: piggybank ∈ N piggybank must be a natural number, that is, a non-zero integer

EVENTS

Initialisation ≜
THEN
act1: piggybank := 0
END

FeedBank ≜
ANY
amount
WHERE
grd1: amount ∈ N
THEN
act1: piggybank := piggybank + amount
END

RobBank ≜
ANY
amount
WHERE
grd1: amount ∈ 1..piggybank The amount must not exceed the contents of piggybank
APPENDIX A. MODELS

THEN
act1: \( \text{piggybank} := \text{piggybank} - \text{amount} \)
END

END CoffeeClub

CONTEXT MembersContext
SETS MEMBER
AXIOMS
axm1: finite(MEMBER)
END

MACHINE MemberShip
REFINES CoffeeClub
SEES MemberShip

VARIABLES
piggybank
members
accounts
coffeeprice

INVARIANTS
inv1: \( \text{piggybank} \in \mathbb{N} \)
inv2: \( \text{members} \subseteq \text{MEMBER} \) \quad \text{each member has unique id}
inv3: \( \text{accounts} \in \text{members} \rightarrow \mathbb{N} \) \quad \text{each member has an account}
inv4: \( \text{coffeeprice} \in \mathbb{N}_1 \) \quad \text{price of a cup of coffee}

EVENTS
Initialisation : \textit{extended} \( \cong \)
THEN
act2: \( \text{members} := \emptyset \) \quad \text{empty set of members}
act3: \( \text{accounts} := \emptyset \) \quad \text{empty set of accounts}
act4: \( \text{coffeeprice} \in \mathbb{N}_1 \) \quad \text{initial coffee price set to arbitrary non-zero value}
END

SetPrice \( \cong \)
ANY amount
WHERE
grd1: \( \text{amount} \in \mathbb{N}_1 \)
THEN
act1: \( \text{coffeeprice} := \text{amount} \)
END
NewMember \triangleq \\
\textsc{any} \textsc{member} \\
\textsc{where} \\
\text{grd1:} \text{ member } \in \textsc{MEMBER} \setminus \textsc{members} \quad \text{choose an unused element} \\
\text{of \textsc{MEMBER}} \\
\textsc{then} \\
\text{act1:} \text{ members } := \text{ members } \cup \{\text{ member }\} \\
\text{act2:} \text{ accounts}(\text{ member }) := 0 \\
\textsc{end}

Contribute \triangleq \\
\textsc{refines} \textsc{FeedBank} \\
\textsc{any} \textsc{amount} \textsc{member} \\
\textsc{where} \\
\text{grd1:} \text{ amount } \in \mathbb{N}_1 \\
\text{grd2:} \text{ member } \in \text{ members} \\
\textsc{then} \\
\text{act1:} \text{ accounts}(\text{ member }) := \text{ accounts}(\text{ member }) + \text{ amount} \\
\text{act2:} \text{ piggybank } := \text{ piggybank } + \text{ amount} \\
\textsc{end}

BuyCoffee \triangleq \\
\textsc{any} \textsc{member} \\
\textsc{where} \\
\text{grd1:} \text{ member } \in \text{ members} \\
\text{grd2:} \text{ accounts}(\text{ member }) \geq \text{ coffeeprice} \\
\textsc{then} \\
\text{act1:} \text{ accounts}(\text{ member }) := \text{ accounts}(\text{ member }) - \text{ coffeeprice} \\
\textsc{end}

FeedBank : \textit{extended} \triangleq \\
\textsc{refines} \textsc{FeedBank} \\
\textsc{any} \\
\textsc{where} \\
\textsc{then} \\
\textsc{end}
RobBank: \textit{extended} \triangleq REFINES RobBank ANY WHERE THEN END

END MemberShip
**CONTEXT** Theories

Some simple number theory theorems

| AXIOMS | 6thm1: \((\forall n \cdot n \in \mathbb{N} \Rightarrow (\exists m \cdot m \in \mathbb{N} \land (n = 2 \cdot m \lor n = 2 \cdot m + 1)))\) | Every natural number is either even or odd
| 6thm2: \((\forall n \cdot n \in \mathbb{N} \Rightarrow n < (n + 1) \cdot (n + 1))\) | Lower bound for square of \(n+1\)
| 6thm3: \(\forall m, n \cdot m \in \mathbb{N} \land n \in \mathbb{N} \land m < n \Rightarrow (m + n)/2 < n\) | Upper bound (without equality) for \((m+n)/2\)
| 6thm4: \(\forall m, n \cdot m \in \mathbb{N} \land n \in \mathbb{N} \land m < n \Rightarrow (m + n)/2 \geq m\) | Lower bound (with equality) for \((m+n)/2\) |
Appendix B

Proof Obligations

All proof obligations have a name and an abbreviation:

<table>
<thead>
<tr>
<th>Id</th>
<th>Name</th>
<th>Needed to discharge</th>
</tr>
</thead>
<tbody>
<tr>
<td>INV</td>
<td>Invariant</td>
<td></td>
</tr>
<tr>
<td>FIS</td>
<td>Feasibility</td>
<td></td>
</tr>
<tr>
<td>WD</td>
<td>Well-definedness</td>
<td></td>
</tr>
<tr>
<td>GRD</td>
<td>Guard</td>
<td></td>
</tr>
<tr>
<td>EQL</td>
<td>Equal</td>
<td></td>
</tr>
</tbody>
</table>
Appendix C

Event-B Concise Summary

Each construct will be given in its presentation form, as displayed in the Rodin toolkit, followed by the ASCII form that is used for input to Rodin.

In the following: $P$, $Q$ and $R$ denote predicates; $x$ and $y$ denote single variables; $z$ denotes a single or comma-separated list of variables; $p$ denotes a pattern of variables, possibly including $\rightarrow$ and parentheses; $S$ and $T$ denote set expressions; $U$ denotes a set of sets; $m$ and $n$ denote integer expressions; $f$ and $g$ denote functions; $r$ denotes a relation; $E$ and $F$ denote expressions; $E,F$ is a recursive pattern, i.e. it matches $e_1, e_2$ and also $e_1, e_2, e_3 \ldots$; similarly for $x,y$;

Freeness: The meta-predicate $\neg\text{free}(z,E)$ means that none of the variables in $z$ occur free in $E$. This meta-predicate is defined recursively on the structure of $E$, but that will not be done here explicitly. The base cases are: $\neg\text{free}(z,\forall z \cdot P \Rightarrow Q)$, $\neg\text{free}(z,\exists z \cdot P \land Q)$, $\neg\text{free}(z,\{z \cdot P | F\})$, $\neg\text{free}(z,\lambda z \cdot P|E)$, and $\text{free}(z,z)$.

In the following the statement that $P$ must constrain $z$ means that the type of $z$ must be at least inferrable from $P$.

In the following, parentheses are used to show syntactic structure; they may of course be omitted when there is no confusion.

Note: Event-B has a formal syntax and this summary does not attempt to describe that syntax. What it attempts to do is to explain Event-B constructs. Some words like expression collide with the formal syntax. Where a syntactical entity is intended the word will appear in italics, e.g. expression, predicate.

The base cases are: $\neg\text{free}(z,(\forall z \cdot P)$, $\neg\text{free}(z,(\exists z \cdot P)$, $\neg\text{free}(z,\{z \cdot P | F\})$, $\neg\text{free}(z,\lambda z \cdot P|E)$, and $\text{free}(z,z)$

In the following:

- the statement “$P$ must constrain $z$” means that the type of $z$ must be at least inferrable from $P$.
- parentheses are used to show syntactic structure; they may of course be omitted when there is no confusion.
C.1 Predicates

1. False : \(\bot\)  \hspace{1cm} \text{false}
2. True : \(\top\)  \hspace{1cm} \text{true}
3. Conjunction : \(P \land Q\)  \hspace{1cm} P \land Q
   \hspace{1cm} \text{Left associative.}
4. Disjunction : \(P \lor Q\)  \hspace{1cm} P \lor Q
   \hspace{1cm} \text{Left associative.}
5. Implication : \(P \Rightarrow Q\)  \hspace{1cm} P \Rightarrow Q
   \hspace{1cm} \text{Non-associative: this means that } P \Rightarrow Q \Rightarrow R \text{ must be parenthesised or an error will be diagnosed.}
6. Equivalence : \(P \Leftrightarrow Q\)  \hspace{1cm} P \Leftrightarrow Q
   \hspace{1cm} \text{Non-associative: this means that } P \Leftrightarrow Q \Leftrightarrow R \text{ must be parenthesised or an error will be diagnosed.}
7. Negation : \(\neg P\)  \hspace{1cm} \text{not } P
8. Universal quantification : \((\forall z . P \Rightarrow Q)\)  \hspace{1cm} \{ tz . P \Rightarrow Q \}
   \hspace{1cm} \text{Strictly, } \forall z . P \text{, but usually an implication.}
   \hspace{1cm} \text{For all values of } z \text{, satisfying } P \text{, } Q \text{ is satisfied.}
   \hspace{1cm} \text{The types of } z \text{ must be inferrable from the predicate } P.
9. Existential quantification : \((\exists z . P \land Q)\)  \hspace{1cm} \{ z . P \land Q \}
   \hspace{1cm} \text{Strictly, } \exists z . P \text{, but usually a conjunction.}
   \hspace{1cm} \text{There exist values of } z \text{, satisfying } P \text{, that satisfy } Q.
   \hspace{1cm} \text{The predicate } P \text{ must be inferrable from the predicate } Q.
10. Equality : \(E = F\)  \hspace{1cm} E = F
11. Inequality : \(E \neq F\)  \hspace{1cm} E \neq F

C.2 Sets

1. Singleton : \(\{E\}\)  \hspace{1cm} \{E\}
2. Enumeration : \(\{E, F\}\)  \hspace{1cm} \{E, F\}
   \hspace{1cm} \text{See note on the pattern } E, F \text{ at top of summary.}
3. Empty set : \(\emptyset\)  \hspace{1cm} \{\}
4. Comprehension : \(\{ z . P \mid F \}\)  \hspace{1cm} \{ z . P \mid F \}
   \hspace{1cm} \text{General form: the set of all values of } F \text{ for all values of } z \text{ that satisfy the predicate } P.
   \hspace{1cm} P \text{ must constrain the variables in } z.
5. Comprehension : \(\{ F \mid P \}\)  \hspace{1cm} \{ F \mid P \}
   \hspace{1cm} \text{Special form: the set of all values of } F \text{ that satisfy the predicate } P.
   \hspace{1cm} \{ F \mid P \} = \{ z . P \mid F \}, \text{where } z \text{ is all the variables in } F.
6. Comprehension : \(\{ x \mid P \}\)  \hspace{1cm} \{ x \mid P \}
   \hspace{1cm} \text{A special case of item 5: the set of all values of } x \text{ that satisfy the predicate } P.
   \hspace{1cm} \{ x \mid P \} = \{ x . P \mid x \}
7. Union : \(S \cup T\)  \hspace{1cm} S \cup T
8. Intersection : \(S \cap T\)  \hspace{1cm} S \cap T
9. Difference : \(S \setminus T\)  \hspace{1cm} S \setminus T
10. Ordered pair : \(E \mapsto F\)  \hspace{1cm} E \mapsto F
    \hspace{1cm} \text{Left associative.}
    \hspace{1cm} \text{In all places where an ordered pair is required, } E \mapsto F \text{ must be used. } E, F \text{ will not be accepted as an ordered pair, it is always a list. } \{x, y . P \mid x \mapsto y\} \text{ illustrates the different usage.}
11. Cartesian product : \(S \times T\)  \hspace{1cm} S \times T
    \hspace{1cm} \{ x \mapsto y \mid x \in S \text{ and } y \in T \}
    \hspace{1cm} \text{Left-associative.}
12. Powerset : \(\mathcal{P}(S)\)  \hspace{1cm} \mathcal{P}(S)
    \hspace{1cm} \{ s \mid s \subseteq S \}
13. Non-empty subsets : \(\mathcal{P}_1(S)\)  \hspace{1cm} \mathcal{P}_1(S)
    \hspace{1cm} \mathcal{P}_1(S) = \mathcal{P}(S) \setminus \{\emptyset\}
14. Cardinality : \(\text{card}(S)\)  \hspace{1cm} \text{card}(S)
    \hspace{1cm} \text{Defined only for finite}(S).
15. Generalized union : \(\text{union}(U)\)  \hspace{1cm} \text{union}(U)
    \hspace{1cm} \text{The union of all the elements of } U.
    \hspace{1cm} \forall U . U \in \mathcal{P}(S) \Rightarrow \text{union}(U) = \{ x \mid x \in S \text{ and } \exists s . s \in U \text{ and } x \in s \}
    \hspace{1cm} \text{where } \neg \text{free}(x, s, U)
16. Generalized intersection \( \text{inter}(U) \)

The intersection of all the elements of \( U \).

\[ \forall U, U \in \text{P}(\text{P}(S)) \Rightarrow \]

\[ \text{inter}(U) = \{ x \mid x \in S \land \forall s \in U \Rightarrow x \in s \} \]

where \( \neg \text{free}(x, s, U) \)

17. Quantified union :

\[ \text{UNION } z. P \mid S \]

\( P \) must constrain the variables in \( z \).

\[ \forall z. P \Rightarrow S \subseteq T \Rightarrow \]

\[ \bigcup(z \cdot P \mid E) = \{ x \mid x \in T \land \exists z \cdot P \land x \in S \} \]

where \( \neg \text{free}(x, z, T) \), \( \neg \text{free}(x, P) \), \( \neg \text{free}(x, S) \)

18. Quantified intersection :

\[ \text{INTER } z. P \mid S \]

\( P \) must constrain the variables in \( z \).

\[ \{ z \cdot P \neq \emptyset , \}

\[ (\forall z \cdot P \Rightarrow S \subseteq T) \Rightarrow \]

\[ \bigcap(z \cdot P \mid S) = \{ x \mid x \in T \land (\forall z \cdot P \Rightarrow x \in S) \} \]

where \( \neg \text{free}(x, z) \), \( \neg \text{free}(x, T) \), \( \neg \text{free}(x, P) \), \( \neg \text{free}(x, S) \)

C.2.1 Set predicates

1. Set membership : \( E \in S \)

\( E : S \)

2. Set non-membership : \( E \notin S \)

\( E /: S \)

3. Subset : \( S \subseteq T \)

\( S <: T \)

4. Not a subset : \( S \not\subseteq T \)

\( S /<: T \)

5. Proper subset : \( S \subset T \)

\( S <<- T \)

6. Not a proper subset : \( s \not\subset t \)

\( S /<<: T \)

7. Finite set : \( \text{finite}(S) \)

\( \text{finite}(S) \Rightarrow S \) is finite.

8. Partition : \( \text{partition}(S, x, y) \)

\[ x \text{ and } y \text{ partition the set } S, \text{ i.e } S = x \cup y \land x \cap y = \emptyset \]

Specialised use for enumerated sets:

\[ \text{partition}(S, \{A\}, \{B\}, \{C\}) . \]

\( S = \{A, B, C\} \land A \neq B \land B \neq C \land C \neq A \)

C.3 BOOL and bool

BOOL is the enumerated set: \{FALSE, TRUE\},

and bool is defined on a predicate \( P \) as follows:

1. \( P \) is provable: \( \text{bool}(P) = \text{TRUE} \)

2. \( \neg P \) is provable: \( \text{bool}(P) = \text{FALSE} \)

C.4 Numbers

The following is based on the set of integers, the set of natural numbers (non-negative integers), and the set of positive (non-zero) natural numbers.

1. The set of integer numbers \( \mathbb{Z} \)

\( \text{INT} \)

2. The set of natural numbers \( \mathbb{N} \)

\( \text{NAT} \)

3. The set of positive natural numbers \( \mathbb{N}_1 \)

\( \text{NAT1} \)

4. Minimum \( \min(S) \)

\( S \subseteq \mathbb{Z} \) and \( \text{finite}(S) \) or \( S \) must have a lower bound.

5. Maximum \( \max(S) \)

\( S \subseteq \mathbb{Z} \) and \( \text{finite}(S) \) or \( S \) must have an upper bound.

6. Sum \( m + n \)

\( m + n \)

7. Difference \( m - n \)

\( m - n \)

8. Product \( m \times n \)

\( m \times n \)

9. Quotient \( m/n \)

\( m / n \)

10. Remainder \( m \mod n \)

\( m \mod n \)

11. Interval \( m..n \)

\( m..n = \{ i \mid m \leq i \land i \leq n \} \)

C.4.1 Number predicates

1. Greater \( m > n \)

\( m > n \)

2. Less \( m < n \)

\( m < n \)

3. Greater or equal \( m \geq n \)

\( m \geq n \)

4. Less or equal \( m \leq n \)

\( m \leq n \)
C.5 Relations

A relation is a set of ordered pairs; a many to many mapping.

1. Relations $S \leftrightarrow T$
   $S \leftrightarrow T \equiv \mathcal{P}(S \times T)$
   Associativity: relations are right associative: $r \in X \iff Y \iff Z = r \in X \iff (Y \iff Z)$.

2. Domain $\text{dom}(r)$
   $\forall \cdot \cdot \cdot r \in S \leftrightarrow T \Rightarrow$
   $\text{dom}(r) = \{x \cdot (\exists y \cdot x \mapsto y \in r)\}$

3. Range $\text{ran}(r)$
   $\forall \cdot \cdot \cdot r \in S \leftrightarrow T \Rightarrow$
   $\text{ran}(r) = \{y \cdot (\exists x \cdot x \mapsto y \in r)\}$

4. Total relation $S \leftrightarrow T$
   if $r \in S \leftrightarrow T$ then $\text{dom}(r) = S$

5. Surjective relation $S \leftrightarrow T$
   if $r \in S \leftrightarrow T$ then $\text{ran}(r) = T$

6. Total surjective relation $S \leftrightarrow T$
   if $r \in S \leftrightarrow T$ then $\text{dom}(r) = S$ and $\text{ran}(r) = T$

7. Forward composition $p \cdot q$
   $\forall \cdot \cdot \cdot p, q \in S \leftrightarrow T \land q \in T \leftrightarrow U \Rightarrow$
   $p \cdot q = \{x \mapsto y \mid (\exists z \cdot x \mapsto z \in p \land z \mapsto y \in q)\}$

8. Backward composition $p \circ q$
   $p \circ q = q \cdot p$

9. Identity $\text{id}$
   $S \cdot \text{id} = \{x \mapsto x \mid x \in S\}$.
   $\text{id}$ is generic and the set $S$ is inferred from the context.

10. Domain restriction $S \cdot r$
    $S \cdot r = \{x \mapsto y \mid x \mapsto y \in r \land x \in S\}$.

11. Domain subtraction $S \ll r$
    $S \ll r = \{x \mapsto y \mid x \mapsto y \in r \land x \notin S\}$.

12. Range restriction $r \cdot T$
    $r \cdot T = \{x \mapsto y \mid x \mapsto y \in r \land y \in T\}$.

13. Range subtraction $r \gg T$
    $r \gg T = \{x \mapsto y \mid x \in r \land y \notin T\}$.

14. Inverse $r^{-1}$
    $r^{-1} = \{y \mapsto x \mid x \mapsto y \in r\}$.

15. Relational image $r[S]$
    $r[S] = \{y \mid \exists x \cdot x \in S \land x \mapsto y \in r\}$.

16. Overriding $r_1 \ll r_2$
    $r_1 \ll r_2 = r_2 \cup (\text{dom}(r_2) \ll r_1)$.

17. Direct product $p \otimes q$
    $p \otimes q = \{x \mapsto (y \mapsto z) \mid x \mapsto y \in p \land x \mapsto z \in q\}$.

18. Parallel product $p \parallel q$
    $p \parallel q = \{x, y, m, n \cdot x \mapsto m \in p \land y \mapsto n \in q \mid (x \mapsto y) \mapsto (m \mapsto n)\}$.

19. Projection $\text{prj}_1$
    $\text{prj}_1$ is generic.
    $(S \times T) \ll \text{prj}_1 = \{(x \mapsto y) \mapsto x \mid x \mapsto y \in S \times T\}$.

20. Projection $\text{prj}_2$
    $\text{prj}_2$ is generic.
    $(S \times T) \ll \text{prj}_2 = \{(x \mapsto y) \mapsto y \mid x \mapsto y \in S \times T\}$.

C.5.1 Iteration and Closure

Iteration and closure are important functions on relations that are not currently part of the kernel Event-B language. They can be defined in a Context, but not polymorphically.

Note: iteration and irreflexive closure will be implemented in a proposed extension of the mathematical language. The operators will be non-associative.

1. Iteration $r^n$
   $r \in S \leftrightarrow S \Rightarrow r^0 = S \ll \text{id} \land r^{n+1} = r \cdot r^n$.
   Note: to avoid inconsistency $S$ should be the finite base set for $r$, i.e. the smallest set for which all $r \in S \leftrightarrow S$.
   Could be defined as a function $\text{iterate}(r \mapsto n)$.

2. Reflexive Closure $r^*$
   $r^* = \cup n \cdot n \in \mathbb{N} \mid r^n$.
   Could be defined as a function $\text{rclosure}(r)$.
   Note: $r^0 \subseteq r^*$.

3. Irreflexive Closure $r^+$
   $r^+ = \cup n \cdot n \in \mathbb{N}_1 \mid r^n$.
   Could be defined as a function $\text{idclosure}(r)$.
   Note: $r^0 \nsubseteq r^+$ by default, but may be present depending on $r$. 
C.5.2 Functions

A function is a relation with the restriction that each element of the domain is related to a unique element in the range; a many to one mapping.

1. Partial functions $S \rightarrow T$
   $$S \rightarrow T = \{ r : r \in S \leftrightarrow T \land r^{-1} \subseteq T' \}.$$

2. Total functions $S \rightarrow T$
   $$S \rightarrow T = \{ f : f \in S \rightarrow T \land \text{dom}(f) = S \}.$$

3. Partial injections $S \rightarrow\rightarrow T$
   $$S \rightarrow\rightarrow T = \{ f : f \in S \rightarrow T \land f^{-1} \in T \rightarrow S \}.$$  
   One-to-one relations.

4. Total injections $S \rightarrow\rightarrow\rightarrow T$
   $$S \rightarrow\rightarrow\rightarrow T = \{ f : f \in S \rightarrow T \land \text{dom}(f) = S \}.$$  

5. Partial surjections $S \rightarrow\rightarrow\rightarrow T$
   $$S \rightarrow\rightarrow\rightarrow T = \{ f : f \in S \rightarrow T \land \text{ran}(f) = T \}.$$  
   Onto relations.

6. Total surjections $S \rightarrow\rightarrow\rightarrow T$
   $$S \rightarrow\rightarrow\rightarrow T = \{ f : f \in S \rightarrow T \land \text{ran}(f) = T \}.$$  

7. Bijections $S \rightarrow\rightarrow\rightarrow T$
   $$S \rightarrow\rightarrow\rightarrow T = \{ f : f \in S \rightarrow T \land \text{ran}(f) = T \}.$$  
   One-to-one and onto relations.

8. Lambda abstraction
   $$\lambda p \cdot P | E$$
   $P$ must constrain the variables in $p$.
   $$\lambda p \cdot P | E = \{ z : P \mid p \mapsto E \}$$, where $z$ is a list of variables that appear in the pattern $p$.

9. Function application $f(E)$
   $$f(E)$$
   where $\text{type}(f) = \mathbb{P}(X \times Y)$.
   **Note:** in Event-B, relations and functions only ever have one argument, but that argument may be a pair or tuple, hence $f(E \mapsto F)$
   $f(E \mapsto F)$ is never valid.

C.6 Models

1. Contexts contain sets and constants used by other contexts or machines.

   CONTEXT Identifier
   EXTENDS Machine_Identifiers
   SETS Identifiers
   CONSTANTS Identifiers
   AXIOMS Predicates
   THEOREMS Predicates
   END

2. Machines contain events.

   MACHINE Identifier
   REFINES Machine_Identifiers
   SEES Context_Identifiers
   VARIABLES Identifiers
   INVARIANT Predicates
   THEOREMS Predicates
   VARIANT Expression
   EVENTS Events
   END

C.6.1 Events

**Event_name**
REFINES Event_identifiers
ANY Identifiers
WHERE Predicates
WITH Witnesses
THEN Actions
END

There is one distinguished event named INITIALISATION used to initialise the variables of a machine, thus establishing the invariant.

C.6.2 Actions

Actions are used to change the state of a machine. There may be multiple actions, but they take effect concurrently, that is, in parallel. The semantics of events are defined in terms of substitutions. The substitution $[G]P$ defines a predicate obtained by replacing the values of the variables in $P$ according to the action $G$. General substitutions are not available in the Event-B language.

**Note on concurrency:** any single variable can be modified in at most one action, otherwise the effect of the actions would, in general, be inconsistent.

1. *skip*, the null action

   *skip* denotes the empty set of actions for an event.
2. Simple assignment action $x := E$  
$:= = \text{“becomes equal to”}: \text{replace free occurrences of } x \text{ by } E$.

3. Choice from set $x \in S$  
$\in = \text{“becomes in”}: \text{arbitrarily choose a value from the set } S$.

4. Choice by predicate $z : | P$  
$:\mid = \text{“becomes such that”}: \text{arbitrarily choose values for the variable in } z \text{ that satisfy the predicate } P$. Within $P$, $x$ refers to the value of the variable $x$ before the action and $x'$ refers to the value of the variable after the action.

5. Functional override $f(x) := E$  
$f(x) := E$  
Substitute the value $E$ for the expression $f$ at point $x$.  
This is a shorthand:  
$f(x) := E = f := f \leftarrow \{x \mapsto E\}$.

6. Multiple action  
$x, y := E, F$  
$x, y := E, F$  
Concurrent assignment of the values $E$ and $F$ to the variables $x$ and $y$, respectively. This is equivalent multiple single actions.

Acknowledgement: Jean-Raymond Abrial, Laurent Voisin and Ian Hayes have all given valuable feedback and corrections at various stages of the evolution of this summary.
Appendix D

Rodin

D.1 The Rodin platform
Bibliography


Index

Actions
- skip, the null action, 129
- Choice by predicate, 130
- Choice from set, 130
- Functional override, 130
- Multiple action, 130
- Simple assignment action, 130

concepts
- $\exists x \cdot P$, 23
- $\forall x \cdot P$, 23
- SETS, 12
- axiom, 12
- extended, 13
- extends, 23
- feasibility, 18
- refinement relation, 10
- theorem, 12
- well-defined, 18

event
- actions, 5
- guards, 5
- parameters, 5

Functions
- Bijections, 129
- Function application, 129
- Lambda abstraction, 129
- Partial functions, 129
- Partial injections, 129
- Partial surjections, 129
- Total functions, 129
- Total injections, 129
- Total surjections, 129

machine
- context, 5
- events, 5
- invariant, 5
- state, 5

Models
- Contexts, 129

Machines, 129

notation
- $\Rightarrow$, 14
- $\Rightarrow^*$, 69
- $\Rightarrow^*$, 69
- $\perp$, 8
- $\cap$, 40
- $\cup$, 14
- $\exists x \cdot P$, 22
- $\forall x \cdot P$, 22
- $\geq$, 7
- $\in$, 6
- $\leq$, 7
- $\mapsto$, 16
- $\sim$, 12
- $\Leftrightarrow$, 23
- $\Rightarrow$, 22
- $\nless$, 12
- $\Leftrightarrow$, 11
- $\Rightarrow$, 11
- $\Rightarrow^*$, 69
- $\Leftrightarrow$, 23
- $\Leftrightarrow$, 23
- bool, 25
- $\emptyset$, 13
- $\mathbb{N}$, 6
- $\mathbb{N}_1$, 7
- BOOL, 25
- $:=$, 7
- $:\in$, 13
- $\mid$, 24
- $\neg$, 11
- $\neq$, 16
- $\notin$, 16
- $\subseteq$, 13
- $\times$, 80
- $T$, 8
- $id$, 42
INDEX

Numbers
- Difference, 127
- Interval, 127
- Maximum, 127
- Minimum, 127
- Product, 127
- Quotient, 127
- Remainder, 127
- Sum, 127
- The set of integer numbers, 127
- The set of natural numbers, 127
- The set of positive natural numbers, 127

Numeric relations
- Greater, 127
- Greater or equal, 127
- Less, 127
- Less or equal, 127

PO
- see proof obligation, 7

Predicates
- Conjunction, 126
- Disjunction, 126
- Equality, 126
- Equivalence, 126
- Existential quantification, 126
- False, 126
- Implication, 126
- Inequality, 126
- Negation, 126
- True, 126
- Universal quantification, 126

proof-obligation
- consistency check, 7
- sequent representation, 7

Relations
- Backward composition, 128
- Direct product, 128
- Domain, 128
- Domain restriction, 128
- Domain subtraction, 128
- Forward composition, 128
- Identity, 128
- Inverse, 128
- Irreflexive Closure, 128
- Iteration, 128
- Overriding, 128
- Parallel product, 128
- Projection, 128
- Range, 128
- Range restriction, 128
- Range subtraction, 128
- Reflexive Closure, 128
- Relational image, 128
- Relations, 128
- Surjective relation, 128
- Total relation, 128
- Total surjective relation, 128

Set
- Cardinality, 126
- Cartesian product, 126
- Comprehension, 126
- Difference, 126
- Empty set, 126
- Enumeration, 126
- Generalized intersection, 127
- Generalized union, 126
- Intersection, 126
- Non-empty subsets, 126
- Ordered pair, 126
- Powerset, 126
- Quantified intersection, 127
- Quantified union, 127
- Singleton, 126
- Union, 126

Set predicates
- Finite set, 127
- Not a proper subset, 127
- Not a subset, 127
- Partition, 127
- Proper subset, 127
- Set membership, 127
- Set non-membership, 127
- Subset, 127

system
- design, 3
- modelling, 3

Todo, 59, 67

variant
- what is it, 27