Augmenting formal development with use case reasoning

Alexei Iliasov

Newcastle University, UK

Abstract. State-based methods for correct-by-construction software development rely on a combination of safety constraints and refinement obligations to demonstrate design correctness. One prominent challenge, especially in an industrial setting, is ensuring that a design is adequate: requirements compliant and fit for purpose. The paper presents a technique for augmenting state-based, refinement-driven formal developments with reasoning about use case scenarios; in particular, it discusses a way for the derivation of formal verification conditions from a high-level, diagrammatic language of use cases, and the methodological role of use cases in a formal modelling process.

1 Introduction

Use cases are a popular technique for the validation of software systems and constitute an important part of requirements engineering process. It is an essential part of the description of functional requirements of a system. There exists a vast number of notations and methods supporting the integration of use cases in a development process (see [8] for a structured survey of use case notations). With few exceptions, the overall aim is the derivation of test inputs for the testing of the final product. We propose to exploit use cases in the course of a stepwise formal development process for the engineering of correct-by-construction systems. We build upon the previous work to add the notion of use case refinement. The results open a way for some interesting methodological and tooling advances such as use case driven model refinement and the automation of use case construction via the mechanisation of use refinement rules.

In this work, we more carefully study the meaning of writing a use case for a formal specification. Previously, we have stated that the semantics of a model with a use case is simply the collection of all the theorems that must be demonstrated for the model correctness and the consistency of the use case and the model. This fits well into the picture of a proof-based semantics. However, from the discussions with the industrial partners of the IST Deploy Project [11], we have realised that the answer is not entirely satisfactory. It does not relate to the concepts familar from requirements engineering stage and does not provide a ground to judge about the appropriateness of the technique in an application to a given problem. We thus consider another form of semantics to supplement the existing proof semantics.

```
MACHINE M
  SEES Context
  VARIABLES v
  INVARIANT I(c, s, v)
  INITIALISATION R(c, s, v')
  EVENTS
    E1 = any vl where g(c, s, vl, v) then S(c, s, vl, v, v') end
    ...
END
```

Fig. 1. Event-B machine structure.

2 Background

The section briefly introduces modelling method Event-B and its extension for expressing use cases.

2.1 Event-B

The basis of our discussion is a formalism called Event-B[2]. It belongs to a family of state-based modelling languages that represent a design as a combination of state (a vector of variables) and state transformations (computations updating variables).

An Event-B development starts with the creation of a very abstract specification. A cornerstone of the Event-B method is the stepwise development that facilitates a gradual design of a system implementation through a number of correctness-preserving *refinement* steps. The general form of an Event-B model (or *machine*) is shown in Figure 1. Such a model encapsulates a local state (program variables) and provides operations on the state. The actions (called *events*) are characterised by a list of local variables (parameters) vl, a state predicate g called *event guard*, and a next-state relation S called *substitution* or event *action*.

Event guard g defines the condition when an event is *enabled*. Relation S is given as a generalised substitution statement [1] and is either deterministic ($x := 2$) or non-deterministic update of model variables. The latter kind comes in two notations: selection of a value from a set, written as $x :\in \{2, 3\}$; and a relational constraint on the next state v', e.g., $x :| x' \in \{2, 3\}$.

The **INVARIANT** clause contains the properties of the system, expressed as state predicates, that must be preserved during system execution. These define the *safe states* of a system. In order for a model to be consistent, invariant preservation is formally demonstrated. Data types, constants and relevant axioms are defined in a separate component called *context*.

Model correctness is demonstrated by generating and discharging *proof obligations* - theorems in the first order logic. There are proof obligations for model consistency and for a refinement link - the forward simulation relation - between the pair of *abstract* and *concrete* models. More details on Event-B, its semantics,

method and applications may be found in [2] and also on the Event-B community website[5]. A concise discussion of the Event-B proof obligations is given in [7].

2.2 The use case extension of Event-B

The approach to use case reasoning is based on our previous work on a graphical notation for expressing event ordering constraints [10, 9]. The extensions is realised as a plug in to the Event-B modelling tool set - the Rodin Platform [13] - and smoothly integrates into the Event-B modelling process. It provides a modelling environment for working with graph-like diagrams describing event ordering properties. In the simplest case, a node of such graph is an event of the associated Event-B machine; an edge is a statement about the relative properties of the connected nodes/events. There are three main edge kinds: **ena**, **dis** and **fis**. Mathematically, they are defined as follows as relations over Event-B events.

$$\begin{aligned} U &= \{f \mapsto g \mid \varnothing \subset f \subseteq S \times S \land \varnothing \subset g \subseteq S \times S\} \\ \mathbf{ena} &= \{f \mapsto g \mid f \mapsto g \in U \land \operatorname{ran}(f) \subseteq \operatorname{dom}(g)\} \\ \mathbf{dis} &= \{f \mapsto g \mid f \mapsto g \in U \land \operatorname{ran}(f) \cap \operatorname{dom}(g) = \varnothing\} \\ \mathbf{fis} &= \{f \mapsto g \mid f \mapsto g \in U \land \operatorname{ran}(f) \cap \operatorname{dom}(g) \neq \varnothing\} \end{aligned}$$

where $f \subseteq S \times S$ is a relational model of an Event-B event (mathematically, an event is a next-state relation). These definitions are converted into *consistency* proof obligations. For instance, if in a use case graph there appears an edge connecting events b and h one would have to prove the following theorem (see [10] for a justification).

$$\forall v, v', p_b \cdot I(v) \land G_b(p_b, v) \land R_b(p_b, v, v') \Rightarrow \exists p_h \cdot G_h(p_h, v') \tag{1}$$

A use case diagram is only defined in an association with one Event-B model, it does not exist on its own. The use case plug in automatically generates all the relevant proof obligations. A change in a diagram or its Event-B model leads to the re-computation of all affected proof obligations. These proof obligations are dealt with, like all other proof obligation types, by a combination of automated provers and interactive proof. Like in the proofs of model consistency and refinement, the feedback from an undischarged use case proof obligation may often be interpreted as a suggestion of a diagram change such as an additional assumptions or assertion - predicate annotations on graph edges that propagate properties along the graph structure. The example in the next section demonstrates how such annotations enable the proof of a non-trivial property.

The use case tool offers a rich visual notation. The basic element of a diagram is an event, visually depicted as a node (in Figure 2, f and g represent events). Event definition (its parameters, guard and action) is imported from the associated Event-B model. One special case of node is skip event, denoted by a grey node colour (Figure 2, 5). Event relations $\mathbf{ena}, \mathbf{dis}, \mathbf{fis}$ are represented by edges connecting nodes ((Figure 2, 1-3)). Depending on how a diagram is drawn, edges are said to be in *and* or *or* relation (Figure 2, 7-8). New events are derived

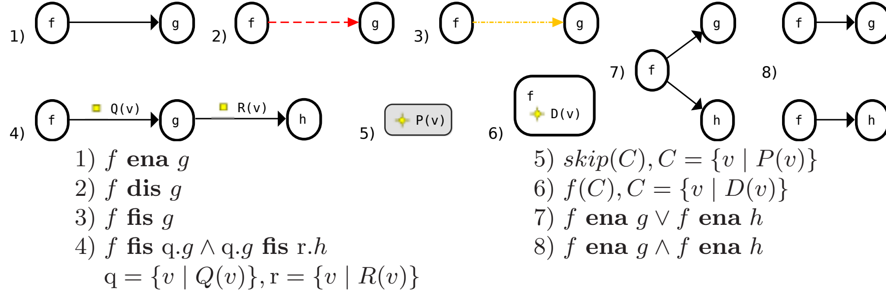


Fig. 2. A summary of the core use case notation and its interpretation.

from model events by strengthening their guards (a case of symmetric assumption and assertion) (Figure 2, 6). Edges may be annotated with constraining predicates inducing assertion and assumption derived events (Figure 2, 4). Not shown on Figure 2 are nodes for the initialisation event start (circle), implicit deadlock event stop (filled circle) and nodes for container elements such as loop (used in the coming example). To avoid visual clutter, the repeating parts of a diagram may be declared separately as diagram *aspects*[10]. Aspects are used in the diagrams from Sections 5.4 and 5.6.

2.3 Small example

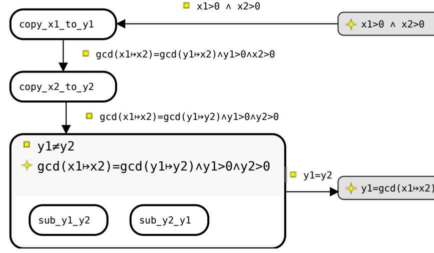
As an illustration of what constitutes a use case we consider a small example concerned with the construction of a function computing the greatest common devisor (GCD) of two numbers. The properties characterise the GCD function $\mathrm{gcd} \in \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$.

$\forall a, b \cdot a, b \in \mathbb{N} \wedge a > b \Rightarrow \mathrm{gcd}(a, b) = \mathrm{gcd}(a - b, b)$
$\forall a, b \cdot a, b \in \mathbb{N} \wedge b > a \Rightarrow \mathrm{gcd}(a, b) = \mathrm{gcd}(a, b - a)$
$\forall a \cdot a \in \mathbb{N} \Rightarrow \mathrm{gcd}(a, a) = a$

The Event-B part of the example is a simple and abstract model. It defines four variables and four operations on these variables.

MACHINE gcd
  VARIABLES $x1, x2, y1, y2$
  INVARIANT $x1 \in \mathbb{N} \wedge x2 \in \mathbb{N} \wedge y1 \in \mathbb{N} \wedge y2 \in \mathbb{N}$
  EVENTS
    copy1 = **begin** $y1 := x1$ **end**
    copy2 = **begin** $y2 := x2$ **end**
    sub1 = **when** $y1 > y2$ **then** $y1 := y1 - y2$ **end**
    sub2 = **when** $y1 < y2$ **then** $y2 := y2 - y1$ **end**
END

With a use case diagram we prove that this Event-B model contains the behaviour that the realises the GCD function. More specifically, the following use case digram makes a provable statement that, starting with some two positive numbers, a certain sequence of event executions results in the computation of the GCD of the numbers.



In the diagram above, rounded rectangles are events, arrows represent **ena** edges. Grey rectangles are assertions or, depending upon the view point, assumptions. The large rounded box is a structured a loop. It contains two events as a loop body (semantically, a choice of the two events) and is annotated with a loop condition and invariant. The only constraint that must be provided by a user is the loop invariant $gcd(x1 \mapsto x2) = gcd(y1 \mapsto y2) \land y1 > 0 \land y2 > 0$. All other edge constraints are then filled in automatically by a modelling assistant. It is certainly not difficult to prove the same property as a refinement link to one step computation of gcd. There is, however, a distinct advantage in using a graphical notation to formulate properties of event orderings, especially if these are interpreted as use case scenarious.

3 Use case semantics

It is possible to define the trace semantics of an Event-B machine. This is useful for us since a use case already resembles a record of event sequences. The traces of Event-B machine M are defined in the following way. Consider set S of finite sequences of event identifiers, $S = \mathbb{P}(\text{seq}(L))$; here L is set of event identifiers for machine M. Take some sequence $s \in S$; using event declarations from machine M one can convert s into a relation $r(s) \subseteq \Omega \times \Omega$ where Ω is the universe of machine states. Ω is the set of all safe states of a machine: $\Omega = \{v \mid I(v)\}$ where $I(v)$ is the machine invariant. Let $\langle\rangle$, $\langle e \rangle$ and $s \frown t$ signify, correspondingly, an empty sequence, a sequence containing sole element e and sequence concatenation; the procedure to obtain $r(s)$ is then the following.

$$\begin{aligned} &r(\langle\rangle) = \text{id}(\Omega) \\ &r(t \frown \langle e \rangle) = [e]_R \circ r(t) \end{aligned} \tag{2}$$

where $[e]_R$ is a relational interpretation of an event $e \in L$ and $\circ$ is the relations composition operator: $(f \circ g)(x) = g(f(x))$. Let G_e, F_e and u_e be, respectively, the guard, the body and parameters of event L. Then $[e]_R \equiv \{v \mapsto v' \mid \exists u_e \cdot (G_e(v, u_e) \land F_e(v, u_e, v'))\}$. As a special case, the relational form of initialisation is $[\text{INIT}]_R \equiv \text{id}(init)$ where $init \subseteq \Omega$ is the set of vectors of initial values for machine variables; also, the relational form of skip (a stuttering step event of a machine) is $[\text{skip}]_R \equiv \text{id}(\Omega)$. Let us examine sequences s from S. Some of them prescribe event orderings that may not be realised because of the restrictions expressed in event guards; that is, relation $r(s)$ is empty. Some

sequences initiate with an event other that initialisation; we shall reject such sequences as we do not know their meaning. What is left is the following set; it defines the traces of machine M:

$$tr(M) = \{s \mid s \in S \wedge t \in S \wedge s = \langle \text{INIT} \rangle \frown t \wedge r(s) \neq \varnothing\} \tag{3}$$

An important property of machine traces is that between any two machine events one could observe, in differing traces, any finite number of implicit skip events. That is, if there is a trace where a is followed by b there must also be a trace where a is followed by skip and then b; another trace where there are two skip's, three skip's and so on. See Chapter 14 in [2] for an explanation why these events are necessary. We record this observation in the following statement.

$$\forall s, t \cdot s \frown t \in tr(M) \Leftrightarrow s \frown \langle \text{skip} \rangle \frown t \in tr(M) \tag{4}$$

The meaning of a use case may now be stated in much clearer terms: *a use case specification defines a set of traces that are guaranteed to be among the traces of a machine*. To justify this claim we first define the traces of a use case and then show that each such trace is to be found in the traces of a machine.

A use case is understood to be a graph $U = (V, E)$ where a node is a tuple $(h, C) \in V$ of an event identifier $h \in L$ and a constraining predicate $C = C(v, u_L)$ [1]. For some node $q = (h, C)$, its relational interpretation is $[q]_U \equiv \{v \mapsto v' \mid \exists u_h \cdot G_h(v, u_h) \wedge C(v, u_h) \wedge F_h(v, u_h, v')\}$. For any such node there is a corresponding event:

$$\forall q \cdot q \in V \Rightarrow (\exists e \cdot e \in L \wedge [q]_U \subseteq [e]_R) \tag{5}$$

It is not difficult to see why it is the case. As a witness for the bound variable e we take z such that $\{z\} = \text{prj}_1[\{q\}]$[2]. It trivially holds that $\text{prj}_1[V] \subseteq L$ and thus $z \in L$. Supposing $q = (h, C)$ it then follows that $z = h$ (since, by the definition of projection, $\{h\} = \text{prj}_1[\{(h, C)\}]$). Finally, condition $[q]_U \subseteq [e]_R$ holds as it is evident from the definitions of $[q]_U$ and $[e]_R$.

The link between use case nodes and machine traces may be lifted to event sequence. Consider set $\mathcal{U}$ of all the paths of graph U. From Condition 5 it follows that $\forall u \cdot u \in \mathcal{U} \Rightarrow \text{prj}_1[\text{ran}(u)] \subseteq L$, where $\text{ran}(u)$ is a set of values, nodes from V, contained in sequence u. Let P be a function mapping a sequences of graph nodes into a sequence of event identifiers; P does this by simply removing the constraining predicate C from each element of the first sequence. From Condition 5 and the statement above it then follows that $\forall u \in \mathcal{U} \Rightarrow P(u) \in S$.

Let us now consider what are the edges E of a use case graph. Set E is partitioned into three subsets, one for each kind of event relation: $E = E_e \oplus E_d \oplus E_f$. In the discussion we focus exclusively on E_e edges. Other edges do not contribute yet to the trace semantics. Let us consider subgraph $U^+ = (V, E_e)$.

[1] On a use case diagram, predicate C appears in an event node under event name with a star-like icon.

[2] prj_1 is the first projection of a cartesian product

We claim that each P-projected path u from the set of paths $\mathcal{U}^+$ of the sub-graph is also found among the traces of machine M.

$$\forall u \in \mathcal{U}^+ \Rightarrow P(u) \in tr(M) \tag{6}$$

For $P(u)$ to be an element of $tr(M)$ it must satisfy the following three criteria (see Definition 3): $P(u)$ must be in S, the first element must the initialisation event and relational interpretation of $P(u)$, $r(P(u))$ must be a non-empty relation. The first condition has been discussed above. The second one we take as an assumption, that is, a use case graph containing paths that do not start with initialisation event is considered to be ill-formed. Since, from the Event-B semantics, set *init* is known to be non-empty, the are only two reason why $r(P(u))$ could be empty: either there is an event in L that is mapped into an empty relation or there is an instance of relational composition of two non-empty relations yielding an empty relation. The former case, an empty relation, is impossible due to the requirement of event feasibility that, for every machine event $e \in L$, guarantees that there is a non-empty set of next states for this event: $I(v) \wedge G_e(v, u_e) \Rightarrow \exists v' \cdot F_e(v, u_e, v')$. Juxtaposition of this property with the definition of $[e]_R$ implies that for every event $e \in L$ it holds that $[e]_R \neq \varnothing$.

The latter case is also impossible due to the properties of a use case diagram. Let us consider two events, a and b, such that a **ena** b, in other words, there is an edge in graph U^+ connecting nodes $(a, _)$ and $(b, _)$. As a reminder, **ena** $= \{f \mapsto g \mid f \subseteq \Omega \times \Omega \wedge g \subseteq \Omega \times \Omega \wedge \mathrm{ran}(f) \subseteq \mathrm{dom}(g)\}$. Since $\mathrm{ran}([a]_R) \subseteq \mathrm{dom}([b]_R)$, relation $[b]_R \circ [a]_R$ is defined for all the values for which $[a]_R$ is defined: $\mathrm{dom}([b]_R \circ [a]_R) = \mathrm{dom}([a]_R)$. This generalises to any number of events: $\mathrm{dom}([e_n]_R \circ \cdots \circ [e_1]_R) = \mathrm{dom}([e_1]_R)$ where e_1 **ena** $e_2 \wedge \cdots \wedge e_{n-1}$ **ena** e_n. Set $\mathrm{dom}([e_1]_R)$ is not empty since, as discussed above, feasible events cannot yield empty relations. Predicate e_1 **ena** $e_2 \wedge \cdots \wedge e_{n_1}$ **ena** e_n is a characterisation of some path $u \in \mathcal{U}^+$ such that $P(u) = \langle e_1, \ldots, e_n \rangle$. Since $\mathrm{dom}(r(u))$ is not empty, it follows that $r(u)$ is not empty for an arbitrary path $u \in \mathcal{U}^+$ and the third criteria of Condition 6 is satisfied.

To summarise, by traces $tr(U)$ of a use case U we understand the set of all sequences $\{P(u) \mid u \in \mathcal{U}^+\}$ where $\mathcal{U}^+$ is set of paths (all starting with event INIT) of a subgraph U^+ constrained to edges **ena**. Condition 6 can be stated in the following, equivalent form.

$$tr(U) \subseteq tr(M) \tag{7}$$

Hence, a use case defines a set of traces that are also the traces of a machine. The crucial property that we have relied upon is that for some two events connected in a use case by an edge of **ena** kind it holds that a **ena** b. The property is translated into a condition on the actions and guards of the two events as given in the Definition 1. By discharging all such conditions for a use case scenario one establishes Property 7.

For practical reasons, it is necessary to deal with complex nodes that do not directly correspond to machine events. One important kind is the class of container elements. These appear due to an hierarchical organisation of large

use cases and are also required to describe loops with a loop invariant. Inside a container element one finds event nodes and, possibly, other containers. To obtain traces of a use case with container elements, such use case must be first flattened. The flattening proceeds from inside so that the procedure is always applied to a container that does not contain other containers.

4 Co-refinement of machines and use cases

With a trace semantics one defines refinement by stating that the traces of a refined model are contained in the traces of the abstract model. For Event-B this is stated with the following equivalence.

$$M \sqsubseteq N \Leftrightarrow \mathfrak{ff}[tr(N)] \subseteq tr(M) \tag{8}$$

where $\mathfrak{ff}(s)$ a mapping function such that $\forall i \cdot i \in \mathrm{dom}(s) \Rightarrow \mathfrak{ff}(s)(i) = \mathfrak{f}(s(i))$. Map $\mathfrak{f}$ translates an event identifier from N into some event identifiers in M. The map may be partitioned into two parts $\mathfrak{f} = \mathfrak{f}_n \cup \mathfrak{f}_r$ where $\mathfrak{f}_n$ maps all the new events of N into the skip event of M and $\mathfrak{f}_r$ maps refined events of N into their abstract counterparts in M. $\mathfrak{f}_r$ is defined explicitly in a model as a part of refined event declarations.

The Event-B refinement is known to satisfy Definition 8[7]. In fact, Event-B refinement is a stronger relation and takes into the account the notions of convergence and enabledness. For this reason, it would be ill-advised to consider trace refinement as a sole criterion of machine refinement.

Conditions 7 and 8 provide a basis for the definition of use case refinement. Assume there is use case U associated with machine M, $tr(U) \subseteq tr(M)$, and another use case W, $tr(W) \subseteq tr(N)$ associated with machine N refining M, $\mathfrak{ff}[tr(N)] \subseteq tr(M)$. Obviously, traces of W are found in traces of M but the relationship between U and W is not certain. Let us first see what kind of relationship we are looking for. Intuitively, N does the some thing as M but in a better way (note this does not follow from Condition 8 which allows N to do less of the same thing). The same principle transfers to use cases. What is expressed in an abstract use case U must be realised in a satisfactory way by concrete use case W. Consider the following illustration. If there is some phenomenon x in U is must be also present, perhaps in differing form, in W. If x is an event the requirement translates to not forgetting to include in W refined versions of abstract events used in U; x may be a complex phenomenon describing a situation where, for instance, event a is followed by event b. In W such a phenomenon may be portrayed literally, by including x as it is defined in U. It could also appear transformed where a and b are replaced by refined version a' and b'. Further, due to Condition 4, if x exists in U there also exists x_1 in U such that between a and b there is one skip event. This means that x' may be formed as a' followed by some new event e and then by b'. Since new event e is mapped into skip it would match phenomenon x_1, that is, $x = \mathfrak{ff}(x_1)$. To summarise, the notion of use case refinement is expressed as the equality of abstract and concrete use case under the mapping $\mathfrak{ff}$.

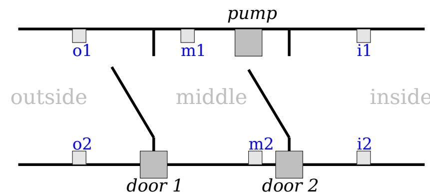
$$U \sqsubseteq W \Leftrightarrow U = \mathfrak{ff}[W] \wedge tr(U) \subseteq tr(M) \wedge tr(W) \subseteq tr(N) \wedge M \sqsubseteq N \qquad (9)$$

The notion of use case refinement embeds the notions of use case/machine consistency and machine refinement: one must not discuss use case refinement in detachment from machines they characterise.

5 Case study

With the case study we illustrate the role of a use case in a formal development. We show how one can formulate a simple scenario to characterise an abstract design and then evolve it hand in hand with the detalisation of a functional model. Thus, at each development stage not only we attend to the issue of correctness but also give an argument for the adequacy of the constructed design.

The problem we study is the construction of control logic for a sluice mechanism. The sluice is placed between areas of greatly differing pressures making it unsafe to operate a simple door. The major components of the sluice are two doors that are tightly sealed when closed, a middle chamber where the pressure may be controlled by the means of a pump, and six buttons placed on the walls as depicted in the diagram below.



In the diagram, $i1, i2, m1, m2, o1, o2$ are buttons. They all have the same functionality and no fixed purpose; however, it is permitted to produce a note for a sluice use that would describe the meaning of the buttons and operational steps required to use the sluice. The following is a brief summary of the system requirements. Consult [4] for the complete list.

SYS the purpose of the system is to allow a user to safely travel between inside or outside areas
ENV4 six buttons, two per each area, are used to interact with a user;
ENV5 at any moment, there is at most one human operator in the system
SAF1 a door may be open only if the pressures in the locations it connects are equalised;
SAF2 at most one door is open at any moment;
SAF3 pressure may only be changed when the doors are closed;

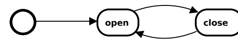
The key use case of the requirements document may be summarised as follows: *when a user is in the inside (outside) area, it is possible for the user to* safely *travel to the outside (inside) area by interacting with the system only by pressing the buttons.*

5.1 Abstract model

The development starts with a high level abstraction that depicts the whole system as a single, conceptual 'door'. The model, although trivial, makes two important statements: the system does not terminate; a closed 'door' eventually opens and vice versa. The following is the complete Event-B model of the initial abstraction.

MACHINE m0
VARIABLES $door$
INVARIANT $door \in \text{DOOR}$
INITIALISATION $door := \text{CLOSED}$
EVENTS
open = **when** $door = \text{CLOSED}$ **then** $door := \text{OPEN}$ **end**
close = **when** $door = \text{OPEN}$ **then** $door := \text{CLOSED}$ **end**
END

A use case of the model resembles a simple automata where a system, once initialised, forever cycles between the only two states.



All the subsequent use cases merely provide a more detailed description of the same phenomenon.

5.2 First refinement

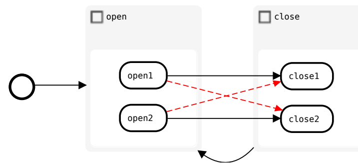
The first refinement explains the meaning of the abstract 'door' by relating its states to the states of two physical doors of the system. The following predicates explain the conenction between the doors.

$door1 = \text{CLOSED} \land door2 = \text{CLOSED} \Rightarrow door = \text{CLOSED}$
$door1 = \text{OPEN} \lor door2 = \text{OPEN} \Rightarrow door = \text{OPEN}$
$\neg(door1 = \text{OPEN} \land door2 = \text{OPEN})$

The last condition expresses safety requirement **SAF2**. The abstract 'door' is now removed from the model and the behaviour is expressed in the terms of the two new doors. The following events describe how the first door opens and closes. The events for the second door are symmetric.

open1 = **when** $door1 = \text{CLOSED} \land door2 = \text{CLOSED}$ **then** $door1 := \text{OPEN}$ **end**
close1 = **when** $door1 = \text{OPEN}$ **then** $door1 := \text{CLOSED}$ **end**

The new use case is constructed mostly automatically by asking the tool to refine the abstract use case.



The light grey boxes depict the split refinement of abstract events open and close, as prescribed by the ff map computed from the definition of concrete

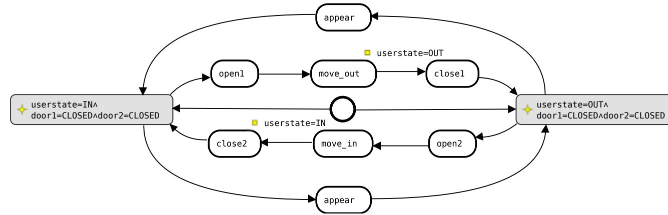
Event-B model. The tool would not know what to do about edges and thus, to be on the safe side, it simply puts new events into containers. The edges piercing the container boundary were manually as were the disabling edges. The latter highlight the fact that these are the impossible event connections.

5.3 Second refinement

In this step, a simple concept of user is introduced. The model expresses that for a user to arrive at a location he must first travel through an open door connecting the location with the middle area. We are not concerned yet with how a user arrives at the middle area. A new variable, $userstate \in \text{USER}$ appears and is manipulated by the following events.

$$\begin{aligned} \mathsf{move_in} &= \textbf{when } userstate \neq \text{IN} \wedge door2 = \text{OPEN} \textbf{ then } userstate := \text{IN} \textbf{ end} \\ \mathsf{move_out} &= \textbf{when } userstate \neq \text{OUT} \wedge door1 = \text{OPEN} \textbf{ then } userstate := \text{OUT} \textbf{ end} \\ \mathsf{appear} &= \textbf{begin } userstate :\in \text{USER} \textbf{ end} \end{aligned}$$

The appear event models the fact that a user may leave the system and another user may appear at either location. In the following use case we identify two important states (the system is fully ready and the user is inside (outside)) around which much of the system evolution revolves.



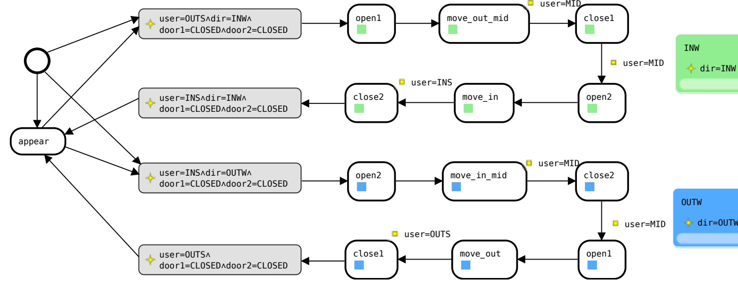
The diagram is an artificial composition of three simpler use cases: a use case for the intialisation, another for the appear event and the main loop use case. Also note the use assertion to propagate the information about user position through events close1 and close2.

5.4 Third refinement

A more detailed user model is introduced by keeping track of the direction in which a user is supposed to travel. This allows us to relate the state when a user is in the middle area to the abstract notions of a user being inside or outside. Abstract variable $userstate \in \{\text{IN}, \text{OUT}\}$ is replaced with concrete variables $user \in \{\text{INS}, \text{OUTS}, \text{MID}\}$ and $dir \in \{\text{INW}, \text{OUTW}\}$.

$$\begin{aligned} & user = \text{INS} \Rightarrow userstate = \text{IN} \\ & user = \text{OUTS} \Rightarrow userstate = \text{OUT} \\ & user = \text{MID} \wedge dir = \text{INW} \Rightarrow userstate = \text{OUT} \\ & user = \text{MID} \wedge dir = \text{OUTW} \Rightarrow userstate = \text{IN} \end{aligned}$$

At this point the use case naturally splits into two symmetric cases, one for each direction of travel. The loop of the previous use case is unfolded for the two complete travel scenarious.



The two boxes to right of the diagram are aspects. They define conditions shared by a large proportion of the diagram. In this case, the conditions state the direction of use travel. The aspect weaving is indicated by small colour-coded square in an event node.

5.5 Fourth, fifth and sixth refinements

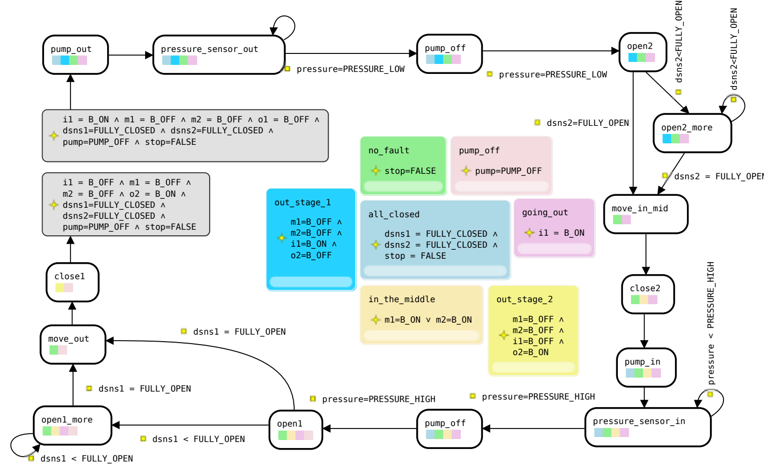
The fourth refinement introduces a more detailed door model to account for doors that may get stuck when partially open: $dsns1 \in \text{FULLY_CLOSED} \mathrel{..} \text{FULLY_OPEN}$. It replaced the abstract notion of door: $dsns1 = \text{FULLY_CLOSED} \Leftrightarrow door1 = \text{CLOSED}$. In the fifth refinement there appears a model of the pump device to control the middle area pressure. The pump is controlled by flag $pump \in PUMP$ and the pressure sensor $pressure \in PRESSURE_LOW \mathrel{..} PRESSURE_HIGH$ reports the current pressure. We are now able to express the remaining safety conditions, **SAF1** and **SAF3**.

$$dsns1 \neq \text{FULLY_CLOSED} \lor dsns2 \neq \text{FULLY_CLOSED} \Rightarrow pump = \text{PUMP_OFF}$$
$$\neg stop = \text{TRUE} \land dsns1 \neq \text{FULLY_CLOSED} \Rightarrow pressure = \text{PRESSURE_HIGH}$$

The sixth refinement gets rid of variables *user* and *dir* and, instead, detects user position and intention using the six available buttons. It is a fairly intricate data refinement step that may be accomplished in more than one way and still satisfy the requirements.

5.6 Seventh refinement

The concluding step adds a use case diagram as a final check of the model is adequacy. It is also a direct encoding of the informal use case mentioned earlier. The following diagram is a fragment refining the part of the previous use case starting with event open2 and ending with event close1. There are ways to cut such diagrams into smaller, self-contained parts[10].



Proof Statistics The proof statistics in terms of generated proof obligations is shown below. The numbers represent the total number of proof obligations, the number of automatically and manually proved ones, the number use case derived obligations, the percentage of manual effort and the proportion use case proof obligations.

Step	Total	Manual	Use case	Manual, %	Use case, %
m1	6	0	**6**	0%	100%
m2	26	0	**7**	0%	27%
m3	15	0	**12**	0%	80%
m4	48	7	**18**	14%	37%
m5	48	4	0	8%	0%
m6	48	4	0	8%	0%
m7	71	1	0	1%	0%
m8	15	3	**15**	20%	100%
Overall	277	19	**58**	7%	21%

Among the 19 manual proofs, 3 are due to use case proof obligations. The Event-B and use case models together with formal proofs may be found in [4]. Identical values for $m5$ and $m6$ are correct.

6 Conclusion

In this work we have presented an approach that, as we believe, makes Event-B more industry friendly by offering a formal counterpart of a well established informal concept. The approach should make it easier to integrate Event-B into an existing development process where use cases are already a part of the requirements engineering process. A hand-in-hand development of use cases and functional specification provides a degree of assurance that a development evolves in the right direction.

The most closely related work is a study of liveness-style theorems for the Classical B [3]. The work introduces a number of notation extensions to construct proofs about 'dynamic' properties of models - properties that span over several

event executions. Like in a use case diagram, the formulation of reachability property requires spelling out a path that would lead to its satisfaction. One advantage of our approach is in the use of graphs to construct complex theorems from simple ones and the propagation of properties along the graph structure. The latter results in interactive modelling/proof sessions where proof feedback leads to small, incremental changes in the diagram.

There are a number of approaches [14, 12, 6] on combining process algebraic specification with event-based formalisms such as Event-B and Action Systems. The fundamental difference is that our technique does not introduce behavioural constraints and is simply a high-level notation for writing certain kind of theorems. It would be interesting to explore how explicit control flow information present in a process algebraic model part may affect the applicability and the practice of the proposed approach.

References

1. J.-R. Abrial. *The B-Book*. Cambridge University Press, 1996.
2. J.-R. Abrial. *Modelling in Event-B*. Cambridge University Press, 2010.
3. J.-R. Abrial and L. Mussat. Introducing Dynamic Constraints in B. In *Proceedings of the Second International B Conference on Recent Advances in the Development and Use of the B Method*, pages 83–128, London, UK, 1998. Springer-Verlag.
4. The door controller model. Event B/Use case specification. 2011. Available at http://iliasov.org/usecase/doorctr.zip.
5. Event-B. Community web site. 2011. http://event-b.org/.
6. Clemens Fischer and Heike Wehrheim. Model-Checking CSP-OZ Specifications with FDR. In Keijiro Araki, Andy Galloway, and Kenji Taguchi, editors, *IFM '99: Proceedings of the 1st International Conference on Integrated Formal Methods*, pages 315–334, London, UK, 1999. Springer-Verlag.
7. Stefan Hallerstede. On the purpose of event-b proof obligations. In *Proceedings of the 1st international conference on Abstract State Machines, B and Z*, ABZ '08, pages 125–138. Springer-Verlag, 2008.
8. Russell R. Hurlbut. A survey of approaches for describing and formalizing use cases. Technical report, Expertech, Ltd., 1997.
9. Alexei Iliasov. Augmenting Event-B Specifications with Control Flow Information. In *NODES 2010*, May 2010.
10. Alexei Iliasov. Use case scenarios as verification conditions: Event-B/Flow approach. In *Proceedings of 3rd International Workshop on Software Engineering for Resilient Systems*, Septembre 2011.
11. Industrial deployment of system engineering methods providing high dependability and productivity (DEPLOY). IST FP7 project, online at http://www.deploy-project.eu/.
12. M.Butler and M.Leuschel. Combining CSP and B for Specification and Property Verification. pages 221–236, 2005.
13. The RODIN platform. Online at http://rodin-b-sharp.sourceforge.net/.
14. H. Treharne, S. Schneider, and M. Bramble. Composing Specifications Using Communication. In *Proceedings of ZB 2003: Formal Specification and Development in Z and B, Lecture Notes in Computer Science*, Vol.2651, Springer, Turku, Finland, June 2003.