Requirements Eng (2017) 22:433–451
DOI 10.1007/s00766-016-0246-6

ORIGINAL ARTICLE

Validation of formal specifications through transformation and animation

Atif Mashkoor[1] · Jean-Pierre Jacquot[2]

Received: 24 March 2014 / Accepted: 24 February 2016 / Published online: 16 March 2016

Abstract A significant impediment to the uptake of formal refinement-based methods among practitioners is the challenge of validating that the formal specifications of these methods capture the desired intents. Animation of specifications is widely recognized as an effective way of addressing such validation. However, animation tools are unable to directly execute (and thus animate) the typical uses of several of the specification constructs often found in ideal formal specifications. To address this problem, we have developed transformation heuristics that, starting with an ideal formal specification, guide its conversion into an animatable form. We show several of these heuristics and address the need to prove that the application of these transformations preserves the relevant behavior of the original specification. Portions of several case studies illustrate this approach.

Keywords Formal methods · Requirements specifications · Validation · Animation · Event-B

The writing of this article is partly supported by the Austrian Ministry for Transport, Innovation and Technology, the Federal Ministry of Science, Research and Economy, and the Province of Upper Austria in the frame of the COMET center SCCH.

✉ Atif Mashkoor
Atif.Mashkoor@scch.at

Jean-Pierre Jacquot
Jean-Pierre.Jacquot@loria.fr

[1] Software Competence Center Hagenberg GmbH, Hagenberg, Austria

[2] LORIA – Université de Lorraine, Vandoeuvre lès Nancy, France

1 Introduction

To be *correct*, a requirements document must be both complete and consistent. The former property concerns the fact that the document references all important requirements. The latter property concerns the fact that no requirement contradicts another one.

While there is no mathematical answer to the issue of completeness, formal techniques can be effectively used to determine the consistency of requirements [1]. During this process, requirements are specified using mathematics- and logic-based notations. There are operative definitions of the notions of verifiability and soundness for texts using such notations. The consistency of the requirements can then be assessed with the help of techniques like theorem proving and model checking.

However, when a document is written in a formal or semiformal language, a third property must also be checked: validity. It concerns the fact that the formal specification expresses the actual customer's requirements. This property can best be attained by involving customers in the formal modeling process.

Traditionally, software engineers distinguish between verification and validation. The former activity checks that a text enjoys some given formal, provable properties. The latter activity checks that the artifact answers the customer's needs. As verification is mathematically based, it has been the main focus of designers of formal methods so far. Current frameworks provide many tools and techniques to help developers produce verified texts, but far less to help produce validated texts.

As compared to validation, verification of a specification is a well-defined process. It ensures that involved expressions do not contradict with each other and maintain certain properties. Additionally, we also have a set of wellengineered assistant tools at our disposal. Theorem provers like ACL2 [2], PVS [3], HOL [4] and Isabelle [5] and model checkers like BLAST [6], NuSMV [7], PRISM [8] and SPIN [9] are already well established in the industry and have been successfully used in several industrial projects [10–13].

Casting requirements into predicates allows one to use proof techniques to assess the consistency. Formalisms, such as Z [14], B [15] or Event-B [16], provide us with further help through the notion of refinement which breaks huge proofs into many smaller ones. Yet, writing the specification and showing its consistency requires a considerable amount of interaction, efforts, and technical skills and know-how. As non-technical stakeholders usually lack these skills and know-how, it is very difficult to integrate such stakeholders into the modeling process unless the formal model is presented to them in a comprehensible form.

The case with validation is different. First, it is highly subjective. Second, we have fewer tools available for it. Third, even the tools available for validation have limitations, for example, inability to deal with all the language constructs, unbounded expressions or implicitly defined functions and operations. This is particularly true of those tools that execute models, for example CoreASM [17], Asmeta tool set [18], VDMTools [19] or ProB [20].

The tools most helpful to validate a specification rely on animation, the process of executing a formal specification by invoking its operational semantics. Animation, a mostly automatic process, is used to reveal the behavior of a formal specification either textually or graphically. This is similar to the structural exploration technique of [21], where a formal model containing a collection of constraints is fed to an analyzer. The analyzer then explores the model by generating sample structures. It also checks properties of the model, generating counterexamples when they are found to not hold.

Validation by animation is appealing even for non-technical stakeholders. However, the catch is that not all specifications are directly animatable; some need to be transformed to achieve execution [22, 23]. During the process of transformation, the non-animatable expressions are replaced by equivalent but animatable counterparts. Then the question is: Are these transformations sound so that the judgments made on such transformed specifications can be considered trustworthy as far as validation is concerned? The main aim of this work is to introduce an animation process based on behavior-preserving transformations for validation of formal specifications.

Verify-Transform-Animate (VTA) [24] is a framework for rigorous verification and validation of requirements specifications written in a formal refinement-based method. VTA relies on theorem provers and model checkers for analyzing the consistency of specifications. Once a specification is verified, it then proceeds for validation by animation. During the animation process, a specification that contains non-animatable traits is first transformed into a behaviorally equivalent animatable form. Animation is then used for validation. The result of the whole process is a specification which is both verified and validated.

The main benefit of this methodology is to enable the early detection of requirements problems (say, misunderstanding about a certain behavior). It comes from the fact that users can be involved in the process of checking the correctness right from the start. Users can join the validation part during the animation process, while leaving the technical proving to the technical experts.

The paper is organized as follows: we first present a brief overview of the VTA framework in Sect. 2. Section 3 discusses the difference between specifications classified in terms of their provability and animatability. Section 4 discusses how the class of a specification can be changed. Section 5 presents some transformational heuristics along with their semantics. Section 6 demonstrates the application of transformations on three case studies. Section 7 presents an evaluation of the proposed animation process. Section 8 provides some related work. Finally, the paper concludes with some proposed future work.

2 VTA

VTA is a proposition to associate validation with refinement. One of the major roles of refinement is to break the verification process into small assessments and to integrate it with the stepwise development process of the specification. VTA, powered by the techniques of verification, transformation and animation, is based on the same principle. VTA allows the correctness of a specification to be assessed throughout the development process.

The flow through the steps of the VTA framework is shown within the rectangle in Fig. 1. It consists of the following steps:

1. Formally specify the requirements by grouping them into observation levels,
2. Verify the specification:
   (a) Discharge all proof obligations (POs) and
   (b) Perform model checking when needed,
3. Transform each non-animatable element of the specification:
   (a) Choose the matching heuristic from the list,
   (b) Check that its applicability conditions hold,
   (c) Prove its application and
   (d) Apply the heuristic,

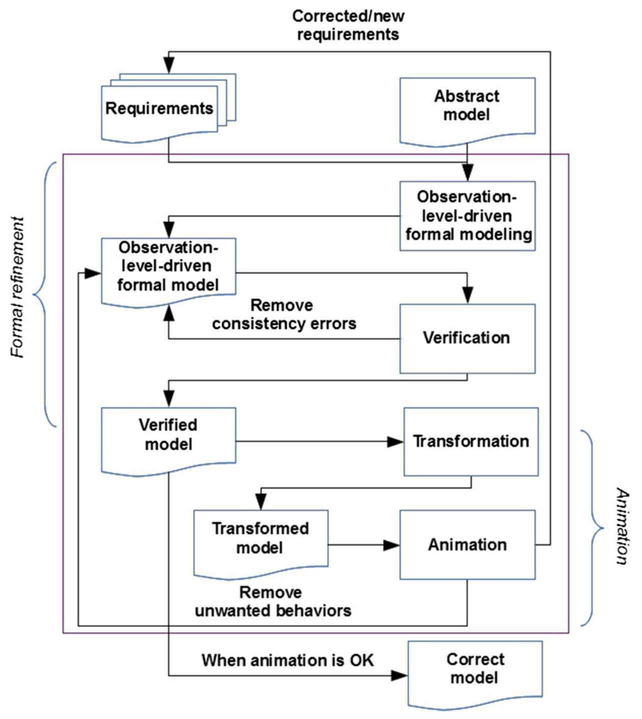
Corrected/new requirements
Requirements
Abstract model
Formal refinement
Observation-level-driven formal modeling
Observation-level-driven formal model
Remove consistency errors
Verification
Verified model
Transformation
Transformed model
Animation
Remove unwanted behaviors
When animation is OK
Correct model

Fig. 1 The VTA framework: structure of a refinement step

4. Validate the specification by its animation. If an unacceptable behavior is encountered, modify the requirements and restart from step one.

2.1 Observation-level-driven formal modeling

In VTA, an abstract requirements model is transformed into a formal specification through a technique that is based on observation levels [24]. An observation level is defined as a focus on a specific part of the model describing a unique aspect such as a specific protocol or a physical decomposition of the system. Grouping refinements into observation levels provides a specification with a superstructure which eases the understanding of the model. This arrangement reflects either the "natural" structure of the system being modeled, particularly when there are physical components, or of its behaviors, i.e., the evolutions of its state. This breakdown facilitates both comprehension and animation of formal requirements. With this approach, the important properties are introduced at the desired level of observation. Each observation level contains one or several refinements. We recommend animation of at least one refinement per observation level. Our rationale is that observation levels are correlated with fundamental characteristics of models which have strong impacts on behaviors. Checking that the specified behaviors are the valid ones, and that there is no bad emergent behavior, is of particular importance. Please see [25] for the detailed discussion.

2.2 Verification

The next step of our proposed framework is based on verification of specifications. While verifying a specification, both deductive verification and model checking are important. The VTA framework supports the usage of both verification techniques where appropriate. We firmly believe that verification should precede validation because there is no point in the validation of an inconsistent specification.

2.3 Transformation

As soon as the specification is verified, we prepare it for animation. If some unsupported features of the language or non-executable elements, such as non-constructive definitions, are encountered, they are transformed using the proposed heuristics (discussed in Sect. 5).

If a problem is discovered, we inspect it and try to match the case with the list of heuristics. This inspection and matching practice includes checking whether the heuristic's application condition holds. The application of a heuristic may raise a PO. We are then required to justify this application. This justification can either be provided in the form of a formal proof (discharge of the PO) or by a rigorous argument that the application of the heuristic would not alter the behavior of the specification.

2.4 Animation

Once the transformations have been applied to remove all uses of unsupported or non-executable language features, the specification is ready for animation. We use animation to demonstrate the behavior of the specification to the stakeholders. If the demonstrated behavior is as per expectations, then we have the verified and 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 to correct the unacceptable behavior. This triggers a loop, i.e., reproving, re-application of heuristics and re-animation until the specification conforms to actual requirements.

The animation cycle stops when all the scenarios that were designed from the informal requirements have been executed and the behavior of the specification has been approved by stakeholders.

First two steps of the VTA framework are out of scope of this paper. Rest of the paper will focus only on transformation and animation steps.

3 Animatability versus provability

Animatability and provability are distinct characteristics of a specification. Both depend on intrinsic properties of models and on the power of the tools used. Animatability is particularly dependent on the tools. Therefore, a specification may fall into one of four classes shown by Table 1.

Just as a faulty program can be executed, an incorrect specification can also be animated. Of course, neither would be an admissible solution to the problem at hand. However, observations on a program's execution can provide developers with insights guiding them toward a correct solution. Likewise, animation that reveals a specification to be invalid provides guidance to the developers on how the specification needs correction.

Formal specifications often make use of constructs that render them non-directly animatable. For example, non-constructive definitions, infinite sets or complex quantified logic expressions make specifications non-animatable. As animation, by nature, heavily depends on tools, so any limitation of the tool will also be a restriction on the class of animatable specifications.

One can always try to produce from the start a specification which belongs to the animatable class "Provable and animatable." However, this is not a good idea for two main reasons. The first reason is that the specifier should avoid over-specification [26]. The second reason concerns the refinement principles that encourage us to make liberal use of abstract definitions, non-determinism and small refinement steps [27].

A well-written specification can later, of course, be brought to the right class for the sake of animation. However, during the process of bringing specifications into an animatable class, the elements which are necessary to discharge POs in the verified specification may be altered or even suppressed. By compromising on proofs, we are at a risk of generating inconsistent specifications. In fact, sometimes we cannot prove within the formal rules of the given formal method that a transformation does not modify the original behavior. This implies that the provability of these transformations must be asserted through other means. In such cases, the mathematical tradition of providing rigorous and convincing arguments as a paper-and-pencil proof of the preservation of the behavior for each transformation heuristic can be followed.

4 Rendering a specification animatable

We may have to change the form of a specification to make it animatable. We do this primarily by reformulating its expressions and adding some constructive elements to it. The techniques to do this (depicted by Fig. 2) are the following.

4.1 Approximation

Approximation is a standard technique to modify a model so that the result is close to the original but has better computational properties. For our purpose, we look for approximations that will result in efficiently executable models. In our transformations, we use two types of approximations: under-approximation and over-approximation. The former is the idea of taking a reasonable subset of the original model, whereas the latter takes a superset. These approximation techniques are based on abstract interpretation [28] and are often used to address state explosion problems in model checking.

Under-approximation can be used to address the problem of non-termination. This is a specific kind of termination which is based on enumeration of values. When a formula is based on an unbounded value an animator may continue enumerating it indefinitely. Consequently, animation fails. Restricting the enumeration within finite bounds addresses the problem. In other cases, where a formula is constituted of complex and composite data structures, such as sequences or lists, the technique of over-approximation can be exploited to simplify the formula and achieve its execution. For instance, a list, which is a total function on an interval of integers, can be over-approximated by a partial function on integers.

The rationale of using approximation for model checking is applicable here as well. For example, if some property exists in the abstract (under-approximate) model, then it holds in the concrete model. However, if the property does not hold in the former, we do not know whether the latter violates this property.

4.2 Refinement

Refinement is an established formal activity to transform an abstract formal specification into a concrete executable program. When possible, VTA uses refinement to transform non-executable high-level non-constructive

Table 1 Classes of specifications

	Non-animatable	Animatable
Non-provable	Non-provable and non-animatable	Non-provable but animatable
Provable	Provable but non-animatable	Provable and animatable

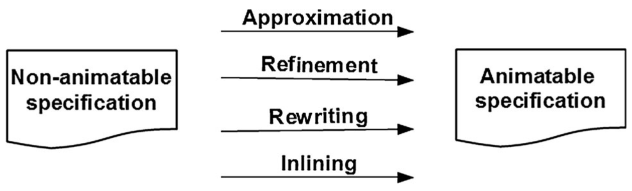

Fig. 2 Types of class changing techniques

formulas and expressions into lower-level animatable and executable elements.

When a specification is refined, we need to prove the abstract–refinement relationship between the two models. This amounts to establish two properties:

1. The refined model maintains the invariant of the abstract model. We must prove that the refined guards are stronger than the originals. Furthermore, the resulting actions do not lead to an incorrect state in the abstract specification. We must also prove that the new events are refinement of the SKIP event (i.e., the "do-nothing" event).
2. The new events do not introduce a divergence. Technically, we must prove there is no infinite chain of new events.

4.3 Rewriting

Rewriting is the process of replacing either some sub-terms or the whole formula with equivalent terms. In VTA, term rewriting is used to simplify non-animatable complex formulas to make them animatable. Application of this technique is fruitful for formalisms such as B or Z, where generalized substitutions are used to describe state modifications. Animators often find it difficult to compute the state transition relation if it contains dynamic functions whose parameters are passed at runtime and depend upon the computations performed by guards. As a solution, the non-computable formula is then partly or completely rewritten by its equivalent counterpart in set algebra or conjunctive normal form (CNF).

4.4 Inlining

Inline/macro-expansion is an optimization technique to replace a call of a function by its body. While writing specifications, this is a common practice to use functions for readability and simplifying proofs. A function based on a case analysis has multiple definitions and cannot be enumerated straightforwardly, thus preventing the execution of the incorporating specification. This problem can be solved by using inline expansion technique, i.e., to replace the function call by its body. Thus, enumeration is no longer required and the animator proceeds with its normal operation.

Inline expansion, in fact, is based on two previously defined transformation techniques: rewriting and refinement. It is rewriting because the function call is being replaced by its body which means semantically both expressions are equivalent. Of course, proper care has to be exerted with the use of the involved variables. It can be defined as refinement since the PO of enabledness preservation (see Sect. 5.2) which must be discharged requires us to prove that if a transition is enabled in the transformed specification, then it should also be enabled in the initial specification and vice-versa. Formally, the enabledness preservation PO is defined by a conjunction where the first formula is a standard Event-B PO for event refinement:

$$\forall S_a, C_a, S_r, C_r, V_a, V_r, x_a, x_r . A_a \wedge A_r \wedge I_a \wedge I_r \Rightarrow (G_r \Rightarrow G_a)$$
$$\wedge$$
$$\forall S_a, C_a, S_r, C_r, V_a, V_r, x_a, x_r . A_a \wedge A_r \wedge I_a \wedge I_r \Rightarrow (G_a \Rightarrow G_r)$$

where S_a, C_a, S_r and C_r represent sets and constants of the abstract and refined specifications, respectively. V_a and V_r denote variables of the abstract and refined specifications, respectively. x_a and x_r represent local variables of the abstract and refined state transition relation, respectively. A_a, A_r, I_a, I_r, G_a, G_r are axioms, invariants and guards of the abstract and refined specifications, respectively.

5 Transformational heuristics and their semantics

From the general principles used to make a specification animatable, we can design practical heuristics tailored to a specific specification language and a specific animation tool. The correctness of heuristics and of their application becomes an issue at two levels. At the usage level, users must be confident that they chose and applied an adequate heuristic. At the formal level, we must guarantee that the behaviors of the transformed model are the same as those of the original model. We address this issue of correctness using a two-step approach.

(a) *Step 1*: we present the heuristics using a pattern and give rigorous arguments to justify their use. We assume that they are applied to an already verified formal text. The pattern is shown in Table 2.

For each heuristic, we first describe the **symptom**, i.e., what indication from the animator of its inability to execute a specification would prompt the use of this heuristic. It also indicates the construct of the model, such as axiom, guard or transition statement, where the problem lies and which is susceptible to modification. The **transform** explains how the original statement must be transformed in order to be animatable. Each transform is based on the execution techniques discussed in Sect. 4. **Caution** is the

Table 2 The heuristic pattern

Heuristic pattern	
Symptom	What reveals the situation, i.e., the error message generated by the animation engine
Transform	The expression schema of the original specification and its transformed counterpart
Caution	Description of the application conditions, hypotheses to check, possible effects and precautions to follow
Justification	A rigorous argument about the validity of the transformation

description of the applicability conditions, the assumptions to check, the possible effects and the precautions to follow. In the **justification** part, we provide a rigorous argument about the validity of the transformation.

(b) *Step 2*: we define a formal semantics of transformations to give a proof of soundness of their application. The proof indicates under which conditions both the original and transformed specifications are behaviorally equivalent, i.e., provided same values, the same sequences of state transitions can be followed on both specifications.

Animating a specification is all about observing its behavior, i.e., its evolution during its execution. Then, the property we want to assure is: "what is observed on the animation of the transformed specification would have been observed on the animation of the initial specification (be it possible)." 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 transitions). Second, during an animation, we can look only at two things: the enabledness status of all transitions and the values of state variables. So, we should express the relationship with these two features of the execution.

5.1 The heuristics

During our experimentation with valuation-based animators, such as Brama [29], we have encountered ten kinds of impediments to animation of formal specifications and designed heuristics to deal with each of them. In the interest of brevity, in this paper we discuss only four of them in detail and summarize the other six. The reader is referred to [30] for a detailed description of all ten of them.

Table 3 contains the list of symbols used in the following sections.

Heuristic 1: Generalize expressions involving complex iterations

This heuristics is motivated by the difficulty of iterating over complex nested predicated expressions. Such expressions can occur when models use types such as lists or trees.

Symptom: Failure of an animator to build iterators of a predicate. The problem lies often with list-like types.

Transform: Take the superset of the expression.

Original $var = \{x | \exists n . n \in \mathbb{N}_1 \wedge x \in 1 \ldots n \rightarrow y\}$

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

Caution: This transformation loosens the constraints on the values, some of which maybe essential to the behavior. For instance, the property that all integer numbers between 1 and the length of the sequence belong to the domain of the function. An animator may not ensure any more that this property holds. The burden of the check is passed onto the input of the values. It must be ensured that animation is performed on a shared set of values between the original and transformed specifications.

Justification: On the subset of shared values, 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 associated with a constant: it does not change during the animation and it keeps its properties,
- the value is associated with a variable: at least one of the POs 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 one. In abstraction framework, this technique is known as over-approximation.

Heuristic 2: Avoid expressions involving mapping of variables in substitutions

Some animators have difficulty with computing set values defined by comprehension. This can often be overcome by rewriting the term using Cartesian product.

Table 3 The symbol table

Symbol	Meaning	Symbol	Meaning
$\vert$	Such that	$\cap$	Intersection
$\exists$	There exists	$\forall$	For all
$\rightarrow$	Total function	$\nrightarrow$	Partial function
$\in$	Element of	$\subseteq$	Subset of
$\mathbb{N}$	Set of natural numbers	$\mathbb{N}_1$	Set of +ve natural numbers
$\mathbb{P}$	Power set	$\mapsto$	Maplet
$\lhd$	Domain restriction	$\rightarrowtail$	Total injection
$\Leftrightarrow$	Logical equivalence	$\Rightarrow$	Logical implication
$\wedge$	Logical conjunction	$\vee$	Logical disjunction
$\neq$	Not equal to	$=$	Equal to
:=	Becomes equal to	:$\vert$	Becomes such that
$>$	Greater than	$\emptyset$	Empty set
$\mathbb{B}$	Boolean	$\times$	Cartesian product

Symptom: Failure of an animator to compute sets of tuples in substitutions. The problem lies in substitutions of the model.

Transform: Rewrite the substitution to avoid mapping.

Original $\{x, y.x \in X \wedge y \in Y | x \mapsto y\}$

Transformed $\{x \in X|x\} \times \{y \in Y|y\}$

Justification: The transformation is simply rewriting of the initial expression as a formula in set algebra. This heuristic can also be used in guards and axioms.

Heuristic 3: Inline the function definition in events

Some formal methods do not distinguish between functions defined as finite maps and functions defined by analytical expressions. The latter are defined as constants using axioms which cannot be assigned a value by enumeration-based animators.

Symptom: Failure of an animator to assign the start-up values to complex functions. The problem is associated with the axioms of the model which define analytical functions.

Transform: Substitute function calls by their inline equivalent

Original (in axiom) $\forall x.x \in S \Rightarrow f(x) = expression(x)$

Original (in transition) $f(v)$

Transformed (in axiom) *true*

Transformed (in transition) Add a new guard $v \in S$ and replace *f*(*v*) with *expression*(*v*).

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

Justification: This is the case of refinement. 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.

Heuristic 4: Replicate transitions which use functions defined "by cases"

Some formal methods do not support conditional constructs such as *if-then-else*. Specifiers must define functions with "cases" through axioms written as disjunctive formulas. Some animators also cannot handle such definitions.

Symptom: Same as Heuristic 3 plus a case analysis.

Transform:

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

Original (in transition)

```
    Transition A
        WHERE ...f(v)...THEN ...f(v)...END
```

Transformed (in axiom) $true$

Transformed (in transitions)

```
    Transition A1
    WHERE ... grdCase1 p(v) THEN ... END
    Transition A2
    WHERE ... grdCase2 q(v) THEN ... END
```

Caution: This heuristic must be followed by the application of Heuristic 3. Check that all cases have been covered. Be particularly careful if the function is applied to several different actual parameters; this may require several applications of this heuristic.

This heuristic entails a major surgery in a specification. A blind application may introduce many copies of state transition relations. By grouping several functions into one transformation, it is possible to reduce the number of duplications.

Justification: This is a case of refinement. The predicates used in "by case" definitions are equivalent to guards in state transitions. They have the same form and are used for the same purpose. The state transition relations $A1$ and $A2$ are the copies of A (except for the new guard); their union is equivalent to A. Hence, the transformed specification has the same behavior as the original specification.

The six other heuristics are summarized below.

Removing the `finite` axioms. Such axioms are introduced in specifications just to discharge the related POs; however, they do not alter the behavior of the specification. Hence, it is safe to remove them.

Specifying the finiteness of a quantified domain. For example, if the range is of natural numbers, specifying a finite range between a minimum and a maximum. This is the issue of termination that is a common animation problem. Our solution to fix it by stating that any variable, parameter or constant can only take finitely possible values is a standard solution for such problems.

Explicitly providing the typing information of all variables and constants used in a predicate. While proving theorems, provers can automatically infer the typing information of involved variables and constants; however, this is not the case with valuation-based animators which explicitly require this information to set up the enumeration process.

Avoiding dynamic function computation in substitutions. This heuristic is similar to Heuristic 2 and requires the same treatment: rewriting.

Complex invariant predicates. Invariants are conditions that must be satisfied by the behavior of a specification. In the case of inability to compute them, either they can be rewritten (like Heuristic 2) or removed (provided that they have already been taken care of during the verification process).

Introduction of "observation" variables. These variables are required due to the limitation of the communication protocol between the animator and the external graphical environment. For example, Adobe Flash® has limited support for data structures. Our solution in this case is to transform the output values unsupported by the external graphical environment.

5.2 Formal semantics of transformations

The transformational heuristics proposed in VTA actually modify the original specification. Therefore, we need to show that, as far as animation is concerned, what is observable on the transformed specification would have been observable on the original specification, if it was animatable.

Our work is based on a kind of trace semantics where we consider sequences of states and transitions. In the following, $Spec_x$ denotes a specification. 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

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

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

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

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

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

where Pr_i denotes the ith projection of the quadruples. We note B_p as the set of all behaviors of the specification $Spec_p$

Relation: the two compared specifications may not have exactly the same events, so we need to introduce a relation between events, Rel, defined as:

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

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

Shared state: a state where all the variables common to both specifications have the same values:

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

Shared behaviors: 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:

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

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

$$B_c^t = \{b_t | b_t \in B_t \land (Rel^{*-1}[\{b_t\}] \subseteq B_o)\}$$

Behavior preservation: 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:

$$validParam(v, s, e, Rel) = G_e(s, v) \\ \land e \in \mathrm{ran}(Rel) \Rightarrow (\exists e' . e' \in Rel^{-1}[\{e\}] \land G_{e'}(s, v)) \\ \land e \in \mathrm{dom}(Rel) \Rightarrow (\exists e' . e' \in Rel[\{e\}] \land G_{e'}(s, v))$$

$$validParam^*(b, Spec, Rel) \\ = \forall\ (s_i, e_i, v_i, t_i) . (s_i, e_i, v_i, t_i) \in b \\ \Rightarrow validParam(v_i, s_i, e_i, Rel)$$

So, the formal definition of behavior preservation is:

$$Spec_t \underset{|Rel}{\tilde{B}} Spec_o \\ \triangleq \forall b_i . b_i \in B_t \land s_1 \in S_c \\ \land\ validParam^*(b_i, Spec_o, Rel) \Rightarrow b_i \in B_c^t$$

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.

$$SameEnabledness(Spec_t, Spec_o, Rel) \\ \triangleq (\forall s, e, v . s \in S_c \land e \in Events(Spec_o) \\ \land validParam(v, s, e, Rel) \land G_e(v, s) \\ \Rightarrow (\exists e' . e' \in Events(Spec_t) \land e' \mapsto e \in Rel \land G_{e'}(v, s))) \\ \land (\forall s, e', v . s \in S_c \land e' \in Events(Spec_t) \\ \land validParam(v, s, e', Rel) \land G_{e'}(v, s) \\ \Rightarrow (\exists e . e \in Events(Spec_o) \land e' \mapsto e \in Rel \land G_e(v, s)))$$

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.

$$SameReachability(Spec_t, Spec_o, Rel) \\ \triangleq (\forall s, t, e, v . s, t \in S_c \land e \in Events(Spec_o) \\ \land validParam(v, s, e, Rel) \land s \xrightarrow{e(v)} t \\ \Rightarrow (\exists e' . e' \in Events(Spec_t) \land e' \mapsto e \in Rel \land s \xrightarrow{e'(v)} t)) \\ \land (\forall s, t, e', v . s, t \in S_c \land e' \in Events(Spec_t) \\ \land validParam(v, s, e', Rel) \land s \xrightarrow{e'(v)} t \\ \Rightarrow (\exists e . e \in Events(Spec_o) \land e' \mapsto e \in Rel \land s \xrightarrow{e(v)} t))$$

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

$$SameClosure(Spec_t, Spec_o, Rel) \\ \triangleq \forall s, t, e, v . s \in S_c \land t \in S_o \land e \in Events(Spec_o) \\ \land validParam(v, s, e, Rel) \land G_e(v, s) \land s \xrightarrow{e(v)} t \Rightarrow t \in S_c$$

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

$$SameEnabledness(Spec_t, Spec_o, Rel) \\ \land SameReachability(Spec_t, Spec_o, Rel) \\ \land SameClosure(Spec_t, Spec_o, Rel) \\ \Rightarrow Spec_t \overset{B}{\sim}_{|Rel} Spec_o$$

Proof Let $Spec_o$ be the original specification and $Spec_t$ be the transformed specification. Let *Rel* be the relation between these specifications. Let $B_t = Behavior(Spec_t)$ and $B_o = Behavior(Spec_o)$. Let $b_t, b_o . b_t \in B_t \land b_o \in B_o$. Now if $SameEnabledness(Spec_o, Spec_t, Rel) \land Same Reachability(Spec_o, Spec_t, Rel) \Rightarrow \exists B_c . b_t, b_o \in B_c$

Same enabledness and reachability means specifications share behaviors. However, some events may lead to non-shared states; therefore, we take closure to consider only the shared states of both specifications, i.e.,

$$\forall s, t, e, v . s \in S_c \land t \in S_o \land e \in Events(Spec_o) \\ \land validParam(v, s, e, Rel) \land G_e(v, s) \land s \xrightarrow{e(v)} t \Rightarrow t \in S_c$$

If the specification has also the same closure (i.e., no transition leads to a non-shared state) in addition to the same enabledness and reachability (shared behaviors), then the specifications are behaviorally equivalent, i.e., any behavior which is observed in the transformed specification would also be observed in the original specification, if it was animatable.

Therefore,

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

□

6 Demonstration of the approach on case studies

We applied the VTA framework to assess the correctness of three specifications, all written in the Event-B specification language [16]. The Event-B method is an offspring of the B method [15] and is designed for system-level modeling and analysis of large reactive systems. It uses set-theory and first-order logic as the specification notation. It also uses the notions of refinements (to represent systems at different levels of abstraction) and theorem proving (to prove the consistency between various refinement levels). Development of Event-B specifications is supported by the RODIN platform [31]. For animation purposes, we used the valuation-based animators Brama [29] and AnimB.[1]

An Event-B specification is composed of *Contexts* which specify the static part of the requirements model and *Machines* which specify the dynamic part of the model. The refinement relation is called *refinement* between machines, and *extension* between contexts. All machines have a special event, INITIALISATION, which specifies the initial state.

The first case study is about a land transport domain model [32, 33]. The second case study is about the landing system of an aircraft [34]. The third case study is about a platooning system [35]. All case studies are available at http://dedale.loria.fr.

The VTA framework explicitly requires all specifications to be proven before proceeding with their animation. The specifications are then animated by creating reasonable behavioral scenarios representing the protocols that would have been expected to occur in reality. The animators are provided with start-up values accordingly.

Not all refinements need to be animated. Some refinements based on small incremental steps are uninteresting from the animation's point of view because they do not bring much information in terms of new behaviors. At least one refinement per observation level was subjected to animation. An interesting point to note is that an Event-B refinement of a specification may introduce uses of non-animatable constructs, which would then need to be transformed to (re)achieve animatability.

The result of the application of heuristics is an animatable specification. In the following, the application of heuristics on formal specifications is presented in a before-after state clearly indicating how the specification has been transformed. When necessary, the application of heuristics is justified in the form of a formal proof.

6.1 Case study 1: the land transport domain model

The specification in this case study is about modeling of the land transportation domain. The term "transportation" refers to the movement of people or goods by vehicles from one location to another. Many important transportation concepts, such as vehicles, hubs (stations, junctions), connections (paths, routes) and movement, appear in this definition of transportation. They must be defined in the domain description. In the specification, we also express properties that any system working within the domain is expected to meet and maintain.

In this specification effort, the focus is on the formal definition of domain's laws, protocols and properties, rather than on the implementation of a particular system. Refinement is used to introduce new notions; the POs serve to guarantee the consistency of the model.

The domain model, shown in Fig. 3, contains one abstract machine `Movement0` and its seven refinements. All machines of the model are shown by green blocks. In parallel with machines, two contexts are being refined. The first is the context `Net`, which models the static properties of the network (its topology, quantities associated with its elements, etc.). The second is the context `StartState` which helps to set and prove the `INITIALISATION` event of the machines. The contexts of the model are shown by blue blocks. Extension between contexts and refinement between machines are shown by single arrow lines, whereas the use of contexts by machines is shown by dashed lines.

The development is structured into four different observation levels. The abstract model, the first two refinements and the fifth refinement sit at the first observation level, which defines the *travel* protocol that means a vehicle can move between two distinctive geographical points (hubs). Though technically realized as the refinement of `Movement4`, the fifth refinement step is logically situated at the first level of observation; it introduces time and concerns only the events at the first level. The third refinement belongs to the second level of observation, which decomposes the travel protocol into further two sub-

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

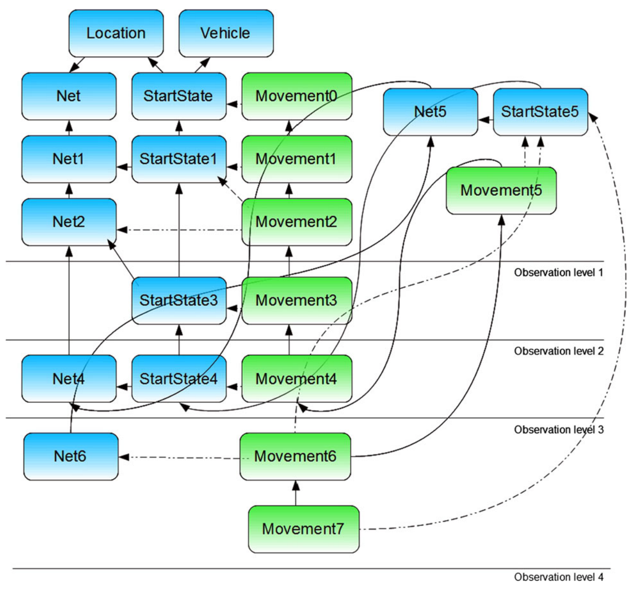

Fig. 3 Event-B model of the land transport domain [25]

protocols *crossing hubs* and *traversing paths*. The fourth refinement belongs to the third observation level, which decomposes the protocol of *crossing a hub* into further sub-protocols of *entrance in a hub*, *leaving a path* and *waiting to enter in a hub*. The sixth and seventh refinements model the fourth observation level, which decomposes the protocol of *traversing a path* into further sub-protocols of *wait to enter on a path*, *leaving a hub*, *moving on a path* and *waiting to move on a path*. Machine `Movement7` completes the introduction of time into the model and concerns the events and situations at this level.

This specification exhibits several properties which call for animation as the mean to check their validity, namely:

- complex data with behavioral constraints (following a route, for instance),
- protocols and iterations (travel as a sequence of hub crossing and path traversing protocols, for instance), and
- unplanned interaction between elements (autonomous vehicles, for instance).

The second refinement of the model introduces the notion of routes in the context `Net2` as shown by the left-hand side of Fig. 4. The constant `routes` is a set of sequences of `paths`; a `path` is an edge in the graph between two `hubs` (stations) which are the vertices. The set of routes is introduced as follows:

$$seqPaths = \{seq | \exists n . n \in \mathbb{N}_1 \land seq \in 1 \ldots n \rightarrowtail paths \land finite(seq) \land card(seq) = n\}$$

As sequence is not a primitive data type in Event-B data structure, we must provide its definition. This definition uses double quantification which the employed animator was unable to support when we tried to animate the model. To make the model animatable, we employ Heuristic 1 to transform the axiom to use the following superset of its expression:

$$seqPaths \in \mathbb{P}(\mathbb{N} \nrightarrow paths)$$

Since the type information of seqPaths has been changed, the model properties `pro1` and `pro2` (see the left-hand side of Fig. 4) expressed in terms of the original type information may no longer hold. Actually, these properties state that valid origin and destination hubs of a route are stations (and not junctions), both hubs belong to the same network, both hubs are connected to each other, and both hubs forbid cyclic connections (it is a domain restriction to avoid infinite circular paths). The properties use functions defined in previous refinements, such as `connectionOrigin/Destination` and `obsNetHubs`, which provide the connections and the hubs of a network, respectively. Both `pro1` and `pro2` are removed. Hence, the specification is now animatable. Figure 4 shows the context `Net2` before and after the application of Heuristic 1.

The most important effect of the application of Heuristic 1 is the invalidation of all proofs, either in `Net2` or in `Movement2` and their subsequent refinements, which relied on the essential property of sequences:

$\forall s . s \in seqPaths \Rightarrow dom(s) = 1..card(s)$

Proof of application of Heuristic 1 Animation requires us to provide actual values for `seqPaths`. Since `seqPaths` is a constant, we just need to ensure that the actual values conform to the axioms of the original `Net2`. Then, since the `Movement2` machine is verified, we are guaranteed that animation will only reach shared legal states. □

6.2 Case study 2: the landing gear system

The second case study deals with the specification of a landing gear system (LGS) of an aircraft. The LGS is in charge of maneuvering landing gears and associated doors. The LGS is composed of three landing sets: front, left and right. Each landing set contains a door, a landing gear and associated hydraulic cylinders. The main parts of the LGS are as following:

1. a mechanical part that contains all the mechanical devices and the three landing sets,
2. a digital part including the control software and
3. a pilot interface.

The corresponding Event-B model specifies the pilot interface, the digital part and the mechanical and hydraulic

```
CONTEXT
  Net2
EXTENDS
  Net1
CONSTANTS
  paths, routes, isRoute, seqPaths
AXIOMS
  typ1 paths ⊆ Connections
  typ2 seqPaths = { seq | ∃ n . n ∈ N1 ∧ seq ∈ 1..n ⤖ paths ∧
         finite (seq) ∧ card(seq) = n}
  typ3 isRoute ∈ seqPaths → 𝔹
  typ4 routes = {sp | sp ∈ seqPaths ∧ isRoute(sp) = TRUE}

  pro1 ∀r.r∈seqPaths ∧
        ((connectionOrigin(r(1)) ∈ stations ∧
        connectionDestination(r(card(r ))) ∈stations∧
        (obsNetHubs[{connectionOrigin(r(1))}] ∩
        obsNetHubs[{connectionDestination(r(card(r)))}] ≠ ∅) ∧
        (∀i. i∈2..card(r) ∧ connectionDestination(r(i−1)) = connectionOrigin(r(i )))
        ∧ connectionOrigin(r(1))≠ connectionDestination(r(card(r))) ∧
        (∀i1,i2 . i1∈1..card(r) ∧ i2∈1..card(r) ∧ i1 ≠ i2 ⇒
        connectionOrigin(r(i1 )) ≠ connectionOrigin(r(i2 )))
        ∧ (∀ i1 , i2 . i1∈1..card(r) ∧ i2∈1..card(r) ∧ i1≠ i2
        ⇒ connectionDestination(r(i1 )) ≠
        connectionDestination(r(i2 )))) ⇔ isRoute(r) = TRUE)
  pro2 ∀c.c∈Connections ⇒
        (connectionDestination(c)∈stations ∧ connectionOrigin(c) ∈ stations ⇒
        (∃r.r∈routes ∧ connectionOrigin(c) = connectionOrigin(r(1)) ∧
        connectionDestination(c) = connectionDestination(r(card(r )))))
END
```

```
CONTEXT
  Net2
EXTENDS
  Net1
CONSTANTS
  paths, routes, isRoute, seqPaths
AXIOMS
  typ1 paths ⊆ Connections
  typ2 seqPaths ∈ ℙ(N ⇸ paths)

  typ3 isRoute ∈ seqPaths → 𝔹
  typ4 routes = {sp|sp∈seqPaths ∧ isRoute(sp) = TRUE}

END
```

Fig. 4 The context `Net2` before (*left*) and after (*right*) the application of Heuristic 1

parts of the system. Additionally, it describes the hardware (gears, doors, sensors, lights, electro-valve, etc.), the normal working of the hardware and software, and the safety properties (normal and emergency modes).

As shown by Fig. 5, the Event-B model of the landing gear system contains one abstract machine `LandingSystem` and its four refinements, all shown by green blocks. In parallel with machines, two contexts `ContextInit` and `Hardware` are also being refined. The former contains the information necessary to set and prove the `INITIALISATION` event of the machines. The latter contains the description of the hardware configuration and status, such as description of landing sets as front, left and right, and handle states as up and down. Additionally, the context `CockpitHardware` contains the description of the pilot interface and the context `Phase_Ident` contains the information regarding readings of the sensors. The contexts of the system are shown by blue blocks. Extension between contexts and refinement between machines are shown by single arrow lines, whereas the use of contexts by machines is depicted by dashed lines.

Figure 5 also shows three levels of observations. The abstract model sits at the first observation level that deals with the status of the plane: ready to land or fly. The first, second and third refinement of the model belongs to the second observation level, which deals with the movement of the mechanical elements of the landing gears (doors, legs, locks, etc.). The fourth refinement sits at the third observation level, which was introduced when we wanted to model the reading of the sensors.

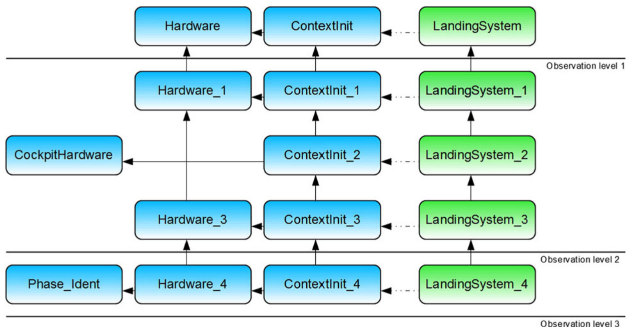

Fig. 5 Event-B model of the landing gear system [25]

An interesting feature of the LGS case study is a requirement that the maneuvers can be interrupted and reversed at any time. So, exercising the events which model the reversal is an important part of the validation. One such event, `restore_up`, introduced in the third refinement, `LandingSystem_3`, updates the related variables using the following pattern:

$$var :| var' \in LANDING\_SETS \rightarrow SENSOR\_OUTPUTS$$
$$\wedge (\forall g.g \in LANDING\_SETS) \Rightarrow (var'(g) = sfalse))$$
or
$$var :| var' \in LANDING\_SETS \rightarrow SENSOR\_OUTPUTS$$
$$\wedge (\forall g.g \in LANDING\_SETS) \Rightarrow (var'(g) = strue))$$

where *sfalse* and *strue* model the binary information sent by the sensors.

During animation, the animator fails to execute these substitutions due to its inability to dynamically map variables to each other. We then rewrote the actions using Heuristic 2 as following to achieve their execution.

$var := LANDING\_SETS \times \{sfalse\}$

or

$var := LANDING\_SETS \times \{strue\}$

Figure 6 shows the event `restore_up` before and after the application of Heuristic 2.

6.3 Case study 3: the platooning system

The third case study deals with the specification of a platooning system. Platooning is a mode of moving where vehicles are synchronized and follow one another closely. A platoon can be seen as a road train where cars are linked by software, instead of hardware. Platooning has several potential uses in an urban mobility system: augmenting throughput, herding unused cars to stations or running transient buses, for instance.

Several platooning control systems are being developed and experimented. The one developed at INRIA [36] and LORIA [37] is based on situated multi-agent (SMA) theory. In this system, each vehicle has its own local control algorithm which uses a perception/decision/action loop. The Event-B specification of the system is presented in [35, 38, 39]. In contrast to the first case study, the structure of the development in this case study can be interpreted as a sequence of refinements toward an implementation. Each refinement decomposes some events to make explicit a part of the general computation.

The Event-B model of the specification is presented by Fig. 7. The specification consists of five machines (four refinements):

- **Platoon**: defines platoons and sets the basic safety property. It contains only one event, `all_move`, where all vehicles change positions while keeping safe distance.
- **Platoon_1**: decomposes the event into one which moves the leader vehicle and one which moves the followers. This organizes the basic "iteration along the platoon" of each move.
- **Platoon_2**: computes the length of each basic move. This leads to the introduction of kinematic functions in the contexts and to the refinement of move events into several ones, each corresponding to a different situation (whether the maximum and minimum speeds are reached or not). This models the *action* part of the SMA.
- **Platoon_3**: introduces the notion of *decision* of the SMA model into the specification. Two events, one for the leader, and one for the followers, are introduced and integrated in the control loop.
- **Platoon_4**: introduces the notion of *perception* which allows decision events to be refined so the actual

```
  restore_up ≙
REFINES
  restore_up
WHERE
  grd1_all_gear_up gear_position = all_up
  grd2_nominal_mode operating_mode = normal
  grd3_abort_command continuation_mode = continue
  grd4_all_raised ∀g.g∈LANDING_SETS ⇒(gear_movement(g) = locked_up
              ∨ gear_movement(g) = stored_up)
  grd5_handle_down handle_state = handle_up
THEN
  act1_door_open door_open :| door_open' ∈ LANDING_SETS →
                 SENSOR_OUTPUT ∧ (∀g. g∈LANDING_SETS ⇒
                 (door_open'(g) = sfalse))
  act2_all_gears_up gear_position := all_up
  act3_all_stored gear_movement := {Front ↦ stored_up, Left ↦ stored_up,
                 Right ↦ stored_up}
  act4_normal_mode operating_mode := normal
  act5_continue continuation_mode := continue
  act6_light_maneuver_off light_maneuver := light_off
  act7_gear_extended gear_extended :| gear_extended' ∈ LANDING_SETS  →
                    SENSOR_OUTPUT ∧ (∀g. g∈LANDING_SETS ⇒
                    (gear_extended'(g) = sfalse))
  act8_gear_retracted gear_retracted :| gear_retracted' ∈ LANDING_SETS →
                    SENSOR_OUTPUT ∧ (∀g. g∈LANDING_SETS ⇒
                    (gear_retracted'(g)  = strue))
  act9_door_closed door_closed :| door_closed' ∈ LANDING_SETS →
                    SENSOR_OUTPUT ∧ (∀g. g∈LANDING_SETS ⇒
                    (door_closed'(g) = strue))
  act10_presurized circuit_presurized :| circuit_presurized ' ∈ SENSOR_OUTPUT
                 ∧ ( circuit_presurized '  = sfalse)
  act11_switch analog_switch :| analog_switch' ∈  SWITCH_POSITIONS ∧
             (analog_switch' =open)
END
```

```
  restore_up ≙
REFINES
  restore_up
WHERE
  grd1_all_gear_up gear_position = all_up
  grd2_nominal_mode operating_mode = normal
  grd3_abort_command continuation_mode = continue
  grd4_all_raised ∀g.g∈LANDING_SETS ⇒(gear_movement(g) = locked_up
              ∨ gear_movement(g) = stored_up)
  grd5_handle_down handle_state = handle_up
THEN
  act1_door_open gear_extended := LANDING_SETS × {sfalse}


  act2_all_gears_up gear_position := all_up
  act3_all_stored gear_movement := {Front ↦ stored_up, Left ↦ stored_up,
                 Right ↦ stored_up}
  act4_normal_mode operating_mode := normal
  act5_continue continuation_mode := continue
  act6_light_maneuver_off light_maneuver := light_off
  act7_gear_extended gear_extended := LANDING_SETS × {sfalse}


  act8_gear_retracted gear_retracted := LANDING_SETS × {strue}


  act9_door_closed door_closed := LANDING_SETS × {strue}


  act10_presurized circuit_presurized :| circuit_presurized ' ∈ SENSOR_OUTPUT
                 ∧ ( circuit_presurized '  = sfalse)
  act11_switch analog_switch :| analog_switch' ∈  SWITCH_POSITIONS ∧
             (analog_switch' = open)
END
```

Fig. 6 The event `restore_up` before (*left*) and after (*right*) the application of Heuristic 2

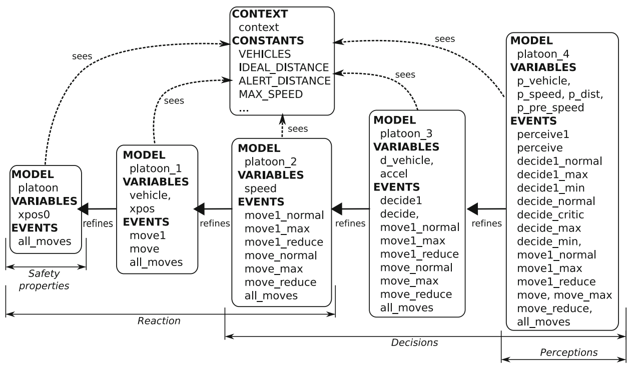

Fig. 7 The Event-B model of the platooning system [35]

computation of the parameters of the control law (acceleration) can be performed.

The result of the last refinement is very close to an implementation, in spirit if not in form. Nevertheless, we decided to use animation to validate the specification for the following reasons: to satisfy our curiosity regarding the heavy use of functions; to compare the results of the animation with the results of simulations that had been made previously; and to confirm the correctness of a certain "formal approximation."

The last reason is a consequence of using discrete tools to model what is inherently continuous. In this case, all POs were discharged, assuming one property, namely $x(y/z) = (xy)/z$, holds. True in $\mathbb{R}$, this property is false in $\mathbb{N}$. However, the difference becomes actually negligible when numerators are much bigger than denominators. Animation with realistic values gives insight on the validity of the "approximation" and on the solidity of the model.

The context of the model contains the notions of *speed* and *acceleration*. Several constants and axioms have been introduced into the context to help introducing the kinematics of a platooning system. The definition of the kinematics is comprised of complex mathematical functions and definitions which are non-animatable. Their non-animatability is primarily due to the complex definition of the functions. It does not allow the assignment of a single start-up value to the constant for animation. In fact, some of the functions are based on multiple definitions, each corresponding to a different case.

The first complexity arose in the refinement `Platoon_2` with the definition of the `new_xpos` function:

$$\forall xpos0, speed0, accel0.\\ ((xpos0 \in \mathbb{N} \land speed0 \in 0..MAX\_SPEED\\ \land accel0 \in MIN\_ACCEL..MAX\_ACCEL)\\ \Rightarrow (new\_xpos(xpos0 \mapsto speed0 \mapsto accel0)\\ = xpos0 + speed0 + (accel0/2)))$$

which models the kinematic law of computing a new position of a vehicle based on its acceleration and speed. It was used in some event guards in the following form and naturally could not be computed because actual values were required by the animators instead of calling a function in the context.

$$nxpos = new\_xpos(xpos(vehicle) \mapsto speed(vehicle)\\ \mapsto magic\_accel)$$

where *magic_accel* denotes a free variable for this refinement, which will be replaced by a state variable later in the development. Using Heuristic 3, we rewrote the guards as

$$nxpos = xpos(vehicle) + speed(vehicle)\\ + (magic\_accel/2))$$

Proof of application of Heuristic 3 The PO indicates that the $G_r \Rightarrow G$ must be proven.

$$nxpos = new\_xpos(xpos(vehicle)\\ \mapsto speed(vehicle) \mapsto magic\_accel) \quad (G)$$

The function *new_xpos* is defined as:

$$new\_xpos(xpos0 \mapsto speed0 \mapsto accel0)\\ = xpos0 + speed0 + (accel0/2)$$

Inlining the definition of function into G with the corresponding local variables:

$$nxpos = xpos(vehicle) + speed(vehicle)\\ + (magic\_accel/2))\\ (G_r)$$

Therefore, $G_r \Rightarrow G$. □

The most important complication came with another kinematic function, `new_xpos_max`, that calculates the position of a vehicle when its speed exceeds the maximum limit if acceleration continued through the entire time step. It is quite similar to `new_xpos`, except there is a case definition, i.e., either the particular vehicle is accelerating or not:

$$\forall xpos0, speed0, accel0.\\ ((xpos0 \in \mathbb{N} \land speed0 \in 0..MAX\_SPEED\\ \land accel0 \in MIN\_ACCEL..MAX\_ACCEL)\\ \Rightarrow (accel0 = 0 \Rightarrow new\_xpos\_max\\ (xpos0 \mapsto speed0 \mapsto accel0)\\ = xpos0 + MAX\_SPEED)\\ \land (accel \neq 0 \Rightarrow new\_xpos\_max\\ (xpos0 \mapsto speed0 \mapsto accel0)\\ = xpos0 + MAX\_SPEED\\ - (((MAX\_SPEED - speed0)\\ * (MAX\_SPEED - speed0))/(2 * accel0))))$$

```
  move1_max ≙
REFINES
  move1
ANY
  magic_accel, nspeed, nxpos
WHERE
  grd1 vehicle  = 1
  grd2 magic_accel ∈ MIN_ACCEL..MAX_ACCEL
  grd3 nspeed = new_speed(speed(vehicle)↦magic_accel)
  grd4 nspeed > MAX_SPEED
  grd5 nxpos = new_xpos_max(xpos(vehicle)↦
         speed(vehicle)↦magic_accel)
WITH
  var1 magic_xpos_vehicle = nxpos
THEN
  act1 vehicle  := vehicle+1
  act2 xpos(vehicle)  := nxpos
  act3 speed(vehicle) := MAX_SPEED
END
```

Fig. 8 The event `move1_max` before the application of Heuristics 3 and 4

The events using `new_xpos_max` function had to be duplicated (Heuristic 4), one with the guard `accel=0` and the other with its negation.

The prime example of such cases is the event `move1_max` which is shown by Fig. 8. The `grd3` of the original event calculates the new speed of a vehicle as:

$$nspeed = new\_speed(speed(vehicle) \mapsto magic\_accel)$$

The speed is then checked against the maximum allowed speed `grd4`, and consequently, a new position for the vehicle is determined in `grd5` as:

$$nxpos = new\_xpos\_max(xpos(vehicle) \mapsto speed(vehicle) \mapsto magic\_accel)$$

To solve the issue, the cases defined to calculate `new_xpos_max` are broken down into two events, each catering for one particular case. Figure 9 shows the transformed `move1_max` event.

The original context and the transformed context `Context_2` that specifies which functions have been relocated to machines are shown in Fig. 10.

Proof of application of Heuristic 4 The PO needs to be proved is

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

The non-animatable expression is the following:

$$nxpos = new\_xpos\_max(xpos(vehicle) \mapsto speed(vehicle) \mapsto magic\_accel)(G_e)$$

The function *new_xpos_max* is defined as:

$$\begin{aligned}
&\mathit{If}\ accel0 = 0\\
&\quad \Rightarrow new\_xpos\_max(xpos0 \mapsto speed0 \mapsto accel0)\\
&\quad = xpos0 + MAX\_SPEED\\
&\mathit{else\ if}\ accel0 \neq 0\\
&\quad \Rightarrow new\_xpos\_max(xpos0 \mapsto speed0 \mapsto accel0)\\
&\quad = xpos0 + MAX\_SPEED\\
&\qquad - (((MAX\_SPEED - speed0)\\
&\qquad * (MAX\_SPEED - speed0))/(2/accel0))
\end{aligned}$$

Inlining the definition of function into G_e while splitting it into G' and G''

```
  move1_max ≙
REFINES
  move1
ANY
  magic_accel, nspeed, nxpos
WHERE
  grd1 vehicle  = 1
  grd2 magic_accel ∈ MIN_ACCEL..MAX_ACCEL
  grd'  magic_accel ≠ 0
  grd3 nspeed = new_speed(speed(vehicle)↦ magic_accel)
  grd4 nspeed > MAX_SPEED
  grd5 nxpos = xpos(vehicle) +
           MAX_SPEED − (((MAX_SPEED −
           speed(vehicle))∗(MAX_SPEED −
           speed(vehicle))) / (2∗magic_accel))
WITH
  var1 magic_xpos_vehicle = nxpos
THEN
  act1 vehicle  := vehicle+1
  act2 xpos(vehicle)  := nxpos
  act3 speed(vehicle) := MAX_SPEED
END
```

```
  move1_max_zero ≙
REFINES
  move1
ANY
  magic_accel, nspeed, nxpos
WHERE
  grd1 vehicle  = 1
  grd2 magic_accel ∈ MIN_ACCEL..MAX_ACCEL
  grd''  magic_accel = 0
  grd3 nspeed = new_speed(speed(vehicle)↦ magic_accel)
  grd4 nspeed > MAX_SPEED
  grd5 nxpos = xpos(vehicle) + MAX_SPEED
WITH
  var1 magic_xpos_vehicle = nxpos
THEN
  act1 vehicle  := vehicle+1
  act2 xpos(vehicle)  := nxpos
  act3 speed(vehicle) := MAX_SPEED
END
```

Fig. 9 The event `move1_max` after the application of Heuristics 3 and 4

```
CONTEXT
  Context2
EXTENDS
  Context1
CONSTANTS
  MAX_SPEED, MIN_ACCEL, MAX_ACCEL, initial_speed, new_speed,
  new_xpos, new_xpos_max, new_xpos_min
AXIOMS
  typ01 MAX_SPEED ∈ N1
  typ02 MAX_ACCEL ∈ N1
  typ03 MIN_ACCEL ∈  INT

  pro01 MIN_ACCEL < 0
  pro02 initial_speed  ∈ 1..VEHICLES → 0..MAX_SPEED
  pro03 ∀ vehi0.(vehi0∈1..VEHICLES ⇒ (∃ speed0.
        (speed0 ∈ 0..MAX_SPEED ∧ initial_speed(vehi0) = speed0)))
  pro04 new_speed ∈ (0..MAX_SPEED X
        MIN_ACCEL..MAX_ACCEL) → INT
  pro05 ∀ speed1,accel1 .
        (speed1∈0..MAX_SPEED ∧ accel1∈
        MIN_ACCEL..MAX_ACCEL ⇒
        new_speed(speed1↦accel1) = speed1 + accel1)
  pro06 new_xpos ∈ (N X 0..MAX_SPEED X
        MIN_ACCEL..MAX_ACCEL)→ N
  pro07 ∀ xpos0,speed0,accel0 . ((xpos0 ∈ N ∧
        speed0 ∈ 0..MAX_SPEED ∧
        accel0 ∈ MIN_ACCEL..MAX_ACCEL) ⇒
        (new_xpos(xpos0↦speed0↦accel0) =
        xpos0 + speed0 + (accel0 / 2)))
  pro08 new_xpos_max ∈ N X 0..MAX_SPEED X
        MIN_ACCEL..MAX_ACCEL → N
  pro09 ∀ xpos0,speed0,accel0 . (xpos0 ∈ N
        ∧ speed0 ∈ 0..MAX_SPEED ∧
        accel0 ∈ MIN_ACCEL..MAX_ACCEL ⇒
        ((accel0 = 0 ⇒
        new_xpos_max(xpos0↦speed0↦accel0)
        = xpos0 + MAX_SPEED) ∧
        (accel0 ≠ 0 ⇒
        new_xpos_max(xpos0↦speed0↦accel0) =
        xpos0 + MAX_SPEED −
        ((( MAX_SPEED − speed0) ∗
        (MAX_SPEED−speed0))/(2∗accel0)))))
  pro10 new_xpos_min ∈ N X 0..MAX_SPEED X
        MIN_ACCEL..MAX_ACCEL → N
  pro11 ∀ xpos0,speed0,accel0 . (xpos0 ∈ N ∧
        speed0 ∈ 0..MAX_SPEED ∧
        accel0 ∈ MIN_ACCEL..MAX_ACCEL ⇒
        ((accel0 = 0 ⇒
        new_xpos_min(xpos0↦speed0↦accel0) =
        xpos0) ∧ (accel0 ≠ 0 ⇒
        new_xpos_min(xpos0↦speed0↦accel0) =
        xpos0 − ((speed0∗speed0) / (2∗accel0)))))
END
```

```
CONTEXT
  Context2
EXTENDS
  Context1
CONSTANTS
  MAX_SPEED, MIN_ACCEL, MAX_ACCEL,
   initial_speed ,
AXIOMS
  typ01 MAX_SPEED ∈ N1
  typ02 MAX_ACCEL ∈ N1
  typ03 MIN_ACCEL ∈ INT

  pro01 MIN_ACCEL < 0
  pro02 initial_speed  ∈ 1..VEHICLES → 0..MAX_SPEED
  pro03 ∀ vehi0.(vehi0∈1..VEHICLES ⇒
        (∃ speed0 . (speed0 ∈ 0..MAX_SPEED ∧ initial_speed(vehi0) = speed0)))

END
```

Fig. 10 The context Context_2 before (left) and after (right) the application of Heuristic 3

G' states:

$$grd'\ magic\_accel \neq 0$$
$$grd5\ nxpos = xpos(vehicle) + MAX\_SPEED$$
$$- (((MAX\_SPEED - speed(vehicle))$$
$$* (MAX\_SPEED - speed(vehicle)))/(2 * magic\_accel))$$

G'' states:

$$grd''\ magic\_accel = 0$$
$$grd5\ nxpos = xpos(vehicle) + MAX\_SPEED$$

Therefore, $G' \vee G'' \Rightarrow G_e(v)$. □

The major discovery from animating the specification was the presence of oscillation in the platoon, i.e., the propagation of a wave inside the platoon without stabilization. The last vehicles of the platoon frequently adjusted their acceleration while the ones in the front ran steadily. Animation's revelation of this undesirable feature showed the specification needed to be improved.

7 Evaluation of the animation process

Breuer et al. [40] listed three qualitative measures that can be used to evaluate any animation process. In addition to completeness, they mention coverage, i.e., how many language constructs are handled; efficiency, i.e., how quickly an animation process is performed; and sophistication, i.e., how many of the animation processes actually terminate.

In addition, [41] provides further criteria to strengthen the evaluation of an animation process, i.e., interactivity, transparency and operational equivalence. Interactivity is the idea that a user should be able to interact with the animator in order to perform better exploration of the specification. Transparency is directly related to the intermediate transformations that help achieve animations of specifications. Finally, operational equivalence of an animator is ensured when its performed operations are equivalent to the specification's operation, and not the operations of the refinements of the specification.

The VTA framework meets most of the stipulated criteria for a desirable animation process. As described in this paper, we are able to compensate for an animation tool's inability to execute specifications. For example, if a specification language construct is not supported by a tool, we promote its rewriting into an equivalent formula that not only extends its coverage but also contributes toward its efficiency and sophistication. Our heuristics that deal with the simplification of formulas, providing missing types, inlining function values, etc., also help achieve efficiency and sophistication.

VTA not only increases the interactivity of users with tools by proposing heuristics but with the help of provided semantics one can also reason about transparency of the proposed transformations. In some cases, transformations are identity functions (e.g., rewriting), so they are highly transparent. However, in case of unsupported elements where specifications need to undergo some structural reordering and optimizations, our proposed semantics provide a basis to argue about the soundness and, consequently, transparency of transformations.

It is not always possible to maintain the operational equivalence between the original and the transformed specification, for example, in case of refinement and approximation. In the transformation process, one can lose certain behaviors, for example, by restricting some inputs, but one cannot have additional behaviors such as new state transitions. We have, therefore, introduced the notion of "fidelity" which is looser than strict behavioral equivalence, but ensures that observations made on the transformed specification equate with the original specification.

8 Related work

The concept of specification animation is not a new one. Program visualizations have been previously used for designing, developing, monitoring and debugging software. Some notable visualization environments spanning across different areas of interest are graphics interface development [42], visualization of concurrent processes [43], etc.

Execution of specifications is a controversial issue. More than two decades ago, Hayes et al. [44] objected to the idea of specification execution. They argued that execution suppresses the expressiveness of a language and as far as specifications are concerned, the latter quality of a specification should be preferred over the former. In addition, they stated that executable specifications can negatively affect implementations.

In response to these concerns, [45] replied that it is the issue of correctness which is the major challenge in software development and not the expressiveness of specification languages. A technique like animation is, in fact, a very powerful method to ensure that specifications are validatable by customers as early as possible, thus minimizing the chances of software faults.

Our approach addresses both issues. Our rules help specifications achieve their animation, and at the same time, we ensure that they remain consistent. Our work can be seen as an extension of the approach presented in [46]. This work highlights the steps of converting a formal problem specification to a final program by applying semantics-preserving transformation rules.

9 Conclusion

We have presented an animation-based process for validation of formal requirements specifications. The idea of stepwise development is further enriched by a proposition of an auxiliary animation step associated with key refinements.

One limiting factor associated with the technique of animation is that not all specifications are animatable, at least, not directly. However, a specification can be transformed into a behaviorally equivalent animatable specification which may be unprovable. We have then proposed several transformations to realize this idea. Naturally, the validity of such a technique depends on semantics of the transformations. We have then developed a specific formal notion based on the behavior-preservation property of a model, to guarantee that the transformations can be trusted.

We have developed heuristics that guide the application of the proposed transformations to make a specification animatable. We have applied them in several case studies, each dealing with a specification written in the Event-B language. When the animators we were using were unable to deal with the form of the specification, our heuristics indicated which transformation(s) to apply. Application of the indicated transformations resulted in animatable specifications. Portions of these case studies have been illustrated in this paper, showing the before and after forms of the specifications, and the proofs confirming the correctness of the transformation applications. In our studies, we applied the transformations manually. Having now established the utility of this approach, we plan to mechanize our transformations.

Despite our transformation rules, animators may still fail to execute a specification, for instance when state spaces are still too big to be blindly explored. For the validation of such specifications, the technique of simulation [47], where users can safely replace nondeterministic expressions by computable expressions in the program generated from the specification, best suits the purpose. In future, we plan to extend the VTA framework also in this direction.

References

1. Heitmeyer CL, Jeffords RD, Labaw BG (1996) Automated consistency checking of requirements specifications. ACM Trans Softw Eng Methodol (TOSEM) 5(3):231–261
2. Kaufmann M, Moore JS (1996) ACL2: an industrial strength version of Nqthm. In: Proceedings of the eleventh annual conference on computer assurance (COMPASS-96)
3. Owre S, Rushby JM, Shankar N (jun 1992) PVS: a prototype verification system. In: Kapur D (ed) 11th international conference on automated deduction (CADE), ser. lecture notes in artificial intelligence, vol 607. Springer, Saratoga, pp 748–752
4. Gordon MJC, Melham TF (1993) Introduction to HOL: a theorem-proving environment for higher-order logic. Cambridge University Press, New York
5. Paulson LC (1994) Isabelle: a generic theorem prover, ser. lecture notes in computer science. Springer, Berlin
6. Beyer D, Henzinger TA, Jhala R, Majumdar R (2007) The software model checker BLAST: applications to software engineering. Int J Softw Tools Technol Transf 9(5):505–525
7. Cimatti A, Clarke E, Giunchiglia E, Giunchiglia F, Pistore M, Roveri M, Sebastiani R, Tacchella A (2002) NuSMV 2: an open source tool for symbolic model checking. In: Brinksma E, Larsen K (eds) Computer aided verification, ser. lecture notes in computer science, vol 2404. Springer, Berlin, pp 359–364
8. Hinton A, Kwiatkowska M, Norman G, Parker D (2006) PRISM: a tool for automatic verification of probabilistic systems. In: Hermanns H, Palsberg J (eds) Tools and algorithms for the construction and analysis of systems, ser. lecture notes in computer science, vol 3920. Springer, Berlin, pp 441–444
9. Holzmann GJ (1997) The model checker SPIN. IEEE Trans Softw Eng 23(5):279–295
10. Butler R, Caldwell J, Carreno V, Holloway C, Miner PS, Di Vito B (1995) NASA Langley's research and technology-transfer program in formal methods. In: Computer assurance, 1995. COMPASS '95. Systems integrity, software safety and process security. Proceedings of the tenth annual conference on June 1995, pp 135–149
11. Kaufmann M, Moore J (1997) An industrial strength theorem prover for a logic based on Common Lisp. Softw Eng IEEE Trans 23(4):203–213
12. Cimatti A (2001) Industrial applications of model checking. In: Cassez F, Jard C, Rozoy B, Ryan M (eds) Modeling and verification of parallel processes, ser. lecture notes in computer science, vol 2067. Springer, Berlin, pp 153–168. doi: 10.1007/3-540-45510-8_6
13. Bormann J, Lohse J, Payer M, Venzl G (1995) Model checking in industrial hardware design. In: Proceedings of the 32Nd annual ACM/IEEE design automation conference, ser. DAC '95. ACM, New York, pp 298–303. doi: 10.1145/217474.217545
14. Spivey JM (1988) Understanding Z: a specification language and its formal semantics. Cambridge University Press, New York
15. Abrial J-R (1996) The B book. Cambridge University Press, New York
16. Abrial J-R (2010) Modeling in event-B: system and software engineering. Cambridge University Press, New York
17. Farahbod R, Gervasi V, Glässer U (2007) CoreASM: an extensible ASM execution engine. Fundam Inf 77(1–2):71–103
18. Gargantini A, Riccobene E, Scandurra P (2008) Model-driven language engineering: the asmeta case study. In: Software engineering advances, 2008. ICSEA '08. The third international conference on, pp 373–378
19. Fitzgerald J, Larsen PG, Sahara S (2008) VDMTools: advances in support for formal modeling in VDM. ACM SIGPLAN Not 43(2):3–11. doi:10.1145/1361213.1361214
20. Leuschel M, Butler M (2008) ProB: an automated analysis toolset for the B method. J Softw Tools Technol Transf 10(2):185–203
21. Jackson D (2006) Software abstractions: logic, language, and analysis. The MIT Press, London
22. Mashkoor A, Jacquot J-P, Souquières J (2009) Transformation heuristics for formal requirements validation by animation. In: 2nd international workshop on the certification of safety-critical software controlled systems (SafeCert'09). York
23. Mashkoor A, Jacquot J-P (2011) Stepwise validation of formal specifications. In: 18th Asia-Pacific software engineering conference (APSEC'11). Ho Chi Minh City, Vietnam
24. Mashkoor A, Jacquot J-P (2011) Utilizing Event-B for domain engineering: a critical analysis. Requir Eng 16(3):191–207
25. Mashkoor A, Jacquot J-P (2015) Observation-level-driven formal modeling. In: High-assurance systems engineering (HASE), 2015 IEEE 16th international symposium, pp 158–165
26. Meyer B (1985) On formalism in specifications. Softw IEEE 2(1):6–26
27. Mashkoor A, Jacquot J-P (2011) Guidelines for formal domain modeling in Event-B. In: High-assurance systems engineering (HASE), 2011 IEEE 13th international symposium, pp 138–145
28. Cousot P, Cousot R (1977) 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. ACM, New York, pp 238–252
29. Servat T (2006) BRAMA: a new graphic animation tool for B models. In: B 2007: Formal specification and development in B. Springer, pp 274–276
30. Mashkoor A (2011) Formal domain engineering: from specification to validation. Ph.D. dissertation, Université de Lorraine. http://tel.archives-ouvertes.fr/tel-00614269/en/
31. Abrial J-R, Butler M, Hallerstede S, Hoang T, Mehta F, Voisin L (2010) Rodin: an open toolset for modelling and reasoning in Event-B. Int J Softw Tools Technol Transf 12(6):447–466
32. Mashkoor A, Jacquot J-P, Souquières J (2009) B Evénementiel pour la Modélisation du Domaine: application au transport. In: Approches formelles dans l'Assistance au Développement de Logiciels (AFADL'09). France, Toulouse, pp 1–19
33. Mashkoor A, Jacquot J-P (2010) Domain engineering with Event-B: some lessons we learned. In: Requirements engineering conference (RE), 2010 18th IEEE international, pp 252–261
34. Boniol F, Wiels V (2014) The landing gear system case study. In: Schewe K-D, Boniol F, Wiels V, Ait Ameur Y (eds) ABZ 2014: the landing gear case study, vol 433. Springer, New York, pp 1–18. doi:10.1007/978-3-319-07512-9_1
35. Lanoix A (2008) 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

36. Daviet P, Parent M (1996) Longitudinal and lateral servoing of vehicles in a platoon. In: Proceedings of the IEEE intelligent vehicles symposium, pp 41–46
37. Scheuer A, Simonin O, Charpillet F (2008) Safe longitudinal platoons of vehicles without communication. INRIA, research report RR-6741. http://hal.inria.fr/inria-00342719/en/
38. Colin S, Lanoix A, Kouchnarenko O, Souquières J (2008) Towards validating a platoon of cristal vehicles using CSP||B. In: Meseguer J, Rosu G (eds) 12th International Conference on Algebraic Methodology and Software Technology (AMAST 2008), ser. LNCS, vol 5140. Springer, pp 139–144
39. Colin S, Lanoix A, Kouchnarenko O, Souquières J (2008) Using CSP||B components: application to a platoon of vehicles. In: 13th International ERCIM Wokshop on Formal Methods for Industrial Critical Systems (FMICS 2008), ser. LNCS. Springer
40. Breuer P, Bowen J (1994) Towards correct executable semantics for Z. In: Bowen J, Hall J (eds) Z User Workshop, Cambridge 1994, ser. Workshops in computing. Springer, London, pp 185–209
41. Utting M (1995) Animating Z: interactivity, transparency and equivalence. In: Software engineering conference, 1995. Proceedings, 1995 Asia Pacific, pp 294–303
42. Clemons E, Greenfield A (1985) The sage system architecture: a system for the rapid development of graphics interfaces for decision support. IEEE Comput Gr Appl 5:38–50
43. Roman G-C, Cox KC (1989) A declarative approach to visualizing concurrent computations. Computer 22(10):25–36
44. Hayes I, Jones C (1989) Specifications are not (necessarily) executable. Softw Eng J 4:330–338
45. Fuchs NE (1992) Specifications are (preferably) executable. Softw Eng J 7:323–334
46. Partsch HA (1990) Specification and transformation of programs: a formal approach to software development. Springer, New York
47. Yang F, Jacquot J-P, Souquières J (2012) The case for using simulation to validate Event-B specifications. In: Proceedings of the 2012 19th Asia-Pacific software engineering conference-Volume 01, ser. APSEC'12. IEEE Computer Society, Washington, pp 85–90