Stepwise Validation of Formal Specifications

Atif Mashkoor, Jean-Pierre Jacquot

Stepwise Validation of Formal Specifications

Atif Mashkoor and Jean-Pierre Jacquot

LORIA – Nancy Université
Vandœuvre lès Nancy, France
Email: {firstname.lastname}@loria.fr

***Abstract*—This paper explores the possibility to incorporate validation in the stepwise development process of formal specifications. Formal methods based on refinement break the intractable proof of the correctness of implementation into a sequence of many smaller proofs. Likewise, the validation of the specification could be broken into smaller steps associated to refinements with the technique of animation. Animating an abstract specification often requires to alter it in ways that proof obligations cannot be discharged anymore. So, we have developed a process and a set of transformation rules whose application produces an animatable specification which may be non-provable, but which is assured to have the same behavior. Guaranteeing behavioral preservation requires us to define an ad-hoc relationship between specifications based on a kind of trace semantics. 10 rules have been identified and proven to preserve behavior. Observations on the use of the technique on two case-studies are presented.**

***Keywords*-Formal methods, B, Event-B, Validation, Animation**

I. Introduction

Despite decades of advocacy, successes in the development of safety-critical systems, easier to read formal notations, and good proof tools, using formal specifications in software development is still not popular. Meanwhile, model-oriented methods, i.e., graphical formalisms, are on the rise and widely practiced. While some, like SCADE [1], have strong semantics and formal basis, most are less well-defined.

One of the reasons of the appeal of graphical formalisms is that users can be associated earlier in the development process. In particular, they can validate developers' understanding of the problem and early decisions. On the other hand, customers are likely to have difficulties in reading and understanding specifications written in a formal, mathematical, language. Actually, formal texts are often distant from the intuitive definition of the concepts and behavior of systems in the real world. Hence, validation [2] has to wait. This implies that the development of the specification requires an uncomfortable level of trust.

This difficulty has been identified long ago [3] along with a solution: presenting a graphical animation of the specification. Tools have been provided to help visualize requirements and system specifications [4]–[8]. The question is then *When* can we begin validation?

Verification [2] raises a similar question. In test-based verification procedures, we need to wait until actual pieces of code are implemented and running. As the cost of correcting errors or misunderstandings increases dramatically during the development life-cycle, it makes a lot of sense to verify and validate as early as possible.

Formal methods such as B [9] are built around the pivotal concept 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 the whole development process. The proof of correctness is the sum of the proofs of small assertions (e.g., invariant preservation or existence of abstraction function) 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 processes. We expect to gain on two levels. First, early detection of problems in the requirements (say, misunderstanding about a certain behavior) should be easier and inexpensive to correct. Second, users can be involved into the development right from the start.

In this work, we focus on the "execution" of a specification as a mean to validate it. Tools like Brama [10] or ProB [4] allow us to animate specifications in B or Event-B [11] before they reach an implementation stage. However, there are restrictions on the kind of specifications that can be animated. Non-constructive definitions, infinite sets, or complex quantified logic expressions are among the list of restrictions. Unfortunately, well-written specifications often use these traits. Indeed, it is even advised that early specifications be highly abstract and non constructive.

While toying with Brama, we observed that small alterations to a specification often allowed us to animate it, but at the expense of losing some of its formal properties: some proof obligations could not be discharged anymore. However, both specifications clearly described a common set of behaviors.

These observations lead us to develop a technique to animate abstract specifications by systematic transformation. The product of the transformations is a specification which may be non provable, but which is guaranteed to have the same behavior as the formally correct initial specification. This goal is achieved through the design of a set of transformational heuristics whose correctness is asserted by a rigorous process.

The paper is organized as follows: next section presents the language and tool we use: Event-B and Brama. Then, we present the animation process and the transformation rules. The formal semantics associated to the correctness of the transformations is discussed thereafter. We present some observations on the application on our technique in two case studies. Finally, we conclude with questions raised by this work and what should now be completed to have a technique that could be used as a standard practice.

II. Language and tools

A. Event-B

Event-B [11] is a formal language for modeling and reasoning about large reactive and distributed systems. It is based on set theory and standard first-order predicate logic. It is supported by RODIN[1], an environment for writing and proving specifications.

An Event-B model is composed of two constructs: machines and contexts. A typical machine defines the dynamic behavior of the model and contains the system variables, invariants, variants, and events. Invariants define the state space of the variables and their safety properties. Variants are related to the correctness of refinements. Events define the transitions between states and consist of guards and actions. A context defines static elements of the model and contains carrier sets, constants, axioms, and theorems. The last two are predicates expressed within the notation of first-order logic and set theory.

There are several relationships between machines and contexts: refinement, extension, and visibility. A machine can be a refinement of one, and only one, machine. A context can extend multiple contexts. A machine can see, that is, use the names and properties of, several contexts and a context can be seen by several machines.

Refinement is a process to add details to a model in a stepwise manner. The advantage of this technique is to break the complexity of the analysis and the proof of the model into smaller elements. Event-B embeds the notion of refinement which is then the basic element of the specification development process. The consistency of the abstract-refinement relationship needs to be proved when a new refinement step is introduced in the model.

The semantics of machines and refinements are given by proof obligations. Proofs ensure that machines meet essential system properties, such as safety, well-definedness, invariant-preservation, etc. The proof obligations generated by the tool must be discharged using provers, either automatically or interactively.

Proving a refinement correct amounts to prove that concrete events maintain the invariant of the abstract model, maintain the abstraction invariant, and, when appropriate, decrease variants monotonically.

B. Brama

Once a model has been specified using Event-B with RODIN, Brama [10], an Eclipse based animation plugin, can be exploited to execute it for its validation.

In Brama, a typical animation session begins by setting the values of the constants in the 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. Next, the user will play the animation by firing events until there is no more enabled event, or the system enters a steady loop, or an error occurs (broken invariant or non-computable action typically).

During the animation loop, Brama does the following:

- it picks values that make the guards true. When several values are possible, the choice is non deterministic;
- it computes the action part of the user's chosen event.

In a specification which includes several refinements, each one can be animated independently. The highly non-deterministic machines which are often found at the initial steps of the specification process may not be animatable, but this does not prevent the animation of further refinements where the non-determinism has been lowered.

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 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 mechanism allows for the automatic firing of events.

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

III. Transformational heuristics

By nature, animation depends heavily 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

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

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

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

do this by applying transformation rules which are designed to keep the behavior unaltered, possibly at the expense of other properties.

While the theoreticians may be interested to know whether the tools' limitations come from some implementation features or have deep mathematical roots; 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.

This section first discusses the technical issues associated with animating an Event-B specification with Brama. One of the important issues is the identification of the features of the language that require transformation. Then we present the designed process to address these issues along with the rationale. Finally, a systematic pattern to describe the transformations is presented along with a few selected heuristics.

A. Animatable versus provable

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) a non-provable specification may be animatable,
3) most well-written specifications are likely to be non-animatable.

The first two sentences were a consequence of the first error message one is likely to encounter: "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 axioms do not change the essence of the specification), this feature of Brama gave us the essential insight to dissociate proofs from animation. We could then focus on transformation rules which preserve the behavior without bothering about preserving proofs (or provability).

Of course, putting proofs aside seems incompatible with the core idea of formal methods. Section IV-A will show how we can demonstrate that the transformations and their applications preserve the behavior.

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 requires explicit typing information for all sets over which iteration is performed in axioms

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

For each situation, we have defined a "heuristic" to transform the original specification into one that can be animated. The heuristics are described following the rigid pattern shown on figure 1.

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

Figure 1. The heuristic pattern

Associated with the process described hereafter, this rigorous description frame, although not strictly formal, allows us to safely use animation to validate specifications.

B. Stepwise validation process

At the verification level, the consistency of refinement-based development processes is guaranteed by the generation of proof obligations and their discharge. Since animation requires us to loosen the provability constraint, the relation between verification and validation at the refinement level becomes an issue.

Our position 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.

A *verified* specification must be the starting point of the animation process. The application of the heuristics will "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 (if necessary) again for proceeding with the validation. This is summed-up in figure 2.

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 latter 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 for two reasons. First, it provides us with a safeguard. Second, and more importantly, it allows us to justify some heuristics with sound arguments.

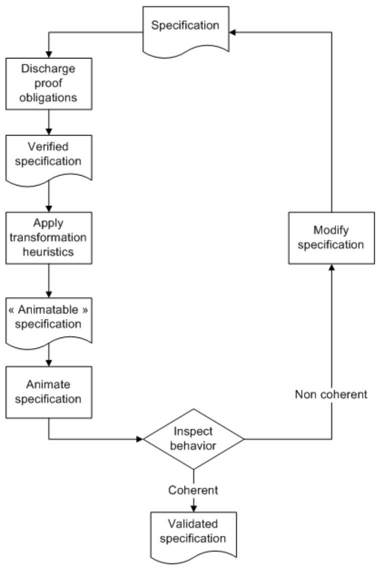

Figure 2. The validation process

C. Heuristics

As can be expected from the previous typology, we have designed 10 heuristics: one per category/case. The list is not closed; we may encounter in the future specifications with un-animatable traits not covered in this list.

Due to space limit, we present and discuss only Heuristics II.2, III.2, IV.1, and IV.2 (see [12] for more details). Heuristic I is about removing the `finite` axioms. This does not alter the behavior of the specification. The proofs of Heuristic II.1 and II.2 are similar: they end up in generating obligations for the parameters of the events. Heuristic II.3 is about providing explicit type information. Heuristic III.1, like III.2, is pure rewriting. Heuristic IV.3 calls for erasing an invariant and will be discussed in section IV-D. Heuristics V is only about the introduction of "observation" variables. They are required by the limitation of the communication protocol between Brama and Flash which is restricted to integers.

1) Heuristic II.2: Generalize list expression:
Symptom: Error message about the impossibility to build the iterators of the predicate.
Transform: Take super-set of the expression

```
Original  var = {x | ∃ n . n ∈ N1 ∧ x ∈ 1 .. n → y}
Transformed var ∈ ℙ (N ⇸ y)
```

Caution: This transformation loosens the constraints on the values, some may be essential to the behavior (for instance, the property that all integers between 1 and the length of the sequence belong to the domain 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.

If the transformed type concerns some parameters of events, it must be proven that only values belonging to the initial set can be selected.

This heuristic is quite specific and motivated by the absence of data-structures such as lists in Event-B. Redefining ad-hoc lists is not difficult but leads to intricate expressions. It should be noted that the problem does not come from the infinite set $\mathbb{N}_1$, but from the doubly quantified structure.

2) Heuristic III.2: 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 ∈ X |  fun(x)}
Transformed var := {ran ({x . x ∈ X | x}  ◁ 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.

One may wonder if this heuristic could not be replaced by a simple advice: "do not use function computation in set definition!" We do not think so. To our taste, the transformed text is far less readable, hence, more difficult to understand, to use in proofs, to maintain, or to correct. This question will be discussed in section VI-A.

3) Heuristic IV.1: Inline in events the functions defined in contexts:
Symptom: Functions defined analytically as constants in contexts can neither be initialized nor computed in events.
Transform: Substitute function calls by their "inlined" equivalent

```
Original  (in  Context)      ∀x. x∈S ⇒  f(x)  = expression(x)
Original  (in  Event)        f(v)
Transformed (in Context)  true
Transformed (in Event)  v∈S ∧ 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 III.2, the "inlining" heuristic is strongly connected to the issue of readability and understandability of formal texts.

4) Heuristic IV.2: Replicate events which use functions defined by "cases":

Symptom: Same as IV.1, plus a function defined "by cases."

Transform:

```
Original  (in  Context) ∀x. x∈S ⇒ (p(x) ⇒ f(x)  = expression(x) ∧
                                  q(x)  ⇒ 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 ...
                               grdc2  q(v)
                             THEN ...
```

Caution: This heuristic must be followed by the application of the Heuristic IV.1. Check that all the 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 "cases" definitions are equivalent to guards in events. They have the same form and 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. After applying the heuristics, it must be proven that when the initial event is enabled, at least one of the new events is also enabled, and that when one of the new event is enabled, the initial event is enabled too.

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

IV. Correctness of the heuristics

The most important advantage of using animation over any other prototyping technique to validate a specification is that we look at the behavior as exactly stated in the text; there is no untrusted intermediate. The catch is that our heuristics modify the text. So, we need to show that, as far as animation is concerned, the transformations are transparent.

A. Which correctness?

As said before, some heuristics do not preserve the semantics in a strict sense. Actually, many of them introduce notable changes in the model. Moreover, some render the text "incorrect" in the sense that the proof obligations cannot be discharged anymore. So, classical formal assessments of the relationship between the initial and transformed specifications, such as refinement, abstraction, or equivalence, cannot be used. We must define an ad-hoc relation.

Animating a specification is all about observing the behavior of a model, i.e., its evolution during an execution. Then, the property we want to assess is: "What is observed on the animation of the transformed specification would have been observed on the animation of the initial specification." Two further points should be noted.

First, we can restrict the relation to a form of inclusion of behaviors rather than a strict equality. We can "lose" behaviors (e.g., by restricting some ranges), but we cannot "add" behaviors (e.g., by allowing or forbidding transitions).

Second, during an animation, we can look only at two things: the enabledness status of all events, and the values of state variables. So, we should express the relationship with these two features of the execution.

B. Definitions

Our relation is based on a kind of trace semantics where we consider sequences of states and events. In the following, $Spec_x$ denotes a specification, i.e., a formal Event-B model. The basic elements of the semantics are then:

State: a mapping of names from set N to values from set V, constrained by the invariant (variables) or axioms (constants) of the specification

$$S = N \to V \ \wedge \forall s. s \in S \Rightarrow Inv(s)$$

Event: a transition from one state to another defined with the help of a guard G_e and a generalized substitution U_e

$$e = \ When\ G_e(s,v)\ \ Then\ U_e(s,v)\ End$$

where s denotes the state and v denotes the non-deterministic values (i.e., parameters) used by the event. We note the firing of an event as

$$s \ \xrightarrow{e(v)}\ t$$

Behavior: a sequence of states and event firing, starting from an initial state

$$\begin{aligned} & b \in seq(S \times E \times \mathbb{P}(V) \times S)\ \wedge \\ & \forall i. i \in dom(b) \Rightarrow\ (\mathrm{Pr}_4(b(i)) = \mathrm{Pr}_1(b(i+1))\ \wedge \\ & \qquad \mathrm{Pr}_1(b(i)) \ \xrightarrow{\mathrm{Pr}_2(b(i))(\mathrm{Pr}_3(b(i)))}\ \mathrm{Pr}_4(b(i)))\end{aligned}$$

where Pr_i denotes the i^{th} projection of the quadruples. We note B_p as the set of all behaviors of the specification $Spec_p$.

The two specifications which are compared may not have exactly the same events. So, we need to introduce a relation between events, Rel, defined as:

$$\begin{aligned} & \forall e'. e' \in Events(Spec_t) \Rightarrow \\ & \qquad \exists e. e \in Events(Spec_o) \wedge e' \mapsto e \in Rel \\ & \forall e. e \in Events(Spec_o) \Rightarrow \\ & \qquad \exists e'. e' \in Events(Spec_t) \wedge e' \mapsto e \in Rel \end{aligned}$$

where $Events(Spec)$ denotes the set of all events of the specification $Spec$.

Shared states is another important notion we introduce. Intuitively, a shared state is a state where all the variables common to both specifications have the same values:

$$S'_o = \{s.s \in S_o | N_t \cap N_o \lhd s\}$$
$$S'_t = \{s.s \in S_t | N_t \cap N_o \lhd s\}$$
$$S_c = S'_o \cap S'_t$$

From this, we define the notion of *shared behaviors* as the behaviors which go through the same sequence of states by firing events related by *Rel*. Let us denote Rel^* the extension of *Rel* to behaviors where each event in a behavior is related to the event at the same position in the other one:

$$\forall b_o, b_t . b_o \in B_o \wedge b_t \in B_t \wedge b_o \mapsto b_t \in Rel^* \Leftrightarrow$$
$$(\forall i . i \in \mathrm{dom}(b_o) \Rightarrow (\mathrm{Pr}_2(b_o(i)) \mapsto \mathrm{Pr}_2(b_t(i)) \in Rel))$$

The shared behaviors between two specifications $Spec_o$ and $Spec_t$, seen from the $Spec_t$ perspective are defined as:

$$B^t_c = \{b_t | b_t \in B_t \wedge (Rel^{*-1}[\{b_t\}] \subseteq B_o)\}$$

C. Behavior preservation

We would like to say that a specification $Spec_t$ preserves the behavior of $Spec_o$ if all the behaviors observed on $Spec_t$ are shared behaviors. This intuitive definition is slightly too broad and should be qualified on two aspects. First, the starting state must be a shared state. Second, all non-deterministic parameters must be admissible in both specifications. This property is expressed by the following predicates:

$$\begin{array}{l} validParam(v,s,e,Rel) = G_e(s,v) \wedge \\ \quad e \in \mathrm{ran}(Rel) \Rightarrow (\exists e'.e' \in Rel^{-1}[\{e\}] \wedge G_{e'}(s,v)) \wedge \\ \quad e \in \mathrm{dom}(Rel) \Rightarrow (\exists e'.e' \in Rel[\{e\}] \wedge G_{e'}(s,v)) \\ validParam^*(b,Spec,Rel) = \\ \quad \forall\ (s_i,e_i,v_i,t_i).(s_i,e_i,v_i,t_i) \in b \Rightarrow \\ \qquad validParam(v_i,s_i,e_i,Rel) \end{array}$$

So, the formal definition of behavior preservation is:

$$\begin{array}{l} Spec_t \overset{B}{\sim}_{|Rel} Spec_o \triangleq \\ \qquad \forall b_i . b_i \in B_t \wedge s_1 \in S_c \wedge \\ \qquad validParam^*(b_i, Spec_o, Rel) \Rightarrow b_i \in B^t_c \end{array}$$

This definition then needs to be connected to what is actually observed during an animation: which events are enabled and what are the values in the states.

SameEnabledness expresses the idea that on the shared states, events in both specifications have the same status (enabled or not); formally, the guard of both events is true.

$$\begin{array}{l} SameEnabledness(Spec_t, Spec_o, Rel) \triangleq \\ \quad (\forall s,e,v.s \in S_c \wedge e \in Events(Spec_o) \wedge \\ \qquad validParam(v,s,e,Rel) \wedge G_e(v,s) \Rightarrow \\ \qquad\quad (\exists e'.e' \in Events(Spec_t) \wedge e' \mapsto e \in Rel \wedge G_{e'}(v,s))) \wedge \\ \quad (\forall s,e',v.s \in S_c \wedge e' \in Events(Spec_t) \wedge \\ \qquad validParam(v,s,e',Rel) \wedge G_{e'}(v,s) \Rightarrow \\ \qquad\quad (\exists e.e \in Events(Spec_o) \wedge e' \mapsto e \in Rel \wedge G_e(v,s))) \end{array}$$

SameReachability expresses the fact that all states that can be reached from a shared state in a specification can also be reached in the other one.

$$\begin{array}{l} SameReachability(Spec_t, Spec_o, Rel) \triangleq \\ \quad (\forall s,t,e,v.s,t \in S_c \wedge e \in Events(Spec_o) \wedge \\ \qquad validParam(v,s,e,Rel) \wedge s \xrightarrow{e(v)} t \Rightarrow \\ \qquad\quad (\exists e'.e' \in Events(Spec_t) \wedge e' \mapsto e \in Rel \wedge s \xrightarrow{e'(v)} t)) \wedge \\ \quad (\forall s,t,e',v.s,t \in S_c \wedge e' \in Events(Spec_t) \wedge \\ \qquad validParam(v,s,e',Rel) \wedge s \xrightarrow{e'(v)} t \Rightarrow \\ \qquad\quad (\exists e.e \in Events(Spec_o) \wedge e' \mapsto e \in Rel \wedge s \xrightarrow{e(v)} t)) \end{array}$$

SameClosure states the idea that a behavior with valid parameters reaches only shared states from a shared state.

$$\begin{array}{l} SameClosure(Spec_t, Spec_o, Rel) \triangleq \\ \quad \forall s,t,e,v.s \in S_c \wedge t \in S_o \wedge e \in Events(Spec_o) \\ \wedge validParam(v,s,e,Rel) \wedge G_e(v,s) \wedge s \xrightarrow{e(v)} t \Rightarrow t \in S_c \end{array}$$

These definitions allow us to give the observation theorem: if two specifications have the three preceding properties, the first preserve the behavior of the second:

$$\begin{array}{l} SameEnabledness(Spec_t, Spec_o, Rel) \wedge \\ SameReachability(Spec_t, Spec_o, Rel) \wedge \\ SameClosure(Spec_t, Spec_o, Rel) \Rightarrow \\ \quad Spec_t \overset{B}{\sim}_{|Rel} Spec_o \end{array}$$

The proof of this theorem is easy. See [12] for details.

D. Application

In this subsection, we discuss the proof of three transformations. They are interesting because they introduce a notion of "proof obligation" which is clearly associated with each instance of the application of the heuristics.

Let us first consider Heuristic II.2 which replaces an expression by a super-set. In this heuristic, the number of events is not modified, so *Rel* is the identity relationship.

It is easy to see that the three properties of enabledness, reachability and closure can only be ensured if the parameters of the events are shared by both specifications. So, to use Heuristic II.2, we need to show that, for all events e:

$$\forall s,v.s \in S_t \wedge G_e(s,v) \Rightarrow validParam(v,s,e,Id)$$

This property does not hold in general, but typical cases may guarantee it. For instance, if v is picked in a constant set, the given values simply need to conform to the replaced complex subset. The proof obligation is then passed down to the procedure used to enter the values prior to the animation.

The second heuristic to consider is Heuristic IV.2 which replicates events. Formally, the major effect of the transformation is to introduce a non-trivial *Rel* relation. Then, the crucial property to ensure concerns enabledness. The proof obligation that can be deduced is:

$$G_e(v) \Rightarrow \exists e'.e' \in Rel[\{e\}] \wedge G'_{e'}(v) \wedge (\forall e'.G'_{e'}(v) \Rightarrow G_e(v))$$

It can be discharged within the Event-B formal framework.

The third heuristic we want to discuss is Heuristic IV.3 which removes invariant. The issue is with the reachability property. Many proof obligations that were discharged during the verification of the original specification actually dealt with the invariant preservation. We are sure that all invariants are maintained since we require that the original specification be verified. Also, since invariants do not impact the computation of values, the transformed specification reach only "legal" states, which are then shared states.

V. Case Studies

A. Cases studies

Both our case-studies deal with transportation. One is the specification of the domain of transportation [13], [14]. It consists of nine refinements. As a domain description, its aim is to define concepts such as travel or travel-time, properties such as collision avoidance, and behavior or protocols that maintain safety properties. Refinements are mostly about enriching the description. To help manage the introduction of complex model's features, we use the notion of observation levels. Those are a super-structure of the specification, which corresponds to different granularities of the description of the features.

The second is the specification of a control algorithm of autonomous cars moving as a platoon [15], [16]. The development guarantees that the vehicles never collide one with the others. We have two versions of the specification: a simplified, mono-dimensionnal (1D) one [17], and a more realistic, bi-dimensionnal (2D) one [18]. As a system specification, refinements are about getting closer to implementation: introducing more concrete data-structure and lowering non-determinism.

B. Observations and lessons

Using animation was a great help on three grounds:

Checking temporal properties: Although the 1D platoon refinements were all totally proven, meaning the last one could be converted directly into a *correct* program, simulations developed elsewhere still uncovered collisions. We found out the cause, a deadlock among events, through the animation. Temporal properties, such as deadlock-freeness, are notoriously difficult to specify and to prove in Event-B. Animation is a very effective way to test them.

Exploring features: The domain model includes complex features and protocols. Their expression is often tricky, some are even spread over several events, invariants and theorems. Getting the mathematical expression right is difficult. Using animation to "fix" tricky specifications is analogous to using debuggers to fix programs. It is a fast way to get correct expressions. Furthermore, animation helped us to get a deeper understanding of the features to specify.

Prototyping: Understanding a specification is difficult, even for its writers. Some definitions may interact in unexpected ways that are difficult to uncover by simply reading the text. Used in a "light" way, our approach allows to produce quickly reasonable prototypes. We used it several times on both case-studies to gain insight about the model and its description, and to identify problematic interactions.

C. Intrinsically unanimatable specification

Some specifications, or more precisely, some refinements, may be non-animatable, even after using the heuristics. The reason lies in the non-determinism which makes the enumeration strategy used to evaluate formulae blow up. Two cases should be considered.

With the domain model, this happened when introducing new features, typically at the first observation level. Then, the refinement was basically about introducing the "vocabulary" connected to the new feature, leaving operational specifications to later refinements. In that case, the lack of animation is not a problem. The new information was not about behavior. The important aspect of animatability is its non monotonicity: an animatable specification can be refined into a non animatable one, which can be itself refined into an animatable one.

With the platooning problem, we have a more difficult case of animation failure. The 2D version, while structurally identical to the 1D version, is not animatable because of the complexity of the state. It is the same state explosion problem faced by model-checkers. Other strategies than our transformations are needed to execute the model.

VI. Conclusion

We have presented a set of techniques and a process to make validation a concurrent activity to the development of a formal specification. While effective, they raise questions about the soundness of such validation, about the possibility to avoid transformation, about their relation to refinement-based development processes, and about the tool adequacy. Following are our answers to some of the questions raised:

A. Writing animatable specifications

A way to avoid the hassle of transformations would be to write specifications which are animatable form the start. Though appealing, we do not think this idea is effective.

If we try to write an animatable specification, we must introduce very early in the design process some arbitrary constraints (cf. Heuristic II.2) and use convoluted expressions (cf. Heuristic III.2). Worse, Heuristics IV shows that we must avoid the elegance of function definitions and replace them by long clumsy expressions. In all situations, the specifier commits a sin: over specification, esoteric notations, and unreadable text. The advice given for programming to keep things simple, general, and readable holds true for specification as well. More errors were corrected during the elaboration of the specifications while discharging the proof obligations and careful cross-reading than during the animation. Of course, they were of different nature.

Good readability and elegance are key factors for the acceptance of formal methods. Animation is just a technical activity which should not impose constraints on the specification.

B. Relation to refinement-based development processes

Provided the existence of the tools mentioned hereafter, our technique is a fast way to engage in validation activities. As seen in section V-B, applying the heuristics allowed us to get "quick and crude" animations which help set up a particular refinement. Of course, the animations then produced are not fully safe; once the refinement is well defined, the rigorous validation process must be carried out.

A specification is correct only if all the preceding refinements have been proven. Such a constraint, all refinement should be validated, would be excessive. As seen in section V-C, some refinements may not be animatable. We think that, for each feature, at least one refinement per observation level should be animated. Obviously, we need more examples to check this hypothesis. We also need to put more attention on the scenarios to use for an effective validation. Our feeling is that refinement of scenarios could follow the formal refinements. Work is needed here.

C. The missing tools

If we want stepwise animation to become a routine technique for developers, we need specific tools.

The Brama animation engine is actually adequate for the task. Improvements in the communication between Brama and Flash, notably by extending the types of values that can be transmitted, would be welcome. However, two tools are sorely lacking at present.

The first would be a tool to help in generating and inputing values into the constants of the contexts. Presently, the task is tedious and error prone. If a general tool would probably be non-realistic, at least a programmatic API to access and to set values from external programs should be implemented.

The second tool is the implementation of the heuristics. The major modifications implied by heuristics of category IV are difficult and tedious to carry by hand. A simple plugin to automate this process would be very practical. The proposed tools are the continuation of this work.

References

[1] "Esterel Technologies: SCADE Language Reference Manual," 2004.

[2] B. W. Boehm, *Software Engineering Economics*, 1st ed. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1981.

[3] R. M. Balzer, N. M. Goldman, and D. S. Wile, "Operational specification as the basis for rapid prototyping," *SIGSOFT Softw. Eng. Notes*, vol. 7, no. 5, pp. 3–16, 1982.

[4] 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*, vol. 27, no. 8, pp. 1065–1084, 2008.

[5] M. Leuschel, L. Adhianto, M. Butler, C. Ferreira, and L. Mikhailov, "Animation and Model Checking of CSP and B using Prolog Technology," in *Proceedings of the ACM Sigplan Workshop on Verification and Computational Logic VCL'2001*, pp. 97–109, 2001.

[6] H. T. Van, A. van Lamsweerde, P. Massonet, and C. Ponsard, "Goal-Oriented Requirements Animation," in *RE '04: Proceedings of the Requirements Engineering Conference, 12th IEEE International*. Washington, DC, USA: IEEE Computer Society, pp. 218–228, 2004.

[7] R. Schmid, J. Ryser, S. Berner, M. Glinz, R. Reutemann, and E. Fahr, "A Survey of Simulation Tools for Requirements Engineering," University of Zurich, Tech. Rep. 2000.06, 2000.

[8] J. I. Siddiqi, I. C. Morrey, C. R. Roast, and M. B. Ozcan, "Towards quality requirements via animated formal specifications," *Ann. Softw. Eng.*, vol. 3, pp. 131–155, 1997.

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

[10] T. Servat, "BRAMA: A New Graphic Animation Tool for B Models," in *B 2007: Formal Specification and Development in B*. Springer-Verlag, pp. 274–276, 2006.

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

[12] A. Mashkoor, "Formal Domain Engineering: From Specification to Validation," Ph.D. dissertation, Université Nancy II, Jul. 2011.

[13] A. Mashkoor and J.-P. Jacquot, "Domain Engineering with Event-B: Some Lessons We Learned," in *18th International Requirements Engineering Conference (RE'10)*, Sydney, Australia, 2010.

[14] A. Mashkoor and J.-P. Jacquot, "Utilizing Event-B for Domain Engineering: A Critical Analysis," *Requirements Engineering*, vol. 16, no. 3, pp. 191–207, 2011.

[15] P. Daviet and M. Parent, "Longitudinal and Lateral Servoing of Vehicles in a Platoon," in *Proceeding of the IEEE Intelligent Vehicles Symposium*, pp. 41–46, 1996.

[16] A. Scheuer, O. Simonin, and F. Charpillet, "Safe Longitudinal Platoons of Vehicles without Communication," INRIA, Research Report RR-6741, 2008.

[17] A. Lanoix, "Event-B Specification of a Situated Multi-Agent System: Study of a Platoon of Vehicles," in *2nd International Symposium on Theoretical Aspects of Software Engineering (TASE'08)*, Nanjing, China, 2008.

[18] F. Yang and J.-P. Jacquot, "Scaling Up with Event-B: A Case Study," in *NASA Formal Methods*, ser. Lecture Notes in Computer Science, M. Bobaru, K. Havelund, G. Holzmann, and R. Joshi, Eds. Springer Berlin / Heidelberg, vol. 6617, pp. 438–452, 2011.