Validation of Formal Specifications through Transformation and Animation

Atif Mashkoor, Jean-Pierre Jacquot

Validation of Formal Specifications through Transformation and Animation

Atif Mashkoor
Software Competence Center Hagenberg GmbH,
Hagenberg, Austria
firstname.lastname@scch.at

Jean-Pierre Jacquot
Université de Lorraine & LORIA,
Vandœuvre-lès-Nancy, France
firstname.lastname@loria.fr

***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**

I. Introduction

To be *correct,* a requirements document must be both complete and consistent. The former property concerns the fact that the document references all the 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 semi-formal 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 well-engineered 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], [11], [12], [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 available tools, particularly those which can execute a model, like CoreASM [17], Asmeta tool-set [18], VDMTools [19], or ProB [20] have limitations such as unsupported constructs, unbounded expressions, or purely implicitly defined functions and operations.

The tools which are most helpful to validate a specification rely on creating, running and evaluating scenarios on a formal model. Among those, animation is a process of executing a specification by invoking its operational semantics. It is mainly an automated process where an animator reveals the behavior of a formal specification either textually or graphically. This technique is similar to structure exploration technique [21] where a formal model containing collection of constraints is fed to an analyzer. The analyzer then explores the model by generating sample structures and check properties of the model by generating counterexamples. This technique of validation 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 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.

the process of transformation, the non-nimatable expressions are replaced by equivalent but animatable counterparts. Then the question is: are these transformations sound enough 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. Like an intractable proof can be broken down into a sequence of many smaller proofs, the validation of a specification can also be associated with its refinement-based development steps. This methodology then provides us with means to check the compliance of a formal specification, that in its initial form may not be animatable due to the inherent non-executable nature of its contents, to actual customers' requirements.

To determine the full-correctness of formal specifications, we employ the framework VTA [24] in which specifications are first checked for consistency and then animated. 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, the specification that contain non-animatable traits is transformed to achieve its execution in such a way that its behavior can be analyzed and reasoned about. The result of the whole correctness-assessment process is a specification which is both verified and validated.

We aim to reap benefits of this methodology in two ways. First, this methodology enables early detection of requirements problems (say, misunderstanding about a certain behavior). Second, 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 section II. Section III discusses the difference between classes of specifications. Section IV discusses how the class of a specification can be changed. Section V presents some transformational heuristics along with their semantics. Section VI demonstrates the application of transformations on three case studies. Section VII presents an evaluation of the proposed animation process. Section VIII provides some related work. Finally, the paper is concluded with some proposed future work.

II. VTA

VTA (Verify-Transform-Animate) is a framework for rigorous verification and validation of requirements specifications written in a formal refinement-based method. 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 Figure 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,
4) Validate the specification by its animation. If an unacceptable behavior is encountered, modify the requirements and restart from step one.

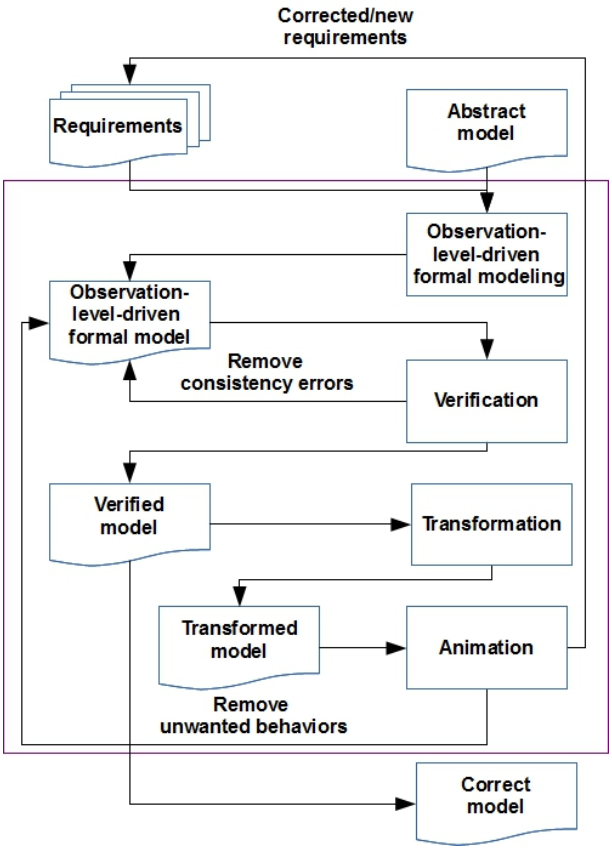

Fig. 1. The VTA framework: structure of a refinement step

A. 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 super-structure 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 break-down 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 to 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 emerging behavior, is of particular importance. Please see [25] for the detailed discussion.

B. 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 must be the starting step because there is no point in the validation of an inconsistent specification.

C. 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 section V).

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 if 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.

D. Animation

Once the transformations have been applied, the specification should now be animatable. Animation would demonstrate the behavior of the specification. 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., re-proving, 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 are out of scope of this paper. Rest of the paper will focus only on transformation and animation steps of the VTA framework.

III. 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 Figure 2.

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

Fig. 2. Classes of specifications

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 of the program's execution can provide developers with precious insights later contributing towards the correct solution. Likewise, animation that reveals a specification to be invalid provides guidance to the developers on how the specification needs correction.

Some important ingredients, often found in formal specifications are among the list of constructs which render these specifications non-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 wo main reasons. The first reason is that the specifier should avoid overspecification [26]. The second reason concerns the refinement principles that encourage us to use liberally 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 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.

IV. 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 Figure 3) are the following.

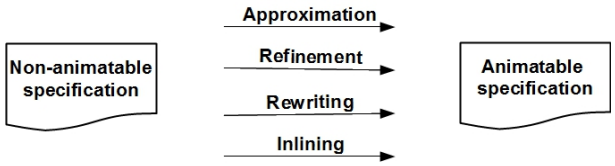

Fig. 3. Types of class changing techniques

A. Approximation

Approximation is a standard technique to modify a model so that the transformed model is not only close to the original model but also has better computational properties. For our purpose, we look for approximations which can be efficiently executed. 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 (over-approximate) specification then it holds in the concrete specification. However, if the property does not hold in the former, we do not know if the latter violates this property.

B. 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 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 original. 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.

C. 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 non-deterministically 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).

D. 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, failing 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 Section V-B) 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 intial 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.

V. 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 transformational heuristics ensure that behaviors observed during the animation of a transformed specification are specified in the original non-animatable specification, possibly at the expense of other formal properties such as provability. The correctness of heuristics and of their application then 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 Figure 4.

Heuristic pattern

Rationale:	What causes the problem, i.e., the logical basis of the animation failure
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

Fig. 4. The heuristic pattern

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 Section IV. **Caution** is the 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 the behavior of a model, 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." 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.

A. 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 tem. In the interest of brevity, in this paper we discuss 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 I contains the list of symbols used in the following sections.

Symbol	Meaning	Symbol	Meaning
$\mid$	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	$:\mid$	Becomes such that
$>$	Greater than	$\emptyset$	Empty set
$\mathbb{B}$	Boolean	$\times$	Cartesian product

TABLE I. THE SYMBOL TABLE

Heuristic 1: Generalize expressions involving complex iterations

This heuristics is motivated by the difficulty of iterating over complex nested predicated expressions. Such expressions come 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 super-set of the expression.

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

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

Caution: 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 as 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 an analytical expressions. The latter are defined as constants using axioms which can not 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.

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 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 decidability 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 iteration 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 adhered by the behavior of a specification. In the case of failure to be able to compute then, either they can be rewritten like heuristic 2 or can also be removed from the specification under the assumption that they already have 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, such as Adobe Flash, which has limited support for data structures. Our solution in this case is to transform the unsupported output values by external graphical environment into the supported ones.

B. 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.

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

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

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

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

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

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

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

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

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

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:

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

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:

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

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

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

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:

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

So, the formal definition of behavior preservation is:

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

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.

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

$SameReachability$ 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.

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

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

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

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:

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

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 $b_t, b_o . b_t \in B_t \wedge b_o \in B_o$. Now if $SameEnabledness(Spec_o, Spec_t, Rel) \wedge SameReachability(Spec_o, Spec_t, Rel) \Rightarrow \exists B_c . b_t, b_o \in B_c$

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.,

$$
\begin{aligned}
&\forall s, t, e, v.s \in S_c \wedge t \in S_o \wedge e \in Events(Spec_o) \\
&\wedge validParam(v, s, e, Rel) \wedge G_e(v, s) \wedge s \xrightarrow{e(v)} t \Rightarrow t \in S_c
\end{aligned}
$$

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.

Therefore,

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

■

VI. 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). Its development 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 observed in the reality. The animators are provided with startup values accordingly.

Not all refinements are 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 a specification may not be animatable while its refinement may be; there is no monotonicity in general.

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.

A. 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"

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

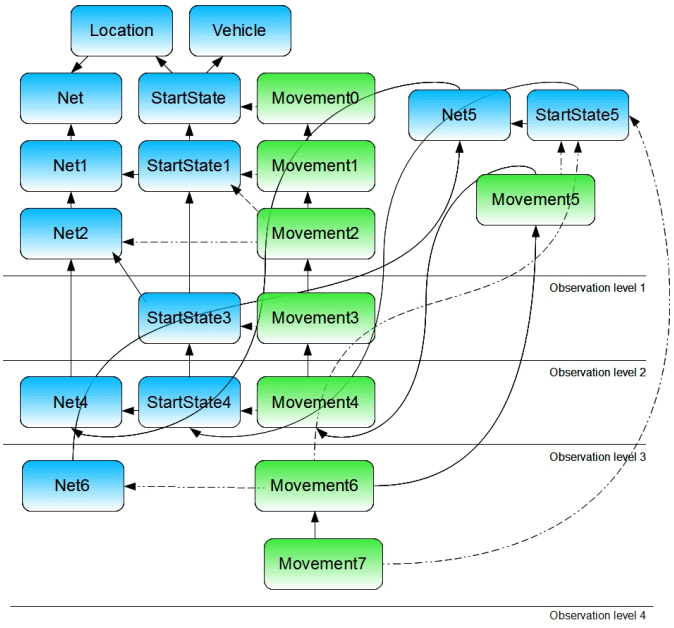

Fig. 5. Event-B model of the land transport domain [25]

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 proof obligations (POs) serve to guarantee the consistency of the model.

The domain model 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 depicted in Figure 5 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 that defines the *travel* protocol which 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 that decomposes the travel protocol into further two sub-protocols *crossing hubs* and *traversing paths*. The fourth refinement belongs to the third observation level that 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 that 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
- non-deterministic 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 Figure 6. 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:

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

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 Figure 6) 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 6 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:

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

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

Proof of application of Heuristic 1: Animation requires us to provide actual values for `seqPath`. Since `seqPath` 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.

■

B. 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 3 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,
3) and a pilot interface.

The corresponding Event-B model specifies the pilot interface, the digital part, and the mechanical and hydraulic 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 Figure 7, 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 7 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 that 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 that was introduced when we wanted to model the reading of the sensors.

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

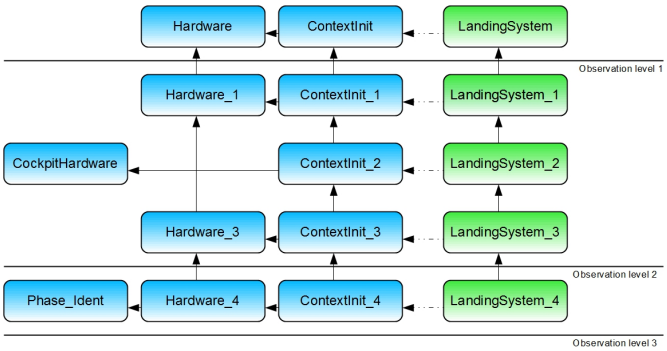

Fig. 7. Event-B model of the landing gear system [25]

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

C. 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. One locally developed is based on Situated Multi-Agent (SMA) theory. Each car has its own local control algorithm which uses a perception/decision/action loop; the platooning behavior is an emerging property [36], [37].

An Event-B specification of the local model has been written [35], [38], [39]. Contrary 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 Figure 9. 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 computation of the parameters of the control law (acceleration) can be performed.

Although the last refinement is very close to an implementation, in spirit if not in form, yet we decided to use animation to validate the specification for several reasons. The first was curiosity as the heavy use of functions was challenging, the second was to compare the results of the animation with the results of simulations that had been previously made, and the last was to confirm that a certain "formal approximation" was legitimate.

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

restore_up ≙
REFINES
restore_up
WHERE
grd1_all_gear_down 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_maneveur_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_down 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_maneveur_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. 8. The event `restore_up` before (left) and after (right) the application of Heuristic 2

animators instead of a 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 further on down 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.

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

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:

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

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 has already reached the maximum. It is quite similar to `new_xpos`, except there is a case definition,

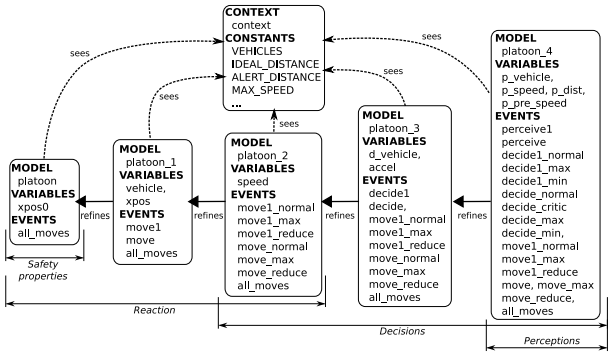

Fig. 9. The Event-B model of the platooning system [35]

i.e., either the particular vehicle is accelerating or not:

$$\begin{array}{l}\forall xpos0, speed0, accel0.\\ \quad ((xpos0 \in \mathbb{N} \wedge speed0 \in 0..MAX\_SPEED \wedge\\ \quad accel0 \in MIN\_ACCEL..MAX\_ACCEL) \Rightarrow\\ \quad\quad (accel0 = 0 \Rightarrow new\_xpos\_max\\ \quad\quad (xpos0 \mapsto speed0 \mapsto accel0) =\\ \quad\quad xpos0 + MAX\_SPEED) \wedge\\ \quad\quad (accel \neq 0 \Rightarrow new\_xpos\_max\\ \quad\quad (xpos0 \mapsto speed0 \mapsto accel0) =\\ \quad\quad xpos0 + MAX\_SPEED -\\ \quad\quad (((MAX\_SPEED - speed0) \times\\ \quad\quad (MAX\_SPEED - speed0))/(2/accel0))))\end{array}$$

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 Figure 11. The `guard3` 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 `guard4` and consequently a new position for the vehicle is determined in `guard5` 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 12 shows the transformed `move1_max` event.

The original and the transformed context `Context_2` that tells which functions have been relocated to machines are shown by Figure 10.

```
  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. 11. The event `move1_max` before the application of Heuristics 3 & 4

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:

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

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

G' states:

$$\begin{array}{c} grd'\ magic\_accel \neq 0 \\ grd5\ nxpos = xpos(vehicle) + MAX\_SPEED - \\ (((MAX\_SPEED - speed(vehicle)) * \\ (MAX\_SPEED - speed(vehicle)))/(2 * magic\_accel)) \end{array}$$

G'' states:

$$\begin{array}{c} grd''\ magic\_accel = 0 \\ grd5\ nxpos = xpos(vehicle) + MAX\_SPEED \end{array}$$

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

The major breakthrough of the animation activity was the discovery of oscillation in the platoon, i.e., the propagation of a wave inside the platoon without stabilization. The last vehicles of the platoon had to adjust their acceleration frequently while the ones in the front run smoothly. Animation shows that this specification needs to be improved on this account as this is an undesirable feature.

VII. 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 is 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, instead of its achieved refinements.

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 contribute towards 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, so they are highly transparent. However, in case of non-supported 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,

```
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

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 can not have additional behaviors such as new state transitions. We have, therefore, introduced the notion of fidelity which, once proved, ensures that observations made on the transformed specification equate with the original specification.

VIII. 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.

Executability 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 executability 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.

```
  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. 12. The event `move1_max` after the application of Heuristics 3 & 4

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.

IX. 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 (preferably) each refinement.

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 "downgraded" into a behaviorally-equivalent animatable specification. 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 of fidelity, based on the behavior-preservation property of a model, to guarantee that the transformations can be trusted.

Despite having transformation rules, animators may still fail to execute a specification. For the validation of such specifications, the technique of simulation [47], where users can safely complete the program generated form the specification, best suits the purpose. In future, we plan to extend the VTA framework in this direction. Implementation of the proposed heuristics in the form of a tool is also a future work.

References

[1] C. L. Heitmeyer, R. D. Jeffords, and B. G. Labaw, "Automated consistency checking of requirements specifications," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 5, no. 3, pp. 231–261, 1996.

[2] M. Kaufmann and J. S. Moore, "ACL2: An Industrial Strength Version of Nqthm," in *Proceedings of the Eleventh Annual Conference on Computer Assurance (COMPASS-96)*, 1996.

[3] S. Owre, J. M. Rushby, , and N. Shankar, "PVS: A prototype verification system," in *11th International Conference on Automated Deduction (CADE)*, ser. Lecture Notes in Artificial Intelligence, D. Kapur, Ed., vol. 607. Saratoga, NY: Springer-Verlag, jun 1992, pp. 748–752.

[4] M. J. C. Gordon and T. F. Melham, *Introduction to HOL: A Theorem-Proving Environment for Higher-Order Logic*. Cambridge University Press, 1993.

[5] L. C. Paulson, *Isabelle: a Generic Theorem Prover*, ser. Lecture Notes in Computer Science. Springer – Berlin, 1994.

[6] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar, "The Software Model Checker BLAST: Applications to Software Engineering," *Int. J. Softw. Tools Technol. Transf.*, vol. 9, no. 5, pp. 505–525, Oct. 2007.

[7] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella, "NuSMV 2: An OpenSource Tool for Symbolic Model Checking," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science, E. Brinksma and K. Larsen, Eds. Springer Berlin Heidelberg, 2002, vol. 2404, pp. 359–364.

[8] A. Hinton, M. Kwiatkowska, G. Norman, and D. Parker, "PRISM: A Tool for Automatic Verification of Probabilistic Systems," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. Lecture Notes in Computer Science, H. Hermanns and J. Palsberg, Eds. Springer Berlin Heidelberg, 2006, vol. 3920, pp. 441–444.

[9] G. J. Holzmann, "The Model Checker SPIN," *IEEE Trans. Software Engineering*, vol. 23, no. 5, pp. 279–295, 1997.

[10] R. Butler, J. Caldwell, V. Carreno, C. Holloway, P. S. Miner, and B. Di Vito, "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*, Jun 1995, pp. 135–149.

[11] M. Kaufmann and J. Moore, "An industrial strength theorem prover for a logic based on Common Lisp," *Software Engineering, IEEE Transactions on*, vol. 23, no. 4, pp. 203–213, Apr 1997.

[12] A. Cimatti, "Industrial applications of model checking," in *Modeling and Verification of Parallel Processes*, ser. Lecture Notes in Computer Science, F. Cassez, C. Jard, B. Rozoy, and M. Ryan, Eds. Springer Berlin Heidelberg, 2001, vol. 2067, pp. 153–168. [Online]. Available: http://dx.doi.org/10.1007/3-540-45510-8_6

[13] J. Bormann, J. Lohse, M. Payer, and G. Venzl, "Model checking in industrial hardware design," in *Proceedings of the 32Nd Annual ACM/IEEE Design Automation Conference*, ser. DAC '95. New York, NY, USA: ACM, 1995, pp. 298–303. [Online]. Available: http://doi.acm.org/10.1145/217474.217545

[14] J. M. Spivey, *Understanding Z: a specification language and its formal semantics*. Cambridge University Press, 1988.

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

[16] ——, *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, 2010.

[17] R. Farahbod, V. Gervasi, and U. Glässer, "CoreASM: An Extensible ASM Execution Engine," *Fundam. Inf.*, vol. 77, no. 1-2, pp. 71–103, Jan. 2007.

[18] A. Gargantini, E. Riccobene, and P. Scandurra, "A Metamodel-based Language and a Simulation Engine for Abstract State Machines," vol. 14, no. 12, pp. 1949–1983, jun 2008.

[19] J. Fitzgerald, P. G. Larsen, and S. Sahara, "VDMTools: Advances in Support for Formal Modeling in VDM," *SIGPLAN Not.*, vol. 43, no. 2, pp. 3–11, Feb. 2008. [Online]. Available: http://doi.acm.org/10.1145/1361213.1361214

[20] M. Leuschel and M. Butler, "ProB: An Automated Analysis Toolset for the B Method," *Journal Software Tools for Technology Transfer*, vol. 10, no. 2, pp. 185–203, 2008.

[21] D. Jackson, *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2006.

[22] A. Mashkoor, J.-P. Jacquot, and J. Souquières, "Transformation Heuristics for Formal Requirements Validation by Animation," in *2nd International Workshop on the Certification of Safety-Critical Software Controlled Systems (SafeCert'09)*, York, UK, 2009.

[23] A. Mashkoor and J.-P. Jacquot, "Stepwise validation of formal specifications," in *18th Asia-Pacific Software Engineering Conference (APSEC'11)*, Ho Chi Minh City, Vietnam, 2011.

[24] ——, "Utilizing Event-B for Domain Engineering: A Critical Analysis," *Requirements Engineering*, vol. 16, no. 3, pp. 191–207, 2011.

[25] ——, "Observation-Level-Driven Formal Modeling," in *High-Assurance Systems Engineering (HASE), 2015 IEEE 16th International Symposium on*, 2015, pp. 158–165.

[26] B. Meyer, "On formalism in specifications," *Software, IEEE*, vol. 2, no. 1, pp. 6–26, Jan 1985.

[27] A. Mashkoor and J.-P. Jacquot, "Guidelines for Formal Domain Modeling in Event-B," in *High-Assurance Systems Engineering (HASE), 2011 IEEE 13th International Symposium on*, 2011, pp. 138–145.

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

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

[30] A. Mashkoor, "Formal Domain Engineering: From Specification to Validation," Ph.D. dissertation, Université de Lorraine, Jul. 2011. [Online]. Available: http://tel.archives-ouvertes.fr/tel-00614269/en/

[31] J.-R. Abrial, M. Butler, S. Hallerstede, and L. Voisin, "An open extensible tool environment for Event-B," in *Proceedings of the 8th international conference on Formal Methods and Software Engineering*, ser. ICFEM'06. Berlin, Heidelberg: Springer-Verlag, 2006, pp. 588–605.

[32] A. Mashkoor, J.-P. Jacquot, and J. Souquières, "B Evénementiel pour la Modélisation du Domaine: Application au Transport," in *Approches Formelles dans l'Assistance au Développement de Logiciels (AFADL'09)*, Toulouse, France, 2009, pp. 1–19.

[33] A. Mashkoor and J.-P. Jacquot, "Domain Engineering with Event-B: Some Lessons We Learned," in *Requirements Engineering Conference (RE), 2010 18th IEEE International*, Sept 2010, pp. 252–261.

[34] F. Boniol and V. Wiels, "The landing gear system case study," in *ABZ 2014: The Landing Gear Case Study*, ser. Communications in Computer and Information Science, F. Boniol, V. Wiels, Y. Ait Ameur, and K.-D. Schewe, Eds. Springer International Publishing, 2014, vol. 433, pp. 1–18. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-07512-9_1

[35] 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.

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

[37] A. Scheuer, O. Simonin, and F. Charpillet, "Safe Longitudinal Platoons of Vehicles without Communication," INRIA, Research Report RR-6741, 2008. [Online]. Available: http://hal.inria.fr/inria-00342719/en/

[38] S. Colin, A. Lanoix, O. Kouchnarenko, and J. Souquières, "Towards Validating a Platoon of Cristal Vehicles using CSP||B," in *12th International Conference on Algebraic Methodology and Software Technology (AMAST 2008)*, ser. LNCS, J. Meseguer and G. Rosu, Eds., no. 5140. Springer-Verlag, Jul. 2008, pp. 139–144.

[39] ——, "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-Verlag, Sep. 2008.

[40] P. Breuer and J. Bowen, "Towards correct executable semantics for Z," in *Z User Workshop, Cambridge 1994*, ser. Workshops in Computing, J. Bowen and J. Hall, Eds. Springer London, 1994, pp. 185–209.

[41] M. Utting, "Animating Z: interactivity, transparency and equivalence," in *Software Engineering Conference, 1995. Proceedings., 1995 Asia Pacific*, 1995, pp. 294–303.

[42] E. Clemons and A. Greenfield, "The sage system architecture: A system for the rapid development of graphics interfaces for decision support," *IEEE Computer Graphics and Applications*, vol. 5, pp. 38–50, 1985.

[43] G.-C. Roman and K. C. Cox, "A declarative approach to visualizing concurrent computations," *Computer*, vol. 22, no. 10, pp. 25–36, 1989.

[44] I. Hayes and C. Jones, "Specifications are not (necessarily) executable," *Software Engineering Journal*, vol. 4, pp. 330–338, November 1989.

[45] N. E. Fuchs, "Specifications are (preferably) executable," *Software Engineering Journal*, vol. 7, pp. 323–334, September 1992.

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

[47] F. Yang, J.-P. Jacquot, and J. Souquières, "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. Washington, DC, USA: IEEE Computer Society, 2012, pp. 85–90.