Use case scenarios as verification conditions: Event-B/Flow approach

Alexei Iliasov

Newcastle University, UK

Abstract. Model-oriented formalisms rely on a combination of safety constraints and satisfaction of refinement obligations to demonstrate model correctness. We argue that for a significant class of models a substantial part of the desired model behaviour would not be covered by such correctness conditions, meaning that a formal development potentially ends with a correct model inadequate for its purpose. In this paper we present a method for augmenting Event-B specifications with additional proof obligations expressed in a visual, diagrammatic way. A case study illustrates how the method may be used to strengthen a model by translating use case scenarios from requirement documents into formal statements over a modelled system.

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 [5] 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. Our approach is different. We propose to exploit use cases in the course of a step-wise formal development process for the engineering of correct-by-construction systems. It is assumed that a library of use case scenarios is available together with a system requirements document and use cases are presented in a sufficiently precise manner. We discuss a technique and tool for expressing use case scenarios as formal verification conditions that become a part of a formal model of a developed system. It is guaranteed that the final product obtained from such a model posesses, by the virtue of the development method, all the properties expressed in use cases scenarios. The overall approach is generally in line with some existing work on formalisation and formal validation of use cases [8]. The approach is investigated on the basis of an extension of the Event-B modelling method [1] with a technique for formally capturing use case scenarios as theorems over model state.

The paper is organised as follows. Section 2 gives a brief overview of Event-B notation and methodology. The motivation behind the approach is presented in Section 3. The essential details of the approach and its integration with Event-B are presented in Section 4. Section 5 gives examples of use cases in the role of validation conditions for an Event-B model of a networking file system.

```
MACHINE M
  SEES Context
  VARIABLES $v$
  INVARIANT $I(c, s, v)$
  INITIALISATION ...
  EVENTS
    $E_1$ = any $vl$ where $g(c, s, vl, v)$ then $S(c, s, vl, v, v')$ end
    ...
END
```

Fig. 1. Event-B model structure.

2 Event-B

Event-B [1] is a formal modelling method for the realisation of correct-by-construction software systems.

An Event-B development starts with the creation of a very abstract specification. The cornerstone of 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 defined by a list of new local variables (parameters) vl, a state predicate g called *event guard*, and a next-state relation S called *substitution* (see the **EVENTS** section in Figure 1).

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

Model correctness is demonstrated by generating and discharging a collection of proof obligations. There are proof obligation demonstrating model consistency, such the preservation of the invariant by the events, and the refinement link to another Event-B model. Putting it as a requirement that an enabled event produces a new state v' satisfying the model invariant, the model *consistency* condition states that whenever an event on an initialisation action is attempted, there exists a suitable new state v' such that the model invariant is maintained - $I(v')$. This is usually stated as two separate conditions: event feasibility and event invariant satisfaction.

FIS $I(c, s, v) \wedge g(c, s, vl, v) \Rightarrow \exists v'.\ S(c, s, vl, v, v')$

INV $I(c, s, v) \wedge g(c, s, vl, v) \wedge S(c, s, vl, v, v') \Rightarrow I(c, s, v')$

The consistency of Event-B models, i.e., verification of well-formedness and invariant preservation as well as correctness of refinement steps is demonstrated by discharging relevant *proof obligations* (such as **INV** and **FIS**) that, collectively, define the *proof semantics* of a model. The Rodin platform [11], a tool supporting Event-B, is an integrated environment that automatically generates necessary proof obligations and manages a collection of automated provers and solvers autonomously discharging the generated theorems.

3 Motivation

A stack may be defined in Event-B as a pair of variables $stack \in 1 \,..\, top \rightarrow \mathbb{N}$ and $top \in \mathbb{N}$ where top is the stack top pointer and $stack$ is a sequence of values representing a stack contents. The following two events define the obvious stack manipulation operations.

$$\begin{aligned}\mathsf{push} &= \textbf{any } v \textbf{ where } v \in \mathbb{N} \textbf{ then } stack(top+1) := v \parallel top := top + 1 \textbf{ end}\\ \mathsf{pop} &= \textbf{when } top > 0 \textbf{ then } stack := 1 \,..\, top - 1 \lhd stack \parallel top := top - 1 \textbf{ end}\end{aligned}$$

It is not difficult to ascertain that the above model is indeed a specification of a stack. However, the model invariants (the typing conditions for variables *stack* and *top* given above) permit a wide range of safe but undesirable behaviours. For instance, we could have made a mistake in the definition of event *push*

$$\mathsf{push\_broken} = \textbf{any } v \textbf{ where } v \in \mathbb{N} \textbf{ then } stack(top+1) := v \textbf{ end}$$

Event push_broken does not violate the invariants and hence there is no feedback from the verification tools. Although the model is correct it cannot be used in the role of a stack, that is, it is an inadequate representation of stack. The problem is not in the specification itself: the invariant is as strong as it can be for this model. While, in some cases, it is possible to find a model abstraction [1] that, via the refinement relation, would demonstrate the necessary conditions, this is not always possible in practice due to often awkward models and interference with the use of refinement as a development method.

The problem is not artificial: in larger models, it is difficult to identify a problem by simply reading model text. An informal inspection of some recent Event-B developments [3] shows that industrially-inclined models (models of a piece of software rather than models of protocols or algorithms) tend to have less restrictive (but not necessarily less numerous) invariants, more involved event actions and exhibit the prevalence of horizontal refinement - a form of data refinement where all new model elements contribute to the behaviour on the new, hidden state. Correspondingly, in such models, much of behavioural specifications is not under an obligation to establish any verification conditions.

The issue of constructing a correct model that does what is expected from it is generally known as a problem of model adequacy and largely falls into the domain of requirements engineering. Here we study an application of one requirements engineering technique - use case scenarios - in the formal setting of the Event-B method. To illustrate the main point of the proposed technique let us consider the following (algebraic) specification of a stack object S holding elements of type $\mathbb{N}$.

[1] One attempt at such an abstraction may be found [9].

$$\begin{array}{ll} \mathit{init} \in S & \mathit{empty}(\mathit{init}) = \mathrm{TRUE} \\ \mathit{empty} : S \rightarrow \mathrm{BOOL} & \mathit{empty}(\mathit{push}(s,v)) = \mathrm{FALSE} \\ \mathit{push} : S \times \mathbb{N} \rightarrow S & \mathit{top}(\mathit{push}(s,v)) = v \\ \mathit{pop} : S \rightarrow S \times \mathbb{N} & \mathit{pop}(\mathit{push}(s,v)) = s \end{array}$$

In a contrast to the Event-B model above, it does not specify how the stack operations update the stack but rather defines few principal properties of stacks. This style makes it easier to spot unexpected model properties as the defining characteristics are given in an explicit and concise form whereas for a model-oriented formalism, like Event-B, one has to do some mental calculations.

An interesting exercise is to translate such algebraic properties into Event-B model theorems (that is, equivalent statements over *stack* and *top*). Notice that $\mathit{pop}(\mathit{push}(s,v)) = s$ may be put as

$$(\mathsf{pop} \circ \mathsf{push}) \subseteq \mathrm{id}(\Sigma)$$

where Σ is the state of the model: $\Sigma = \{s \times t \mid t \in \mathbb{N} \wedge s \in 1\,..\,t \rightarrow \mathbb{N}\}$; s and t are shorthands for *stack* and *top*. The relations *push* and *pop* are easily derived from the event definitions:

$$\mathsf{pop} = \{(s \mapsto t) \mapsto (s' \mapsto t') \mid t > 0 \wedge s' = 1\,..\,t-1 \lhd s \wedge t' = t-1\}$$
$$\mathsf{push} = \{(s \mapsto t) \mapsto (s' \mapsto t') \mid v \in \mathbb{N} \wedge s' = s \mathbin{<\!\!+} \{t+1 \mapsto v\} \wedge t' = t+1\}$$

The condition is proven by expanding the relational composition into a join:

$$\begin{aligned} \mathsf{pop} \circ \mathsf{push} &= \{(s \mapsto t) \mapsto (s'' \mapsto t'') \mid v \in \mathbb{N} \wedge s' = s \cup \{t+1 \mapsto v\} \wedge t' = t+1 \wedge \\ &\qquad t' > 0 \wedge s'' = 1\,..\,t'-1 \lhd s' \wedge t'' = t'-1\} \\ &= \{(s \mapsto t) \mapsto (s'' \mapsto t'') \mid \cdots \wedge s'' = s \wedge t'' = t\} \\ &= \{x \mapsto x \mid x \in \Sigma\} = \mathrm{id}(\Sigma) \end{aligned}$$

In the same manner, one can establish that condition $\mathsf{pop} \circ \mathsf{push\_broken} \subseteq \mathrm{id}(\Sigma)$ does not hold and therefore formally rule out the push_broken version of the *push* event.

The main difficulty in checking $(\mathsf{pop} \circ \mathsf{push}) \subseteq \mathrm{id}(\Sigma)$ is the construction of the verification goal from the event definitions. The resultant theorem is trivial for an automated prover. The proposal we discuss in this paper allows one to construct this kind of theorems very easily and in large quantities (when necessary) using a simple visual notation.

4 Flow language

The section presents the semantics of the Flow language and its visual notation.

4.1 Flow theorems

Consider the following relations over pairs of relations on some set S.

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

The definitions are concerned with the properties of a composite relation $g \circ f$. f **ena** g states that $g \circ f$ is defined for every value on which f is defined - $dom(g \circ f) = dom(f)$; relation f **dis** g implies that $g \circ f = \varnothing$; f **fis** g means that there is at least one pair of values satisfying relation $g \circ f$: $\exists v, u \cdot u\ (g \circ f)\ v$. The relations enjoy the following properties.

$$\begin{aligned} \mathbf{dis} \cap \mathbf{fis} = \varnothing &\Leftrightarrow \neg(f\ \mathbf{dis}\ g \wedge f\ \mathbf{fis}\ g) \\ \mathbf{ena} \cap \mathbf{dis} = \varnothing &\Leftrightarrow \neg(f\ \mathbf{ena}\ g \wedge f\ \mathbf{dis}\ g) \\ \mathbf{dis} \cup \mathbf{fis} = U &\Leftrightarrow f\ \mathbf{dis}\ g \vee f\ \mathbf{fis}\ g \end{aligned}$$

Let p_e, G_e, R_e characterise the parameters, guard and action of an event e. Assuming that the consistency proof obligations are discharged, the universe of system states Σ is said to be its safe states: $\Sigma = \{v \mid I(v)\}$. An event e is a next-state relation of the form $e \subseteq \Sigma \times \Sigma$. Let $\mathsf{before}(e) \subseteq \Sigma$ and $\mathsf{after}(e) \subseteq \Sigma$ signify the domain and the range of relation e. The set $\mathsf{before}(e)$ corresponds to the enabling states defined by the event guard and $\mathsf{after}(e)$ is a set of possible new states computed by the event:

$$\begin{aligned} &\mathsf{before}(e) = \{v \mid I(v) \wedge \exists p_e \cdot G_e(p_e, v)\} \\ &\mathsf{after}(e) = \{v' \mid I(v) \wedge \exists p_e \cdot (G_e(p_e, v) \wedge R_e(p_e, v, v'))\} \end{aligned}$$

Let b and h be some events. Taking into the account the definitions of before and after, relations **ena**, **dis**, **fis** may be expanded as follows.

$$\begin{aligned} b\ \mathbf{ena}\ h \Leftrightarrow\ & \mathsf{after}(b) \subseteq \mathsf{before}(h) \\ \Leftrightarrow\ & \{v' \mid I(v) \wedge \exists p_b \cdot (G_b(p_b, v) \wedge R_b(p_b, v, v'))\} \\ \subseteq\ & \{v \mid I(v) \wedge \exists p_h \cdot G_h(p_h, v)\} \\ \Leftrightarrow\ & \forall v, v', p_b \cdot I(v) \wedge G_b(p_b, v) \wedge R_b(p_b, v, v') \Rightarrow \exists p_h \cdot G_h(p_h, v')\ (\mathbf{FENA}) \end{aligned}$$

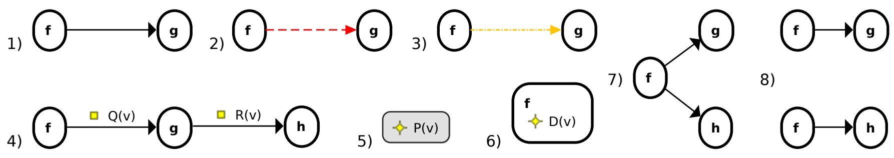
$$\begin{aligned} b\ \mathbf{dis}\ h \Leftrightarrow\ & \mathsf{after}(b) \cap \mathsf{before}(h) = \varnothing \\ \Leftrightarrow\ & \mathsf{after}(b) \subseteq \Sigma \setminus \mathsf{before}(h) \\ \Leftrightarrow\ & \mathsf{after}(b) \subseteq \{v \mid I(v)\} \setminus \{v \mid I(v) \wedge \exists p_h \cdot G_h(p_h, v)\} \\ \Leftrightarrow\ & \mathsf{after}(b) \subseteq \{v \mid I(v) \wedge \forall p_h \cdot \neg G_h(p_h, v)\} \\ \Leftrightarrow\ & \forall v, v', p_b, p_h \cdot I(v) \wedge G_b(p_b, v) \wedge R_b(p_b, v, v') \Rightarrow \neg G_h(p_h, v')\ \ (\mathbf{FDIS}) \end{aligned}$$

$$\begin{aligned} b\ \mathbf{fis}\ h \Leftrightarrow\ & \mathsf{after}(b) \cap \mathsf{before}(h) \neq \varnothing \\ \Leftrightarrow\ & \exists v, v', p_b, p_h \cdot I(v) \wedge G_b(p_b, v) \wedge R_b(p_b, v, v') \wedge G_h(p_h, v') \qquad (\mathbf{FFIS}) \end{aligned}$$

Conditions (**FENA**, **FDIS**, **FFIS**) are the main flow verification conditions. Event-B proof obligations of these form are automatically derived by a tool supporting the approach.

Assumptions and assertions It is convenient to construct new events from existing events. We define two operators for this: *assumption* and *assertion*. An assumption constraints the enabling set of an event while an assertion constraints the set of new states computed by the event.

$$\begin{aligned} P.e &\overset{\text{def}}{=} \{t \mapsto r \mid t \mapsto r \in e \wedge t \in P\} \qquad \mathsf{before}(P.e) = \mathsf{before}(e) \cap P \\ e.Q &\overset{\text{def}}{=} \{t \mapsto r \mid t \mapsto r \in e \wedge r \in Q\} \qquad \mathsf{after}(e.Q) = \mathsf{after}(e) \cap Q \end{aligned}$$



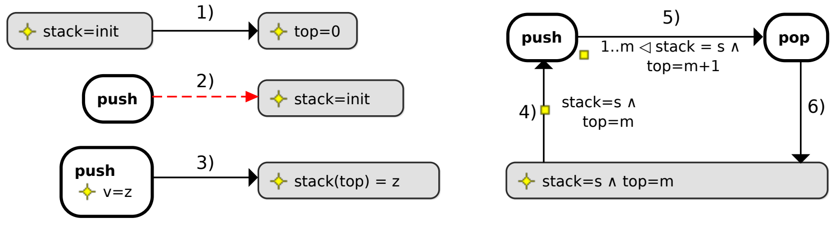
1) f **ena** g
2) f **dis** g
3) f **fis** g
4) f **fis** $\mathrm{q}.g \wedge \mathrm{q}.g$ **fis** $\mathrm{r}.h$
$\mathrm{q} = \{v \mid Q(v)\}, \mathrm{r} = \{v \mid R(v)\}$
5) $\mathit{skip}(C), C = \{v \mid P(v)\}$
6) $f(C), C = \{v \mid D(v)\}$
7) f **ena** $g \vee f$ **ena** h
8) f **ena** $g \wedge f$ **ena** h

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

It is trivial to see that such constrained events are safe: $P.e \subseteq e \wedge e.Q \subseteq e$. One special case is a constraint used in the roles of both assumption and assertion: $e(D) = D.e \cap D.e$. It is easy to see that $e(D)$ may be obtained by adding a guard predicate L to event e such that $D = \{v \mid L(v)\}$. It often necessary to make statement about a state rather than an event. This is done with the help of the skip event that does not change system state: $\mathrm{skip} = \mathrm{id}(\Sigma)$. Then $\mathrm{skip}(D)$ is a stuttering step constrained to states D.

4.2 Graphical notation

The approach is realised by a tool employing a visual, diagrammatic depiction of Flow theorems. A Flow diagram always exists in an association with one Event-B model. The theorems expressed in a Flow diagram are statements about the behaviour of the associated model. 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 **ena**, **dis**, **fis** are represented by edges connecting nodes ((Figure 2, 1-3)). Depending on how a diagram is drawn, edges (flow theorems) are said to be in *and* or *or* relation (Figure 2, 7-8). New events are derived 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) and implicit deadlock event stop (filled circle). The diagrams like those in Figure 2 (except 5 and 6 which are next-state relations rather than relations over events) are translated into theorems and appear as additional *proof obligations* for the associated Event-B model. A change in the diagram or Event-B model would automatically lead to the recomputation of affected proof obligations. Flow proof obligations are dealt with, like all other proof obligation types, by a combination of automated provers and interactive proof. Like in proofs of model consistency



1) $\mathrm{skip}(stack = \mathrm{init})$ **ena** $\mathrm{skip}(top = 0)$
2) $push$ **dis** $\mathrm{skip}(stack = \mathrm{init})$
3) $push(v = z)$ **ena** $\mathrm{skip}(stack(top) = z)$
4) $\mathrm{skip}(i)$ **ena** $i.push$
5) $i.push$ **ena** $j.pop$
6) $j.pop$ **ena** $\mathrm{skip}(i)$

$i = \{s \mapsto m\}$ $\quad j = \{t \mapsto m + 1 \mid 1 \,..\, m \lhd t = s\}$

Fig. 3. Flow specification for the verification of stack properties.

and refinement, the feedback from an undischarged Flow proof obligation may often be interpreted as a suggestion of a diagram change such as an additional assumptions or assertion.

Stack example Let us revisit the stack example from Section 3. The Flow diagram in Figure 3 constructs theorems checking that the algebraic stack properties are satisfied by the Event-B model of stacks. There are five theorems in the diagram. The first three of these check properties $empty(init) = \mathrm{TRUE}$, $empty(push(s, v)) = \mathrm{FALSE}$ and $top(push(s, v)) = v$. In the diagram, $init = \varnothing, z \in \mathbb{N}, m \in \mathbb{N}1$ and s is some arbitrary stack state. Property $pop(push(s, v)) = s$ is decomposed into three theorems: $push$ is enabled for an arbitrary state; after $push$, event pop is enabled; pop after $push$ returns the stack into the original state.

It is not difficult to formally demonstrate that the property $(\mathsf{pop} \circ \mathsf{push}) \subseteq \mathrm{id}(\Sigma)$ is implied by the Flow specification. Note that $i.push$ **ena** $j.pop \Leftrightarrow i.push$ **ena** $\mathrm{skip}(j) \wedge \mathrm{skip}(j)$ **ena** pop. Then the Flow theorems may be trabslated into set theoretic statements as follows.

$$\begin{array}{ll}
\mathrm{skip}(i)\ \textbf{ena}\ push & \Leftrightarrow i \subseteq \mathrm{dom}(push) \\
\mathrm{skip}(j)\ \textbf{ena}\ pop & \Leftrightarrow j \subseteq\ dom(pop) \\
i.push\ \textbf{ena}\ \mathrm{skip}(j) & \Leftrightarrow push[i] \subseteq j \\
j.pop\ \textbf{ena}\ \mathrm{skip}(i) & \Leftrightarrow pop[j] \subseteq i
\end{array}$$

The statements above is merely the predicate form of a relational join for pop and push : $i \subseteq \mathrm{dom}(push) \wedge push[i] \subseteq j \wedge j \subseteq \mathrm{dom}(pop) \wedge pop[j] \subseteq i \Rightarrow (pop{\circ}push)[i] \subseteq i$. Since i is an arbitrary stack state, it follows that $(\mathsf{pop}{\circ}\mathsf{push}) \subseteq \mathrm{id}(\Sigma)$.

4.3 Structuring

The Flow language offers a number of structuring mechanisms to help in the construction of larger diagrams. Some of the more important of them are ad-

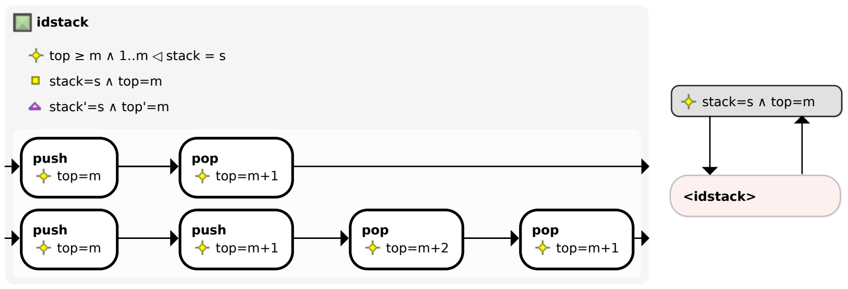


Fig. 4. Folding of idempotent stack scenarious.

dressed by a container diagram element called sub-scenario. A sub-scenario plays differing roles depending on whether it is open or closed.

An *open sub-scenario* defines a property that must be preserved by every event of contained scenarios. Because of the similarity with the way the condition of a safety invariant is formulated, such a property is called an *interval invariant*. Interval invariants are often used when a property is maintained for some part of scenario, and, compared to the propagation of such property via assumptions and assertions, it results in a larger set of (generally) simpler proof obligations as the interval invariant preservation is treated independently from the core theorems of enabling, disabling and feasibility. It also helps to produce more legible diagrams.

A *closed sub-scenario* is a black-box container with a well-defined interface. Externally, it appears to be a simple event (although no such event exists in the associated machine). Internally, it is a Flow specification that may be developed and manipulated independently from the rest of the diagram, subject to some additional constraints. Besides an interval invariant, a closed sub-scenario defines pre- and postconditions. The precondition is a predicate defining a state that is guaranteed to enable at least one of the contained scenarios. A postcondition is a next-state relation characterising the overall effect of a contained scenario as a relation between the initial state of the closed sub-scenario and the state computed by the last event of the scenario. A closed sub-scenario may be linked to a separate diagram file to facilitate collaborative development. An example of a closed sub-scenario is given on diagram in Figure **??** (a green square icon in the left-top distinguishes a closed scenario from an open one). From top to bottom, the predicates are: interval property, precondition and postcondition. In this example, the closed sub-scenario characterises scenarios that, when considered in isolation, have no effect on the stack size and contents. Two trivial examples of such scenarios are defined within the subscenario. On the right-hand size of the diagram is an example of a closed sub-scenario in the role of an event node.

The reason for having the sub-scenario element in two versions rather than two separate element kinds is that the construction of a closed scenario is invariably preceded by a stage when an open sub-scenario is defined. The first step in folding a scenario part into a closed sub-scenario is the identification of a property that holds for the whole part. It is such property that makes a piece of diagram distinct from the rest and, typically, is central to demonstrating the postcondition property.

Sometimes the same constraint or diagram pattern is used repeatedly throughout a scenario. To avoid visual clutter, such repeating parts may be declared separately as *aspects*. An aspect may define a predicate and contain a Flow diagram. The semantics of these are defined by the point of integration. For instance, a link to an aspect from an event node conjuncts the aspect property with existing event constraints.

5 Case study

As a case study we consider a model of networking file system loosely inspired by the design of NFSv4 [13]. The model focuses on the behaviour of a server accepting requests from clients. Somewhat unusually, requests come in the form of simple programs - operation sequences - executed non-atomically (interleaved with other such requests) on the server side. The server is free to schedule incoming requests as it likes (in a real system this has to do with file locking mechanism). At any moment, there may be any number of running requests but the operations on a file system are always executed atomically. A request may be aborted if the server discovers that the current operation may not be executed. If a request succeeds, the client may receive some data as the request result.

Such system architecture makes it difficult to introduce server functionality as series of small-step refinements and provide strong safety invariants. On the other hand, although it is a relatively small model, there is a number of interesting use cases.

The basic notions of the model are the following.

- m	file system (current state)
- Q	set of known (accepted) requests
- q	set of active (running) requests, $q \subseteq Q$
- $p(t)$	operation vector of request $t \in q$

The state of a running request t is characterised by a pointer $c(t)$ to the current operation in the operation vector $p(t)$, an error flag $e(t)$ and a pair of data register $r_1(t), r_2(t)$ used to store parameter values for the request operations. The first register is normally used to store the path to a file and the second holds data that may be written to file or was read from a file.

register	*meaning*
$c(t)$	operation counter
$r_1(t)$	data register 1
$r_2(t)$	data register 2
$e(t)$	error flag

The request scheduler of the server is defined by four high-level operations: the acceptance of a new request; starting the execution of a request; request finalisation; request abort. The latter two create and send a reply message to a client that has submitted the request.

operation	*semantics*	*meaning*
NEW r, d_1, d_2	$Q' = Q \cup \{r \mapsto d_1 \mapsto d_2\}$	accepts new request (r, d_1, d_2)
FIN r	$q', Q' = \{r\} \lhd q, \{r\} \lhd Q$ [2]	finalises the current request
PICK r	$q', c'(t) = q' \cup \{r\}, 1$ $p'(r) \mapsto r'_1(r) \mapsto r'_2(r) = Q(r)$	prepares a requests for execution
ABORT r	$q', Q' = \{r\} \lhd q, \{r\} \lhd Q$	aborts the current request

We define a small subset of possible request operations: addition of a new file (not existing already in the system); deletion of an existing file; overwriting the contents of an existing file; file read; file search; and register swap acting as a connector between some other operations. The following is the summary of operations and their beaviour.

name	*semantics*	*condition*	*meaning*
ADD	$m' = m \cup \{r_1 \mapsto r_2\}$	$r_1 \notin m$	adds new file r_1 with contents r_2
DELETE	$m' = \{r_1\} \lhd m$	$r_1 \in m$	deletes existing file r_1
UPDATE	$m' = m \mathbin{\lhd\!\!\!-} \{r_1 \mapsto r_2\}$	$r_1 \in m$	overwrites existing file r_1 with r_2
READ	$r'_2 = m(r_1)$	$r_1 \in m$	reads an existing file
LOOKUP	$r'_2 = r_1$	$r_1 \in m$	looks for a file
XCHG	$r'_1, r'_2 = r_2, r_1$		register swap

All the operations implicitly update program counter $c(t)$. In addition, when the associated condition is not satisfied, an operation does nothing but raises flag $e(t)$. For the convenience of modelling, operations in $p(t)$ are given in reverse order: $1 \mapsto \mathsf{DELETE}, 2 \mapsto \mathsf{ADD}$ is understood as first ADD and afterwards DELETE.

We have build an Event-B model of the system comprised of four refinement steps. As an illustration, at the last refinement step, the ADD operation is realised by the following event.

```
add = **any** $t$ **where**
        $q \neq \varnothing \ t \in q$
        $c(t) > 0$
        $p(t)(c(t)) = \mathsf{ADD}$
        $r1(t) \in \mathrm{DATA} \setminus \{\mathrm{NIL}\}$
        $r1(t) \notin \mathrm{dom}(m)$
    **then**
        $m(r1(t)) := r2(t)$
        $c(t) := c(t) - 1 \parallel p(t) := 1 \,..\, c(t) - 1 \lhd p(t)$
    **end**
```

The event picks some request t such that it is not finished $(c(t) > 0)$ and its current operation is ADD, i.e., $p(t)(c(t)) = \mathsf{ADD}$. The event also requires that $r1(t)$ contains a valid fresh file name.

In the continuation of the section we discuss three use cases introducing additional verification constraints into the model.

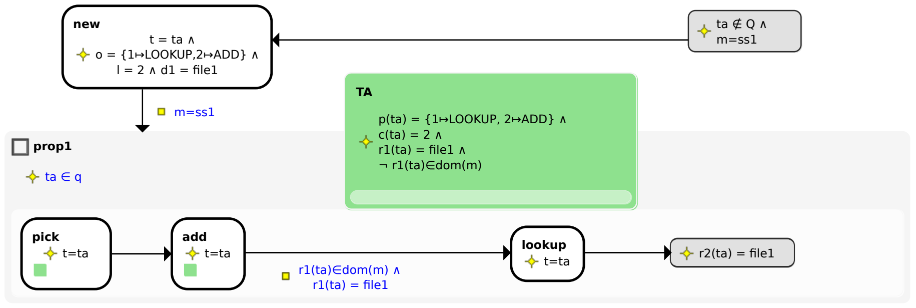


Fig. 5. File look-up after addition.

5.1 Use case 1: file look-up after addition

The use case states that a new file (not previously known in the file system) added with operation ADD may be found with operation LOOKUP immediately after executing ADD. The scenario checks that the notion of file creation implemented by ADD is compatible with the notion of file existence implemented by LOOKUP.

The Flow diagram (see Figure **??**) encoding the use case starts with a premise that the file system is in some state $ss1$ and request name ta is not in the pool Q of ready requests. We also rely on the fact that a file with name *file1* does not exists in $m = ss1$. The first step is the creation of a new request made of operations ADD and LOOKUP with register one set to file name *file1*. It is not important what ADD writes in file *file1* thus register two is left unconstrained in event new. To simplify the definition of the remaining of the scenario, the information about the request operations is deposited in an aspect (green box titled *TA*). Events pick and add integrate this aspect.

After a request is formed it may be selected for execution. This is achieved with event pick. After pick, it is known that ta is some currently running request and this is reflected in the sub-scenario property $ta \in q$. Event add may be executed after pick since, from the aspect definition, it is known that the current operation of the operation vector of ta is pointing at value ADD. The after-state of add defines an updated operation counter $c'(ta)$ pointing at operation LOOKUP. This information is deduced automatically by the provers. Finally, after event lookup we are interested in stating that the second register holds a value corresponding to the name of the created file. To construct the proof it is necessary to propagate the information about the effect of executing ADD in the request. The property we need is that *file1* now exists in the file system and the file we are looking for with operation LOOKUP is, in fact, the same file *file1*. Once the property is added the proof goes automatically.

On this and the following Flow diagrams, predicates highlighted in a ligher shade indicate constraints discovered during the proof session. In other words, these elements were not a part of the initial diagram but were necessary to accomplish the proofs.

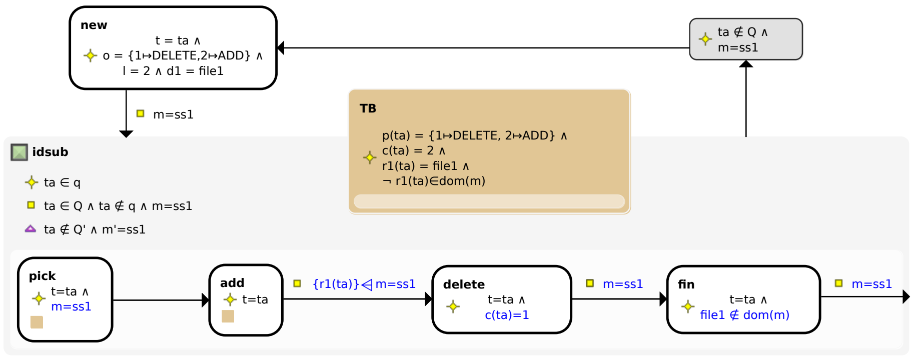


Fig. 6. Idempotent requests template and a sample request specification.

5.2 Use case 2: idempotent requests

An important family of use cases are those concerned with requests that do not change file system state if executed without an interference from other requests. The only effect of executing such an (idempotent) request is that it disappears from the pool of prepared requests. There is a large number of scenarios that fit this template. The common property is that they start and end with the same file system state ($m = ss1 \land m' = ss1$) while some previously inactive request ta would be removed from the pool of ready requests ($ta \in Q \land ta \notin Q'$). Since the only way to remove a request from Q is by executing it till completion or abort, these properties (formally, pre- and postconditions) define an execution of an idempotent request. (READ), (LOOKUP) and (XCHG) are the only single-operation requests that should not change the file system state (the former two might result in an aborted request).

As discussed in Section 4.3, a closed sub-scenario defines a family of independent sub-projects that may be, logically and technically, considered in separation from the main Flow diagram. This is an important practical consideration as it allows one to construct large diagrams without overloading the tool infrastructure. There is also a degree of proof economy as the top three arrows are theorems proven just once for the whole family of sub-projects.

One example of an idempotent request is shown on the diagram in Figure **??**. It is programmed to delete a file added in the same request. Since the addition operation requires that an added file is not already present in the file, the subsequent deletion of an added file essentially nullifies the effect of addition. With condition $\{r1(ta)\} \lhd m = ss1$ we prove that, in the after-state of add, the state of the file system is exactly as before except the appearance of newly added file $r1(ta)$. This property allows us to prove that event delete followed by fin returns the file system into its original state $m = ss1$ by deleting file $r1(ta)$. After finalizing the request (event delete) we prove that the system is back into the state where ta is not a known request and $m = ss1$.

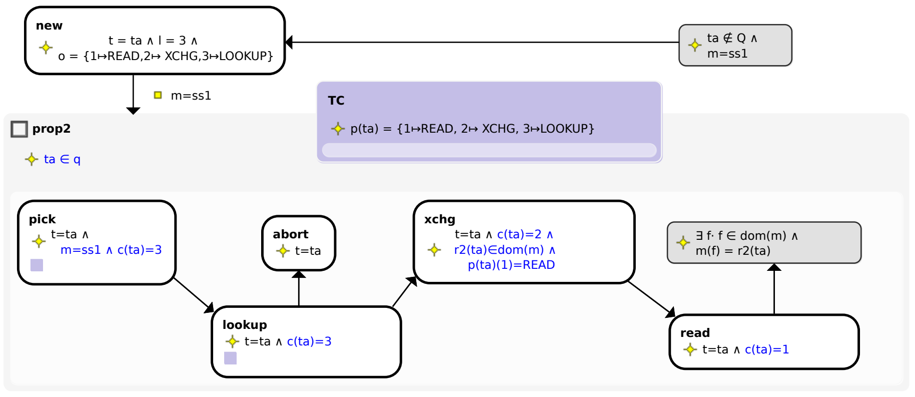


Fig. 7. If a file is found it may be read.

5.3 Use case 3: file may be read if found

This use case checks the interplay of file existence and file reading. The flow diagram in Figure **??** checks that operation LOOKUP either aborts a request or stores in register $r2$ the name of some existing file and, after copying $r2$ into $r1$, reading of the file with operation READ always succeed and delivers a result in register $r2$ that is the contents of some existing file.

In this scenario, we use branching to select only one possible case of LOOKUP execution. Also, the link between LOOKUP and READ includes an intermediate operation XCHG which requires the propogation via event constraints of the relevant results of lookup execution.

An interesting point is, assuming that operations READ, XCHG and ABORT are correct, that this use case gives a complete characterisation of the behaviour of LOOKUP: the operation either fails or stores in r_2 the name of a file that exists in the file system. This hints at the possibility of synthesis of model parts from detailed Flow diagrams.

The model of the case study (including both the Event-B model and the Flow diagrams) is available to download as a Rodin project [9]. In addition to the core Rodin Platform it is necessary to install the Rodin Flow plugin (instructions may be found in [10]). The diagrams in this paper were produced by printing the diagrams directly from the tool into a PDF file. For presentation purposes, variable names in the diagrams were shortened.

The three scenarios combined have produced 25 non-trivial proof obligations of which 22 were discharged automatically and the other 3 required a few simple steps in the interactive prover. This is a promising indication of the approach scalability. It is necessary to note that scenario diagrams were constructed in a number of iterations and it took several failed proof attempts to arrive at the presented Flow diagrams. The overall number of proof obligations for the model is 167 with 17 assisted proofs.

6 Conclusion

Flow may be seen as a form of a primitive temporal logic with a very limited expressive power compared, for instance, to CTL. To illustrate this point, assume $[f]$ is a predicate for "f has fired". Then Flow relations may be compared to CTL statements as follows:

$$\begin{aligned}
f\ \mathbf{ena}\ g &\Rightarrow ([f] \Rightarrow EX[g]) \\
f\ \mathbf{dis}\ g &\Leftrightarrow ([f] \Rightarrow AX\neg[g]) \\
f\ \mathbf{fis}\ g &\Leftarrow ([f] \Rightarrow EX[g])
\end{aligned}$$

Thus, individual flow theorems characterise the immediate future behaviour and it would take a chain of such theorems to approximate more interesting statements. For instance, from a fact that f enables g and g enables h one may conclude that after f there is a path leading to h: $(f\ \mathbf{ena}\ g \wedge g\ \mathbf{ena}\ h) \Rightarrow ([f] \Rightarrow EF[h])$. In principle, a statement like $[f] \Rightarrow EF[g]$ is expressible as a flow scenario for any two events f and g[3]. On the other hand, in most cases, it should be impractical to approximate $[f] \Rightarrow AF[g]$ and $[f] \Rightarrow AG[g]$ as this requires formulation of possible continuation scenarios and a proof (possibly, via **dis**) that no other continuation exists.

The most closely related work is a study of liveness-style theorems for the Classical B [2]. 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 Flow, 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 [12, 7, 4, 14] on combining process algebraic specification with event-based formalisms such as Event-B and Action Systems. The fundamental difference is that Flow 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 Flow approach.

Acknowledgments

This work is supported by FP7 ICT DEPLOY Project and the EPSRC/UK TrAmS platform grant.

[3] This requires a side condition that any loops are proven convergent in the Event-B part.

References

1. J.-R. Abrial. *Modelling in Event-B*. Cambridge University Press, 2010.
2. 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.
3. Event-B.org. Event-B model repository. 2011. Available at http://deploy-eprints.ecs.soton.ac.uk/view/type/rodin=5Farchive.html.
4. 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.
5. Russell R. Hurlbut. A survey of approaches for describing and formalizing use cases. Technical report, Expertech, Ltd., 1997.
6. M. Leuschel and M. Butler. ProB: A Model Checker for B. In Araki Keijiro, Stefania Gnesi, and Mandrio Dino, editors, *Formal Methods Europe 2003*, volume 2805, pages 855–874. Springer-Verlag, LNCS, 2003.
7. M.Butler and M.Leuschel. Combining CSP and B for Specification and Property Verification. pages 221–236, 2005.
8. Victor M. Mendoza-Grado. Formal Verification of Use Cases. In *Requirements Engineering: Use Cases and More*, 1995.
9. Flow Models of stack and NFS. Event B/Flow specification. 2011. Available at http://iliasov.org/usecase/nfs.zip.
10. Flow Plugin. Event-B wiki page. 2011. Available at http://wiki.event-b.org/index.php/Flows.
11. The RODIN platform. Online at http://rodin-b-sharp.sourceforge.net/.
12. 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.
13. NFSv4 web page. Network File System Version 4 . 2011. Available at http://datatracker.ietf.org/wg/nfsv4/.
14. Jim Woodcock and Ana Cavalcanti. The Semantics of Circus. In *ZB '02: Proceedings of the 2nd International Conference of B and Z Users on Formal Specification and Development in Z and B*, pages 184–203, London, UK, 2002. Springer-Verlag.