System Modelling & Design Using Event-B

Ken Robinson

Last updated: October 10, 2010

Contents

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:

MACHINE **CoffeeClub**

VARIABLES

piggybank This machine state is represented by the variable, *piggybank*, denoting a supply of money for the coffee club.

INVARIANTS

inv1: $piggybank \in \mathbb{N}$ piggybank must be a natural number

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 explore the proposed behaviour of a design.

It is important to understand that in order to explore a design we will be concerned with *what* happens, *when* it happens and *what* changes of state are associated with an event.

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 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.

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 and 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 Axiomatice Basis for Computer Programming*

1976 Edsgar 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: Assignment 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 intended to be 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.

3.1 Machines

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

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 desired facilities of the coffee club.

MACHINE **CoffeeClub**

VARIABLES

piggybank	This machine state is represented by the variable, *piggybank*, denoting a supply of money for the coffee club.

INVARIANTS

inv1:	$piggybank \in \mathbb{N}$	piggybank must be a natural number

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

math	ascii	
$\in$	`:`	set membership
$\mathbb{N}$	`NAT`	the set of natural numbers = non-negative integers

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 by the event.

Initialisation $\widehat{=}$

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 $\widehat{=}$

ANY

amount	amount to be added to *piggybank*

WHERE

grd1:	$amount \in \mathbb{N}1$	if *amount* were 0 then this event could always fire

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
$\mathbb{N}1$	`NAT1`	the set of non-zero natural numbers

RobBank $\widehat{=}$
ANY
amount
WHERE
grd1: $amount \in \mathbb{N}1$
grd2: $amount \leq piggybank$ There must be enough in the piggybank
THEN
act1: $piggybank := piggybank - amount$
END

Notation

math	ascii	
$\leq$	`<=`	*less than or equal*

END **CoffeeClub**

Proof Obligations: Sequent representation

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*[22] having the following form:

$$\boxed{hypotheses \vdash goal}$$

Figure 3.1: Sequent representation of PO

The meaning of the PO shown in 3.1 is

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.

Notation

math	ascii	
$\top$	`true`	Boolean true
$\bot$	`false`	Boolean false

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.

INITIALISATION/inv1/INV: $\boxed{\vdash 0 \in \mathbb{N}}$

This is requesting a proof that the initialisation, $piggybank := 0$, establishes the invariant $piggybank \in \mathbb{N}$.

FeedBank/inv1/INV: $\boxed{\begin{array}{r} piggybank \in \mathbb{N} \\ amount \in \mathbb{N}1 \end{array} \vdash piggybank + amount \in \mathbb{N}}$

This is requesting a proof that the actions of FeedBank, $piggybank := piggybank + amount$ maintains the invariant, $piggybank \in \mathbb{N}$. This is clearly true as both $piggybank$ and $amount$ are natural numbers.

RobBank/inv1/INV: $\boxed{\begin{array}{r} piggybank \in \mathbb{N} \\ amount \in \mathbb{N}1 \\ amount \leq piggybank \end{array} \vdash piggybank - amount \in \mathbb{N}}$

Similar to the preceding POs, but this time verifying that $piggybank \in \mathbb{N}$ is maintained after the action $piggybank := piggybank - amount$. This is not quite so simple, but the guard $amount \leq piggybank$ ensures the invariant is maintained.

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. 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.

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.

3.2 Refinement

Refinement is a process that is used 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.

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.

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 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.

Consistently 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 machine being refined, the refined machine may introduce new events, but the new events *must not* change the state of any included from 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.

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. 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

axiom	an axiom is a property that is asserted; it cannot be proved
theorem	a theorem is a property that is implied by *axioms* or *invariants*; it must be proved

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.

Context MembersContext

CONTEXT **MembersContext**
SETS
    MEMBER
AXIOMS
    axm1: $\mathit{finite}(MEMBER)$
END

Concepts

SETS	Sets declared in SETS clause of a context are non-empty, opaque sets.

Notation

math	ascii	
finite	`finite`	$finite(S)$ 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

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. 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.

It should be clear that the events *FeedBank* and *RobBank* are unchanged in the refinement, but *NewMember*, *SetPrice*, *BuyCoffee* and *Contribute* are new. For that reason *FeedBank* and *RobBank* will be omitted here. They can be found in the appendix A.1.

MACHINE **MemberShip**

REFINES

CoffeeClub

SEES

MembersContext

VARIABLES

piggybank	
members	the set of current members
accounts	the member accounts
coffeeprice	the price of a cup of coffee

INVARIANTS

inv1:	$piggybank \in \mathbb{N}$	
inv2:	$members \subseteq MEMBER$	each member has unique id
inv3:	$accounts \in members \rightarrow \mathbb{N}$	each member has an account
inv4:	$coffeeprice \in \mathbb{N}1$	any price other than free!

Notation

math	ascii	
$\subseteq$	`<:`	subset $\subset$ 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.
$\nrightarrow$	`+->`	denotes a *partial* function. If $f \in X \nrightarrow Y$ and $x \in X$, then $f(x)$ is not necessarily defined.

EVENTS

Initialisation : *extended* $\widehat{=}$

THEN

act2:	$members := \varnothing$	empty set of members
act3:	$accounts := \varnothing$	empty set of accounts
act4:	$coffeeprice :\in \mathbb{N}1$	initial coffee price set to arbitrary non-zero value

END

Notation

math	ascii	
$:\in$	`::`	"becomes in": $x :\in e$ means assign to x any element of the set s
$\varnothing$	`{}`	empty set

SetPrice $\widehat{=}$
ANY
  amount
WHERE
  grd1: $amount \in \mathbb{N}1$
THEN
  act1: $coffeeprice := amount$
END

NewMember $\widehat{=}$
ANY
  member
WHERE
  grd1: $member \in MEMBER \setminus members$ choose an unused element of MEMBER
THEN
  act1: $members := members \cup \{member\}$
  act2: $accounts(member) := 0$
END

Notation

math	ascii	
$\setminus$	`\`	Set subtraction: $S \setminus T$ is the set of elements in S that are not in T
$\cup$	`\/`	Set union: $S \cup T$ is the set of elements that are in either S or T

Contribute $\widehat{=}$
ANY
  amount
  member
WHERE
  grd1: $amount \in \mathbb{N}1$
  grd2: $member \in members$
THEN
  act1: $accounts(member) := accounts(member) + amount$
  act2: $piggybank := piggybank + amount$
END

BuyCoffee $\widehat{=}$
ANY
  member
WHERE
  grd1: $member \in members$
  grd2: $accounts(member) \geq coffeeprice$

THEN
  act1: $accounts(member) := accounts(member) - coffeeprice$
END

FeedBank : ***extended*** $\widehat{=}$
REFINES
  FeedBank
ANY

WHERE

THEN

END

RobBank : ***extended*** $\widehat{=}$
REFINES
  RobBank
ANY

WHERE

THEN

END

END **MemberShip**

Proof Obligations

INITIALISATION:inv1/INV: $\boxed{\vdash\ 0 \in \mathbb{N}}$

INITIALISATION:inv3/INV: $\boxed{\vdash\ \varnothing \in \varnothing \to \mathbb{N}}$

INITIALISATION:inv4/INV: $\boxed{coffeeprice' \in \mathbb{N}1 \vdash coffeeprice' \in \mathbb{N}1}$

INITIALISATION:act4/FIS: $\boxed{\vdash\ \mathbb{N}1 \neq \varnothing}$

SetPrice/inv4/INV: $\boxed{\begin{array}{r} coffeeprice \in \mathbb{N}1 \\ amount \in \mathbb{N}1 \end{array} \vdash amount \in \mathbb{N}1}$

NewMember/inv3/INV: $\boxed{\begin{array}{r} accounts \subseteq members \to \mathbb{N} \\ member \in MEMBER \setminus members \end{array} \vdash \begin{array}{l} accounts \lhd\!\!\!+ \{member \mapsto 0\} \\ \in \\ members \cup \{member\} \to \mathbb{N} \end{array}}$

Contribute/inv1/INV:

$$\begin{array}{r} piggybank \in \mathbb{N} \\ amount \in \mathbb{N}1 \\ member \in members \end{array} \vdash piggybank + amount \in \mathbb{N}$$

Contribute/inv3/INV:

$$\begin{array}{r} account \in members \rightarrow \mathbb{N} \\ amount \in \mathbb{N}1 \\ member \in members \end{array} \vdash \begin{array}{l} accounts \mathbin{\lhd\!\!-} \{member \mapsto accounts(member) + amount\} \\ \in \\ members \rightarrow \mathbb{N} \end{array}$$

Contribute/piggybank/EQL:

$$\begin{array}{r} amount \in \mathbb{N}1 \\ member \in members \end{array} \vdash piggybank = piggybank + amount$$

Contribute/act2/WD:

$$\begin{array}{r} amount \in \mathbb{N}1 \\ member \in members \end{array} \vdash \begin{array}{l} member \in \mathrm{dom}(accounts) \\ \wedge\, accounts \in MEMBER \nrightarrow \mathbb{Z} \end{array}$$

BuyCoffee/grd2/WD:

$$member \in members \vdash \begin{array}{l} member \in \mathrm{dom}(accounts) \\ \wedge\, accounts \in MEMBER \nrightarrow \mathbb{Z} \end{array}$$

BuyCoffee/inv3/INV:

$$\begin{array}{r} accounts \in members \rightarrow \mathbb{N} \\ member \in members \\ accounts(member) \geq coffeeprice \end{array} \vdash \begin{array}{l} accounts \\ \mathbin{\lhd\!\!-} \{member \mapsto accounts(member) - coffeeprice\} \\ \in \\ members \rightarrow \mathbb{N} \end{array}$$

BuyCoffee/act1/WD:

$$\begin{array}{r} member \in members \\ accounts(member) \geq coffeeprice \end{array} \vdash \begin{array}{l} member \in \mathrm{dom}(accounts) \\ \wedge\, accounts \in MEMBER \nrightarrow \mathbb{Z} \end{array}$$

Notation

math	ascii	
$\neq$	`/=`	$a \neq b$ = a *is not equal to* b
$\mapsto$	`|->`	$a \mapsto b$ (a *maps to* b) is the ordered pair of a and b

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 $\widehat{=}$
REFINES
FeedBank
ANY
member
WHERE
grd1: $member \in members$
THEN
act1: $accounts(member) := accounts(member) + amount$
act2: $piggybank := piggybank + amount$
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.

What are the new POs?

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

INITIALISATION:act4/FIS: $\mathbb{N}1 \neq \varnothing$

Contribute/act2/WD:
$member \in \mathrm{dom}(accounts) \wedge accounts \in MEMBER \mapsto \mathbb{N}1$

BuyCoffee/grd2/WD:
$member \in \mathrm{dom}(accounts) \wedge accounts \in MEMBER \mapsto \mathbb{N}1$

BuyCoffee/act1/WD:
$member \in \mathrm{dom}(accounts) \wedge accounts \in MEMBER \mapsto \mathbb{N}1$

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

$$coffeeprice' \in \mathbb{N}1$$

This will be only feasible if $\mathbb{N}1 \neq \varnothing$, 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 $x \in \mathrm{dom}(f)$, in this case $member \in \mathrm{dom}(accounts)$. This is guaranteed by the guard $member \in members$ and the invariant $accounts \in member \to \mathbb{N}$.

Concepts

well-defined	some expressions, especially function applications, may not be defined everywhere. For example, $f(x)$ is only defined if x is in the domain of f, ie $x \in \mathrm{dom}(f)$.
feasibility	specifying a property with a predicate does not carry with it the promise that there exist solutions that satisfy the predicate. For example $x+1 = x-1$ cannot be satisfied by any $x \in \mathbb{N}$. Feasibility is concerned with showing that instances that satisfy a predicate do exist. Feasibility can be extremely difficult to prove and many famous conjectures, for example Fermat's last theorem and the four colour problem, have been solved only recently. Fermat's last theorem was unsolved for over 300 years.

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 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 view of the behaviour of the events in the model.

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.

Animation Constraints

Animators generally, and AnimB in particular impose a stronger finiteness constraint than the finiteness constraints imposed by Event-B. If MemberShip is animated with AnimB, it will be found that the type of *piggybank*, namely $\mathbb{N}$, is unsatisfactory. AnimB requires a subrange of $\mathbb{N}$, hence we will have to specify a maximum amount for *piggybank*, *MAXBANK*, and then specify *piggybank* as a subrange $0 \,..\, MAXBANK$. As a consequence various guards also have to be modified. The following shows a modified version of *CoffeeClub* and a context *PiggyBank* in which *MAXBANK* is given the (arbitrary) value 1000.

CONTEXT **PiggyBank**
CONSTANTS
    MAXBANK
AXIOMS
    axm1: $MAXBANK \in \mathbb{N}1$
ANIMB
    MAXBANK 1000

END

MACHINE **CoffeeClub**
SEES
  PiggyBank
VARIABLES
  piggybank
INVARIANTS
  inv1: $piggybank \in 0 \,..\, MAXBANK$
EVENTS
Initialisation $\widehat{=}$
THEN
  act1: $piggybank := 0$
END

FeedBank $\widehat{=}$
ANY
  amount
WHERE
  grd1: $amount \in 1 \,..\, MAXBANK - piggybank$
THEN
  act1: $piggybank := piggybank + amount$
END

RobBank $\widehat{=}$
ANY
  amount
WHERE
  grd1: $amount \in 1 \,..\, piggybank$
THEN
  act1: $piggybank := piggybank - amount$
END

END **CoffeeClub**

The context *MembersContext*, as before declares a finite set *MEMBER*, but AnimB requires a finite set with explicit members. 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.

CONTEXT **MembersContext**
SETS
  MEMBER
AXIOMS
  axm1: $finite(MEMBER)$

ANIMB VALUES
  MEMBER $\{m1, m2, m3\}$ Define a set of 3 members
END

The machine *MemberShip* is as before.

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 the model with the most precise and concise specification of what it is we want to model.

4.1 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.

Definition and Model

We start with a definition of the square root we want to compute.

Let num be the number whose integer square root we want to compute and $sqrt$ be the 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:

$$
\begin{aligned}
num &\in \mathbb{N} && (4.1)\\
sqrt &\in \mathbb{N} \rightarrow \mathbb{N} && (4.2)\\
sqrt(num) \times sqrt(num) &\leq num && (4.3)\\
num &< (sqrt(num) + 1) \times (sqrt(num) + 1) && (4.4)\\
& && (4.5)
\end{aligned}
$$

We will use a context to define the constant num, whose square root we wish to compute. The value of num is any natural number.

CONTEXT **SquareRoot_ctx**
EXTENDS
    Theories
CONSTANTS
    num
AXIOMS

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

END

This context extends a context that contains some theorems that will be useful in the discharge of proof obligations:

CONTEXT **Theories**

AXIOMS

thm1: $\forall n \cdot n \in \mathbb{N} \Rightarrow (\exists m \cdot m \in \mathbb{N} \wedge (n = 2 * m \vee n = 2 * m + 1))$

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

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

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

END

We model *sqrt* as follows:

MACHINE **SquareRoot**

SEES

SquareRoot_ctx

VARIABLES

sqrt

INVARIANTS

inv1: $sqrt \in \mathbb{N}$

EVENTS

Initialisation $\widehat{=}$

THEN

act1: $sqrt :\in \mathbb{N}$

END

SquareRoot $\widehat{=}$

WHERE

grd1: $sqrt * sqrt > num$ check sqrt

grd2: $num \leq (sqrt + 1)(sqrt + 1)$ is not already computed

THEN

act1: $sqrt :| (sqrt' \in \mathbb{N} \wedge sqrt' * sqrt' \leq num \wedge num < (sqrt' + 1) * (sqrt' + 1))$

END

END **SquareRoot**

Notation

math	ascii	
$:\mid$	:|	"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 $x := x + 1$.

Note: For a first simple exercise the guards to *SquareRoot* could be omitted. They prevent the event running forever

When we look at the Proof obligations we see:

INITIALISATION/inv1/INV: $\boxed{sqrt' \in \mathbb{N} \vdash sqrt' \in \mathbb{N}}$

SquareRoot/inv1/INV:
$$\boxed{\begin{array}{r} sqrt \in \mathbb{N} \\ sqrt' \in \mathbb{N} \\ sqrt' * sqrt' \leq num \\ num < (sqrt' + 1) * (sqrt' + 1) \end{array} \vdash sqrt' \in \mathbb{N}}$$

SquareRoot/act1/FIS:
$$\boxed{num \in \mathbb{N} \vdash \begin{array}{l} \exists sqrt' \cdot sqrt' \in \mathbb{N} \\ sqrt' * sqrt' \leq num \\ num < (sqrt' + 1) * (sqrt' + 1) \end{array}}$$

These are all easy to discharge except the feasibility PO for SquareRoot act1.

It should be clear that the model correctly specifies *sqrt*, by embedding the definition of *sqrt*.

What this is asking is:

> *it is all very well to write a predicate such as this about $sqrt'$, but show that such an animal exists!*

The important point is that it is easy to write predicates that do not have a solution. The typical way of discharging such a PO is to give a *witness*, that is a value for the existential variable(s) that *satisfies* the quantified predicate.

It is clear that we cannot do this, so the PO will be left undischarged. Rodin allows the PO to be *reviewed*, indicating that the PO has been looked at and it is believed to be true.

We will implicitly satisfy the PO by proceeding with a refinement that will demonstrate how a value can be computed for *sqrt*.

It is important to understand that, if it is not possible to discharge a feasibility PO —such as the above for *sqrt*— then it will not be possible to complete a concrete refinement, in which we produce a method of computing *sqrt* and discharge all POs generated through the refinement.

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:

$$\begin{array}{l} sqrt * sqrt \leq num \\ num < (sqrt + 1) * (sqrt + 1) \end{array}$$

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

$$\begin{array}{l} low * low \leq num \\ num < high * high \\ low < high \end{array}$$

and move low and $high$ closer together until

$$low + 1 = high$$

at which point low will be the desired value of $sqrt$.

We can now model this idea of *splitting the invariant.*

MACHINE **SquareRootR1**
REFINES
  SquareRoot
SEES
  SquareRoot_ctx
VARIABLES
  sqrt
  low
  high
INVARIANTS
  inv1: $low \in \mathbb{N}$
  inv2: $high \in \mathbb{N}$
  inv3: $low + 1 \leq high$
  inv4: $low * low \leq num$
  inv5: $num < high * high$
  thm1: $low + 1 \neq high \Rightarrow low < (low + high)/2$
  thm2: $(low + high)/2 < high$
VARIANT
  $high - low$
EVENTS
Initialisation $\widehat{=}$
THEN
  act1: $sqrt :\in \mathbb{N}$
  act2: $low :| low' \in \mathbb{N} \wedge low' * low' \leq num$
  act3: $high :| high' \in \mathbb{N} \wedge num < high' * high'$
END

SquareRoot $\widehat{=}$
REFINES
  SquareRoot
WHERE
  grd1: $low + 1 = high$
  thm1: $low * low \leq num$
  thm2: $num < high * high$
THEN
  act1: $sqrt := low$
END

Improve $\widehat{=}$
STATUS convergent
ANY
  l
  h
WHERE

grd1: $low + 1 \neq high$
grd2: $l \in \mathbb{N} \wedge low \leq l \wedge l * l \leq num$
grd3: $h \in \mathbb{N} \wedge h \leq high \wedge num < h * h$
grd4: $l + 1 \leq h$
grd5: $h - 1 < high - low$
THEN
act1: $low, high := l, h$
END

END **SquareRootR1**

Parameters

Parameters represent arbitrary, nondeterministic values over which we have no control. Despite that they have significant influence on the conditions under which an event may fire. In the above we have introduced parameters l and h to represent improvements in *low* and *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*.

Convergence and Variants

The event *Improve* has been given a status of *convergent*. The reason is that the original single event *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 *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 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 yield a natural number 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.

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: $low \,..\, high$.

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 *SquareRoot* cannot fire because the slave events can fire indefinitely.

Improving Improve

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

MACHINE **SquareRootR2**
REFINES
    SquareRootR1
SEES
    SquareRoot_ctx
VARIABLES
    sqrt
    low
    high
INVARIANTS

EVENTS
Initialisation : ***extended*** $\widehat{=}$
THEN

END

SquareRoot : ***extended*** $\widehat{=}$
REFINES
    SquareRoot
WHERE

THEN

END

Improve1 $\widehat{=}$
REFINES
    Improve
ANY
    m
WHERE
    grd1: $low + 1 \neq high$
    grd2: $m \in \mathbb{N}$
    grd3: $low < m \wedge m < high$
    grd4: $m * m \leq num$ m is a better value for low
WITH
    l: $l = m$
    h: $h = high$

THEN
    act1: $low := m$
END

Improve2 $\widehat{=}$
REFINES
    Improve
ANY
    m
WHERE
    grd1: $low + 1 \neq high$
    grd2: $m \in \mathbb{N}$
    grd3: $low < m \land m < high$
    grd4: $m * m > num$ m is a better value for $high$
WITH
    l: $l = low$
    h: $h = m$
THEN
    act1: $high := m$
END

END **SquareRootR2**

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 *Improve*. Parameters l and h have *disappeared.* To enable the verification that *Improve1* and *Improve2* do refine *Improve* we have to give what is known as a *witness* for l and h. This will show how the new parameters *simulate* the old.

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 mid way between low and $high$

We will adopt for 2 this refinement.

MACHINE **SquareRootR3**
REFINES
    SquareRootR2
SEES
    SquareRoot_ctx
VARIABLES
    sqrt
    low
    high
INVARIANTS

EVENTS
Initialisation : ***extended*** $\widehat{=}$
THEN

END

SquareRoot : ***extended*** $\widehat{=}$
REFINES
SquareRoot
WHERE

THEN

END

Improve1 $\widehat{=}$
REFINES
Improve
ANY
m
WHERE
grd1: $low + 1 \neq high$
grd2: $m = (low + high)/2$
grd3: $m * m \leq num$ m is a better value for low
THEN
act1: $low := m$
END

Improve2 $\widehat{=}$
REFINES
Improve
ANY
m
WHERE
grd1: $low + 1 \neq high$
grd2: $m = (low + high)/2$
grd3: $m * m > num$ m is a better value for $high$
THEN
act1: $high := m$
END

END **SquareRootR3**

Refining SquareRootR3

SquareRootR3 is still not completely *concrete* as it depends on the *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

$$mid * mid > num$$

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.

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

SquareRoot : *extended* $\widehat{=}$
REFINES
    SquareRoot
WHERE

THEN

END

Improve1 $\widehat{=}$
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 $\widehat{=}$
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**

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. The sequence we have used does highlight the fact that there are many different strategies for choosing the value of m.

Exercise

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

4.2 Modelling a parametric argument

In the model above we modeled the argument to the *SquareRoot* event 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 $\widehat{=}$
THEN
act1: $sqrt :\in \mathbb{N}$
END

SquareRoot $\widehat{=}$
ANY
num
WHERE

grd1: $num \in \mathbb{N}$

THEN

act1: $sqrt :| (sqrt' \in \mathbb{N}$
$\land sqrt' * sqrt' \leq num$
$\land num < (sqrt' + 1) * (sqrt' + 1))$

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

Theories

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: $low * low \leq numv$
inv6: $numv < high * high$
inv7: $active \in BOOL$
inv8: $active = FALSE \Rightarrow$
$sqrt * sqrt \leq numv \land$
$(sqrt + 1) * (sqrt + 1) > numv$

VARIANT

$high - low$

EVENTS

Initialisation $\widehat{=}$

THEN

act1: $numv := 0$
act2: $low := 0$
act3: $high := 1$
act4: $sqrt := 0$
act5: $active := FALSE$

END

SquareRoot $\widehat{=}$

REFINES

SquareRoot

WHERE

grd1: $low + 1 = high$

grd2: $active = TRUE$

WITH

num: $num = numv$

THEN

act1: $sqrt := low$

act2: $active := FALSE$

END

Improve $\widehat{=}$

STATUS convergent

ANY

l

h

WHERE

grd1: $low + 1 \neq high$

grd2: $l \in \mathbb{N} \wedge low \leq l \wedge l * l \leq num$

grd3: $h \in \mathbb{N} \wedge h \leq high \wedge num < h * h$

grd4: $l + 1 \leq h$

grd5: $h - 1 < high - low$

grd6: $active = TRUE$

THEN

act1: $low, high := l, h$

END

activate $\widehat{=}$

ANY

num

WHERE

grd1: $num \in \mathbb{N}$

grd2: $active = FALSE$

THEN

act1: $numv := num$

act2: $low :| low' \in \mathbb{N} \wedge low' * low' \leq num$

act3: $high :| high' \in \mathbb{N} \wedge num < high' * high'$

act4: $active := TRUE$

END

END **SquareRootR1**

Notation

math	ascii	
BOOL	`BOOL`	$\mathrm{BOOL} = \{\mathrm{TRUE}, \mathrm{FALSE}\}$
bool		$\mathrm{bool} = \{\top, \bot\}$

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

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 ($active$ = TRUE) and the quiescent state ($active$ =

FALSE) when a square root has been found.

Remainder of development

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

MACHINE **SquareRootR4**
REFINES
    SquareRootR3
SEES
    Theories
VARIABLES
    sqrt
    numb
    low
    high
    active
    mid
INVARIANTS
    inv1: $mid = (low + high)/2$
EVENTS
Initialisation $\widehat{=}$
THEN
    act1: $numv := 0$
    act2: $low := 0$
    act3: $high := num + 1$
    act4: $sqrt := 0$
    act5: $active := FALSE$
    act6: $mid := 0$
END

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

Activate $\widehat{=}$
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 $\widehat{=}$
REFINES
Improve1
WHERE
grd1: $low + 1 \neq high$
grd2: $mid * mid \leq numv$
WITH
m: $m = mid$
THEN
act1: $low := mid$
act2: $mid := (mid + high)/2$
END

Improve2 $\widehat{=}$
REFINES
Improve2
WHERE
grd1: $low + 1 \neq high$
grd2: $numv < mid * mid$
grd3: $active = TRUE$
WITH
m: $m = mid$
THEN
act1: $high := mid$
act2: $mid := (low + mid)/2$
END

END **SquareRootR4**

Converting to programming code

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

$low := 0;$

$high := num + 1;$

while $low + 1 \neq high$ {

$mid := (low + high)/2$

if $mid * mid \leq num${

$low := mid$

}

  else $high := mid$

}

$sqrt := low$

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 *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 should be as strong as possible, 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.

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

Notation

math	ascii	
partition	`partition`	partition(S,(s1),...,(sn)) means $S = s_1 \cup \ldots \cup s_n$, and the sets $s_1, \ldots, s_n$ are pairwise disjoint: $s_i \cap s_j = \varnothing$
$\cap$	/\	Set intersection: $S \cap T$ is the elements of elements that are in both S and T

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

$LIGHTS = \{Red, Green, Amber\}$
$Red \neq Green$
$Green \neq Amber$
$Amber \neq Green$

Sets $LIGHTS$ and $DIRECTION$ are finite enumerated sets.

- $LIGHTS$ has 3 distinct colours, and
- $DIRECTION$ has 2 distinct directions.

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

MACHINE **SimpleChangeLights**
SEES
    SimpleTwoWay1
VARIABLES
    lights

END **SimpleChangeLights**

At the moment *lights* is simply declared as a total function from $DIRECTION$ to $LIGHTS$, 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 *false*;
- whenever the invariant is *true* the intersection must be safe.

That is:

$$\neg(safe) \Rightarrow \neg(invariant) \tag{5.1}$$

and

$$invariant \Rightarrow safe \tag{5.2}$$

Note that instead of $x = \top$, we will simply write x, and wherever we might write $x = \bot$, we will simply write $\neg x$, for example *safe* and *invariant* in the above.

Notation

math	ascii	
$\neg$	`not`	negation: $\neg P$ negates the predicate P

5.2 will be recognised as the contrapositive of 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

$$lights(NorthSouth) \in \{Green, Amber\} \Rightarrow lights(EastWest) = Red$$

or

$$lights(EastWest) \in \{Green, Amber\} \Rightarrow lights(NorthSouth) = Red$$

	Red		Amber
Red	safe	safe	safe
Green	safe	unsafe	unsafe
Amber	safe	unsafe	unsafe

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

$$lights(NorthSouth) = Red \vee lights(EastWest) = Red$$

$$Red \in \mathrm{ran}(lights)$$

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.

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: $OTHERDIR \in DIRECTION \to DIRECTION$
  axm4: $OTHERDIR(NorthSouth) = EastWest$
  axm5: $OTHERDIR(EastWest) = NorthSouth$
  thm1: $\forall dir \cdot dir \in DIRECTION$
  $\Rightarrow$
  $OTHERDIR(OTHERDIR(dir)) = dir$
  thm2: $OTHERDIR \,;OTHERDIR \subseteq id$
END

Notation

math	ascii	
id	`id`	id is the *identity* relation: for all $x \in X$, $x \mapsto x \in id$. id is generic; to restrict it to a particular set X domain restriction can be used: $X \lhd id$
$\lhd$	`<|`	Domain restriction: $s \lhd r$ is the subset of r in which the domain is restricted to the set s

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 identify relation on the set $DIRECTION$. Again, this is tested by proposing a theorem.

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
  SimpleTwoWay0
VARIABLES

lights

INVARIANTS

inv1: $lights \in DIRECTION \rightarrow LIGHTS$

inv2: $lights(NorthSouth) \in \{Green, Amber\} \Rightarrow lights(EastWest) = Red$

thm1: $lights(Northsouth) \in \{Green, Amber\} \Rightarrow Lights(EastWest) = Red$

EVENTS

Initialisation $\widehat{=}$

THEN

act1: $lights = \{NorthSouth \mapsto Red, EastWest \mapsto Red\}$

END

ToAmber $\widehat{=}$

ANY

dir

WHERE

grd1: $lights(dir) = Green$ Sequencing

thm1: $lights(OTHERDIR(dir)) = Red$ Safety

THEN

act1: $lights(dir) := Amber$

END

ToGreen $\widehat{=}$

ANY

dir

WHERE

grd1: $lights(dir) = Red$ Sequencing

grd2: $lights(OTHERDIR(dir)) = Red$ Safety

THEN

act1: $lights(dir) := Green$

END

ToRed $\widehat{=}$

ANY

dir

WHERE

grd1: $lights(dir) = Amber$ Sequencing; Safety is preserved

THEN

act1: $lights(dir) := Red$

END

END **SimpleChangeLights**

5.2 A Multiway Intersection

The multiway intersection consists of:

DIRECTION: a finte 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 **TrafficLights2_ctx**

SETS

LIGHTS

DIRECTION

CONSTANTS

Red

Green

Amber

CONFLICT

AXIOMS

axm1:	$partition(LIGHTS, \{Red\}, \{Green\}, \{Amber\})$
axm2:	$finite(DIRECTION)$
axm3:	$CONFLICT \in DIRECTION \leftrightarrow DIRECTION$
axm4:	$CONFLICT \cap id = \varnothing$
axm5:	$CONFLICT^{-1} = CONFLICT$
thm1:	$\forall d \cdot d \in DIRECTION \Rightarrow d \notin CONFLICT[\{d\}]$
thm2:	$\forall d1, d2 \cdot d1 \notin CONFLICT[d2] \Rightarrow d2 \notin CONFLICT[d1]$

END

Notation

math	ascii	
$\notin$	`/:`	*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:

$$\forall d1, d2.d1 \mapsto d2 \in CONFLICT \quad (5.3)$$
$$LIGHTS[d1] \in \{Green, Amber\} \quad \Rightarrow \quad LIGHTS(d2) = Red; \quad (5.4)$$

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

axm5: (symmetry) conflicts are symmetric: d1 *conflicts with* d2 $\Rightarrow$ d2 *conflicts with* d1;

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: d2 *does not conflict with* d1 $\Rightarrow$ d1 *does not conflict with* d2.

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 intersectin 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 **ChangeLights2**
SEES
TrafficLights2_ctx

VARIABLES
lights
INVARIANTS
inv1: $lights \in DIRECTION \to \{Red, Green\}$
inv2: $\forall d \cdot d \in DIRECTION \land lights(d) = Green$
$\Rightarrow lights[CONFLICT[\{d\}]] \subseteq \{Red\}$
inv3: $finite(lights)$
EVENTS
Initialisation $\widehat{=}$
THEN
act1: $lights :| lights' \in DIRECTION \to \{Red, Green\}$
$\land (\forall d \cdot d \in DIRECTION \land lights'(d) = Green$
$\Rightarrow lights'[CONFLICT[\{d\}]] \subseteq \{Red\})$
END

RedToGreen $\widehat{=}$
ANY
adir
WHERE
grd1: $lights(adir) = Red$
THEN
act1: $lights := lights \mathbin{<\!\!+} (CONFLICT[\{adir\}] \times \{Red\}) \mathbin{<\!\!+} \{adir \mapsto Green\}$
END

ToRed $\widehat{=}$
ANY
adir
WHERE
grd1: $lights(adir) = Green$
THEN
act1: $lights(adir) := Red$
END

END **ChangeLights2**

Notation

math	ascii	
$\mathbin{<\!\!+}$	<+	Override: $r \mathbin{<\!\!+} s$ yields the relation r *overridden by* the relation s. As far as possible $r \mathbin{<\!\!+} s$ behaves like s: $r \mathbin{<\!\!+} s = \mathrm{dom}(s) \lhd r \cup s$

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 *ChangeLight2R* 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, *lights* is refined to *xlights*, (*extra lights*), that introduces Amber.

MACHINE **ChangeLight2R**
REFINES
ChangeLight2
SEES
TrafficLights2_ctx
VARIABLES
xlights
delay
rdir
togreen
tored
INVARIANTS

inv1:	$xlights \in DIRECTION \rightarrow LIGHTS$	
inv2:	$\forall d \cdot d \in DIRECTION \land xlights[\{d\}] \subseteq \{Green, Amber\}$ $\Rightarrow xlights[CONFLICT[\{d\}]] \subseteq \{Red\}$	
inv3:	$rdir \in DIRECTION$	
inv4:	$togreen \in BOOL$	
inv5:	$tored \in BOOL$	
inv6:	$togreen = TRUE \Rightarrow tored = FALSE$	
inv7:	$togreen = TRUE$ $\Rightarrow CONFLICT[\{rdir\}] \lhd lights$ $= CONFLICT[\{rdir\}] \lhd xlights$	
inv8:	$delay \subseteq DIRECTION$	
inv9:	$tored = TRUE$ $\Rightarrow (xlights \Leftarrow \{rdir \mapsto Red\} = lights \Leftarrow \{rdir \mapsto Red\})$	
inv10:	$togreen = FALSE \land tored = FALSE$ $\Rightarrow lights = xlights$	
thm1:	$finite(xlights)$	
thm2:	$\forall d, b, a \cdot d \in DIRECTION$ $\land b \in LIGHTS \land a \in LIGHTS \land a \neq b$ $\land xlights(d) = b$ $\Rightarrow$ $card((xlights \Leftarrow \{d \mapsto a\}) \rhd \{b\})$ $=$ $card(xlights \rhd \{b\}) - 1$	changing light in direction d from b (= before)to a (= after) decreases number of colour b lights by 1
thm3:	$\forall d, b, a \cdot d \in DIRECTION$ $\land b \in LIGHTS \land a \in LIGHTS \land a \neq b$ $\land xlights(d) = b$ $\Rightarrow$ $card((xlights \Leftarrow \{d \mapsto a\}) \rhd \{a\})$ $=$ $card(xlights \rhd \{a\}) + 1$	changing light in direction d from b (= before) to a (=after) increases number of colour a lights by 1
thm4:	$\forall d, b, a, c \cdot d \in DIRECTION$ $\land b \in LIGHTS \land a \in LIGHTS \land c \in LIGHTS$ $\land xlights(d) = b \land c \neq a \land c \neq b$ $\Rightarrow$ $card((xlights \Leftarrow \{d \mapsto a\}) \rhd \{c\})$ $=$ $card(xlights \rhd \{c\})$	changing light in direction d from b (= before) to a (=after) does not change number of colour c, c /= a, c /= b

Notation

math	ascii	
$\lhd$	«|	Domain subtraction: $s \lhd r$ is the subset of r in which s has been subtracted from the domain of r
$\rhd$		Range restriction: $r \rhd s$ is the subset of r in which the range is restricted to the set s

EVENTS

Initialisation $\widehat{=}$

WITH

lights' : $lights' = xlights'$

THEN

act1: $xlights :\mid$ $xlights' \in DIRECTION \to \{Red, Green\}$ $\wedge (\forall d \cdot d \in DIRECTION \wedge xlights'(d) = Green$ $\Rightarrow$ $xlights'[CONFLICT[\{d\}]] \subseteq \{Red\})$

act2: $delay := \varnothing$

act3: $togreen, tored := FALSE, FALSE$

act4: $rdir :\in DIRECTION$

END

RedToGreen $\widehat{=}$

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 $\widehat{=}$

ANY

adir

WHERE

grd1: $togreen = FALSE$

grd2: $tored = FALSE$

grd3: $xlights(adir) = Red$

THEN

act1: $rdir := adir$

act2: $togreen := TRUE$

END

GreenToAmber $\widehat{=}$

STATUS Convergent
ANY
  dir
WHERE
  grd1: $togreen = TRUE$
  grd2: $dir \in CONFLICT[\{rdir\}]$
  grd3: $xlights(dir) = Green$
  grd4: $dir \notin delay$
THEN
  act1: $xlights(dir) := Amber$
  act2: $delay := delay \cup \{dir\}$
END

AmberToRed $\widehat{=}$
STATUS ordinary
convergent ANY
  dir
WHERE
  grd1: $togreen = TRUE$
  grd2: $dir \in CONFLICT[\{rdir\}]$
  grd3: $xlights(dir) = Amber$
  grd4: $dir \notin delay$
THEN
  act1: $xlights(dir) := Red$
  act2: $delay := delay \cup \{rdir\}$
END

Delay $\widehat{=}$
STATUS ordinary
convergent ANY
  dir
WHERE
  grd1: $dir \in delay$
THEN
  act1: $delay := delay \setminus \{dir\}$
END

ToRed $\widehat{=}$
REFINES
  ToRed
WHERE
  grd1: $tored = TRUE$
  grd2: $xlights(rdir) = Amber$
  grd3: $rdir \notin delay$
WITH
  adir : $adir = rdir$
THEN
  act1: $xlights(rdir) := Red$
  act2: $tored := FALSE$
END

ToRedInit $\widehat{=}$
ANY
adir
WHERE
grd1: $xlights(adir) = Green$
grd2: $tored = FALSE$
grd3: $togreen = FALSE$
THEN
act1: $rdir := adir$
act2: $tored := TRUE$
END

ToAmber $\widehat{=}$
WHERE
grd1: $tored = TRUE$
grd2: $xlights(rdir) = Green$
grd3: $rdir \notin delay$
THEN
act1: $xlights(rdir) := Amber$
act2: $delay := delay \cup \{rdir\}$
END

VARIANT
$4 * card(xlights \rhd \{Green\}) + 2 * card(xlights \rhd \{Amber\}) + card(delay)$

END **ChangeLight2R**

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.

State Change

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

$x := e$ *x becomes equal to* the value of e

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

$x :| P$ *x becomes such that* it satisfies the *before-after* predicate P

$x :\in s$ *x becomes in* the set s

All of the above, except apparently $:\in$, can be extended to *multiple assignment*: $x, y := e_1, e_2$ and $x, y :\mid P$, and recursively to many variables. The variables must be distinct.

Note: all assignments can be written in the form: $x, y :\mid P$.

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,

$$x :\mid x' = x + 1$$

and

$$x := x + 1$$

are equivalent.

Similarly we can write

$$x, y :\mid x' = x + 1 \wedge y' = y + 1$$

or

$$x, y := x + 1, y + 1.$$

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,

$$\begin{array}{rl} & [x, y := y - 1, x + 1]x - y < x + y \\ = & (y - 1) - (x + 1) < (y - 1) + (x + 1) \\ \mathit{or} & y - x - 2 < y + x \end{array}$$

This gives the weakest constraint on the *before* state such that $x, y := y - 1, x + 1$ will give an *after* state satisfying $x - y < x + y$.

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 *afer* the action.

Contexts

Sets
S
Constants
C
Axioms
A
Theorems
T_c

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.

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

$$\begin{aligned} &(\exists S, C \cdot a_1 \wedge a_2 \wedge \ldots \wedge a_n) \\ &\quad \equiv \quad \exists S, C \cdot a_1 \wedge (\exists S, C \cdot a_1 \Rightarrow a_2 \wedge \ldots \wedge a_n) \end{aligned}$$

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 $\exists S, C \cdot a_i$.

Theorems

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

$$(\forall S, C . A \Rightarrow T_c)$$

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.

Machines

The form of a machine is:

Context	
	S, C
Variables	
	V
Invariant	
	I
Theorems	
	T_v
Variant	
	Var
Events	
	E

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:

$$\forall S, C, V \cdot A \wedge I \Rightarrow T_v$$

Note: where we have A we could also have $A \wedge T_c$, but since $A \Rightarrow C$ this does not gain any extra strengthening.

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

$$\forall S, C \cdot A \Rightarrow [V := E(S, C)]\ I$$

Events

Events have the following form

ANY	
	x
WHERE	
	G
THEN	
	$Action$

Event: Proof Obligations

There may be feasibility POs: that there exist parameters P that will satisfy the guards G

$$\forall S, C \cdot A \wedge \exists V, x \cdot I \wedge G$$

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.

$$\forall S, C, V, x \cdot A \wedge I \wedge G \Rightarrow [Action]I$$

Machine Refinements

The form of a refinement machine is

Context	
	S_r, C_r
Variables	
	V_r
Invariant	
	I_r
Theorems	
	T_v^+
Events	
	E_r *refines* E
	E
Variant	
	Var

where E_r represents a refined event and E represents new normal events.

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.

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.

Refined Events

Refined Events have the following form

ANY	
	x_r
WHERE	
	G_r
WITH	
	$w : W$
THEN	
	$Action_r$

Proof Obligations for Refined Events

guard refinement

$\forall S, C, S_r, C_r, V, V_r, x, x_r \cdot A \wedge A_r \wedge I \wedge I_r \Rightarrow (G_r \Rightarrow G)$

witness

$\forall S, C, S_r, C_r, V, V_r, x, x_r \cdot A \wedge A_r \wedge I \wedge I_r \Rightarrow (\exists w \cdot W)$

Simulation

$\forall S, C, S_r, C_r, V, V_r, x, x_r \cdot A \wedge A_r \wedge I \wedge I_r \wedge W \wedge [Action_r]I_r \Rightarrow [Action]I$

where A_r denotes the refinement axioms.

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.

Natural number variant

$\forall S, C, S_r, C_r, V, V_r, x, x_r \cdot A \wedge I \wedge I_r \wedge W \Rightarrow Var \in \mathbb{N}$

$\forall S, C, S_r, C_r, V, V_r, x, x_r \cdot A \wedge I \wedge I_r \wedge W \Rightarrow [Action_r]Var < Var$

Set variant

$\forall S, C, S_r, C_r, V, V_r, x, x_r \cdot A \wedge I \wedge I_r \wedge W \Rightarrow \mathit{finite}\ \ Var$

$\forall S, C, S_r, C_r, V, V_r, x, x_r \cdot A \wedge I \wedge I_r \wedge W \Rightarrow card([Action_r]Var) < card(Var)$

6.2 One Point Rule

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

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

$$(\forall x \cdot x \in X \wedge x = e \Rightarrow P(x)) = P(e)$$

By a similar argument,

$$(\exists x \cdot x \in X \wedge x = e \wedge P(x)) = P(e)$$

Strictly, each should be conjuncted with $\exists x \cdot x \in X \wedge x = e$.

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: $QUEUE = \{q \mid q \in \mathbb{N} \rightarrowtail\!\!\!\!\twoheadrightarrow TOKEN \land finite\, q \land \mathrm{dom}\, q = 1 \,..\, card(q)\}$
  thm1: $\varnothing \in QUEUE$
END

7.2 A Queue machine

MACHINE **QueueA**
SEES
  QueueContext
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: $queuetokens \subseteq TOKEN$

inv2: $queue \in QUEUE$

inv3: $qsize \in \mathbb{N}$

inv4: $queue \in 1 \,..\, qsize \rightarrowtail\!\!\!\!\twoheadrightarrow queuetokens$

thm1: $\forall i, j \cdot i \in dom(queue) \land i \neq j$
$\Rightarrow$
$queue(i) \neq queue(j)$

thm2: $queuetokens = ran(queue)$

inv5: $queueitems \in queuetokens \rightarrow ITEM$

thm3: $card(queue) = qsize$

thm4: $queue^{-1} \in queuetokens \rightarrowtail\!\!\!\!\twoheadrightarrow 1 \,..\, qsize$

thm5: $queuetokens \neq \varnothing \Rightarrow qsize \neq 0$

Notation

math	ascii	
$\twoheadrightarrow$ (partial)	+>>	Partial surjection: a surjective function is an *onto* relation which maps to all elements of the range.
$\rightarrowtail\!\!\!\!\twoheadrightarrow$	>->>	Total bijection: a total bijective function is a *one-to-one* and *onto* relation which maps all elements of the domain

EVENTS

Initialisation $\widehat{=}$

THEN

act1: $queuetokens := \varnothing$

act2: $queue := \varnothing$

act3: $qsize := 0$

act4: $queueitems := \varnothing$

END

Enqueue $\widehat{=}$

ANY

item

qid

WHERE

grd1: $item \in ITEM$

grd2: $qid \in TOKEN \setminus queuetokens$

THEN

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

act2: $queue(qsize + 1) := qid$

act3: $queueitems(qid) := item$

act4: $qsize := qsize + 1$

END

Dequeue $\widehat{=}$
WHERE
grd1: $0 < qsize$
THEN
act1: $queue :\mid queue' \in 1 \,..\, qsize - 1 \rightarrowtail\!\!\!\!\rightarrow queuetokens \setminus \{queue(1)\}$
$\wedge\, (\forall i \cdot i \in 1 \,..\, qsize - 1 \Rightarrow queue'(i) = queue(i + 1))$
act2: $queueitems := \{queue(1)\} \lhd\!\!\!- queueitems$
act3: $queuetokens := queuetokens \setminus \{queue(1)\}$
act4: $qsize := qsize - 1$
END

Unqueue $\widehat{=}$
ANY
qid
WHERE
grd1: $qid \in queuetokens$
thm1: $qsize \neq 0$
THEN
act1: $queue :\mid queue' \in 1 \,..\, (qsize - 1) \rightarrowtail\!\!\!\!\rightarrow queuetokens \setminus \{qid\}$
$\wedge\, (qsize = 1 \Rightarrow queue' = \varnothing)$
$\wedge\, (qsize > 1 \Rightarrow$
$(\forall i \cdot i \in 1 \,..\, queue^{-1}(qid) - 1 \Rightarrow queue'(i) = queue(i))$
$\wedge$
$(\forall j \cdot j \in queue^{-1}(qid) + 1 \,..\, qsize \Rightarrow queue'(j - 1) = queue(j)))$
act2: $queueitems := \{qid\} \lhd\!\!\!- queueitems$
act3: $queuetokens := queuetokens \setminus \{qid\}$
act4: $qsize := qsize - 1$
END

END **QueueA**

7.3 A Context that defines an abstract Queue datatype

CONTEXT **QueueType**

Check extension|| EXTENDS

QueueContext

CONSTANTS

ENQUEUE

DEQUEUE

DELETE

AXIOMS

axm1: $ENQUEUE \in QUEUE \times TOKEN \rightarrow QUEUE$

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

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

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

axm4: $DEQUEUE \in QUEUE \nrightarrow QUEUE$

axm5: $dom(DEQUEUE) = QUEUE \setminus \{\varnothing\}$

axm6: $\forall q \cdot q \in QUEUE \land q \neq \varnothing$
$\Rightarrow$
$DEQUEUE(q) \in 1 \,..\, card(q) - 1 \rightarrowtail\!\!\!\!\!\twoheadrightarrow ran(q) \setminus \{q(1)\}$

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

thm2: $\forall q \cdot q \in dom(DEQUEUE) \Rightarrow$
$dom(DEQUEUE(q)) = 1 \,..\, card(q) - 1$

axm8: $\forall q,i \cdot q \in dom(DEQUEUE) \land i \in dom(DEQUEUE(q))$
$\Rightarrow$
$DEQUEUE(q)(i) = q(i+1)$

axm9: $DELETE \in QUEUE \times \mathbb{N}_1 \nrightarrow QUEUE$

axm10: $\forall q,i \cdot q \in QUEUE \land i \in dom(q)$
$\Rightarrow$
$DELETE(q \mapsto i) \in 1 \,..\, card(q) - 1 \rightarrowtail\!\!\!\!\!\twoheadrightarrow ran(q) \setminus \{q(i)\}$

axm11: $\forall q,i \cdot q \in QUEUE \land i \in dom(q)$
$\Leftrightarrow$
$q \mapsto i \in dom(DELETE)$

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

thm3: $\forall q,i \cdot q \mapsto i \in dom(DELETE)$
$\Rightarrow$
$dom(DELETE(q \mapsto i)) = 1 \,..\, card(q) - 1$

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

END

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: $queuetokens \subseteq TOKEN$

inv2: $queue \in QUEUE$

inv3: $qsize = card(queue)$

inv4: $queue \in 1 \mathrel{..} qsize \rightarrowtail\!\!\!\!\twoheadrightarrow queuetokens$

thm1: $\forall i, j \cdot i \in dom(queue) \land j \in dom(queue) \land i \neq j \Rightarrow queue(i) \neq queue(j)$

thm2: $queuetokens = ran(queue)$

inv5: $queueitems \in queuetokens \rightarrow ITEM$

thm3: $queue^{-1} \in queuetokens \rightarrowtail\!\!\!\!\twoheadrightarrow 1 \mathrel{..} qsize$

thm4: $(\forall qid \cdot qid \in TOKEN \setminus queuetokens \Rightarrow ENQUEUE(queue \mapsto qid) = queue \lhd\!\!\!- \{qsize + 1 \mapsto qid\})$

thm5: $\forall qid \cdot qid \in queuetokens \Rightarrow queue \mapsto queue^{-1}(qid) \in dom(DELETE)$

thm6: $qsize \neq 1 \Rightarrow (\forall qid, i \cdot qid \in queuetokens \land i \in 1 \mathrel{..} (queue^{-1}(qid) - 1) \Rightarrow (DELETE(queue \mapsto queue^{-1}(qid)))(i) = queue(i))$

thm7: $qsize \neq 1 \Rightarrow (\forall qid, i \cdot qid \in queuetokens \land i \in queue^{-1}(qid) + 1 \mathrel{..} qsize \Rightarrow (DELETE(queue \mapsto queue^{-1}(qid)))(i - 1) = queue(i))$

thm8: $\forall qid \cdot qid \in queuetokens \Rightarrow queue^{-1}(qid) \leq qsize$

EVENTS

Initialisation $\widehat{=}$

THEN

act1: $queuetokens := \varnothing$

act2: $queue := \varnothing$

act3: $qsize := 0$

act4: $queueitems := \varnothing$

END

Enqueue $\widehat{=}$
REFINES
Enqueue

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

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

Unqueue $\widehat{=}$
REFINES
Unqueue
ANY
qid
WHERE
grd1: $qid \in queuetokens$
THEN
act1: $queue := DELETE(queue \mapsto queue^{-1}(qid))$
act2: $queueitems := \{qid\} \lhd queueitems$
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.

CONTEXT **Iteration**
EXTENDS
    00 Queuetype

CONSTANTS
    iterate
    iclosure

AXIOMS

axm1: $iterate \in (TOKEN \leftrightarrow TOKEN) \times \mathbb{N} \rightarrow (TOKEN \leftrightarrow TOKEN)$

axm2: $\forall r \cdot r \in TOKEN \leftrightarrow TOKEN$
$\Rightarrow$
$iterate(r \mapsto 0) = TOKEN \lhd id$

axm3: $\forall r, n \cdot r \in TOKEN \leftrightarrow TOKEN \land n \in \mathbb{N}_1$
$\Rightarrow$
$iterate(r \mapsto n) = iterate(r \mapsto n - 1); r$

thm1: $\forall s \cdot s \subseteq \mathbb{N} \land 0 \in s$
$\land$
$(\forall n \cdot n \in s \Rightarrow n + 1 \in s) \Rightarrow \mathbb{N} \subseteq s$

thm2: $\forall r, n \cdot r \in TOKEN \leftrightarrow TOKEN \land n \in \mathbb{N}_1$
$\Rightarrow$
$dom(iterate(r \mapsto n)) \subseteq dom(r)$

thm3: $\forall r, n \cdot r \in TOKEN \leftrightarrow TOKEN \land n \in \mathbb{N}_1$
$\Rightarrow$
$ran(iterate(r \mapsto n)) \subseteq ran(r)$

axm4: $iclosure \in (TOKEN \leftrightarrow TOKEN) \rightarrow (TOKEN \leftrightarrow TOKEN)$

axm5: $\forall r \cdot r \in TOKEN \leftrightarrow TOKEN$
$\Rightarrow$
$iclosure(r) = (\bigcup n \cdot n \in \mathbb{N}_1 | iterate(r \mapsto n))$

thm4: $\forall r \cdot r \in TOKEN \leftrightarrow TOKEN$
$\Rightarrow$
$dom(iclosure(r)) \subseteq dom(r)$

END

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

INVARIANTS

inv1: $qfirst \in TOKEN$
inv2: $qlast \in TOKEN$
inv3: $qsize \neq 0 \Rightarrow qfirst = queue(1)$
inv4: $qsize \neq 0 \Rightarrow qlast = queue(qsize)$
inv5: $qnext \in queuetokens \rightarrowtail\!\!\!\!\!\!+\; queuetokens$
inv6: $dom(qnext) = queuetokens \setminus \{qlast\}$
inv7: $qnext \cap id = \varnothing$
inv8: $ran(qnext) = queuetokens \setminus \{qfirst\}$
thm1: $qsize = 1 \Rightarrow qfirst = qlast$
inv9: $\forall i \cdot i \in 1\,..\,qsize \land i < qsize \Rightarrow qnext(queue(i)) = queue(i+1)$
thm2: $qsize \geq 1 \Rightarrow iterate(qnext \mapsto 0)[\{qfirst\}] = \{queue(1)\}$
thm3: $qsize \geq 1 \Rightarrow (\forall n \cdot n \in 1\,..\,qsize - 1 \land iterate(qnext \mapsto n-1)[\{qfirst\}] = \{queue(n)\} \Rightarrow iterate(qnext \mapsto n)[\{qfirst\}] = \{queue(n+1)\})$
thm4: $qsize \geq 1 \Rightarrow (\forall n \cdot n \in 1\,..\,qsize - 1 \Rightarrow iterate(qnext \mapsto n-1)[\{qfirst\}] = \{queue(n)\})$
thm5: $qsize \geq 1 \Rightarrow iclosure(qnext)[\{qfirst\}] = queuetokens$

Notation

math	ascii	
$\rightarrowtail\!\!\!\!\!\!+$	>+>	partial injective function; injections are one-to-one relations

EVENTS
Initialisation $\widehat{=}$
THEN
act1: $queuetokens := \varnothing$
act2: $qsize := 0$
act3: $queueitems := \varnothing$
act4: $qfirst :\in TOKEN$
act5: $qlast :\in TOKEN$
act6: $qnext := \varnothing$
END

Enqueue0 $\widehat{=}$
REFINES
Enqueue

ANY
  item
  qid
WHERE
  grd1: $item \in ITEM$
  grd2: $qid \in TOKEN \setminus queuetokens$
  grd3: $qsize = 0$
THEN
  act1: $queuetokens := queuetokens \cup \{qid\}$
  act2: $queueitems(qid) := item$
  act3: $qsize := qsize + 1$
  act4: $qfirst := qid$
  act5: $qlast := qid$
END

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

Dequeue0 $\widehat{=}$
REFINES
  Dequeue
WHERE
  grd1: $qsize = 1$
THEN
  act1: $qsize := qsize - 1$
  act2: $queuetokens := queuetokens \setminus \{qfirst\}$
  act3: $queueitems := \{qfirst\} \lhd queueitems$
  act4: $qnext := \{qfirst\} \lhd qnext$
END

Dequeue1 $\widehat{=}$
REFINES
  Dequeue
WHERE
  grd1: $qsize > 1$

THEN
act1: $qsize := qsize - 1$
act2: $queuetokens := queuetokens \setminus \{qfirst\}$
act3: $queueitems := \{qfirst\} \lhd queueitems$
act4: $qfirst := qnext(qfirst)$
act5: $qnext := \{qfirst\} \lhd qnext$
END

Unqueue0 $\widehat{=}$
REFINES
Unqueue
ANY
qid
WHERE
grd1: $qid \in queuetokens$
grd2: $qsize = 1$
THEN
act1: $queueitems := \{qid\} \lhd queueitems$
act2: $queuetokens := queuetokens \setminus \{qid\}$
act3: $qsize := qsize - 1$
END

Unqueue1 $\widehat{=}$
REFINES
Unqueue
ANY
qid
WHERE
grd1: $qid \in queuetokens$
grd2: $qsize > 1$
grd3: $qid = qfirst$
THEN
act1: $queueitems := \{qid\} \lhd queueitems$
act2: $queuetokens := queuetokens \setminus \{qid\}$
act3: $qsize := qsize - 1$
act4: $qfirst := qnext(qid)$
act5: $qnext := \{qid\} \lhd qnext$
END

Unqueue2 $\widehat{=}$
REFINES
Unqueue
ANY
qid
WHERE
grd1: $qid \in queuetokens$
grd2: $qsize > 1$
grd3: $qlast = qid$
THEN

act1: $queueitems := \{qid\} \lhd queueitems$
act2: $queuetokens := queuetokens \setminus \{qid\}$
act3: $qsize := qsize - 1$
act4: $qlast := qnext^{-1}(qid)$
act5: $qnext := qnext \rhd \{qid\}$
END

Unqueue3 $\widehat{=}$
REFINES
Unqueue
ANY
qid
WHERE
grd1: $qid \in queuetokens$
grd2: $qsize > 1$
grd3: $qfirst \neq qid$
grd4: $qlast \neq qid$
THEN
act1: $queueitems := \{qid\} \lhd queueitems$
act2: $queuetokens := queuetokens \setminus \{qid\}$
act3: $qsize := qsize - 1$
act4: $qnext(qnext^{-1}(qid)) := qnext(qid)$
END

END **QueueR**

Notation

math	ascii	
r^{-1}	`r~`	inverse of r, that is $r; r^{-1} \subseteq id$. Note Rodin, even in marked-up form retains the ˜

7.6 Further refinement

While the current refinement can be considered to be close to a concrete model that would map reasonably easily into a concrete implementation there is one construct that cannot be considered as concrete: $qnext^{-1}$ 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 $qnext^{-1}$, 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: $deleting = TRUE \Rightarrow qprev \in dom(qnext)$

inv8: $deleting = TRUE$ $\Rightarrow$ $qidv \in iclosure(qnext)[\{qprev\}]$

EVENTS

Initialisation : ***extended*** $\widehat{=}$

THEN

act7: $deleting := FALSE$

act8: $qprev :\in TOKEN$

act9: $qidv :\in TOKEN$

END

Enqueue0 : ***extended*** $\widehat{=}$

REFINES

Enqueue0

ANY

WHERE

grd4: $deleting = FALSE$

THEN

END

Enquire1 : ***extended*** $\widehat{=}$

REFINES

Enquire1

ANY

WHERE

grd4: $deleting = FALSE$

THEN

END

Dequque0 : ***extended*** $\widehat{=}$
REFINES
Dequque0
ANY

WHERE
grd2: $deleting = FALSE$
THEN

END

Dequeue1 : ***extended*** $\widehat{=}$
REFINES
Dequque1
WHERE
grd2: $deleting = FALSE$
THEN

END

Unqueue0 : ***extended*** $\widehat{=}$
REFINES
Unqueue0
ANY

WHERE
grd3: $deleting = FALSE$
THEN

END

Unqueue1 : ***extended*** $\widehat{=}$
REFINES
Unqueue1
ANY

WHERE

THEN

END

Unqueue2 $\widehat{=}$
REFINES
Unqueue2
WHERE

grd1: $deleting = TRUE$
grd2: $qnext(qprev) = qidv$
grd3: $qlast = qidv$
WITH
qid: $qid = qidv$
THEN
act1: $queueitems := \{qidv\} \lhd queueitems$
act2: $queuetokens := queuetokens \setminus \{qid\}$
act3: $qsize := qsize - 1$
act4: $qlast := qprev$
act5: $qnext := qnext \rhd \{qidv\}$
act6: $deleting := FALSE$
END

Unqueue3 $\widehat{=}$
REFINES
Unqueue3
WHERE
grd1: $deleting = TRUE$
grd2: $qnext(qprev) = qidv$
grd3: $qidv \neq qlast$
WITH
qid qid = qidv
THEN
act1: $queueitems := \{qidv\} \lhd queueitems$
act2: $queuetokens := queuetokens \setminus \{qidv\}$
act3: $qsize := qsize - 1$
act4: $qnext(qprev) := qnext(qidv)$
act5: $deleting := FALSE$
END

UnqueueI $\widehat{=}$ Initialise for search
ANY
qid
WHERE
grd1: $qid \in queuetokens$
grd2: $qsize > 1$
grd3: $qfirst \neq qid$
grd4: $deleting = FALSE$
THEN
act1: $qprev := qfirst$
act2: $qidv := qid$
act3: $deleting := TRUE$
END

UnqueueS $\widehat{=}$ Search for predecessor
STATUS convergent
WHERE

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

THEN

act1: $qprev := qnext(qprev)$

END

VARIANT

$iclosure(qnext)[\{qprev\}]$

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 list 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 $LIFT$.

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

- **MOVING:** the lift is *active* and moving;
- **STOPPED:** the lift is active and stopped;
- **IDLE:** the lift is *inactive*, but capable of becoming active.

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 $0 \mathrel{..} MAXFLOOR$, where $MAXFLOOR$ is at least 1, giving distinct top and bottom floors.

Lift Context

CONTEXT **Lift_ctx**

SETS

- DIRECTION
- STATUS
- LIFT

CONSTANTS

- MAXFLOOR
- FLOOR
- UP
- DOWN
- IDLE
- STOPPED
- MOVING
- CHANGE

AXIOMS

- axm1: $MAXFLOOR \in \mathbb{N}1$
- axm2: $FLOOR = 0 \mathrel{..} MAXFLOOR$
- axm3: $\mathit{finite}\, LIFT$
- axm4: $DIRECTION = \{UP, DOWN\}$
- axm5: $UP \neq DOWN$
- axm6: $\mathit{partition}(STATUS, \{IDLE\}, \{STOPPED\}, \{MOVING\})$
- axm7: $CHANGE \in DIRECTION \rightarrowtail\!\!\!\!\twoheadrightarrow DIRECTION$
- axm8: $CHANGE = \{UP \mapsto DOWN, DOWN \mapsto UP\}$
- thm1: $FLOOR \neq \varnothing$
- thm2: $\mathit{finite}\, FLOOR$
- thm3: $\mathit{finite}\, STATUS$
- thm4: $\mathit{finite}\, DIRECTION$
- thm5: $\mathit{finite}\, CHANGE$

END

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:	$liftposition \in LIFT \rightarrow FLOOR$
thm1:	$finite\, liftposition$
inv2:	$liftstatus \in LIFT \rightarrow STATUS$
thm2:	$finite\, liftstatus$
inv3:	$liftdirection \in LIFT \rightarrow DIRECTION$
thm3:	$finite\, liftdirection$
inv4:	$\forall l \cdot l \in LIFT \wedge liftposition(l) = 0$ $\Rightarrow liftdirection(l) = UP$
inv5:	$\forall l \cdot l \in LIFT \wedge liftposition(l) = MAXFLOOR$ $\Rightarrow liftdirection(l) = DOWN$
thm4:	$\forall l \cdot l \in LIFT \wedge liftdirection(l) = DOWN$ $\Rightarrow liftposition(l) \neq 0$
thm5:	$\forall l \cdot l \in LIFT \wedge liftdirection(l) = UP$ $\Rightarrow liftposition(l) \neq MAXFLOOR$

EVENTS
Initialisation $\widehat{=}$
THEN

act1:	$liftposition := LIFT \times \{0\}$
act2:	$liftdirection := LIFT \times \{UP\}$
act3:	$liftstatus := LIFT \times \{IDLE\}$

END

Notation

math	ascii	
$\times$	**	$A \times B$ is the set of all maplets, $a \mapsto b$, in which $a \in A$ and $b \in B$

IdleLift $\widehat{=}$ Idle lifts cannot move

ANY
    lift
WHERE
    grd1: $liftstatus(lift)$ STOPPED
THEN
    act1: $liftstatus(lift) := IDLE$
END

ActivateLift $\widehat{=}$ Ready an Idle lift to enable moving

ANY
    lift
WHERE
    grd1: $liftstatus(lift)$ IDLE
THEN
    act1: $liftstatus(lift) := STOPPED$
END

StartLift $\widehat{=}$ Models starting of a stopped lift, maintaining the previous direction

ANY
    lift
WHERE
    grd1: $liftstatus(lift) = STOPPED$
THEN
    act1: $liftstatus(lift) := MOVING$
END

ChangeDir $\widehat{=}$ Models the changing of direction of a STOPPED lift

ANY
    lift
WHERE
    grd1: $liftstatus(lift) = STOPPED$
    grd2: $liftposition(lift) \neq 0$
    grd3: $liftposition(lift) \neq MAXFLOOR$
THEN
    act1: $liftdirection(lift) := CHANGE(liftdirection(lift))$
END

MoveUp $\widehat{=}$ Models a lift moving up to the next floor and continuing to move

ANY

WHERE
    grd1: $liftstatus(lift) = MOVING$
    grd2: $liftdirection(lift) = UP$
    grd3: $liftposition(lift) \neq MAXFLOOR - 1$
THEN
    act1: $liftposition(lift) := liftposition(lift) + 1$
END

MoveUpAndStop $\widehat{=}$ Models a lift moving up to the next floor and stopping

ANY
lift
WHERE
grd1: $liftstatus(lift) = MOVING$
grd2: $liftdirection(lift) = UP$
THEN
act1: $liftposition(lift) := liftposition(lift) + 1$
$liftdirection :| \; liftdirection' \in LIFT \rightarrow DIRECTION$
$\wedge \, (liftposition(lift) + 1 = MAXFLOOR$
$\Rightarrow$
act2: $liftdirection' = liftdirection \lhd\!\!- \{lift \mapsto DOWN\})$
$\wedge \, (liftposition(lift) + 1 \neq MAXFLOOR$
$\Rightarrow$
$liftdirection' = liftdirection)$
act3: $(liftstatus(lift) := STOPPED$
END

MoveDown $\widehat{=}$ Models a lift moving down to the next floor and continuing to move

ANY
lift
WHERE
grd1: $liftstatus(lift) = MOVING$
grd2: $liftdirection(lift) = DOWN$
grd3: $liftposition(lift) \neq 1$
THEN
act1: $liftposition(lift) := liftposition(lift) - 1$
END

MoveDownAndStop $\widehat{=}$ Models a lift moving down to the next floorand stopping

ANY
lift
WHERE
grd1: $liftstatus(lift) = MOVING$
grd2: $liftdirection(lift) = DOWN$
THEN
act1: $liftposition(lift) := liftposition(lift) - 1$
$liftdirection :| \; liftdirection' \in LIFT \rightarrow DIRECTION$
$\wedge \, (liftposition(lift) = 1$
$\Rightarrow$
act2: $liftdirection' = liftdirection \lhd\!\!- \{lift \mapsto UP\})$
$\wedge \, (liftposition(lift) + 1 \neq 1$
$\Rightarrow$
$liftdirection' = liftdirection)$
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.

Door Context

CONTEXT **Doors_ctx**
SETS
DOORS
CONSTANTS
CLOSED
OPENING
OPEN
CLOSING
AXIOMS
axm1: $partition(DOORS, \{CLOSED\}, \{OPENING\}, \{OPEN\}, \{CLOSING\})$
END

Lift Plus Doors

MACHINE **LiftPlusDoors**
REFINES
BasicLift
SEES
Lift_ctx
Doors_ctx
VARIABLES
liftposition
liftstatus
liftdirection
liftdoorstatus
INVARIANTS
inv1: $liftdoorstatus \in LIFT \to DOORS$
thm1: $finite(liftdoorstatus)$
inv2: $\forall l \cdot l \in LIFT \land liftstatus(l) \in \{MOVING, IDLE\}$
$\Rightarrow$
$liftdoorstatus(l) = CLOSED$
thm2: $\forall l \cdot l \in LIFT \land liftdoorstatus(l) \in \{OPENING, OPEN\}$
$\Rightarrow$
$liftstatus(l) = STOPPED$
EVENTS
INITIALISATION : ***extended*** $\widehat{=}$
THEN
act4: $liftdoorstatus := LIFT \times \{CLOSED\}$
END

OpenLiftDoor $\widehat{=}$ Open lift door: lift must be STOPPED
ANY
lift
WHERE
grd1: $liftstatus(lift) = STOPPED$
grd2: $liftdoorstatus(lift) = OPENING$
THEN
act1: $liftdoorstatus(lift) := OPEN$
END

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

IdleLift : *extended* $\widehat{=}$ Idle lifts cannot move
REFINES
IdleLift
WHERE
grd2: $liftdoorstatus(lift) = CLOSED$
END

ActivateLift : *extended* $\widehat{=}$ Ready an Idle lift to enable moving
REFINES
ActivateLift
THEN
act2: $liftdoorstatus :| liftdoorstatus' \in LIFT \rightarrow DOORS \wedge$
$((liftdoorstatus' = liftdoorstatus \lhd\!\!- \{lift \mapsto CLOSED\})$
$\vee$
$(liftdoorstatus' = liftdoorstatus \lhd\!\!- \{lift \mapsto OPENING\}))$
END

StartLift : *extended* $\widehat{=}$
REFINES
StartLift
WHERE
grd2: $liftdoorstatus(lift) = CLOSED$
END

ChangeDir : *extended* $\widehat{=}$
REFINES
ChangeDir

END

MoveUp : ***extended*** $\widehat{=}$ Models a lift moving up to the next floor and continuing to move
REFINES
MoveUp
END

MoveUpAndStop : ***extended*** $\widehat{=}$ Models a lift moving up to the next floor and stopping
REFINES
MoveUp
THEN
act4: $liftdoorstatus(lift) := OPENING$
END

MoveDown : ***extended*** $\widehat{=}$ Models a lift moving down to the next floor and continuing to move
REFINES
MoveDown
END

MoveDownAndStop : ***extended*** $\widehat{=}$ Models a lift moving down to the next floorand stopping
REFINES
MoveDownAndStop
THEN
act4: $liftdoorstatus(lift) := OPENING$
END

END **LiftPlusDoors**

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
INVARIANTS
  inv1: $floordoorstatus \in LIFT \to (FLOOR \to DOORS)$
  thm1: $finite(floordoorstatus)$
  inv2: $\forall l \cdot l \in LIFT \land liftdoorstatus(l) \neq OPEN \Rightarrow floordoorstatus(l)(liftposition(l)) = CLOSED$
  inv3: $\forall l, f \cdot l \in LIFT \land f \in FLOOR \setminus \{liftposition(l)\} \Rightarrow floordoorstatus(l)(f) = CLOSED$
  thm2: $\forall l, f \cdot l \in LIFT \land f \in FLOOR \land liftstatus(l) = MOVING \Rightarrow floordoorstatus(l)(f) = CLOSED$
  thm3: $\forall l \cdot l \in LIFT \land floordoorstatus(l)(liftposition(l)) \neq CLOSED \Rightarrow liftdoorstatus(l) \neq CLOSED$
  inv4: $\forall l \cdot l \in LIFT \land floordoorstatus(l)(liftposition(l)) \neq CLOSED \Rightarrow liftstatus(l) = STOPPED$
EVENTS
INITIALISATION : *extended* $\widehat{=}$
THEN
  act5: $floordoorstatus := LIFT \times \{FLOOR \times \{CLOSED\}\}$
END

OpenFloorDoor $\widehat{=}$
ANY
  lift
WHERE

grd1: $liftstatus(lift) = STOPPED$
grd2: $liftdoorstatus(lift) = OPEN$
grd3: $floordoorstatus(lift)(liftposition(lift)) = OPENING$
THEN
act1: $floordoorstatus(lift) := floordoorstatus(lift) \lhd\!\!- \{liftposition(lift) \mapsto OPEN\}$
END

CloseFloorDoor $\widehat{=}$
ANY
lift
WHERE
grd1: $floordoorstatus(lift)(liftposition(lift)) = OPEN$
THEN
act1: $floordoorstatus(lift) := floordoorstatus(lift) \lhd\!\!- \{liftposition(lift) \mapsto CLOSED\}$
END

OpenLiftDoor : ***extended*** $\widehat{=}$
REFINES
OpenLiftDoor
WHERE

THEN
act2: $floordoorstatus(lift) := floordoorstatus(lift) \lhd\!\!- \{liftosition(lift) \mapsto OPENING\}$
END

CloseLiftDoor : ***extended*** $\widehat{=}$
REFINES
CloseLiftDoor
WHERE
grd2: $floordoorstatus(lift)(liftposition(lift)) = CLOSED$
END

StartLift : ***extended*** $\widehat{=}$
REFINES
StartLift
WHERE
grd2: $liftdoorstatus(lift) = CLOSED$
END

ChangeDir : ***extended*** $\widehat{=}$
REFINES
ChangeDir
END

MoveUp : ***extended*** $\widehat{=}$ Models a lift moving up to the next floor
REFINES

MoveUp
END

MoveUpAndStop : ***extended*** $\hat{=}$ Models a lift moving up to the next floor and stopping
REFINES
MoveUpAndStop
END

MoveDown : ***extended*** $\hat{=}$ Models a lift moving down to the next floor
REFINES
MoveDown
END

MoveDownAndStop : ***extended*** $\hat{=}$ 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 intentions: both inside the lifts and on the floors of the building.

Buttons inside Lift

Buttons Context

CONTEXT
SETS
BUTTONS
CONSTANTS
ON
OFF
AXIOMS
axm1: $partition(BUTTONS, \{ON\}, \{OFF\})$
END

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

INVARIANTS
  inv1: $liftbuttons \in LIFT \rightarrow (FLOOR \rightarrow BUTTONS)$
  inv2: $liftschedule \in LIFT \rightarrow \mathbb{P}(FLOOR)$
  thm1: $\forall l \cdot l \in LIFT \Rightarrow finite(liftschedule(l))$
  inv3: $\forall l, f \cdot l \in dom(liftbuttons) \land f \in dom(liftbuttons(l))$
  $\Rightarrow (liftbuttons(l)(f) = ON \Rightarrow f \in liftschedule(l))$
  inv4: $\forall l \cdot l \in LIFT \land liftposition(l) \in liftschedule(l)$
  $\Rightarrow liftstatus(l) = STOPPED$
  thm2: $\forall l \cdot l \in LIFT \land liftstatus(l) = MOVING$
  $\Rightarrow liftposition(l) \notin liftschedule(l)$
EVENTS

INITIALISATION : *extended* $\widehat{=}$
THEN
  act6: $liftbuttons := LIFT \times \{FLOOR \times \{OFF\}\}$
  act7: $liftschedule := LIFT \times \{\varnothing\}$
END

SelectFloor $\widehat{=}$
ANY
  lift
  floor
WHERE

grd1: $floor \in FLOOR$
grd2: $liftbuttons(lift)(floor) = OFF$
grd3: $liftposition(lift) \neq floor$
THEN
act1: $liftbuttons(lift) := liftbuttons(lift) \lhd\!\!- \{floor \mapsto ON\}$
act2: $liftschedule(lift) := liftschedule(lift) \cup \{floor\}$
END

MoveUp : ***extended*** $\widehat{=}$ Models a lift moving up to the next floor
REFINES
MoveUp
WHERE
grd4: $liftschedule(lift) \neq \varnothing$
grd5: $liftposition(lift) < max(liftschedule(lift))$
grd6: $liftposition(lift) + 1 \notin liftschedule(lift)$
END

MoveUpAndStop : ***extended*** $\widehat{=}$ Models a lift moving up to the next floor and stopping
REFINES
MoveUpAndStop
WHERE
grd3: $liftposition(lift) + 1 \in liftschedule(lift)$
END

MoveDown : ***extended*** $\widehat{=}$ Models a lift moving down to the next floor
REFINES
MoveDown
WHERE
grd4: $liftschedule(lift) \neq \varnothing$
grd5: $liftposition(lift) > min(liftschedule(lift))$
grd6: $liftposition(lift) - 1 \notin liftschedule(lift)$

END

MoveDownAndStop : ***extended*** $\widehat{=}$ Models a lift moving down to the next floor and stopping
REFINES
MoveDownAndStop
WHERE
grd3: $liftposition(lift) - 1 \in liftschedule(lift)$
END

ActivateLiftClosed $\widehat{=}$ Ready an Idle lift to enable moving, but leave doors CLOSED
REFINES
ActivateLift
ANY
lift
WHERE

grd1: $liftstatus(lift)$ IDLE
grd2: $liftchedule(lift) \neq \varnothing$
grd3: $liftposition(lift) \notin liftschedule(lift)$
THEN
act1: $liftstatus(lift) := STOPPED$
act2: $liftdoorstatus := liftdoorstatus \lhd\!\!- \{lift \mapsto CLOSED\}$
END

ActivateLiftOpen $\widehat{=}$ Ready an Idle lift to enable moving, but commence opening doors
REFINES
ActivateLift
ANY
lift
WHERE
grd1: $liftstatus(lift)$ IDLE
grd2: $liftchedule(lift) \neq \varnothing$
grd3: $liftposition(lift) \notin liftschedule(lift)$
THEN
act1: $liftstatus(lift) := STOPPED$
act2: $liftdoorstatus := liftdoorstatus \lhd\!\!- \{lift \mapsto OPENING\}$
END

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

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

OpenFloorDoor : *extended* $\widehat{=}$
REFINES
OpenFloorDoor

WHERE
grd4: $liftposition(lift) \in liftschedule(lift)$
END

CloseFloorDoor : *extended* $\widehat{=}$
REFINES
CloseFloorDoor
THEN
act2: $liftschedule(lift) := liftschedule(lift) \setminus liftposition(lift)$
act3: $liftbuttons(lift) := liftbuttons(lift) \lhd\!\!\!- \{liftposition(lift) \mapsto OFF\}$
END

OpenLiftDoor : *extended* $\widehat{=}$
REFINES
OpenLiftDoor
WHERE
grd3: $liftposition(lift) \in liftschedule(lift)$
END

CloseLiftDoor : *extended* $\widehat{=}$
REFINES
CloseLiftDoor
END

IdleLift : *extended* $\widehat{=}$ Idle lifts cannot move
REFINES
IdleLift
WHERE
grd3: $liftschedule(lift) = \varnothing$
END

StartLift : *extended* $\widehat{=}$
REFINES
StartLift
WHERE
grd3: $liftschedule(lift) \neq \varnothing$
grd4: $liftdirection(lift) = DOWN$
$\Rightarrow$
$liftposition(lift) > min(liftschedule(lift))$
grd5: $liftdirection(lift) = UP$
$\Rightarrow$
$liftposition(lift) < max(liftschedule(lift))$
grd6: $liftposition(lift) \notin liftschedule(lift)$
END

ChangeDir : ***extended*** $\widehat{=}$
REFINES
ChangeDir
WHERE
grd4: $liftschedule(lift) \neq \varnothing$
$liftdirection(lift) = UP$
grd5: $\Rightarrow$
$liftposition(lift) > max(liftschedule(lift))$
$liftdirection(lift) = DOWN$
grd6: $\Rightarrow$
$liftposition(lift) < min(liftschedule(lift))$
END

END **LiftButtons**

Floor Buttons

The next layer refines *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:

Id	Name	Needed to discharge
INV	Invariant	
FIS	Feasibility	
WD	Well-definedness	
GRD	Guard	
EQL	Equal	

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

$$phone \in FRIENDS \leftrightarrow PHONE$$

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

$$phone = \{jim \mapsto 0456123456, lisa \mapsto 0423456234, jim \mapsto 0293984321, \ldots\}$$

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: $f \in X \nrightarrow Y$, a function that *may not* be defined everwhere in X, for example a function between *friend* and their *partner*.

Total functions: $f \in X \rightarrow Y$, a function that is defined everywher 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: $f \in X \rightarrowtail\!\!\!\!\!\!+ Y$, or $f \in X \rightarrowtail Y$, *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: $f \in X \twoheadrightarrow\!\!\!\!\!\!+ Y$, or $f \in X \twoheadrightarrow Y$, *onto* functions, where each value of Y is equal to $f(x)$ for some value of x in X.

Bijective functions: $f \in X \rightarrowtail\!\!\!\!\!\twoheadrightarrow Y$, *one-to-one and onto* functions.

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

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.

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 $members \cup \{newmember\}$. 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,

$$A \subseteq B \wedge a \in A \Rightarrow a \in B.$$

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, $card(S)$, the number of elements in S. Infinite sets do not have cardinality; EventB does not have an infinity.

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

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

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

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

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

Also,

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

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:

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

A rule for products is

$$a \mapsto b \in A \times B \Rightarrow a \in A \wedge b \in EventB$$

The following sets are used in the exercises:

$$\begin{array}{lcl} NAMES & = & \{Jack, Jill\}; \\ PHONE & = & \{123, 456, 789\} \end{array}$$

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 card($NAMES \times PHONE$)?
    m) What is card($NAMES \times \{\}$)?
    n) What is $\mathbb{P}(S)$?
    o) Given card(S) $= N$, what is card($\mathbb{P}(S)$)?
    p) What is card($\mathbb{P}(NAMES \times PHONE)$)?
    q) What does card($\mathbb{P}(NAMES \times PHONE)$) 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 a *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 $X \leftrightarrow Y = \mathbb{P}(X \times Y)$. This could be expressed by a rule:

$$r \in X \leftrightarrow Y \Rightarrow r \in \mathbb{P}(X \times Y)$$

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

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

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

11. Given $phonebook \in NAMES \leftrightarrow PHONE$,
    a) Give some examples of *phonebook*.
    b) Give $NAMES \leftrightarrow PHONE$.
    c) What is $\mathrm{card}(NAMES \leftrightarrow PHONE)$?

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

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

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

$$r \in X \leftrightarrow Y \wedge s \subseteq X \Rightarrow s \lhd r \subseteq r \wedge \mathrm{dom}(s \lhd r) \subseteq s$$

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

$$r \in X \leftrightarrow Y \wedge s \subseteq Y \Rightarrow r \rhd s \subseteq r \wedge \mathrm{ran}(r \rhd s) \subseteq s$$

Relational image The relational image $r[s]$ give the image of a set s under the relation r: the mapping of all elements of s according to the maplets in r:

$$r[s] = \mathrm{ran}(s \lhd r)$$

Relational image is the counterpart for relations of functional application for functions; the former being many-to-many and the latter many-to-one.

12. Given $phonebook = \{Jack \mapsto 123, Jack \mapsto 789, Jill \mapsto 456, Jill \mapsto 789\}$
    a) What is $\mathrm{dom}(phonebook)$?
    b) What is $\mathrm{ran}(phonebook)$?
    c) What is $phonebook \mathbin{\lhd\!\!\!+} \{Jack \mapsto 123\}$?

d) What is $\{Jack\} \lhd phonebook$?

e) What is $\{Jack\} \lhd\!\!\!- phonebook$?

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

g) What is $phonebook \rhd\!\!\!- \{123, 789\}$?

h) What is $phonebook[\{Jack\}]$?

Functions Functions are *many-to-one relations.* $X \nrightarrow Y$ 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:

$$f \in X \nrightarrow Y \wedge x \mapsto u \in f \wedge x \mapsto v \Rightarrow u = v$$

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

$$f \in X \nrightarrow Y \wedge x \mapsto y \Rightarrow f(x) = y$$

13. a) Give $NAMES \nrightarrow PHONE$.

    b) What is $\mathrm{card}(NAMES \nrightarrow PHONE)$?

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:

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

14. a) Give $NAMES \rightarrow PHONE$.

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

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

$$f \in X \rightarrowtail\!\!\!\!\!+ Y \wedge u \mapsto y \in f \wedge v \mapsto y \in f \Rightarrow u = v$$

15. a) Give $NAMES \rightarrowtail\!\!\!\!\!+ PHONE$.

    b) What is $\mathrm{card}(NAMES \rightarrowtail\!\!\!\!\!+ PHONE)$?

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

$$f \in X \rightarrowtail Y \Rightarrow f \in X \rightarrow Y \wedge f \in X \rightarrowtail\!\!\!\!\!+ Y$$

16. a) Give $NAMES \rightarrowtail PHONE$.

    b) What is $\mathrm{card}(NAMES \rightarrowtail PHONE)$?

Surjective functions $X \mathrel{+\!\!\!\!\twoheadrightarrow} Y$ 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:

$$f \in X \mathrel{+\!\!\!\!\twoheadrightarrow} Y => \mathrm{ran}(f) = Y$$

17. a) Give $NAMES \mathrel{+\!\!\!\!\twoheadrightarrow} PHONE$.
    b) What is $\mathrm{card}(NAMES \mathrel{+\!\!\!\!\twoheadrightarrow} PHONE)$?

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

$$f \in X \twoheadrightarrow Y \Rightarrow f \in X \rightarrow Y \wedge f \in X \nrightarrow Y$$

18. a) Give $NAMES \twoheadrightarrow PHONE$.
    b) What is $\mathrm{card}(NAMES \twoheadrightarrow PHONE)$?

Bijective functions $X \mathrel{\rightarrowtail\!\!\!\!\twoheadrightarrow} Y$ is the set of *all (total) injective and surjective* functions formed from X and Y. A bijective function is *total*, *injective* and *surjective*.

$$f \in X \mathrel{\rightarrowtail\!\!\!\!\twoheadrightarrow} Y \Rightarrow f \in f \rightarrow Y \wedge f \in f \mathrel{\rightarrowtail\!\!\!\!\!+} Y \wedge f \in f \mathrel{+\!\!\!\!\twoheadrightarrow} Y$$

19. a) Give $NAMES \mathrel{\rightarrowtail\!\!\!\!\twoheadrightarrow} PHONE$.
    b) What is $\mathrm{card}(NAMES \mathrel{\rightarrowtail\!\!\!\!\twoheadrightarrow} PHONE)$?
    c) Why is a partial bijection unnecessary?

20. Suppose $STUDENTS$ 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 $results \in STUDENTS \nrightarrow \mathbb{N}$, 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, $drivers$, $taxis$ and $assigned$ constrained by

$$\begin{array}{rcl} drivers & \in & \mathbb{P}(DRIVERS) \\ taxis & \in & \mathbb{P}(TAXIS) \\ assigned & \in & drivers \mathrel{\rightarrowtail\!\!\!\!\!+} taxis \end{array}$$

    where $DRIVERS$ is the set of possible drivers, $TAXIS$ is the set of possible taxis, $drivers$ is the set of drivers working for the company, $taxis$ is the set of taxis owned by the company, and $assigned$ is a function recording the assignment of drivers to taxis.

    The arrow inj denotes a *partial injective* function. An injective function is a one-to-one function.

    a) Why is $assigned$ a function?
    b) Why is $assigned$ a partial function?
    c) Why is $assigned$ an injective function?
    d) Specify the drivers who are currently assigned.

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) $f \in X \rightarrow Y \Rightarrow f \in X \pfun Y$

b) $f \in X \pinj Y \Rightarrow f \in X \pfun Y$

c) $f \in X \pinj Y \Rightarrow f \in X \rightarrow Y$

d) $f \in X \inj Y \Rightarrow f \in X \rightarrow Y$

e) $f \in X \inj Y \Rightarrow f \in X \pfun Y$

f) $f \in X \psurj Y \Rightarrow f \in X \pfun Y$

g) $f \in X \psurj Y \Rightarrow f \in X \rightarrow Y$

h) $f \in X \surj Y \Rightarrow f \in X \rightarrow Y$

i) $f \in X \surj Y \Rightarrow f \in X \pfun Y$

j) $f \in X \pfun Y \Rightarrow \mathrm{dom}(f) \subset X$

k) $f \in X \rightarrow Y \Rightarrow \mathrm{ran}(f) = Y$

l) $f \in X \pfun Y \wedge x \in \mathrm{dom}(f) \Rightarrow f[\{x\}] = \{f(x)\}$

m) $(r^{-1})^{-1} = r$

n) $r \in X \leftrightarrow Y \Rightarrow \mathrm{dom}(r^{-1}) = \mathrm{ran}(r)$

o) $r \in X \leftrightarrow Y \Rightarrow \mathrm{ran}(r^{-1}) = \mathrm{dom}(r)$

p) $r \in X \leftrightarrow Y \Rightarrow \mathrm{ran}(r) \in Y$

q) $r \in X \leftrightarrow Y \Rightarrow \mathrm{ran}(r) \subseteq Y$

r) $r \in X \leftrightarrow Y \Rightarrow \mathrm{ran}(r) \in \mathbb{P}(Y)$

23. $\mathrm{union}(S)$ is the generalised union of the elements of S, that is, if S is a set of sets, then $\mathrm{union}(S)$ is the union of all of the sets that are contained in S. What is $\mathrm{union}(\{\})$?

24. $\mathrm{inter}(S)$ is the generalised intersection of the elements of S, that is, if S is a set of sets, then $\mathrm{inter}(S)$ is the intersection of all of the sets that are contained in S. What is $\mathrm{inter}(\{\})$?

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 $\mathsf{disjoint}$ 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.

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: $MAXPRICE \in \mathbb{N}$
    axm2: $PRICE = 0 \mathrel{..} MAXPRICE$
    axm3: $SHELF = PRODUCT \nrightarrow \mathbb{N}_1$
    axm4: $partition(PRODUCT, (Milk), (Cheese), (Cereal)$
    axm5: $BASKET = PRODUCT \nrightarrow \mathbb{N}_1$
END

Explain the sets and constants you see in the above machine.

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
INVARIANTS

inv1:	$shelf \in SHELF$
inv2:	$trolleys \subseteq TROLLEY$
inv3:	$products \subseteq PRODUCT$
inv4:	$price \in products \rightarrow PRICE$
inv5:	$products = dom(price)$
inv6:	$dom(shelf) = products$
inv7:	$customers \in trolleys \rightarrow BASKET$
inv8:	$\forall t \cdot t \in dom(customers)$ $\Rightarrow dom(customers(t)) \subseteq products$
inv9:	$reorderlevel \in products \nrightarrow \mathbb{N}_1$
inv10:	$reorder \subseteq products$
inv11:	$topay \in trolleys \nrightarrow \mathbb{N}$
inv12:	$stock \in products \rightarrow \mathbb{N}$

EVENTS

INITIALISATION $\widehat{=}$

. . .

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.

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: $\mathit{sibling} \in \mathit{people} \leftrightarrow \mathit{people}$

2. the *brother* and *sister* relationships between people;

   brother and *sister* are similar to *sibling*, indeed each is a subset of *sibling*: $\mathit{brother} \subseteq \mathit{sibling}$, $\mathit{sister} \subseteq \mathit{sibling}$.

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 beteeen 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 $a \mapsto b$, where $\mapsto$ is pronounced "maps to".

j) What is $NAMES \times PHONE$?

$$NAMES \times PHONE = \{ \\ Jack \mapsto 123, Jack \mapsto 456, Jack \mapsto 789, \\ Jill \mapsto 123, Jill \mapsto 456, Jill \mapsto 789 \\ \}$$

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 $\mathbb{P}(S)$? $\mathbb{P}(S)$, the *powerset of* S is the set of all subsets of S.

o) Given $\text{card}(S) = N$, what is $\text{card}(\mathbb{P}(S))$? $\text{card}(\mathbb{P}(S)) = 2^N$

p) What is $\text{card}(\mathbb{P}(NAMES \times PHONE))$?
$\text{card}(\mathbb{P}(NAMES \times PHONE)) = 2^{\text{card}(NAMES \times PHONE)} = 2^6 = 64$

q) What does $\text{card}(\mathbb{P}(NAMES \times PHONE))$ 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

   ii) is correct. A partial function may happen to be total. If $X \twoheadrightarrow Y$ 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 $X \rightarrow Y \subseteq X \twoheadrightarrow Y$

   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 (n).

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 $X \leftrightarrow Y = \mathbb{P}(X \times Y)$.

    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 $NAMES \leftrightarrow PHONE$.

$$NAMES \leftrightarrow PHONE = \{ \\ \{\}, \\ \{Jack \mapsto 123\}, \{Jack \mapsto 456\}, \{Jack \mapsto 789\}, \{Jill \mapsto 123\}, \{Jill \mapsto 456\}, \{Jill \mapsto 789\}, \\ \{Jack \mapsto 123, Jack \mapsto 456\}, \{Jack \mapsto 123, Jack \mapsto 789\}, \{Jack \mapsto 456, Jack \mapsto 789\}, \\ \{Jill \mapsto 123, Jill \mapsto 456\}, \{Jill \mapsto 123, Jill \mapsto 789\}, \{Jill \mapsto 456, Jill \mapsto 789\}, \\ \{Jack \mapsto 123, Jill \mapsto 123\}, \{Jack \mapsto 123, Jill \mapsto 456\}, \{Jack \mapsto 123, Jill \mapsto 789\}, \\ \{Jack \mapsto 456, Jill \mapsto 123\}, \{Jack \mapsto 456, Jill \mapsto 456\}, \{Jack \mapsto 456, Jill \mapsto 789\},$$

$\{Jack \mapsto 789, Jill \mapsto 123\}, \{Jack \mapsto 789, Jill \mapsto 456\}, \{Jack \mapsto 789, Jill \mapsto 789\},$
$\{Jack \mapsto 123, Jack \mapsto 456, Jack \mapsto 789\},$
$\{Jack \mapsto 123, Jack \mapsto 456, Jill \mapsto 123\}, \{Jack \mapsto 123, Jack \mapsto 789, Jill \mapsto 123\},$
$\{Jack \mapsto 456, Jack \mapsto 789, Jill \mapsto 123\}, \{Jack \mapsto 123, Jack \mapsto 456, Jill \mapsto 456\},$
$\{Jack \mapsto 123, Jack \mapsto 789, Jill \mapsto 456\}, \{Jack \mapsto 456, Jack \mapsto 789, Jill \mapsto 456\},$
$\{Jack \mapsto 123, Jack \mapsto 456, Jill \mapsto 789\}, \{Jack \mapsto 123, Jack \mapsto 789, Jill \mapsto 789\},$
$\{Jack \mapsto 456, Jack \mapsto 789, Jill \mapsto 789\}, \{Jill \mapsto 123, Jill \mapsto 456, Jack \mapsto 123\},$
$\{Jill \mapsto 123, Jill \mapsto 789, Jack \mapsto 123\}, \{Jill \mapsto 456, Jill \mapsto 789, Jack \mapsto 123\},$
$\{Jill \mapsto 123, Jill \mapsto 456, Jack \mapsto 456\}, \{Jill \mapsto 123, Jill \mapsto 789, Jack \mapsto 456\},$
$\{Jill \mapsto 456, Jill \mapsto 789, Jack \mapsto 456\}, \{Jill \mapsto 123, Jill \mapsto 456, Jack \mapsto 789\},$
$\{Jill \mapsto 123, Jill \mapsto 789, Jack \mapsto 789\}, \{Jill \mapsto 456, Jill \mapsto 789, Jack \mapsto 789\},$
$\{Jill \mapsto 123, Jill \mapsto 456, Jill \mapsto 789\},$
$\{Jack \mapsto 123, Jack \mapsto 456, Jack \mapsto 789, Jill \mapsto 123\},$
$\{Jack \mapsto 123, Jack \mapsto 456, Jack \mapsto 789, Jill \mapsto 456\},$
$\{Jack \mapsto 123, Jack \mapsto 456, Jack \mapsto 789, Jill \mapsto 789\},$
$\{Jack \mapsto 123, Jack \mapsto 456, Jill \mapsto 123, Jill \mapsto 456\},$
$\{Jack \mapsto 123, Jack \mapsto 456, Jill \mapsto 123, Jill \mapsto 789\},$
$\{Jack \mapsto 123, Jack \mapsto 456, Jill \mapsto 456, Jill \mapsto 789\},$
$\{Jack \mapsto 123, Jack \mapsto 789, Jill \mapsto 123, Jill \mapsto 456\},$
$\{Jack \mapsto 123, Jack \mapsto 789, Jill \mapsto 123, Jill \mapsto 789\},$
$\{Jack \mapsto 123, Jack \mapsto 789, Jill \mapsto 456, Jill \mapsto 789\},$
$\{Jack \mapsto 456, Jack \mapsto 789, Jill \mapsto 123, Jill \mapsto 456\},$
$\{Jack \mapsto 456, Jack \mapsto 789, Jill \mapsto 123, Jill \mapsto 789\},$
$\{Jack \mapsto 456, Jack \mapsto 789, Jill \mapsto 456, Jill \mapsto 789\},$
$\{Jill \mapsto 123, Jill \mapsto 456, Jill \mapsto 789, Jack \mapsto 123\},$
$\{Jill \mapsto 123, Jill \mapsto 456, Jill \mapsto 789, Jack \mapsto 456\},$
$\{Jill \mapsto 123, Jill \mapsto 456, Jill \mapsto 789, Jack \mapsto 789\},$
$\{Jack \mapsto 123, Jack \mapsto 456, Jack \mapsto 789, Jill \mapsto 123, Jill \mapsto 456\},$
$\{Jack \mapsto 123, Jack \mapsto 456, Jack \mapsto 789, Jill \mapsto 123, Jill \mapsto 789\},$
$\{Jack \mapsto 123, Jack \mapsto 456, Jack \mapsto 789, Jill \mapsto 456, Jill \mapsto 789\},$
$\{Jack \mapsto 123, Jack \mapsto 456, Jill \mapsto 123, Jill \mapsto 456, Jill \mapsto 789, \},$
$\{Jack \mapsto 123, Jack \mapsto 789, Jill \mapsto 123, Jill \mapsto 456, Jill \mapsto 789, \},$
$\{Jack \mapsto 456, Jack \mapsto 789, Jill \mapsto 123, Jill \mapsto 456, Jill \mapsto 789, \},$
$\{Jack \mapsto 123, Jack \mapsto 456, Jack \mapsto 789, Jill \mapsto 123, Jill \mapsto 456, Jill \mapsto 789\}$
$\}$

c) What is card$(NAMES \leftrightarrow PHONE)$? card$(NAMES \leftrightarrow PHONE) =$ card$(\mathbb{P}(NAMES \times PHONE)) = 64$

12. $X \nrightarrow Y$ is the set of *all partial* functions formed from X and Y.

a) Give $NAMES \nrightarrow PHONE$.

$NAMES \nrightarrow PHONE = \{$

$$
\begin{aligned}
&\{\},\\
&\{Jack \mapsto 123\}, \{Jack \mapsto 456\}, \{Jack \mapsto 789\}, \{Jill \mapsto 123\}, \{Jill \mapsto 456\}, \{Jill \mapsto 789\},\\
&\{Jack \mapsto 123, Jill \mapsto 123\}, \{Jack \mapsto 123, Jill \mapsto 456\}, \{Jack \mapsto 123, Jill \mapsto 789\},\\
&\{Jack \mapsto 456, Jill \mapsto 123\}, \{Jack \mapsto 456, Jill \mapsto 456\}, \{Jack \mapsto 456, Jill \mapsto 789\},\\
&\{Jack \mapsto 789, Jill \mapsto 123\}, \{Jack \mapsto 789, Jill \mapsto 456\}, \{Jack \mapsto 789, Jill \mapsto 789\}\\
&\}
\end{aligned}
$$

b) What is card$(NAMES \rightarrow\!\!\!\!\!\!+ \; PHONE)$? 16

13. $X \rightarrow Y$ is the set of *all total* functions formed from X and Y.

a) Give $NAMES \rightarrow PHONE$.

$$
\begin{aligned}
&NAMES \rightarrow PHONE = \{\\
&\quad \{Jack \mapsto 123, Jill \mapsto 123\}, \{Jack \mapsto 123, Jill \mapsto 456\}, \{Jack \mapsto 123, Jill \mapsto 789\},\\
&\quad \{Jack \mapsto 456, Jill \mapsto 123\}, \{Jack \mapsto 456, Jill \mapsto 456\}, \{Jack \mapsto 456, Jill \mapsto 789\},\\
&\quad \{Jack \mapsto 789, Jill \mapsto 123\}, \{Jack \mapsto 789, Jill \mapsto 456\}, \{Jack \mapsto 789, Jill \mapsto 789\}\\
&\quad \}
\end{aligned}
$$

b) What is card$(NAMES \rightarrow PHONE)$? $9 = 3 \times 3$

14. $X \rightarrowtail\!\!\!\!\!\!+ \; Y$ is the set of *all partial, injective* functions formed from X and Y.

a) Give $NAMES \rightarrowtail\!\!\!\!\!\!+ \; PHONE$.

$$
\begin{aligned}
&NAMES \rightarrowtail\!\!\!\!\!\!+ \; PHONE = \{\\
&\quad \{\},\\
&\quad \{Jack \mapsto 123\}, \{Jack \mapsto 456\}, \{Jack \mapsto 789\}, \{Jill \mapsto 123\}, \{Jill \mapsto 456\}, \{Jill \mapsto 789\},\\
&\quad \{Jack \mapsto 123, Jill \mapsto 456\}, \{Jack \mapsto 123, Jill \mapsto 789\},\\
&\quad \{Jack \mapsto 456, Jill \mapsto 123\}, \{Jack \mapsto 456, Jill \mapsto 789\},\\
&\quad \{Jack \mapsto 789, Jill \mapsto 123\}, \{Jack \mapsto 789, Jill \mapsto 456\}\\
&\quad \}
\end{aligned}
$$

b) What is card$(NAMES \rightarrowtail\!\!\!\!\!\!+ \; PHONE)$? $13 = 6 + 6 + 1$

15. $X \rightarrowtail Y$ is the set of *all total, injective* functions formed from X and Y.

a) Give $NAMES \rightarrowtail PHONE$.

$$
\begin{aligned}
&NAMES \rightarrowtail PHONE = \{\\
&\quad \{Jack \mapsto 123, Jill \mapsto 456\}, \{Jack \mapsto 123, Jill \mapsto 789\},\\
&\quad \{Jack \mapsto 456, Jill \mapsto 123\}, \{Jack \mapsto 456, Jill \mapsto 789\},\\
&\quad \{Jack \mapsto 789, Jill \mapsto 123\}, \{Jack \mapsto 789, Jill \mapsto 456\}\\
&\quad \}
\end{aligned}
$$

b) What is card$(NAMES \rightarrowtail PHONE)$? 6

16. $X \twoheadrightarrow\!\!\!\!\!\!+ \; Y$ is the set of *all partial, surjective* functions formed from X and Y.

a) Give $NAMES \mathrel{+\mkern-14mu\twoheadrightarrow} PHONE$.
$NAMES \mathrel{+\mkern-14mu\twoheadrightarrow} PHONE = \{\}$

b) What is $\mathrm{card}(NAMES \mathrel{+\mkern-14mu\twoheadrightarrow} PHONE)$? 0

17. $X \twoheadrightarrow Y$ is the set of *all total, surjective* functions formed from X and Y.

a) Give $NAMES \twoheadrightarrow PHONE$.
$NAMES \twoheadrightarrow PHONE = \{\}$

b) What is $\mathrm{card}(NAMES \twoheadrightarrow PHONE)$? 0

18. $X \mathrel{\rightarrowtail\mkern-18mu\twoheadrightarrow} Y$ is the set of *all (total) bijective* functions formed from X and Y.

a) Give $NAMES \mathrel{\rightarrowtail\mkern-18mu\twoheadrightarrow} PHONE$.
$NAMES \mathrel{\rightarrowtail\mkern-18mu\twoheadrightarrow} PHONE = \{\}$

b) What is $\mathrm{card}(NAMES \mathrel{\rightarrowtail\mkern-18mu\twoheadrightarrow} PHONE)$? 0

c) Why is a partial bijection unnecessary? A partial bijection $X \mathrel{+\mkern-14mu\rightarrowtail\mkern-18mu\twoheadrightarrow} Y$ can be represented by a total injections $Y \rightarrowtail X$.

19. This set exercises some important relational operators.

Given $phone = \{Jack \mapsto 123, Jack \mapsto 789, Jill \mapsto 456, Jill \mapsto 789\}$

a) What is $\mathrm{dom}(phone)$? $\{Jack, Jill\}$.

b) What is $\mathrm{ran}(phone)$? $\{123, 456, 789\}$.

c) What is $phone \oplus \{Jack \mapsto 123\}$? $\{Jack \mapsto 123, Jill \mapsto 456, Jill \mapsto 789\}$.

d) What is $\{Jack\} \lhd phone$? $\{Jack \mapsto 123, Jack \mapsto 789\}$.

e) What is $\{Jack\} \mathbin{\lhd\mkern-14mu-} phone$? $\{Jill \mapsto 456, Jill \mapsto 789\}$.

f) What is $phone \rhd \{123, 789\}$? $\{Jack \mapsto 123, Jack \mapsto 789, Jill \mapsto 789\}$.

g) What is $phone \mathbin{\rhd\mkern-14mu-} \{123, 789\}$? $\{Jill \mapsto 456\}$.

h) What is $phone[\{Jack\}]$? $\{123, 789\}$.

20. Students pass a subject if they gain at least 50 marks in the final examination. Given a function $results : \mathsf{STUDENTS} \mathrel{+\mkern-14mu\rightarrow} \mathbb{N}$, that yields the examination result for a particular student, specify

a) the set of students that pass: $\mathrm{dom}(results \rhd \{n \mid n \in \mathbb{N} \land n \geq 50\})$

$$\{s \mid s \in \mathrm{dom}(results) \land results(s) \geq 50\}$$

or

$$\mathrm{dom}(results \rhd \{n \mid n \in \mathbb{N} \land n \geq 50\})$$

b) the set of students that fail.

$$\{s \mid s \in \mathrm{dom}(results) \land results(s) < 50\}$$

21. If we were modelling a taxi fleet company we might have three variables, *drivers*, *taxis* and *assigned* constrained by

$$\begin{aligned} drivers &: \mathbb{P}\,\mathsf{DRIVERS} \\ taxis &: \mathbb{P}\,\mathsf{TAXIS} \\ assigned &: drivers \mathrel{+\mkern-14mu\rightarrowtail} taxis \end{aligned}$$

where *drivers* is the set of drivers working for the company, *taxis* is the set of taxis owned by the company, and *assigned* is a function recording the assignment of drivers to taxis.

The arrow $\rightarrowtail$ 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 *assigned* a function? It is a *function* because a driver would be assigned to at most one taxi.

b) Why is *assigned* a partial function?
It is *partial* because at any time not all drivers would necessarily be assigned to a taxi.

c) Why is *assigned* an injective function?
It is *injective* because a taxi would be assigned to at most one driver.

d) Specify the drivers who are currently assigned.
$\mathrm{dom}(assigned)$

e) Specify the drivers who are currently unassigned.
$drivers - \mathrm{dom}(assigned)$

f) Specify the taxis that are currently assigned.
$\mathrm{ran}(assigned)$

g) Specify the taxis that are currently unassigned.
$taxis - \mathrm{ran}(assigned)$

22. Are the following rules correct or incorrect?

a) $f \in X \to Y \Rightarrow f \in X \nrightarrow Y$
Correct, a total function is a partial function.

b) $f \in X \rightarrowtail\!\!\!\!\!\nrightarrow Y \Rightarrow f \in X \nrightarrow Y$
Correct, a partial injection is a partial function.

c) $f \in X \rightarrowtail\!\!\!\!\!\nrightarrow Y \Rightarrow f \in X \to Y$
Incorrect, a partial injection is not a total function.

d) $f \in X \rightarrowtail Y \Rightarrow f \in X \to Y$
Correct, a total injection is a total function.

e) $f \in X \rightarrowtail Y \Rightarrow f \in X \nrightarrow Y$
Correct, a total injection is a partial function.

f) $f \in X \twoheadrightarrow\!\!\!\!\!\!\nrightarrow Y \Rightarrow f \in X \nrightarrow Y$
Correct, a partial surjection is a partial function.

g) $f \in X \twoheadrightarrow\!\!\!\!\!\!\nrightarrow Y \Rightarrow f \in X \to Y$
Incorrect, a partial surjection is not a total function.

h) $f \in X \twoheadrightarrow Y \Rightarrow f \in X \to Y$
Correct, a total surjection is a total function.

i) $f \in X \twoheadrightarrow Y \Rightarrow f \in X \nrightarrow Y$
Correct, a total surjection is a partial function.

j) $f \in X \nrightarrow Y \Rightarrow \mathrm{dom}(f) \subset X$
Incorrect.

k) $f \in X \to Y \Rightarrow \mathrm{ran}(f) = Y$
Incorrect.

l) $f \in X \nrightarrow Y \wedge x \in \mathrm{dom}(f) \Rightarrow f[\{x\}] = \{f(x)\}$
Correct.

m) $(r^{-1})^{-1} = r$
Correct.

n) $r \in X \leftrightarrow Y \Rightarrow \mathrm{dom}(r^{-1}) = \mathrm{ran}(r)$
Correct.

o) $r \in X \leftrightarrow Y \Rightarrow \mathrm{ran}(r^{-1}) = \mathrm{dom}(r)$
Correct.

p) $r \in X \leftrightarrow Y \Rightarrow \mathrm{ran}(r) \in Y$
Incorrect.

q) $r \in X \leftrightarrow Y \Rightarrow \mathrm{ran}(r) \subseteq Y$
Correct.

r) $r \in X \leftrightarrow Y \Rightarrow \mathrm{ran}(r) \in \mathbb{P}(Y)$
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 union : $\mathbb{P}\,\mathbb{P}\,X \to \mathbb{P}\,X$ that maps a set of subsets of X to its generalized union. What is union $\varnothing$?

    a) $\mathrm{union}(U) = \{\ x : X | \exists u \cdot (u \in U \wedge x \in u)\ \}$

    b) $$\begin{aligned} \mathrm{union}(\varnothing) &= \{\ x \mid x \in X \wedge \exists u \cdot (u \in \varnothing \wedge x \in u)\ \} \\ &= \{\ x \mid x \in X \wedge \text{false}\ \} \\ &= \varnothing \end{aligned}$$

    To shed a bit more light on this, it is clear that $\mathrm{union}(\{x\}) = \{x\}$ for any $x \in X$. But, $\mathrm{union}(\{x\}) = \mathrm{union}(\{x\}, \varnothing) = \mathrm{union}(\{x\}) \cup \mathrm{union}(\varnothing)$. It follows that $\mathrm{union}(\varnothing)$ 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 inter $\mathbb{P}\,\mathbb{P}\,X \to \mathbb{P}\,X$ that maps a set of subsets of X to its generalized intersection. What is inter $\varnothing$?

    a) $\mathrm{inter}\, U = \{\ x \mid x \in X \wedge \forall u \cdot (u \in U \Rightarrow x \in u)\ \}$

    b) $$\begin{aligned} \mathrm{inter}\, \varnothing &= \{\ x : X \mid \forall u \cdot (u \in \varnothing \Rightarrow x \in u)\ \} \\ &= \{\ x : X | \top\ \} \\ &= X \end{aligned}$$

    To shed a bit more light on $\mathrm{inter}(\varnothing)$, it is clear that $\mathrm{inter}(\{\{x\}\}) = \{x\}$. Now consider $\mathrm{inter}(P, \{x\})$, where P is a list of sets. Then, $\mathrm{inter}(P, \{x\}) = \mathrm{inter}(\{P\}) \cap \{x\}$. Now take the case where the list P is empty, $\{x\} = \mathrm{inter}(\varnothing) \cap \{x\}$ for all $x \in X$. Therefore, $\mathrm{inter}(\varnothing)$ 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 disjoint that holds between a pair of subsets of X exactly when they are *disjoint*.

    $\mathsf{disjoint}(u, w) = u \cap w = \varnothing$

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.
    $\mathsf{partition}X = \{\ w \mid w \in \mathbb{P}(\mathbb{P}(X)) \wedge \mathrm{union}(w) = X \wedge \forall(u1, u2) \cdot (u1 \in w \wedge u2 \in w \Rightarrow u1 \neq u2) \wedge \mathsf{disjoint}(u1, u2)\ \}$

    b) Which of the subsets of $\{a, b\}$ are partitions of $\{a, b\}$?
    $\{\{a\}, \{b\}\}$, $\{\varnothing, \{a\}, \{b\}\}$, $\{\{a, b\}\}$, $\{\varnothing, \{a, b\}\}$.

Appendix A

Models

A.1 Coffee Club

MACHINE **CoffeeClub**

VARIABLES

piggybank denotes a supply of money for the coffee club.

INVARIANTS

inv1: $piggybank \in \mathbb{N}$ piggybank must be a natural number, that is, a non-zero integer

EVENTS

Initialisation $\widehat{=}$

THEN

act1: $piggybank := 0$

END

FeedBank $\widehat{=}$

ANY

amount

WHERE

grd1: $amount \in \mathbb{N}1$

THEN

act1: $piggybank := piggybank + amount$

END

RobBank $\widehat{=}$

ANY

amount

WHERE

grd1: $amount : \mathbb{N}1$

grd2: $amount \leq piggybank$ There must be enough in the piggybank

THEN

act1: $piggybank := piggybank - amount$

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:	$piggybank \in \mathbb{N}$	
inv2:	$members \subseteq MEMBER$	each member has unique id
inv3:	$accounts \in members \rightarrow \mathbb{N}$	each member has an account
inv4:	$coffeeprice \in \mathbb{N}1$	price of a cup of coffee

EVENTS
Initialisation : ***extended*** $\widehat{=}$
THEN

act2:	$members := \varnothing$	empty set of members
act3:	$accounts := \varnothing$	empty set of accounts
act4:	$coffeeprice :\in \mathbb{N}1$	initial coffee price set to arbitrary non-zero value

END

SetPrice $\widehat{=}$
ANY
amount
WHERE
grd1: $amount \in \mathbb{N}1$
THEN
act1: $coffeeprice := amount$
END

NewMember $\widehat{=}$
ANY
member

WHERE
grd1: $member \in MEMBER \setminus members$ choose an unused element of MEMBER
THEN
act1: $members := members \cup \{member\}$
act2: $accounts(member) := 0$
END

Contribute $\widehat{=}$
ANY
amount
member
WHERE
grd1: $amount \in \mathbb{N}1$
grd2: $member \in members$
THEN
act1: $accounts(member) := accounts(member) + amount$
act2: $piggybank := piggybank + amount$
END

BuyCoffee $\widehat{=}$
ANY
member
WHERE
grd1: $member \in members$
grd2: $accounts(member) \geq coffeeprice$
THEN
act1: $accounts(member) := accounts(member) - coffeeprice$
END

FeedBank : ***extended*** $\widehat{=}$
REFINES
FeedBank
ANY

WHERE

THEN

END

RobBank : ***extended*** $\widehat{=}$
REFINES
RobBank
ANY

WHERE

THEN

END

END **MemberShip**

A.2 SquareRoot

CONTEXT **SquareRoot_ctx**
CONSTANTS
  num
AXIOMS
  axm1: $num \in \mathbb{N}$
END

MACHINE **SquareRoot**
SEES
  SquareRoot_ctx
VARIABLES
  sqrt
INVARIANTS
  inv1: $sqrt \in \mathbb{N}$
EVENTS
Initialisation $\widehat{=}$
THEN
  act1: $sqrt :\in \mathbb{N}$
END

SquareRoot $\widehat{=}$
THEN
  act1: $sqrt :| \; (sqrt' \in \mathbb{N} \\ \wedge sqrt' * sqrt' \leq num \\ \wedge num < (sqrt' + 1) * (sqrt' + 1))$
END

END **SquareRoot**

MACHINE **SquareRootR1**
REFINES
  SquareRoot
SEES
  SquareRoot_ctx
VARIABLES
  sgrt
  low
  high
INVARIANTS
  inv1: $low \in \mathbb{N}$
  inv2: $high \in \mathbb{N}$
  inv3: $low + 1 \leq high$
  inv4: $low * low \leq num$
  inv5: $low < high * high$
VARIANT
$high - low$
EVENTS
Initialisation $\widehat{=}$
THEN

act1: $sqrt :\in \mathbb{N}$
act2: $low :| \; low' \in \mathbb{N} \wedge low' * low' \leq num$
act3: $high :| \; high' \in \mathbb{N} \wedge num < high' * high'$
END

SquareRoot $\widehat{=}$
REFINES
SquareRoot
WHERE
grd1: $low + 1 = high$
THEN
act1: $sqrt := low$
END

Improve $\widehat{=}$
STATUS convergent
ANY
l
h
WHERE
grd1: $low + 1 \neq high$
grd2: $l \in \mathbb{N} \wedge low \leq l \wedge l * l \leq num$
grd3: $h \in \mathbb{N} \wedge h \leq high \wedge num < h * h$
grd4: $l + 1 \leq h$
grd5: $h - 1 < high - low$
THEN
act1: $low, high := l, h$
END

END **SquareRootR1**

MACHINE **SquareRootR2**
REFINES
SquareRootR1
SEES
SquareRoot_ctx
VARIABLES
sgrt
low
high
INVARIANTS
inv1: $low \in \mathbb{N}$
inv2: $high \in \mathbb{N}$
inv3: $low + 1 \leq high$
inv4: $low * low \leq num$
inv5: $low < high * high$
VARIANT
$high - low$
EVENTS
Initialisation : ***extended*** $\widehat{=}$
THEN

END

SquareRoot : *extended* $\widehat{=}$
REFINES
    SquareRoot
WHERE

THEN

END

Improve1 $\widehat{=}$
REFINES
    Improve
ANY
    m
WHERE
    grd1: $low + 1 \neq high$
    grd2: $m \in \mathbb{N}$
    grd3: $low < m \land m < high$
    grd4: $m * m \leq num$ m is a better value for low
WITH
    l: $l = m$
    h: $h = high$
THEN
    act1: $low := m$
END

Improve2 $\widehat{=}$
REFINES
    Improve
ANY
    m
WHERE
    grd1: $low + 1 \neq high$
    grd2: $m \in \mathbb{N}$
    grd3: $low < m \land m < high$
    grd4: $m * m > num$ m is a better value for $high$
WITH
    l: $l = low$
    h: $h = m$
THEN
    act1: $high := m$
END

END **SquareRootR2**

MACHINE **SquareRootR3**
REFINES
    SquareRootR2
SEES
    SquareRoot_ctx
VARIABLES

sqrt
low
high

INVARIANTS

thm1: $(\forall n \cdot n \in \mathbb{N} \Rightarrow (\exists m \cdot m \in \mathbb{N} \land (n = 2 * m \lor n = 2 * m + 1)))$ n is even or odd

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

VARIANT

$high - low$

EVENTS

Initialisation : ***extended*** $\widehat{=}$

THEN

END

SquareRoot : ***extended*** $\widehat{=}$

REFINES

SquareRoot

WHERE

THEN

END

Improve1 $\widehat{=}$

REFINES

Improve

ANY

m

WHERE

grd1: $low + 1 \neq high$

grd2: $m = (low + high)/2$

grd3: $m * m \leq num$ m is a better value for low

THEN

act1: $low := m$

END

Improve2 $\widehat{=}$

REFINES

Improve

ANY

m

WHERE

grd1: $low + 1 \neq high$

grd2: $m = (low + high)/2$

grd3: $(m * m > num$ m is a better value for $high$

THEN

act1: $high := m$

END

END **SquareRootR3**

MACHINE **SquareRootR4**

REFINES
SquareRootR3
SEES
SquareRoot_ctx
VARIABLES
sqrt
low
high
mid
INVARIANTS
inv1: $low \in \mathbb{N}$
inv2: $high \in \mathbb{N}$
inv3: $low + 1 \leq high$
inv4: $low * low \leq num$
inv5: $num < high * high$
inv6: $mid = (low + high)/2$
VARIANT
$high - low$
EVENTS
Initialisation $\widehat{=}$
THEN
act1: $sqrt :\in \mathbb{N}$
act2: $low := 0$
act3: $high := num + 1$
act4: $mid := (num + 1)/2$
END

SquareRoot $\widehat{=}$
REFINES
SquareRoot
WHERE
grd1: $low + 1 = high$
THEN
act1: $sqrt := low$
END

Improve1 $\widehat{=}$
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 $\widehat{=}$
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**

Appendix B

Proof Obligations

All proof obligations have a name and an abbreviation:

Id	Name	Needed to discharge
INV	Invariant	
FIS	Feasibility	
WD	Well-definedness	
GRD	Guard	
EQL	Equal	

Appendix C

Event-B Concise Summary

Each construct will be given in its presentation form, as displayed in the Rodin toolkit, followed by the $\boxed{\text{ASCII form}}$ 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 $\mapsto$ 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, *ie* it matches e_1, e_2 and also $e_1, e_2, e_3 \ldots$; similarly for x, y;

Freeness: The meta-predicate $\neg \mathit{free}(z, E)$ 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: $\neg \mathit{free}(z, \forall z \cdot P \Rightarrow Q)$, $\neg \mathit{free}(z, \exists z \cdot P \wedge Q)$, $\neg \mathit{free}(z, \{z \cdot P \mid F\})$, $\neg \mathit{free}(z, \lambda z \cdot P|E)$, and $\mathit{free}(z, z)$.

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:$\neg \mathit{free}(z, (\forall z \cdot P)$, $\neg \mathit{free}(z, (\exists z \cdot P)$, $\neg \mathit{free}(z, \{z \cdot P \mid F\})$, $\neg \mathit{free}(z, \lambda z \cdot P|E)$, and $\mathit{free}(z, z)$

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 : $\bot$ `false`
2. True : $\top$ `true`
3. Conjunction : $P \wedge Q$ `P & Q`
   Left associative.
4. Disjunction : $P \vee Q$ `P or Q`
   Left associative.
5. Implication : $P \Rightarrow Q$ `P => Q`
   Non-associative: this means that $P \Rightarrow Q \Rightarrow R$ must be parenthesised or an error will be diagnosed.
6. Equivalence : $P \Leftrightarrow Q$ `P <=> Q`.
   $P \iff Q = P \Rightarrow Q \wedge Q \Rightarrow P$
   Non-associative: this means that $P \Leftrightarrow Q \Leftrightarrow R$ must be parenthesised or an error will be diagnosed.
7. Negation : $\neg P$ `not P`
8. Universal quantification :
   $(\forall z \cdot P \Rightarrow Q)$ `(!z.P => Q)`
   Strictly, $\forall z \cdot P$, but usually an implication.
   For all values of z, satisfying P, Q is satisfied.
   The types of z must be inferrable from the predicate P.
9. Existential quantification :
   $(\exists z \cdot P \wedge Q)$ `(#z.P & Q)`
   Strictly, $\exists z \cdot P$, but usually a conjunction.
   There exist values of z, satisfying P, that satisfy Q.
   The predicate P must be inferrable from the predicate P.
10. Equality : $E = F$ `E = F`
11. Inequality : $E \neq F$ `E /= F`

C.2 Sets

1. Singleton set : $\{E\}$ `{E}`
2. Set enumeration : $\{E, F\}$ `{E, F}`
   See note on the pattern E, F at top of summary.
3. Empty set : $\varnothing$ `{}`
4. Set comprehension : $\{\ z \cdot P \mid F\ \}$ `{ z . P | F }`
   General form: the set of all values of F for all values of z that satisfy the *predicate* P. P must *constrain* the variables in z.
5. Set comprehension : $\{\ F \mid P\ \}$ `{ F | P }`
   Special form: the set of all values of F that satisfy the *predicate* P. In this case the set of bound variables z are all the free variables in F.
   $\{\ F \mid P\ \} = \{\ z \cdot P \mid F\ \}$, where z is all the variables in F.
6. Set comprehension : $\{\ x \mid P\ \}$ `{ x | P }`
   A special case of item 5: the set of all values of x that satisfy the *predicate* P.
   $\{\ x \mid P\ \} = \{\ x \cdot P \mid x\ \}$
7. Union : $S \cup T$ `S \/ T`
8. Intersection : $S \cap T$ `S /\ T`
9. Difference : $S \setminus T$ `S \ T`
   $S \setminus T = \{x \mid x \in S \wedge x \notin T\}$
10. Ordered pair : $E \mapsto F$ `E |-> F`.
    $E \mapsto F \neq (E, F)$
    Left associative.
    In all places where an ordered pair is required, $E \mapsto F$ must be used. E, F will not be accepted as an ordered pair, it is always a list. $\{x, y \cdot P \mid x \mapsto y\}$ illustrates the different usage.
11. Cartesian product : $S \times T$ `S ** T`
    $S \times T = \{x \mapsto y \mid x \in S \wedge y \in T\}$
    Left-associative.
12. Powerset : $\mathbb{P}(S)$ `POW(S)`
    $\mathbb{P}(S) = \{s \mid s \subseteq S\}$
13. Non-empty subsets : $\mathbb{P}_1(S)$ `POW1(S)`
    $\mathbb{P}_1(S) = \mathbb{P}(S) \setminus \{\varnothing\}$
14. Cardinality : $\mathrm{card}(S)$ `card(S)`
    Defined only for *finite*(S).
15. Generalized union : $\mathrm{union}(U)$ `union(U)`
    The union of all the elements of U.
    $\forall U \cdot U \in \mathbb{P}(\mathbb{P}(S)) \Rightarrow$
    $\mathrm{union}(U) = \{x \mid x \in S \wedge \exists s \cdot s \in U \wedge x \in s\}$
    where $\neg \mathit{free}(x, s, U)$

16. Generalized intersection : inter(U) `inter(U)`
The intersection of all the elements of U.
$U \neq \varnothing$,
$\forall U \cdot U \in \mathbb{P}(\mathbb{P}(S)) \Rightarrow$
$\text{inter}(U) = \{x \mid x \in S \wedge \forall s \cdot s \in U \Rightarrow x \in s\}$
where $\neg \mathit{free}(x, s, U)$

17. Quantified union : `UNION z.P | S`
$\cup z \cdot P \mid S$
P must *constrain* the variables in z.
$\forall z \cdot P \Rightarrow S \subseteq T \Rightarrow$
$\cup(z \cdot P \mid E) = \{x \mid x \in T \wedge \exists z \cdot P \wedge x \in S\}$
where $\neg \mathit{free}(x, z, T)$, $\neg \mathit{free}(x, P)$, $\neg \mathit{free}(x, S)$, $\neg \mathit{free}(x, z)$

18. Quantified intersection : `INTER z.P | S`
$\cap z \cdot P \mid S$
P must *constrain* the variables in z,
$\{z \mid P\} \neq \varnothing$,
$(\forall z \cdot (P \Rightarrow S \subseteq T)) \Rightarrow$
$\cap z \cdot P \mid S = \{x \mid x \in T \wedge (\forall z \cdot P \Rightarrow x \in S)\}$
where $\neg \mathit{free}(x, z)$, $\neg \mathit{free}(x, T)$, $\neg \mathit{free}(x, P)$, $\neg \mathit{free}(x, S)$.

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 : $\mathit{finite}(S)$ `finite(S)`
$\mathit{finite}(S) \Leftrightarrow$ *S is finite.*
8. Partition : $\mathit{partition}(S, x, y)$ `partition(S,x,y)`
x and y partition the set S, *ie* $S = x \cup y \wedge x \cap y = \varnothing$

   Specialised use for enumerated sets: $\mathit{partition}(S, \{A\}, \{B\}, \{C\})$.
$S = \{A, B, C\} \wedge A \neq B \wedge B \neq C \wedge C \neq A$

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: bool(P) = TRUE
2. $\neg P$ is provable: bool(P) = 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 $\mathbb{Z}$ `INT`
2. The set of natural numbers $\mathbb{N}$ `NAT`
3. The set of positive natural numbers $\mathbb{N}_1$ `NAT1`
$\mathbb{N}_1 = \mathbb{N} \setminus \{0\}$
4. Minimum min(S) `min(S)`
$S \subset \mathbb{Z}$ and $\mathit{finite}(S)$ or S must have a lower bound.
5. Maximum max(S) `max(S)`
$S \subset \mathbb{Z}$ and $\mathit{finite}(S)$ or S must have an upper bound.
6. Sum $m + n$ `m + n`
7. Difference $m - n$ `m - n`
$n \leq m$
8. Product $m \times n$ `m * n`
9. Quotient m/n `m / n`
$n \neq 0$
10. Remainder $m \mathbin{\mathit{mod}} n$ `m mod n`
$n \neq 0$
11. Interval $m \mathbin{..} n$ `m .. n`
$m \mathbin{..} n = \{\ i \mid m \leq i \wedge i \leq n\ \}$

Number predicates

1. Greater $m > n$ `m > n`
2. Less $m < n$ `m < n`
3. Greater or equal $m \geq n$ `m >= n`
4. Less or equal $m \leq n$ `m <= n`

C.5 Relations

A relation is a set of ordered pairs; a many to many mapping.

1. Relations $S \leftrightarrow T$ `S <-> T`
   $S \leftrightarrow T = \mathbb{P}(S \times T)$
   Associativity: relations are *right associative*:
   $r \in X \leftrightarrow Y \leftrightarrow Z = r \in X \leftrightarrow (Y \leftrightarrow Z)$.

2. Domain $\mathrm{dom}(r)$ `dom(r)`
   $\forall r \cdot r \in S \leftrightarrow T \Rightarrow$
   $\mathrm{dom}(r) = \{x \cdot (\exists y \cdot x \mapsto y \in r)\}$

3. Range $\mathrm{ran}(r)$ `ran(r)`
   $\forall r \cdot r \in S \leftrightarrow T \Rightarrow$
   $\mathrm{ran}(r) = \{y \cdot (\exists x \cdot x \mapsto y \in r)\}$

4. Total relation $S \leftrightarrow\!\!\!\!\!\leftarrow T$ `S <<-> T`
   if $r \in S \leftrightarrow\!\!\!\!\!\leftarrow T$ then $\mathrm{dom}(r) = S$

5. Surjective relation $S \leftrightarrow\!\!\!\!\!\rightarrow T$ `S <->> T`
   if $r \in S \leftrightarrow\!\!\!\!\!\rightarrow T$ then $\mathrm{ran}(r) = T$

6. Total surjective relation $S \twoheadleftarrow\!\!\!\!\!\twoheadrightarrow T$ `S <<->> T`
   if $r \in S \twoheadleftarrow\!\!\!\!\!\twoheadrightarrow T$ then $\mathrm{dom}(r) = S$ and $\mathrm{ran}(r) = T$

7. Forward composition $p \,;\, q$ `p ; q`
   $\forall p, q \cdot p \in S \leftrightarrow T \wedge q \in T \leftrightarrow U \Rightarrow$
   $p \,;\, q = \{x \mapsto y \mid (\exists z \cdot x \mapsto z \in p \wedge z \mapsto y \in q)\}$

8. Backward composition $p \circ q$ `p circ q`
   $p \circ q = q \,;\, p$

9. Identity id `id`
   $S \lhd \mathrm{id} = \{x \mapsto x \mid x \in S\}$.
   id is generic and the set S is inferred from the context.

10. Domain restriction $S \lhd r$ `S <| r`
    $S \lhd r = \{x \mapsto y \mid x \mapsto y \in r \wedge x \in S\}$.

11. Domain subtraction $S \ntriangleleft r$ `S <<| r`
    $S \ntriangleleft r = \{x \mapsto y \mid x \mapsto y \in r \wedge x \notin S\}$.

12. Range restriction $r \rhd T$ `r |> T`
    $r \rhd T = \{x \mapsto y \mid x \mapsto y \in r \wedge y \in T\}$.

13. Range subtraction $r \ntriangleright T$ `r |>> T`
    $r \ntriangleright T = \{x \mapsto y \mid y \in r \wedge y \notin T\}$.

14. Inverse r^{-1} `r~`
    $r^{-1} = \{y \mapsto x \mid x \mapsto y \in r\}$.

15. Relational image $r[S]$ `r[S]`
    $r[S] = \{y \mid \exists x \cdot x \in S \wedge x \mapsto y \in r\}$.

16. Overriding $r_1 \Leftarrow r_2$ `r1 <+ r2`
    $r_1 \Leftarrow r_2 = r_2 \cup (\mathrm{dom}(r_2) \ntriangleleft r_1)$.

17. Direct product $p \otimes q$ `p >< q`
    $p \otimes q = \{x \mapsto (y \mapsto z) \mid x \mapsto y \in p \wedge x \mapsto z \in q)\}$.

18. Parallel product $p \parallel q$ `p || q`
    $p \parallel q = \{x, y, m, n \cdot x \mapsto m \in p \wedge y \mapsto n \in q \mid (x \mapsto y) \mapsto (m \mapsto n)\}$.

19. Projection prj_1 `prj1`
    prj_1 is generic.
    $(S \times T) \lhd \mathrm{prj}_1 = \{(x \mapsto y) \mapsto x \mid x \mapsto y \in S \times T\}$.

20. Projection prj_2 `prj2`
    prj_2 is generic.
    $(S \times T) \lhd \mathrm{prj}_2 = \{(x \mapsto y) \mapsto y \mid x \mapsto y \in S \times T\}$.

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 □
   $r \in S \leftrightarrow S \Rightarrow r^0 = S \lhd \mathrm{id} \wedge r^{n+1} = r \,;\, r^n$.
   Note: to avoid inconsistency S should be the finite *base* set for r, ie the smallest set for which all $r \in S \leftrightarrow S$.
   Could be defined as a function $iterate(r \mapsto n)$.

2. Reflexive Closure r^* □
   $r^* = \cup n \cdot n \in \mathbb{N} \mid r^n$.
   Could be defined as a function $rclosure(r)$.
   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 $iclosure(r)$.
   Note: $r^0 \not\subseteq r^+$ by default, but may be present depending on r.

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 $S \nrightarrow T$ `S +-> T`
   $S \nrightarrow T = \{r \cdot r \in S \leftrightarrow T \wedge r^{-1} \,;\, r \subseteq T \lhd \mathrm{id}\}$.

2. Total functions $S \to T$ `S --> T`
   $S \to T = \{f \cdot f \in S \nrightarrow T \wedge \mathrm{dom}(f) = S\}$.

3. Partial injections $S \rightarrowtail\!\!\!\!\!\!\!+\; T$ `S >+> T`
   $S \rightarrowtail\!\!\!\!\!\!\!+\; T = \{f \cdot f \in S \nrightarrow T \wedge f^{-1} \in T \nrightarrow S\}$.
   One-to-one relations.

4. Total injections $S \rightarrowtail T$ `S >-> T`
   $S \rightarrowtail T = S \rightarrowtail\!\!\!\!\!\!\!+\; T \cap S \to T$.

5. Partial surjections $S \twoheadrightarrow\!\!\!\!\!\!\!+\; T$ `S +->> T`
   $S \twoheadrightarrow\!\!\!\!\!\!\!+\; T = \{f \cdot f \in S \nrightarrow T \wedge \mathrm{ran}(f) = T\}$.
   Onto relations.

6. Total surjections $S \twoheadrightarrow T$ `S ->> T`
   $S \twoheadrightarrow T = S \twoheadrightarrow\!\!\!\!\!\!\!+\; T \cap S \to T$.

7. Bijections $S \rightarrowtail\!\!\!\!\!\twoheadrightarrow T$ `S >->> T`
   $S \rightarrowtail\!\!\!\!\!\twoheadrightarrow T = S \rightarrowtail T \cap S \twoheadrightarrow T$.
   One-to-one and onto relations.

8. Lambda abstraction
   $(\lambda p \cdot P \mid E)$ `(%p.P|E)`
   P must *constrain* the variables in p.
   $(\lambda p \cdot P \mid E) = \{z \cdot P \mid p \mapsto E\}$, where z is a list of variables that appear in the pattern p.

9. Function application $f(E)$ `f(E)`
   $E \mapsto y \in f \Rightarrow E \in \mathrm{dom}(f) \wedge f \in X \nrightarrow Y$, where $type(f) = \mathbb{P}(X \times Y$.
   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 |-> F)`
   $f(E, F)$ 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	

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.

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 $x := E$ `x := E`
   $:=$ = *"becomes equal to"*: replace free occurrences of x by E.

3. Choice from set $x :\in S$ `x :: S`
   $:\in$ = *"becomes in"*: arbitrarily choose a value from the set S.

4. Choice by predicate $z :| P$ `z :| P`
   $:|$ = *"becomes such that"*: arbitrarily choose values for the variable in z 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:
   $f(x) := E \;=\; f := f \lhd\!\!\!- \{x \mapsto E\}$.

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

[1] J.-R. Abrial. *The B-Book: Assigning Programs to Meanings.* Cambridge University Press, 1996.

[2] J.-R. Abrial. *Modelling in Event-B: System and Software Engineering.* Cambridge University Press, 2010.

[3] R.-J. Back. A calculus of refinement for program derivations. Technical Report Report Ser. A 54, Departments of Information Processing and Mathematics, Swedish University of Åbo, Åbo, Finland, 1987.

[4] R.-J. Back. Procedural abstraction in the refinement calculus. Technical Report Report Ser. A 55, Departments of Information Processing and Mathematics, Swedish University of Åbo, 1987.

[5] J. W. Backus, F. L. Bauer, J. Green, C. Katz, J. McCarthy, A. J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J. H. Wegstein, A. van Wijngaarden, and M. Woodger. Revised report on the algorithm language algol 60. *Commun. ACM*, 6(1):1–17, 1963.

[6] Edsgar W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.

[7] Event-B and Rodin Platform. `http://wiki.event-b.org/`.

[8] Event-B and Rodin Documentation Wiki. `http://wiki.event-b.org/index.php/Main_Page`.

[9] Robert Floyd. Assigning meaning to programs. In *Proceedings of Symposium on Applied Mathematics*, pages 19–21, 1967.

[10] David Gries. *A Science of Programming.* Springer-Verlag, 1981.

[11] I.J. Hayes, editor. *Specification Case Studies.* Prentice-Hall International, 1987.

[12] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, October 1969.

[13] C. B. Jones. *Software Development: A Rigorous Approach.* Prentice Hall International, 1980.

[14] C. B. Jones. *Systematic Software Development using VDM.* Prentice Hall International, 2nd edition, 1990. ISBN 0-13-880733-7.

[15] Carroll Morgan. *Programming from Specifications.* International Series in Computer Science. Prentice-Hall, 1990.

[16] Deploy Project. Deliverables. `http://www.deploy-project.eu/html/deliverables.html`.

[17] J.M. Spivey. *The Z Notation: A Reference Manual.* Prentice-Hall International, 1989.

[18] J.M. Spivey. *The Z Notation: A Reference Manual.* Prentice-Hall International, 2nd edition, 1992.

[19] Wikipedia. `http://en.wikipedia.org/wiki/B-Method`.

[20] Wikipedia. http://en.wikipedia.org/wiki/Jean-Raymond_Abrial.

[21] Wikipedia. http://en.wikipedia.org/wiki/Paternoster.

[22] Wikipedia. http://en.wikipedia.org/wiki/Sequent.

[23] Wikipedia. http://en.wikipedia.org/wiki/Z_notation.

[24] Niklaus Wirth. Program development by stepwise refinement. *Commun. ACM*, 26(1):70–74, 1983.

Index