Mastering System Analysis and Design through Abstraction and Refinement

Michael Butler
Electronics and Computer Science
University of Southampton, UK
mjb@ecs.soton.ac.uk

Abstract. The complexity of requirements and complexity of operating environments make error detection in early stages of software system development difficult. This paper makes an argument for the use of formal modelling and verification in early stages of system development to identify and eliminate errors in a timely fashion. Precision is key to eliminating errors in requirements while abstraction is key to mastering requirements complexity. The paper outlines the way in which precision and abstraction may be achieved through modelling and how refinement allows the complexity to be managed through layering. The role of model validation and model verification in improving the quality of formal models and in improving the quality of the requirements is also outlined. The formalism used throughout is Event-B supported by the Rodin toolset.

Keywords. Abstraction, refinement, validation, verification, Event-B, Rodin, ProB

Introduction

This paper addresses the key role played by formal modelling and verification in software systems engineering. Modelling may be used at all stages of the development process from requirements analysis to system acceptance testing. Formal modelling and verification lead to deeper understanding and higher consistency of specification and design than informal or semi-formal methods. In order to manage system complexity, abstraction and refinement are key methods for structuring the formal modelling effort since they support separation of concerns and layered reasoning. A refinement approach means that models represent different abstraction levels of system design; consistency between abstraction levels is ensured by formal verification.

We use the Event-B formal modelling language developed by Abrial [1] and the associated Rodin[1] toolset for Event-B [2]. Event-B is a state-based formal method for system-level modelling and analysis. The key features of Event-B are the use of set theory as a modelling notation, the use of refinement to represent systems at different abstraction levels and the use of mathematical proof to verify the consistency between refinement levels.

We start by motivating the need for abstraction and formal modelling. We then provide insight into the abstraction process through an illustrative example of a secure ac-

[1]Available from www.event-b.org

cess control system. We show how our abstraction of the system may be formalised in Event-B and the role played by formal verification (both model checking and automated proof) in identifying errors in the model. We also outline the process of validation of a formal model against informal requirements. The abstraction of the access control system is focused on *what* the system achieves (its purpose) rather than *how* it is achieved. We then show how features that were ignored in our abstraction may be introduced to the formalisation in a layered and consistent way through refinement. Not only does refinement specify *how* the intended purpose is achieved, but the effort required to verify the consistency of the refinement w.r.t. its abstraction provides insight into *why* the mechanism used works.

1. Motivation

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. It has long been recognised that the cost of fixing a specification or design error increases the later in the development that error is identified. The diagram in Figure 1 illustrates the results of a survey from 1990 of industrial projects providing evidence of this phenomenon [12]. Based on data from these projects it shows the cost of fixing requirements errors (errors that are introduced in the requirements analysis phase) at various stages of the development cycle. The graph shows that, for the projects reported on, the cost of fixing requirements errors discovered in the acceptance testing phase is between 30 and 70 times that of the cost of fixing errors discovered in the requirements phase; that scaling rises to between 40 and 1000 for requirements errors discovered in the operations phase in this survey.

Unfortunately a common feature of many large software engineering projects is that numerous problems are discovered very late in the development cycle, making them very expensive to fix. Many of these errors may have been introduced early in the development cycle, but are discovered much later on when the system is being tested, close to or even after deployment. Figure 2 illustrates a profile of the rate of error discovery over the development cycle. The graph shows most errors being discovered during testing phases with the error discovery falling close to and after deployment.

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. Figure 3 illustrates an idealised profile where the error discovery rate is higher in the earlier stages. 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 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

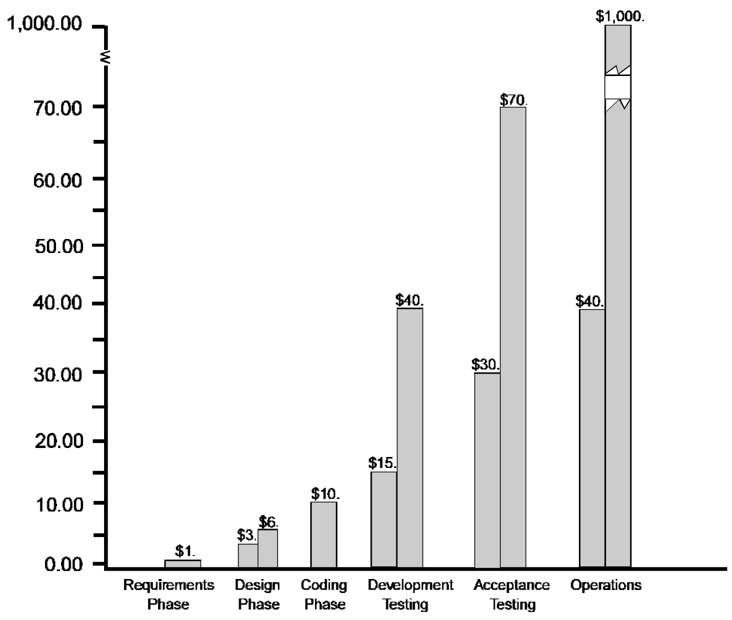


Figure 1. Cost of fixing requirements errors at different development stages [12]

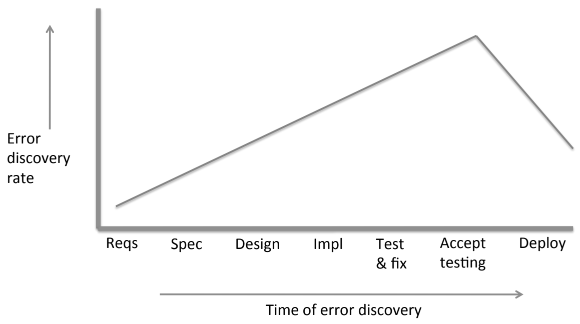


Figure 2. Typical error discovery rate at different stage of development

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*. As we will see, 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 abstrac-

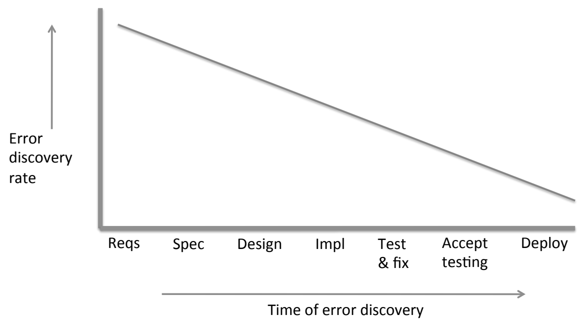


Figure 3. Idealised error discovery rate through early stage formal modelling and analysis

tion 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 (Figure 2), we would arrive at an ideal profile in which more errors are discovered as soon as they are introduced as illustrated by Figure 3.

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

1. *Users* are authorised to engage in *activities*.
2. User *authorisation* may be added or revoked.
3. Activities take place in *rooms*
4. Users gain access to a room using a one-time *token* provided they have authority to engage in the room activities.
5. Tokens are issued by a central *authority*.
6. A room *gateway* allows access with a token provided the token is valid .

Figure 4. List of requirements on the access control system

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 abstraction 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.

In the next section we will illustrate the process of abstraction through an example. Before we proceed however it is important to distinguish two forms of abstraction used in the context of software verification. Model checking of programs relies on being able to construct an abstraction of that program in order to reduce the (computational) complexity of automated verification. This form of abstraction takes a (formal) program as an input and can be automated – we refer to it as *program abstraction*. In this paper, we are addressing a different form of abstraction, which we refer to as *problem abstraction*. Problem abstraction is a creative process that takes (informal) requirements as input to produce a formal abstration. The purpose of problem abstraction is to increase our understanding of a problem (and remove errors in understanding).

2. Abstraction: Access Control Example

In this section we consider an example of an access control system, constructing an abstract formal model in Event-B. The example is intended to give the reader an insight into the abstraction process as well as giving a taste of the Event-B modelling language. In the next section (Section 3) we will look at validation of the formal model against the informal requirements. In Section 4 we will outline the role that tool-supported model verification plays in improving the quality of the model. This will in turn be followed in Section 5 by an outline of how model refinement can be used to relate our abstraction to the mechanisms used to achieve the access control purpose.

We consider a secure building consisting of a collection of rooms in which various restricted activities take place. Users of the building may have authority to engage in the activities. The access control system uses a token mechanism to control access by users to rooms so that a user should be allowed to enter a room only if they have sufficient authority.

Figure 4 provides a list of informal requirements on the access control system. We assume that it has already been established that the requirements satisfy the needs of the stakeholders (requirements validation). Presenting the requirements in list form, as in Figure 4, will facilitate the process of ensuring that they are accurately represented by the formal model (model validation). As a first step towards constructing an abstract

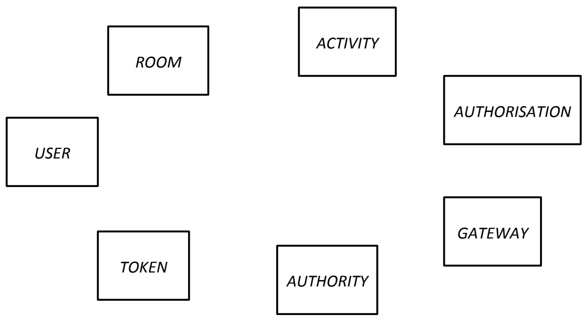


Figure 5. Entities identified from the requirements list

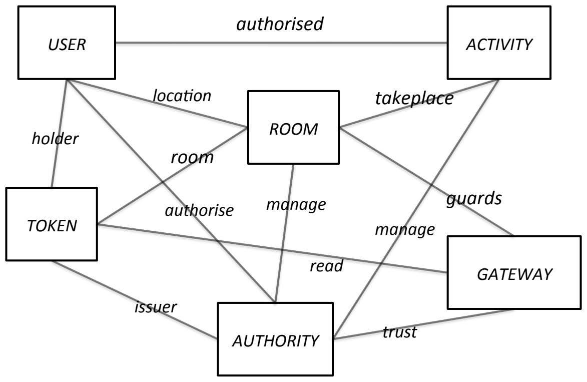


Figure 6. Relevant relationships between entities

model, it is useful to identify the entities involved in the problem. An easy way of doing this is to identify the nouns used in the list of requirements. In Figure 4 we have highlighted, in italics, the first occurrence of each noun. In Figure 5 we have represented the identified problem entities in a graphical way. This will facilitate the identification of the relationships between those entities that are relevant to the problem.

Entities and the relationships between them

To identify relevant relationships between entities, we take any pair of entities from the diagram in Figure 5 and make a judgement about whether there is a relationship between them that is relevant to the problem of access control. For example, if we consider the entities *USER* and *ROOM*, we can conclude that it is relevant to know which room a user is located in. To make this explicit, we draw a line between both entities in the diagram and give it a meaningful name, say *location*. This relationship, along with several other relevant relationships that we identified, is shown in Figure 6. This is an example of the well-established concept of entity-relationship diagram [5] (a precursor to the class diagrams found in UML [10]). The reader will notice that the *AUTHORISATION* entity that appears in Figure 5 has been removed from Figure 6. The reason for this is that, in constructing Figure 6, we made a judgement that the notion of authorisation is best represented as a relationship between a user and an activity and this relationship has been

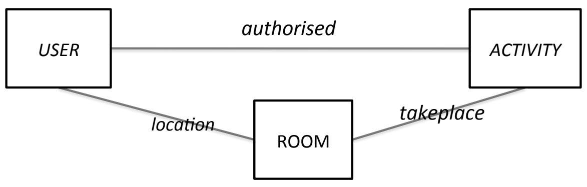


Figure 7. Simplifed entity-relationship diagram for access control

identified and labelled *authorised* in Figure 6. This makes having *AUTHORISATION* as an entity in its own right redundant so we remove it.

Now the entity-relationship diagram of Figure 6 is looking somewhat complex due to the number of relationships that we have identified. Before proceeding with completion of a formal model, we reflect on whether all of the entities and relationships we have identified are necessary in order to arrive at an appropriate abstraction of the access control problem. To do this we need to identify what we regard to be the purpose of the access control system. Our judgement is as follows:

> The purpose of our system is to enforce an access control *policy*.
>
> The access control policy is that users may only be in a room if they are authorised to engage in all the activities that may take place in that room

Simplifying the entities and their relationships

To express the policy we only require three entities, *USER* , *ROOM* and *ACTIVITY*, together with the relationships between these. This observation leads to the much simpler entity-relationship diagram shown in Figure 7 that contains three entities and three relationships between them. This simplification is an example of problem abstraction: we focus on the entities that are most relevant to describing the purpose of the system being analysed and ignore those that are less relevant to the purpose. Of course entities such as the tokens and the gatekeepers are an important part of the means by which the purpose is achieved so we will not ignore them forever. In Section 5 we will show how the token mechanism can be layered into our formal modelling through refinement.

For the purposes of formal modelling we treat the three entities of Figure 7 are distinct abstract types. Given these types, we can define the three relations mathematically as follows:

$$authorised \in USER \leftrightarrow ACTIVITY$$

$$takeplace \in ROOM \leftrightarrow ACTIVITY$$

$$location \in USER \leftrightarrow ROOM$$

Thus, $authorised$ is declared as a relation between $USER$ and $ACTIVITY$, $takeplace$ is declared as a relation between $ROOM$ and $ACTIVITY$ and $location$ is declared as a relation between $USER$ and $ROOM$. For readers not familiar with the mathematical notation of Event-B, we provide an overview of the mathematical concepts used in this paper as an appendix. Given the above declarations, we can express the security policy using the following predicate:

$$\forall u, r \cdot u \mapsto r \in location \Rightarrow takeplace[\{r\}] \subseteq authorised[\{u\}] \qquad (1)$$

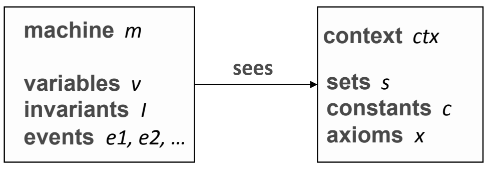


Figure 8. General structure of Event-B machine and context

Here $takeplace[\{r\}]$ specifies the image of r under the $takeplace$ relation, i.e., the set of activities that r is related to or the set of activities that take place in room r. Similarly $authorised[\{u\}]$ represents the set of activities that user u is authorised to engage in. Thus, (1) specifies that, for any user u and room r, if u is located in r, i.e., $u \mapsto r \in location$, then the set of activities that take place in the room ($takeplace[\{r\}]$) must be included in the set of activities that the user is authorised to engage in ($authorised[\{u\}]$). In other words, the user must have sufficient authorisation to be in the room.

Structure of an Event-B specification

Before presenting the remaining details of the access control specification, we outline the general structure of an Event-B specification. A specification consists of a static part, specified in a *context*, and a dynamic part, specified in a *machine*. An Event-B context contains the following elements:

- **Sets**: Abstract types used in specification to distinguish various entities
- **Constants**: Logical variables whose value remain constant
- **Axioms**: Predicates that specify assumptions about the constants.

An Event-B machine contains the following elements:

- **Variables**: State variables whose values can change
- **Invariants**: Predicates that specify properties about the variable that should always remain true.
- **Initialisation**: Initial values for the abstract variables
- **Events**: Guarded actions specifying ways in which the variables can change. Events may have parameters.

A machine may *see* the static elements defined in a context meaning that these elements are visible within the machine. The structure of a specification is outlined in Figure 8

Abstract access control in Event-B

Clearly the entities *USER*, *ROOM* and *ACTIVITY* from the diagram in Figure 7 should be specified as abstract types in a context. In addition, for the purposes of this paper, we assume that the allocation of activities to rooms cannot be modified during operation and thus we specify the $takeplace$ relation as a constant[2]. Accordingly we specify the following Event-B context for the access control system:

[2]The requirements are vague about whether the allocation of activities to rooms may be changed. We made a judgement to treat it as fixed but we could have chosen to make it changeable.

context $AccessControlContext1$
sets $USER,\ ROOM,\ ACTIVITY$
constants $takeplace$
axioms

$$@axm1\ \ takeplace\ \in ROOM \leftrightarrow ACTIVITY$$

Here and elsewhere we label predicates ($@axm1$) in the model for ease of reference.

We assume that users may change locations (implicit in Requirement 4 of Figure 4). Furthermore, although not explicitly state in the requirements of Figure 4, we shall assume that a user can be in at most one location at any time. We also assume that user authorisations may change (implicit in Requirement 2 of Figure 4). Accordingly we treat the $location$ and $authorised$ relations as variables. The assumption about users being in at most one room, together with the access control policy, are specified with variables and invariants in an Event-B machine as follows:

machine $AccessControl1$
sees $AccessControlContext1$
variables $location,\ authorised$
invariants

$$\begin{aligned}&@inv1\ \ location\ \in\ USER \rightarrowtail\!\!\!\!\!+\ ROOM\\&@inv2\ \ authorised\ \in\ USER \leftrightarrow ACTIVITY\\&@inv3\ \ \forall u, r\ \cdot\ u \mapsto r \in location\ \ \Rightarrow\\&\qquad\qquad takeplace[\{r\}]\ \subseteq\ authorised[\{u\}]\end{aligned}$$

events $\cdots$

Note that invariant $@inv1$ specifies that $location$ is a partial function, i.e., each user is mapped to at most one room. Invariant $@inv3$ specifies the access control policy as already identified. So far we have not identified the relevant events that model ways in which the variables can change. The $location$ relation may change when some user enters or leaves a room, thus we will introduce $EnterRoom$ and $LeaveRoom$ events. The $authorised$ relation is modified when user authorisation is added or revoked, thus we will introduce $AddAuthorisation$ and $RemoveAuthorisation$ events to the formal model.

We consider the $EnterRoom$ event. In Event-B an event of a machine consists of a list of guards and a list of actions. The guards specify the conditions that must hold for the event to be executed - an event can only be executed when all of its guards are satisfied. The effect of an event is specified as a list of assignments to variables of the machine. An event may have parameters representing possible values that determine the effect of the event, e.g., the $EnterRoom$ event is parameterised by a user u and the room r they are entering. In order for a user to enter a room they must have sufficient authorisation to enter that room, i.e., they must have authorisation for all the activities that take place in that room. This condition will be represented as a guard in the specification of the $EnterRoom$ event. The full specification of the $EnterRoom$ event is as follows:

```
event $EnterRoom$
  any $u, r$ where
    $@grd1\ \ u \in USER$
    $@grd2\ \ u \notin dom(location)$
    $@grd3\ \ r \in ROOM$
    $@grd4\ \ takeplace[\{r\}] \subseteq authorised[\{u\}]$
  then
    $@act1\ \ location(u) := r$
  end
```

Here $@grd2$ specifies that the user u is not already located in some room while $@grd4$ specifies that the user has sufficient authorisation to enter room r.

We have used two forms of predicate in the specification of our Event-B machine, invariants and guards. Let us clarify the distinction between these uses of predicates:

Invariants: Invariants specify properties of model variables that should always remain true no matter what events get executed. Execution of an event that leads to a violation of an invariant is undesirable.

Guards: Guards specify enabling conditions under which events may occur. Guards should be strong enough to ensure invariants are maintained by the actions of an event but not so strong that they prevent desirable behaviour.

The $LeaveRoom$ event has a simpler guard than $EnterRoom$ - the only condition is that the user is currently in the room:

```
event $LeaveRoom$
  any $u, r$ where
    $@grd1\ \ u \mapsto r \in location$
  then
    $@act1\ \ location := \{u\} \lhd location$
  end
```

Here, action $@act1$ removes the mapping from u to r from the $location$ relation.

Our first attempt at specifying the authorisation modification events are as follows: $AddAuthorisation$ adds a mapping from a user to an activity to $authorised$ provided it is not already present while $RemoveAuthorisation$ removes a mapping that is present.

```
event $AddAuthorisation$
  any $u, a$ where
    $@grd1 \;\; u \in USER$
    $@grd2 \;\; a \in ACTIVITY$
    $@grd3 \;\; u \mapsto a \notin authorised$
  then
    $@act1 \;\; authorised := authorised \cup \{u \mapsto a\}$
  end

event $RemoveAuthorisation$
  any $u, a$ where
    $@grd1 \;\; u \in USER$
    $@grd2 \;\; a \in ACTIVITY$
    $@grd3 \;\; u \mapsto a \in authorised$
  then
    $@act1 \;\; authorised := authorised \setminus \{u \mapsto a\}$
  end
```

In Section 4, when we apply model verification to the machine we will see that the $RemoveAuthorisation$ event, as specified above, may lead to a violation of the access control policy.

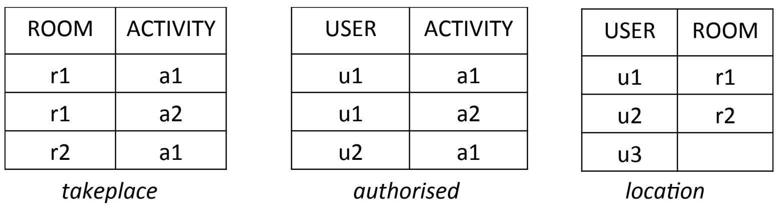
In Event-B we also specification a special initialisation event for a model that defines how the variables are to be initialised before other events are performed. For the access control model, we specify that the both variables are empty, i.e., there are no authorisations and no users are in rooms:

```
event $initialisation$
  $@act1 \;\; authorised := \varnothing$
  $@act2 \;\; location := \varnothing$
```

3. Model validation

Now that we constructed an abstract model in Event-B, we attempt to validate the model against the informal requirements of Figure 4. One useful validation technique for Event-B models is to use an animation tool such as ProB [14] or AnimB[3]. With these tools, the abstract types are instantiated with some illustrative values, e.g., the type $USER$ is instantiated with the values $u1$, $u2$, $u3$, and the model can be executed on this models. The execution is driven by the user and at each step the state can be inspected. Figure 9 represents the state that is reached by executing the following sequence of events on our model of the access control system:

[3]www.animb.org



ROOM	ACTIVITY
r1	a1
r1	a2
r2	a1

takeplace

USER	ACTIVITY
u1	a1
u1	a2
u2	a1

authorised

USER	ROOM
u1	r1
u2	r2
u3	

location

Figure 9. Result of animating model through a sequence of events

$$\begin{array}{l} initialisation \\ AddAuthorisation(u1, a1) \\ AddAuthorisation(u1, a2) \\ AddAuthorisation(u2, a1) \\ EnterRoom(u1, r1) \\ EnterRoom(u2, r2) \end{array}$$

Figure 9 represents each of the relations *takeplace*, *authorised* and *location* as tables. For example, the $location$ table shows that $u1$ is related to $r1$ under the $location$ relation, i.e., a state corresponding to user $u1$ being in room $r1$. Manual inspection of the tables in Figure 9 shows that they do represent a state that satisfies the key security invariant $@inv3$. However, rather than using manual inspection to check for satisfaction of invariants, in Section 4 we will show how model checking and proof can be used to do this in a systematic and automated way. 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 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 whether and 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 10 where the explanations of how a requirement is represented in the formal model are in shown in the third column. For example, the annotation on Requirement 1 provides an explanation of how that requirement is represented in the formal model through the $authorised$ relation. Three of the requirements (4, 5, 6) are not addressed by our abstraction as indicated by the annotations. The reason they are not addressed is that we chose to abstract away several entities, such as *TOKEN*, in our abstract model.

As a result of our abstraction and formalisation we have identified some additional informal requirements that we believe would be valuable to record. As indicated in Figure 10, Requirements 7 to 10 are the new requirements that we have identified. These additional requirements make explicit some properties that were either missing or were implicit. For example, the original requirements said nothing about whether the activities that take place in rooms was fixed or changeable (missing requirement). The access control policy was implicit in Requirement 4 (implicit requirement). We have made it explicit in Requirement 7.

Id	Requirement	Representation in model
1.	Users are authorised to engage in activities.	This is represented by the $authorised$ variable that relates users to the activities they are authorised to engage in.
2.	User authorisation may be added or revoked.	This is represented by the *AddAuthorisation* and *RemoveAuthorisation* events.
3.	Activities take place in rooms.	This is represented by the events for adding and revoking authorisation.
4.	Users gain access to a room using a one-time token provided they have authority to engage in the room activities.	Not represented at this stage.
5.	Tokens are issued by a central authority.	Not represented at this stage.
6.	A room gateway allows access with a token provided the token is valid.	Not represented at this stage.
7.	**New requirement:** Users may only be in a room if they are authorised to engage in all activities that may take place in that room.	This is represented by invariant $@inv3$.
8.	**New requirement:** Users may enter rooms if they have sufficient authorisation for that room. Users may always leave a room they are in.	This is represented by the *EnterRoom* and *LeaveRoom* events.
9.	**New requirement:** A user may be in at most one room.	This is represented by $location$ being a partial function.
10.	**New requirement:** The allocation of activities to rooms is fixed.	This is represented by $takeplace$ being a constant (not a variable).

Figure 10. Revised list of requirements with tracing information

4. Model verification

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 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.

Consider the state of the access control system shown in Figure 9. As already explained, this state is reachable by executing a particular sequence of events. In the state represented both $u1$ and $u2$ are in a room and have appropriate authorisation to be there. Now if the next event to be performed was $RemoveAuthorisation(u1, a2)$ the state reached would be as shown in Figure 11. This new state is in *incorrect* state, that is, it violates the invariant since user $u1$ is still in room $r1$ even though activity $a2$ takes place in $r1$ and they no longer have authority to perform that activity. Indeed, ProB can automatically find a sequence of events that lead to an invariant violation (known as a

ROOM	ACTIVITY
r1	a1
r1	a2
r2	a1

takeplace

USER	ACTIVITY
u1	a1
u2	a1

authorised

USER	ROOM
u1	r1
u2	r2
u3	

location

Figure 11. Incorrect state reached when *RemoveAuthorisation(u1,a2)* is applied to Figure 9

event E
 any p **where**
  $@grd \quad G(p, v)$
 then
  $@act \quad v := F(p, v)$
 end

Invariant Preservation PO:

Hyp1 : $I(v)$

Hyp2 : $G(p, v)$

Goal : $I(\ F(p, v)\ )$

Figure 12. Invariant preservation proof obligation for an event

counterexample). The counterexample that leads to the state in Figure 9 is not the shortest possible counterexample. ProB can automatically find a shorter counterexample such as the following:

$$\begin{array}{l} initialisation \\ AddAuthorisation(u1, a1) \\ AddAuthorisation(u1, a2) \\ EnterRoom(u1, r1) \\ RemoveAuthorisation(u1, a1) \end{array}$$

The counterexample demonstrates an error in the model: if authorisation for $a2$ is removed while user $u1$ is still in the room, i.e., has entered the room but not left it, then the invariant is violated. A moments reflection confirms that this is indeed an issue with access control that has not been addressed by the informal requirements: does it make sense to remove authorisation if the user is currently located in a room in which the activity takes place?

Before addressing a solution to this issue, we look at how the error is reflected in the proof obligation (PO) for invariant preservation. Figure 12 shows a definition of this PO. The left side of the figure provides a schematic specification of an event E with a guard represented 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 12 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 representation of 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 model and the automated provers of Rodin are able to discharge all of the generated POs except for one. The specification of the $RemoveAuthorisation$ event together with invariant $@inv3$ (the access control policy) give rise to the following PO that cannot be proved:

$$\begin{aligned}
&\mathsf{Hyp1}: \ \forall u, r \cdot u \mapsto r \in location \ \Rightarrow \ takeplace[\{r\}] \subseteq authorised[\{u\}] \\
&\mathsf{Hyp2}: \ u \mapsto a \in authorised \\
&\mathsf{Goal}: \ \forall u, r \cdot u \mapsto r \in location \ \Rightarrow \\
&\qquad\qquad takeplace[\{r\}] \subseteq \underline{(authorised \setminus \{u \mapsto a\})}\,[\{u\}]
\end{aligned}$$

Here, Hyp1 is the invariant to be preserved and Hyp2 is the guard of the event. The event makes an assignment to the $authorised$ variable and thus the goal is formed by substituting $authorised$ by $(authorised \setminus \{u \mapsto a\})$. 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, $(authorised \setminus \{u \mapsto a\})[\{u\}]$, is reduced compared with that in the hypothesis, Hyp1, while the left-had side, $takeplace[\{r\}]$, remains unchanged.

This failing PO highlights the fact that the $RemoveAuthorisation$ event removes activity a from the right-hand side of the set inequality without removing it from the right-hand side. If that activity was not a member of the left-hand side, then removing from the right-hand side would be ok, i.e.,

$$S \subseteq T \wedge a \notin S \quad \Rightarrow \quad S \subseteq (T \setminus \{a\}). \tag{2}$$

The following property will ensure that a is not a member of the left-hand side ($takeplace[\{r\}]$):

$$u \mapsto r \in location \quad \Rightarrow \quad r \mapsto a \notin takeplace. \tag{3}$$

Now since $location$ is a partial function, u is related to at most one room r. Thus property (3) is equivalent to the following property:

$$u \in dom(location) \quad \Rightarrow \quad location(u) \mapsto a \notin takeplace. \tag{4}$$

Now property (4) will be available as a hypothesis in the PO for $RemoveAuthorisation$ if the property is a guard of the event. Adding this property as a guard gives us the following revised specification of the event:

```
event $RemoveAuthorisation$
  any $u, a$ where
    $@grd1 \quad u \in USER$
    $@grd2 \quad a \in ACTIVITY$
    $@grd3 \quad u \mapsto a \in authorised$
    $@grd4 \quad u \in dom(location) \Rightarrow location(u) \mapsto a \notin takeplace$
  then
    $@act1 \quad authorised := authorised \setminus \{u \mapsto a\}$
  end
```

Id	Requirement	Representation in model
1.	$\cdots$	
2b.	**New requirement:** User authorisation may not be revoked while a user is located in a room in which the activity takes place.	This is represented by guard $@grd3$ of the *RemoveAuthorisation* event.
3.	$\cdots$	

Figure 13. Second revision of requirements

This revised definition of $RemoveAuthorisation$ has an additional guard specifying that we can only remove authorisation for an activity a from user u provided the activity does not take place in the room in which the user is located (if they are in a room). With this the Rodin provers are able to discharge all of the invariant-preservation POs for the abstract model.

The counterexample generated by the ProB model checker using the original version of the event highlighted a problem with the specification of the event. This stronger condition for removing authorisation was identified through our attempt to prove that the original specification of the event maintained the access control invariant. 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 access control 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 *RemoveAuthorisation* event with the stronger guards holds. A more elaborate solution might be to introduce a notion of pending authorisation removals. These are authorisation removals that are marked as pending if a user is in a room in which the activity takes place but that only take effect after the user has left the room. This solution would also allow us to retain the access control invariant.

Give the revision to the model we have decided to adopt, it is appropriate to modify the requirements document accordingly. We add an additional requirement that is related to our existing requirement about adding and revoking authorisation (Requirement 2). This new requirement (Requirement 2b) is shown in Figure 13.

5. Refinement

In this section we show how the token mechanism that we previously abstracted away from may be introduced to the formal modelling through refinement. A user needs to acquire a token in order to enter a room. A token has a holder and a room for which it grants access to the holder. Figure 14 shows an entity-relationship diagram in which the *TOKEN* entity has been introduced, along with two new relations between tokens and users and between tokens and rooms.

An Event-B machine *m2* may be declared to be a refinement of some other Event-B machine *m1*. In this case we refer to *m1* as the abstract machine and *m2* as the refined machine. Machine *m1* is said to be a correct refinement of *m1* if any behaviour that may

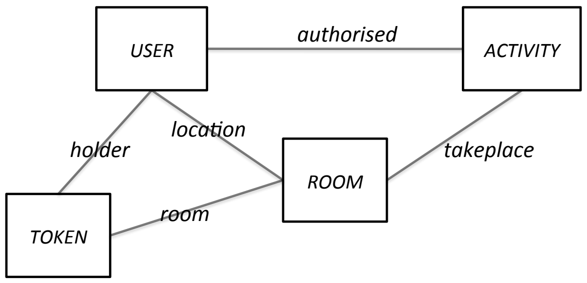


Figure 14. Entity-relationship diagram with tokens

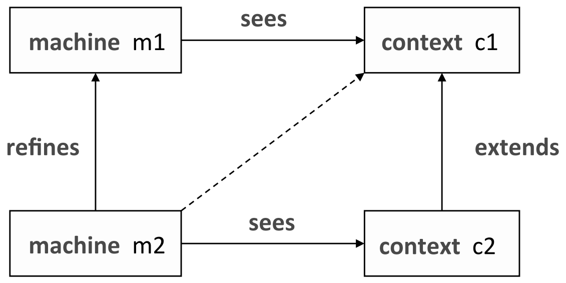


Figure 15. Structuring refinement

be exhibited by *m2* is also a possible behaviour of *m1*. Refinement represents our expectation that the behaviour of *m2* should conform to the behaviour of *m1*. Of course declaring that *m2* refines *m1* does not on its own guarantee the correctness of a refinement. Rather the declaration gives rise to proof obligations that need to be discharged in order to guarantee the correctness of a refinement. When refining a machine, it is common to specify new types and constants to be used in the refinement. This is achieved by specifying a new context for the refined machine. If the specification of any new types and constants depend on the types and constants used by the abstract machine, the new context is declared to be an extension of the context of the abstract model. The relationships between a machine and its refinement, as well as their respective contexts, is illustrated by Figure 15. This figure shows the refinement declaration from *m2* to *m1*, together with the relationships with their contexts. A refined context *c2* is declared as an extension of the abstract context *c1* meaning context *c2* may refer to types and constants specified in context *c1*. The dashed line from machine *m2* to context *c1* indicates that *m2* implicitly see definitions in *c1* (via *c2*).

For the refinement of the access control system, we introduce the type *TOKEN* together with the relation between *TOKEN* and *USER*, called $holder$, and between *TOKEN* and *ROOM*, called $room$ (Figure 14). Clearly *TOKEN* needs to be introduced as a new type. We need to decide whether $holder$ and $room$ should be specified as variables or constants. If we need to specify that the holder of a token may change, then this relation should be specified as a variable. However the assumption we will make is that once a token is issued, its holder and room attributes will not change. Thus we will model the $holder$ and $room$ relations as constants. This leads to the following context, *AccessControlContext2* that extends *AccessControlContext1*:

context $AccessControlContext2$
extends $AccessControlContext1$
sets $TOKEN$
constants $room\ \ holder$
axioms

$@axm1\ \ room\ \in TOKEN \rightarrow ROOM$
$@axm2\ \ holder\ \in TOKEN \rightarrow USER$

Here $room$ and $holder$ are total functions so each token t is associated with a single room, written $room(t)$, and a single holder, written $holder(t)$. We need to assume that the gatekeeper to a room can check somehow that the user trying to gain entry using a token is indeed the same user who the token is related to via the $holder$ relation. For example, this might be achieved through some biometric mechanism though we do not address this issue in this paper.

Our first attempt at specifying the variables and invariants of a refinement of the token system are as follows:

machine $AccessControl2$
refines $AccessControl1$
sees $AccessControlContext2$
variables $location,\ authorised,\ valid$
invariants

$@inv4\ \ valid\ \subseteq\ TOKEN$

events $\cdots$

Here we have explicitly declared *AccessControl*2 to be a refinement of *AccessControl*1 and we have declared that *AccessControl*2 can see the extended context *AccessControl-Context*2. The refining machine has one additional variable, $valid$, which represents the set of tokens that have been issued and have yet to be used to gain access to a room.

In Event-B, every event of the refining machine is either a refinement of some event of the abstract machine or is a so-called *new* event introduced in the refinement that has no corresponding abstract event. The event for creating a new token is one such new event. It has no corresponding abstract event since the abstract model did not include a notion of token thus it was not appropriate to model token creation at the abstract level. The $IssueToken$ event has three parameters, a user, a room and a token:

```
**event** $IssueToken$
    **any** $u, r, t$ **where**
        $@grd1 \;\; u \in USER$
        $@grd2 \;\; r \in ROOM$
        $@grd3 \;\; takeplace[\{r\}] \subseteq authorised[\{u\}]$
        $@grd4 \;\; t \in TOKEN \setminus valid$
        $@grd5 \;\; room(t) = r$
        $@grd6 \;\; holder(t) = u$
    **then**
        $@act1 \;\; valid := valid \cup \{t\}$
    **end**
```

Here the effect of the event is to add the new token t to the set of valid tokens ($@act1$). The token can only be issued if the user has sufficient authorisation to be able to enter the room ($@grd3$). The new token should not already be on the set of valid tokens ($@grd4$). The value chosen to represent token t must be such that the room and the holder attributes match the room and user attributes ($@grd5$, $@grd6$). At an abstract level we do not distinguish between input and output parameters – parameters simply represent values that are chosen nondeterministically in a way that satisfies the guards. In the implementation the values for u and r could be chosen externally by a (human) administrator while the value for t could be chosen internally by the token issue mechanism. It is the responsibility of the implementation to ensure that the guards are enforced.

In the abstract specification of the $EnterRoom$ event, the main guard is the check that the user has sufficient authority, i.e.,

$$takeplace[\{r\}] \subseteq authorised[\{u\}] \tag{5}$$

Now, in our refinement we wish to represent the design requirement that a user must provide a valid token in order to gain access to a room. At the point at which the user is entering a room, the validity of the token is the only check that is made. Property (5) does not need to be checked at that point since it will already have been checked at the point at which the token is issued ($IssueToken$ event). The refined $EnterRoom$ event is thus specified as follows:

```
**event** $EnterRoom$ **refines** $EnterRoom$
    **any** $u, r, t$ **where**
        $@grd1 \;\; u \in USER \setminus dom(location)$
        $@grd2 \;\; r \in ROOM$
        $@grd3 \;\; t \in valid$
        $@grd4 \;\; room(t) = r$
        $@grd5 \;\; holder(t) = u$
    **then**
        $@act1 \;\; location(u) := r$
        $@act2 \;\; valid := valid \setminus \{t\}$
    **end**
```

The declaration '$EnterRoom$ **refines** $EnterRoom$' indicates that this refining event is intended to be a refinement of the corresponding abstract event. Although we use the same name for both the abstract and refining event here, in Event-B the names may be different. The refining $EnterRoom$ event has an additional parameter compared with the abstract version of the event, namely the token t. The token is required to be valid ($@grd3$) and its $room$ and $holder$ attributes must correspond to the user and the room they are trying to enter ($@grd4, @grd5$). Just like the abstract $EnterRoom$ event, the refining version sets the location of the user to room r ($@act1$). The refining version also removes the token used to enter the room from the set of valid tokens ($@act2$). This is because the token is required to be a one time (single use) token.

In some cases we require a refining event to have the same specification as the corresponding abstract event or maybe to have additional behaviour. This can be achieved in Event-B using a special case of refinement called *event extension*. This specifies that the refining event includes all of the parameters, guards and actions of the abstract event together with additional parameters, guards and events. Only the additional elements need to be specified in the refining event. Use of the keyword **extends** means that the specification elements of the abstract event are implicitly included. We specify the initialisation of the refining machine in this way:

event $initialisation$ **extends** $initialisation$
  $@act3 \quad valid := \varnothing$

As well as initialising the variables that are present in the abstract machine, the refining initialisation also refines the newly introduced variables, $valid$, to be empty. The other events from the abstract model are retained unchanged:

event $LeaveRoom$ **extends** $LeaveRoom$

event $AddAuthorisation$ **extends** $AddAuthorisation$

event $RemoveAuthorisation$ **extends** $RemoveAuthorisation$

6. Refinement verification

In order to verify that the refining model conforms to the abstract model, there are two main forms of proof obligation to be proved: *invariant preservation* and *guard refinement*. Invariant preservation is as in Figure 12. Guard refinement involves proving that the guards of the refined event are not weaker than the guards of the abstract events, i.e., that if the refining event is enabled in some state then in any corresponding abstract state the abstract event is also enabled. The definitions of the refinement POs we use are given in Figure 16. Here $E2$ and $E1$ represent a refining event and its corresponding abstract event respectively. The variables of the abstract model are represented by v and the new variables introduced in the refining machine are represented by w. Verification of the refinement makes use of invariants that state properties of w and v combined. These are sometimes referred to as gluing invariants as they are used to specify properties that glue together abstract and refining variables. In Figure 16 these gluing invariants are repre-

```
event $E1$                                   event $E2$ refines $E1$
  any $p$ where                                any $p, q$ where
    @$grd1$  $G1(p,v)$                           @$grd2$  $G2(p,q,w)$
  then                                         then
    @$act1$  $v := F1(p,v)$                      @$act1$  $v := F1(p,v)$
  end                                            @$act2$  $w := F2(p,q,w)$
                                               end
```

Guard Refinement PO:

Hyp1 : $J(v,w)$

Hyp2 : $G2(p,q,w)$

Goal : $G1(p,v)$

Figure 16. Guard refinement when $E2$ refines $E1$ with gluing invariant J

sented by $J(v,w)$. In the guard refinement PO we see that the gluing invariants and the refining guard are hypothesis while the abstract guard is the goal. In other words we are required to prove that the abstract guard follows from the refining guard and the gluing invariants. When a refining event is an extension refinement, then the refined guards trivially satisfy the guard refinement PO since, in that case, the effective guards of the refining event are the abstract guards conjoined with the refining guards. New events in a refinement do not need to satisfy the guard refinement PO since they do not have a corresponding abstract event. For a fuller definition and explanation of all the Event-B POs, the reader is referred to Abrials' book [1].

The only event of the refining model that is a proper refinement (as opposed to an extension refinement) is the $EnterRoom$ event. The guard refinement PO for this event is as follows:

Hyp1 : $t \in valid$

Hyp2 : $r = room(t)$

Hyp3 : $u = holder(t)$

Goal : $takeplace[\{r\}] \subseteq authorised[\{u\}]$

Here the hypotheses are the guards of the refining $EnterRoom$ event while the goal is a guard of the corresponding abstract event. As it stands, this PO is not provable: there is no overlap between variables and constants of the hypotheses with those of the goal so effectively the PO contains no useful hypotheses that would contribute to proving the goal. The hypotheses refer only to refining variables and constants while the goal refers only to abstract ones. What is missing from the PO is any hypothesis that would link the abstract and refining elements, i.e., a gluing invariant.

In this case, the above unprovable PO actually suggests a gluing invariant that would allow the PO to be discharged:

$$\forall t, r, u \cdot t \in valid \ \wedge \ r = room(t) \ \wedge \ u = holder(t) \ \Rightarrow \\ takeplace[\{r\}] \subseteq authorised[\{u\}] \tag{6}$$

Here we have simply constructed an invariant with the hypotheses as antecedent and goal as consequent and quantified over the parameter variables t, r and u. Because of the equations for quantified variables r and u in the antecedent, we can eliminate them to get the following:

$$\forall t \cdot t \in valid \;\;\Rightarrow\;\; takeplace[\{room(t)\}] \subseteq authorised[\{holder(t)\}] \tag{7}$$

Now, simply converting an unprovable PO into an invariant is not always the right solution to getting a PO proved. For example, in Section 4 we saw that the way to make a PO provable was by adding a guard to the event rather than an invariant. In other cases it may be that the actions of an event are incorrectly specified and need to be fixed. Before adding Property 7 as a gluing invariant to our refining model, it is appropriate to make a (human) judgement about whether we believe this really is an invariant of the system. In this case it seems clear that this property should be an invariant: the property states that

> for any valid token, the set of activities that can take place in the room associated with the token must be included in the set of activities that the holder of the token is authorised to engage in.

If a token is valid we would expect it to be the case that the holder of the token has sufficient authority to enter the room associated with the token and thus we add Property (7) as an invariant to the refining model as follows:

machine $AccessControl2$
refines $AccessControl1$
sees $AccessControlContext2$
variables $location,\ authorised,\ valid$
invariants

$@inv4\ \ valid \subseteq TOKEN$
$@inv5\ \ \forall t \cdot t \in valid \Rightarrow$
$\qquad takeplace[\{room(t)\}] \subseteq authorised[\{holder(t)\}]$

events $\cdots$

The addition of $@inv5$ to the refining model allows the Rodin provers to discharge the guard refinement PO for the $EnterRoom$ event. Of course, the addition of $@inv5$ in turn gives rise to additional invariant preservation POs. We will return to this issue but for now we consider validation of the refining model against the informal requirements together with improvements to the requirements. Figure 17 shows a further revision of the requirements to reflect insights gained through the refinement. The gluing invariant we identified specifies precisely what it means for a token to be valid and we decide it is useful to reflect this in the requirements. In Figure 17 we replace the original Requirement 4 by two requirements, 4a and 4b. Requirement 4a states that a user must have a valid token to enter a room while Requirement 4b defines what it means for a token to be valid. Also in Figure 17 we have elaborated on the extent to which Requirements 5 and 6 are represented by the refining model.

As already stated, the addition of $@inv5$ gives rise to additional invariant preservation POs. It is easy to see why the $IssueToken$ event preserves this invariant be-

Id	Requirement	Representation in model
	...	
4a.	**Revised requirement (replacing 4):** Users gain access to a room using a one-time *token* provided the token is valid.	Guards of refined *EnterRoom* event ensure token is valid. Action $@act2$ of *EnterRoom* event ensures token cannot be used again (one-time).
4b.	**Revised requirement (replacing 4):** A token is valid provided the holder of that token is authorised to engage in all activities that can take place in the room associated with the token .	Invariant $@inv5$ of refined model.
5.	Tokens are issued by a central *authority*.	Token issue is represented by *TokenIssue* event. The central authority not explicitly represented at this stage.
6.	A room *gateway* allows access with a token provided the token is valid.	Access with valid token is represented by refined $EnterRoom$ event. Other properties of the gateway are not explicitly represented at this stage.
	...	

Figure 17. Third revision of requirements

cause it includes a guard to check that the user for the new token has sufficient authorisation. The invariant preservation POs for all the other events, except one, can be proved using the Rodin provers. The one event that does not preserve the token-validity invariant ($@inv5$) is the same event that causes problems in the abstract model, the $RemoveAuthorisation$ event. The invariant preservation PO is as follows:

$$\begin{aligned}
\mathsf{Hyp1}:\ & \forall t \cdot t \in valid \Rightarrow \\
& \quad takeplace[\{room(t)\}] \subseteq authorised[\{holder(t)\}]] \\
\mathsf{Hyp2}:\ & u \mapsto a \in authorised \\
\mathsf{Hyp3}:\ & u \in dom(location) \Rightarrow location(u) \mapsto a \notin takeplace \\
\mathsf{Goal}:\ & \forall t \cdot t \in valid \Rightarrow \\
& \quad takeplace[\{room(t)\}] \subseteq \underline{(authorised \setminus \{u \mapsto a\})}\,[\{holder(t)\}]
\end{aligned}$$

In this PO, the hypotheses are the token-validity invariant and the guards of the $RemoveAuthorisation$ event while the goal is the invariant with the substitution applied to the variable that is modified by that event (as underlined in the goal). As was the case in Section 4, where we identified a problem with preservation of the access control invariant by $RemoveAuthorisation$, the right-hand side of the set inequality in the goal is reduced compared with that in the hypothesis while the right-hand side remains unchanged. $RemoveAuthorisation$ removes activity a from the right-hand side of the set inequality without removing it from the right-hand side. If that activity was not a member of the left-hand side, then it removing from the right-hand side would be ok. We can ensure this by adding the following guard:

$$\forall t \cdot t \in valid \land u = holder(t) \;\; \Rightarrow \;\; room(t) \mapsto a \notin takeplace \tag{8}$$

That is, if the user holds a valid token, then the activity being removed from the user cannot take place in the room that the token is associated with. We revise our specification of the *RemoveAuthorisation* event by adding Property (8) as a guard. This is achieved through extension refinement of the abstract event since the only revision is the addition of a guard:

```
event RemoveAuthorisation extends RemoveAuthorisation
  when @grd5  $\forall t \cdot t \in valid \land u = holder(t) \;\Rightarrow\; room(t) \mapsto a \notin takeplace$
```

With the addition of this guard, the Rodin provers are able to discharge the invariant preservation PO for the token-validity invariant.

We might conclude that this additional constraint on removing authorisation is not satisfactory however. As it stands, once a token has been issued, we cannot remove authorisation that would conflict with that token until after the token has been used. We have not specified any mechanism for invalidating a token. Human judgement might suggest that such a mechanism ought to be provided. At the level of abstraction at which our model is specified, it is very easy to specify an event that invalidates a token as follows:

```
event RescindToken
  any t where
    @grd1  $t \in valid$
  then
    @act1  $valid := valid \setminus \{t\}$
  end
```

The more difficult engineering question is how token invalidation can be achieved. One approach commonly used with secure tokens is for them to be time-stamped with an expiry time and if they have not been used by that time, then they become invalid. Another approach would be for the central authority to distribute a black-list of invalid tokens periodically. Let us suppose that the decision to made to incorporate a time-stamp mechanisms on tokens so that tokens are stamped with an expiry time when they are issued and that this is checked by a room gatekeeper. In Figure 18 we have defined some additional requirements to reflect this decision. Treating the time-stamp requirements could be addressed as a further refinement of the formal model. We leave this as an exercise for the reader!

7. What, how, why

In treating the access control system through a combination of abstraction, refinement, validation and verification, we have distinguished and addressed three important questions about this particular system:

1. What does it achieve?
2. How does it work?

Id	Requirement	Representation in model
1.	$\cdots$	
2c.	**New requirement:** User authorisation may not be revoked while a user holds a valid token that would allow them to enter a room in which the activity takes place.	This is represented by guard $@grd5$ of the *RemoveAuthorisation* event.
	$\cdots$	
4c.	**New requirement:** A token is valid provided its time-stamp has not expired.	The time-stamp requirement is not addressed at this stage.
	$\cdots$	
11.	**New requirement:** Tokens are time stamped with an expiry time and a valid token is rescinded if it has not been used by that time.	This is partially represented by the *RescindToken* event. The time stamp mechanism is not addressed at this stage.

Figure 18. Fourth revision of requirements

3. Why does it work?

Typically, when constructing a software solution, the tendency is to focus on the second question: how it works. That is what a program is after all – a structured set of instructions explaining how some purpose is achieved. If we are to ensure that a system is fit for purpose, we need to understand that purpose clearly. By focusing on the purpose of the access control system and identify a minimal set of entities required to express this in a formal way (users, rooms, activities), we were able to express the first question in a clear manner:

- **What does it achieve?** The system allows users to enter an leave rooms in a way that satisfies the access control policy. The access control policy is that, if a user is in room, then that user must be authorised to engaged in all activities that can take place in the room

This question is answered precisely in the abstract Event-B model through a small number events and variables and through expression of the access control policy as an invariant:

$$\forall u, r \cdot u \mapsto r \in location \;\Rightarrow\; takeplace[\{r\}] \subseteq authorised[\{u\}]$$

The access control rights are maintained centrally and made available to room gatekeepers through the token mechanism. The enforcement of the access control is achieved through the issue and validity checking of tokens.

- **How does it work?** The access control mechanism works by issuing tokens to users provided they have sufficient authorisation and by checking the validity of tokens when a user attempts to enter a room.

These properties are expressed clearly as guards in the refining model of the access control system. The $IssueToken$ event ensures there is sufficient authorisation when a token is issued through the following guard:

$$takeplace[\{r\}] \subseteq authorised[\{u\}]$$

The refining $\mathit{EnterRoom}$ event ensures the validity of a token through the following guard:

$$t \in \mathit{valid} \;\; \wedge \;\; \mathit{room}(t) = r \;\; \wedge \;\; \mathit{holder}(t) = u$$

The attempt to verify the correctness of the refining machine required the identification of a gluing invariant expressing an obvious (in hindsight) property of valid tokens. The gluing invariant is as follows:

$$\forall t \cdot t \in \mathit{valid} \;\; \Rightarrow \;\; \mathit{takeplace}[\{\mathit{room}(t)\}] \subseteq \mathit{authorised}[\{\mathit{holder}(t)\}]$$

This key property of valid tokens essentially explains why the token machanism works.

- **Why does it work?** For any valid token, the holder of the token must be authorised to engage in all activities that can take place in the room associated with the token.

This final question is rarely expressed explicitly in software developments. The normal mechanism for convincing ourselves that a software system is fit for purpose is through testing. While comprehensive testing does provide evidence that a system works, it does not provide evidence of *why* a system works. The gluing invariant provides evidence of why the token mechanism achieves the required purpose. The derivation of this invariant illustrates the way in which the attempt to prove correctness encourages the identification of key properties that can provide insight into why a particular solution works.

8. Guidelines for system level reasoning

Traditional program verification methods, as exemplified by Hoare logic [8], are designed for reasoning about the correctness of programs. However, rather than reasoning about programs, what we have given an overview of here is what might be termed *system level reasoning*. By this we mean reasoning about an overall system rather than just the software parts of a system. We achieve system-level reasoning through the use of a formal modelling language, Event-B, which is independent of any particular programming language. We identified various entities in the system and abstracted these to the key entities to focus on the main purpose. This purpose was then modelled formally in Event-B and formal verification was applied to the model.

Reasoning about the system was much more than proving certain properties of our model, e.g., invariant preservation proofs. Several different forms of reasoning were deploy: identification of the various entities in the system and their relationships, identification of the main purpose, abstraction from design details – all of these are forms of reasoning. Constructing the formal model of the abstraction and applying model verification to the model (both model checking and proof) are other forms of reasoning the we deployed. Validation of the model against the requirements through human judgements and revising the requirements accordingly are other forms of reasoning. Deciding what design elements to incorporate in refinement steps are, again, forms of reasoning. All these forms of reasoning complement each other in helping us to understand the purpose of system, understand how that purpose is achieved and understand why it is achieved correctly by the chosen solution.

There are many examples of systems that have been reasoned about using Event-B:

- train signalling system [1]
- mechanical press system [1]
- flash-based filestore [6]
- electronic purse system [4]
- cruise control system [16]
- part of a space craft system [7]

We can identify some generic guidelines on how to arrive at a useful abstraction of a system. Abstraction can be viewed as a process of *simplifying our understanding* of a system. The simplification should focus on the *intended purpose* of the system while *ignoring details of how* that purpose is achieved. For example, we focused on enforcement of the access control policy as representing the intended purpose and ignored the mechanisms through which that is achieved in our initial abstraction.

The modeller needs to make judgements about what they believe to be the *key features* of the system. If the purpose is to *provide some service*, then the abstraction should focus on what a system does from the perspective of the service users . In this case 'users' might be computing agents as well as humans. If the purpose is to *control, monitor or protect* some phenomenon, then the the abstraction should focus on those phenomenon, considering in what way they should be monitoring, controlled or protected and should ignore the way in which this is achieved.

Once we get a grip on the intended purpose of a system through formalisation of the abstraction, we construct refined models to layer in various features. Refinement is a process of enriching or modifying a model in order to *augment* the functionality being modelled, or *explain how* some purpose is achieved. Refinement can be performed in a series of stages resulting in a series of models forming a refinement chain. Refinement facilitates abstraction since it allows us to *postpone* treatment of some system features to later refinement steps. Abstraction and refinement together allow us to manage system complexity in the design process.

Event-B provides a notion of *consistency* of a refinement. We use proof to verify the consistency of a refinement step while failing proof can help us *identify inconsistencies* in a refinement step. Automated tools such as model checkers and automated proof systems, as found in the Rodin system for Event-B, play a key role in improving the quality of models through identification of errors, pin-pointing of required invariants and proofs of consistency. These models and invariants in turn increase our understanding of a problem and its proposed solution and this can be achieved long before the system is implemented and tested. These lead to improvements in the quality of requirements documents, inform design decisions and provide evidence of the extent to which a system as implemented is fit for purpose.

In Event-B system behaviour is modelled discretely – discrete in the sense that the behaviour of the system is represented by the occurrence of events in discrete steps. For some systems, especially systems that monitor and control physical processes, it is sometimes appropriate to model behaviour with continuous models. The ability to do this using Event-B is a topic of on-going research.

We conclude by summarising some key messages about systems-level reasoning as outlined here:

- 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 invariant discovery and consistency proofs.
- the role of tools is to make verification as automatic as possible, helping us to pin-point errors.
- The role of model refinement is to allow us to manage complexity through multiple levels of abstraction and associated reasoning.

The readers should note that there are several other formal modelling systems besides Event-B that can be used for the kind of system level reasoning outlined in this paper. Examples include Alloy [9], ASM [3], TLA [13], VDM [11] and Z [15].

A. Overview of mathematical operators of Event-B

This appendix gives an overview of mathematical language used in the paper. It is not intended as a comprehensive introduction to the mathematical language of Event-B. For that the reader is referred to [1].

A predicate is a logical property about some variables that is either true or false depending on the values of the variables, e.g., $x < y$ is true whenever the value of x is less than y and is false otherwise.

Predicate operators

The following table presents the operators used to form predicates (P and Q represent any predicate):

Symbol	Usage	Explanation
$\wedge$	$P \wedge Q$	And: true when both P and Q are true.
$\vee$	$P \vee Q$	Or: true when either P or Q (or both) are true.
$\neg$	$\neg P$	Not: true when P is false and false otherwise.
$\Rightarrow$	$P \Rightarrow Q$	Implication: if P holds then Q must hold.
$\Leftrightarrow$	$P \Leftrightarrow Q$	Equivalence: $(P \Rightarrow Q) \wedge (Q \Rightarrow P)$
$\forall$	$\forall x \cdot P$	Forall: P is true for every value of x.
$\exists$	$\exists x \cdot P$	Exists: P is true for some value of x.

Predicates about sets

A set is an unordered collection of elements. An element is either in a set or not in that set. The following table explains the basic predicates about sets (S and T represent sets and x represents elements of sets):

Symbol	Usage	Explanation
$\in$	$x \in S$	Membership: x is an element of the set S.
$\notin$	$x \notin S$	Non-membership: shorthand for $\neg(x \in S)$.
$=$	$S = T$	Equality: S and T contain exactly the same elements.
$\subseteq$	$S \subseteq T$	Subset: all elements of S are also elements of T.

Set operators

Set operators form sets from arguments that are sets. The following table explains some of the basic set operators by explaining the conditions under which an element is in the result of applying the operator (S and T represent sets and x represents elements of sets):

Symbol	Usage	Explanation
$\cup$	$S \cup T$	Union: $x \in S \cup T \;\Leftrightarrow\; x \in S \vee x \in T$.
$\cap$	$S \cap T$	Intersection: $x \in S \cap T \;\Leftrightarrow\; x \in S \wedge x \in T$.
$\setminus$	$S \setminus T$	Difference: $x \in S \setminus T \;\Leftrightarrow\; x \in S \wedge x \notin T$.
$\{\ \}$	$\{x\}$	Singleton: set containing the single element x.
$\varnothing$	$\varnothing$	Empty set: set containing no elements.

Relations

Relations are sets of pairs, where pairs are compound elements, written $x \mapsto y$, consisting of a first element x and a second element y. The following table explains some operators associated with relations (x and y represent elements, S and T represent sets, r represents relations):

Symbol	Usage	Explanation
$\mapsto$	$x \mapsto y$	Pair: $x \mapsto y$ has first element x and second element y.
$\leftrightarrow$	$S \leftrightarrow T$	Relations: $r \in S \leftrightarrow T$ means r is a set of pairs of the form $x \mapsto y$ with $x \in S$ and $y \in T$.
dom	$dom(r)$	Domain of r: $x \in dom(r) \;\Leftrightarrow\; \exists y \cdot x \mapsto y \in r$.
ran	$ran(r)$	Range of r: $y \in ran(r) \;\Leftrightarrow\; \exists x \cdot x \mapsto y \in r$.
$[\ ]$	$r[S]$	Image of S under r: $y \in r[S] \;\Leftrightarrow\; \exists x \cdot x \in S \wedge x \mapsto y \in r$.
$\lhd$	$S \lhd r$	Domain subtraction: $x \mapsto y \in (S \lhd r) \;\Leftrightarrow\; x \mapsto y \in r \;\wedge\; x \notin S$.

Functions

Functions are special cases of relations where each domain element is mapped to exactly one range element. Because any x in the domain of a function f is mapped to a single value in the range, we write $f(x)$ for that single range value (function application). The following table explains some operators associated with functions (S and T represent sets, r represents relations):

Symbol	Usage	Explanation
$\pfun$	$S \pfun T$	Partial functions: $f \in (S \pfun T) \;\Leftrightarrow$ $\forall x, y, y' \cdot x \mapsto y \in f \;\wedge\; x \mapsto y' \in f \;\Rightarrow\; y = y'$.
$\rightarrow$	$S \rightarrow T$	Total functions: $f \in (S \rightarrow T) \;\Leftrightarrow\; f \in (S \pfun T) \;\wedge\; dom(f) = S$.
$f(x)$	$f(x)$	Function application: provided $f(x)$ is well-defined, i.e., $f \in S \pfun T \;\wedge x \in dom(f)$, then $f(x) = y \;\Leftrightarrow\; x \mapsto y \in f$.

References

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

[2] J.-R. Abrial, M. Butler, S. Hallerstede, T.S. Hoang, F. Mehta, and L. Voisin. Rodin: an open toolset for modelling and reasoning in Event-B. *STTT*, 12(6):447–466, 2010.

[3] Egon Börger and Robert F. Stärk. *Abstract State Machines. A Method for High-Level System Design and Analysis*. Springer, 2003.

[4] M. Butler and D. Yadav. An incremental development of the Mondex system in Event-B. *Formal Asp. Comput.*, 20(1):61–77, 2008.

[5] Peter Pin-Shan Chen. The entity-relationship model — toward a unified view of data. *ACM Trans. Database Syst.*, 1(1):9–36, March 1976.

[6] K. Damchoom and M. Butler. Applying event and machine decomposition to a flash-based filestore in Event-B. In *SBMF 2009*, volume 5902, pages 134–152. Springer LNCS, 2009.

[7] Asieh Salehi Fathabadi, Abdolbaghi Rezazadeh, and Michael Butler. Applying atomicity and model decomposition to a space craft system in Event-B. In Mihaela Gheorghiu Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi, editors, *NASA Formal Methods*, volume 6617 of *Lecture Notes in Computer Science*, pages 328–342. Springer, 2011.

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

[9] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis.* MIT Press, 2006.

[10] Ivar Jacobson, Grady Booch, and James Rumbaugh. *The Unified Software Development Process*. Addison Wesley Longman, 1998.

[11] C.B. Jones. *Systematic Software Development using VDM*. Prentice Hall, 1990.

[12] Warren Kuffel. Extra time saves money. *Computer Language 1990*, December 1990.

[13] Leslie Lamport. The temporal logic of actions. *ACM Trans. Program. Lang. Syst.*, 16(3):872–923, May 1994.

[14] M. Leuschel and M. Butler. ProB: An automated analysis toolset for the B Method. *Intl. J. on Software Tools for Technology Transfer*, 10(2):185–203, 2008.

[15] M.J. Spivey. *The Z Notation: A reference manual*. Prentice Hall, 1992.

[16] Sanaz Yeganefard, Michael Butler, and Abdolbaghi Rezazadeh. Evaluation of a guideline by formal modelling of cruise control system in Event-B. In César Muñoz, editor, *NASA Formal Methods*, volume NASA/CP-2010-216215 of *NASA Conference Proceedings*, pages 182–191, 2010.