Transformational Heuristics for Animation - Towards Stepwise Validation of Specications

Atif Mashkoor, Jean-Pierre Jacquot

Transformational Heuristics for Animation - Towards Stepwise Validation of Specifications [*]

Atif Mashkoor[†], Jean-Pierre Jacquot[‡§]

December 7, 2010

Abstract

In formal methods, a key idea to assess that an implementation is correct is to break its verification into smaller proofs associated with each refinement step. Likewise, the technique of animation could be used during refinement process to break its validation into smaller assessments. Animating an abstract specification often requires to alter it in order to make it animatable. So we design a set of heuristics whose application transforms non-animatable specifications into animatable specifications and then based on these transformational heuristics, we develop a rigorous validation framework for stepwise validation of formal specifications.

1 Introduction

Formal languages are notorious for their comprehension difficulties. Furthermore, well written specifications often introduce abstract objects and operations that have no intuitive concrete counterpart. Hence, validation has to wait. This not only implies that the development of specifications requires an uncomfortable level of trust but it also raises an important question: when can we start validating?

Verification also raises a similar question. In test-based verification procedures, we need to wait until actual piece of code is implemented and running. As the cost of correcting errors or misunderstandings in requirements increases dramatically during the development life-cycle, it makes a lot of sense to verify and validate as early as possible.

The pivotal concept of formal methods, such as Event-B [Abr10] is the notion of refinement and its relation to correctness. The assessment of the correctness of a piece of code, its verification, is no more a unique big process step but it is broken down into small pieces along with the whole development process. The proof of correctness is then the sum of all proofs of small assertions (invariant preservation, well-formedness, existence of abstraction function, etc.) associated to each refinement. Problems are then detected early. While a formal refinement process does not preclude a testing activity, the latter will be more focused on finding true implementation errors, not requirement problems.

Our aim is to introduce validation into refinement based software development processes. We reap the benefits of our approach at two levels. First, early detection of problems in the

[*] Work partially supported by ANR under project ANR-06-SETI-017 TACOS (`http://tacos.loria.fr`) and by Pôle de Compétitivité Alsace/Franche-Comté under CRISTAL project (`http://www.projet-cristal.org`).

[†] Atif.Mashkoor@loria.fr

[‡] Jean-Pierre.Jacquot@loria.fr

[§] LORIA – Nancy Université, Vandoeuvre-Lès-Nancy, France

requirements (say, misunderstanding about a certain behavior) becomes easier and inexpensive to correct. Second, customers are involved into the development right from the start.

In this work, we focus on the "execution" of specification as a mean to validate it. However, be it the inability of animator to perform standard operations or specification itself consists of non-executable elements, such as non-constructive definitions, infinite sets, or complex quantified logic expressions, there are restrictions on the kind of specifications that can be animated.

We then design a set of heuristics which assists in animation of abstract specifications by systematic transformations. These transformational heuristics are designed to keep the behavior of specifications unaltered, possibly at the expense of other formal properties, such as provability. The correctness of these heuristics is then rigorously asserted with the help of a rigorous process.

This report is organized as follows: section 2 introduces the language and tool we have used for this work; section 3 discusses the concept of validation by animation; section 4 debates about animatable and non-animatable specifications; section 5 discusses the shortcomings of animator Brama; section 6 provides details on going from non-executable to executable specifications; section 7 provides some definitions we have used in the work; we then present our proposed problem solving transformational heuristics in section 8; followed by section 9, which highlights our proposed stepwise validation framework; a discussion on animation concludes this report.

2 Language and Tool

2.1 Event-B

Event-B is a formal language for modeling and reasoning about large reactive and distributed systems. Event-B is provided with tool support in the form of a platform for writing and proving specifications called RODIN[1].

An Event-B model is composed of two constructs: MACHINE and CONTEXT. Machine, which defines the dynamic behavior of the model, contains the system variables, invariants which define the state space of the variables and their safety properties, theorems, variants, and events. Context, which defines the static behavior of the model, contains carrier sets, constants, axioms, and theorems.

The refinement process is used to progress from abstract specifications to concrete and elaborated specifications. In refinements, new variables can be introduced and old variables can be refined to concrete ones. New events may also be introduced as long as they do not prevent forever the old ones from being triggered. Variants are explicitly introduced to ensure this property. Proof obligations are generated to ensure the consistency and correctness of both models: the abstract model and its refinement.

2.2 Brama

Brama [Ser06] is an animator for Event-B specifications. It is an Eclipse based plug-in for the Event-B platform RODIN. Brama can be used in two complementary modes: either Brama can be manually controlled from within the RODIN interface or it can be connected to a Flash[2] graphical animation through a communication server; it then acts as the engine which controls the graphical effects.

A typical animation session begins by setting the values of the constants in different contexts seen (either directly or transitively) by the animated machine. Then, the user must fire the `INITIALISATION` event which is, at that time, the only enabled event. After this, the user will

[1] http://rodin-b-sharp.sourceforge.net

[2] Flash is a registered trademark of Adobe Systems Inc.

play the animation by firing the events until there is no more enabled event, the system enters to a steady loop, or an error occurs (broken invariant or non computable action typically).

A graphical interface can be connected to Brama in the form of a Flash application and events can be directly fired from there. A mechanism of observers is provided. Expressions and predicates can be individually monitored and their value is communicated to the Flash program each time it changes. Last, a scheduler mechanisms allows for the automatic firing of events.

3 Validation by animation

Once a model has been formally specified, there are two important steps which realize its correctness: verification and validation. These two distinct yet closely related concepts are based on different techniques. While proof tools guarantee the consistency of the specification (verification), they are of little help to check if the specification models the desired behavior (validation). The figure 1 explains this phenomenon.

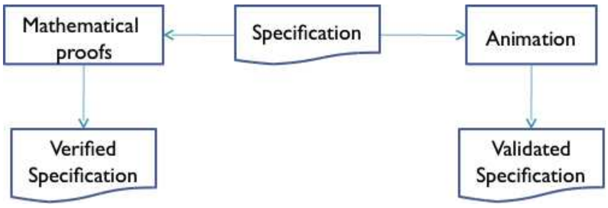


Figure 1: Verification vs. Validation

There are several ways to validate a specification: prototyping, structured walkthrough, transformation into a graphical language, animation, and others. All concur to the same goal: to evaluate a system to assess its conformance to its requirements, which later contributes in the demonstration that the system is operational. To answer the question of validation we use the technique of animation [HLP10]. Animation is a validation process which simulates the execution of the specification thus allowing the specifier to check that the specification has the intended behavior. Like the verification of the model can be broken down into smaller proofs associated with each refinement step, we integrate the technique of animation with each *observation level* [MJ10] of specification to break its validation into smaller assessments in order to ensure that it represents actual requirements.

For our work, we make a distinction between refinements and observation levels. Refinements allow concretization of specification while inducing proof obligations for formal correctness but their flat structuring process may impair its readability. Observation levels, on the other hand, provide specification with a super-structure which eases its understanding. They also facilitate independent introduction of new properties. Strictly speaking, observation levels are refinements which change the levels of abstraction.

Animation can be used early during the elaboration of the specification: there is no need to wait until it is finished. As a relatively low cost activity, animation can be frequently used during the process to validate important refinement steps. It then provides us with a validation tool consistent with the refinement structure of the specification process. This property may also be very interesting for certification of software due to several reasons. One of them is the fact that problems are detected close to the point where their cause was introduced. This facilitates the understanding of the cause. Another reason is the fact that an unforeseen behavior may be associated with a specific refinement. If we see a refinement as a formalization of a requirement, then we have an indication that some interactions between requirements need to be investigated.

4 Animatable vs non-animatable

Animation by nature heavily depends on tools. Any limitation of the tool will be a restriction on the class of animatable specifications. To validate a specification which does not belong to this class, we need to "bring it in". We do this by applying transformation rules which are designed to keep the behavior unaltered, possibly at the expense of other properties, such as "provability".

While it would be interesting for the theoretician to know whether some tools' limitations come from implementation features or have a deep mathematical reason; we, as practitioners, are more interested in designing practical rules for one particular tool. However, it is important to have an explicit rule design technique so that the current effort can be leveraged and transposed to other tools.

One can wonder why we do not produce an animatable specification at first. The reason is that our transformation rules "downgrade" the initial specification on two important counts: the specification becomes far less readable and, more importantly, may become unprovable. The transformation process tends to alter and suppress elements that are essential to discharge proof obligations.

The first observation we made when trying to animate a specification was the distinction between a provable specification and an animatable specification:

1. a provable specification may not be animatable,
2. an unprovable specification may be animatable,
3. most well written specifications are likely to be non animatable.

Like a "bad" program can be executed, an incorrect specification can also be animated, of course both would not solve the purpose. On the other hand, some important ingredients of specifications, such as non-constructive definitions, infinite sets, or complex quantified logic expressions are among the list of constructs, which are non-animatable. Unfortunately well-written specifications often use these traits. Indeed, it is even advised that early specifications be highly abstract and non constructive.

The first two bulleted items were a consequence of the first error message one is likely to encounter with the animator Brama: "Brama does not support finite axioms". Since these axioms are mandatory to discharge the well-formedness proof-obligations generated when using carrier sets, the case was settled. Beyond the anecdote (removing such technical axioms do not change the essence of the specification), this feature of Brama gave us the essential insight to dissociate proofs and animations. We could then focus on transformation rules which preserve behavior without bothering about preserving proofs (or provability).

Of course, by putting proofs aside, we are at risk of generating incorrect specifications. In fact, sometimes a transformation may not be provable within the formal Event-B rules. This implies that the correction of these transformations must be asserted through other means. We have then chosen to follow the mathematical tradition of providing rigorous and convincing arguments as a proof of the preservation of behavior for such transformation rules.

Our proposed pragmatic approach based on controlled transformations of specification is designed with a strong constraint: to replace non computable expressions by computable expressions while guaranteeing that the specification keeps the same behavior. We do not require the transformations to maintain the provability of the specification: we do not mind if some proof obligations cannot be discharged.

Since our aim is to validate a specification, we insist that the starting point of the animation job must be a fully proven specification: there would be no point in validating an unproven specification. The specification is then somehow downgraded to be animated.

5 Limitations of Brama

The situations where Brama cannot animate a specification can be arranged in a typology of five typical cases:

I Brama does not support the *finite* clause in axioms

II Brama must interpret quantifications as iterations

- **II.1** Brama only operates on finite sets
- **II.2** Brama cannot compute finite sets defined in comprehension with nested quantification
- **II.3** Brama explicitly requires typing information of all those sets over which iteration is performed in an axiom

III Brama cannot compute dynamic functional bindings in substitutions

- **III.1** Brama does not support dynamic mapping of variables in substitutions
- **III.2** Brama does not support dynamic function computation in substitutions

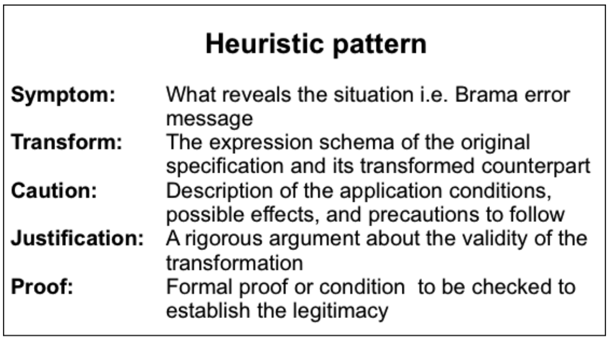
IV Brama does not compute arbitrary functions

- **IV.1** Functions with analytical definitions in context cannot be computed in events
- **IV.2** Functions using case analysis can not be expressed in a single event
- **IV.3** Invariants based on function computations can not be evaluated

V Brama has limited communication with its external graphical animation environment

These animation errors are due to two main reasons: either it is the limitation of the animator or the expression itself is too complicated to be executed.

For each situation, we have defined a heuristic to transform the original specification into one that can be animated. The heuristics are described following a rigid pattern shown by figure 2.



Heuristic pattern

Symptom:	What reveals the situation i.e. Brama error message
Transform:	The expression schema of the original specification and its transformed counterpart
Caution:	Description of the application conditions, possible effects, and precautions to follow
Justification:	A rigorous argument about the validity of the transformation
Proof:	Formal proof or condition to be checked to establish the legitimacy

Figure 2: The heuristic pattern

For each heuristic we first describe the *symptoms* i.e. in which particular cases this heuristic should be used. The *transform* explains how the original statement must be transformed in order to be animatable. *Caution* is the description of the applicability conditions, the possible effects, and the precautions to follow. In the *justification* part we provide a rigorous argument about the validity of the transformation. We describe why this solution works. In *proof* part we describe the formal condition to be checked to keep both specifications cohesive to each other. Although not strictly formal but this rigorous and clear description frame allows us to use animation safely to validate specifications.

It should be noted that our choice of tool, Brama, is contingent. At the time, it was the only one able to animate Event-B specifications. More recent tools such as AnimB[3] and ProB [LB03], are now available and fully compatible with Event-B. While our rules should surely be adapted to these specific tools, we suspect that the general philosophy of animation we have adopted is still valid.

6 From non-executable to executable

The technique of animation is based on execution of specifications, thus non-animatability means non-execution. Therefore, the rationale behind the proposed heuristics is to transform non executable specifications and make them executable. This goal is achieved primarily by reformulating the expressions and by adding some constructive elements to the specifications. Some of the constructive elements which we have used to make these specifications executable are: usage of extension for a finite domain; definition of upper and lower bounds to ensure termination; simplification of complex formulas, such as lists and sequences; rewriting of complex non-constructive expressions into executable format; inline/macro expansion of the formula instead of calling the function; decomposition of events to include all the cases defined by functions; etc. Our main constructive techniques are depicted by figure 3 and discussed as following:

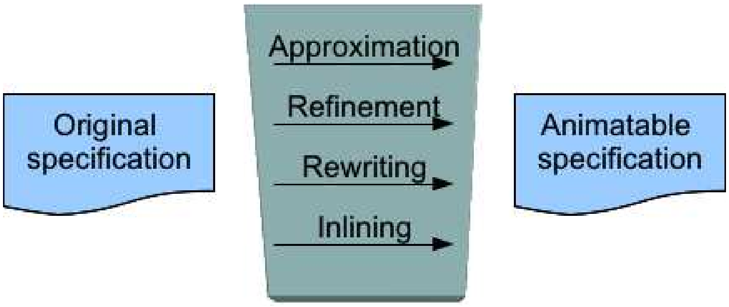


Figure 3: Types of transformational heuristics

6.1 Approximation

Approximation is a standard mathematical phenomenon to represent something close enough to the original and is a reasonably fast solution to be useful for computation and execution. Approximations are used when original formulas are too complex to compute or take longer time to compute than expected. In our transformations we use two types of approximations: under-approximation and over-approximation. These approximation techniques are based on abstract interpretation [CC77] and are often used to address state explosion problems in model checking.

We use under-approximation to address the termination problem. This is a specific termination problem which is based on enumeration of values. When a formula is based on an unbounded value it becomes non terminal because animator will continue enumerating it infinitely. Consequently Brama fails to execute such expressions. Therefore, we define their upper and lower bounds which settle the case.

[3] http://www.animb.org

In other cases, where we deal with complex data structures, such as sequences or lists, we exploit the over-approximation technique. The primitive data type of sequence is not provided in Event-B. In the time of need, we can use its standard definition which is based on nested quantifications (see section 8.3 for more details). As we discussed in previous paragraph that (un-bounded) quantifications cause problems during animation so consequently this definition fails to animate. Apparently it seems that under-approximation can again be the solution and defining bounds will solve the problem but a quantification over another quantification becomes very complex expression for animator to execute. Therefore we simplify the expression using over-approximation technique to achieve its executability.

The idea behind this transformation is two fold: first, simplification of formula to replace non-executable elements with something executable, second, to employ an holistic approach to quickly analyze and establish that if some property exists in the abstract (over-approximate) system then it holds in the concrete system that it abstracts. However, if the property is absent in the abstract system, we do not know if the concrete system violates this property.

6.2 Refinement

Refinement is an established formal activity to transform an abstract (high-level) formal specification into a concrete (low-level) executable program. This is exactly how we use this technique and transform our non-executable high level non-constructive formulas and expressions into low-level animatable and executable elements. Although transformations achieved by approximations can also be discussed in terms of abstract-refinement relationship, but the case of supply of missing type, case of event decomposition, and specially the case of introduction of observation variable/invariants/events are the cases of pure refinement.

The following proof obligation must be proved in order to define the abstract-refinement relationship between original and transformed specifications:

$P(s,c) \wedge I(s,c,v) \wedge Q(s,t,c,d) \wedge J(s,t,c,d,v,w) \wedge H(s,t,c,d,w) \wedge S(s,t,c,d,w,w') \Rightarrow G(s,c,v) \wedge \exists v'.(R(s,c,v,v') \wedge J(s,t,c,d,v',w'))$

Where v defines the variables of the abstract machine and w defines the variables of refined machine, s and c define the sets and constants of the abstract context, and t and d define sets and constants for refined context. The axioms on the sets and constants of the abstract context are denoted by $P(s,c)$ and on refined context by $Q(s,t,c,d)$. The invariant of abstract machine are denoted by $I(s,c,v)$ and refined machine by $J(s,t,c,d,v,w)$. $G(s,c,v)$ is the guard of the abstract event whose before-after predicate are defined by $R(s,c,v,v')$. The corresponding concrete event of the refined machine has the guard $H(s,t,c,d,w)$ and the before-after predicate $S(s,t,c,d,w,w')$.

6.3 Rewriting

Rewriting is a method to replace formulas and expressions with their equivalent counterparts. The rationale behind using this technique for transformation is same as others: to replace non-executable elements with their equivalent but executable counterparts.

In generalized substitutions, dynamic functions whose parameters are passed at runtime (non-deterministically) and depend upon the computations performed by guards, are hard for animator to compute. Same hardships are also faced when animators have to compute sets of tuples in generalized substitutions. This is animator's inability to perform some standard operations. Our approach towards these animation problems is to reformulate the non-computable formulas by their counterparts in set algebra. While less readable, they have same semantics and easy enough for animation.

6.4 Inlining

Inline/macro expansion is an optimization technique to replace the call of the function by its body. While writing specifications, this is a common practice to use functions for readability and simplifying proofs. These functions are generally defined in contexts. We use constants for their typing and axioms for their body definitions. Now we know that we need to feed the values to constants in order to perform animation on them. When these constants (which in turn are functions) are given values, they must adhere with the axioms which define the body of the function. Depending upon the axioms, sometimes there are more values for each function which need to be tabulated. This tabulation of values, at the time of feeding values, is not possible in Brama. The situation becomes further more complicated, when function definition consists of cases. Consequently, animation fails to execute.

This problem can be solved by using inline expansion technique i.e. to replace the function call by its body. We take the function body from the context and replace it in events where they are called. Like this we do not have to pre-define values at compile time and animator gives the values to guards at run time by itself, so problem is solved and animation is possible.

Inline expansion technique, in turn, is based on two previously defined transformational techniques: rewriting and refinement. It is rewriting because we are replacing the function call by its body which means semantically both expressions are equivalent, of course proper care has to be exerted with the use of involved variables. It can be defined as the refinement of the original machine if we can prove the enabledness preservation of the involved events. Following proof obligation must be proved:

$\forall S, C, Sr, Cr, V, Vr, x, xr. A \wedge Ar \wedge I \wedge Ir \Rightarrow (Gr \Rightarrow G)$

Where S and C respectively represent sets and constants of the abstract context, Sr and Cr respectively represent sets and constants of the refined context, V is the variable of the abstract machine, Vr is the variable of the refined machine, x is the local variable of the abstract event, xr is the variable of refined event, and A, Ar, I, Ir, G, Gr are the axioms, invariants and guards of abstract and refined machines respectively.

Before we describe our proposed heuristics in details one by one, let us first introduce some vocabulary and formal definitions in the forthcoming subsection which will help understanding what do we exactly mean by some technical terms.

7 Definitions

This subsection provides the formal definitions of the elementary concepts used while arguing about the correctness of the heuristics.

7.1 State

State is a set of mappings of variables to values constrained by conditions expressed with the help of invariants.

$state = \{variable \mapsto value\}$

7.2 Event

Event is a transition from one state to another. Event E is made of a guard G, which is a predicate built on state variables and expresses necessary condition for the transition, and generalized substitution S, which describes the way how state is modified.

$E(v) = \text{When } G(v) \text{ Then } S(v) \text{ End}$

We say that a state t is reached from a state s after an event $e(v)$ where e is the event and v is the parameter supplied to the event and we express this like $s \xrightarrow{e(v)} t$.

7.3 Behavior

Behavior is a sequence of sates and events. A behavior of a specification (b_s) is defined as:

$b_s = \{(s_1, e_1(v_1), s_2), (s_2, e_2(v_2), s_3), ..., (s_{\text{n}-1}, e_{\text{n}-1}(v_{\text{n}-1}), s_n)\}$

Speaking in terms of states, a state is reachable from its previous state provided previous (enabled) event and parameter.

$s_i = e_{\text{i}-1}(v_{\text{i}-1})[s_{\text{i}-1}]$

7.4 Transformation relation

There exists a transformation relation between a transformed specification and its original specification. We say that for all transformed events e' in a transformed specification $Spec_t$ there exists an event e in the original specification $Spec_o$ and a transformation relation $TransRel$ between both of these events and vice versa.

$\forall\ e'.\ e' \in Spec_t \Rightarrow \exists\ e\ .\ e \in Spec_o \wedge e' \mapsto e \in TransRel$

$\forall\ e.\ e \in Spec_o \Rightarrow \exists\ e'\ .\ e' \in Spec_t \wedge e' \mapsto e \in TransRel$

7.5 Shared states

Shared states are the common states of both original and transformed specifications. Let S_o and S_t be the set of all states in the original specification and the transformed specification respectively then S_c is a set of common states of both specifications.

$S_c = S_o \cap S_t$

7.6 Shared parameters

The set of shared parameters contains the values (parameters) which are legal/permissible in both (the original and the transformed) specifications. Let V_o and V_t be the set of all legal values in the original specification and the transformed specification respectively then V_c is a set of common values of both specifications.

$V_c = V_o \cap V_t$

7.7 Shared behaviors

Let $B_o = \{\{(s_1, e_1(v_1), s_2), (s_2, e_2(v_2), s_3), ..., (s_{\text{n}-1}, e_{\text{n}-1}(v_{\text{n}-1}), s_n)\}, ...\}$ and

$B_t = \{\{(s'_1, e'_1(v'_1), s'_2), (s'_2, e'_2(v'_2), s'_3), ..., (s'_{\text{n}-1}, e'_{\text{n}-1}(v'_{\text{n}-1}), s'_n)\}, ...\}$ be the set of all behaviors of the original and the transformed specifications respectively. Let b_o and b_t be any two behaviors of B_o and B_t respectively, then b_o and b_t are shared if they share same states and parameters, and their events establish a transform relation with each other.

$B_c = \{(b_o, b_t) | b_o, b_t, i.b_o \in B_o \wedge b_t \in B_t \wedge i \in dom(b_o) \wedge dom(b_o) = dom(b_t) \wedge s'_i = s_i \wedge v'_i = v_i \wedge e'_i \mapsto e_i \in TransRel\}$

Let us define a relationship $TransRel^*$ between original (B_o) and transformed (B_t) behaviors (i.e. a set of couples of b_o and b_t).

$TransRel^* \in B_o \leftrightarrow B_t$

$TransRel^* \in \mathbb{P}(B_o \times B_t)$

$TransRel^* \in \{\forall b_o, b_t, i.b_o \in B_o \wedge b_t \in B_t \wedge events(b_{t_i}) \mapsto events(b_{o_i}) \in TransRel\}$

Now if seen from transformed specification perspective, then

$B_c^t = \{b_t | b_t.b_t \in B_t \wedge TransRel^{*-1}[\{b_t\}] \subseteq B_o\}$
and if seen from original specification perspective, then
$B_c^o = \{b_o | b_o.b_o \in B_o \wedge TransRel^{*}[\{b_o\}] \subseteq B_t\}$

7.8 Behavioral equivalence

Two specifications $Spec_o$ and $Spec_t$ are behaviorally equivalent if all interesting behaviors[4] (starting from a shared state) observed in $Spec_t$ are shared with $Spec_o$.

$Spec_o \overset{B}{=} Spec_t \triangleq \forall b_i.b_i \in B_t \wedge s_1 \in S_c \Rightarrow b_i \in B_c^t$

7.9 Properties of behavioral equivalence

Following are the essential properties of behaviorally equivalent specifications:

1. Non-emptiness

   The sets of shared states, parameters, and behaviors are non-empty sets.

   $S_c \neq \emptyset$

   $V_c \neq \emptyset$

   $B_c \neq \emptyset$

2. Shared behaviors share states.

   $\forall b.b \in B_c \Rightarrow S_b \subseteq S_c$

7.10 Proofs for behavioral equivalence

The transformed specification $Spec_t$ is behaviorally equivalent to original specification $Spec_o$ ($Spec_t \overset{B}{\equiv} Spec_o$) if following properties hold:

- Enabledness preservation

  If an event is enabled with certain parameter at certain state in original specification then its transform must be enabled in transformed specification given same parameter and state.

  $Enabledness(Spec_o) = Enabledness(Spec_t) \triangleq [\forall s, e, v.s \in S_c \wedge e \in Spec_o \wedge v \in V_c \wedge enabled(e, v, s) \Rightarrow (\exists e'.e' \in Spec_t \wedge e' \mapsto e \in TransRel \wedge enabled(e', v, s))] \wedge [\forall s, e', v.s \in S_c \wedge e' \in Spec_t \wedge v \in V_c \wedge enabled(e', v, s) \Rightarrow (\exists e.e \in Spec_o \wedge e' \mapsto e \in TransRel \wedge enabled(e, v, s))]$

- State reachability

  If a state is reachable in original specification after an event with certain parameter then same state should be reachable in transformed specification as well given transform of the event and same parameter.

  $Reachability(Spec_o) = Reachability(Spec_t) \triangleq [\forall s, t, e, v.s, t \in S_c \wedge e \in Spec_o \wedge v \in V_c \wedge s \xrightarrow{e(v)} t \Rightarrow (\exists e'.e' \in Spec_t \wedge e' \mapsto e \in TransRel \wedge s \xrightarrow{e'(v)} t)] \wedge [\forall s, t, e', v.s, t \in S_c \wedge e' \in Spec_t \wedge v \in V_c \wedge s \xrightarrow{e'(v)} t \Rightarrow (\exists e.e \in Spec_o \wedge e' \mapsto e \in TransRel \wedge s \xrightarrow{e(v)} t)]$

[4] An interesting behavior is a behavior which starts from a shared state and we are interested to observe in the transformed specification and would have been observable in the original specification as well.

- Closure property

  All the states reachable from a shared state, after an event with a shared parameter, are shared states as well.

  $\forall s, t, e, v. s \in S_c \land t \in S_o \land e \in Spec_o \land v \in V_c \land enabled(e, v, s) \land s \xrightarrow{e(v)} t \Rightarrow t \in S_c$

- Behavioral equivalence

  If two specifications are behaviorally equivalent then they have same enabledness and reachability.

  $Spec_o \overset{B}{=} Spec_t \Rightarrow Enabledness(Spec_o) = Enabledness(Spec_t) \land Reachability(Spec_o) = Reachability(Spec_t)$

8 Transformational heuristics

The aforementioned animation problems of Brama discussed in section 5 lead us to design 10 transformation heuristics, one for each case. We designed the heuristics to preserve the behavior of the specification, not its formal properties. In particular, the transformed specification may not be provable in RODIN platform. The correctness of the transformation is then a crucial issue.

Since not all heuristics maybe provable within Event-B formal logic system, we relied on the mathematical tradition of "rigorous arguments". For this to work, we need a basic assumption: the initial specification text must have been formally verified. Most of the arguments given in the justification clause of heuristic rely on this hypothesis.

As aforementioned, to transform non-animatable specifications into animatable ones, our proposed heuristics are mainly based on four kinds of transformational techniques: approximation, refinement, inlining, and rewriting. While former is the technique related to state exploration from abstract interpretation point of view, refinement is a standard formal technique used to transform abstract specifications into concrete ones, and inlining and rewriting are the techniques used to facilitate computations for animator while preserving the semantics.

Kinds of transformations

Following are the kinds of transformational heuristics:

1. Context

   In contexts, we transform its axioms. Axioms are either modified or removed during the transformation process.

2. Machine

   In machines, we introduce transformations at two levels:

   (a) Invariant

   At invariant level, we remove an invariant from the specification.

   (b) Event

   At event level, we transform each event into its equivalent event. If some event does not require any heuristic application, its transformed counterpart is the image of it. In other cases, we introduce transformations in following fashions:

   i. Guards

   We modify the guard of an event by inlining the value of the involved function.

ii. Substitution
We modify the substitution of the event by rewriting it.

iii. Decomposition
A transformed event E' is the decomposition of (original event) E into multiple events and the multiplicity of events depends upon the original function definition which leads to event decomposition and cases defined by it.
$E'(v) = \{E'_1(v), E'_2(v), ..., E'_n(v)\}$
Where
$E'_i(v) =$ When $G'_i(v)$ Then $S(v)$ End
Guard of E is also decomposed accordingly and composition of all (decomposed) guards forms the original guard.
$G(v) = G'_1(v) \vee G'_2(v) \vee ... \vee G'_n(v)$
$\forall v.G(v) \Rightarrow \exists i.G'_i(v)$
$\forall i, v.G'_i(v) \Rightarrow G(v)$
$G'_i(v)$ is a guard of the corresponding decomposed event $E'_i(v)$. Please note that the generalized substitution is kept same as of the original event E.

8.1 Remove the axiom *finite* from the specification

Symptom: Error message that keyword *finite* is not supported.
Pattern: Remove all the instances of axiom *finite* from the specification.
Caution: Removal of axioms *finite* invalidates many well-formedness proof-obligations.
Justification: Axioms like finiteness and non-emptiness can be considered as purely technical axioms [MJS08] [MJS09a]. They do not bring much information into the specified system whose implementation will necessarily be finite, even if it could conceptually be infinite. These technical axioms are required by the inference rules used by the provers. Since all the sets of values will be defined by extension, the animation will work upon necessarily finite values. The behavior is trivially maintained.
Proof: This heuristic is the application of transformational kind 1. Because of its technical nature as defined in justification part, this heuristic does not require any formal proof.

8.2 Specify the finiteness of a quantified domain

Symptom: Error message about dependent variables which do not have an iterator.
Pattern: Limit the range of the list.

Original $n.n \in \mathbb{N} \Rightarrow expression(n)$

Transformed $n.n \in min..max \Rightarrow expression(n)$

Caution: The range must be wide enough so that the values computed during the animation never fall outside it. Some proof obligations may become impossible to discharge (e.g, $n + 1 \in \mathbb{N}$).
Justification: This heuristic is the opposite of the previous rule; the argumentation on the necessary finiteness of the values during animation holds. The major difference with the previous rule is the necessity to check during the whole animation that the range is always wide enough. If this condition is ensured, then the behavior is the same.

In broader formal framework spectrum, this is the example of refinement. The newly constructed expression is a refined version of the original expression which contains lesser but precise values. From more focused abstraction framework's point of view, this is under-approximation, which allows us to check quickly the state reachability by exploring the subset of the reachable states.

Proof: This heuristic is the application of transformation kind 1. In order to prove the correctness of this heuristic we need to show that the introduced limitation does not preclude some legitimate states which exist in original specification. Therefore it is imperative to check that following (closure) condition holds:

$\forall s, t, e, v. s \in S_c \wedge t \in S_t \wedge e \in Spec_o \wedge v \in V_c \wedge enabled(e, v, s) \wedge s \xrightarrow{e(v)} t \Rightarrow t \in S_c$

Now if the condition $v \in V_c$ can be ensured then this proof is straightforward. Since this heuristic explicitly requires the given range to be wide enough to incorporate all the legitimate values therefore the correctness can be ensured if this pre-condition is met.

For proofs like $n + 1 \in \mathbb{N}$ or $n - 1 \in \mathbb{N}$, we can use lazy proof approach or proof by demand i.e. always extending or retracting max and min to incorporate the desired value into the range.

8.3 Generalize expressions involving complex iterations

Symptom: Error message about the impossibility to build the iterators of the predicate.
Transform: Take super-set of the expression.

Original $var = \{x | \exists n. n \in \mathbb{N}1 \wedge x \in 1..n \rightarrow y\}$

Transformed $var \in \mathbb{P}(\mathbb{N} \nrightarrow y)$

Caution: Although there is an apparent similarity with the problem dealt with Rule 8.2, the situation is very different: the computation requires two levels of iterations. This transformation loosens the constraints on the values, some maybe essential to the behavior (for instance, the property that all integer between 1 and the length of the sequence belong to the range of the function). Brama cannot ensure anymore that the properties hold. The burden of the check is passed onto the input of the values.
Justification: On the subset of values shared by the specification (that is, those values respecting the constraints left out by the generalization), both specifications must have the same behavior. Two cases must be considered:

- the value is a constant: it does not change during the animation and it keeps its properties,
- the value is a variable: at least one of the proof obligations in the initial specification deals with proving that the result of the computation belongs to the set. Since the initial specification is verified, the values in the modified specification have the same property.

This is an example of abstraction because the transformed formula is an abstraction of the original formula. In abstraction framework, this technique is known as over-approximation. The idea behind this technique is that if a property holds in the abstract (over-approximate) system then it holds in the concrete system that it abstracts. However, if the property does not hold in the abstract system, we do not know if the concrete system violates this property.
Proof: This heuristic is the application of transformation kind 1. In order to prove the correctness of this heuristic we need to show that considering additional values do not leave out the original values which must have been part of the original specification. Again the (closure) condition states that:

$\forall s, t, e, v. s \in S_c \wedge t \in S_t \wedge e \in Spec_o \wedge v \in V_c \wedge enabled(e, v, s) \wedge s \xrightarrow{e(v)} t \Rightarrow t \in S_c$

Now if the condition $v \in V_c$ can be ensured then this proof is straightforward. Since this heuristic takes the super set of the values then it implicitly contains all the abstracted values.

8.4 Explicitly provide the typing information of all sets used in an axiom

Symptom: Error message about the impossibility to build the iterators of the predicate.
Transform: Always provide the type of variables.

original $x \Rightarrow expression(x)$
Transformed $x.x \in X \Rightarrow expression(x)$

Caution: The type provided must be consistent with the type inferred by the provers. Pay special attention in case of subtypes, for example set of even or odd (natural) numbers. In such cases take the super type i.e. $\mathbb{N}$.

Justification: Brama does not use the information derived by the provers. The provided set is actually redundant. Brama needs it to set up the iteration process. Two cases must be considered:

- if the type is equal to a carrier set, or a subset, the modified expression is just a redundant form of the initial expression,
- if the type is an infinite set, such as $\mathbb{N}$, then Rule 8.2 should also be applied. The same caution and reasoning apply.

Proof: This heuristic is the application of transformation kind 1. This is the case of supplying the missing type therefore semantics remain intact. Provided type must be fully consistent with the inferred type. Although Rodin doesn't show any proof obligation for this heuristic but this heuristic can also be justified as the refinement of the system.

8.5 Avoid dynamic mapping of variables in substitutions

Symptom: Error message: "Default number can not be casted to IMapplet". Brama does not compute sets of tuples in substitutions.

Transform: Rewrite the substitution to avoid mapping.
Original $var := \{x, y.x \in X \wedge y \in Y | x \mapsto y\}$
Transformed $var := (\{x \in X | x\} \times \{y \in Y | y\})$

Justification: The transformation is simply a rewriting of the initial expression as a formula in set algebra. While less readable, it has the same semantics.

Proof: This heuristic is the application of transformation kind 2(b)ii. This is the case of rewriting and if the transformed expression is the exact translation of the original expression in set algebra, then no other condition needs to be checked.

8.6 Avoid dynamic function computation in substitutions

Symptom: Error message: "Related invariant is broken after executing the event". Brama cannot apply a function defined by its graph in a substitution.

Transform: Rewrite the substitution to avoid function computation.
Original $var := \{x.x \in X | fun(x)\}$
Transformed $var := \{ran(\{x.x \in X | x\} \lhd fun)\}$

Justification: The transformation is simply a rewriting of the initial expression as a formula in set algebra. While less readable, it has the same semantics.

Proof: This heuristic is the application of transformation kind 2(b)ii. This is the case of rewriting and if the transformed expression is the exact translation of the original expression in set algebra, then no other condition needs to be checked.

8.7 Inline the functions defined in contexts in events

Transform: Substitute function calls by their "inlined" equivalent
Original (in Context) $\forall x.x \in S \Rightarrow f(x) = expression(x)$
Original (in Event) $f(v)$

Transformed (in Context) $true$

Transformed (in Event) $v \in S \wedge expression(v)$

Caution: All occurrences of f in the specification must be replaced; special care must be exerted when replacing formal parameters by actual values.

Justification: In a mathematical context, the value $f(v)$ is equal to its definition expression where v has been substituted to x; both expressions are interchangeable.

Contexts in Event-B are precisely meant to contain constants and general definitions, such as functions. Using this structure eases the proofs and provides better legibility. As for 8.5 and 8.6, the "inlining" heuristic is strongly connected to the issue of readability and understandability of formal texts.

Proof: This heuristic is the application of transformation kind 2(b)i. This is the case of inline expansion and following condition needs to be checked to ensure consistency:

$\forall S, C, Sr, Cr, V, Vr, x, xr. A \wedge Ar \wedge I \wedge Ir \Rightarrow (Gr \Rightarrow G)$. Since $Ir = I$ because invariants are not changed in this heuristic and $Ar \subset A$ because axioms of refined contexts are subset of original context, therefore we are left to prove: $\forall S, C, Sr, Cr, V, Vr, x, xr. A \wedge I \Rightarrow (Gr \Rightarrow G)$. See section 6.4 for more details about the proof.

8.8 Replicate events which use functions defined "by cases"

Symptom: Same as 8.7, plus a function defined "by cases"

Transform:

Original (in Context) $\forall x. x \in S \Rightarrow (p(x) \Rightarrow f(x) = expression(x) \wedge q(x) \Rightarrow f(x) = expression'(x))$

Original (in Machine) EVENTS

EVENT A

WHEN ...f(v)...

THEN ...f(v)...

Transformed (in Context) true

Transformed (in Machine) EVENTS

EVENT A1

WHEN ...

grdc1 p(v)

THEN ...

EVENT A2

WHEN ...

grdc1 q(v)

THEN ...

Caution: This heuristic must be followed by the application of 8.7. Check that all cases have been covered. Be particularly careful if the function is applied to several, different actual parameters; this may require several application of this heuristic.

Justification: The predicates used in the case definitions are equivalent to guards in events. They have the same form and are used for the same purpose. Events A1 and A2 are copies of A, except for the new guard: their union is equivalent to A. Hence the transformed specification has the same behavior as the initial specification.

This heuristic entails major surgery in the specification. A blind application may introduce many copies of the events. By using the structures of other guards (some may already prevent cases in the function definition to be used) and by grouping several function into one transformation, it is possible to reduce the number of duplications.

Proof: This heuristic is the application of transformation kind 2(b)iii. This is the case of inline expansion followed by event decomposition. Along with the proof defined for the previous heuristic, the additional condition needs to be checked is $\forall v.G(v) \Rightarrow \exists i.G'_i(v)$ and $\forall i, v.G'_i(v) \Rightarrow G(v)$ i.e. new guards cater all the cases defined by original guard.

8.9 Remove Invariants

Symptom: Error message about dependent variables which do not have an iterator.
Transform: Remove the related invariant.
Caution: Removal of invariant may pop up some proof-obligations.
Justification: Invariants express the conditions which specification must adhere. Removal of invariant is safe because (1) invariant do not modify behaviors (they are only observed) and (2) proof-obligations related to maintaining the invariant have already been successfully discharged.
Proof: This heuristic is the application of transformation kind 2a. Since we started from the proven specification and all (invariant related) proof obligations are already discharged, therefore it is safe to use this heuristics. Generally behavior of a specification is changed when newer states are reachable. Since no new state transition is defined therefore behavior is maintained. Although some of the consistency proof obligations can not be discharged anymore in Event-B formal framework.

8.10 Introduce observation variables/invariants/events

Symptom: Front end requires observatory elements
Transform: Introduce observation variables/invariants/events to the specification.
Caution: Use this heuristic only for observatory purposes not for introduction of new behavior
Justification: The observation variables, invariants and events are introduced to the specification when we want to demonstrate a particular behavior in external flash application. Since the flash interface is bound to Event-B specification where the actual values are being changed so it's easier to introduce new constructs there rather than at front end. These new constructs are purely cosmetic changes to the specification and only facilitates the graphical look of the specification and does not define any new behavior.
Proof: This is pure refinement and standard consistency checking proofs can ensure the correctness of this heuristic.

9 The 3-step validation framework

We now introduce our rigorous requirement validation framework for refinement based software development. In our work, we integrate the technique of animation with each observation level of specification to break its validation into smaller assessments. Our proposed stepwise validation framework, for each observations level, is summed up by figure 4 and as follows:

1. start from a *fully* verified specification. This step is essential.

2. for each non animatable trait:

   (a) pick an appropriate rule

   (b) check that the applicability conditions hold

   (c) prove that the argument used in the justification part of the rule is valid

3. animate for validation. If an anomalous behavior is encountered, modify the initial specification, prove it to be correct, and restart from step one.

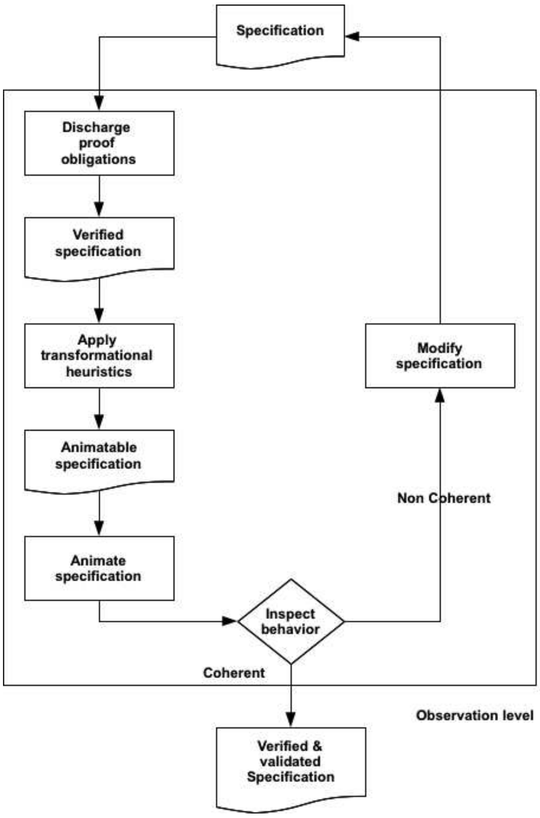


Figure 4: The stepwise validation framework

9.1 Step 1: Verification

The step 1 of our proposed validation framework is based on verification of specifications. Our belief is that there is no point in validating a specification which could not be verified! Such a specification is a dead-end as far as formal development is concerned. In our proposed validation process, a verified specification must be the starting point of the animation process. The application of the heuristics may downgrade it to a non provable specification. Running the animation may uncover some mistakes. These entail the modification of the initial specification, which then must be verified, and transformed again for proceeding with the validation.

It is important to note that the order between verification and validation is the reverse of what a development relying on tests would use. In the later case, there is no point in engaging a costly series of tests on a piece of code which does not fulfill users' needs. We give verification preeminence over validation mainly for two reasons. First, it provides us with a reasonable safeguard. Second, and more importantly, it allows us to justify some heuristics with sound arguments. For instance, let us consider two heuristics. One calls for the erasure of an invariant 8.9. This is safe because (1) invariant do not modify behaviors (they are only observed) and (2) proof-obligations related to maintaining the invariant have been successfully discharged. Another heuristic calls for the replacement of a set defined through complex properties of its elements by a simpler super-set 8.3. Provided we exert great care when feeding the animation with values which conform to the "complex" set, the transformation is safe because proof obligations have been discharged under the assumption that the values belonged to the "complex" set, and (1) either the values are only used (they are constants), and so properties are trivially maintained, or (2) the values are computed, but then at least one of the discharged proof-obligation was about the belonging of the computed value to the "complex" set. Though less direct, the justification for the other heuristics rely heavily on the fact that they are applied to verified texts.

9.2 Step 2: Transformation

As soon as all proof obligations have been discharged, we start animating the specification. This animation process is often struck either due to some shortcomings of animators or by some non-executable elements which are used to specify the behavior. This is the point where we introduce our proposed heuristics to the stepwise validation process. Whenever we discover any element in the specification which is non-executable, we inspect the problem and try to match the case with the list of our proposed heuristics. This inspection and matching practice includes checking if the same application condition holds as defined by heuristic framework and also that the use of this heuristic can be justified. This justification can either be provided in the form of formal proof or by a rigorous argument that application of heuristic would not change the behavior of the specification.

We have designed our heuristics with a very strong guideline: they must preserve the behavior of the specification. Behavior of a specification is defined by sequence of states and transitions. Based on *precise semantics*, the *transformation relation* of heuristics maintains shared behaviors between the original and the transformed specification. Some transformations, such as approximations may change the set of states or affect the provability. The proofs of *enabledness preservation*, *state reachability*, *closure*, are then employed to assert the *behavior equivalence*. Correctness of transformations are then again justified by other means, such as defining a *refinement-relationship* between the original and the transformed specification. Standard refinement and consistency checking conditions could then be proved in order to assert correctness rigorously.

9.3 Step 3: Animation

Once transformations have been applied, it means now specification is animatable. Animatable specification would demonstrate the behavior of the specification. If the demonstrated behavior is as per expectations then we have both the verified and the validated specification in our hands. However, if this is not the case and a closer look at the specification has revealed deviations from the intended behavior, then we need to go back to the initial specification and would have to correct the anomalous behavior. This triggers the loop i.e. re-proving, re-application of the heuristics, and re-animation until the specification conforms to actual requirements.

10 Animation: A reflection

Although animation is a strong contender to be a standard validation technique for real life projects, but during this academic research project it has been used as a "light-weight" validation technique. By light-weight validation, we mean that we did not have real customers and Software Requirement Specifications (SRS), consequently no "systematic" and "real" validation. Absence of real customers also indicates the absence of requirements coverage matrices. Even for covered requirements, we do not use different oracles to prove or disprove whether some test has passed or failed. We also have not defined and categorized different system's outputs against which we can compare our specification. No test plans, neither individual test cases nor collaborative test suites have been defined. Then it becomes a very pertinent question that what is the role of animation as a "hardcore" validation technique.

During our experience, we have discovered that animation is a multi-disciplinary validation technique. Animation can be used for variety of activities during software development. Primarily, animation is used as a quality assurance activity i.e. to gain confidence in specifications. It can also be used as prototyping. The benefit over here is that we can convert the specification into a prototype without translating it into code. It then acts as a quick and low cost validation technique.

Animation is also a modeler's first hand choice to quickly analyze what he/she has specified. After defining one behavior and before going to the next, specifier can use animation to be sure that he is going along the right path.

Animation can also be an aide during verification. Sometimes proofs became difficult to discharge. With animation at our disposal, we can thoroughly investigate some of the interactions among involved guards and axioms. It then provides us with some insight about the specification and helps us simplifying the proof.

Animation can also be beneficial in reducing software faults. There are many reasons for software faults, such as unrecognized requirements, bugs, etc. All of these result in errors and omissions in specifications. Some of the obvious or *never-thought-like-that* requirements can be discovered while animating the specification. The unintended behavior can then point out how the system should have worked ideally and what else need to be included in order to achieve desired results. Bug hunting with animation becomes far easier because deviation from the intended behavior clearly identifies where and what is wrong in the specification.

11 Related work

The concept of animation of specifications as a mean of prototyping and validation dates back to 80's [BGW82]. Since then many tools have been provided to help visualize requirements and system specifications, e.g. [SMRO97, VvLMP04, BLLS08].

There are two school of thoughts as far as execution of specifications is concerned. While one group believes that specifications should not necessarily be executable [HJ89], other group negates this idea and advocates that specifications should preferably be executable [Fuc92], some even propose transformation mechanisms to do so [Par90]. We find ourself inclined towards latter and our work is a continuation of this effort.

12 Conclusion and future work

This research explores the possibility to incorporate validation of formal specifications into their stepwise development process. We propose a rigorous validation framework for software specifications by systematic transformations.

Our approach is novel in a sense that we break the requirements validation process into small steps and integrate it into stepwise development of specifications. We are now able to detect and rectify errors right away even those which were still left behind after verification. Our framework, based on *low-cost* transformations and activities like animation, reduces the overall cost and time of validation process.

We have successfully utilized our framework for the validation of two safety critical case studies: a formal specification of land transportation domain [MJS09b] and a formal specification of platooning systems [MJ09]. We have also used our framework to validate gradual introduction of formalism into the requirement engineering phase [MM10].

Though we are assured that applications of heuristics are independent of each other i.e. application of one heuristic does not contradict with other and one application does not depend on another, yet this belief must be put on test. In future, we intend to formally establish that applications of heuristics are independent. We also plan to check that these heuristics are also compatible with animators other than Brama.

References

[Abr10] J.-R. Abrial. *Modeling in Event-B: System and Software Engineering.* Cambridge University Press, 2010.

[BGW82] Robert M. Balzer, Neil M. Goldman, and David S. Wile. Operational specification as the basis for rapid prototyping. *SIGSOFT Softw. Eng. Notes*, 7(5):3–16, 1982.

[BLLS08] J. Bendisposto, M. Leuschel, O. Ligot, and M. Samia. La validation de modèles Event-B avec le plug-in ProB pour Rodin. *Technique et Science Informatiques*, 27(8):1065–1084, 2008.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL '77: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252, New York, NY, USA, 1977. ACM.

[Fuc92] Norbert E. Fuchs. Specifications are (preferably) executable. *Software Engineering Journal*, 7:323–334, September 1992.

[HJ89] Ian Hayes and Cliff Jones. Specifications are not (necessarily) executable. *Software Engineering Journal*, 4:330–338, November 1989.

[HLP10] Stefan Hallerstede, Michael Leuschel, and Daniel Plagge. Refinement-animation for event-b - towards a method of validation. In *Second International Conference on Abstract State Machines, Alloy, B and Z (ABZ'10)*, pages 287–301, Orford, Canada, 2010.

[LB03] M. Leuschel and M. Butler. ProB: A model checker for B. In Keijiro Araki, Stefania Gnesi, and Dino Mandrioli, editors, *FME 2003: Formal Methods*, LNCS 2805, pages 855–874. Springer-Verlag, 2003.

[MJ09] A. Mashkoor and J.-P. Jacquot. Incorporating Animation in Stepwise Development of Formal Specification. Research Report INRIA-00392996, LORIA, Nancy, France, Jun 2009. http://hal.inria.fr/inria-00392996/en/.

[MJ10] Atif Mashkoor and Jean-Pierre Jacquot. Domain Engineering with Event-B: Some Lessons We Learned. In *18th International Requirements Engineering Conference (RE'10)*, Sydney, Australia, 2010.

[MJS08] A. Mashkoor, J.-P. Jacquot, and J. Souquières. Domain Modeling with Event-B: An Experience with Transportation Domain. Research Report INRIA-00326253, LORIA, September 2008. http://hal.inria.fr/inria-00326253/en.

[MJS09a] Atif Mashkoor, Jean-Pierre Jacquot, and Jeanine Souquières. B événementiel pour la modélisation du domaine: application au transport. In *Approches Formelles dans l'Assistance au Développement de Logiciels (AFADL'09)*, pages 1–19, Toulouse, France, 2009.

[MJS09b] Atif Mashkoor, Jean-Pierre Jacquot, and Jeanine Souquières. Transformation Heuristics for Formal Requirements Validation by Animation. In *2nd International Workshop on the Certification of Safety-Critical Software Controlled Systems - SafeCert'09*, York, UK, 2009.

[MM10] Atif Mashkoor and Abderrahman Matoussi. Towards validation of requirements models. In *2nd International Conference on Abstract State Machines (ASM), Alloy, B and Z (ABZ'10)*, Orford, Canada, 2010.

[Par90] Helmut A. Partsch. *Specification and transformation of programs: a formal approach to software development.* Springer-Verlag New York, Inc., New York, NY, USA, 1990.

[Ser06] Thierry Servat. BRAMA: A New Graphic Animation Tool for B Models. In *B 2007: Formal Specification and Development in B*, pages 274–276. Springer-Verlag, 2006.

[SMRO97] Jawed I. Siddiqi, Ian C. Morrey, Chris R. Roast, and Mehmet B. Ozcan. Towards quality requirements via animated formal specifications. *Ann. Softw. Eng.*, 3:131–155, 1997.

[VvLMP04] Hung Tran Van, Axel van Lamsweerde, Philippe Massonet, and Christophe Ponsard. Goal-oriented requirements animation. In *RE '04: Proceedings of the Requirements Engineering Conference, 12th IEEE International*, pages 218–228, Washington, DC, USA, 2004. IEEE Computer Society.