Reasoned Modelling with Event-B

Michael Butler
30 January 2017

University of Southampton

Abstract. This paper provides an overview of how the Event-B language and verification method can be used to model and reason about system behaviour. Formal modelling and reasoning help to increase understanding and reduce defects in requirements specification. Sets and relations play a key role in modelling as do operators on these structures. Precise definitions and rules are provided in order to help the reader gain a strong understanding of the mathematical operators for sets and relations. While the emphasis is on mathematical reasoning, particularly through invariant proofs, the paper also covers less formal reasoning such as identification of problem entities supported by class diagrams and validation of formal models against informal requirements. The use of tools for animation, model checking and proof is also outlined.

1 Introduction

This paper provides an introduction to formal modelling using the Event-B language and method [1]. We make no strong assumptions about the existing knowledge of the reader other than in interest in learning about the approach and a willingness to start to put it into practice.

It is useful to motivate the role and value of the formal methods that we are outlining and advocating in this paper. Essentially it is about improving the processes that are used to engineer software-based systems so that specification and design errors are identified and rectified as soon as possible in the system development cycle. From the earliest days of software engineering it has been recognised that the cost of fixing a specification or design error is higher the later in the development that error is identified. This is summarised by the following observation about software development by Boehm [2]:

> **Boehm's First Law:** *Errors are more frequent during requirements and design activities and are more expensive the later they are removed.*

This observation is bourne out by many studies of software engineering projects. For example, a 2013 report from the Carnegie-Mellon Software Engineering Institute (SEI) highlights studies showing that requirements and architecture defects make up approximately 70% of all system defects and that 80% of these defects are discovered late in the development life cycle [3].

Early identification of errors through formal modelling

Clearly, identifying errors at the point at which they have become expensive to fix, long after they were introduced, is undesirable. More desirable would be to discover errors as soon as possible when they are less expensive to fix. So, why is it difficult to achieve this ideal profile in practice? Common errors introduced in the early stages of development are errors in understanding the system requirements and errors in writing the system specification. Without a rigorous approach to understanding requirements and constructing specifications, it can be very difficult to uncover such errors other than through testing of the software product after a lot of development has already been undertaken. Why is it difficult to identify errors that are introduced early in the development cycle? One reason is lack of precision in formulating specifications resulting in ambiguities and inconsistencies that are difficult to detect and may store up problems for later. Another reason is too much complexity, whether it is complexity of requirements, complexity of the operating environment of a system or complexity of the design of a system.

To overcome the problem of lack of precision, we advocate the the use of *formal modelling*. As well as encouraging precise descriptions, formal modelling languages are supported by verification methods that support the discovery and elimination of inconsistencies in models. But precision on its own does not address the problem of complex requirements and operating environments. Complexity cannot be eliminated but we can try to master it. To master complexity, we advocate the use of *abstraction*. Abstraction is about simplifying our understanding of a system to arrive at a model that is focused on what we judge to be the key or critical features of a system. A good abstraction will focus on the purpose of a system and will ignore details of how that purpose is achieved. We do not ignore the complexity indefinitely: instead, through incremental modelling and analysis, we can layer our understanding and analysis of a system. This incremental treatment of complexity is the other side of the coin to abstraction, namely, *refinement*.

The Event-B modelling approach is intended for early stage analysis of computer systems. It provides a rich modelling language, based on set theory, that allows precise descriptions of intended system behaviour (models) to be written in an abstract way. It provides a mathematical notion of consistency together with techniques for identifying inconsistencies or verifying consistency within a model. It also provides a notion of refinement of models together with a notion of consistency between a model and its refinement. By abstracting and modelling system behaviour in Event-B, it is possible to identify and fix requirements ambiguities and inconsistencies at the specification phase, much earlier in the development cycle than system testing. In this way, rather than having an error-discovery profile in which most errors are discovered during system testing, we would arrive at an ideal profile in which more errors are discovered as soon as they are introduced. This paper will focus on precision and verification of consistencies in abstract specifications and does not cover refinement of models. Section 13 points to some further reading on refinement.

Requirements and formal models

We assume that the results of any requirements analysis phase is a requirements document written in natural language. There remains a potentially large gap between these informal requirements and a formal model. In this paper we will touch on this gap but not address it in any comprehensive way. In the context of a system development that involves both informal requirements and formal specification, it is useful to distinguish two notions of validation as follows:

- *Requirements validation* involves analysing the extent to which the (informal) requirements satisfy the needs of the stakeholders.
- *Model validation* involves analysing the extent to which the (formal) model accurately captures the (informal) requirements.

Both of these forms of validation require the use of human judgement, ideally by a range of stakeholders. In addition, we can perform mathematical judgements on a formal model. We refer to this use of mathematical judgements are *model verification*, that is, the extent to which a model satisfies a given set of mathematical judgements. Key to the effective use of model verification is strong tool support that automates the verification effort as much as possible. Arriving at good abstractions, formalising them, enriching models through refinement and making mathematical judgements all require skill and effort. This upfront effort is sometimes referred to as *front-loading*: putting more effort than is usual into the early development stages in order to save test and fix effort later.

Overview of paper

Logic and set theory are the mathematical basis of Event-B. In this paper we explain how these mathematical concepts are used to write precise specifications in the form of Event-B models and how we reason about such models using mathematics. We use *sets* as a form of abstract data structure to model collections of entities that have a certain status and we define *events* that specify ways in which these sets may be manipulated to represent changes in the status of entities. For example, Section 2 shows how a set is used to model collections of users who have permission to be in a building and presents events for adding users to this set when they are registered and for removing users from this set when they are de-registered. Mathematical operators on sets allow us to easily specify manipulations of sets and Section 3 provides a brief overview of the set operators used throughout this paper while Section 5, covers issues that arise with finiteness of sets and determining the size of sets.

Sets are used to model collections of entities of the same kind. When we want to model connections between different kinds of entities, we use *relations* which are covered in Sections 6 and 7.

The main unit of specification in Event-B is a *machine* and this is introduced in Section 4. A machine contains a list of variables and a list of events that modify the variables in precisely defined ways. A machine also contains a list of *invariants* that describe desired properties of the variables of a machine, e.g., *users inside the building must have permission to be there.*

Many set operators are defined using mathematical logic. For example, *intersection* of sets is defined in terms of *logical conjunction*: an element x is in the intersection of sets S and T if x is in S *and* x is in T. Section 3 gives a brief overview of the main logical operators used and the connection between logic and sets. At various stages in the paper additional mathematical operators are introduced to support the required modelling. Mathematical definitions are provided to help the reader's understanding of the operators and to support mathematical reasoning. Logic allows us to reason about machines, in particular, it allows us to prove that events of a machine preserve constraints specified by invariants and this is covered in several places in the paper.

We use several case studies to illustrate the use of the modelling and reasoning concepts of Event-B. Sections 2–4 use a simple example of a system for controlling access to a building. This case study is used to in illustrate the use of sets as abstract data structures and the use of invariants for specifying desired properties of structures. The case study is also used to provide the initial illustration of the use of mathematical reasoning to verify properties of a machine. In Sections 8 and 9 we use a generalisation of the access control system that manages access to a collection of buildings rather than a single building. This case study is used to illustrate the use of relations (e.g., between users and buildings) and to consolidate the concepts from the earlier sections. A *function* is a special case of a relation and we use an example of a simple banking system to illustrate the use of functions in modelling in Section 10.

While reading and understanding a specification written in a language such as Event-B requires a relatively small amount of training, the ability to write a formal specification requires more skill and, as with programming, that skill is best developed through practice. Using the access control example, Sections 8 and 9 provide guidelines on how an Event-B model can be constructed from a list of informal system requirements. The author has found that the use of class diagrams provides a useful initial bridge between informal requirements and formal models involving relations and functions. Class diagrams help to identify in a graphical way the various entities appropriate for a system model and the various connections between the different kinds of entities. For example, in an access control model, *users* and *buildings* are two relevant entities and the access rights are represented by an association between those entities.

A key advantage of Event-B is the availability of tool support for reasoning about formal models. Sections 11 and 12 provide an overview of tool support (animation, model checking, proof obligation generation and automated proof) that is available to support model validation and verification.

Section 13 briefly overviews some material that provides a deeper treatment of Event-B than this paper and also overviews other related formal modelling and analysis methods.

2 Modelling with sets and invariants

We illustrate modelling with sets through an example of access control to a building. The system should allow only registered users to enter the building and should keep track of which users are inside the building. We start by considering two sets, *in*, representing the users who are inside the building and *out*, representing the users who are outside the building. The two sets are illustrated by the Venn diagram in Figure 1.

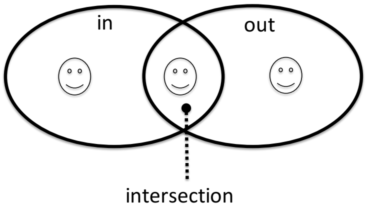

Fig. 1. Venn diagram for *in* and *out*

The diagram illustrates that we might have users in the overlapping area between the two sets (the intersection) and users in the non-overlapping areas. However, for this particular example, we would not expect any users to be both inside and outside the building so we would like to rule this possibility out as illustrated in Figure 2. When the intersection of two sets is empty, we say the sets are *disjoint* and disjointness is illustrated in Figure 3 by having no overlap between the sets. This disjointness property may be represented by the following mathematical equation:

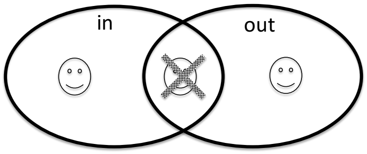

Fig. 2. Venn diagram: empty intersection

$$in \cap out = \varnothing$$

The equation says that the intersection of the two sets ($in \cap out$) is empty ($\varnothing$).

The system we are modelling is dynamic in that users may enter or leave the building. In our model, this will be reflected by changes to the sets *in* and *out*. Thus we treat *in* and *out* as *variables* whose values may be changed. We make

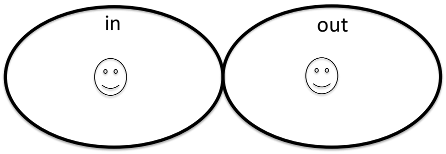

Fig. 3. Venn diagram: disjoint sets

the following declaration:

$$\textbf{variables} \quad in, out$$

While the values of the variables may change, the disjointness property should remain true. An *invariant* is a property of one or more variables that should be preserved by an changes to the variables so, not matter what changes occur in the system, it should never get into a state in which the invariant is falsified. We require the disjointness equation to be an invariant of our access control model so we declare:

$$\textbf{invariant} \quad in \cap out = \varnothing$$

We mentioned the concept of registered users at the beginning of this section so we introduce a set, called *register*, representing registered users. We will allow the set of registered user to change, e.g., by adding a new user to the register, so we declare *register* to be a variable:

$$\textbf{variable} \quad register$$

Only registered users should be allowed in the building and we model this property by requiring *in* to be contained entirely within *register* as illustrated in Figure 4. As the diagram illustrates, a user who is in the building must also be registered. The diagram also illustrates that some users may be registered without being in the building. We say that *in* is a *subset* of *register*, written in mathematical notation as: $in \subseteq register$. We declare this subset property on *in* and *register* as an invariant:

$$\textbf{invariant} \quad in \subseteq register$$

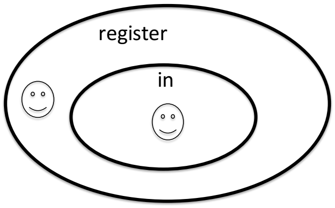

Fig. 4. Venn diagram: subset

What about the relationship between the set *out* and the set *register*? Up to now we have not been clear about whether *out* represents all possible users including those that are not registered. Let us make a modelling decision that *out* represents exactly those users who are registered and are outside the building. Thus registered users are either in or out. This is illustrated by Figure 5 which shows that *register* is the *union* of *in* and *out*. Mathematically, the union is written as $in \cup out$ and we declare the union property as an invariant:

$$\textbf{invariant} \quad register = in \cup out$$

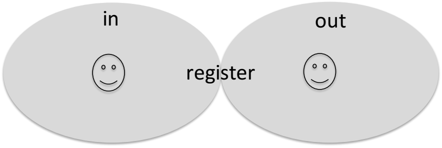

Fig. 5. Venn diagram: set union

We can add behaviours to the model, such as a user entering the building or leaving the building, by specifying *events*. An event defines an atomic transition on states, that is, it defines a relationship between a state before the event is executed and the resulting state after the event is executed. An atomic transition representing a user entering the building is illustrated by Figure 6. This shows Venn diagrams for the variables *in* and *out* both before and after execution of the *Enter* event. In the before state, user u is in the set *out* while in the after state u is in the set *in*, i.e., the *Enter* event moves user u from *out* to *in*. As shown in Figure 6, the specification of the *Enter* event has three parts:

- *parameter* u representing the user who is entering the building
- a *guard* requiring that the user is in the set *out*
- an *action* that moves u from *out* to *in*,

The *Enter* event is specified in Event-B notation as follows (including comments):

```
Enter  ≙
    any u where
       grd1:  $u \in out$                  // u must be registered and outside
    then
       act1:  $in := in \cup \{u\}$          // add u to in
       act2:  $out := out \setminus \{u\}$       // remove u from out
    end
```

Here the keyword **any** indicates that u is a parameter. The guard of the event appears between the **where** and **then** keywords while the actions appears

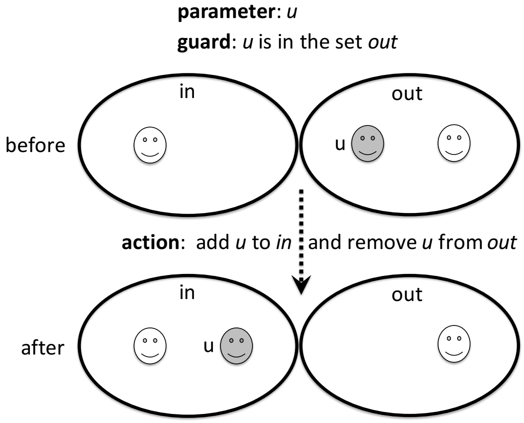
parameter: *u*
guard: *u* is in the set *out*
in
out
before
u
action: add *u* to *in* and remove *u* from *out*
in
out
after
u

Fig. 6. Illustrating the *Enter* event

between the **then** and **end** keywords. The guard labelled $grd1$ requires that u is in the set out, written $u \in out$. The actions of the event specify *assignments* that modify some of the variables of the model, e.g., the action labelled $act1$ assigns the value $in \cup \{u\}$ to the variable in. The action labelled $act2$ uses *set difference* to remove u from out: $s \setminus t$ is the difference between sets s and t, i.e,. the set elements of s that are not in t.

Although the *Enter* event contains several actions, the order in which the actions appear does not matter as all of the actions are executed together, not in series.

The syntax for specifying events will be described systematically in Section 4. Before presenting further details of the model of the building access control we give a quick overview, in the next section, of the key concepts of set theory that are important in the Event-B notation.

3 Overview of Set Theory

Here we list some key features of sets:

- A *set* is a collection of *elements*.
- Elements are *not ordered* by a set.
- Set *membership* is an important relationship between an element and a set. We write $x \in S$ to specify that element x is a member of set S.
- Elements may themselves be sets, i.e., we can have a set of sets.
- Sets may be *enumerated* within braces, e.g., the set $\{ a, b, c \}$ contains three elements, a, b and c.
- The set containing no elements, the *empty* set, is written $\varnothing$.

Set membership is a boolean property relating an element and a set, i.e., either x is in S or x is not in S. This means that there is no concept of an element occurring more that once in a set, e.g.,

$$\{a, a, b, c\} \;=\; \{a, b, c\}.$$

Set membership says nothing about the relationship between the elements of a set other than that they are members of the same set. This means that the order in which we enumerate a set is not significant, e.g.,

$$\{a, b, c\} \;=\; \{b, a, c\}.$$

These two characteristics distinguish sets from data structures such as lists or arrays where elements appear in order and the same element my occur multiple times.

3.1 Typing and powersets

All the elements of a set must have the same *type* where a type is a special kind of set known as a *carrier set*. Figure 7 illustrates that the variables *register*, *in* and *out* are all subsets of the carrier set *USER*. In Event-B we use an invariant to define the carrier set of a variable, e.g.,

$$\textbf{invariant} \quad register \subseteq USER$$

This declaration means that all the elements of the set *register* have the type *USER*. If we also have the invariant $register = in \cup out$, the elements of *in* and *out* must have the same type as the elements of *register*. That is, the type of *in* and *out* can be inferred from the type of *register* because of the invariant $register = in \cup out$. All the elements of a set must have the same type.

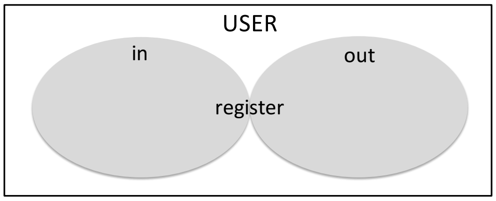

Fig. 7. Carrier set *USER*

A carrier set is maximal in that it is not a subset of any other set. A model may contain several carrier sets and these are implicitly disjoint from each other. For example, we could have a model that contains two carrier sets *USER* and *BUILDING*. We cannot combine carrier sets using set union, intersection or difference, e.g., $USER \cup BUILDING$ is invalid. In Section 6 we will see that we can combine carrier sets in another way to form relations. A carrier set remains fixed during the execution of a model, i.e., actions of an event cannot assign to a carrier set.

Suppose C is a carrier set. To define the type of an element x to be C, we simply declare $x \in C$. If S is not a carrier set and $S \subseteq C$, then the declaration $x \in S$ means that the type of x is C.

The Event-B notation has an in-built carrier set representing integers, written $\mathbb{Z}$. Elements of this set can be written using the usual literals, e.g., 1, 2, 3. The Event-B notation supports the usual arithmetic operators for integers such as addition and multiplication.

A *powerset* of a set is the set of all subsets of that set. For set S, we write $\mathbb{P}(S)$ for the powerset of S. For example,

$$\mathbb{P}(\{a,b,c\}) = \{\ \varnothing,\ \{a\},\ \{b\},\ \{c\},\ \{a,b\},\ \{a,c\},\ \{b,c\},\ \{a,b,c\}\ \}$$

Note that both the empty set and the set itself are contained within a set's powerset.

Up to now we have referred to the type of the elements of a set, e.g., all the elements of *register* have type *USER*. What about the type of the set itself (as opposed to the elements of the set)? We use the powerset operator to define the type of the set itself:

If the elements of a set S are of type C then the type of S is $\mathbb{P}(C)$.

For example, we have:

- the type of the set $register$ is $\mathbb{P}(USER)$
- the type of the set $\{1, 2, 3\}$ is $\mathbb{P}(\mathbb{Z})$

3.2 Expressions and predicates

Expressions are syntactic structures for specifying values (elements or sets). Literals (e.g., 3, $\varnothing$) are basic expressions as are variables (e.g., $register$) and carrier sets (e.g., *USER*). Compound expressions are formed by applying expressions to operators such as $x + y$ and $S \cup T$ to any level of nesting.

Predicates are syntactic structures for specifying *logical* statements, i.e., statements that are either *true* or *false* (but not both). Equality of expressions is an example predicate, e.g., $register = in \cup out$. Set membership and subset relations are other examples. For integer elements we can write ordering predicates such as $x \leq y$. Assume that a, S, T, x and y are expressions (S and T are set expressions while x and y are integer expressions). We have available the following basic predicates:

Basic predicates: $a \in S, \quad S \subseteq T, \quad S = T, \quad x = y, \quad x < y, \quad x \leq y$

Predicate operators: Compound predicates are formed using the standard logical operators listed in the following table (assume P and Q are predicates):

Name	Predicate	Definition
Negation	$\neg P$	P does *not* hold
Conjunction	$P \ \wedge \ Q$	both P *and* Q hold
Disjunction	$P \ \vee \ Q$	either P holds *or* Q holds
Implication	$P \ \Rightarrow \ Q$	if P holds, then Q holds

Quantified predicates: We have seen that a predicate P may refer to one or more variables, e.g., $x \leq y$. We can quantify over a variable of a predicate universally or existentially:

Name	Predicate	Definition
Universal Quantification	$\forall x \cdot P$	P holds for *all* x
Existential Quantification	$\exists x \cdot P$	P holds for *some* x

In the predicate $\forall x \cdot P$ the quantification is over all possible values in the type of the variable x. Typically we constrain the range of values using implication, e.g., we could specify that every element of the set *in* is also an element of the set *register*:

$$\forall u \ \cdot \ u \in in \ \Rightarrow \ u \in register$$

In the case of existential quantification we typically constraint the range of values using conjunction, e.g., we could specify that integer z has a positive square root as follows:

$$\exists y \ \cdot \ 0 \leq y \ \wedge \ y \times y = z$$

Free and bound variables: A variable that is universally or existentially quantified in a predicate is said to be a *bound* variable. A variable referenced in a predicate that is not bound variable is called a *free* variable. For example, in the above predicate, y is bound while z is free.

Predicates on sets can be defined in terms of the logical operators as follows:

Name	Predicate	Definition
Subset	$S \subseteq T$	$\forall x \cdot x \in S \Rightarrow x \in T$
Set equality	$S = T$	$S \subseteq T \wedge T \subseteq S$

3.3 Set Operators

We already used expression operators on sets such as union and intersection. We now defines these operators more precisely using predicates. A predicate provides a way of defining a set: the set of elements that satisfy the predicate. Consider the union $S \cup T$. The elements of the union are those elements that are either in S or in T. More precisely, the set $S \cup T$ is defined by the set of elements x satisfying the predicate $x \in S \vee x \in T$. The following table provides definitions of the set operators using logical operators:

Name	Predicate	Definition
Union	$x \in S \cup T$	$x \in S \vee x \in T$
Intersection	$x \in S \cap T$	$x \in S \wedge x \in T$
Difference	$x \in S \setminus T$	$x \in S \wedge x \notin T$
Powerset	$x \in \mathbb{P}(S)$	$x \subseteq S$
Empty set	$x \in \varnothing$	$false$

Note that $x \notin T$ is a shorthand for $\neg(x \in T)$. Similarly we can use the shorthand $S \neq T$ for $\neg(S = T)$.

4 Structuring models with machines

We have already introduced several Event-B constructs such as carrier sets, variables, invariants and events. So is there a way of packaging these into components? A *machine* is an Event-B component in which the variables, invariants, and events are placed. Carrier sets that are required by a machine are placed in a separate component called a *context*. An Event-B context can also contain constants and axioms. The axioms are predicates that define properties of the carrier sets and the constants. For example, for our building access control example, we might want to model a capacity constraint on the building. We could do this by introducing a constant *max_capacity* and an axiom stating that *max_capacity* is greater than zero.

4.1 Context

A context with name $C1$ has the following form:

context $C1$

sets ⟨*list of carrier sets*⟩

constants ⟨*list of constants*⟩

axioms ⟨*list of labelled axioms*⟩

end

The following example is a context for the building access model which introduces a carrier set and a constant:

context *BuildingContext*
sets *USER*
constants *max_capacity*
axioms
  axm1: $\mathit{max\_capacity} \in \mathbb{Z}$
  axm2: $\mathit{max\_capacity} > 0$
end

Each axiom in the context is a predicate. For traceability purposes, each axiom in a context is given a unique label (e.g., axm1). The axioms in this context specify that *max_capacity* is an integer (axm1) whose value is assumed to be greater than zero (axm2).

4.2 Machine

In Event-B, a machine defines the dynamic behaviour of a model through events that are guarded by and act on the variables. The events are expected to maintain the invariants; we will see later how this is verified. A machine may *see* one or more contexts which provide the carrier sets, constants and axioms to be used by the machine. A machine with name M has the following form:

machine $M1$

sees $\langle$*list of context names*$\rangle$

variables $\langle$*list of variables*$\rangle$

invariants $\langle$*list of labelled invariants*$\rangle$

events $\langle$*list of events*$\rangle$

end

For example, part of the machine for the building access is specified as follows:

```
machine  $Building$
sees  $BuildingContext$
variables  $register,\ in,\ out$
invariants
    inv1:  $register \subseteq USER$
    inv2:  $register = in \cup out$
    inv3:  $in \cap out = \varnothing$
events  ...
```

This machine is named *Building*; it sees the previously defined *BuildingContext* and it contains three variables. *register*, *in* and *out*. As discussed previously, the invariants specify that registered users are of type *USER* (inv1), registered users are either inside or outside (inv2) and no user is both inside and outside (inv3).

We postpone treatment of any building capacity constraint to later.

In Section 2 we showed the *Enter* event which models a user entering the building. Here we present the general syntax of event definitions. Each event of a machine has a name, a list of parameters, a list of guards and a list of actions structured as follows:

```
$\langle name \rangle \;\hat{=}$
    any $\langle$list of parameters$\rangle$ where
      $\langle$list of labelled guards$\rangle$
    then
      $\langle$list of labelled actions$\rangle$
    end
```

Guards are predicates that specify conditions on the machine variables and the event parameters. Each action assigns a value to a machine variable and has the form:

$$\langle \mathit{variable} \rangle \;:=\; \langle \mathit{expression} \rangle$$

For example, here is the *Enter* event again:

```
$Enter \ \hat{=}$
    any $u$ where
        grd1:  $u \in out$
    then
        act1:  $in := in \cup \{u\}$
        act2:  $out := out \setminus \{u\}$
    end
```

An event may be executed for particular values of the parameters when all its guards are satisfied. When an event is executed, all of the actions of that event are performed simultaneously. Because of the simultaneity, it is not allowed for two different actions in an event to assign to the same variable as this would lead to conflicting updates. As with invariants, the guards and actions are labelled.

4.3 Preserving Invariants

When specifying an event, it is important to ensure that the invariants are preserved by its actions. We can assume that the invariants are satisfied prior to execution of the event and we need to demonstrate that the actions do not result in any invariant being violated.

Let us consider whether the *Enter* event preserves the invariants of the access control model. Invariant $inv1$ refers to the *register* variable only and, since none of the actions modify *register*, this invariant is trivially preserved. Invariant $inv2$ is an equation specifying that *register* is the union of *in* and *out*:

$$register \ = \ in \cup out \tag{1}$$

The actions of the *Enter* event modify the variables in right-hand side of the equation but not the left-hand side. However since u is moved from *out* to *in*, the overall value on the right-hand side remains unchanged and the equation remains valid. More precisely, the effect of the actions of the *Enter* event on the invariant can be represented by replacing each variable in the invariant by the expression on the right-hand side of the assignment to that variable, i.e., replace *in* by $in \cup \{u\}$ and *out* by $out \setminus \{u\}$, giving:

$$register \ = \ \underline{(in \cup \{u\})} \ \cup \ \underline{(out \setminus \{u\})} \tag{2}$$

The result of replacing *in* and *out* are underlined in equation (2). We say that invariant $inv2$ is preserved when equation (2) follows from equation (1) and the guard of *Enter*, that is, when proving tequation (2), we can assume that equation (1) holds and that the guards of the *Enter* event hold. The proof is as follows:

$$\begin{array}{ll}
 & (in \cup \{u\}) \ \cup \ (out \setminus \{u\}) \\
= & \quad \text{“}\cup\text{ is associative and commutative”} \\
 & in \cup (out \setminus \{u\}) \cup \{u\}
\end{array}$$

$$\begin{array}{ll} = & \quad \text{“}grd1 : \; u \in out\text{”} \\ & in \cup out \\ = & \quad \text{“}inv2\text{”} \\ & register \end{array}$$

Each step in the simple proof is justified either by appealing to a rule of set theory (the first step), by appealing to an event guard (the second step) or to the invariant *inv*2 (the third step) with the justification indicated by “inverted commas”. Both union and intersection are associative and commutative as captured in this table:

Description	Rule
Union associative	$(s \cup t) \cup u \;=\; s \cup (t \cup u)$
Union commutative	$s \cup t \;=\; t \cup s$
Intersection associative	$(s \cap t) \cap u \;=\; s \cap (t \cap u)$
Intersection commutative	$s \cap t \;=\; t \cap s$

The second step in the above proof of equation (2) relies on the following simplification rule for sets which states that if x is in set s, then subtracting x from s and adding x to the result yields s:

Description	Rule
Simplify	$x \in s \;\;\Rightarrow\;\; (s \setminus \{x\}) \cup \{x\} \;=\; s$

An advantage of the actions of an event being executed simultaneously is that we do not need to consider intermediate states in which invariants might be violated. For example, if *act*1 and *act*2 were executed sequentially, *inv*2 would be violated in between *act*1 and *act*2 before being re-established by *act*2. To re-iterate: the actions within an event are always executed simultaneously and not sequentially.

We we have shown that the *Enter* event maintains *inv*1 and *inv*2. We now consider the remaining invariant, *inv*3. Invariant *inv*3 specifies that *in* and *out* are disjoint. If we removed action *act*2 from the *Enter* event, this would lead to a violation of *inv*2 as u would end up both *in* and *out*. However, since both actions together move u from *out* to *in*, their disjointness is preserved. Let us prove this mathematically. Invariant *inv*2 is:

$$in \cap out \;=\; \varnothing \tag{3}$$

As we have seen, the effect of the actions of the *Enter* event on the invariant can be represented by replacing each variable in the invariant by the expression on the right-hand side of the assignment to that variable, i.e., replace *in* by $in \cup \{u\}$ and *out* by $out \setminus \{u\}$, giving:

$$\underline{(in \cup \{u\})} \;\cap\; \underline{(out \setminus \{u\})} \;=\; \varnothing \tag{4}$$

The proof of this is captured by the following general rule about sets which states that if two sets are disjoint then removing elements from one and adding them to the other maintains the disjointness:

Description	Rule
Keep disjoint	$s \cap t = \varnothing \quad \Rightarrow \quad (s \setminus r) \;\cap\; (t \cup r) \;=\; \varnothing$

4.4 Machine Initialisation

Every machine has a special event (named **initialisation**) that initialises the machine variables. The acccess control machine is initialised by setting all three variables to be empty:

initialisation $\hat{=}$
act1: $in \;:=\; \varnothing$
act2: $out \;:=\; \varnothing$
act3: $register \;:=\; \varnothing$

An initialisation event has no guards nor parameters and the assignment expressions (right-hand side) cannot refer to the machine variables. This is because no assumptions can be made about the values of the variables prior to initialisation. The initialisation should *establish* the invariant, i.e., the values assigned to the variables together should satisfy the invariants. In this case, all three invariants are trivially established, i.e.,

$$\varnothing \;\subseteq\; USER$$
$$\varnothing \cap \varnothing \;=\; \varnothing$$
$$\varnothing \;=\; \varnothing \cup \varnothing$$

4.5 Other Access Control Events

We now look at some of the other events for access control: exiting the building, registering a new user and de-registering a user. The *Exit* event is the opposite of the *Enter* event: the user should be in the building and is moved from *in* to *out*:

```
$Exit \hat{=}$
    any $u$ where
        grd1: $u \in in$
    then
        act1: $in := in \setminus \{u\}$
        act2: $out := out \cup \{u\}$
    end
```

This event maintains the invariants based on similar arguments that we used previously for the *Enter* event.

When registering a user, we need a 'fresh' value to represent the new user, i.e, a value that is not already in the set *register*. This fresh value is then added to the set of registered users. The event could be specified as follows:

```
$RegisterUser1 \hat{=}$
    any $u$ where
        grd1: $u \in USER$
        grd2: $u \notin register$
    then
        act1: $register := register \cup \{u\}$
    end
```

The first guard gives a type to u while the second guard ensures that u is fresh. Let us consider whether the action violates any invariants. It turns out that the action violates the equation of $inv2$ ($register = in \cup out$): it expands the left-hand side without expanding the right-hand side. We can resolve this by adding an action that also expands the right-hand side of the equation. We can do this by adding u to *in* or to *out*. In this case, it makes more sense to add u to *out* rather than *in*, as we would not expect that the new user would end up inside the building immediately at the point at which they are registered. Thus, an improved version of the event is specified as follows:

```
$RegisterUser2 \hat{=}$
    any $u$ where
        grd1: $u \in USER$
        grd2: $u \notin register$
    then
        act1: $register := register \cup \{u\}$
        act2: $out := out \cup \{u\}$
    end
```

This specification of the user registration does maintain $inv3$. Let us prove this mathematically. Invariant $inv3$ is:

$$register \;=\; in \;\cup\; out \tag{5}$$

The actions of the event modify this to the following equation:

$$\underline{register \cup \{u\}} \;=\; in \;\cup\; \underline{(out \cup \{u\})} \tag{6}$$

We prove that this equation follows from the invariant:

$$\begin{array}{ll} & register \cup \{u\} \\ = & \quad \text{“}inv3\text{”} \\ & (in \ \cup \ out) \cup \{u\} \\ = & \quad \text{“}\cup \text{ is associative”} \\ & in \cup (out \cup \{u\}) \end{array}$$

A user who is already registered may be de-registered by removing them from *register*. Removing u from *register* without removing u from *in* or *out* will lead to a violation of *inv*2. One solution is to remove u from both *in* and *out* leading to the following specification of the event for de-registering:

```
$DeRegisterUser1 \ \hat{=}$
    any $u$ where
        grd1:  $u \in register$
    then
        act1:  $register := register \setminus \{u\}$
        act2:  $out := out \setminus \{u\}$
        act3:  $in := in \setminus \{u\}$
    end
```

This specification will preserve *inv*2 since u is removed from both sides of the equation. This event is applicable whether registered user u is inside or outside the building. However, if we consider a building access control system, it probably does not make sense to de-register a user while they are inside the building so we strengthen the guard to specify that u is outside (and registered):

```
$DeRegisterUser2 \ \hat{=}$
    any $u$ where
        grd1:  $u \in out$
    then
        act1:  $register := register \setminus \{u\}$
        act2:  $out := out \setminus \{u\}$
    end
```

Note that this version does not modify *in*. If u was a member of *in*, this would result in a violation of *inv*2. However, from the guard of the event we know that u is an element of *out* and, since *in* and *out* are disjoint (*inv*3), we know that u cannot be an element of *in*. Thus it is sufficient to remove u from *out* in order to maintain *inv*2.

We leave it as an exercise for the reader to prove that *DeRegisterUser2* preserves the invariants. The following rules are used in the proofs:

Description	Rule
Distribute difference	$(s \cup t) \setminus r \;=\; (s \setminus r) \;\cup\; (t \setminus r)$
Simplify	$x \notin s \quad\Rightarrow\quad s \setminus \{x\} \;=\; s$
Keep disjoint	$s \cap t = \varnothing \quad\Rightarrow\quad (s \setminus r) \;\cap\; t \;=\; \varnothing$

4.6 Machine behaviour and nondeterminism

A simple way of thinking about the behaviour of an Event-B machine is as a *transition system* that moves from one state to another through execution of events. The states of a machine are represented by the different configurations of values for the variables. The variables of a machine are initialised by execution of the special **initialisation** event. An event is enabled in some state for some parameter values if all of the guards of the event are satisfied. For example, the *Enter* event in the access control model is enabled for parameter value $u1$ in any state in which $u1$ is an element of the variable *out*.

In any state that a machine can reach, an enabled event is chosen to be executed to define the next transition. If several events are enabled in a state, then the choice of which event occurs is nondeterministic. Also, if an event is enabled for several different parameter values, the choice of value for the parameters is nondeterministic – the choice just needs to satisfy the event guards. For example, in the *RegisterUser2* event, the choice of value for parameter u is nondeterministic, with the choice of value being constrained by the guards of the event to ensure that it is a fresh value.

Treating the choice of event and parameter values as nondeterministic is an abstraction of different ways in which the choice might be made in an implementation of the model. For example, if it is an interactive system, the choice might be offered to a user via a graphical interface. If it is an information processing system, the choice might be made by some scheduler. If the machine reaches a state in which no event is enabled, then it is said to be *deadlocked*.

5 Finiteness, cardinality and well-definedness

Previously we considered the possibility of placing a constraint on the the number of users allowed inside the building at any one time. We could represent this as an invariant specifying that the number of elements in the set *in* is bounded by the constant *max_capacity*. In set theory, the number of elements in a set is called its *cardinality* and in Event-B this is written as $card(S)$. For example,

$$card(\{a, b, c\}) = 3.$$

However a word of caution: cardinality is only defined for finite sets. If S is an infinite set, then $card(S)$ is undefined. Whenever we use the $card$ operator, we must ensure that it is only applied to a finite set. This issue of *well-definedness* applies to some other operators as well. For example, division by zero is not well-defined and when using division we must ensure that the divisor is not zero.

As is standard in set theory, sets in Event-B may be finite or infinite. For example, the set of integers is infinite. A carrier set defined in a context is infinite unless we explicitly specify that it is finite. Naturally, an enumerated set, e.g., $S = \{a, b, c\}$, is finite. We can specify that a set S is finite using the predicate $finite(S)$. In the building access system, we would expect the set of people who are inside the building to be finite which we write as $finite(in)$. Initially in is empty and thus finite. The only way of expanding the set in is through the *Enter* event which adds one user at a time. Thus the set in can never become infinite.

To model the finiteness and capacity constraints on the access control, we extend the set of invariants of the machine as follows:

```
invariants
    ...
    inv4:  $finite(in)$
    inv5:  $card(in) \leq max\_capacity$
```

In $inv5$, $card(in)$ is well-defined since we know that in is finite from $inv4$. Considering preservation of $inv4$, the only event that expands in is the *Enter* event and it maintains the finiteness of in ($inv4$) by the argument outlined above. For $inv5$, $max\_capacity$ is a constant so cannot decrease during execution of the machine so we only need to consider events that might cause $card(in)$ to increase. As we have already said, *Enter* is the only event that expands in and thus increases $card(in)$. The *Enter* event as previoulsy specified places no constraint on the number of people already in the building so we need to strengthen it by adding guard $grd2$ as follows:

```
$Enter2 \;\hat{=}$
    any $u$ where
        grd1:  $u \in out$
        grd2:  $card(in) < max\_capacity$
    then
        act1:  $in := in \cup \{u\}$
        act2:  $out := out \setminus \{u\}$
    end
```

Note that $grd2$ requires $card(in)$ to be strictly less that $max\_capacity$ in order to ensure that the size of the resulting value for in is less than or equal to $max\_capacity$.

The following table summaries the finiteness and cardinality operators we have just introduced. The table also includes a column to indicate when a predicate or expression is well-defined:

Name	Operator	Meaning	Well-definedness
Finite	$\mathit{finite}(S)$	set S is finite	*true*
Cardinality	$\mathit{card}(S)$	number of elements in set S	$\mathit{finite}(S)$

Note that some of our definition tables do not have a well-definedness column. In these cases the predicate or expression is always well-defined.

The following rules about finiteness and cardinality are used to prove that $inv4$ and $inv5$ are preserved by the events:

Description	Rule
Finite union	$finite(s) \land finite(t) \quad \Rightarrow \quad finite(s \cup t)$
Finite difference	$finite(s) \quad \Rightarrow \quad finite(s \setminus t)$
Increase card	$x \notin s \quad \Rightarrow \quad card(s \cup \{x\}) \;=\; card(s) + 1$
Decrease card	$x \in s \quad \Rightarrow \quad card(s \setminus \{x\}) \;=\; card(s) - 1$

6 Introducing relations

We have seen how sets can be used to model access control for a building. We introduced a carrier set to represent users but we did not introduce a carrier set to represent buildings. The reason for not introducing buildings is that our model was intended for a single building and the identity of that building was implicit. Let us consider generalising our modelling of access control to a system with multiple buildings. For this we introduce a carrier set representing buildings so that we can distinguish different buildings. Figure 8 illustrates the two distinct carrier sets, one for users and the other for buildings.

Rather than allowing registered users to enter any building, we would like to model a more fine-grained control over which buildings a user is allowed to enter. This is illustrated in Figure 9 which represents a permission relation between users and buildings. An arrow from a user to a building indicates that that particular user has permission to enter that building. For example, in Figure 9, user $u1$ has permission to enter two of the buildings, $b1$ and $b2$. Figure 9 represents three different sets: a set of *users*, a set of *buildings* and a set of *arrows* between users and buildings. Mathematically an arrow from user u to building b is represented by a *pair* of elements, written $u \mapsto b$. A *relation* is represented

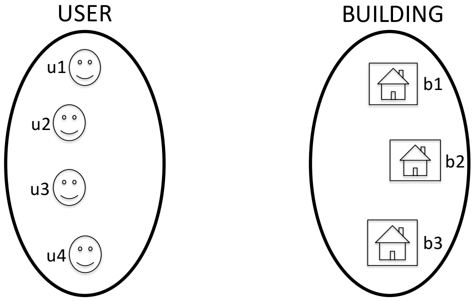

Fig. 8. Distinct carrier sets

by a set of pairs, for example, the permission relation of Figure 9 is represented by the following set P of pairs:

$$P = \{\ u1 \mapsto b1,\quad u1 \mapsto b2,\quad u2 \mapsto b1,\quad u2 \mapsto b3,\quad u4 \mapsto b2,\quad u4 \mapsto b3\ \}$$

The permission model demonstrates that a relation allows us to connect distinct carrier sets. Management of relationships between different kinds of entities is a key role of many computerised systems, including access control, business systems, information systems and communications systems. Thus relations are a useful mathematical structure for modelling such systems.

A pair $u \mapsto b$ has a *first* element u and a *second* element b. Given a set of pairs, it is useful to refer to the set of the first elements of all pairs, called the *domain*, and the set of second elements, called the *range*. For the example relation P above, we have

$$\begin{aligned} dom(P) &= \{\ u1, u2, u4\ \} \\ ran(P) &= \{\ b1, b2, b3\ \} \end{aligned}$$

Here, $dom(P)$ represents the set of users who have permission to enter some building while $ran(P)$ represents the set of buildings for which some user has permission to enter.

Figure 9 labels the permission relation as *many-to-many*. This means that many different domain elements can be mapped to the same range element, e.g., $u1$ and $u2$ are both mapped to $b1$, and also that the same domain element can be mapped to many different range elements, e.g., $u1$ is mapped to both $b1$ and $b2$.

As well as modelling the permission relation, we can also model the current location of a user using a relation as illustrated in Figure 10. We would not expect a user to be located in more than one building at a time and thus the location relation is required to be a *many-to-one* relation, meaning that a domain element can be mapped to exactly one range element rather than many. A many-to-one relation still allows many different domain elements to be mapped to the same range element, e.g., $u2$ and $u4$ are both located in the same building in

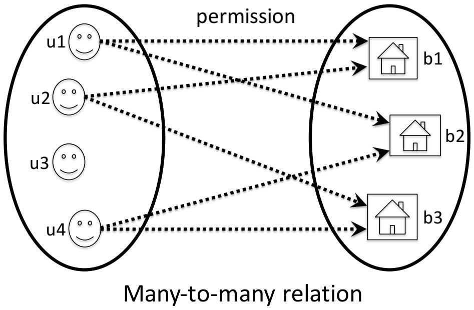

Fig. 9. Permission relation

Figure 10. Many-to-one relations are also called *functions* and are covered in more detail in Section 7.3.

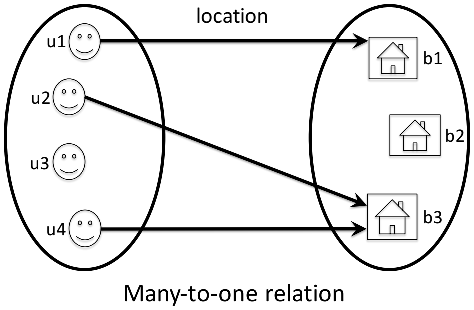

Fig. 10. Location relation

Since the permission and location relations are themselves sets, we can formulate a connection between them. For an access control system we require that if a user is located in a building, then they have permission to be in that building. This requirement is represented by specifying that *location* is a subset of *permission*, i.e., any pair in the *location* relation is also a pair of the *permission* relation. The connection between the two relations is illustrated in Figure 11 where *location* is clearly a subset of *permission*.

A many-to-one relation is a special case of a many-to-many relation. A further special case is a *one-to-one* relation in which each domain element is related to exactly one range element and each range element is related to exactly one domain element. This is illustrated in Figure 12 where the *location* relation is such that users are mapped one-to-one with buildings, i.e., no two users are located in the same building. If we required single occupancy for the buildings,

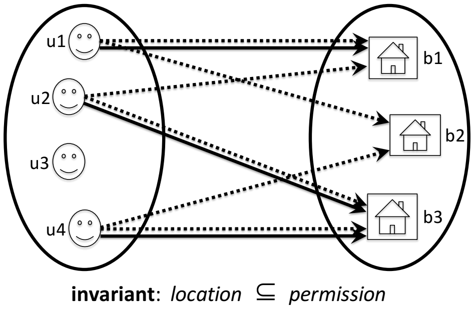

Fig. 11. Location satisfies permission

then we could represent this with an invariant specifying that *location* is one-to-one.

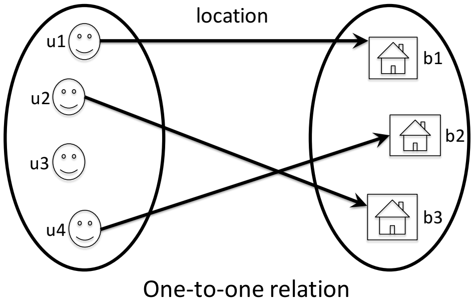

Fig. 12. Location with single occupancy

7 Cartesian products and relations

We have seen that an ordered pair is an element consisting of two parts, a *first* part and a *second* part, and is written as $x \mapsto y$. Given two sets S and T, we can form what is called their *Cartesian product*. This is the set of all those pairs whose first component is in S and second component is in T. The Cartesian product of S with T is written $S \times T$. For example, the Cartesian product of $\{a, b, c\}$ with $\{1, 2\}$ is expanded to a set of pairs as follows:

$$\begin{array}{rcl} \{a,b,c\} \times \{1,2\} & = & \{\ a \mapsto 1,\ a \mapsto 2, \\ & & \quad b \mapsto 1,\ b \mapsto 2, \\ & & \quad c \mapsto 1,\ c \mapsto 2\ \} \end{array}$$

Here we see, for example, that $a \mapsto 1$ is an element of the Cartesian product since a is in $\{a, b, c\}$ and 1 is in $\{1, 2\}$. More generally $x \mapsto y$ is an element of $S \times T$ when x is in S and y is in T as shown in the following table:

Name	Predicate	Definition
Cartesian product	$x \mapsto y \;\; \in \;\; S \times T$	$x \in S \;\; \wedge \;\; y \in T$

The following derivation shows that the product of any set with the empty set is itself empty ($S \times \varnothing \;=\; \varnothing$):

$$\begin{array}{ll}
 & x \mapsto y \;\in\; S \times \varnothing \\
= & \quad \text{“Definition of } \times\text{”} \\
 & x \in S \;\wedge\; y \in \varnothing \\
= & \quad \text{“Definition of } \varnothing\text{”} \\
 & x \in S \;\wedge\; false \\
= & \quad \text{“Logic”} \\
 & false \\
= & \quad \text{“Definition of } \varnothing\text{”} \\
 & x \mapsto y \;\in\; \varnothing
\end{array}$$

7.1 Type constructors and structured types

In Section 3.1, we saw that the powerset operator is used to define the type of a set. The powerset operator can be used to construct a type $\mathbb{P}(T)$ from a type T so we refer to it as a *type constructor*. Similarly, Cartesian product is a type constructor: the type $S \times T$ is constructed from the types S and T. A *structured type* is a type formed using a type constructor such as $\mathbb{P}$ or $\times$.

- Powerset ($\mathbb{P}$) is the type constructor for sets.
- Cartesian product ($\times$) is the type constructor for ordered pairs.

In Event-B, constants, variables, parameter and expressions have a type and these types come in three forms

- Basic type: integer ($\mathbb{Z}$), Boolean.
- Carrier set, e.g., *USER*, *BUILDING*.
- Structured type: $\mathbb{P}(S), \quad S \times T$.

The type constructors can be nested and combined to form more complex structured types such as:

- Set of sets: $\mathbb{P}(\mathbb{P}(T))$
- Set of pairs: $\mathbb{P}(S \times T)$
- Pair of sets: $\mathbb{P}(S) \times \mathbb{P}(T), \quad S \times \mathbb{P}(T), \quad \mathbb{P}(S) \times T$

The following table presents some example expressions and their corresponding structured type:

Expression	Type
$\{5, 6, 3\}$	$\mathbb{P}(\mathbb{Z})$
$4 \mapsto 7$	$\mathbb{Z} \times \mathbb{Z}$
$\{5, 6, 3\} \mapsto 7$	$\mathbb{P}(\mathbb{Z}) \times \mathbb{Z}$
$\{\ 4 \mapsto 8,\ 3 \mapsto 0,\ 2 \mapsto 9\ \}$	$\mathbb{P}(\mathbb{Z} \times \mathbb{Z})$

7.2 Relations

Through the permission example (Figure 9) we have seen that a relation is modelled as a set of pairs, i.e., a structured type formed using both the × and $\mathbb{P}$ constructors. Because this structured type is a useful modelling construct, it is given its own symbol in Event-B: we write $S \leftrightarrow T$ as a shorthand for $\mathbb{P}(S \times T)$. The following table provides the definition of the relation arrow:

Name	Predicate	Definition
Relation	$r \ \in \ S \leftrightarrow T$	$r \ \in \ \mathbb{P}(S \times T)$

For the access control example, we may specify that the *permission* variable is a (many-to-many) relation with the following invariant:

$$\textbf{invariant} \quad permission \in USER \leftrightarrow BUILDING$$

Here is another example of a relation, named *directory*, that relates people to phone numbers:

$$\textbf{invariant} \quad directory \in PERSON \leftrightarrow NUMBER$$

A possible value for the directory is as follows:

$$\begin{aligned} directory = \{\ & mary \mapsto 287573, \\ & mary \mapsto 398620, \\ & john \mapsto 829483, \\ & jim \mapsto 398620\ \} \end{aligned}$$

It is worth pointing out the difference between the two arrow symbols used in representing relations:

$\leftrightarrow$ combines *two sets* to form a *set*.

$\mapsto$ combines *two elements* to form an *ordered pair*.

We already introduced the domain and range of a relation. These are defined by the following table:

Name	Predicate	Definition
Domain	$x \in dom(R)$	$\exists y \cdot\ x \mapsto y \ \in\ R$
Range	$y \in ran(R)$	$\exists x \cdot\ x \mapsto y \ \in\ R$

For the example directory shown above, we have:

$$\begin{aligned}\mathrm{dom}(directory) &= \{mary, john, jim\}\\ \mathrm{ran}(directory) &= \{287573, 398620, 829483\}\end{aligned}$$

Note that when we declare a constant or variable to be a relation between two sets, as well as defining its type, we are implicitly constraining the domain and range of the relation: Suppose we have $s \subseteq S$ and $t \subseteq T$ and we declare $r \in s \leftrightarrow t$, then it follows that

$$\begin{gathered} r \in \mathbb{P}(S \times T)\\ dom(r) \subseteq s\\ ran(r) \subseteq t \end{gathered}$$

7.3 Functions

From Figure 10 we saw that the *location* relation should be a many-to-one relation, i.e., each user is located in at most one building at any moment. The many-to-one property means that if a user u is in the domain of *location*, then that user is mapped to a single building by the location relation. In that case, we can write $location(u)$ to refer to the building that u is located in. For example, from Figure 10, we have:

$$\begin{gathered} location(u1) = b1\\ location(u2) = b3\\ location(u4) = b3 \end{gathered}$$

If a user u is not in the domain of *location*, then $location(u)$ is not well-defined. For example, from Figure 10, $u3$ is not in the domain of *location* therefore $location(u3)$ is not well-defined.

In general, a many-to-one relation f is said to be *functional*. This is written as $f \in S ⇸ T$ and means that every element in the domain of f is mapped to exactly one element in the range. The functionality property is specified mathematically by stating that if a domain value x is mapped to range value y, then x cannot be mapped to any other range value y'. This is shown in the following table:

Name	Predicate	Definition
Partial Function	$f \ \in \ S ⇸ T$	$f \ \in \ S \leftrightarrow T \quad \wedge$ $\forall x, y, y' \cdot \ x \mapsto y \in f \ \wedge \ y' \neq y$ $\Rightarrow \quad x \mapsto y' \notin f$

Note that when we declare $f \in S ⇸ T$ we say that f is a *partial* function. It is said to be partial because there may be values in the set S that are not in the domain of f. For example, from Figure 10, $u3$ is in *USER* but is not in the domain of *location*. A relation is said to be a *total* function from S to T when it is a partial function and its domain is exactly S:

Name	Predicate	Definition
Total Function	$f \ \in \ S \rightarrow T$	$f \ \in \ S ⇸ T \quad \wedge \quad dom(f) = S$

We have seen that we can write $location(u1)$ since *location* is functional. In general, when a relation f is functional, we can treat it as a mathematical function and write $f(x)$ for the value that x is mapped to. For $f(x)$ to be well-defined, two conditions must hold: f must be functional and x must be in the domain of f. This is shown in the following definition:

Name	Expression	Meaning	Well-definedness
Function Application	$f(x)$	$f(x) = y$ $\Leftrightarrow \ x \mapsto y \in f$	$f \in S ⇸ T \quad \wedge$ $x \in dom(f)$

This definition uses the *if and only if* $(\Leftrightarrow)$ logical operator: $P \Leftrightarrow Q$ is short for $P \Rightarrow Q \quad \wedge \quad Q \Rightarrow P$.

8 Access control specification

Now that we have explained relations and functions, we will make use of them to construct an Event-B specification of access control for multiple buildings. We start by presenting the high-level requirements in an informal way. As already stated, computer-based system is designed to satisfy some requirements in the real world and it is usual to express system requirements in natural language.

Documentation of the requirements in natural language will guide the construction of the Event-B specification and will also provide a form of “sanity check” against which to validate the Event-B specification. It helps understanding if we try to describe the intended purpose of the system being designed in a concise way. For the access control system this is as follows:

> **Purpose:** The purpose of the access control system is to ensure that users may be in a building only if they have permission to be in that building.

We provide a more detailed list of requirements, giving each one a label so that we can refer to it later. In each of the following requirements “the system” refers to the access control system:

- **FUN1:** The system shall maintain a register of recognised users and shall provide operations for managing the user register.
- **FUN2:** The system shall maintain a register of protected buildings and shall provide operations for managing the building register.
- **FUN3:** The system shall maintain the permissions for each user, determining the building they are allowed to enter, and shall provide operations for managing the permissions.
- **FUN4:** The system shall allow a user to enter a building provided they have permission.
- **FUN5:** The system shall allow a user to exit a building without constraint.
- **ASM1:** A user will be in at most one building at any time.
- **ASM2:** A user cannot move directly from one building to another building.

Most of these requirements are *functional requirements*[1], that is, requirements defining the intended function of the system. The last requirements in the list are *assumptions* about the environment in which the system is operating, e.g., we assume that there is a physical constraint on users which means they cannot be in more than one building at any time.

We referrred to the requirements as *high-level.* By this we mean there is not necessarily enough detail in the requirements to build the system. For example, **FUN4** does not provide detail on how a user would enter a building or how they might be prevented from entering. Nonetheless, we will see that it is still feasible and useful to make a formal analysis of the high-level requirements in Event-B.

8.1 Set and relations for access control

From the requirements **FUN1** and **FUN2** we identify two kinds of entity in the system, users and buildings. These give rise to two carrier sets for our specification of the system as defined in the following context:

context $BuildingAccessContext$
sets $USER$, $BUILDING$
end

[1] Not to be confused with a functional (many-to-one) relation!

Having identified the carrier sets, we consider what set variables to include in the model, i.e., variables that are subsets of a carrier set. Looking at **FUN1**, we see that a variable set of registered users is required. We will call this variable *user*, where $user \subseteq USER$. Similarly, **FUN2** suggests a variable set of buildings so we introduce a variable $building \subseteq BUILDING$. These two set variables are specified in an Event-B machine as follows:

machine $BuildingAccess$
sees $BuildingAccessContext$
variables $user,\ building,\ \ldots$
invariants

inv1: $user \subseteq USER$

inv2: $building \subseteq BUILDING$

Naming convention: Although it is not required by the Event-B language, we will use all UPPER case letters for names of carrier sets. When a model has multiple carrier sets representing different kinds of entity, we will use a lower case version of a carrier set name for the variable corresponding to the set of instances of that entity. For example, the user entity is represented by the carrier set *USER* and the set of instances (i.e., the register users) is represented by the variable *user*. While the carrier set is fixed, the instance set may be expanded or reduced through execution of events.

We also want to identify any required relational variables for our specification. **FUN3** suggests a relation to represent user permissions, while **FUN4** suggests a relation to represent user location. The diagrams in Figures 9 and 10 illustrate the permission and location relations between users and buildings. These diagrams are useful for illustrating specific instances of relations but they do not provide a general representation. To illustrate relations between sets more generally we use the class diagram shown in Figure 13. A *class diagram* is a construct from object oriented design that is used to represent classes and associations between classes. In Figure 13, the sets are represented as classes (the boxes) while the relations are presented as associations (the arrows). An association represents a relation between the indicated sets. We place the relevant mathematical symbol next to the name of the relation to indicate its nature

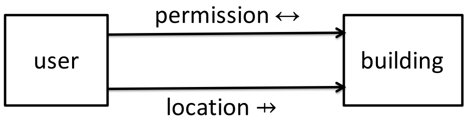

Fig. 13. Relations for access control

(many-to-many, many-to-one, etc). Thus Figure 13 indicates that *permission* is a relation between *user* and *building*:

$$permission \;\in\; user \leftrightarrow building$$

Because of **ASM1**, Figure 13 indicates that *location* is a partial function from *user* to *building*, giving rise to the following mathematical specification:

$$location \;\in\; user \pfun building$$

From the requirements, and with the aid of a class diagram, we have identified two kinds of variables for our Event-B specification:

- Set variables: *user*, *building*
- Relation variables: *permission*, *location*

The full list of variables and corresponding invariants is specified as follows:

variables *user*, *building*, *location*, *permission*
invariants
inv1: $user \;\subseteq\; USER$
inv2: $building \;\subseteq\; BUILDING$
inv3: $permission \;\in\; user \leftrightarrow building$
inv4: $location \;\in\; user \pfun building$

It is important to observe that invariants *inv*3 and *inv*4 specify constraints between multiple variables (as well as defining the types of the relation variables). For example, included in *inv*3 is the constraint that the domain of *permission* is included in *user*. This means that the system only maintains permission information for registered users. The range of *permission* is constrained to be included in *building* which means that any permissions that a user has can only be for registered buildings. Requirement **FUN3** does not precisely state these two constraints though **FUN3** could be interpreted as requiring that permission is only between registered users and registered buildings. In the mathematical representation, the constraints are specified precisely. Similarly, *inv*4 specifies that only registered users may be located in buildings and those buildings must be registered.

Our access control specification contains two relation variables and we consider whether we can identify an invariant that constrains the connection between these two variables. In fact we have already identified such an invariant in Figure 11 where *location* is required to be included in *permission*. Thus our model has one additional invariant:

invariants
...
inv5: $location \;\subseteq\; permission$

Invariant *inv*5 addresses **FUN4**, the main access control requirement.

The list of invariants for the *BuildingAccess* machine may be classified into three kinds:

1. Constraints between set variables (*inv*1, *inv*2).
2. Constraints between a relational variable and set variables (*inv*3, *inv*4).
3. Constraints between relational variables (*inv*5).

The first two kinds of invariant can often be easily identified from a class diagram derived from the requirements. The class diagram is constructed by identifying the main entities suggested by the requirements (e.g., *USER* and *BUILDING*) and the relevant relationships between them (e.g. *permission* and *location*). The third kind of invariant does not always follow directly from a class diagram and may come directly from the requirements. In Figure 13, because *permission* and *location* have the same source and target, the question of whether one is a subset of the other is suggested.

From the requirements, we have identified carrier sets, variables and invariants but the requirements also suggest events to be included in the Event-B specification. Here we identify a list of events by systematically reviewing each requirement:

- **FUN1** : Suggests *RegisterUser* and *DeRegisterUser* events.
- **FUN2** : *RegisterBuilding* and *DeRegisterBuilding*.
- **FUN3** : *AddPermission* and *RemovePermission*.
- **FUN4** : *EnterBuilding*.
- **FUN5** : *ExitBuilding*.
- **ASM1** : *EnterBuilding*.
- **ASM2** : *EnterBuilding*.

As can be seen with the *EnterBuilding* event, it is sometimes the case that different requirements will give rise to the same event. This is because the requirements may describe different aspects of the behaviour represented by the event. For example, both **FUN4** and **ASM1** put constraints on when a user may enter a building.

8.2 Expansion events

When a specification involves sets, it is common to have events for expanding sets (e.g., *RegisterUser*) and reducing sets (e.g., *DeRegisterUser*). We look at the expansion events first. Events to specify registration of users and buildings are similar to the registration event presented in Section 4.5:

```
RegisterUser  ≙
    any u where
        grd1:  $u \notin user$
    then
        act1:  $user := user \cup \{u\}$
    end
```

```
$RegisterBuilding \;\hat{=}$
    any $b$ where
        grd1:  $u \notin building$
    then
        act1:  $building := building \cup \{u\}$
    end
```

Notice that we omitted a guard specifying that parameter b is an element of $BUILDING$. This is because the type of b can be inferred from $grd1$ since the set $building$ has type $\mathbb{P}(BUILDING)$. Similarly for the $RegisterUser$ event.

The $AddPermission$ event gives a registered user b permission to enter a registered building b by adding the ordered pair $u \mapsto b$ to the $permission$ relation:

```
$AddPermission \;\hat{=}$
    any $u, b$ where
        grd1:  $u \in user$
        grd2:  $b \in building$
    then
        act1:  $permission := permission \cup \{u \mapsto b\}$
    end
```

The guards of this event are required in order to preserve invariant $inv3$ which constrains the domain and range of the $permission$ relation. For example, if $grd1$ was $u \in USER$ instead then the event might violate the invariant by giving permission to a non-registered user.

Here is an alternative version of the event that adds a set of buildings bs to the users permission rather than a single building:

```
$AddMultiPermission \;\hat{=}$
    any $u, bs$ where
        grd1:  $u \in user$
        grd2:  $bs \subseteq building$
    then
        act1:  $permission := permission \cup (\{u\} \times bs)$
    end
```

The expression $\{u\} \times bs$ defines a relation that maps u to each element in bs. The following rules are used to prove that $inv3$ is preserved:

Description	Rule
Product Relation	$s \subseteq S \;\wedge$ $t \subseteq T$ $\Rightarrow \quad s \times t \;\in\; S \leftrightarrow T$
Union Relation	$q \in S \leftrightarrow T \;\wedge$ $r \in S \leftrightarrow T$ $\Rightarrow \quad (q \cup r) \;\in\; S \leftrightarrow T$

The event modelling a user entering a building is parameterised by the entering user and the building they are entering:

```
$EnterBuilding \;\hat{=}$
    any $u, b$ where
        grd1:  $u \notin dom(location)$
        grd2:  $u \mapsto b \in permission$
    then
        act1:  $location := location \cup \{u \mapsto b\}$
    end
```

From **ASM2** we expect that a user is not located in any building when they try to enter a building hence $grd1$ which specifies that u is not in the domain of *location*. From **FUN4**, we identify $grd2$ which specifies that u has permission to enter b. The effect of $act1$ is to add the ordered pair $u \mapsto b$ to *location*.

The invariants that the *EnterBuilding* event affects are $inv4$ and $inv5$. Invariant $inv5$, which specifies that *location* is included in *permission*, is maintained because of $grd2$. Invariant $inv4$ specifies that *location* is functional. Adding a mapping for u to *location* maintains the functionality of *location* because $grd1$ specifies that u is not already mapped to any buildings. Without $grd1$, the event could violate the functionality as u might end up being mapped to more than one building in *location*. The following rule about expanding a partial function captures this property. It states that if f is functional and x is not in the domain of f, then $f \cup \{x \mapsto y\}$ is also functional:

Description	Rule
Function Extension	$f \in S \mathrel{+\!\!\!\!\rightarrow} T \;\wedge$ $x \notin dom(f) \;\wedge$ $x \mapsto y \in S \times T$ $\Rightarrow \quad (f \cup \{x \mapsto y\}) \;\in\; S \mathrel{+\!\!\!\!\rightarrow} T$

8.3 Reduction events and domain subtraction

We have already seen how we can reduce sets using the set difference operator. We can use this to model a user u exiting a building b as follows:

```
$ExitBuilding \;\hat{=}$
    any $u, b$ where
        grd1:   $u \mapsto b \in location$
    then
        act1:   $location := location \setminus \{u \mapsto b\}$
    end
```

Alternatively, we can use an operator on relations called *domain subtraction* to model a user exiting a building. Domain subtraction, written $A ⩤ R$, takes two arguments, a relation a relation $R \in S \leftrightarrow T$ and a set $A \subseteq S$, and removes those pairs from R whose first part is in A. This is illustrated by the following equation which shows the result of domain subtracting a set containing a user from an example of the *location* relation:

$$\{u2\} \;⩤\; \{\; u1 \mapsto b1,\; u2 \mapsto b3,\; u4 \mapsto b3 \;\} \quad = \quad \{\; u1 \mapsto b1,\; u4 \mapsto b3 \;\}$$

Here we see that the mapping from $u2$ to $b3$ is removed to give the reduced set on the right-hand side. The general definition of the operator is as follows:

Name	Predicate	Definition
Domain Subtraction	$x \mapsto y \;\in\; A ⩤ R$	$x \mapsto y \;\in\; R \;\;\wedge\;\; x \notin A$

Here is a specification of the *ExitBuilding* event that has just one parameter, the user u. It removes u from the *location* function using domain subtraction:

```
$ExitBuilding \;\hat{=}$
    any $u$ where
        grd1:   $u \in dom(building)$
    then
        act1:   $location := \{u\} ⩤ location$
    end
```

This event preserves the permission invariant ($inv5$) and the functionality of *location* ($inv3$). This follows from the following rules which show that domain subtraction reduces a relation and that inclusion preserves functionality:

Description	Rule
Domain Subtract Inclusion	$(A ⩤ R) \;\subseteq R$
Inclusion Functional	$f \in S ⇸ T \;\;\wedge$ $g \subseteq f$ $\Rightarrow \quad g \in S ⇸ T$

8.4 Invariant violation

While exiting a building causes no problems from the point of view of invariant preservation, removing permissions can be problematic. Consider the following specification of an event that removes all permissions for a user u using the domain subtraction operation:

```
$RemovePermissions1 \;\hat{=}$
    any $u$ where
        grd1:  $u \in user$
    then
        act1:  $permission := \{u\} ⩤ permission$
    end
```

Here is a reminder of the permission inclusion invariant:

$$\text{inv5:} \quad location \subseteq permission$$

Action *act*1 of the *RemovePermissions1* event results in the following modified version of *inv*5:

$$location \subseteq \underline{\{u\} ⩤ permission}$$

This does not follow from *inv*5. The problem is that we are reducing the right-hand side of this set inclusion without reducing the left-hand side. If the user u was located in a building and we remove all their permissions, then after executing the *RemovePermissions1* event, u would still be in a building but they would no longer have permission to be there, thus violating *inv*5!

8.5 Fixing the violation

One solution to this invariant violation problem would be to add an action to also remove the user from whatever building they are in by removing them from the domain of *location* as well:

```
$RemovePermissions2 \;\hat{=}$
    any $u$ where
        grd1:  $u \in user$
    then
        act1:  $permission := \{u\} ⩤ permission$
        act2:  $location := \{u\} ⩤ location$
    end
```

With this version of the event, the modified invariant becomes:

$$\underline{\{u\} ⩤ location} \subseteq \underline{\{u\} ⩤ permission}$$

This inclusion follows from invariant *inv*5 which means that *RemovePermissions2* preserves *inv*5. Mathematically this is because domain subtraction is *monotonic*. In general, an operation *op* is said to be monotonic when it preserves inclusion between sets:

Name	Definition
Monotonic	$S \subseteq T \;\Rightarrow\; op(S) \;\subseteq\; op(T)$

Description	Rule
Domain Subtract *Monotonic*	$R \subseteq Q \;\Rightarrow\; A ⩤ R \;\subseteq\; A ⩤ Q$

Another way of ensuring that $inv5$ is maintained would be to allow permissions for u to be removed only if they are not currently inside a building. Here is a version that only modifies *permission* but has an additional guard, specifying that the user is not located in any building:

```
$RemovePermissions3 \;\widehat{=}$
    **any** $u$ **where**
        grd1:  $u \in user$
        grd2:  $u \notin dom(location)$
    **then**
        act1:  $permission := \{u\} ⩤ permission$
    **end**
```

To see why this preserves the permission invariant we make use of the following rule; this states that if x is not in the domain of relation R then removing x from the domain of R has no effect:

Description	Rule
Simplify *Domain Subtract*	$x \notin dom(R) \;\Rightarrow\; (\{x\} ⩤ R) = R$

The modified invariant resulting from *RemovePermissions3* is

$$location \;\subseteq\; \underline{\{u\} ⩤ permission}$$

Because of $grd1$ we can assume that $u \notin dom(location)$ and therefore that $location = \{u\} ⩤ location$ so this is the same as the following inclusion:

$$\{u\} ⩤ location \;\subseteq\; \{u\} ⩤ permission$$

As before, this inclusion follows from $inv5$ by monotonicity.

We have seen that the *RemovePermissions1* event does not preserve the permission inclusion invariant ($inv5$) while the other two versions of permission removal, *RemovePermissions2* and *RemovePermissions3*, do preserve the invariant. Clearly we would want to rule out *RemovePermissions1* since it fails to satisfy a mathematical judgement. Although *RemovePermissions2* preserves the permission inclusion invariant, it does combine two separate functions into one atomic event. At the very least it would be more appropriate to reflect the dual role in the name the event, e.g., *ExitAndRemovePermission*. We note that *location* is a *monitoring* variable that is used to keep track of the physical location of users while *permission* is a *conceptual* variable that models a key concept in access control which does not reflect any physical entities. We prefer to keep changes to monitoring variables (e.g., *location*) separate from changes to conceptual variables (e.g., *permission*) so we use *RemovePermissions3* as our specification of permission removal. This choice does mean that permission cannot be removed from a user for a building they are currently located in until after they exit the building. If it was deemed important, we might have a mechanism to force a user to exit a building but we treat this as out of scope of our analysis. Whenever the construction of the Event-B model raises ambiguities about the requirements (such as whether we can remove permissions for a user who is located in a building), then we should consider asking the system provider (the client) to clarify the requirements.

It is ok to remove permission from a user for a particular building even if they are located in another building? Here is an event that does this:

```
$RemoveSinglePermission \;\; \hat{=}$
    **any** $u, b$ **where**
        grd1:  $u \mapsto b \in permission$
        grd2:  $u \mapsto b \notin location$
    **then**
        act1:  $permission := permission \setminus \{u \mapsto b\}$
    **end**
```

Here $grd2$ does not prevent u from being located in some building b' that is different to b. To see why this event preserves $inv5$, consider the effect of the action $act1$ on the invariant:

$$location \;\subseteq\; \underline{permission \setminus \{u \mapsto b\}}$$

Because of $grd2$, $location = location \setminus \{u \mapsto b\}$, so this inclusion is the same as

$$location \setminus \{u \mapsto b\} \;\subseteq\; permission \setminus \{u \mapsto b\}$$

This inclusion follows from $inv5$ by monotonicity of set difference (set difference, set union and set intersection are all monotonic).

8.6 Range subtraction

The domain subtraction operator ($\lhd$) is used to remove pairs from a relation based a domain set argument. There is also a *range subtraction* operator ($\rhd$) that removes pairs based on a range set argument. For example:

$$\{\ u1 \mapsto b1,\ u2 \mapsto b3,\ u4 \mapsto b3\ \} \;\rhd\; \{b1\} \;\;=\;\; \{\ u2 \mapsto b3,\ u4 \mapsto b3\ \}$$

Notice that in the case of domain subtraction, the set argument comes first and the relation comes second ($A \lhd R$) while the arguments are swapped for the range operator ($R \rhd B$). The operator definition is as follows:

Name	Predicate	Definition
Range Subtraction	$x \mapsto y \ \in\ R \mathbin{⩥} B$	$x \mapsto y \ \in\ R \ \ \wedge\ \ y \notin B$

The event to remove a building from the registered buildings makes use of range subtraction to remove all permissions associated with that building:

```
$DeRegisterBuilding \ \ \hat{=}$
    any $b$ where
        grd1:  $b \in building$
        grd2:  $b \notin ran(location)$
    then
        act1:  $building := building \setminus \{b\}$
        act2:  $permission := permission \mathbin{⩥} \{b\}$
    end
```

While b is removed from *building* by action *act*1, action *act*2 removes any permission associated with b from *permission*. This is required in order preserve invariant *inv*3 which specifies that *permission* is a relation between *user* and *building*. The following rules show how the relational subtraction operators reduce the domain/range of a relation:

Description	Rule
Domain/Range Reduction	$R \in S \leftrightarrow T \quad \Rightarrow$ $(A \mathbin{⩤} R) \ \in\ (S \setminus A) \leftrightarrow T$ $(R \mathbin{⩥} B) \ \in\ S \leftrightarrow (T \setminus B)$

Notice that *grd*2 of the *DeRegisterBuilding* event requires that b has no occupants (no users are located in b). This ensures that the event maintains *inv*4 stating that the range of *location* is included in *building* and also maintains the permission inclusion invariant *inv*5 when the permissions for b are removed by *act*2.

The event to de-register a user is specified as follows:

```
$DeRegisterUser \ \ \hat{=}$
    any $b$ where
        grd1:  $u \in user$
        grd2:  $u \notin dom(location)$
    then
        act1:  $user := user \setminus \{u\}$
        act2:  $permission := \{u\} \mathbin{⩤} permission$
    end
```

Guard *grd*2 requires that u is not located in a building (preserving *inv*5), while action *act*2 removes all permissions for u (preserving *inv*3).

For both of the de-register events, we required that the building is unoccupied (for *DeRegisterBuilding*) or the user is not in a building (for *DeRegisterUser*). This was to ensure the preservation of the invariant $inv5$. An alternative way of preserving $inv5$, that does not require guards on *location*, would be to reduce *location* as well, as we saw with *Remove Permission2*. As with permission removal, we prefer to keep location changes and user registration changes as separate events. This is again because *location* is a monitoring variable while the property of being registered is conceptual.

9 Query events

The events we have looked at so far all include actions that change one or more variables. Sometimes we are interested in querying information about a system such as the location of a user. Here is is the specification of such an event:

```
$QueryLocation \;\; \hat{=}$
    **any** $u, result$ **where**
        grd1:  $u \in dom(location)$
        grd2:  $result \in BUILDING$
        grd3:  $result = location(u)$
    **end**
```

The event has two parameters: u, the user whose location is being queried, and *result*, the result of the query. In this case the result of the query is the location of u. The Event-B language does not have an explicit notion of an output parameter. We adopt the convention of naming a parameter representing an output as *result*. Typically the value of a result parameter will be defined by an exact equation such as $grd3$ above. We refer to an event that specifies a result but does not modify any variables as a *query event*.

The guard that specifies the type of the result in *QueryLocation*, $grd2$, is not strictly necessary since the type can be inferred from the equation in $grd3$. However making the type of *result* explicit makes the specification clearer.

Another query we could perform on the access control system would be to find out the set of buildings that a particular user has permission to enter. To do that we use the *relational image* operator. Given a relation $R \in S \leftrightarrow T$ and a set $A \subseteq S$, the expression $R[A]$ represents the set of range elements corresponding to some domain element in A. For example, consider again the following simple relation:

$$\begin{aligned} directory = \{ \; & mary \mapsto 287573, \\ & mary \mapsto 398620, \\ & john \mapsto 829483, \\ & jim \mapsto 398620 \; \} \end{aligned}$$

If we want to identify the set of numbers that *mary* is mapped to, we write $directory[\{mary\}]$ where

$$directory[\{mary\}] \quad = \quad \{ \; 287573, \; 398620 \; \}$$

Note the argument within the brackets must be a set of domain elements rather than a single element which is why we do not write $directory[mary]$.

In general, a range element y is in the relational image of A under R if there is some element x in A that is mapped to y by R. This specified precisely in the following table:

Name	Predicate	Definition
Relational Image	$y \in R[A]$	$\exists x \cdot\ x \in A \ \ \wedge \ \ x \mapsto y \ \in \ R$

Suppose there are no elements of A mapped to range elements by R. In that case there is no x in A satisfying $x \mapsto y \ \in \ R$ and therefore $R[A]$ will be empty.

The event to query the permissions of a user makes use of relational image:

```
$QueryPermissions \ \ \hat{=}$
    **any** $u, result$ **where**
        grd1:  $u \in user$
        grd2:  $result \subseteq BUILDING$
        grd3:  $result = permission[\{u\}]$
    **end**
```

Here the result is specified as the relational image of $\{u\}$ under the permission relation, i.e., the set of buildings for which u has permission. In the case that the user has no permissions, then the result will be the empty set.

We have seen that relational image allows us to specify a query on a relation going from domain elements to range elements. To perform a query in the opposite direction, from range to domain, we can take the *inverse* of a relation, written R^{-1}. The inverse of R is the result of swapping the order of each pair in R. For example, the inverse of the *directory* relation specified above is as follows:

$$\begin{aligned} directory^{-1} = \{ \ & 287573 \mapsto mary, \\ & 398620 \mapsto mary, \\ & 829483 \mapsto john, \\ & 398620 \mapsto jim \ \} \end{aligned}$$

We can use this to query the people associated with phone number 398620 as follows:

$$directory^{-1}[\ \{398620\}\ ] = \{\ mary,\ jim\ \}$$

The inverse operator is defined in the following table:

Name	Predicate	Definition
Relational Inverse	$y \mapsto x \ \in \ R^{-1}$	$x \mapsto y \ \in \ R$

Using inverse and image, a query event that provides the set of users who have permission to enter building b is specified as follows:

```
$QueryBuildingUsers \;\hat{=}$
    any $b, result$ where
        grd1:   $b \in building$
        grd2:   $result \subseteq USER$
        grd3:   $result = permission^{-1}[\{b\}]$
    end
```

We can use a similar query event to provide the set of occupants of a building:

```
$QueryBuildingOccupants \;\hat{=}$
    any $b, result$ where
        grd1:   $b \in building$
        grd2:   $result \subseteq USER$
        grd3:   $result = location^{-1}[\{b\}]$
    end
```

Note that while *location* is functional, the inverse of *location* might not be. This is illustrated by Figure 10 where two different users, $u2$ and $u4$ are located in $b3$. This means that in the inverse relation, $b3$ is mapped to two different users and thus $location^{-1}$ is not functional. If a relation is one-to-one, e.g., Figure 12, then its inverse is also functional. A one-to-one function is also called *injective* and is declared as $f \in S \mathrel{\mkern4mu\mapstochar\mkern-4mu\rightarrowtail} T$. An injective function is defined as a function whose inverse is also functional:

Name	Predicate	Definition
Injective Function	$f \in S \mathrel{\mkern4mu\mapstochar\mkern-4mu\rightarrowtail} T$	$f \in S \mathrel{\mkern4mu\mapstochar\mkern-4mu\rightarrow} T \quad \wedge \quad f^{-1} \in T \mathrel{\mkern4mu\mapstochar\mkern-4mu\rightarrow} S$

9.1 Requirements Tracing

In order to be systematic about validation of the model against the requirements, we will re-visit the list of requirements and annotate each one with a explanation of how it is represented in the Event-B model. This is a form of *tracing* information: a means of tracing from a requirement through to a part, or parts, of the formal model. This is shown as a table in Figure 14 where the explanations of how a requirement is represented in the formal model are in shown in the second column. For example, the annotation on **FUN1** provides an explanation of how that requirement is represented in the formal model through the *user* variable and the *RegisterUser* and *UnRegisterUser* events.

10 Simple Bank

We make use of some of the techniques shown so far to develop a model of a simple banking system. This case study emphasises the use of functions and

Requirement	Representation in model
FUN1	This is represented by the *user* variable modelling registered users (*inv*1) and by the *RegisterUser* and *DeRegisterUser* events.
FUN2	This is represented by the *building* variable modelling registered buildings (*inv*2) and by the *RegisterBuilding* and *DeRegisterBuilding* events.
FUN3	This is represented by the *permissions* variable (*inv*3) and by the *AddPermission* and *RemovePermission* events.
FUN4	This is represented by the permission invariant (*inv*5) and by the *EnterBuilding* event. Guard *grd*2 of the *EnterBuilding* event ensures that the entering user has permission.
FUN5	This is represented by the *ExitBuilding* event. There is no constraint on this event other than the user is located in a building (*grd*1).
ASM1	This is represented by the location being functional (*inv*4).
ASM2	This is represented by *grd*1 of the *EnterBuilding* event which requires that the entering user is not currently located in any building.

Fig. 14. List of requirement labels with tracing information

introduces an additional mathematical concept (function override). The case study also serves to re-enforce the steps that may be taken in developing an Event-B model from a list of requirements including the use of a class diagram to identify the main entities and the relationship between them. Here is a list of functional requirements for the simple bank:

- **FUN1:** The system shall maintain a register of bank **customers** and shall provide operations for managing the customer set.
- **FUN2:** The system shall maintain a **name** and **address** for each customer and shall provide operations for managing these.
- **FUN3:** The system shall allow customers to have several **accounts** and allow customers to share accounts.
- **FUN4:** The system shall provide operations for managing the account set.
- **FUN5:** The system shall maintain a **balance** for each account.
- **FUN6:** The system shall ensure that account balances are never negative.
- **FUN7:** The system shall provide operations for depositing and withdrawing funds to and from an account and for transferring funds between accounts.

10.1 Sets and relations

The first step in developing the model is to identify some carrier sets. In the functional requirements above we have highlighted some nouns in bold, e.g., **customers** in **FUN1**. Requirement **FUN2** introduces names and addresses at a high level and does not define a specific format. We will treat names and addresses as abstract values and model them using carrier sets. These highlighted nouns suggest the carrier sets shown in Figure 15. Note that we only highlighted the first occurrence of a noun to avoid duplication. Figure 15 does not identify a carrier set for **balance** in **FUN5**. For simplicity we decide to model the amount of money in an account as an integer value; we could have chosen more detail such as currency units (e.g., euros) and subunits (e.g., cents). Integers are already part of the Event-B language.

NAME
ACCOUNT
CUSTOMER
ADDRESS

Fig. 15. Carrier sets for simple bank

Figure 15 gives rise to the following context for our simple bank model:

context *BankContext*
sets *ACCOUNT*, *CUSTOMER*, *NAME*, *ADDRESS*
end

Next we identify whether any of the requirements suggest relations between sets. **FUN2** suggests a relation between *CUSTOMER* and *NAME* and between *CUSTOMER* and *ADDRESS*. We name these relations *name* and *address* respectively and they are shown in Figure 16. **FUN3** suggests a relation between *CUSTOMER* and *ACCOUNT* which we name as *accounts*. **FUN5** suggests a relation between *ACCOUNT* and integers which we name as *balance*.

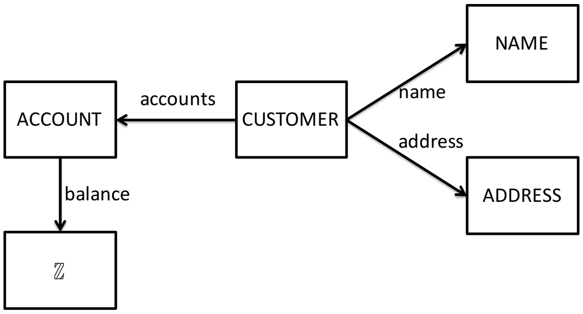

Fig. 16. Adding relations for the bank

Having identified the main sets and required relationships between them, we add a bit more precision to the diagram. For the building access model we distinguished the (fixed) carrier sets for users and buildings from the (variable) registered sets and we do the same for the simple bank. In Figure 17 we have replaced the *CUSTOMER* and *ACCOUNT* carrier sets by the *customer* and *account* variable sets. The variable sets represent registered customers and accounts respectively and they allow us to represent constraints such as requiring customers to be registered in order to have accounts. In order to avoid confusion with the set *account*, we have changed the name of the relation between *customer* and *account* to *cust_acc* in Figure 17. The other way in which we finesse the diagram is to determine the nature of each relation (many-to-many, etc). Since **FUN3** requires that a customers may have multiple accounts, we conclude that *cust_acc* should be a many-to-many relation. From **FUN2** we conclude that each customer has one name and one address hence we conclude that these should be functional. We mark them as *total* functions from *customer* to indicate that each registered customer has a name and an address. Similarly from **FUN5** we conclude that *balance* should be a total function from *account*.

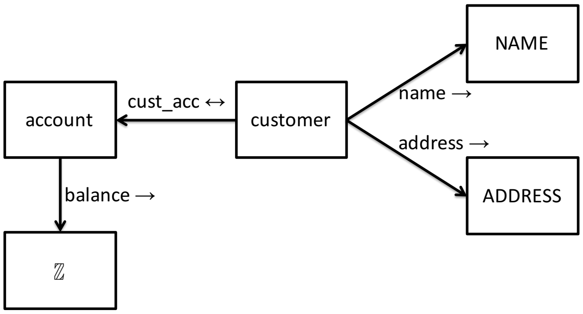

Fig. 17. Finessing the bank model

We have not introduced variable sets corresponding to **NAME** nor **ADDRESS**. The reason is that we regard these sets as *secondary*. By this we mean that we are not interested in the values from these sets in their own right and we are only interested in them as attributes of the other sets. We refer to the non-secondary sets (*customer*, *account*) as *primary*. One indicator of a secondary set is that it has no outgoing arrows in the class diagram, and only has incoming arrows. This is the case for the sets *NAME*, *ADDRESS* and indeed ℤ in Figure 17. However this is not a hard-and-fast rule: the *building* set of Figure 13 has no out-going arrows but we still treat it as a primary set since the requirements explicitly stated that a set of registered buildings should be maintained. In the simple bank there is no requirement to maintain a set of registered addresses and names independent of the customer to which they belong. Neither is there a requirement to maintain a set of balance values independent of the accounts to which they belong.

Construction of the class diagram of Figure 17 allows to identify the set and relation variables. The primary sets *customer* and *account* become variable sets while the relations *name*, *address*, *cust_acc* and*balance* become relation variables:

machine *Bank*
sees *BankContext*
variables *customer*, *account*, *name*, *address*, *cust_acc*, *balance*
invariants
inv1: $\mathit{customer} \subseteq \mathit{CUSTOMER}$
inv2: $\mathit{account} \subseteq \mathit{ACCOUNT}$
inv3: $\mathit{name} \in \mathit{customer} \rightarrow \mathit{NAME}$
inv4: $\mathit{address} \in \mathit{customer} \rightarrow \mathit{ADDRESS}$
inv5: $\mathit{cust\_acc} \in \mathit{customer} \leftrightarrow \mathit{account}$
inv6: $\mathit{balance} \in \mathit{account} \rightarrow \mathbb{Z}$

An advantage of having the variable set *customer* in the model is that it allows us to specify that the functions *name* and *address* have exactly the same domain. All of the above invariants are derived directly from Figure 17 (which in turn was derived from the requirements via the other two class diagrams).

We study the requirements again to check if there are any further invariants we could identify. The only requirement from which we can identify a further invariant is **FUN6** which states that account balances are never negative (a rather conservative requirement for a bank!). We can represent this requirement by strengthening $inv6$ to specify that the range of *balance* is the set of naturals rather than integers (naturals are written $\mathbb{N}$ and represent all the non-negative integers, i.e., those $n \in \mathbb{Z}$ where $n \geq 0$):

invariants
...
inv6b: $\mathit{balance} \in \mathit{account} \rightarrow \mathbb{N}$

An alternative formulation of *FUN5* is to specify that the balance of each account is non-negative using universal quantification:

invariants
...
inv6: $\mathit{balance} \in \mathit{account} \rightarrow \mathbb{Z}$
inv7: $\forall a \cdot a \in \mathit{account} \Rightarrow \mathit{balance}(a) \geq 0$

In $inv7$ we restrict the quantification to those a in the set *account*. Since *balance* is total on the set *account*, the expression $\mathit{balance}(a)$ is guaranteed to be well-defined.

We are not yet done with identifying invariants. Although we might not be able to identify this explicitly from the requirements, we need to be careful about the domain and range of the *cust_acc* relation. Invariant $inv5$ specifies that the domain of *cust_acc* is a subset of *customer* but does not specify that the domain is equal to *customer*. This means we may have customers who have no accounts associated with them. Similarly *inv*5 allows for register accounts that have no customers associated with them. The requirements are not clear on this and we now have the opportunity to be more precise.

We decide that we may have a customer who has no accounts, e.g, this might arise when we register a customer before we create any accounts for them. Thus $dom(cust\_acc)$ does not need to equal *customer* and can be a subset. However, we decide that it is not ok to have an account that has no customers associated with it. We introduce a further invariant to specify that every account is associated with some customer:

```
invariants
    ...
    inv8:   $ran(cust\_acc) = account$
```

The combination of *inv*5 and *inv*8 means that each customer has *zero* or more accounts, while each account has *one* or more customers.

10.2 Expansion events

We introduced a distinction between primary and secondary sets and we identified that *customer* and *account* as the primary sets in Figure 17. It is for the primary sets that we introduce expansion events (events that expand some set of elements). The *customer* set is expanded by the event for registering a new customer:

```
$RegisterCustomer \;\hat{=}$
    any $c, n, a$ where
        grd1:  $c \notin customer$
        grd2:  $n \in NAME$
        grd3:  $a \in ADDRESS$
    then
        act1:  $customer := customer \cup \{c\}$
        act2:  $name := name \cup \{c \mapsto n\}$
        act3:  $address := address \cup \{c \mapsto a\}$
    end
```

Similar to registration of new users in the building access example, the new customer c is represented by a ‘fresh’ value (*grd*1). Since we are expanding *customer* (*act*1), and since *name* and *address* are total functions on *customer*, we also need to expand *name* and *address* (*act*2 and *act*3) . The values for the name and address of the new customer are provided as parameters n and a.

The following rule shows how extending a total function maintains functionality. It shows that the extended function $(f \cup \{x \mapsto y\})$ is total on an expanded domain $(S \cup \{x\})$.

Description	Rule
Total *Function* *Extension*	$f \in S \rightarrow T \ \wedge$ $x \notin S \ \ \wedge \ \ y \in T$ $\Rightarrow \ \ (f \cup \{x \mapsto y\}) \ \in \ (S \cup \{x\}) \rightarrow T$

Our *RegisterCustomer* event does not associate any accounts to the new customer. This does not violate any invariants since we concluded that a customer may have zero or more accounts. Since we also concluded that an account must have at least one associated customer (*inv*8), we need to associate at least one customer with a newly created account. We chose to associate a set of customers with a newly created account and this set will need to be non-empty. Since *balance* is total on *account*, we also need to associate a balance value with the newly created account; we will set the balance to be zero. We specify the event as follows:

```
$CreateAccount \ \ \hat{=}$
    any $a, cs$ where
        grd1:  $a \notin account$
        grd2:  $cs \subseteq customer$
        grd3:  $cs \neq \varnothing$
    then
        act1:  $account := account \cup \{a\}$
        act2:  $cust\_acc := cust\_acc \cup (cs \times \{a\})$
        act3:  $balance := balance \cup \{a \mapsto 0\}$
    end
```

A note on the naming of these events: we used 'register' to name expansion event for customers (*RegisterCustomer*) while we used 'create' for accounts (*CreateAccount*). The reason is that values in *customer* correspond to entities that are external to the bank while accounts are entities that are internal to the bank. To use our previously-introduced terminology, *customer* is a monitoring variable while *account* is a conceptual variable. Of course naming is matter of taste and judgement. Our distinction between registration and creation is simply a guideline.

The above expansion events contribute to addressing the requirements for 'managing' the set of accounts (**FUN1**) and the set of accounts (**FUN4**). Both these requirements also suggest reduction events for the primary sets, e.g., *DeRegisterCustomer* and *DeleteAccount*. **FUN4** also suggests events for expanding and reducing the set of customers associated with an account, e.g., *AddAccountCustomer* and *RemoveAccountCustomer*. We leave the specification of these to the reader. As before, care must be taken to ensure that all the invariants are preserved by these reduction events.

10.3 Function override

Requirement **FUN2** suggests events for modifying the address of a customer and possibly even the name of a customer. **FUN7** suggests events for increasing and decreasing the balance of an account and for transferring funds between accounts. Specification of all of these involve modifying a function so that the range value that some domain element is mapped to is updated, e.g., to withdraw money from account a, the *balance* function gets updated so that the value associated with a is changed to a smaller value. To represent function update mathematically we use the *function override* operator.

We illustrate the use of this operator with an example first. Assume that the *balance* function has the following value:

$$\mathit{balance} \quad = \quad \{\ a1 \mapsto 100,\ a2 \mapsto 350,\ a3 \mapsto 800,\ a4 \mapsto 50\ \ \}$$

If we want to change the balance of account $a2$ to 300, we use function override ($\mathbin{\lhd\mkern-12mu-}$) with *balance* as the first argument and a mapping from $a2$ to 300 as the second argument, written $\mathit{balance} \mathbin{\lhd\mkern-12mu-} \{\ a2 \mapsto 300\}$. The following equation shows the result of this overriding:

$$\mathit{balance} \mathbin{\lhd\mkern-12mu-} \{\ a2 \mapsto 300\} \quad = \quad \{\ a1 \mapsto 100,\ a2 \mapsto \underline{300},\ a3 \mapsto 800,\ a4 \mapsto 50\ \ \}$$

As highlighted in the resulting function on the right-hand side, 350 has been replaced by 300. Function override is a combination of domain subtraction and set union, i.e., $f \mathbin{\lhd\mkern-12mu-} \{a \mapsto b\}$ is the same as removing the existing mapping for a from f using domain subtraction and adding the updated mapping using union:

$$f \mathbin{\lhd\mkern-12mu-} \{a \mapsto b\} \quad = \quad (\{a\} \mathbin{⩤} f) \cup \{a \mapsto b\}$$

More generally, the second argument for function override is itself a function, $f \mathbin{\lhd\mkern-12mu-} g$, rather than just a singleton mapping $f \mathbin{\lhd\mkern-12mu-} \{a \mapsto b\}$. The general definition also uses domain subtraction and set union as shown in the following table:

Name	Expression	Definition
Function override	$f \mathbin{\lhd\mkern-12mu-} g$	$(\mathit{dom}(g) \mathbin{⩤} f) \cup g$

The specification of the event for depositing money in an account, *IncreaseBalance*, uses function override to update the value of balance:

```
$IncreaseBalance \;\hat{=}$
    **any** $a, m$ **where**
        grd1:  $a \in \mathit{account}$
        grd2:  $m > 0$
    **then**
        act1:  $\mathit{balance} := \mathit{balance} \mathbin{\lhd\mkern-12mu-} \{a \mapsto \mathit{balance}(a) + m\}$
    **end**
```

Here m is the amount to be deposited in account a. We require m to be greater than zero since adding zero would seem rather pointless ($grd2$). In $act1$ the balance of account a is updated to the value $balance(a) + m$.

It is worth noting the difference between extending a function using union and updating a function using function override. Function extension is used when adding a new value to the domain, e.g., expanding the domain of *balance* when creating an account. Function override is used when modifying the range value associated with an existing domain element, e.g., modifying the *balance* of an existing account when depositing money.

The following rules about function override support mathematical reasoning. The first rule shows the conditions under which a function override $(f \mathbin{<\!\!\!+} \{x \mapsto y\})$ remains a total function. The second rule shows that the result of applying a function override $(f \mathbin{<\!\!\!+} \{x \mapsto y\})$ to domain value w depends on whether w is the same as or different to x:

Description	Rule
Total Function Update	$f \in S \rightarrow T \;\wedge$ $x \in S \;\wedge\; y \in T$ $\Rightarrow\; (f \mathbin{<\!\!\!+} \{x \mapsto y\}) \;\in\; S \rightarrow T$
Apply Function Update	$f \in S \rightarrow T \;\wedge\; w \in S \;\Rightarrow$ $w = x \;\Rightarrow\; (f \mathbin{<\!\!\!+} \{x \mapsto y\})(w) = y$ $w \neq x \;\Rightarrow\; (f \mathbin{<\!\!\!+} \{x \mapsto y\})(w) = f(w)$

The shape of action $act1$ in *IncreaseBalance* is a common one when updating functions at a single domain point, i.e., it has the form $f := f \mathbin{<\!\!\!+} \{x \mapsto E\}$. Because update of a function at a single point is a common action in Event-B, it may be written in a simple syntactic form $f(x) := E$. This syntactic form is defined by the following table:

Name	Action	Definition
Function single assignment	$f(x) := E$	$f := f \mathbin{<\!\!\!+} \{x \mapsto E\}$

Using this form, the *IncreaseBalance* event is specified as follows:

```
$IncreaseBalance \;\hat{=}$
  any $a, m$ where
    grd1:  $a \in account$
    grd2:  $m > 0$
  then
    act1:  $balance(a) := balance(a) + m$
  end
```

The *DecreaseBalance* event is specified in a similar way with the balance being decreased:

```
DecreaseBalance  ≙
    any a, m where
        grd1:  $a \in account$
        grd2:  $m > 0$
        grd3:  $m \leq balance(a)$
    then
        act1:  $balance(a) := balance(a) - m$
    end
```

With this event, the amount to be withdrawn should not exceed the current balance of the account ($grd3$). This is to ensure that the balance does not go negative ($inv7$). Let us reason about this more precisely. Recall that $inv7$ is a quantification over accounts as follows:

$$\text{inv7:} \quad \forall a \cdot a \in account \Rightarrow balance(a) \geq 0$$

Action $act1$ of *DecreaseBalance* is equivalent to assigning an overridden function to *balance* and so gives rise to the following modified invariant:

$$\forall a' \cdot a' \in account \;\Rightarrow\; \underline{(balance \mathbin{\lhd\!\!\!-} \{a \mapsto balance(a) - m\})}(a') \geq 0 \tag{7}$$

Note that here we have renamed the quantified variable a to a'. This is to avoid a name clash with the event parameter a. The *Apply Function Update* rule shown above suggests that we reason about (7) by considering two cases: $a = a'$ and $a \neq a'$.

In the case that $a = a'$, (7) is simplified by the *Apply Function Update* rule to the following:

$$\forall a' \cdot a' \in account \land a' = a \;\Rightarrow\; balance(a) - m \geq 0 \tag{8}$$

This is equivalent to $balance(a) \geq m$ which follows from $grd3$ of *DecreaseBalance.*

In the case that $a \neq a'$, (7) is simplified to the following:

$$\forall a' \cdot a' \in account \land a' \neq a \;\Rightarrow\; balance(a') \geq 0 \tag{9}$$

This follows from $inv7$.

Requirement **FUN7** requires an event for transferring money from one account, a, to another account, b. This is specified as follows, and as with the *DecreaseBalance* event, requires that the amount to be transferred does not exceed the balance of the source account a:

```
$TransferBalance \ \hat{=}$
    any $a, b, m$ where
        grd1:  $a \in account$
        grd2:  $b \in account$
        grd3:  $a \neq b$
        grd4:  $m > 0$
        grd5:  $m \leq balance(a)$
    then
        act1:  $balance := balance \mathbin{<\!\!\!+} \{ \ a \mapsto balance(a) - m,$
                                          $b \mapsto balance(b) + m \ \}$
    end
```

Note that *grd*3 requires that the source and target accounts are distinct (to avoid pointless transfers). The action *act*1 uses function override to update the two account balances simultaneously. We might be tempted to write the actions of *TransferBalance* as two single updates as follows:

```
act1:  $balance(a) := balance(a) - m$
act2:  $balance(b) := balance(b) + m$
```

This is syntactically invalid in Event-B as it involves two actions assigning to the same variable in a single event and so we avoid this form.

We can introduce event parameters representing values local to the event to increase the readability of the specification of *TransferBalance*:

```
$TransferBalance \ \hat{=}$
    any $a, b, m, na, nb$ where
        grd1:  $a \in account$
        grd2:  $b \in account$
        grd3:  $a \neq b$
        grd4:  $m > 0$
        grd5:  $m \leq balance(a)$
        grd6:  $na = balance(a) - m$
        grd7:  $nb = balance(b) + m$
    then
        act1:  $balance := balance \mathbin{<\!\!\!+} \{ \ a \mapsto na, \ b \mapsto nb \ \}$
    end
```

Here *na* and *nb* represent the new balances of *a* and *b* respectively whose values are defined by *grd*6 and *grd*7.

Events to update the name or address of a customer can also be specified using function override. We leave these to the reader. The requirements do not explicitly mention queries on the bank data such as the balance of an account or the customers associated with an account. We leave these for the reader to specify.

11 Model validation through animation

A very useful validation technique for Event-B models is to use an animation tool such as ProB [4] or AnimB[2]. With these tools, the carrier sets are instantiated with some illustrative values, e.g., the carrier set *USER* is instantiated with the values *u1*, *u2*, *u3*, and the model can be executed with these values. The execution is driven by the modeller and at each step the state can be inspected. For the purposes of animating our access control model, let us assume that the carrier set *USER* is instantiated with the values $u1, u2, u3$ and that $BUILDING$ is instantiated with the values $b1, b2, b3$. Figure 18 represents the state that is reached by executing the following sequence of events on our model of the building access control system:

$$\begin{array}{l} initialisation \\ RegisterBuilding(b1) \\ RegisterBuilding(b2) \\ RegisterUser(u1) \\ AddPermission(u1, b1) \\ EnterBuilding(u1, b1) \end{array}$$

Figure 18 shows the values of the sets *user* and *building* and the relations *permission* and *location* as tables. The *user* and *building* tables show that there is one registered user and two registered buildings. The *permission* table shows that $u1$ has permission to enter $b1$ while the *location* table shows that $u1$ is located in building $b1$. These values for the variables are what we would expect to see after execution of the given sequence of events. Figure 18 also shows the events that are enabled in the reached state. We see that two more users $(u2, u3)$ and one more building $(b3)$ can be registered. At the bottom of the list of enabled events we see that user $u1$ may leave building $b1$. We can see that the *EnterBuilding* event is not in the list of enabled events. This is as expected since the only registered user, $u1$, is currently in building $b1$ and there is no means to directly enter one building from another.

The value of the animation is that it helps us make human judgements about whether the behaviour specified by the model is what we would expect given the informal requirements. In this case we can make a judgement that the values of the tables correspond to what we would expect after the given sequence of event executions is performed. Inspecting the enabled events allows to check that the guards of the events are sufficiently strong, e.g., the fact that *EnterBuilding* is not in the list of enabled events in Figure 18 helps us to validate the guards specified for that event.

The event sequence above registers two buildings and one user. Here is a second event sequence that continues from the first, adding a second user $u2$,

[2] www.animb.org

u1

user

b1, b2

building

u1	*b1*

permission

u1	*b1*

location

Enabled Events
$RegisterUser(u2), RegisterUser(u3)$
$DeRegisterUser(u1)$
$RegisterBuilding(b3)$
$DeRegisterBuilding(b1), DeRegisterBuilding(b2)$
$AddPermission(u1, b2)$
$RemovePermissions(u1)$
$ExitBuilding(u1)$

Fig. 18. Result of animating model through first sequence of events

u1, u2

user

b1, b2

building

u1	*b1*
u2	*b1*
u2	*b2*

permission

u2	*b2*

location

Enabled Events
$RegisterUser(u3)$
$DeRegisterUser(u1), DeRegisterUser(u2)$
$RegisterBuilding(b3)$
$DeRegisterBuilding(b1), DeRegisterBuilding(b2)$
$AddPermission(u1, b2)$
$RemovePermissions(u1), RemovePermissions(u2)$
$EnterBuilding(u1, b1)$
$ExitBuilding(u2)$

Fig. 19. Result of animating model through second sequence of events

giving that user permissions, entering $u2$ in building $b2$ and exiting user $u1$:

$$\begin{array}{l}
RegisterUser(u2) \\
AddPermission(u2, b1) \\
AddPermission(u2, b2) \\
EnterBuilding(u2, b2) \\
ExitBuilding(u1)
\end{array}$$

The state resulting from continuing from the state of Figure 18 is shown in Figure 19. In this figure, $u2$ has been added to *user*, two rows have been added to *permission* and the *location* table has been updated. We see that an animation tool allows us to execute sequences of events on sample data values and inspect the effect of these on a representation of the state of a machine and on the enabledness of events.

user
u1, u2

building
b1, b2

permission	
u1	*b1*

location	
u2	*b2*

Fig. 20. Incorrect state reached when *RemovePermissions1(u2)* is applied to state in Figure 19

12 Model verification

Manual inspection of the tables in Figures 18 and 19 shows that they both represent states satisfying invariants $inv1$ to $inv5$. However, rather than using manual inspection to check for satisfaction of invariants, model verification can be used to do this in a systematic and automated way. Model verification involves making mathematical judgements about the model. The main mathematical judgement we apply to the abstract model is to determine whether the invariants are guaranteed to be maintained by the events. Mathematical judgements are formulated as *proof obligations* (PO). These are mathematical theorems whose proof we attempt to discharge using a deductive proof system. In the Rodin toolset [5] for Event-B, mechanical proof of POs may be complemented by the use of the ProB model checker which searches for invariant violations by exploring the reachable states of a model.

Previously we argued that the *RemovePermissions1* event could violate the permission inclusion invariant ($inv5$). Let us see how this plays out in animation of the model. Consider the state of the access control system shown in Figure 19. As already explained, this state is reachable by executing a particular sequence of events. In this state, $u2$ is in building $b2$ and has permission to be there. Now if the next event to be performed was $RemovePermissions1(u2)$, the state reached would be as shown in Figure 20. This new state is an *incorrect* state, that is, it violates the permission inclusion invariant since user $u2$ is still in building $b2$ even though $u2$ not longer has permission to be there. Indeed, ProB can automatically find a sequence of events that lead to an invariant violation (known as a *counterexample*). The counterexample that leads to the state in Figure 20 is not the shortest possible counterexample. ProB can automatically find a shorter counterexample that leads to violation of the permission inclusion invariant such as the following:

$$\begin{array}{l} initialisation \\ RegisterBuilding(b1) \\ RegisterUser(u1) \\ AddPermission(u1, b1) \\ EnterBuilding(u1, b1) \\ RemovePermissions1(u1) \end{array}$$

We look at how the error is reflected in the proof obligation (PO) for invariant preservation. Figure 21 shows a definition of this PO. The left side of the figure provides a schematic specification of an event E with a guard represented

```
$E \ \hat{=}$
    any $p$ where
        @$grd$   $G(p,v)$
    then
        @$act$   $v := F(p,v)$
    end
```

Invariant Preservation PO:

$$\begin{array}{ll} \mathsf{Hyp1:} & I(v) \\ \mathsf{Hyp2:} & G(p,v) \\ \mathsf{Goal:} & I(\ F(p,v)\ ) \end{array}$$

Fig. 21. Invariant preservation proof obligation for an event

by $G(p,v)$ and an action represented by $F(p,v)$. Here p represents the event parameters and v represents the variables on the machine on which the event operates. We write $G(p,v)$ to indicate that p and v are free variables of the predicate G. Assuming that $I(v)$ represents a invariant of the machine, the right hand side of Figure 21 shows the PO used to prove that the invariant is maintained by event E. The PO is in the form of a list of hypotheses and a goal. The PO is discharged by proving that the goal is true assuming that the hypotheses are true. In this case, the hypotheses are the invariant itself (Hyp1) and the guard of the event (Hyp2). The goal is the invariant with the free occurrences of variable v replaced by $F(p,v)$, the value assigned to v by the action of the event.

The Rodin tool for Event-B generates the invariant preservation POs for all of the events of the access control model and the automated provers of Rodin are able to discharge all of the generated POs except for one: the specification of the *RemovePermissions1* event together with invariant *inv*5 give rise to the following PO that cannot be proved:

$$\begin{array}{ll} \mathsf{Hyp1:} & location \ \subseteq \ permission \\ \mathsf{Hyp2:} & u \in user \\ \mathsf{Goal:} & location \ \subseteq \ \underline{\{u\} \mathbin{⩤} permission} \end{array}$$

Here, Hyp1 is the invariant to be preserved and Hyp2 is the guard of the event. The event makes an assignment to the *permission* variable and thus the goal is formed by substituting *permission* by $\{u\} \mathbin{⩤} permission$. The result of the substitution is underlined in the goal. The problem here is that the right-hand side of the set inequality in the goal, $\{u\} \mathbin{⩤} permission$, is reduced compared with that in the hypothesis, Hyp1, while the left-had side, *location*, remains unchanged (as discussed in Section 8.4).

To address this problem with the *RemovePermissions1* event, we provided two alternative specifications of permission removal. For example, the specification of the *RemovePermissions3* event together with *inv*5 gives rise to the following PO that can be proved beause of the additional hypothesis provided by the additional guard:

$$\begin{array}{ll} \mathsf{Hyp1:} & location \ \subseteq \ permission \\ \mathsf{Hyp2a:} & u \in user \\ \mathsf{Hyp2b:} & u \notin dom(location) \\ \mathsf{Goal:} & location \ \subseteq \ \{u\} \mathbin{⩤} permission \end{array}$$

The counterexample generated by the ProB model checker highlighted a problem with the specification of the *RemovePermissions1* event. This stronger condition for removing permission was identified through our attempt to prove that the original specification of the event maintained the permission inclusion invariant, leading to *RemovePermission3*. It is appropriate that we make a (human) judgement about the validity of this stronger specification of removing authorisation. Is it a reasonable constraint? Well, if we expect the access control policy to hold always, we have no choice: without the stronger guard, the event cannot maintain the permission inclusion invariant. We could remove the invariant completely from the model but that seems like an unsatisfactory solution since it would mean we were not addressing the main purpose of access control in our formalisation and would undermine what we can reasonably state in our requirements. For the purposes of this paper, we make the judgement that the invariant should stay and thus the revised version of the event, *RemovePermission3*, with the stronger guards holds.

13 Further Reading

Refinement is a key concept in Event-B and is used for structuring complex specifications and for relating abstract models with more concrete, implementation-oriented models. We have not covered refinement in this paper because of space limitations. For a comprehensive introduction to modelling, refinement and proof in Event-B see Abrial's book on the topic [1]. For an overview of the role and practice of refinement in Event-B see [6]. Event-B evolved from the B Method which was also developed by Abrial [7]. The B Method was developed to model and reason about software systems and has module structuring mechanisms similar to modular programming languages. Event-B was designed to reason about systems that may include hardware and physical components as well as software. Some component-based structuring mechanisms for Event-B are described in [8].

In Section 4.6 we saw that the choice of value for a parameter is treated as nondeterministic: any value that satisfies the guards may be chosen. In Event-B, it is also possible to specify nondeterministic actions of the following form [1]:

$$v \;:=\; v' \mid P(v, v')$$

Here $P(v, v')$ is a predicate that describes a relation between the before and after values of variable v. The nondeterministic action states that v should be assigned a new value v' such that $P(v, v')$ holds. For example, assuming that x is an integer variable, then the following action increases x by a nondeterministic amount:

$$x \;:=\; x' \mid x < x'$$

Nondeterministic actions have a *feasibility* proof obligation which requires that there exists some value v' satisfying $P(v, v')$ when the invariant and event guards hold [1]. In this paper, we only made use of deterministic actions and used the choice of parameter values to represent nondeterminism within an event.

Our reason for using this style is that it allows the nondeterministically chosen value to be available across all of the actions of an event. For example, in the *RegisterUser2* event in Section 4.5, the parameter u is used in both actions so that the same nondeterministically chosen value for u is added to *register* and to *out*.

The mathematical language of Event-B (logic and set theory) is similar to the mathematical language of the B Method. These in turn were influenced by the Z notation [9] and VDM [10]. The use of class diagrams to aid the construction of Event-B models, as used in this paper, was inspired by the UML-B notation which provides a graphical syntax for parts of Event-B [11].

For more information on the Rodin tool see [5]. The Rodin tool can be downloaded via the Event-B.org[3] website which also contains a Rodin User Manual[4]. The Atelier B tool[5] supports the B Method. For details of the ProB tool see [4] and the ProB website[6]. ProB is available as a plug-in for Rodin as is the AnimB tool[7].

14 Concluding

This paper provided an overview of how the Event-B language and verification method can be used to model and reason about system behaviour. Reasoning about the system is not just about proving invariant properties. Several different forms of reasoning were deployed in addition to mathematical reasoning: identification of the main purpose of a system, abstraction from design details in requirements, identification of the various entities in the system and their relationships – all of these are forms of reasoning. Constructing the formal model based on the requirements is another form of reasoning as is validation of the model against the requirements through human judgement. All these forms of reasoning complement each other in helping us to understand the purpose of a system and the constraints on the system.

Event-B encourages us to identify the main entities of the problem under consideration and the relationships between those entities. It also encourages us to identify what properties should always hold (invariants), under what conditions system transitions are allowed (guards) and the effect of those transitions on the system state (actions). We have seen how the mathematical structures chosen can encourage us to identify different kinds of events such as set expansion events, set reduction events and query events.

This paper emphasised mathematical reasoning as this is a particular strength of a specification language such as Event-B. The paper presented definitions and rules in order to help the reader gain a strong understanding of the mathematical

[3] `www.event-b.org`

[4] `www3.hhu.de/stups/handbook/rodin/current/html/index.html`

[5] `www.atelierb.eu/en/outil-atelier-b/`

[6] `www3.hhu.de/stups/prob/index.php/The_ProB_Animator_and_Model_Checker`

[7] `www.animb.org`

operators and their properties. Understanding the properties of the mathematical operators helps ensure that we are choosing the appropriate operators in order to specify an intended effect. It allows us to check that the mathematics is being used in an appropriate way, both from a validation point of view (is the specification meeting the requirements?) and a correctness point of view (is the specification maintaining invariants?).

Many of the invariants used in this paper were in the form of equations ($E = F$) and inclusions ($E \subseteq F$). Typically the actions of an event modify one or both sides of an equation or inclusion. We used two main ways of preserving the equations and inclusions: either adding sufficient actions to ensure both sides of an equation or inclusion are modified in similar ways or using guards and properties of the operators to verify that modifying only one side still preserves the equation or inclusion.

We quoted Boehm's First Law in the introduction. Let us quote Boehm's Second Law [2] in the conclusion:

> **Boehm's Second Law:** *Prototyping significantly reduces requirements and design errors, especially for user errors.*

We would argue that a formal model in a language such as Event-B acts as a form of early prototype allowing us to uncover and fix errors in requirements. Of course, while formal modelling addresses the key concepts in the problem being solved by a software system, it does not deal with the important issue of user interfaces (which can cause the user errors referred to in Boehm's Second Law); a software prototype remains an important tool in uncovering requirements on user interfaces. Formal modelling and reasoning help to uncover conceptual errors in requirements while software prototypes help uncover user interface errors.

We conclude by summarising some key messages:

- The role of problem abstraction and formal modelling is to increase understanding of a problem leading to good quality requirements and design documents with low error rates.
- The role of model validation is to ensure that formal models adequately represent the intended behaviour of a system.
- The role of model verification is to improve the quality of models through formulation of invariants and reasoning about those invariants, including rectifying specifications where appropriate.

References

1. Abrial, J.R.: Modeling in Event-B: System and Software Engineering. Cambridge University Press (2010)
2. Boehm, B.W.: Software Engineering Economics. 1st edn. Prentice Hall PTR, Upper Saddle River, NJ, USA (1981)
3. Feiler, P., Goodenough, J., Gurfinkel, A., Weinstock, C., Wrage, L.: Four pillars for improving the quality of safety-critical software-reliant systems. Technical report, Software Engineering Institute, Carnegie-Mellon University (2013) `https://resources.sei.cmu.edu/asset_files/WhitePaper/2013_019_001_47803.pdf`.

4. Leuschel, M., Butler, M.: ProB: An automated analysis toolset for the B Method. Intl. J. on Software Tools for Technology Transfer **10**(2) (2008) 185–203 `http://eprints.soton.ac.uk/262886/`.
5. Abrial, J.R., Butler, M., Hallerstede, S., Hoang, T., Mehta, F., Voisin, L.: Rodin: an open toolset for modelling and reasoning in Event-B. STTT **12**(6) (2010) 447–466 `http://dx.doi.org/10.1007/s10009-010-0145-y`.
6. Butler, M.: Mastering system analysis and design through abstraction and refinement. In: Engineering Dependable Software Systems, IOS Press (2013) `http://eprints.soton.ac.uk/349769/`.
7. Abrial, J.R.: Modeling in Event-B: System and Software Engineering. Cambridge University Press (1996)
8. Silva, R., Pascal, C., Hoang, T., Butler, M.: Decomposition tool for Event-B. Software: Practice and Experience **41**(2) (February 2011) 199–208 `http://www.eprints.soton.ac.uk/271714/`.
9. Woodcock, J., Davies, J.: Using Z - Specification, Refinement, and Proof. Prentice-Hall (1996) `http://www.usingz.com`.
10. Jones, C.: Systematic Software Development using VDM. Prentice Hall (1990)
11. Snook, C., Butler, M.: UML-B: Formal modelling and design aided by UML. ACM Transactions on Software Engineering and Methodology **15**(1) (January 2006) 92–122 `http://eprints.soton.ac.uk/260169/`.