Available online at www.sciencedirect.com

ScienceDirect

Science of Computer Programming 00 (2010) 1–24

Science of Computer Programming

Validation of Formal Models by Refinement Animation

Stefan Hallerstede, Michael Leuschel, Daniel Plagge

Institut für Informatik, Universität Düsseldorf
Universitätsstr. 1, D-40225 Düsseldorf

Abstract

We provide a detailed description of refinement in Event-B, both as a contribution in itself and as a foundation for the approach to simultaneous animation of multiple levels of refinement that we propose. We present an algorithm for simultaneous multi-level animation of refinement, and show how it can be used to detect a variety of errors that occur frequently when using refinement. The algorithm has been implemented in ProB and we applied it to several case studies, showing that multi-level animation is tractable also on larger models. We present empirical results and discuss how the algorithm can be combined with symmetry reduction.

Keywords: Refinement, Model Checking, Constraint-Solving, Tools, Industrial Applications, Event-B

1. Introduction and Motivation

We consider formal modelling of software systems an important phase in the development process. It permits us to reason about the system to be developed early on in the development using abstractions of the system. We believe that this reasoning is a key to improvement of the quality of the developed software. The approach to reasoning varies depending on what kind of problems one wants to uncover. It can reach from formal proof to just "trying out" a model. We expect to benefit by improving our understanding of a system (by analysing models of it) and by creating an adequate formal model that can serve as the reference for what is to be considered "correct" in later stages of the development process. As much as possible reasoning should be supported by a software tool. The work presented in this article is based on the Event-B modelling method [1] and the complementing software tool Rodin [2]. The core of the Rodin tool provides automatic generation of proof obligations that can be analysed to improve understanding of a model. Often proof obligations give good indications of how to make an improvement in case of inconsistencies in a model. However, there are also many occasions where proof obligations do not point directly to a problem or where a model does not contain inconsistencies but is still "incorrect" (see, e.g., the Earley parser example discussed in [3]). In many such cases animation is a useful tool to gain further insight into a model. The Rodin plugins ProB [4, 5], Brama[1] [6], and AnimB [7] provide animation facilities for Event-B.

When dealing with complex models, refinement can be used to introduce the many details gradually, achieving a reduced complexity at each refinement level. Proof obligations, once discharged, demonstrate correctness of a

Email addresses: `halstefa@cs.uni-duesseldorf.de` (Stefan Hallerstede), `leuschel@cs.uni-duesseldorf.de` (Michael Leuschel), `plagge@cs.uni-duesseldorf.de` (Daniel Plagge)

[1]Brama requires an older version (0.9.2.x) of Rodin at the time of writing.

refinement: the more detailed model behaves within the limits set by the more abstract model. However, it can be difficult to analyse a refinement relationship only by means of associated proof obligations. Animation can be used to complement proof as a method of validation providing a broader view on the properties of formal models. All three animation plugins mentioned above provide some means to animate refinements. In this article we investigate their relative capabilities and how to advance refinement animation in order to turn it into a tool for refinement validation. This serves as a blueprint for the evolution of ProB in terms of animation support.

Before starting the investigation we should be clear about the objectives of animation when used for validation. What is the purpose of animating a model across multiple levels of refinement? In Event-B several concepts play a role in refinement. Most prominently, these are invariants, guards, actions, and witnesses. If a refinement fails, any combination of those concepts may be involved. Animation should help to locate the cause of a problem in the model, pointing to specific invariants, guards, and so on, if possible. However, even if a refinement is formally correct, there can still be problems with the model. This concerns, in particular, properties that have not been formalised. Animation should make it easy to experiment with a model, visualising potential problems. We try to integrate this aspect of animation with the first one as far as possible. Otherwise consistent tool support for both would be difficult to realise.

In Section 2 we introduce the Event-B notation (using the syntax of the Event-B text editor "Camille" [8]) and give a concise description of the fundamentals of Event-B refinement. As far as we are aware there is no single place where all the essential aspects are described and motivated in such detail. All of this is needed in order to present the basic refinement-animation algorithm in Section 3. The presentation of the algorithm is interspersed with methodical remarks on validation. In Section 4 we present some concrete examples on how to use ProB for refinement validation and some brief description of case studies to which it has been applied. Finally, Sections 6 and 7 contain a discussion of related work and a conclusion.

2. Modelling and Refinement in Event-B

Event-B can be used to model complex intricate systems. To understand the system and the model of the system we need to reason thoroughly about the model. Such reasoning is the principal purpose of Event-B. The basic concepts of Event-B are characterised by means of proof obligations; they are the core of the Event-B method. However, they are not an exclusive means of reasoning. Based on an operational interpretation of a model we can also animate it to gain deeper understanding. In this section we parallel the presentation of Event-B proof obligations, in particular, refinement, with ideas of animation. This demonstrates well how animation complements proof. Because there is no single software tool for animation that provides all that is needed, we use the three tools ProB, Brama, and AnimB at the same time.

2.1. Contexts

Event-B models are described in terms of the two basic constructs: *contexts* and *machines*. Contexts specify static parts of a model, that is, *carrier sets* and *constants* that are constrained by *axioms*, introduced by the keywords sets, constants, and axioms, as shown in Fig. 1. Usually, axioms are quite simple formulas; contexts are intended to be

```
context CofCtxt
constants full empty half level
sets FILL
axioms
   @Ffhe partition(FILL, {full}, {half}, {empty})
   @lvl level = (0 .. 2 × {empty}) ∪ (3 .. 7 × {half}) ∪ (8 .. 11 × {full})
end
```

Figure 1: Context of Coffee Dispenser Model

used to parametrise machines. We mention contexts here mainly because of the role they play in animation. For any particular animation specific values for all constants have to be found. ProB does this automatically using constraint-solving techniques to find proper values that satisfy all axioms. The constraint-solving also determines whether the axioms contain a contradiction. In order to use carrier sets, constants, and axioms of a context in a machine, the context needs to be referenced in the machine: the machine sees the context. For instance, in Fig. 3 machine *CoffeeM* sees context *CofCtxt*. Contexts are described in more detail in [1, 9].

2.2. Notation for Variables and Substitutions

We follow the style of [1] of expressing variables and substitution in formulas. Let $v = v_1, \ldots, v_n$ be a sequence of n distinct variables, $t = t_1, \ldots, t_n$ a sequence of n formulas and F a formula. Then $F[t/v]$ is obtained from F by replacing simultaneously all free occurrences of each v_i by t_i.

We let $F(v)$ denote a formula, whose free variables are among $v_1, \ldots, v_n$. Once the formula $F(v)$ has been introduced, we denote by $F(t)$ the formula $F[t/v]$ with v replaced by t. By v' we denote the variables $v'_1, \ldots, v'_n$, used for naming after-states of state transitions. For (disjoint) variable sequences v and w we denote their concatenation by v, w. These notions could also be used with Event-B constants. But we do not do this in this article where we focus on refinement relationships and consequently on variables.

2.3. Machines

Machines provide behavioural properties of Event-B models. Machines may contain *variables*, *invariants*, *events*, and *variants*. Variables v define the state of a machine. They are constrained by invariants $I(v)$. Possible state changes are described by means of events. Each event is composed of a *guard* $G(t, v)$ and an *action* $S(t, v)$, where t are *parameters* of the event. The guard states the necessary condition under which an event may occur, and the action describes how the state variables evolve when the event occurs. We denote an event $E(v)$ by one of the forms shown in Fig. 2. Form (b) is used if event $E(v)$ does not have parameters, and form (c) if in addition the guard equals *true*.

```
any  $t$  when
  $G(t, v)$
then
  $S(t, v)$
end
```

(a) Event (General Form)

```
when
  $G(v)$
then
  $S(v)$
end
```

(b) Event without Parameters

```
begin
  $S(v)$
end
```

(c) Event without Parameters or Guard

Figure 2: Syntax of Events

A dedicated event of the third form is used for *INITIALISATION*. In the formal exposition below, we assume without loss of generality that the most general first form is used.

The action of an event is composed of *assignments* of the form: $x := E(t, v)$ or $x :\in E(t, v)$ or $x :\mid Q(t, v, x')$, where x is some or all of the variables v, $E(t, v)$ expressions, and $Q(t, v, x')$ a predicate. The second form assigns x to an element of a set, and the third form assigns to x a value satisfying a predicate. The first two can be formally defined in terms of the third form: $x := E(t, v) \mathrel{\widehat{=}} x :\mid x' = E(t, v)$ and $x :\in E(t, v) \mathrel{\widehat{=}} x :\mid x' \in E(t, v)$. The effect of an assignment is described by a before-after predicate:

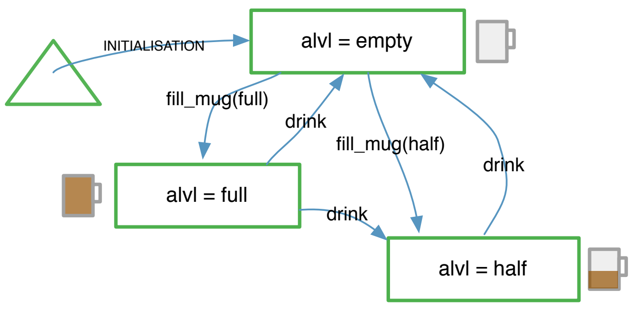
$$\text{before-after predicate of “}x :\mid Q(t, v, x')\text{”} \quad \widehat{=} \quad Q(t, v, x')$$

A before-after predicate describes the relationship between the state just before an assignment has occurred, x, and the state just after the assignment has occurred, x'. All assignments of an action $S(t, v)$ occur simultaneously which is expressed by conjoining their before-after predicates, yielding a predicate $A(t, v, x')$. Variables y that do not appear on the left-hand side of an assignment of an action are not changed by the action. Formally, this is achieved by conjoining $A(t, v, x')$ with $y' = y$, yielding the predicate

$$\boldsymbol{S}(t, v, v') \quad \widehat{=} \quad A(t, v, x') \wedge y' = y \quad .$$

```
machine  $CoffeeM$  sees  $CofCtxt$
variables  $alvl$
invariants
   @inv1  $alvl \in FILL$
variant  $(\{full \mapsto 2, half \mapsto 1, empty \mapsto 0\})(alvl)$
events
   event  $INITIALISATION$
      begin
         @mf  $alvl := empty$
      end
   event  $fill\_mug$
      any  $x$  when
         @g0  $alvl = empty$
         @g1  $x \neq alvl$
      then
         @a1  $alvl := x$
      end
   convergent event  $drink$
      when
         @g1  $alvl \neq empty$
      then
         @a1  $alvl :\in \{empty, half\} \setminus \{alvl\}$
      end
end
```

(a) Coffee Dispenser Machine



(b) State Space for *CoffeeM* (with Additional Mugshots)

Figure 3: Coffee Dispenser Model

2.3.1. *Running Example*

We use the coffee dispenser model in Fig. 3 (with context in Fig 1) and Fig. A.19 for illustration of refinement-animation. In the abstract machine *CoffeeM* the dispenser can fill a mug half or fully; the state of the mug is represented by the variable *alvl* (abstract level). As a special service the dispenser can also drink the coffee.

In the first refined machine *CoffeeR1* a feature is introduced for inserting an arbitrary number of coins into the dispenser. A coin is consumed each time a mug is filled. In the second refined machine *CoffeeR2*, the number of coins maximally accepted is limited and the amount of coffee contained in a mug is represented numerically by a the variable *clvl* (concrete level).

2.3.2. *Animation in ProB*

The before-after predicate can be used to compute the state space of a machine, a graph where each node represents a state of the machine and each arc the execution of an event. Fig. 3b contains the state space of the *CoffeeM* (Fig. 3a), as computed by the ProB tool. The triangle represent a special root node, where the variables and constants of a machine have not yet been set. An animator lets the user navigate the state space by choosing the events to be fired. A model checker will systematically explore the state space, looking for various errors in the machine.

2.4. *Machine Consistency*

Invariants are supposed to hold initially and whenever variable values are changed by an event. Obviously, this does not hold a priori and, thus, needs to be proved. The corresponding proof obligation for every event is called *invariant preservation*, formally,

$$I(v) \land G(t,v) \land \mathbf{S}(t,v,v') \Rightarrow I(v') \quad . \tag{1}$$

For the *INITIALISATION* of a machine there is a special form of this proof obligation, without invariant and guard in the hypothesis. By proving *action feasibility* for an event,

$$I(v) \ \wedge \ G(t,v) \ \Rightarrow \ (\exists v' \cdot \boldsymbol{S}(t,v,v')) \quad ,$$

as well, we achieve that $\boldsymbol{S}(t,v,v')$ provides an after state whenever $G(t,v)$ holds. This means that the guard indeed represents the enabling condition of the event. For the *INITIALISATION* action feasibility is just $(\exists v' \cdot \boldsymbol{S}(t,v,v'))$ because it does not have a guard and the invariant cannot be assumed to hold before initialisation of a machine.

2.4.1. Enabledness

We can prove that machine M is deadlock-free, that is, always some event is enabled to occur. Let $E_1(v)$, $E_2(v)$, ..., $E_n(v)$ be the events of machine M without *INITIALISATION*, with parameter lists t_1, t_2, ..., t_n, and guards $G_1(t_1,v)$, $G_2(t_2,v)$, ..., $G_n(t_n,v)$. Machine M is deadlock-free if some choice of parameter values makes one of the guards of the events true:

$$I(v) \ \Rightarrow \ (\exists t_1 \cdot G_1(t_1,v)) \ \vee \ (\exists t_2 \cdot G_2(t_2,v)) \ \vee \ \ldots \ \vee \ (\exists t_n \cdot G_n(t_n,v))$$

The Rodin tool does not support enabledness proof obligations at the moment. But ProB supports analysis of liveness properties and animation can show a deadlock (where all events except for the initialisation are disabled).

2.5. Machine Refinement

A machine N can refine at most one other machine M. We call M the *abstract machine* and N a *concrete machine*. The state of the abstract machine is related to the state of the concrete machine by a *gluing invariant* $J(v,w)$ associated with the concrete machine N, where v are the variables of the abstract machine and w the variables of the concrete machine. The restriction that there is at most one abstract machine to a concrete machine permits us to store the gluing invariant $J(v,w)$ in the concrete machine. The full invariant of the concrete machine is the conjunction of the (full) invariant of the abstract machine conjoined with the gluing invariant. The invariants are accumulated during refinements. In particular, they are immediately available as hypotheses in proofs.

We call events of an abstract machine *abstract events* and those of a concrete machine *concrete events*. Abstract events are refined by concrete events. Each abstract event must be refined by at least one concrete event. The most common case is an abstract event $E(v)$ refined by one concrete event $F(w)$. Let abstract event $E(v)$ and concrete event $F(w)$ be as specified in Fig. 4. Informally, concrete event $F(w)$ refines abstract event $E(v)$ if, whenever the gluing

```
event $E(v)$
  any
    $t$
  when
    $G(t,v)$
  then
    $S(t,v)$
  end
```

(a) Abstract Event $E(v)$

```
event $F(w)$ refines $E(v)$
  any
    $u$
  when
    $H(u,w)$
  with
    $W(t,v',u,v,w,w')$
  then
    $T(u,w)$
  end
```

(b) Concrete Event $F(w)$

Figure 4: Event Refinement

invariant $J(v,w)$ is true:

(i) the guard of $F(w)$ is stronger than the guard of $E(v)$, and

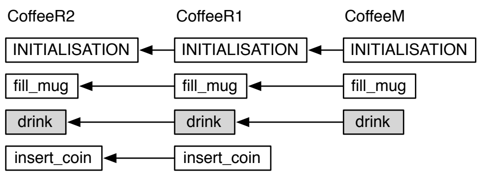
(ii) for every possible execution of $F(w)$ there is a corresponding execution of $E(v)$ which simulates $F(w)$ such that the gluing invariant remains true after execution of both events.

Superposition refinement [10] is supported explicitly by way of *extending* events. For instance, in Fig. A.19 some events carry the attribute extends. This means that all parameters, guards, and actions are copied literally from the abstract event. Note that the event $F(w)$ contains one more component $W(t, v', u, v, w, w')$ following the keyword with, called the *witnesses*. We return to its rôle in Section 2.7 below.

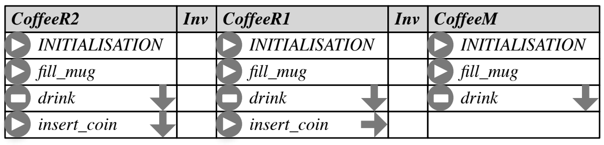
An event can be *split* in a refinement. This happens if more than one concrete event refines abstract event $E(v)$. A concrete event can also *merge* several abstract events into one concrete event. This is only permitted if the parameters of the merged abstract events agree on their types and their actions are identical. The latter restrictions have been introduced in order to keep the associated proof obligations simple. It is not possible for such a concrete event to extend the merged abstract events.

2.5.1. Refinement Animation

To check whether the guard of a concrete event is stronger, we also need to animate the corresponding abstract machine. Fig. 5a shows a graphical visualisation (created with Brama) of an animation of the coffee dispenser model described earlier.[2] White boxes signal enabled events, grey boxes disabled events. As can be seen, all the refinement



(a) in Brama



CoffeeR2	*Inv*	*CoffeeR1*	*Inv*	*CoffeeM*
▶ *INITIALISATION*		▶ *INITIALISATION*		▶ *INITIALISATION*
▶ *fill_mug*		▶ *fill_mug*		▶ *fill_mug*
■ *drink* ↓		■ *drink* ↓		■ *drink* ↓
▶ *insert_coin* ↓		▶ *insert_coin* →		

(b) Improved

Figure 5: Coffee Dispenser Refinement Animation

levels are animated concurrently. Brama's representation also shows at a glance that whenever a refined event is enabled, then all of its ancestor events are also enabled.

The view underlying Fig. 5a is operational. It focuses solely on event execution. If we wanted to use it for analysing a formal model, we would need to add information. In particular, information about gluing invariants would be useful (Fig. 5b). For instance, it could be shown which event could violate the invariant. We have indicated more information that could be provided by the icons used in Fig. 5b: the icon "▶" indicates that an event is enabled; the icon "■" indicates that an event is not enabled; the icon "↓" indicates that a convergent event decreases the variant; the icon "→" indicates that an anticipated event does not increase the variant. Note, that this is not a purely cosmetic change: the animator must supply all necessary information.

2.6. Common Variables and Common Parameters

As far as animation and model checking are concerned, refinement introduces a new challenge: we no longer have just a single machine that needs to be animated as in Fig. 3b, but a series of machines, each with its own state.[3]

In order to check the gluing invariant, we need to access variables from various machines. This raises a new issue. In Section 2.5 we have simply assumed that all variables v are refined by new variables w, and all parameters t are refined by new parameters u. The variables v and parameters t "disappear" in the refinement. In practice, variables and parameters can be repeated in a refinement. Abstract machines and concrete machines can have variables in common, and abstract events and concrete events parameters. By convention, when repeating variables and parameters abstract and concrete counterparts are assumed to be equal.[4]

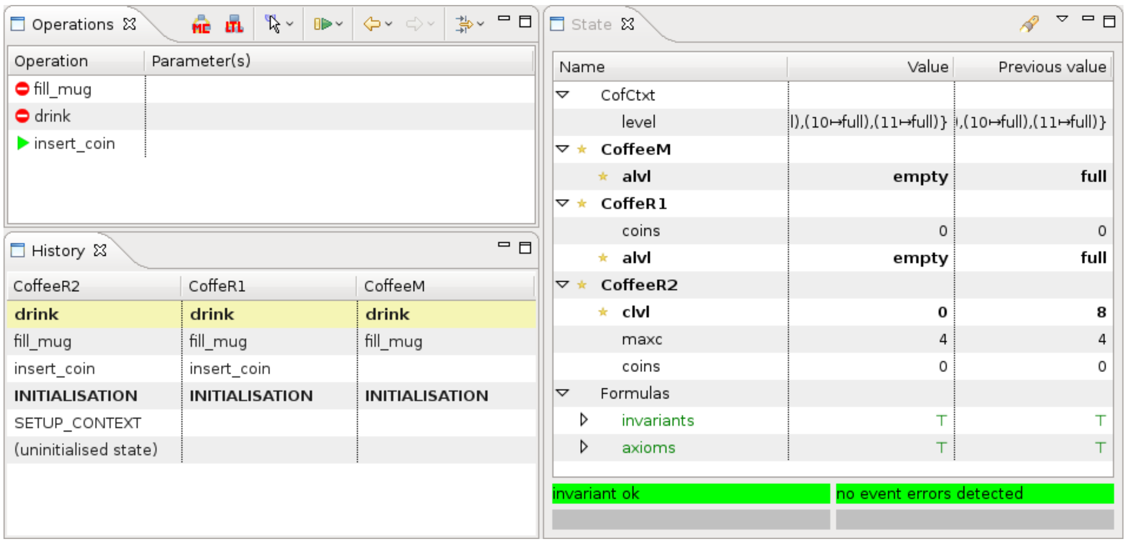
[2]The drawing corresponds to the output generated by Brama. The original screenshot can be found in [11].

[3]Earlier versions of ProB avoided this problem by animating each refinement level separately, at the cost of not being able to check the gluing invariant and of less user feedback.

[4]Once a variable has disappeared in the course of several refinements it cannot reappear. The reason for this is that the equality cannot be established by means of a machine that does not contain the variable. Furthermore, invariants are accumulated in Event-B. So it is not possible to reintroduce a variable with a different meaning.

2.6.1. *Animation*

For refinement animation of machines this means that we have basically two options: First, variables must be renamed in each machine and gluing invariants generated. This approach would allow us to visualise machines with deviating behaviour. A second approach would be to treat repeated variables internally as one, but we must ensure that the algorithm will still detect the errors and find a way to visualise them. After presenting the algorithm, we will explain in detail in Section 3.2.1 why we opt for the latter.



Operation	Parameter(s)
fill_mug	
drink	
insert_coin	

CoffeeR2	CoffeR1	CoffeeM
drink	drink	drink
fill_mug	fill_mug	fill_mug
insert_coin	insert_coin	
INITIALISATION	INITIALISATION	INITIALISATION
SETUP_CONTEXT		
(uninitialised state)		

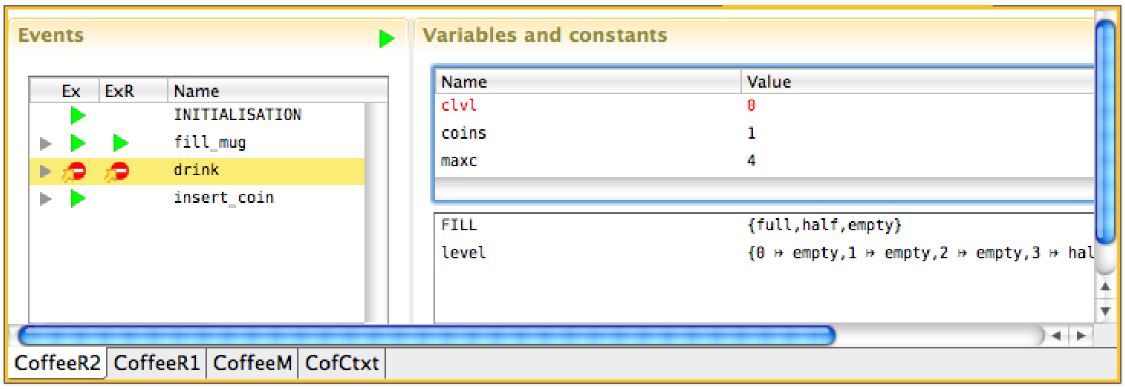
Name	Value	Previous value
CofCtxt		
level	l),(10↦full),(11↦full)}	,(10↦full),(11↦full)}
CoffeeM		
alvl	empty	full
CoffeR1		
coins	0	0
alvl	empty	full
CoffeeR2		
clvl	0	8
maxc	4	4
coins	0	0
Formulas		
invariants	T	T
axioms	T	T

invariant ok | no event errors detected

Figure 6: ProB Operations and State View after the Trace *insert_coin*, *fill_mug*, *drink*

2.6.2. *Example*

In the running example, the machine *CoffeeM* of Fig. 3a and its refinement *CoffeeR1* of Fig. A.19a have the variable *alvl* in common. Fig. 6 shows ProB animating the Coffee example. As can be seen in the newly developed hierarchical "State View", the variable *alvl* occurs twice, once in *CoffeeM* and once in *CoffeeR1*. We can also see that the variable *alvl* disappears in the refinement *CoffeeR2*. In Fig. 7 we show how the AnimB animator displays a state of multiple refinement-levels; each refinement level is given its own tab.



Ex	ExR	Name
		INITIALISATION
		fill_mug
		drink
		insert_coin

Name	Value
clvl	0
coins	1
maxc	4

FILL	{full,half,empty}
level	{0 ↦ empty,1 ↦ empty,2 ↦ empty,3 ↦ hal

Figure 7: AnimB View after *insert_coin*,*insert_coin*, *fill_mug*, *drink*

2.7. *Refined Events and Witnesses*

The predicate $W(t, v', u, v, w, w')$ denotes *witnesses*. Somewhat simplified, they link the abstract parameters t and the abstract variables v' to concrete parameters u and variables and w' (see also Fig. 9). Witnesses describe for each event separately how the refinement is achieved. Let $K(u, v, w, w') \mathrel{\widehat{=}} I(v) \wedge J(v, w) \wedge H(u, w) \wedge \boldsymbol{T}(u, w, w')$.

We discuss splitting of events in more detail and comment below on the differences when merging events in a refinement. The proof obligations for concrete events are called *guard strengthening*:

$$K(u, v, w, w') \wedge W(t, v', u, v, w, w') \Rightarrow G(t, v) \quad , \tag{2}$$

action simulation:

$$K(u, v, w, w') \wedge W(t, v', u, v, w, w') \Rightarrow \boldsymbol{S}(t, v, v') \quad , \tag{3}$$

and *invariant preservation*:

$$K(u, v, w, w') \wedge W(t, v', u, v, w, w') \Rightarrow J(v', w') \quad . \tag{4}$$

We have to prove *witness feasibility* in order to be able to add the witness predicate $W(t, v', u, v, w, w')$ to the premises in the proof obligations above:

$$K(u, v, w, w') \Rightarrow (\exists t, v' \cdot W(t, v', u, v, w, w')) \quad . \tag{5}$$

In general, witnesses would be required for all parameters p of an event but when a parameter is repeated in a refined event, by convention, it is assumed to be equal to the corresponding abstract parameter. If a parameter is not repeated an explicit witness is required. (The Rodin tool creates the *default witness* "*true*" if none is specified in the latter case. This witness does not constrain the relationship between abstract and concrete parameters and variables. Hence, usually default witnesses are not sufficient to establish the refinement relationship.) For variables the rule when witnesses are needed is more complicated: whenever a variable x that disappears occurs in a non-deterministic assignment in the abstract event, in the refined event a witness for the post-state variable v' is required.

Aside. As described in [9], in order to verify that $F(w)$ refines $E(v)$ we have to prove

$$I(v) \wedge J(v, w) \wedge H(u, w) \wedge \boldsymbol{T}(u, w, w') \Rightarrow \exists t, v' \cdot G(t, v) \wedge \boldsymbol{S}(t, v, v') \wedge J(v', w') \quad . \tag{6}$$

In a proof of this statement we prefer to instantiate the existentially quantified parameters t and variables v' by expressions that can in some way be inferred from the premises. This idea is generalised to the witnesses used in Event-B. Witnesses are predicates that provide values to satisfy the existentially quantified conclusion of the statement.

Refinement according to (6) is established by proof obligations (2) to (5):

$$\begin{array}{ll}
 & (2) \wedge (3) \wedge (4) \\
\Rightarrow & \quad \{ \text{ predicate logic } \} \\
 & K(u, v, w, w') \wedge W(t, u, v, v', w, w') \Rightarrow G(t, v) \wedge \boldsymbol{S}(t, v, v') \wedge J(v', w') \\
\Rightarrow & \quad \{ \text{ instantiate } t, v' := t, v' \text{ in goal } \} \\
 & K(u, v, w, w') \wedge W(t, u, v, v', w, w') \Rightarrow \exists t, v' \cdot G(t, v) \wedge \boldsymbol{S}(t, v, v') \wedge J(v', w') \\
\Rightarrow & \quad \{ \ t \text{ and } v' \text{ not free in } K(u, v, w, w') \ \} \\
 & K(u, v, w, w') \wedge (\exists t, v' \cdot W(t, u, v, v', w, w')) \Rightarrow \exists t, v' \cdot G(t, v) \wedge \boldsymbol{S}(t, v, v') \wedge J(v', w') \\
\Rightarrow & \quad \{ \text{ by (5) } \} \\
 & (6)
\end{array}$$

Aside. Two or more events can be merged in a refinement. In this case some restrictions apply for the abstract events being merged. Figure 8 shows a concrete event $F(w)$ that merges two abstract events $E_1(v)$ and $E_2(v)$. (It can be easily generalised to any finite number of merged events.) Merging of $E_1(v)$ and $E_2(v)$ is only defined if identically named parameters of t_1 and t_2 have identical types and the actions $S_1(t_1, v)$ and $S_2(t_2, v)$ are identical. Term $t_1 \cup t_2$

```
event $E_1(v)$
  any
    $t_1$
  when
    $G_1(t_1, v)$
  then
    $S_1(t_1, v)$
  end
```

```
event $E_2(v)$
  any
    $t_2$
  when
    $G_2(t_2, v)$
  then
    $S_2(t_2, v)$
  end
```

(a) Abstract Events $E_1(v)$ And $E_2(v)$

```
event $F(w)$ refines $E_1(v)$ $E_2(v)$
  any
    $u$
  when
    $H(u, w)$
  with
    $W(t_1 \cup t_2, v', u, v, w, w')$
  then
    $T(u, w)$
  end
```

(b) Concrete Event $F(w)$ Merging $E_1(v)$ And $E_2(v)$

Figure 8: Merging Two Events

denotes the “union” of the parameter lists t_1 and t_2. Due to the restrictions we have imposed on the abstract events, only the guard strengthening proof obligation needs to be adapted to the merging case (with $K(u, v, w, w')$ as defined above):

$$K(u, v, w, w') \;\wedge\; W(t_1 \cup t_2, v', u, v, w, w') \;\Rightarrow\; G_1(t_1, v) \;\vee\; G_2(t_2, v) \quad .$$

Merging does not pose new challenges for animation, hence, we do not discuss it when presenting the animation algorithm. The brief presentation of event merging given here should suffice as evidence for this claim (and as justification for the omission of event merging in Section 3).

2.7.1. Example

Event *fill_mug* in *CoffeeR2* contains the witness $x = level(clvl')$ for the abstract parameter x of *fill_mug* in *CoffeeR1*. This means intuitively, that every execution of *fill_mug* in *CoffeeR2* corresponds to an execution of *fill_mug* in *CoffeeR1* with parameter x set to $level(clvl')$. Event *fill_mug* in *CoffeeR1* must be enabled for $x = level(clvl')$ and the gluing invariant $alvl = level(clvl)$ must hold after executing the abstract and concrete event. Similarly, *drink* in *CoffeeR2* contains a witness $alvl' = level(clvl')$ for the abstract variable *alvl*. (Note, that it is just invariant @lvl in Fig. A.19b with all variables primed.)

2.7.2. Animation in ProB

Witnesses are the key concept that makes refinement animation possible. Indeed, (trace) refinement animation and refinement checking in classical B (see Section 5) require for every concrete state to keep track of the *set* of all abstract states for which the gluing invariant holds. Only if this set becomes empty, have we found an error in the refinement. The size of state space necessary for simulation grows exponentially. In Event-B, by contrast, the witnesses pinpoint the states which have to satisfy the refinement relationship (see Fig. 9).

2.8. New Events, Convergence, and Anticipation

In the course of refinement, often *new events* $F(w)$ are introduced into a model. New events must be proved to refine the implicit abstract event “*skip*” that does nothing and is always enabled, that is, we have to prove *invariant preservation*:

$$K(u, v, w, w') \;\Rightarrow\; J(v, w') \quad .$$

Moreover, it may be proved that new events do not collectively diverge. This is done by means of *convergence* and *anticipation*. Convergence is verified by proving that a specified *variant* $W(w)$ is bounded from below:

$$I(v) \;\wedge\; J(v, w) \;\wedge\; H(u, w) \;\Rightarrow\; W(w) \geq 0$$

and is decreased by each new event

$$I(v) \;\wedge\; J(v, w) \;\wedge\; H(u, w) \;\wedge\; \boldsymbol{T}(u, w, w') \;\Rightarrow\; W(w') < W(w) \qquad (7)$$

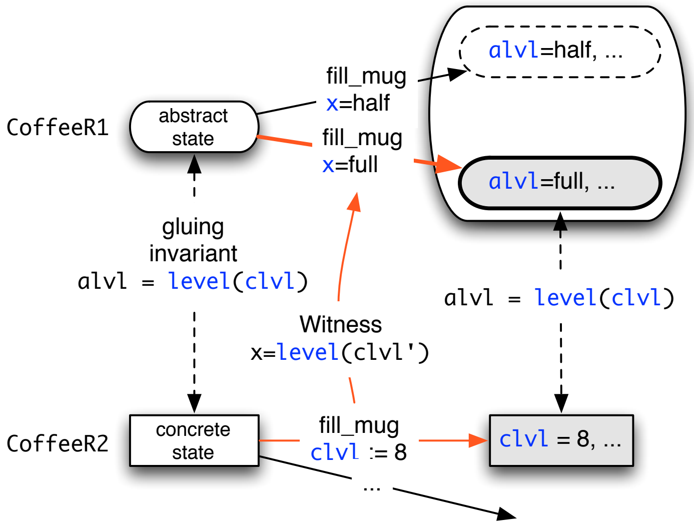


Figure 9: Witnesses and Multi-Level Animation

where we assume that the variant is an integer expression. (Instead of an integer expression also a finite set expression can be used.) We call events that satisfy these two proof obligations *convergent*.

An event is *anticipated* (to be convergent) if instead of (7) it satisfies the weaker condition (8):

$$I(v) \ \wedge \ J(v,w) \ \wedge \ H(u,w) \ \wedge \ \mathbf{T}(u,w,w') \ \Rightarrow \ W(w') \leq W(w) \quad . \tag{8}$$

Anticipated events can be used to prove convergence on a lexicographic order [12] or just to delay convergence proofs [13]. Anticipated events can be refined by anticipated or convergent events, but must ultimately be refined by a convergent event. Convergent events can only be refined by convergent events, that is, they stay convergent. Fig. 10 shows a typical scenario of the use of convergent and anticipated events.[5] “Ordinary” events (events that are neither

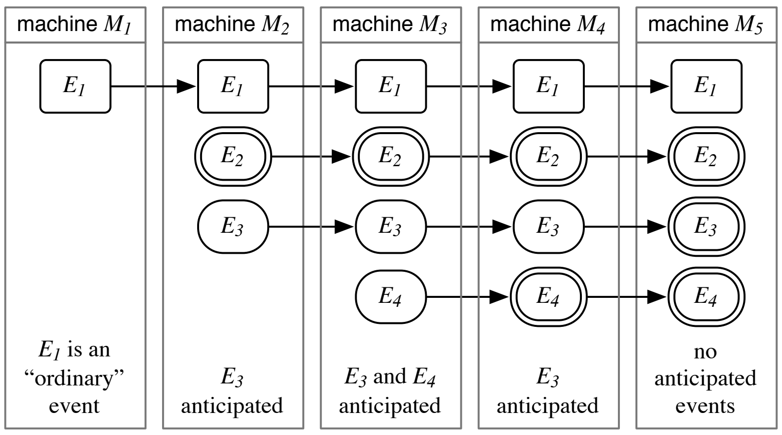


Figure 10: Convergent and Anticipated Events

[5]See also Fig. 5b.

convergent nor anticipated) are shown as boxes □, anticipated events as ellipses ◯, and convergent events as bold ellipses ◎. The new event E_2 in machine M_2 is convergent and stays convergent in machines M_3 to M_5. The new event E_3 in machine M_2 is anticipated and becomes convergent only in machine M_5. The new event E_4 in machine M_3 becomes convergent in machine M_4 "before" event E_3. The interest of anticipated events is mostly this possibility to carry out convergence proofs when it is most convenient, usually avoiding explicit specification of lexicographic variants.

Roughly speaking, if $V(v)$ is the variant of an anticipated event $E(v)$, and $W(w)$ a convergent event $F(w)$ that refines $E(v)$, then $F(w)$ is convergent with respect to the lexicographic variant $(V(v), W(w))$. For $V(v)$ we have verified that it is not increased by $E(v)$, that is, $V(v') \leq V(v)$. Because $F(w)$ refines $E(v)$ it also does not increase $V(v)$ (modulo the gluing invariant). In particular, if $F(w)$ leaves $V(v)$ unchanged, $V(v') = V(v)$, then $W(w)$ is decreased, hence, $V(v') < V(v) \lor (V(v') = V(v) \land W(w') < W(w))$.

2.8.1. Example

Event *insert_coin* in Fig A.19a is anticipated in *CoffeeR1* and is then proven convergent in *CoffeeR2* (Fig A.19b) by introducing an upper bound on the number of inserted coins.

2.9. Enabledness of Refined and New Events

The concrete machine N is relatively deadlock-free with respect to the abstract machine M if some concrete event is enabled whenever some abstract event is enabled. That is, the disjunction of the guards of all abstract events (without *INITIALISATION*) implies the disjunction of the guards of all concrete events (without *INITIALISATION*). This is equivalent to the claim that each abstract event (except for *INITIALISATION*) implies the disjunction of the guards of all concrete events (without *INITIALISATION*) . Whenever the abstract machine may continue by means of some event $E(v)$ with guard $G(t, v)$, the concrete machine may continue by means of at least one of the concrete events $F_1(w)$, $F_2(w)$, ..., $F_n(w)$ with parameter lists $u_1, u_2, \ldots, u_n$, and guards $H_1(u_1, w)$, $H_2(u_2, w)$, $\ldots$, $H_n(u_n, w)$,

$$I(v) \land J(v, w) \land G(t, v) \Rightarrow (\exists u \cdot H(u, w)) \lor (\exists u_1 \cdot H_1(u_1, w)) \lor \ldots \lor (\exists u_k \cdot H_k(u_k, w)) \quad .$$

Neither the Rodin proof obligations generation nor ProB model-checking or animation support the notion of relative deadlock-freedom at the moment. ProB animation would show a deadlock, because first the concrete events are animated and then their abstractions if a solution is found.

3. Description of the Multi-Level Animation Algorithm

In this section we describe the validation and animation algorithm in detail. We point out in the presentation of the algorithm how it indicates problems with particular proof obligations. We also show how feedback to the user needs to be considered. Producing informative output from an animation with good performance is a challenge. For this reason the algorithm makes heavy use of ProB's existing functionality. In particular, ProB provides methods to find values for variables that satisfy predicates occurring in Event-B models.

Below we limit the discussion to animation; but the algorithm is identical for model checking: the model checker uses the same technique to determine the state space.

3.1. Preprocessing

The algorithm is applied to a particular refinement machine M_i of a model. In a pre-processing step, all ancestor machines $M_0, \ldots, M_{i-1}$ of M_i are loaded and all contexts seen by $M_0, \ldots, M_i$ are merged by collecting the declared constants and joining the axioms. The invariant is obtained by conjoining all invariants of $M_0, \ldots, M_i$.

We transform each event of M_i to an internal representation. The representation is outlined on the right hand side of Fig. 11. Usually the list of abstract events contains just one entry. If an event refines *skip* or belongs to the most abstract machine M_0, the list of abstract events is empty. If the event refines several events, it will contain all of those events.

E.g., if we animate *CoffeeR1*, the event *drink* has one abstract event, the event *drink* of *CoffeeM*, which itself has an empty list of abstract events. *insert_coin* would have no abstract events because it refines *skip*.

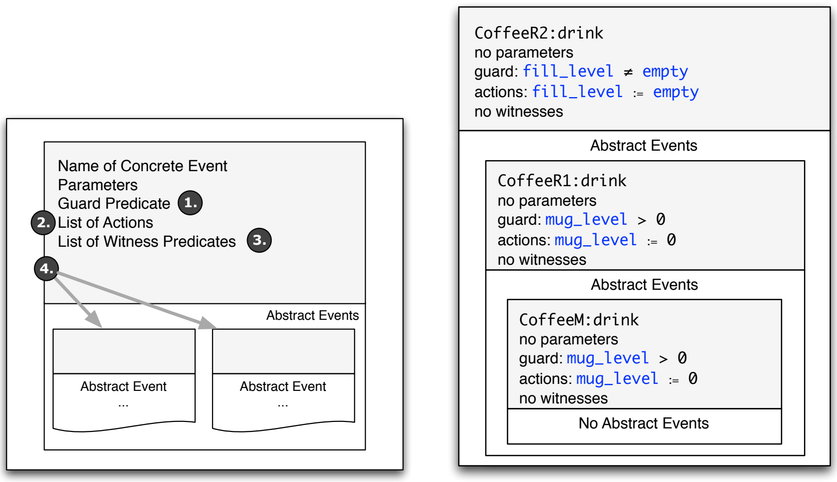


Figure 11: Illustration of the Algorithm And One Particular Event Structure

3.2. The Animation Algorithm

The animator executes events depending on the current state of a model. It maintains a state consisting of all constants of the seen contexts as well as all variables of the machines $M_0, \ldots, M_i$.

In a first step the animator tries to find values for the constants that satisfy all axioms. Subsequently, the animator executes in each step an event of the most concrete machine M_i; then it executes the corresponding abstract events from the concrete event to the most abstract event.

When all of this has been done, the animator is ready for the next step.

The algorithm to animate a particular event works as follows (item numbers correspond to those of Fig. 11, left hand side):[6]

1. Search for possible values for the parameters by evaluating its guard. If no values are found, the event is disabled.
2. Execute each action by evaluating the respective before-after predicate. If no solution is found, report an error. The possible reasons for a failing action are:
   a. The predicate P of an action $v\ :|\ P$ is not satisfiable or the set S of an action $v\ :\in\ S$ is empty. Both cases show violations of the event feasibility proof obligation.
   b. The new value v' of a variable v is constrained by a witness of a refined event (see step 3), but the abstract action cannot assign a corresponding value to v'. This indicates a violation of the action simulation proof obligation.
3. For each witness evaluate its predicate and try to find values for the witnessed variable. If no value is found for a witness, report an error, because a witness should have at least one solution (by the witness feasibility proof obligation).
4. a. If the list of abstract events is empty, a complete solution has been found for this event that leads to a new state. It consists of the values newly assigned by the actions plus the variables unchanged by the actions.
   b. If there are one or more abstract events, choose one non-deterministically and evaluate its guard like in step 1. If it evaluates to true, continue recursively with step 2, otherwise try the next event.
   If no guard evaluates to true, report an error, because the guard of the refinement is weaker than that of the abstract event (violation of the guard strengthening proof obligation).

All four steps can be non-deterministic and we generate all solutions (limited to a maximum number) with backtracking.

[6]With respect to animation *INITIALISATION* is not treated differently from any other events (see left of Fig. 11) except that it is enforced to occur once upon start of an animation.

3.2.1. Common Variables and Parameters

As mentioned above we have basically two options for dealing with common variables between a model and its refinement. One is to treat them as distinct variables x_a and x_c and to add the implicit gluing invariant $x_a = x_c$. If an abstract event $x := 1$ is now erroneously refined by $x := 2$, the animator can visualise the error by showing the post state $x_a = 1, x_c = 2$ and indicating that the invariant $x_a = x_c$ is violated.

But this approach has a fundamental problem as the following example shows. Let the action of the abstract event be $x :\in \{1, 2\}$ and the corresponding action of the refined event $x := 1$. Our algorithm would find two possible post states, $x_a = 1, x_c = 1$ and $x_a = 1, x_c = 2$. The first one is valid, the second one violates the implicit gluing invariant. Thus ProB would wrongly show an error for a correct refinement.

A solution to prevent the spurious error is to add implicit witnesses $x'_a = x'_c$ for all common variables. Then after assigning $x'_c = 1$, the evaluation of $x'_a = x'_c$ results in $x'_a = 1$. The abstract assignment $x :\in \{1, 2\}$ is then just a check $x'_a = 1 \in \{1, 2\}$. We still would find an error for an invalid concrete action (like $x := 3$), namely a violation of the action simulation proof obligation in step 2a. But adding the implicit witness has the effect that the algorithm never finds solutions with different abstract and concrete values. So there is no need to distinguish between them in the animation. The same applies to common parameters.

A slightly different solution would be to add implicit witnesses only for those variables which are assigned non-deterministically in the abstract event. This would allow us to generate a state which violates the gluing invariant that we could present the user in case of deterministic assignments. For non-deterministic assignments we still have to indicate a violation of the action simulation proof obligation in the event. It is not clear to us if the presentation of an invalid result state is easier to understand than the explanation of an error found in the event. So there is no clear benefit of this approach, but the downside would be a less consistent algorithm and a larger representation of a state by duplicating variables. Thus we have chosen the approach where we do not distinguish between common variables.

3.2.2. Example

Fig. 12 shows a detailed example run for the event *fill_mug* in the refinement *CoffeeR2* to demonstrate the animation algorithm. The table displays what is happening for each refinement level and step of the algorithm. The last column contains an entry if new values for variables have been found by evaluating the predicates.

We have chosen an arbitrary state where *fill_mug* is enabled as a starting point: $alvl = empty$, $clvl = 0$, $coins = 2$, $maxc = 4$. The example shown leads to a new state $alvl = full$, $clvl = 9$, $coins = 1$, $maxc = 4$. The variables *maxc* has not been modified by any action, so its value is just copied to the new state.

Note that the evaluation of action @ffl (*CoffeeR2*, step 2) chooses the value 9 of $clvl'$ non-deterministically, in total there are 4 different possibilities, leading to 4 different states with $clvl \in 8 \,..\, 11$.

Refinement	Step		found values
CoffeeR2	1.	evaluate the guards to see whether the event is enabled:	
		@gc2: $coins > 0 \equiv 2 > 0 \equiv \top$	
		@ml: $clvl \in level^{-1}[\{empty\}] \equiv 0 \in 0 \,..\, 2 \equiv \top$	
CoffeeR2	2.	action @delc2: $coins' = coins - 1 \equiv coins' = 2 - 1$	$coins' = 1$
		action @ffl: $clvl' \in level^{-1}[\{full\}] \equiv clvl' \in 8 \,..\, 11$	$clvl' = 9$
CoffeeR2	3.	witness for the abstract parameter x: $x = level(clvl') = level(9) = full$	$x = full$
CoffeeR2	4.	check the guard of the abstract event: @gc: $coins > 0 \equiv \top$	
CoffeeR1	2.	action @delc: $coins' = coins - 1 \equiv 1 = 2 - 1 \equiv \top$	
CoffeeR1	3.	(no witness to check)	
CoffeeR1	4.	check the guards of the abstract event:	
		@g0: $alvl = empty \equiv empty = empty \equiv \top$	
		@g1: $x \neq alvl \equiv full \neq empty \equiv \top$	
CoffeeM	2.	action @a1: $alvl' = x$	$alvl' = full$
CoffeeM	3.	(no witness to check)	
CoffeeM	4.	no more abstract events, stop	

Figure 12: Example Run of the Refinement Algorithm for Event *fill_mug*

3.2.3. *Animation of Convergent and Anticipated Events*

If we have successfully found a possible event leading from one state to another, we can easily check if the convergence criteria are satisfied. The principle is quite simple: for each convergent event of an animated model, we check if the variant V is decreased and non-negative by the predicates $V > V'$ and $V \geq 0$ resp. $V \supset V'$ when the variant is a set. If the event refines another convergent event, we omit the test: in the lexicographic order constructed by refinement, events that have been shown to decrease a variant in an abstraction of some concrete machine may increase the variant of that concrete machine. Similarly, we can check anticipated events (with $V \geq V'$ and $V \geq 0$ resp. $V \supseteq V'$), but we cannot omit the test if an event is a refinement of an anticipated event. Violations of specified variants are treated by ProB similarly to guard strengthening violations as shown in Fig. 13.

3.2.4. *Animating Only a Part of the Refinement Chain*

Above we presumed that the user wants to animate a refinement M_i and all its ancestors $M_0, \ldots, M_{i-1}$. But we also permit the user to limit the animation to the refinements between M_i and an "upper" refinement M_k with $0 \leq k \leq i$ instead of M_0. Then variables of not animated models and parts of predicates that contain references to those variables will be removed, resulting in weaker predicates.

Such a reduction of the animated refinement levels usually leads to smaller representation of states and allows the user to focus on the parts of the model he is currently interested in. However, the downside is that the limitation might hide some errors or introduce spurious errors:

- The gluing invariant can refer to abstract variables, weakening it means that some invariant violations are not detected anymore.
- Weakened witnesses can result in finding values for abstract parameters or variables that were not possible with the original predicates. Those values might cause other errors (e.g. violation of the simulation proof obligation).

If we limit the animation to a single refinement ($k = i$) the animation behaves like that of ProB without multi-level support.

4. Refinement-Validation with ProB

Refinement animation can be used to validate models. We present some specific problems that can be analysed by animation and discuss a selection of case studies to which it has been applied.

4.1. *Detection of Specific Problems*

Below we show on various modified versions of the coffee model (Fig. 3a) and its refinements (Fig. A.19), how the new multi-level animation algorithm allows ProB to detect a variety of refinement errors. Note that in contrast to AnimB and Brama, ProB is also a model checker that can systematically detect refinement errors.

4.1.1. *Guard Weakening*

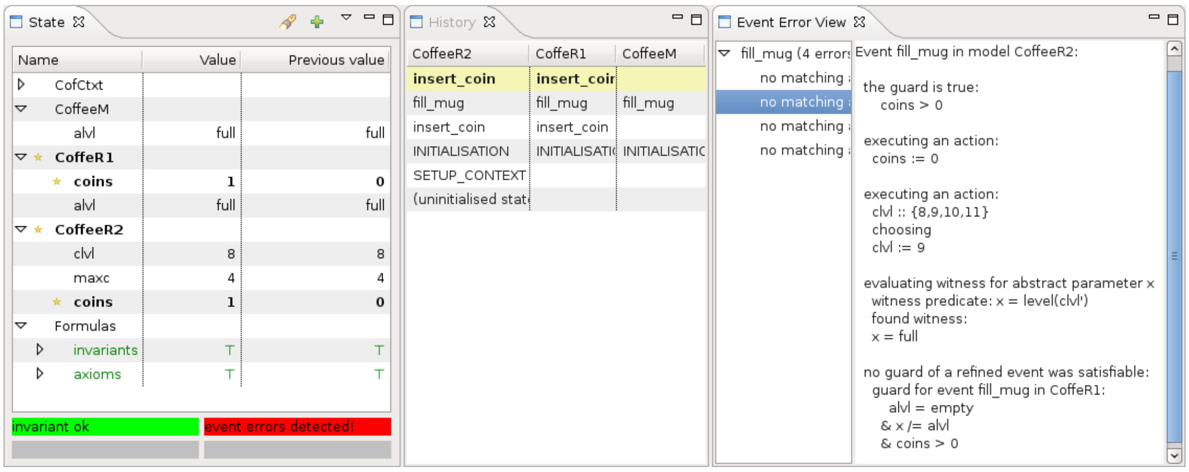
If we remove the guard @ml from the event *fill_mug* in *CoffeeR2*, we violate the guard strengthening proof obligation. As can be seen in Fig. 13, ProB's model checker using our new algorithm detects this problem straightaway (case 4a of our algorithm), leading us to a state where *fill_mug* is enabled in *CoffeeR2* but not in the abstract machines.

4.1.2. *Witness Disables Abstract Guard*

A similar error message appears if we keep the guards as they are, but inject an error in the witness. E.g, when using $x = empty$ as witness for *fill_mug*, ProB detects that there is a solution for the witness, but that the witness does not enable the abstract event.

4.1.3. *Witness Not Feasible*

Next, let us use the witness $x = level(clvl') \wedge x = empty$ for event *fill_mug*. Here case (3) of our algorithm detects an error for *fill_mug* (after executing *insert_coin*), and ProB displays the error message: "No solution found for witness of the abstract parameter x in event CoffeeR2:fill_mug". The animator AnimB does not detect this error (but it did detect the previous two errors).



State

Name	Value	Previous value
CofCtxt		
CoffeeM		
alvl	full	full
CoffeR1		
coins	1	0
alvl	full	full
CoffeeR2		
clvl	8	8
maxc	4	4
coins	1	0
Formulas		
invariants	⊤	⊤
axioms	⊤	⊤

invariant ok
event errors detected!

History

CoffeeR2	CoffeR1	CoffeeM
insert_coin	insert_coin	
fill_mug	fill_mug	fill_mug
insert_coin	insert_coin	
INITIALISATION	INITIALISATI	INITIALISATIC
SETUP_CONTEXT		
(uninitialised stat		

Event Error View

fill_mug (4 errors
no matching
no matching
no matching
no matching

Event fill_mug in model CoffeeR2:

the guard is true:
  coins > 0

executing an action:
  coins := 0

executing an action:
  clvl :: {8,9,10,11}
  choosing
  clvl := 9

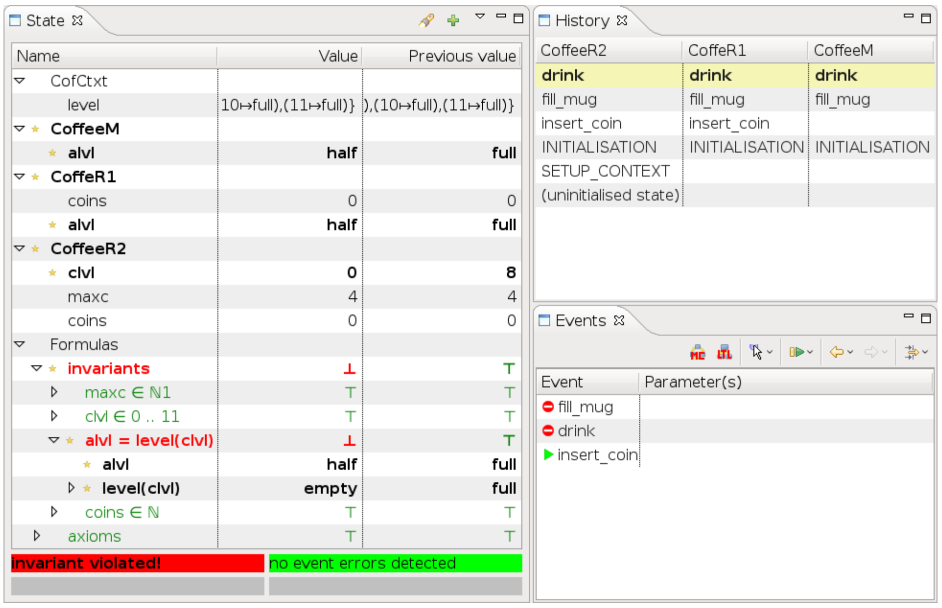
evaluating witness for abstract parameter x
  witness predicate: x = level(clvl')
  found witness:
  x = full

no guard of a refined event was satisfiable:
  guard for event fill_mug in CoffeR1:
    alvl = empty
    & x /= alvl
    & coins > 0

Figure 13: Violation of Guard Strengthening (ProB)

4.1.4. *Witness Violates Invariant*

Finally, we try to specify the witness $alvl' \in \{empty, half\} - \{alvl\}$ for the event drink, which does not guarantee that the abstract event will satisfy the gluing invariant. As can be seen in Fig. 14, ProB finds an invariant violation



State

Name	Value	Previous value
CofCtxt		
level	10↦full),(11↦full)}	),(10↦full),(11↦full)}
CoffeeM		
alvl	half	full
CoffeR1		
coins	0	0
alvl	half	full
CoffeeR2		
clvl	0	8
maxc	4	4
coins	0	0
Formulas		
invariants	⊥	⊤
maxc ∈ ℕ1	⊤	⊤
clvl ∈ 0 .. 11	⊤	⊤
alvl = level(clvl)	⊥	⊤
alvl	half	full
level(clvl)	empty	full
coins ∈ ℕ	⊤	⊤
axioms	⊤	⊤

invariant violated!
no event errors detected

History

CoffeeR2	CoffeR1	CoffeeM
drink	drink	drink
fill_mug	fill_mug	fill_mug
insert_coin	insert_coin	
INITIALISATION	INITIALISATION	INITIALISATION
SETUP_CONTEXT		
(uninitialised state)		

Events

Event	Parameter(s)
fill_mug	
drink	
insert_coin	

Figure 14: Violation of Gluing Invariant (ProB)

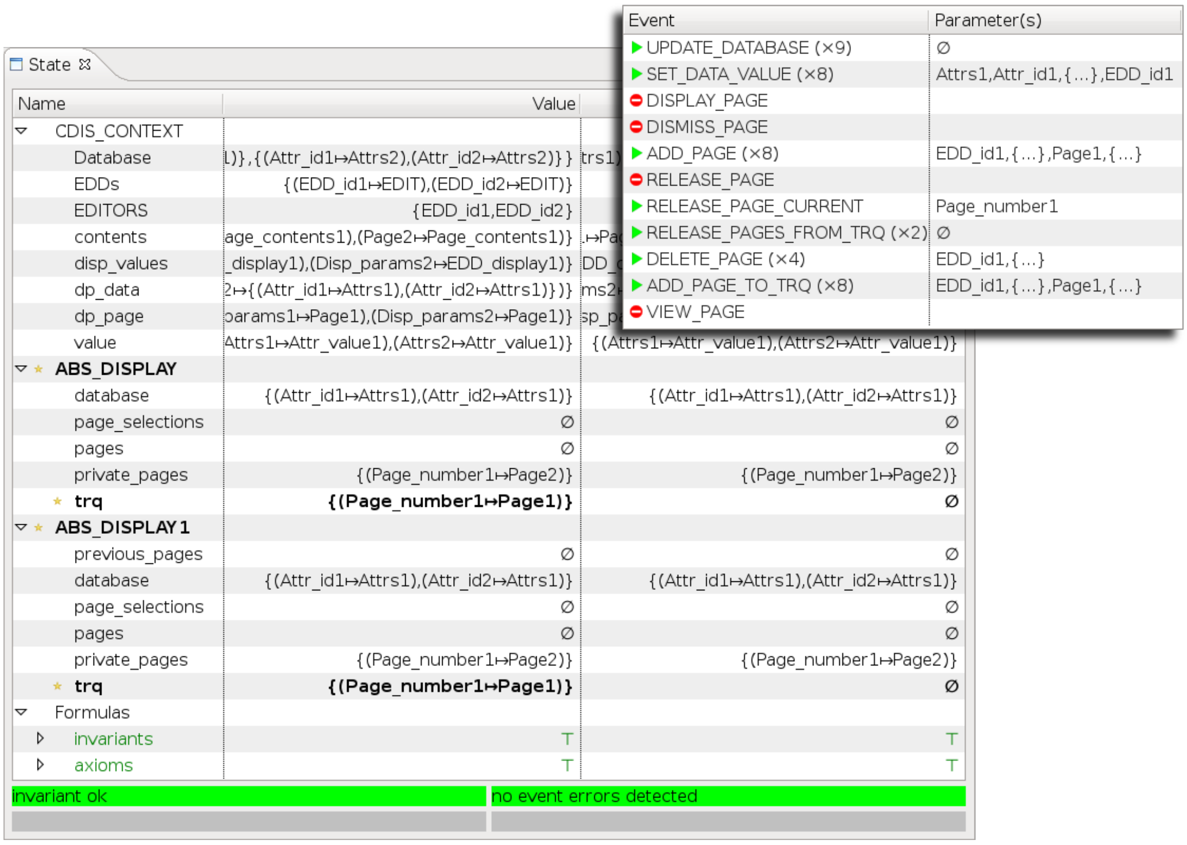
error ($alvl = level(clvl)$ is false) directly after the drink event.

Note that, AnimB detects an error in the model, but only later when trying to execute the *fill_mug* event after the erroneous drink event.

In practice, validation by animation complements the proof-based methodology of core Event-B. Corresponding methodological benefits of using animation of Event-B models are discussed in more detail in [14].

4.2. Application to Case Studies

We have successfully applied the new multi-level animation of ProB on a two-level model of SAP service choreographies [15]. We have also tested the tool on the CDIS air traffic control case study carried out in the EU project Rodin. Figure 15 contains a screenshot of the first two levels; we have successfully animated all 7 levels of the full model concurrently.


State

Name	Value	
CDIS_CONTEXT		
Database	l)},{(Attr_id1↦Attrs2),(Attr_id2↦Attrs2)}}	trs1)
EDDs	{(EDD_id1↦EDIT),(EDD_id2↦EDIT)}	
EDITORS	{EDD_id1,EDD_id2}	
contents	age_contents1),(Page2↦Page_contents1)}	.↦Pa
disp_values	_display1),(Disp_params2↦EDD_display1)}	DD_
dp_data	2↦{(Attr_id1↦Attrs1),(Attr_id2↦Attrs1)})}	ms2
dp_page	oarams1↦Page1),(Disp_params2↦Page1)}	sp_p
value	Attrs1↦Attr_value1),(Attrs2↦Attr_value1)}	{(Attrs1↦Attr_value1),(Attrs2↦Attr_value1)}
ABS_DISPLAY		
database	{(Attr_id1↦Attrs1),(Attr_id2↦Attrs1)}	{(Attr_id1↦Attrs1),(Attr_id2↦Attrs1)}
page_selections	∅	∅
pages	∅	∅
private_pages	{(Page_number1↦Page2)}	{(Page_number1↦Page2)}
trq	**{(Page_number1↦Page1)}**	**∅**
ABS_DISPLAY1		
previous_pages	∅	∅
database	{(Attr_id1↦Attrs1),(Attr_id2↦Attrs1)}	{(Attr_id1↦Attrs1),(Attr_id2↦Attrs1)}
page_selections	∅	∅
pages	∅	∅
private_pages	{(Page_number1↦Page2)}	{(Page_number1↦Page2)}
trq	**{(Page_number1↦Page1)}**	**∅**
Formulas		
invariants	⊤	⊤
axioms	⊤	⊤

invariant ok

no event errors detected

Event	Parameter(s)
▶ UPDATE_DATABASE (×9)	∅
▶ SET_DATA_VALUE (×8)	Attrs1,Attr_id1,{...},EDD_id1
⛔ DISPLAY_PAGE	
⛔ DISMISS_PAGE	
▶ ADD_PAGE (×8)	EDD_id1,{...},Page1,{...}
⛔ RELEASE_PAGE	
▶ RELEASE_PAGE_CURRENT	Page_number1
▶ RELEASE_PAGES_FROM_TRQ (×2)	∅
▶ DELETE_PAGE (×4)	EDD_id1,{...}
▶ ADD_PAGE_TO_TRQ (×8)	EDD_id1,{...},Page1,{...}
⛔ VIEW_PAGE	

Figure 15: Animating the Rodin CDIS Case Study

Another case study was a complete development of the quicksort algorithm in Event-B, consisting of ten machines and two contexts. We have successfully animated and model-checked all the ten levels concurrently. Animation showed how the algorithm "works" at different abstraction levels. This is valuable for explaining an otherwise static model of an algorithm.

We have also successfully animated concurrently 14 levels of an elevator model solution by ETH Zürich. This has uncovered a potential problem in the model, namely that starting at a certain refinement level, the lift is no longer able to move (but the doors can be opened and closed and the buttons can be pressed; so there is no deadlock in the conventional sense).

5. Comparison with Trace Refinement and Empirical Evaluation

5.1. Comparing with Trace Refinement Checking

An important related work is the refinement checking algorithm in [16]. This algorithm checks trace refinement between the abstract model and the refined model, i.e., it checks that every trace of events of the refined model is also a valid trace of the abstract model.

For example, for our running *Coffee* example, all the possible traces of the abstract model can be deduced from the state space in Figure 3. If the refined model can perform the trace *INITIALISATION*, *drink*, then the algorithm from [16] would detect an error, as this trace cannot be performed by the abstract model *CoffeeM*.

The main differences between our new multi-level validation algorithm and [16] can be summarised as follows:

- As the algorithm from [16] works on the level of traces, it can be applied to other formalisms, such as CSP [17]. It can even be used to check refinement relations between CSP and classical B [18] models.
- The tools for classical B and Event-B generate proof obligations for forward refinement, which is not complete. I.e., there are systems which are related by trace refinement but where no forward refinement can be established [19]. The new algorithm in this paper adheres to the forward refinement methodology of Event-B, whereas [16] checks trace refinement.
- The algorithm of [16] checks the refinement relation between two models; our new algorithm checks an entire refinement chain in one go (but restricted by the state space of the most concrete model).
- The algorithm of [16] computes the state space of the refined model on-the-fly, but it has to compute the entire state space of the abstract model *before* starting the refinement checking (for the running *Coffee* example, this corresponds to Figure 3). The new algorithm in this paper performs the entire validation on-the-fly. In particular, only the relevant parts of the state space of the abstract model(s) are computed. We will illustrate this in the empirical results below.
- The algorithm of [16] performs the refinement checking depth-first; the new algorithm can use any of ProB's search methods (depth-first, breadth-first and the default mixed depth-first/breadth-first search). Especially for large state spaces, this increases the likelihood of detecting errors; see [20].
- The algorithm of [16] computes the state space of the abstract model independently of the refined model and thus does not check the gluing invariant. Furthermore, in its current form, it cannot make use of Event-B's witnesses and hence has to keep track of *sets* of states in the abstract model. Our algorithm makes use of the witnesses, which often allows us to only relate a single abstract state to a refined state. It also allows us to relate parameters of the refined event to parameters of the abstract event. E.g., in our running example, *fill_mug* has no parameter in *CoffeeR2*, while it has the parameter x in *CoffeeR1* and *CoffeeM*; the witness $x = level(clvl')$ in *CoffeeR2* tells us which parameter value to use in the abstract models. The algorithm of [16] was devised in a setting of CSP and classical B, where parameters of events or operations cannot be refined. E.g., in the running example it would report a counter-example (*INITIALISATION*, *fill_mug*) for *CoffeeR2*, unless we tell it to ignore all parameters in traces.

To evaluate the practical performance difference between the two algorithms, we have adapted the scheduler example from [16] (and originally from [21]) for Event-B. The abstract model *Scheduler0* schedules a certain number of processes (which we have fixed to 6 in the experiments), and keeps them in the sets *waiting*, *ready* and *active*. At most one process can be active at any one time. In the refinement *Scheduler1* there is a queue for the ready processes (the abstract model just keeps them in a set). We have also generated two incorrect refinements: in *Scheduler1_err* the event *swap_ready* is not shifting the indexes in the queue and in *Scheduler1_terr* there is a missing guard in the event *new*.

The experimental results are summarised in Table 1. The times are in milliseconds, and do not include the time to load the models. The experiments were run on a MacBook Pro with a 3.06 GHz Core2Duo processor, and ProB 1.3.2 compiled with SICStus Prolog 4.1.2. In essence, we can draw the following conclusions.

Scheduler1: there is no big performance difference here between the multilevel validation and the trace checking. However, the multilevel validation also checks the gluing invariant. Furthermore, symmetry reduction [22, 23] can be applied; which in this case dramatically reduces the model checking time by a factor of 38.6. We explain why symmetry can be more easily exploited with our multi-level algorithm further below in subsection 5.3.

Scheduler1_err: Here we see that the trace refinement checking does not catch the error introduced in the model, as the error only manifests itself in the violation of the gluing invariant. In this case, this leads to a big difference in performance, as both the abstract and refined state space are fully explored by the trace refinement check.

	Benchmark	Multi-level	Multi-level with symmetry [22]	Trace-Refinement [16]
Consistency Checking	*Scheduler0*	1930 ms	50 ms	1930 ms
Refinement Checking	*Scheduler1*	8760 ms	70 ms	1930 ms + 7260 ms
	Scheduler1_err	360 ms	30 ms	1930 ms + *16090 ms
	Scheduler1_terr	10 ms	20 ms	1930 ms + 60 ms

*: no error found, as gluing invariant not checked

Table 1: Comparing Trace-Refinement Checking and Multi-Level Validation

Scheduler1_terr: Here, the trace refinement checking does catch the error, as this time it does result in a trace of the refined model which cannot be mimicked by the abstract model. Note, however, that there is a big performance difference, as the algorithm from [16] has to explore the full state space of the abstract model (even though only a small part of it was needed to find the counterexample).

5.2. Application to other Formalisms

In general we think the algorithm can be adapted to formalisms which support a concept of refinement similar to that of Event-B. In the following we have a closer look at classical B and ASM which have many concepts in common with Event-B.

5.2.1. Classical B

An obvious candidate for adapting the presented algorithm is classical B [18]. Like Event-B, classical B has machines with variables and invariants on them. Operations can change the values of variables by *generalised substitutions*. Unlike events in Event-B operations can have preconditions.

Lets assume an abstract B machine with invariant I and an operation with precondition P_A and substitution S_A. It's refinement has a gluing invariant J and an operation with precondition P_R and substitution S_R. Let $[S]P$ denote the weakest precondition that ensures that an execution of S results in a state satisfying P. A refinement must fulfil two conditions:

- Starting from a state where I, J and P_A hold, for each execution of S_R there must exists a corresponding execution of S_A such that J holds in the resulting state. Formally this is expressed by $I \wedge J \wedge P_A \Rightarrow [S_R]\neg[S_A]\neg J$.
- The precondition of the refined operation must be weaker than the abstract operation's precondition, formally $I \wedge J \wedge P_A \Rightarrow P_R$.

Whereas the presented algorithm follows a strict bottom-up (from refinement to abstract machine) approach, animating classical B could probably require a combination of a top-down and bottom-up approach: First we evaluate P_A on a state, then we check if P_R also holds. Then we can execute S_R to find the concrete values of a new state. Afterwards we check if there is a corresponding abstract substitution that fulfils J.

In future work we will further investigate this approach. There are still several outstanding technical issues (e.g., how to give the user helpful feedback in case of errors) as well as fundamental issues (e.g., have we thought of all possible situations regarding preconditions?). We expect that witnesses or a similar concept will have to be introduced in classical B as well. Preconditions could be handled similarly to guards, except for the order in which preconditions have to be evaluated, as indicated above.

5.2.2. ASM

ASM provides a very generic framework for refinement [24]. Unlike Event-B, it allows the refinement of an arbitrary number of abstract operations by an arbitrary number of concrete operations. Thus we do not see how we could apply this algorithm to that liberal form of refinement.

Of course, using a restricted form of ASM refinement could make our approach applicable with only small modifications. The only difference would be that ASM rules can be enabled simultaneously whereas Event-B events cannot.

5.3. Symmetry Reduction for Multi-Level Validation

The reason symmetry reduction cannot be (easily) applied when using the refinement checking algorithm from [16], is that the abstract and refined model are explored separately. Hence, symmetry reduction would have to be applied to these two steps separately as well. A big problem now is that the canonical form chosen for the abstract model may be incompatible with the one chosen for the refined model, leading to erroneous counterexamples and/or to real counterexamples not being found.

Let us look at the example in Figure 16. The abstract model *ProcSet* contains a variable x, which stores process identifiers taken from the carrier set *Proc*, declared in the context *PCtxt*. The event *new* can be used to add elements to x, and the event *del* to remove elements from x. In the refined model *ProcSeq* the process identifiers are stored in a sequence q. The model also stores the size of the sequence as a separate variable. Observe that the elements of *Proc* can be interchanged, leading to symmetries in the state space. Notably, all successor states of the initial state $x = \varnothing$ are symmetric in the abstract model. Similarly, all successor states of the initial state $q = \varnothing, n = 0$ are symmetric in the refined model. Let us assume that we have two process identifiers: $Proc = \{P1, P2\}$. The top of Figure 17 contains the initial state and the successor states for the abstract and refined model. The symmetry reduction algorithm from [23] will now chose one representative per equivalence class. Say the symmetry reduction chooses the successor of $new(P1)$ for the canonical representative for the abstract model, but for some reason (e.g., due to the other data structure used) chooses the successor of $new(P2)$ in the refined model. As we can see in the bottom of Fig. 17, in this case $new(P1), del(P2)$ or $new(P2), del(P2)$ are traces of the symmetry reduced refined model, which cannot be mimicked by the symmetry reduced abstract model.[7] Hence, the trace refinement checking algorithm would erroneously report that refinement fails. In multilevel animation, the canonical form is computed on the combined state of the abstract and refined model, and as such respects the relationship between them. This is illustrated in Fig. 18, where one representative is chosen for the combined abstract and refined state.

```
context $PCtxt$
sets $Proc$
end
```

```
machine $ProcSet$
sees $PCtxt$
variables $x$
invariants

  @xproc $x \subseteq Proc$

events
  event $INITIALISATION$
    begin
      @i $x := \varnothing$
    end
  event $new$
    any $p$ when
      @newp $p \notin x$
    then
      @incx $x := x \cup \{p\}$
    end
  event $del$
    any $p$ when
      @pinx $p \in x$
    then
      @subx $x := x \setminus \{p\}$
    end
end
```

```
machine $ProcSeq$ refines $ProcSet$
sees $PCtxt$
variables $q$ $n$
invariants
  @nnat $n \in \mathbb{N}$
  @qinj $q \in 1 \,..\, n \rightarrowtail Proc$
  @glue $\mathrm{ran}(q) = x$
events
  event $INITIALISATION$
    begin
      @iq $q, n := \varnothing, 0$
    end
  event $new$ refines $new$
    any $p$ when
      @newp $p \notin \mathrm{ran}(q)$
    then
      @incq $q, n := q \mathbin{<\!\!\!+} \{n{+}1 \mapsto p\}, n{+}1$
    end
  event $del$ refines $del$
    any $p$ when
      @pinq $q[\{n\}] = \{p\}$
    then
      @subq $q, n := n \mathbin{\lhd\!\!-} q, n{-}1$
    end
end
```

Figure 16: A Model Symmetric in the Set of Processes *Proc*

[7]The permutation flooding technique [25] could probably be made to work more easily, as it explicitly adds "permutation" actions to the state space.

Cardinality of Proc	States	Validation Time	States	Validation Time
	(without symmetry)		(with symmetry)	
1	3	0.001 s	3	0.002 s
2	6	0.004 s	4	0.003 s
3	17	0.012 s	5	0.006 s
4	66	0.049 s	6	0.009 s
5	327	0.278 s	7	0.013 s
6	1,958	1.618 s	8	0.018 s
7	13,701	11.325 s	9	0.025 s
8	100,001	90.930 s	10	0.033 s

Table 2: Experimental Results for Multi-level Validation with Symmetry Reduction

In Table 2 we show how effective symmetry reduction can be for multi-level validation. We have run exhaustive multi-level model checking for the refined *ProcSeq* model of Fig. 16 with and without symmetry reduction. We have used the symmetry reduction technique from [23].

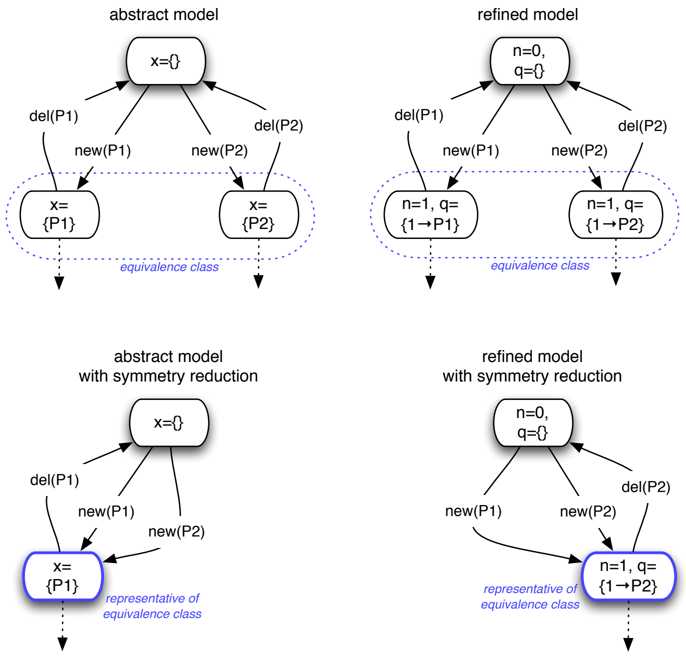


Figure 17: Illustration of Symmetry Reduction for Trace Refinement Checking

5.4. *More Experimental Results*

The experiments below were again run on a MacBook Pro with a 3.06 GHz Core2Duo processor, and ProB 1.3.2 compiled with SICStus Prolog 4.1.2.

To evaluate the cost of treating multiple levels at once we have run our algorithm on an Event-B model for an elevator by ETH Zürich with an abstract machine and 13 refinements. We have measured the time to check 100 states for each of the 14 machines. The results are summarised in Table 3a. Note that the state spaces for levels 0 to 4 have less than 100 nodes each, hence the full state space was explored in the given time. Starting at level 5, the state space has more than 100 states; we have stopped the algorithm at 100 nodes to be able to compare the time required to treat the same number of nodes. Also note that, we have always animated all levels above the given level (i.e.,

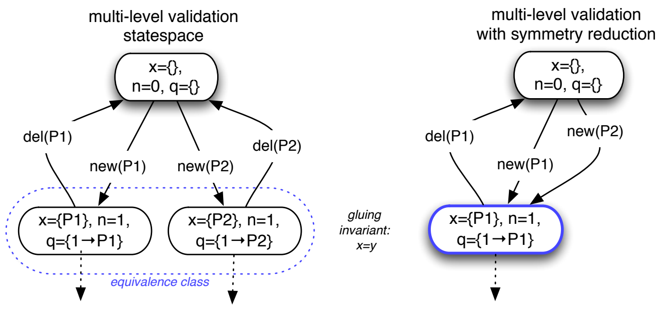


Figure 18: Illustration of Symmetry Reduction for Multi-Level Validation

Level	Validation Time (100 states)	Cost for extra level	Time per level
0	0 ms		
1	0 ms		
2	20 ms		
3	30 ms		
4	80 ms		
5	100 ms		
6	130 ms	30.0%	18.57
7	160 ms	23.1%	20.00
8	190 ms	18.8%	21.11
9	840 ms	342.1%	84.00
10	1110 ms	32.1%	100.91
11	1550 ms	39.6%	129.17
12	1790 ms	15.5%	137.69
13	2160 ms	20.7%	154.29

(a) ETH Elevator (Levels 0–4 have less than 100 states)

Level	Validation Time (complete)	Nodes	Cost for extra level	Time per node and level
0	300 ms	34		8.82
1	60 ms	18	-80.0%	1.67
2	110 ms	18	83.3%	2.04
3	170 ms	44	54.5%	0.97
4	190 ms	65	11.8%	0.58
5	260 ms	65	36.8%	0.67
6	310 ms	66	19.2%	0.67
7	360 ms	66	16.1%	0.68
8	410 ms	66	13.9%	0.69
9	460 ms	66	12.2%	0.70

(b) Quicksort Model

Table 3: Experimental Results for Multi-level Validation

when validating the level 13 model, we have also validated levels 0–12). As we can see, there is no simple linear dependence between the validation time and the number of levels animated. However, up to level 8 the runtime seems to increase linearly with the number of levels animated. Then there is a big jump at level 9 (which introduces 6 new variables of type total function), after which the runtime seems to increase again linearly (albeit with a steeper slope).

We have conducted a similar experiment for the Quicksort model from [26], for a particular array to be sorted. The results are summarised in Table 3b. Here, we can see that between level 0 and 1 the number of nodes and the validation time actually decreases: the refined model is more deterministic and has, hence, a reduced state space. The non-deterministic choice of a sorting permutation in the abstraction is implemented by a deterministic algorithm computing such a permutation. The validation time per node and level actually goes down until level 4, and then remains relatively stable from level 5 onwards.

We have also applied our algorithm on industrial models from SAP, Bosch and Siemens within the Deploy project. In all instances, the multi-level algorithm dealt very well with large number of refinement levels.

6. More Related Work

As already indicated above, the tools Brama and AnimB are also capable of performing multi-level animation of Event-B models, and have partially inspired this work. Unfortunately there is little scientific or technical documentation available for either of these tools. A few notable differences are

- Both Brama and AnimB require the user to specify explicitly values for constants; that is, we had to "calculate" the Cartesian products for the level constant in Fig. 1 by hand.
- ProB can be driven by a model checker to systematically search for errors, and to validate LTL formulas.
- ProB uses a constraint-solving approach to find solutions for predicates, while AnimB and Brama seem to rely on pure enumeration. As such, ProB can evaluate much more complicated guards and predicates than AnimB or Brama.

Another animator for Event-B is [27]; but it does not yet seem to support refinement animation. The same is true for the animator in [28] for classical B.

Animation was already considered valuable in the tools supporting the B Method, Atelier B [29] and the B-Toolkit [30]. Both contain support for animating formal models. However, the support is very rudimentary. It relies on the user to supply parameter values to carry out a step of the simulation engine. If the supplied values do not satisfy the guard, this is shown to the user but the simulation continues.

Abstract State Machines [31] have traditionally an operational conception of modelling. The tool CoreASM [32] support animation of Abstract State Machines based on an executable core of the notation. Refinement [24] is not supported yet. The refinement notion of Abstract State Machines is very general. Thus, it may be difficult to achieve efficient animation without introducing restrictions on permissible refinements. We also believe that the concept of witnesses could be a key for Abstract State Machine refinement animation, too.

Tools for VDM [33] have also provided animation features. VDMTools [34] provides animation but not of refinements. An early attempt on animation of VDM models has been made in [35]. However, the approach relies on the user to compare simulation traces of abstract and concrete models. We think automation of this is indispensable, because relying on the user for trace inspection the approach appears error-prone.

We have not considered animation of formalisms based on automata or Petri nets here because the challenges in their animation are quite different from those we face in Event-B, a state-based formalism with forward refinement at its core, where the model is described in terms of first-order predicate formulas.

7. Conclusion

We have presented a description of refinement in Event-B and have shown how a suitable animation and validation algorithm can be developed. The key ingredient that makes the algorithm tractable are the witnesses of Event-B. We have implemented the algorithm within ProB, and have shown how a variety of refinement errors can now be detected effectively. We have applied the technique to various case studies, and have animated up to 14 levels simultaneously. The algorithm presented in the article also achieves further performance improvements over previous work based on trace refinement, for instance, by applying symmetry reduction to multiple levels of refinement at once. It would be interesting to see how the new ideas apply to classical B, in particular. We leave this for future work.

We consider proof and animation complementary techniques of validation. Hence, animation does not need to provide all capabilities that formal proof provides. For instance, for finding candidates of invariants formal proof appears superior; for checking whether they are always satisfied, i.e., finding counter examples, animation appears superior. Moreover, animation can be used to reason about properties that have not (yet) been formalised.

We would like to incorporate some improvements into the animator. We would like to be able to better visualise relative deadlocks, i.e., states where all concrete events are disabled but some abstract event is enabled. For now, animation shows in case of a relative deadlock that all concrete events are disabled and does not analyse the abstract events further. Moreover, Event-B types and definitions would be needed that support animation better. For instance, transitive closures are defined as sets of all transitive closures of all relations of a certain type. For animation, it would be sufficient to compute only the closures of the relations actually appearing in a model. We would like to link the animator closer to the other components of the Rodin tool. We would also like to indicate certain errors found in Event-B models by ProB in associated proof obligations, for instance, so that the violated proof obligations can be marked as "not provable."

Acknowledgement. We are grateful to Thierry Lecomte and Christophe Métayer for their help with using Brama and AnimB and their help with technical questions concerning the two tools. We also want to thank the anonymous reviewers for their helpful comments. Part of this research has been EU funded FP7 project 214158: DEPLOY (Industrial deployment of advanced system engineering methods for high productivity and dependability).

References

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

[2] J.-R. Abrial, M. Butler, S. Hallerstede, An open extensible tool environment for Event-B., in: ICFEM06, LNCS 4260, Springer, 2006, pp. 588–605.

[3] J. Bendisposto, M. Leuschel, O. Ligot, M. Samia, La validation de modèles Event-B avec le plug-in ProB pour RODIN, Technique et Science Informatiques 27 (2008) 1065–1084.

[4] M. Leuschel, M. Butler, ProB: A model checker for B, in: K. Araki, S. Gnesi, D. Mandrioli (Eds.), FME 2003: Formal Methods, LNCS 2805, Springer-Verlag, 2003, pp. 855–874.

[5] M. Leuschel, M. J. Butler, ProB: an automated analysis toolset for the B method, STTT 10 (2008) 185–203.

[6] T. Servat, Brama: A new graphic animation tool for B models, in: J. Julliand, O. Kouchnarenko (Eds.), Proceedings B 2007, LNCS 4355, Springer, 2006, pp. 274–276.

[7] C. Métayer, http://www.animb.org/index.xml, 2010. AnimB Homepage.

[8] J. Bendisposto, F. Fritz, M. Leuschel, Developing Camille, a text editor for Rodin, in: Workshop on Tool Building in Formal Methods.

[9] J.-R. Abrial, S. Hallerstede, Refinement, decomposition, and instantiation of discrete models: Application to event-B, Fundam. Inform 77 (2007) 1–28.

[10] R.-J. Back, Refinement Calculus II: Parallel and Reactive Programs, in: J. W. deBakker, W. P. deRoever, G. Rozenberg (Eds.), Stepwise Refinement of Distributed Systems, volume 430 of *Lecture Notes in Computer Science*, Springer, 1989, pp. 67–93.

[11] S. Hallerstede, M. Leuschel, D. Plagge, Refinement-Animation for Event-B – Towards a Method of Validation, in: M. Frappier, U. Glässer, S. Khurshid, R. Laleau, S. Reeves (Eds.), ABZ, volume 5977 of *Lecture Notes in Computer Science*, Springer, 2010, pp. 287–301.

[12] J.-R. Abrial, D. Cansell, D. Méry, Refinement and Reachability in EventB, in: H. Treharne, S. King, M. Henson, S. Schneider (Eds.), ZB 2005, volume 3455 of *LNCS*, pp. 222–241.

[13] S. Hallerstede, A (Small) Improvement of Event-B?, in: Proceedings of Dagstuhl Seminar on Refinement Based Methods for the Construction of Dependable Systems (09381), Dagstuhl, pp. 48–52.

[14] S. Hallerstede, M. Leuschel, How to explain mistakes, in: J. Gibbons, J. N. Oliveira (Eds.), TFM 2009, LNCS 5846, Springer, 2009, pp. 105–124.

[15] S. Wieczorek, V. Kozyura, A. Roth, M. Leuschel, J. Bendisposto, D. Plagge, I. Schieferdecker, Applying Model Checking to Generate Model-based Integration Tests from Choreography Models, in: Proceedings TESTCOM/FATES 2009, Springer-Verlag, 2009, p. to appear.

[16] M. Leuschel, M. Butler, Automatic refinement checking for B, in: K.-K. Lau, R. Banach (Eds.), Proceedings ICFEM'05, LNCS 3785, Springer-Verlag, 2005, pp. 345–359.

[17] C. Hoare, Communicating Sequential Processes, Prentice Hall, 1985.

[18] J.-R. Abrial, The B-Book, Cambridge University Press, 1996.

[19] J. He, C. A. R. Hoare, J. W. Sanders, Data refinement refined., in: B. Robinet, R. Wilhelm (Eds.), ESOP 86, European Symposium on Programming, Saarbrücken, Federal Republic of Germany, March 17-19, 1986, Proceedings, volume 213 of *Lecture Notes in Computer Science*, Springer, 1986, pp. 187–196.

[20] M. Leuschel, The high road to formal validation, in: E. Börger, M. Butler, J. P. Bowen, P. Boca (Eds.), ABZ, volume 5238 of *Lecture Notes in Computer Science*, Springer, 2008, pp. 4–23.

[21] B. Legeard, F. Peureux, M. Utting, Automated boundary testing from Z and B, in: Proceedings FME'02, LNCS 2391, Springer-Verlag, 2002, pp. 21–40.

[22] M. Leuschel, T. Massart, Efficient approximate verification of B via symmetry markers, in: Proceedings International Symmetry Conference, Edinburgh, UK, pp. 71–85.

[23] C. Spermann, M. Leuschel, ProB gets nauty: Effective symmetry reduction for B and Z models, in: Proceedings Symposium TASE 2008, IEEE, Nanjing, China, 2008, pp. 15–22.

[24] E. Börger, The ASM refinement method, Formal Aspects of Computing 15 (2003) 237–257.

[25] M. Leuschel, M. Butler, C. Spermann, E. Turner, Symmetry reduction for B by permutation flooding, in: Proceedings B2007, LNCS 4355, Springer-Verlag, Besancon, France, 2007, pp. 79–93.

[26] S. Hallerstede, Proving Quicksort Correct in Event-B, Electr. Notes Theor. Comput. Sci 259 (2009) 47–65.

[27] I. Aït-Sadoune, Y. Aït-Ameur, Animating Event B models by formal data models, in: T. Margaria, B. Steffen (Eds.), ISoLA, volume 17 of *Communications in Computer and Information Science*, Springer, 2008, pp. 37–55.

[28] F. Ambert, F. Bouquet, S. Chemin, S. Guenaud, B. Legeard, F. Peureux, M. Utting, N. Vacelet, BZ-testing-tools: A tool-set for test generation from Z and B using constraint logic programming, in: Proceedings of FATES'02, pp. 105–120. Technical Report, INRIA.

[29] Clearsy, Atelier B tool homepage, 2010. http://www.atelierb.societe.com/.

[30] BCore, B-Toolkit tool homepage, 2010. http://www.b-core.com/btoolkit.html.

[31] E. Börger, R. Stärk, Abstract State Machines: A Method for High-Level System Design and Analysis, Springer-Verlag, 2003.

[32] R. Farahbod, V. Gervasi, U. Glässer, CoreASM: An extensible ASM execution engine, Fundam. Inform 77 (2007) 71–103.

[33] C. B. Jones, Systematic Software Development Using VDM, Prentice Hall, 2nd edition, 1990.

[34] C. Systems, VDMTools tool homepage, 2010. http://www.vdmtools.jp/en/index.php.

[35] Y. Ledru, Specification and animation of a bank transfer using KIDS/VDM, Autom. Softw. Eng 4 (1997) 33–51.

Appendix A. Refinements of the Coffee Model

```
machine CoffeeR1 refines CoffeeM
sees CofCtxt
variables alvl coins
invariants
  @ci coins ∈ ℕ
events
  event INITIALISATION
    extends INITIALISATION
    begin
      @ai coins := 0
    end
  event fill_mug
    extends fill_mug
    when
      @gc coins > 0
    then
      @delc coins := coins − 1
    end
  convergent event drink
    extends drink
  end
  anticipated event insert_coin
    begin
      @insc coins := coins + 1
    end
end
```

(a) First Refinement *CoffeeR1*

```
context ImpCtxt
constants maxc
axioms @amc maxc ∈ ℕ1
end

machine CoffeeR2 refines CoffeeR1
sees CofCtxt ImpCtxt
variables clvl coins
invariants
  @ifl clvl ∈ 0 .. 11
  @lvl alvl = level(clvl)
variant maxc − coins
events
  event INITIALISATION
    begin
      @cci coins := 0
      @fli clvl := 0
    end
  event fill_mug refines fill_mug
    when
      @gc2 coins > 0
      @ml clvl ∈ level⁻¹[{empty}]
    with
      @x x = level(clvl′)
    then
      @delc2 coins := coins − 1
      @ffl clvl :∈ level⁻¹[{full}]
    end
  convergent event drink refines drink
    when
      @dgfl clvl ∉ level⁻¹[{empty}]
    with
      @alvl’ alvl′ = level(clvl′)
    then
      @dfl clvl :∈ level⁻¹[{empty, half} \ {level(clvl)}]
    end
  convergent event insert_coin extends insert_coin
    when
      @gmc coins < maxc
    end
end
```

(b) Second Refinement *CoffeeR2*

Figure A.19: Refinements of Coffee Dispenser Machine