An Introduction to the Event-B Modelling Method

Thai Son Hoang

Abstract This chapter is a short introduction to the Event-B modelling method for discrete transition systems. Important mechanisms for the step-wise development of the formal models, such as *context extension* and *machine refinement*, are discussed. Consistency of the models are presented in terms of proof obligations and are illustrated by concrete examples.

1 Introduction

Event-B [2] is a modelling method for formalising and developing systems whose components can be modelled as discrete transition systems. An evolution of the (classical) B-method [1], Event-B is now centred around the general notion of *events*, which are also found in other formal methods such as Action Systems [4], TLA [6] and UNITY [5].

Event-B models are organised in terms of the two basic constructs: *contexts* and *machines*. Contexts specify the static part of a model whereas machines specify the dynamic part. The role of the contexts is to isolate the parameters of a formal model and their properties, which are assumed to hold for all instances. A machine encapsulates a transition system with the state being specified by a set of variables and transitions modelled by a set of guarded events.

Event-B allows models to be developed gradually via mechanisms such as *context extension* and *machine refinement*. These techniques enable users to develop target systems from their abstract specifications, and subsequently introduce more implementation details. More importantly, properties that are proved at the abstract level are maintained through refinement, hence are guaranteed to be satisfied also by later refinements. As a result, correctness proofs of systems are broken down and distributed amongst different levels of abstraction, which are easier to manage.

Thai Son Hoang
Institute of Information Security, ETH-Zurich, Switzerland e-mail: htson@inf.ethz.ch

The rest of this chapter is structured as follows. We give some brief overview of the Event-B mathematical language in Section 2. In Section 3, we give an informal description of our running example. Subsequently, we show the basic constructs of Event-B in Section 4 (contexts) and Section 5 (machines). We present the mechanisms for context extension in Section 6 and machine refinement in Section 7.

2 The Event-B Mathematical Language

The basis for the formal models in Event-B is first-order logic and a typed set-theory. We are not going to give a full details of the Event-B logic here. For more information, we refer the reader to [2, 8]. We present some main ingredients of the Event-B mathematical language that are important for understanding the Event-B models of the example in the sequel.

The first-order logic of Event-B contains standard operators such as *conjunction* ($\wedge$), *disjunction* ($\vee$), *implication* ($\Rightarrow$), *negation* ($\neg$), *equivalence* ($\Leftrightarrow$), *universal quantification* ($\forall$), and *existential quantification* ($\exists$). Two constants are defined, namely $\top$ and $\bot$ denoting *truth* and *falsity*, respectively.

2.1 Set Theory

An important part of the mathematical language is its set theoretical notation, with the introduction of the membership predicate $E \in S$, meaning that expression E is a member of set S. A set expression can be a variable (depending on its type). Moreover, a set can be explicitly defined by listing its members (*set extension*), *e.g.*, $\{E_1, \ldots, E_n\}$. Other basic set constructs include Cartesian product, power set, and set comprehension. Given two set expressions S and T, the *Cartesian product* of S and T, denoted as $S \times T$, is the set of mappings (ordered pairs) $x \mapsto y$ where $x \in S$ and $y \in T$. The *power set* of S, denoted as $\mathbb{P}(S)$, is the set of all sub-sets of S. Finally, given a list of variables x, a predicate P constraining x and an expression E depending on x, the *set comprehension* $\{x \cdot P \mid E\}$ is the set of elements E, where P holds.

A key feature of the Event-B set theoretical notation is the models of relations as sets of mappings. Subsequently, different types of relations and functions are also defined as sets of mappings with different additional properties. Given two set expressions S and T, $S \leftrightarrow T$ denotes the set of all *binary relations* from S to T. Similarly, $S \nrightarrow T$ denotes the set of all *partial functions* from S to T, and $S \rightarrow T$ denotes the set of all *total functions* from S to T. Definition of these relations can be seen below, expressed using set memberships.

Construct	Definition
$r \in S \leftrightarrow T$	$r \in \mathbb{P}(S \times T)$
$f \in S \nrightarrow T$	$f \in S \leftrightarrow T \wedge (\forall x, y_1, y_2 \cdot x \mapsto y_1 \in f \wedge x \mapsto y_2 \in f \Rightarrow y_1 = y_2)$
$f \in S \rightarrow T$	$f \in S \nrightarrow T \wedge (\forall x \cdot x \in S \Rightarrow (\exists y \cdot x \mapsto y \in f))$

Intuitively, a binary relation r from S to T is a set of mappings $x \mapsto y$ where $x \in S$ and $y \in T$. A partial function f from S to T is a binary relation from S to T where each element x in S has *at most* one mapping to T. A total function f from S to T is a partial function from S to T where each element x in S has *exactly* one mapping to T.

2.2 Types

Variables in Event-B are strongly typed. A type in Event-B can be some built-in type (*e.g.*, BOOL, $\mathbb{Z}$) or some user-defined type. Moreover, given T, T_1 and T_2 being types, the Cartesian product $T_1 \times T_2$ and the power set $\mathbb{P}(T)$ are also types. In contrast with most strongly typed programming languages, the type of variables are not presented when they are declared. Instead, they are inferred from constraining properties of variables. Typically, a property of the form $x \in E$ where E is of type $\mathbb{P}(T)$ allows us to infer that x has the type T.

2.3 Well-definedness

Event-B requires that every formula to be *well-defined* [7, 8]. Informally, one has to prove that partial functions (either pre-defined, *e.g.*, division $\div$, or user-defined) are never evaluated outside of their domain. Ill-defined expressions, (such as $x \div 0$) should be avoid. A syntactic operator $\mathscr{L}$ is used to map formulae to their corresponding well-definedness conditions. Table 1 shows the definition of well-definedness condition using $\mathscr{L}$ for some formulae in Event-B. Here x is some variable, P and Q are predicates, E, E_1, E_2 are some expressions, and f is a binary relation from S to T. Moreover $\mathrm{dom}(f)$ denotes the domain of f, *i.e.*, the set of all elements in S that connect to some element in T.

Notice that by using $\mathscr{L}$, we assume a *well-definedness order* from left to right (*e.g.*,, for $P \wedge Q$) for formulae (this is similar to evaluating conditional statements, *e.g.*, && or ||, in several programming languages).

Formula	Well-definedness condition
x	$\top$
$\neg P$	$\mathscr{L}(P)$
$P \wedge Q$	$\mathscr{L}(P) \wedge (P \Rightarrow \mathscr{L}(Q)$
$\forall x \cdot P$	$\forall x \cdot \mathscr{L}(P)$
$E_1 \div E_2$	$\mathscr{L}(E_1) \wedge \mathscr{L}(E_2) \wedge E_2 \neq 0$
$E_1 \leq E_2$	$\mathscr{L}(E_1) \wedge \mathscr{L}(E_2)$
$\text{card}(E)$	$\mathscr{L}(E) \wedge \text{finite}(E)$
$f(E)$	$\mathscr{L}(E) \wedge f \in S \mathrel{+\!\!\!\rightarrow} T \wedge E \in \text{dom}(f)$

Table 1 Calculating well-definedness conditions using $\mathscr{L}$

2.4 Sequents

Event-B defines *proof obligations*, which must be proved to show that formal models fulfil their specified properties. Often these verification conditions are expressed in term of *sequents*. A sequent $H \vdash G$ means that the *goal* G holds under the assumption of the set of *hypotheses* H. The obligations are discharged using some inference rules which we will not describe here. Instead, we will give some informal justifications on how the proof obligations can be discharged. The purpose of presenting to proof obligations within this chapter is to illustrate various conditions need to be proved to maintain consistency of the example.

3 Example. A Course Management System

The running example that we are going to use for illustrating Event-B is a course management system. We describe a requirements document of the system as follows. A club has some fixed *members*, amongst them are *instructors* and *participants*. Note that a member can be both an instructor and a participant.

ASM 1	Instructors are members of the club.

ASM 2	Participants are members of the club.

There are pre-defined *courses* that can be offered by a club. Each course is associated with exactly one fixed instructor.

ASM 3	There are pre-defined courses.

ASM 4	Each course is assigned to one fixed instructor.

A course is either *opened* or *closed* and is managed by the system.

REQ 5	A course is either *opened* or *closed*.

REQ 6	The system allows to open a closed course.

REQ 7	The system allows to close an opened course.

The number of opened courses is limited.

REQ 8	The number of opened courses cannot exceed a given limit.

Only when a course is opened, participants can *register* for the course. An important constraint for registration is that an instructor cannot attend his own courses.

REQ 9	Participants can only register for an opened course.

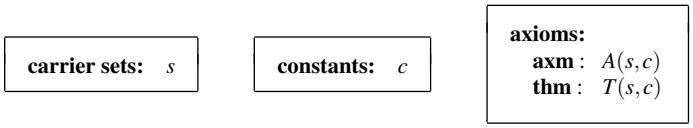
REQ 10	Instructors cannot attend their own courses.

In subsequent sections, we develop a formal model according to the above requirements document. In particular, we will give backward pointers to the above requirements, in order to justify how they are formalised in the Event-B model.

4 Contexts

A context may contain *carrier sets*, *constants*, *axioms*, and *theorems*. Carrier sets are user-defined types. By convention, a carrier set s is *non-empty*, *i.e.*, satisfying $s \neq \varnothing$, and *maximal*, *i.e.*, satisfying $\forall x \cdot x \in s$. Constants c denote *static objects* within the

development[1]. Axioms are *presumed properties* of carrier sets and constants. Theorems are *derived properties* of carrier sets and constants. Carrier sets and constants model parameters of the development. Moreover, axioms state parameters' properties which are assumed to hold for all possible instances of them. A context C with carrier sets s, constants c, axioms $A(s,c)$, and theorems $T(s,c)$ can be presented as follows.



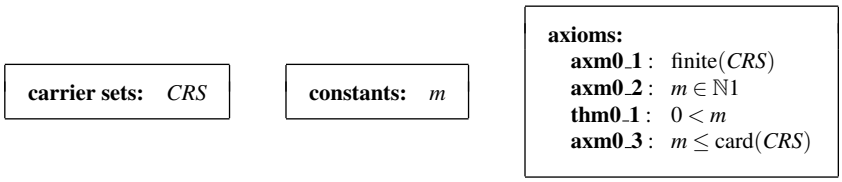
Note that we present axioms and theorems using different *labels*, *i.e.*, **axm** and **thm**. Later on, we also use different labels for other modelling elements, such as, invariants (**inv**), guards (**grd**) and actions (**act**).

Proof obligations are generated to ensure that the theorems are derivable from *previously* defined axioms. This is captured by the following proof obligation rule called THM.

$$A(s,c) \vdash T(s,c)\ . \qquad \text{(THM)}$$

4.1 Example. Context coursesCtx

In this initial model, we focus on the opening and closing of courses by the system. As a result, our initial context coursesCtx contains a carrier set *CRS* denoting the set of courses that can be offered by the club (ASM 3). Moreover, coursesCtx includes a constant m denoting the maximum number of courses that the club can have at the same time (with respect to requirement REQ 8). The context coursesCtx is as follows.



Note that we label the axioms and theorems with the prefixes denoting the role of the modelling elements, *i.e.*, **axm** and **thm**, with some numbers. For example, **axm0_1** denotes the first (*i.e.*, 1) axiom for the initial model (*i.e.*, 0). We apply this systematic labelling through out our development.

The assumption on *CRS* and m are captured by the axioms and theorems as follows. Axiom **axm0_1** states that *CRS* is finite. Axiom **axm0_2** states that m is a member of the set of natural numbers (i.e., m is a natural number). Finally, axiom **axm0_3** states that m cannot exceed the number of possible courses that can be

[1] When referring to carrier sets s and constants c, we usually allow for multiple carrier sets and constants, i.e., they may be "vectors".

offered by the club, represented as card(CRS), the cardinality of CRS. A derived property of m is presented as theorem **thm0_1**.

thm0_1/THM

A proof obligation is generated for **thm0_1** as follows. Notice that **axm0_3** does not appear in the set of hypotheses for the obligation, since it is declared after **thm0_1**. By convention, each proof obligation is labelled according to the element involved and the name of the proof obligation rule. Here **thm0_1**/THM indicates that it is a THM proof obligation for **thm0_1**.

finite(CRS) $m \in \mathbb{N}1$ $\vdash$ $0 < m$	**thm0_1**/THM

The obligations can be trivially discharged since $\mathbb{N}1$ is the set of all positive natural numbers, *i.e.*, $\{1,2,\ldots\}$.

axm0_3/WD

It is required to prove that **axm0_3** is well-defined. The corresponding proof obligation is as follows.

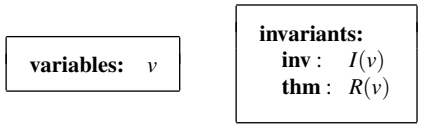
finite(CRS) $m \in \mathbb{N}$ $0 \leq m$ $\vdash$ finite(CRS)	**thm0_1**/WD

Since the goal appears amongst the hypotheses, the proof obligation can be discharged trivially. Note that the order of appearance of the axioms is important. In particular, **axm0_1** needs to be declared before **axm0_3**.

5 Machines

Machines specify behavioural properties of Event-B models. In order to have access to information of context C defined in Section 4, machine M must connect with C. When machine M *sees* context C, it has access to C's carrier sets s and constants c, *e.g.*, to refer to them when modelling, and C's axioms $A(s,c)$ and theorems $T(s,c)$, *e.g.*, to use them as assumptions during proving. For clarity in the following presentation, we do not refer explicitly to the modelling elements of C. Note that in general a machine can see several contexts.

Machines M may contain *variables*, *invariants*, *theorems*, *events*, and a *variant*. Variables v define the *state* of a machine and are *constrained* by invariants $I(v)$. Theorems $R(v)$ are *additional properties* of v derivable from $I(v)$.



A proof obligation (also called THM) is generated to prove that the theorem $R(v)$ is derivable from $I(v)$.

$$I(v) \vdash R(v) \qquad \text{(THM)}$$

Possible state changes are described by events (see Section 5.1). The variant is used to prove convergence properties of events (see Section 5.2).

5.1 Events

An event e can be represented by the term

$$\mathsf{e} \mathrel{\widehat{=}} \textbf{any } x \textbf{ where } G(x,v) \textbf{ then } Q(x,v) \textbf{ end} \quad , \tag{1}$$

where x stands for the event's *parameters*[2], $G(x,v)$ is the *guard* (the conjunction of one or more predicates) and $Q(x,v)$ is the *action*. The guard states the necessary condition under which an event may occur. The event is said to be *enable* in some state, if there exists some value for its parameter x that makes its guard $G(x,v)$ holds in that state. The action describes how the state variables evolve when the event occurs. We use the short form

$$\mathsf{e} \mathrel{\widehat{=}} \textbf{when } G(v) \textbf{ then } Q(v) \textbf{ end} \tag{2}$$

when the event does not have any parameters, and we write

$$\mathsf{e} \mathrel{\widehat{=}} \textbf{begin } Q(v) \textbf{ end} \tag{3}$$

when, in addition, the event's guard always holds (*i.e.*, equals to $\top$). A dedicated event in the form of (3) is used for the *initialisation* event (usually represented as init). Note that events may be annotated with their convergence status, which we will look at in Section 5.2.

The action of an event is composed of one or more *assignments* of the form

$$a := E(x,v) \tag{4}$$

or

[2] When referring to variables v and parameters x, we usually allow for multiple variables and parameters, i.e., they may be "vectors".

$$a :\in E(x,v) \tag{5}$$

or

$$a :| \ P(x,v,a') \ , \tag{6}$$

where a are some of the variables contained in v, $E(x,v)$ is an expression, and $P(x,v,a')$ is a predicate. Note that the variables on the left-hand side of the assignments contained in an action must be disjoint. In (5), a must be a single variable, whereas it can be a vector of variables in (4) and (6). In particular, in (4), if a is a vector containing $n > 0$ variables, then E must also be a vector of expressions, one for each of the n variables. Assignments of the form (4) are *deterministic*, whereas the other two forms are *nondeterministic*. In (5), a is assigned an element of a set $E(x,v)$. (6) refers to P which is a *before-after predicate* relating the values v (before the action) and a' (afterwards). (6) is also the most general form of assignment and nondeterministically selects an after-state a' satisfying P and assigns it to a. Note that the before-after predicates for the other two forms are as expected; namely, $a' = E(x,v)$ and $a' \in E(x,v)$, respectively.

All assignments of an action $Q(x,v)$ occur simultaneously, which is expressed by conjoining together their before-after predicates. Hence each event corresponds to a before-after predicate $\boldsymbol{Q}(x,v,v')$ established by conjoining all before-after predicates associated with each assignment and $b = b'$, where b are unchanged variables. Note that the initialisation init therefore corresponds to an *after predicate* $\boldsymbol{K}(v')$, since there are no states before initialisation.

5.1.1 Proof Obligations

We describe below some important proof obligation rules for Event-B machines, namely, invariant establishment and preservation, and action feasibility.

Invariant Establishment and Preservation

An essential feature of an Event-B machine M is its invariant $I(v)$. It shows properties that hold in every reachable states of the machine. Obviously, this does not hold priori for any machines and invariants, therefore must be proved. A common technique for proving an invariant property is to prove it by induction: (1) to prove that the property is established by the initialisation init (*invariant establishment*), and (2) to prove that the property is maintained whenever variables change their values (*invariant preservation*).

Invariant establishment states that any possible state after initialisation given by the after predicate $\boldsymbol{K}(v')$ must satisfy the invariant I. The proof obligation rule is as follows.

$$\boldsymbol{K}(v') \ \vdash \ I(v') \tag{INV}$$

Invariant preservation requires to prove that every event occurrences *re-establish* the invariants I. More precisely, for every event e, under the assumption of the invariants I and e's guard G, we must prove that the invariants still hold in any possible state after the event's execution given by the before-after predicate $\boldsymbol{Q}(x, v, v')$. The proof obligation rule is as follows.

$$I(v), G(x, v), \boldsymbol{Q}(x, v, v') \vdash I(v') \tag{INV}$$

Note that in practice, by the property of conjunctivity, we can prove the establishment and preservation of each invariant separately.

Feasibility

Feasibility states that the action of an event is always feasible whenever the event is enabled. In other words, there are always possible after values for the variables, satisfying the before-after predicate. In practice, we prove feasibility for individual assignment of the event action. For deterministic assignments, feasibility holds trivially. The feasibility proof obligation generated for a non-deterministic assignment of the form $a :| P(x, v, a')$ is as follows.

$$I(v), G(x, v) \vdash \exists a' \cdot P(x, v, a') \tag{FIS}$$

5.2 Event Convergence

A set of events can be proved to collectively converge. In other words, these events cannot take control forever and hence one of the other events eventually occurs. We call these events *convergent*. To prove this, one gives a *variant* V, which maps a state v to a natural number. One then proves that each convergent event strictly decreases V. More precisely, let e be a convergent event, where v is the state before executing e and v' is the state after. Then for each such e, v, and v', one proves that $V(v') < V(v)$, under the additional assumptions of all invariants and of the guard of e. Since the variant maps a state to a natural number, V induces a well-founded ordering on system states given by the strictly less than order ($<$) of their images under V. The following proof obligation rules apply to every convergent event, where VAR concerns with the decrement of the variant and NAT ensures that the variant is a natural number when the event is enabled.

$$I(v), G(x, v), \boldsymbol{Q}(x, v, v') \vdash V(v') < V(v) \tag{VAR}$$

$$I(v), G(x, v) \vdash V(v) \in \mathbb{N} \tag{NAT}$$

Note that in some cases the convergence of some events cannot be immediately shown, but only in a later refinement. In this case, their convergence is *anticipated*

and we must prove that $V(v') \leq V(v)$, that is, these anticipated events do not increase the variant. Anticipated events must obey NAT, and the following proof obligation rule, also called VAR.

$$I(v), G(x,v), \boldsymbol{Q}(x,v,v') \vdash V(v') \leq V(v) \qquad \text{(VAR)}$$

As mentioned above, the variant V is a natural number. Alternatively, V can be a *finite* set expression. In this case, for convergent events, one has to prove that it decreases the variant according to the strict subset-inclusion $\subset$ ordering. For anticipated events, we ensure that these events do not increase the variant by proving that $V(v') \subseteq V(v)$. The proof obligation rule VAR is adapted accordingly.

For proving that the variant V is a finite set, the following proof obligation rule called FIN applies.

$$I(v) \vdash \text{finite}(V(v)) \qquad \text{(FIN)}$$

Note that FIN needs to be proved once, *i.e.*, does not depend on the set of convergent and anticipated events (cf. NAT).

The convergence attribute of an event is denoted by the keyword **status** with three possible values: *convergent*, *anticipated*, and *ordinary* (for events which are neither convergent nor anticipated). Events are *ordinary* by default.

5.3 Deadlock-freeness

A machine M is said to be *deadlocked* in some state if all of its events are disabled in that state. *Deadlock-freeness* for M ensures that there are always some enabled events during M's execution. Assume that M contains a set of n events e_i ($i \in 1 \ldots n$) of the following form.

$$\mathsf{e_i} \mathrel{\widehat{=}} \textbf{any } x_i \textbf{ where } G_i(x_i, v) \textbf{ then } Q_i(x_i, v) \textbf{ end}$$

The proof obligation rule for deadlock-freeness[3] is as follows.

$$I(v) \vdash (\exists x_1 \cdot G_1(x_1, v)) \lor \ldots \lor (\exists x_n \cdot G_n(x_n, v)) \qquad \text{(DLF)}$$

5.4 Example. Machine m0

We develop machine m0 of the initial model, focusing on courses opening and closing. This machine sees context coursesCtx as developed in Section 4.1, hence as a result has access to the carrier set *CRS* and constant *m*. We model the set of opened courses by a variable, namely *crs* (REQ 5). Invariant **inv0_1** states that it is a subset of available courses *CRS*. A consequence of this invariant and of axiom **axm0_1** is

[3] Typically, this is encoded as a theorem in the machine after all invariants.

that *crs* is finite, and this is stated in m0 as theorem **thm0_2**. Requirement REQ 8 is directly captured by invariant **inv0_2**: the number of opened courses, *i.e.*, card(*crs*) is bounded above by *m*. Initially, all courses are closed hence *crs* is set to the empty set ($\varnothing$).

variables: *crs*

invariants:
inv0_1 : $crs \subseteq CRS$
thm0_2 : finite(crs)
inv0_2 : $\text{card}(crs) \leq m$

init
begin
$crs := \varnothing$
end

We model the opening and closing of courses using two events OpenCourses and CloseCourses as follows (REQ 6 and REQ 7).

OpenCourses
status *ordinary*
when
grd0_1 : $\text{card}(crs) \neq m$
thm0_3 : $crs \neq CRS$
then
act0_1 : $crs :| crs \subset crs' \wedge \text{card}(crs') \leq m$
end

CloseCourses
status *anticipated*
any *cs* **where**
grd0_1 : $cs \subseteq crs$
grd0_2 : $cs \neq \varnothing$
then
act0_1 : $crs := crs \setminus cs$
end

We choose purposely to model these events using different features of Event-B. In OpenCourses, we use a nondeterministic action to model the fact that some new courses are opened, *i.e.*, $crs \subset crs'$, as long as the number of opened courses will not exceed its limit, *i.e.*, $\text{card}(crs') \leq m$. The guard of the event states that the current number of opened courses has not yet reached the limit.

CloseCourses models the set of courses that are going to be closed using parameter *cs*. It is a non-empty set of currently opened courses which is captured by CloseCourses' guard. The action is modelled straightforwardly by removing *cs* out of the set *crs*.

We set the convergence status for OpenCourses and CloseCourses to be *ordinary* and *anticipated*, respectively. We delay the reasoning about the convergence of CloseCourses to later refinements. Our intention is to prove that there can be only finitely many occurrences of CloseCourses between any two OpenCourses events.

5.4.1 Proof obligations

We present some of the obligations to illustrate what needs to be proved for the consistency of m0. We applied the proof obligation rules as showed earlier in this Section. Notice that we can take the axioms and theorems of the seen context coursesCtx as hypotheses in the proof obligations. For clarity, we show only parts of the hypotheses that are relevant for discharging the proof obligations. Moreover, we also show the proof obligations in their simplified form, *e.g.*, when the events' assignments are deterministic.

thm0_2/THM

This obligation corresponds to the rule THM, in order to ensure that **thm0_2** is derivable from previously declared invariants.

$$\frac{\begin{array}{l}\ldots\\ \text{finite}(CRS)\\ crs \subseteq CRS\\ \vdash\\ \text{finite}(crs)\end{array}}{\textbf{thm0\_2}/\text{THM}}$$

The proof obligation holds trivially since *crs* is a subset of a finite set, *i.e.*, *CRS*.

init/**inv0_2**/INV

This obligation ensures that the initialisation init establishes invariant **inv0_2**.

$$\frac{\begin{array}{l}\ldots\\ 0 \leq m\\ \vdash\\ \text{card}(\varnothing) \leq m\end{array}}{\text{init}/\textbf{inv0\_2}/\text{INV}}$$

Since the cardinality of the empty set $\varnothing$ is 0, the proof obligation holds trivially.

OpenCourses/**thm0_3**/THM

This obligation ensures that **thm0_3** is derivable from the invariants and the previously declared guards of OpenCourses.

$$\frac{\begin{array}{l}\ldots\\ m \leq \text{card}(CRS)\\ crs \in \mathbb{P}(CRS)\\ \text{card}(crs) \leq m\\ \text{card}(crs) \neq m\\ \vdash\\ crs \neq CRS\end{array}}{\text{OpenCourses}/\textbf{thm0\_3}/\text{THM}}$$

Informally, we can derive from the hypotheses that $\text{card}(crs) < \text{card}(CRS)$, hence *crs* must be different from *CRS*.

OpenCourses/**act0_1**/FIS

This obligation corresponds to rule FIS and ensures that the nondeterministic assignment of OpenCourses is feasible when the event is enabled.

$$\begin{array}{l} \ldots \\ crs \neq CRS \\ \mathrm{card}(crs) \leq m \\ \mathrm{card}(crs) \neq m \\ \vdash \\ \exists crs' \cdot crs \subset crs' \wedge \mathrm{card}(crs') \leq m \end{array} \quad \Big| \quad \mathsf{OpenCourses/\textbf{act0\_1}/FIS}$$

The reasoning about the proof obligation is as follows. Since *crs* is different from *CRS*, there exists an element *c* which is closed, *i.e.*, not in *crs*. By adding *c* to the set of opened courses, we strictly increase the number of opened courses by 1. Moreover, the number of opened courses after executing the event is still within the limit since originally it is strictly below the limit.

CloseCourses/**inv0_2**/INV

This obligation corresponds to rule INV and is simplified accordingly since the assignment is deterministic. The purpose of the obligation is to prove that CloseCourses maintains invariant **inv0_2**.

$$\begin{array}{l} \ldots \\ \mathrm{card}(crs) \leq m \\ \vdash \\ \mathrm{card}(crs \setminus cs) \leq m \end{array} \quad \Big| \quad \mathsf{CloseCourses/\textbf{inv0\_2}/INV}$$

Since removing some courses *cs* from the set of opened courses *crs* can only reduce its number, the proof obligation can be trivially discharged.

DLF/THM

The deadlock-freeness condition is encoded as theorem **DLF** of machine m0, which results in the following proof obligation.

$$\begin{array}{l} \ldots \\ 0 < m \\ \vdash \\ (\mathrm{card}(crs) \neq m) \ \vee \ (\exists cs \cdot cs \subseteq crs \wedge cs \neq \varnothing) \end{array} \quad \Big| \quad \mathsf{\textbf{DLF}/THM}$$

We reason as follows. If $\mathrm{card}(crs) \neq m$, the goal trivially holds. Otherwise, *i.e.*, $\mathrm{card}(crs) = m$, since $m \neq 0$, we have that $crs \neq \varnothing$. As a result, we can prove that $\exists cs \cdot cs \subseteq crs \wedge cs \neq \varnothing$ by instantiating *cs* with *crs* itself.

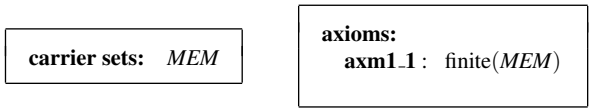
6 Context Extension

Context extension is a mechanism for introducing more static details into an Event-B development. A context can *extend* one or more contexts. When mentioning that a context D extends another context C, we call C and D the abstract context and concrete context, respectively. By extending C, D "inherits" all the abstract elements of C, *i.e.*, carrier sets, constants, axioms and theorems. This means that (1) a context extending D also implicitly extends C, (2) a machine seeing D also implicitly sees C. As a result, proof obligation rule THM for D also has additional assumptions in the form of the axioms and theorems from the abstract context C.

Subsequently, we present three new contexts that we use in the next refinement of our running example.

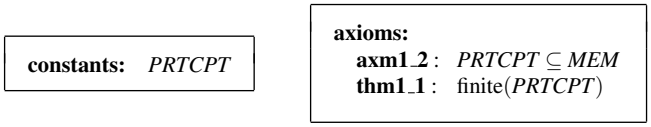
6.1 Context membersCtx

This is an initial context (*i.e.*, does not extend any other context) containing a carrier sets *MEM*. *MEM* represents the set of club members, with an axiom stating that it is finite.

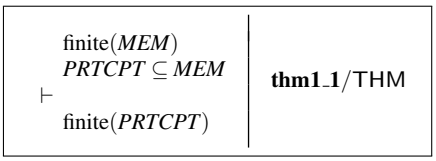


6.2 Context participantsCtx

This context extends the previously defined context membersCtx and is as follows.



Constant *PRTCPT* denotes the set of participants which must be members of the club as specified in ASM 2 (**axm1_2**). Theorem **thm1_1** states that there can be only a finite number of participants, which gives rise to the following trivial proof obligation.



An important point is that axiom **axm1_1** of the abstract context membersCtx appears as a hypothesis in the proof obligation.

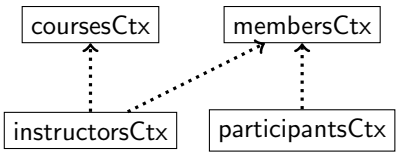
6.3 Context instructorsCtx

This context extends two contexts coursesCtx and membersCtx, and introduces two constants, namely *INSTR* and *instrs*. *INSTR* models the set of instructors which are members of the club as specified by ASM 1 (**axm1_3**). Constant *instrs* models the relationship between courses and instructors and is constrained by **axm1_4**: it is a *total function* from *CRS* to *INSTR*, hence directly formalises requirement ASM 4. Recall the definition of a total function f from a set S to a set T: f is a relation from S to T where every element in S has exactly one mapping to some element in T.

constants: $INSTR, instrs$

axioms:
axm1_3 : $INSTR \subseteq MEM$
axm1_4 : $instrs \in CRS \rightarrow INSTR$

The hierarchy of context extensions for our example is summarised in the following diagram.



7 Machine Refinement

Machine refinement is a mechanism for introducing details about the dynamic properties of a model [2]. For more details on the theory of refinement, we refer the reader to the Action System formalism [4], which has inspired the development of Event-B. We present here the proof obligations defined in Event-B related to refinement. When speaking about a machine N refining another machine M, we refer to M as the *abstract* machine and N as the *concrete* machine.

Despite the fact that the formal definition of Event-B refinement does not distinguish between *superposition refinement* and *data refinement*, we illustrate them in separate sections to show different aspects of the two. In superposition refinement, the abstract variables of M are retained in the concrete machine N, with possibly some additional concrete variables. In data refinement, the abstract variables v are replaced by concrete variables w and, subsequently, the connection between M and N are represented by the relationship between v and w. In fact, more often, Event-B refinement is a mixture of both superposition and data refinement: some abstract variable are retained, while the others are replaced by new concrete variables.

7.1 Superposition Refinement

As mentioned earlier, in superposition refinement, variables v of the abstract machine M are kept in the refinement, *i.e.*, as part of the state of N. Moreover, N can have some additional variables w. The concrete invariants $J(v,w)$ specify the relationship between the old and new variables. Each abstract event e is refined by a concrete event f (later on we will relax this one-to-one constraint). Assume that the abstract event e and the concrete event f are as follows.

$$\mathsf{e} \mathrel{\widehat{=}} \textbf{any } x \textbf{ where } G(x,v) \textbf{ then } Q(x,v) \textbf{ end}$$
$$\mathsf{f} \mathrel{\widehat{=}} \textbf{any } x \textbf{ where } H(x,v,w) \textbf{ then } R(x,v,w) \textbf{ end}$$

We assume now that e and f have the same parameters x. The more general case, where the parameters are different, is presented in Section 7.2.

Somewhat simplifying, we say that f refines e if the guard of f is stronger than that of e (*guard strengthening*), concrete invariants J are maintained by f, and abstract action Q simulates the concrete action R (*simulation*). These conditions are stated as the following proof obligation rules.

$$I(v), J(v,w), H(x,v,w) \vdash G(x,v) \qquad \text{(GRD)}$$

$$I(v), J(v,w), H(x,v,w), \boldsymbol{R}(x,v,w,v',w') \vdash \boldsymbol{Q}(x,v,v') \qquad \text{(SIM)}$$

$$I(v), J(v,w), H(x,v,w), \boldsymbol{R}(x,v,w,v',w') \vdash J(v',w') \qquad \text{(INV)}$$

In particular, if the guard and action of an abstract event are retained in the concrete event, the proof obligations GRD and SIM are trivial, hence we only need to consider INV for proving that the gluing invariants are re-established.

Proof obligations are generated to ensure that each assignment of concrete event f is feasible. In the case where the action of the abstract event is retained in f, we only need to prove the feasibility of any additional assignment.

In the course of refinement, *new events* are often introduced into a model. New events must be proved to refine the implicit abstract event SKIP, which does nothing, *i.e.*, does not modify abstract variables v.

7.1.1 Machine m1

Machine m1 sees contexts instructorsCtx and participantsCtx. As a result, it implicitly sees coursesCtx and membersCtx. Variable *crs* is retained in this refinement. An additional variable *prtcpts* representing information about course participants is introduced. Invariant **inv1_1** models *prtcpts* as a relation between the set of opened courses *crs* and the set of participants *PRTCPT*. Requirement REQ 10 is directly modelled by invariant **inv1_2**: for every opened course c, the instructor of that course, *i.e.*, $instrs(c)$, is not amongst its participants, represented by $prtcpts[\{c\}]$.

variables: $crs, prtcpts$

invariants:
inv1_1 : $prtcpts \in crs \leftrightarrow PRTCPT$
inv1_2 : $\forall c \cdot c \in crs \Rightarrow instrs(c) \notin prtcpts[\{c\}]$

init
begin
...
$prtcpts := \varnothing$
end

Initially, there are no opened courses hence *prtcpts* is assigned to be $\varnothing$.

The original abstract event OpenCourses stays unchanged in this refinement, while an additional assignment is added to CloseCourses to update *prtcpts* by removing the information about the set of closing courses *cs* from it.

CloseCourses
status *anticipated*
any *cs* **where**
...
then
...
act1_2 : $prtcpts := cs \lhd prtcpts$
end

A new event is added, namely Register, to model the registration of a participant p for an opened course c. The guard of the event ensures that p is not the instructor of the course (**grd1_3**) and is not yet registered for the course (**grd1_4**). The action of the event update *prtcpts* accordingly by adding the mapping $c \mapsto p$ to it.

Register
status *convergent*
any p, c **where**
grd1_1 : $p \in PRTCPT$
grd1_2 : $c \in crs$
grd1_3 : $p \neq instrs(c)$
grd1_4 : $c \mapsto p \notin prtcpts$
then
act1_1 : $prtcpts := prtcpts \cup \{c \mapsto p\}$
end

We attempt to prove that Register is convergent and CloseCourses is anticipated using the following variant.

variant: $(crs \times PRTCPT) \setminus prtcpts$

The variant is a set of mappings, each links an opened course to a participant who has *not* registered for the respective course.

We present some of the important proof obligations for m1. For events OpenCourses and CloseCourses, proof obligations GRD and SIM are trivial. Consequently, we only need to consider INV for these old events.

CloseCourses/**inv1_2**/INV

This obligation is to ensure that **inv1_2** is maintained by CloseCourses. The obligation is trivial, in particular, given that $c \notin cs$, we have $(cs \lhd prtcpts)[\{c\}]$ is the same as $prtcpts[\{c\}]$.

$$\begin{array}{l} \ldots \\ \forall c \cdot c \in crs \Rightarrow instrs(c) \notin prtcpts[\{c\}] \\ \vdash \\ \forall c \cdot c \in crs \setminus cs \Rightarrow instrs(c) \notin (cs \lhd prtcpts)[\{c\}] \end{array} \quad \Big| \quad \text{CloseCourses/}\mathbf{inv1\_2}\text{/INV}$$

Register/**inv1_1**/INV

This obligation is to guarantee that **inv1_1** is maintained by the new event Register.

$$\begin{array}{l} \ldots \\ prtcpts \in crs \leftrightarrow PRTCPT \\ p \in PRTCPT \\ c \in crs \\ \vdash \\ prtcpts \cup \{c \mapsto p\} \in crs \leftrightarrow PRTCPT \end{array} \quad \Big| \quad \text{Register/}\mathbf{inv1\_1}\text{/INV}$$

FIN

This obligation is to ensure that the declared variant used for proving convergence of events is finite (FIN). This is trivial, since the set of opened course *crs* and the set of participants *PRTCPT* are both finite.

$$\begin{array}{l} \ldots \\ \text{finite}(crs) \\ \text{finite}(PRTCPT) \\ \vdash \\ \text{finite}((crs \times PRTCPT) \setminus prtcpts) \end{array} \quad \Big| \quad \text{FIN}$$

CloseCourses/VAR

This proof obligation corresponds to rule VAR ensuring that anticipated event CloseCourses does not increase the variant.

$$\begin{array}{l} \ldots \\ \vdash \\ \quad ((crs \setminus cs) \times PRTCPT) \setminus (cs \lhd prtcpts) \\ \quad \subseteq \\ \quad (crs \times PRTCPT) \setminus prtcpts) \end{array} \quad \Big| \quad \text{CloseCourses/VAR}$$

Register/VAR

This proof obligation corresponds to rule VAR to ensure that the convergent event Register decreases the variant. This is trivial since a new mapping $c \mapsto p$ is added to *prtcpts*, effectively increasing *prtcpts*, hence strictly decreasing the variant.

$$\begin{array}{l} \ldots \\ c \mapsto p \notin prtcpts \\ \vdash \\ \quad (crs \times PRTCPT) \setminus (prtcpts \cup \{c \mapsto p\})) \\ \quad \subset \\ \quad (crs \times PRTCPT) \setminus prtcpts) \end{array} \quad \Big| \quad \text{Register/VAR}$$

7.2 Data Refinement

In data refinement, abstract variables v are removed and replaced by concrete variables w. The states of abstract machine M are related to the states of concrete machine N by *gluing invariants* $J(v,w)$. In Event-B, the gluing invariants J are declared as invariants of N and also contain the *local* concrete invariants, *i.e.*, those constraining only concrete variables w.

Again, we assume the one-to-one correspondence between an abstract event e and a concrete event f. Let e and f be as follows[4].

$$\mathsf{e} \mathrel{\widehat{=}} \textbf{any } x \textbf{ where } G(x,v) \textbf{ then } Q(x,v) \textbf{ end}$$
$$\mathsf{f} \mathrel{\widehat{=}} \textbf{any } y \textbf{ where } H(y,w) \textbf{ then } R(y,w) \textbf{ end}$$

Similar to superposition refinement, we can say that f refines e if the guard of f is stronger than the guard of e (*guard strengthening*), and the gluing invariants $J(v,w)$ establish a simulation of f by e (*simulation*). This condition is captured by the following proof obligation rule.

[4] Concrete events may be annotated with abstract events name and witnesses, which we will show later.

$$\begin{array}{l} \quad I(v) \\ \quad J(v,w) \\ \quad H(y,w) \\ \quad \boldsymbol{R}(y,w,w') \\ \vdash \\ \quad \exists x,v' \cdot G(x,v) \ \wedge \ \boldsymbol{Q}(x,v,v') \ \wedge \ J(v',w') \end{array} \tag{7}$$

In order to simplify and split the above proof obligation, Event-B introduces the notion of "witnesses" for the abstract parameters x and the after value of the abstract variables v'. Witnesses are predicates of the form $W_1(x,y,v,w,w')$ (for x), and $W_2(v',y,v,w,w')$ (for v'), which are required to be *feasible*. The corresponding proof obligations are as follows.

$$I(v), J(v,w), H(y,w), \boldsymbol{R}(y,w,w') \ \vdash \ \exists x \cdot W_1(x,y,v,w,w') \tag{WFIS}$$

$$I(v), J(v,w), H(y,w), \boldsymbol{R}(y,w,w') \ \vdash \ \exists v' \cdot W_2(v',y,v,w,w') \tag{WFIS}$$

Typically, the witnesses are declared deterministically, *i.e.*, of the form $x = \ldots$ or $v' = \ldots$. In these cases, the witnesses are trivially feasible, hence the corresponding proof obligations are omitted.

Given the witnesses, the refinement proof obligation (7) is replaced by three different proof obligations as follows.

$$I(v), J(v,w), H(y,w), W_1(x,y,v,w,w') \ \vdash \ G(x,v) \tag{GRD}$$

$$I(v), J(v,w), H(y,w), \boldsymbol{R}(y,w,w'), W_1(x,y,v,w,w'), W_2(v',y,v,w,w') \ \vdash \ \boldsymbol{Q}(x,v,v') \tag{SIM}$$

$$I(v), J(v,w), H(y,w), \boldsymbol{R}(y,w,w'), W_1(x,y,v,w,w'), W_2(v',y,v,w,w') \ \vdash \ J(v',w') \tag{INV}$$

The concrete event f can be denoted with the abstract event e (using keyword **refines**) and the witnesses (using keyword **with**), as follows.

```
f
  refines  e
  any  y  where
    H(y,w)
  with
    x :  W1(x,y,v,w,w')
    v' :  W2(v',y,v,w,w')
  then
    R(y,w)
  end
```

The action of the concrete event is required to be feasible. The corresponding proof obligation FIS is similar to the one presented for the abstract machine, with the exception that both abstract and gluing invariants can be assumed.

For newly introduced events, similar to superposition refinement, they must be proved to refine the implicit abstract event SKIP, which is unguarded and does nothing, *i.e.*, does not modify abstract variables v. In this case, GRD is trivial, since the abstract guard is $\top$. For SIM and INV, we omit references to W_1 (since there are no parameters for the abstract SKIP event). Moreover, the witness W_2 for v' is trivial: $v' = v$.

As mentioned earlier, in general, Event-B refinement is a mixture of both superposition and data refinement. Often, some (not all) abstract variables are retained in the refinement, while the other abstract variables are replaced by new concrete variables. Similarly, some abstract parameters can be present in the concrete event, where other parameters are replaced by some new concrete parameters. In general, we only need to give witnesses to disappearing variables and parameters.

The one-to-one correspondence between the abstract and concrete events can be relaxed. When an abstract event e is refined by more than one concrete event f, we say that the abstract event e is *split* and prove that each concrete f is a valid refinement of the abstract event. Conversely, several abstract events e can be refined by one concrete f. We say that these abstract events are *merged* together. A requirement for merging events is that the abstract events must have identical actions. When merging events, we need to prove that the guard of the concrete event is stronger than the disjunction of the guards of the abstract events.

The concrete machine N can be proved to be *relatively deadlock-free* with respect to the abstract machine M. It means that if M can continue in some state, so can N. Assume that M contains a set of n events e_i ($i \in 1 \ldots n$) of the following form.

$$\mathsf{e_i} \mathrel{\widehat{=}} \textbf{any } x_i \textbf{ where } G_i(x_i, v) \textbf{ then } Q_i(x_i, v) \textbf{ end}$$

Assume that N contains a set of m events f_j ($j \in 1 \ldots m$) of the following form.

$$\mathsf{f_j} \mathrel{\widehat{=}} \textbf{any } y_i \textbf{ where } H_i(y_i, w) \textbf{ then } R_i(y_i, w) \textbf{ end}$$

The proof obligation rule for relative deadlock-freeness[5] is as follows.

$$\begin{array}{l} \quad I(v) \\ \quad J(v,w) \\ \quad (\exists x_1 \cdot G_1(x_1,v)) \lor \ldots \lor (\exists x_n \cdot G_n(x_n,v)) \\ \vdash \\ \quad (\exists y_1 \cdot H_1(y_1,w)) \lor \ldots \lor (\exists y_m \cdot H_m(y_m,w)) \end{array} \qquad \text{(REL\_DLF)}$$

7.2.1 Machine m2

We perform a data refinement by replacing abstract variables *crs* and *prtcpts* by a new concrete variable *atnds*. This machine does not explicitly model any require-

[5] Typically, this is encoded as a theorem in N after declaration of all invariants.

ments from Section 3: it implicitly inherits requirements from previous abstract machines. As stated in invariant **inv2_1**, *atnds* is a *partial function* from *CRS* to some set of participants (*i.e.*, member of $\mathbb{P}(PRTCPT)$). Invariants **inv2_2** and **inv2_3** act as gluing invariants, linking abstract variables *crs* and *prtcpts* with concrete variable *atnds*. Invariant **inv2_2** specifies that *crs* is the domain *atnds*. Invariant **inv2_3** states that for every opened courses c, the set of participants attending that course represented abstractly as $prtcpts[\{c\}]$ is the same as $atnds(c)$.

variables: *atnds*

invariants:
inv2_1 : $atnds \in CRS \pfun \mathbb{P}(PRTCPT)$
inv2_2 : $crs = \mathrm{dom}(atnds)$
inv2_3 : $\forall c \cdot c \in crs \Rightarrow prtcpts[\{c\}] = atnds(c)$
thm2_1 : $\mathrm{finite}(atnds)$

init
begin
$atnds := \varnothing$
end

We illustrate our data refinement by the following example. Assume that the available courses *CRS* are $\{c_1, c_2, c_3\}$, with c_1 and c_2 being opened, *i.e.*, $crs = \{c_1, c_2\}$. Assume that c_1 has no participants, and p_1 and p_2 are attending c_2. Abstract variable *prtcpts* hence contains two mappings as follows $\{c_2 \mapsto p_1, c_2 \mapsto p_2\}$. The same information can be represented by the concrete variable *atnds* as follows $\{c_1 \mapsto \varnothing, c_2 \mapsto \{p_1, p_2\}\}$.

We refine the events using data refinement as follows. Event OpenCourses is refined by OpenCourse where one course (instead of possibly several courses) is opened at a time. The course that is opening is represented by the concrete parameter c.

OpenCourses
when
grd0_1 : $\mathrm{card}(crs) \neq m$
thm0_1 : $crs \neq CRS$
then
act0_1 : $crs :| crs \subset crs' \wedge \mathrm{card}(crs') \leq m$
end

OpenCourse
refines OpenCourses
any c **where**
grd2_1 : $c \notin \mathrm{dom}(atnds)$
grd2_2 : $\mathrm{card}(atnds) \neq m$
with
$crs' = crs \cup \{c\}$
then
act2_1 : $atnds(c) := \varnothing$
end

The concrete guards ensure that c is a closed course and the number of opened course ($\mathrm{card}(atnds)$) has not reached the limit m. The action of OpenCourse sets the initial participants for the newly opened course c to be the empty set. In order to prove the refinement relationship between OpenCourse and OpenCourses, we need to give the witness for the after value of the disappearing variable crs'. In this case, it is specified as $crs' = crs \cup \{c\}$, *i.e.*, adding the newly opened course c to the original set of opened courses *crs*.

Abstract event CloseCourses is refined by concrete event CloseCourse, where one course c (instead of possibly several courses cs) is closed at a time. The guard and action of concrete event CloseCourse are as expected.

CloseCourses
status *anticipated*
any cs **where**
grd0_1 : $cs \subseteq crs$
grd0_2 : $cs \neq \varnothing$
then
act0_1 : $crs := crs \setminus cs$
act2 : $prtcpts := cs \lhd prtcpts$
end

CloseCourse
refines CloseCourses
status *convergent*
any c **where**
grd2_1 : $c \in \mathrm{dom}(atnds)$
with
$cs = \{c\}$
then
act2_1 : $atnds := \{c\} \lhd atnds$
end

We need to give the witness for the disappearing abstract parameter cs. It is specified straightforwardly as $cs = \{c\}$. Notice also that we change the convergence status of CloseCourse from *anticipated* to *convergent*. We use the following variant to prove that CloseCourse is convergent.

variant: $\mathrm{card}(atnds)$

The variant represents the number of mappings in *atnds*, and since it is a partial function, it is also the same as the number of elements in its domain, *i.e.*, $\mathrm{card}(atnds) = \mathrm{card}(\mathrm{dom}(atnds))$. As a result, the variant represent the number of opened courses.

Event Register is refined as follows[6], such that references to crs and $prtcpts$ in guard and action are replaced by references to $atnds$.

(abs_)Register
any p, c **where**
grd1_1 : $p \in PRTCPT$
grd1_2 : $c \in crs$
grd1_3 : $p \neq instrs(c)$
grd1_4 : $c \mapsto p \notin prtcpts$
then
act1_1 : $prtcpts := prtcpts \cup \{c \mapsto p\}$
end

(cnc_)Register
refines Register
any p, c **where**
grd2_1 : $p \in PRTCPT$
grd2_2 : $c \in \mathrm{dom}(attendees)$
grd2_3 : $p \neq instrs(c)$
grd2_4 : $p \notin atnds(c)$
then
act2_1 : $atnds(c) := atnds(c) \cup \{p\}$
end

We now show some proof obligations for proving the refinement of m1 by m2.

OpenCourse/**act0_1**/SIM

This proof obligation corresponds to rule SIM, to ensure that the action **act0_1** of abstract event OpenCourses can simulate the action of concrete event OpenCourse. Notice the use of the witness for crs' as a hypothesis in the obligation.

[6] We use prefixes (abs_) and (cnc_) to denote the abstract and concrete version of the event, accordingly.

$$\begin{array}{l} \ldots \\ atnds \in CRS \pfun \mathbb{P}(PRTCPT) \\ crs = \mathrm{dom}(attendees) \\ c \notin \mathrm{dom}(attendees) \\ \mathrm{card}(attendees) \neq m \\ crs' = crs \cup \{c\} \\ \vdash \\ crs \subset crs' \land \mathrm{card}(crs') \leq m \end{array} \quad \text{OpenCourse/}\mathbf{act0\_1}\text{/SIM}$$

CloseCourse/**grd0_1**/GRD

This proof obligation corresponds to rule GRD, to ensure that the guard of concrete event CloseCourse is stronger than the abstract guard **grd0_1** of abstract event CloseCourses. Note the use of the witness for *cs* as a hypothesis in the obligation.

$$\begin{array}{l} \ldots \\ crs = \mathrm{dom}(atnds) \\ c \in \mathrm{dom}(atnds) \\ cs = \{c\} \\ \vdash \\ cs \subseteq crs \end{array} \quad \text{CloseCourse/}\mathbf{grd0\_1}\text{/GRD}$$

CloseCourse/NAT

This proof obligation corresponds to rule NAT on page 10, it ensures that the variant is a natural number when CloseCourse is enabled.

$$\begin{array}{l} \ldots \\ \mathrm{finite}(atnds) \\ \vdash \\ \mathrm{card}(atnds) \in \mathbb{N} \end{array} \quad \text{CloseCourse/NAT}$$

CloseCourse/VAR

This proof obligation corresponds to rule VAR on page 10, it ensures that the variant is strictly decreased by CloseCourse. The obligation is trivial since the variant represents the number of opened courses and CloseCourse closes one of them.

$$\begin{array}{l} \ldots \\ atnds \in CRS \pfun PRTCPT \\ c \in \mathrm{dom}(atnds) \\ \vdash \\ \mathrm{card}(\{c\} \domsub atnds) < \mathrm{card}(atnds) \end{array} \quad \text{CloseCourse/VAR}$$

8 Summary of the Development

The summary of the hierarchy of the development is illustrated in Figure 1.

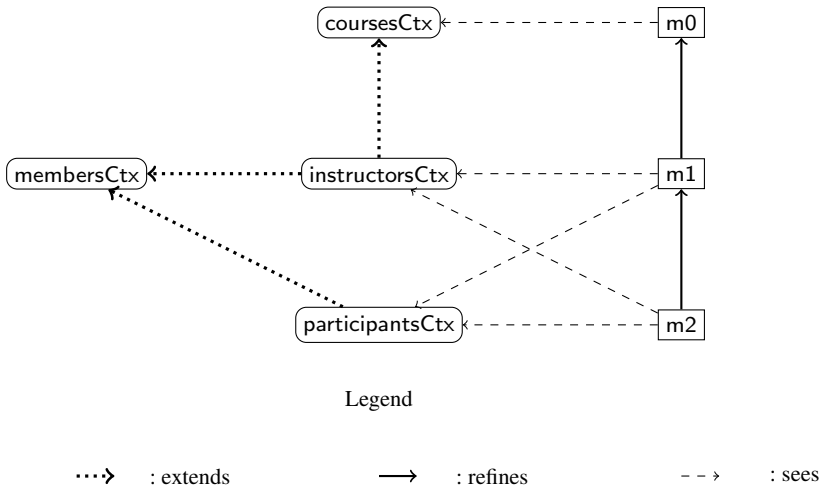


Fig. 1 Development hierarchy

Table 2 summarises how assumptions and requirements are taken into account in our formal development. Note that the last refinement m2 does not explicitly take into account any requirements. Indeed, the requirements are implicitly *inherited* from the abstract machines through the refinement relationship.

Requirement	Models	Requirement	Models
ASM 1	instructorsCtx	REQ 6	m0
ASM 2	participantsCtx	REQ 7	m0
ASM 3	coursesCtx	REQ 8	m0
ASM 4	instructorsCtx	REQ 9	m1
REQ 5	m0	REQ 10	m1

Table 2 Requirements Tracing

The development is formalised and proved using supporting Rodin platform[7] for Event-B [3] and is available on-line[8]. The summary of the proof statistics for the development is showed in Table 3. Around 50% of the proof obligations appear in m2, where we perform data refinement. Typically, data refinement involves radical

[7] At the time of writing, we use Rodin version 2.4.0.

[8] http://deploy-eprints.ecs.soton.ac.uk/371/

changes to developments, since by replacing abstract variables with concrete variables, it is also necessary to adapt the events accordingly.

Constructs	Proof obligations	Automatic (%)	Manual (%)
coursesCtx	2	2 (100%)	0 (0%)
membersCtx	0	0 (N/A)	0 (N/A)
instructorsCtx	0	0 (N/A)	0 (N/A)
participantsCtx	1	1 (100%)	0 (0%)
m0	11	8 (73%)	3 (27%)
m1	14	13 (93%)	1 (7%)
m2	29	26 (90%)	3 (10%)
Total	57	50 (88%)	7 (12%)

Table 3 Proof statistics

References

1. J-R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.
2. J-R. Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, May 2010.
3. 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. *Internation Journal on Software Tools for Technology Transfer (STTT)*, April 2010.
4. R-J. Back. Refinement Calculus II: Parallel and Reactive Programs. In J. W. deBakker, W. P. deRoever, and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems*, volume 430 of *LNCS*, pages 67–93, Mook, The Netherlands, May 1989. Springer-Verlag.
5. K. Chandy and J. Misra. *Parallel Program Design: a Foundation*. Addison-Wesley, 1989.
6. L. Lamport. The temporal logic of actions. *Transactions on Programming Languages and Systems (TOPLAS)*, 16(3):872–923, May 1994.
7. F. Mehta. *Proofs for the Working Engineer*. PhD thesis, ETH-Zurich, 2008.
8. M. Schmalz. The logic of Event-B. Technical Report 698, Institute of Information Security, October 2010. http://www.inf.ethz.ch/research/disstechreps/techreports/show?serial=698&lang=en.