The Case for Using Simulation to Validate Event-B Specifications

Faqing Yang, Jean-Pierre Jacquot, Jeanine Souquières

▶ To cite this version:

Faqing Yang, Jean-Pierre Jacquot, Jeanine Souquières. The Case for Using Simulation to Validate Event-B Specifications. APSEC2012 - The 19th Asia-Pacific Software Engineering Conference, The University of Hong Kong, Dec 2012, Hongkong, China. pp.85-90, 10.1109/APSEC.2012.66 . hal-00772812

HAL Id: hal-00772812
https://inria.hal.science/hal-00772812v1

Submitted on 11 Jan 2013

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

The Case for Using Simulation to Validate Event-B Specifications

Faqing Yang Jean-Pierre Jacquot Jeanine Souquières
LORIA – Université de Lorraine, France
Email: {firstname.lastname}@loria.fr

Abstract

This paper addresses the validation of formal specifications in Event-B through the execution of the specification. Current tools for Event-B, animators and translators, can execute only a restricted set of specifications. So, we propose a third technique, simulation, in which users and tools co-operate to produce an executable instance of the model. After a short presentation of Event-B and our simulation framework, JeB, we show how to use it on two reasonably complex specifications. Observations and analysis from the point of view of validation are presented and discussed.

1. Introduction

Asserting the correctness of a piece of software relies on two activities: verification–Have we built the piece right?–and validation–Have we built the right piece?– The strength of refinement-based methods such as B or Event-B [1] is to break down the huge task of formal verification into a sequence of manageable proofs. Each refinement generates small proof obligations (POs) and if new refinements are introduced only when all previous POs are discharged, the whole sequence produces a verified piece of software.

This procedure has two major advantages: first, it requires only "small" proofs to carry out the formal verification; second, errors are detected very close to the point where they are introduced. With the advent of usable support tools, such as Rodin[1], practitioners have far less reasons to shun formal methods. However, there is still the problem of validating the formal models. The verification process is too costly to allow for a trial-and-error kind of procedure.

We think that validation should be organized like verification: along the refinement chain. In principle, validation is simple: experts or potential users read the formal specification to judge if it represents a good model of the expected system. In practice, this does not work well since we, as humans, are not very good at reading and analyzing long mathematical texts. A better way is to look at executions of the specification and to judge whether the observed behaviours are consistent with the expected ones [2], [3]. The problem then becomes technical: how to execute the model?

Within the Rodin platform, several *animators* have been developed to execute Event-B specifications. Unfortunately, the class of directly animatable specifications is limited. We have shown how to safely transform specifications [4] to extend this class. However, many specifications are still non-animatable. To further extend the class of executable specifications, we propose to use *simulation,* i.e., the generation of a prototype of the model in a programming language.

JeB (JavaScript simulation framework for Event-B) is based on an observation and a simple idea. We observed that while some of our specifications are hard to animate, we could easily write programs to emulate them. The explanation is that using non-animatable features such as non-determinism and high abstractions in early refinements is recommended, even when we know how they will be reified. The idea was then to associate users to the generation of the simulation so their intelligence would assist the translator in the few cases where automatic solutions could not be applied.

This paper discusses the validity of the simulation approach for the validation of Event-B specifications. We analyze the use of JeB on two case studies. Both refer to the same problem: an algorithm for controlling a platoon of vehicles [5]. A simplified version in 1 Dimension [6] is used as a reference as it is easy to animate. The realistic version in 2 Dimensions [7] is hard to animate but is easy to simulate.

The following presents Event-B (Section 2) and related work (Section 3). After, Section 4 presents JeB with a particular emphasis on its design and its organization. Next, Section 5 details how to generate and set-

1. Rigorous Open Development Environment for Complex Systems: http://www.event-b.org

```
MACHINE platoon0
SEES context0
VARIABLES xpos0
INVARIANTS
typing        : xpos0 ∈ 1..VEHICLES → ℤ
non_collision : ∀ v . v ∈ 2..VEHICLES ⇒
          xpos0(v-1) - xpos0(v) > CRITICAL_DISTANCE
EVENTS
INITIALISATION
  BEGIN
  act1    : xpos0 := initial_xpos
  END
all_moves
  ANY
  magic_xpos
  WHERE
  typing  : magic_xpos ∈ 1..VEHICLES → ℤ
  spaced  : ∀ v . v ∈ 2..VEHICLES ⇒
   magic_xpos(v-1)-magic_xpos(v) > CRITICAL_DISTANCE
  THEN
  act1    : xpos0 := magic_xpos
  END
END
```

Figure 1: Event-B model of a platoon

up the simulations. Then, Section 6 analyses how the simulations can be used for validation. Last, Section 7 gives some research directions.

2. Event-B Language

Event-B is a formal framework to specifiy complex systems. It is a *state-based* method: a system is modeled as a mapping from names to values (a state) constrained by invariants. Events make the state evolve; they model behaviours. Event-B is a *formal* method: a model can be mathematically proven correct. The proofs concern mainly the preservation of the invariants. Event-B is a *refinement-based* method: a concrete implementation of an abstract model is derived through a sequence of refinements whose correctness can be proven. Event-B is supported by the Rodin environment which allows users to edit models, to generate the proof-obligations, to discharge the proofs, and to transform or animate the specifications.

Figure 1 shows an abstract Event-B model of a platooning system in 1D. It consists in a state with one variable (`xpos0`), an invariant with two predicates, and one event `all_moves`. `INITIALISATION` is a pseudo-event which describes the starting state of the model. The event `all_moves` has a parameter: `magic_xpos`. It expresses a guarded substitution on the state. Six POs are generated, the most important being the preservation of the `non_collision` predicate. In this present case, the proofs are easy.

An intuitive operational interpretation of an Event-B model consists in the repeated execution of a four step procedure: (1) to pick (or compute) and assign values to the events' parameters, (2) to compute the set of enabled events, (3) to choose one enabled event, and (4) to pick (or compute) and assign values to the substituted variables.

The model can be validated by observing the evolution of the state's values and the sequences of events fired during executions. Technically, assigning values to parameters, choosing an event to fire and picking the substituted values introduce non-determinism.

3. Related Work

Two kinds of techniques to execute Event-B models have been developed: animation and translation.

Brama [8], ProB [9] or AnimB[2] are *animators* which interpret directly the Event-B specification. As a main advantage, these tools have the shortest path between the specification and the observations of system behaviours. Their main limit comes from abstract types (Section 5.3.1) and non-determinism. The execution engine needs actual values for abstract types and for parameters of events. Brama and AnimB use enumerations of potential values; ProB uses constraint-solving techniques. Both strategies fail either through combinatorial explosion (complex, unconstrainted domains) or lack of value.

Animators have another limit: they often require changes in the formal text before its execution. Brama requires complex transformations [10]; ProB requires one refinement to give explicit values to constants if we want set up realistic animations. This preparation step is time-consuming; it is also a source for errors.

B2C [11] and B2ALL [12] are *translators*. They transform Event-B models into programs written in a mainstream language such as C or Java. On the one hand, translators provide us with a safe translation: the program implements the model. On the other hand, the translation can only be made on deterministic models. So, only the last refinement of the development can be executed using translators.

Validation requires a good visualization of the behaviour of the system. Animators provide us with API toward graphic systems, however, their limitation makes the construction of a nice display a complex and time-consuming activity. Programs generated by translators can be augmented with graphics by using standard graphic libraies. This is also time-consuming.

Our approach mixes the advantages of both approaches by producing *simulators*. From animators, we retain the general execution model and the management of non-deterministic features. From translators, we retain the generation of programs in a mainstream language. Our major contribution is to provide users

2. AnimB is available at http://www.animb.org

Requirement	Description
REQ-1	Easy and cheap building of simulations
REQ-2	Simulation consistent with model
REQ-3	Integrated graphic interface
REQ-4	Possibility to guide the generation
REQ-5	Generated code extendable by users
REQ-6	Full user-control on simulations
REQ-7	Easy building of ad-hoc graphics

Table 1: Requirements for JeB

with facilities to guide the translation (annotations), to provide hand-coded functions to generate non-deterministic values, and to set easily graphic displays.

4. JeB: a Simulation Framework

The JeB simulation framework is intended to complement animators and translators when they cannot be used. Table 1 summarizes the main requirements.

4.1. Design Philosophy

The model validation is a three-step process: (1) generate the simulator, (2) set up a particular simulation environment, including visualization, and (3) run the simulator.

We use JavaScript/HTML as the target language. Modern browsers provide us with the graphic-rich environment we need for our simulations. JavaScript was also chosen because of its technical features. Objects allow the JeB translator to produce code whose structure is close to the Event-B text structure. It is easy to instrument the code for specific observations. Prototypes allow users to provide their own functions to override the generated ones. We use this feature to separate the generated function stubs from the main simulator code; so, users need only to work with files separated from the main code.

Methodological and technical reasons account for why refinement-based methods promote a slow introduction of determinism and concrete data-structure during the development. Yet, specifiers have often clear ideas about possible implementations for abstract entities. JeB allows specifiers to add annotations in Event-B to help generate reasonably efficient code.

We consider that simulation should be a collaborative process between automated tools and humans. Specifiers, experts and users are involved at four levels: (1st) specifiers provide the annotations prior to the translation in the Event-B text, (2nd) specifiers provide hand-coded functions in the user configuration files, (3rd) experts may provide functions, typically for generating event parameters in different scenarios, and extra graphic visualizations, and (4th) users can interact during the simulation.

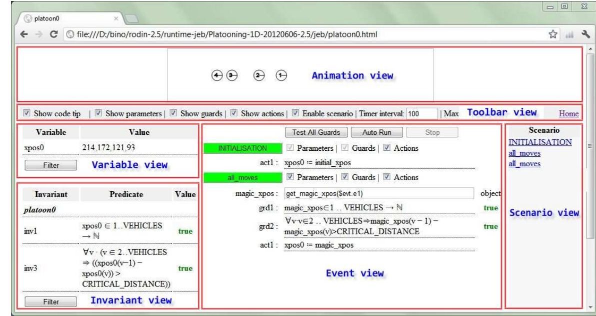

Figure 2: Simulation window

4.2. Simulator User Interface

The user interface of the simulator consists of six views (Figure 2.) The top-most area is a graphic, ad-hoc, view of the state. Below is a toolbar view to set up general parameters. The lowest area is organized in three columns. On the left, variables and invariants are shown. In the middle, all events, with their parameters, guards, actions and activation status, are displayed. On the right, the history of the simulation is kept as a sequence of events and parameters.

5. Creation of Simulations

5.1. Event-B Specification of Platooning

JeB was motivated by our studies on platooning where we aimed at proving that some control algorithms are safe. The safety condition we retained is the absence of collision between a vehicule and its predecessor. This definition of “safe” is of course a simplified version of what would be needed for moving on actual roads, but it is highly representative.

We developed in Event-B a local version of a well known platooning algorithm [5]. It uses only perceptions of the preceding vehicle to take decisions, hence, it is very robust. The hypothesis that longitudinal (i.e., speed) and lateral (i.e., turning) controls are independant prompted us to develop two models. The first model considers only longitudinal control, as if vehicles move on a rail. It allowed us to study the general structure of the specification, within and between refinements, and to identify critical issues. In the second, we develop the more realistic bi-dimensional model.

Both models have the same structure which consists of an abstract machine and four refinements. Each refinement introduces a clearly identified concept.

As ProB and Brama can animate the 1D model, it serves us as a benchmark. As animators fail on the 2D model, it serves as a test-bed for assessing the feasability of the simulation approach [3].

5.2. 1D Simulations

The JeB translator is applied as the level of Rodin projects. Each machine, either abstract or refined, and each context is translated into a file in a common directory, "`<Rodin wkp/<Ev-B project>/jeb`". A simple naming scheme facilitates the navigation in the simulator code.

5.2.1. Minimal Simulation. After the simulator of the most abstract machine `platoon0` has been generated, we must provide some values for the abstract constants. This can be done in one of four ways:

1) annotate the constant in the specification text with `@value=<integer>;` before running the JeB translator,
2) set the value in the `jeb.user.js` file after running the JeB translator,
3) set the value in the `<context>.js` file,
4) set the value in the `<machine>.user.js` file.

With (1) and (2) the values of constants are permanently kept. With (3) and (4), the values need to be set again after each run of the translator since new instances of the files `<context>.js` and `<machine>.user.js` are then created. With (1), (2) and (3) the values are available to all machines while, with (4), they are restricted to only one.

By using the second technique, we add the following two lines in the `jeb.user.js` file (the prefix `$ctx` denotes constants):

```
$ctx.VEHICLES = 4;
$ctx.CRITICAL_DISTANCE = 20;
```

The button `INITIALISATION` turns then green, indicating an enabled event. The next step is to provide an implementation for the `magic_xpos` parameter. Events' parameters model either true parameters or local variables in the usual programming sense. JeB translates each true parameter par_i as a function get_par_i. By default, those functions are called at the beginning of each simulation cycle. Since parameters are highly non-deterministic, the JeB translator generates only function stubs that should be replaced in the `<machine>.user.js` file, for instance:

```
// TODO Auto-generated function stub:
// argument generator
var get_magic_xpos = function(event) {~};
```

3. Simulators are accessible at http://dedale.loria.fr/?q=en/JeB

Assuming we replace this stub by a probabilistic value generator, we can put the simulation in `Auto Run` mode and observe how the state evolves. Another possibility is simply to enter values manually at each simulation cycle. We could then drive the simulation toward certain specific configurations. We can switch from one mode to the other at any time during execution of the simulation.

5.2.2. Graphic Display. The JeB simulator includes an HTML5 `canvas` to display graphically the state. Users can provide two functions: `jeb.animator.init` which initializes the display, and `jeb.animator.draw` which draws an image of the system state at each simulation cycle.

5.2.3. Simulation of the Refinements. All refinements in the 1D model can be executed with JeB. The work required on each refinement is similar to what is presented above and summed up in Table 2, with the number of entities, constants or functions provided by the user. It should be noted that the functions for the graphic animation are defined only once for all simulations.

5.3. 2D Simulations

The 1D and 2D models share the same structure, but not the same complexity. Animators cannot execute the 2D model because of two features: abstract or "too big" carrier set, and functions defined by properties. The following analysis explain how simulation provides users with possibilities to execute the models.

5.3.1. Carrier Sets. The 2D specification requires the modeling of a notion of *space*. For the most abstract refinements, we used an abstract carrier set, `Point`, as we did not want to commit specific characteristics too early in the specification. Further down the development, `Point` will be refined into a kinematic space with 6 dimensions: $(x, y, \gamma^\theta, \sigma^\theta, v, \kappa)$ representing the geometric position, the orientation, the velocity and the trajectory's curvature. Animators cannot execute with either symbolic `Points` or 6-uplets. The former cannot be instanciated by meaningful enumerable values; the latter leads to combinatorial explosion. Yet, implementing `Point` as objects with six access functions is simple and sufficient for simulation.

5.3.2. Functions Defined by Properties. The 2D specification uses several functions associated with the notion of distance, such as the distance between two vehicles, the deviation from the trajectory, or

Model		1D	2D
Platoon0	functions	1	4
	constants	2	4
	sets		1
Platoon1	functions	1	1
	constants		
Platoon2	functions		7
	constants	3	6
Platoon3	functions		
	constants		
Platoon4	functions		
	constants	1	3

Table 2: Number of required user-defined entities

the closest point of a curve from a position for instance. A computational definition of thoses function is not needed until very concrete refinements have been reached. It may even be impossible to give as it depends on the actual geometry of the vehicles. We only need some standard properties of measure functions at the abstract levels.

Of course, executions require a computational definition, which is actually quite straightforward using euclidean distances. An interesting note is that, depending on the implementation of trajectories, the definition of some distances may vary and generate subtly different behaviours. JeB then allows us to test several implementations before commiting to a refinement.

Table 2 presents the numbers of entities that we were required to hand-code to run the simulations. As can be seen, there was not much to do. The validation of each refinement does not require a heavy investment.

6. Exploitation of Simulations

To be useful as a validation tool, JeB must allow users to detect anomalous behaviours or to experiment with some. Here are presented some observations on the simulations of our 2 models.

6.1. Observations

During its development, the 1D specification went through several stages where the model was correct but specified unintended behaviours. We checked that the use of JeB could reveal some of the "problems."

An early version allowed vehicles to move backward, contrary to an implicit assumption. This undesirable behaviour is highly visible on the graphic display in auto-run mode.

Although fully proven, the version published in [6] contains a deadlock. In some circonstances, no event is enable and the execution halts. Implementations of this model show the vehicles colliding. With the simulation produced with JeB, we could easily provoke and analyze the deadlock situations. As all refinements can be executed, locating the introduction of the error in `platoon2` was straightforward.

Sometimes, the speed of vehicles oscillates around the average platoon's speed. This is an emergent, undesired, behaviour. Oscillations are quite visible on the graphic display; their apparition can be finely analyzed.

When executing the 2D specification we concentrate observations on a few questions. The most important is the deadlock issue. Thorough executions of `platoon2` make us confident that the model is deadlock free. The study of oscillations is also possible. Speed oscillations are similar in 1D and 2D models; lateral oscillations are new. As already mentioned, setting up the observations for lateral oscillations raised many interesting questions about the computational definition of "closest point" or of "distance."

6.2. Analysis from a Validation Point of View

The 1D specification consists of 15 events and contains around 130 individual logical formulas. Many of those formulas are very simple as they express some typing property. Yet, getting the specification right was a difficult task. In this section, we discuss how JeB would have helped in this task.

The backward movement problem was found by a picky human reader. Platooning systems with backward moves may be designed but at the probable expense of higher complexity. To use Brama to animate the model, we had to wait for `platoon4` to be defined, yet the correction is better done at the most abstract model. JeB can execute this model, and so, would have helped detect the issue earlier.

The issue of deadlock-freeness in Event-B specifications is a complex one. The standard POs do not protect from deadlocks. It is possible to automatically build a theorem and POs which ensure the absence of deadlock [13]. However, the size of the formulae to prove grows very fast with the number of events and guards. Proving this theorem is highly time-consuming. Actually, we discovered the existence of deadlocks when we animated the concrete model with Brama. Since the problem is introduced two refinements earlier, and JeB can execute all refinements, we would have spared a lot of time if we had "fixed" the model then. Using simulation before proving a deadlock-freeness theorem makes sense.

The improvement offered by JeB on 1D model execution is mostly about cost and practicality. The effort to build the simulations of the different machines is

Elements	Manual	Total	Rate
1D Platooning			
sets	0	0	-
constants	6	15	40%
parameter functions	2	11	18%
animation functions	2	2	100%
code size (KB)	3	290	1%
2D Platooning			
sets	1	1	100%
constants	24	50	48%
parameter functions	2	43	5%
animation functions	2	2	100%
code size (KB)	7	890	1%
Pacemaker			
sets	1	1	100%
constants	15	15	100%
parameter functions	0	26	0%
code size (KB)	1	954	1%

Table 3: Summary of creating simulation effort

small. Furthermore, it is spread over all the refinements since large parts of user's code can be reused. The cost of using ProB on 1D model is low if we use it only for deadlocks detection, but it would sharply increase if we try to set up a graphic display. Because JeB is based on a rich graphic substrate, we could build a simple informative display in a few lines.

With the 2D model, only Jeb allows us to analyze the system's behaviours. The issue raised in the observations about the distance functions is actually a crucial one for the direction of next refinements. In practice, trajectories are not continuous lines, but sequences of points perceived through imperfect sensors. We can expect that the tracking behaviour of the vehicles is dependent of the actual computations of the distances. JeB allows us to experiment several practical algorithms before we commit them to the next refinement.

Table 3 gives the relative effort needed for creating the simulations on three models. It is worth noting that the amount of user-provided code is small: around 1%. Also, although the simulations of the refinements are independent, user-provided code can often be shared. This is particularly true of the argument generator functions. Preliminary observations on the simulation of the specification of a pacemaker developed outside of our group confirm the figures.

7. Conclusion and Future Work

JeB proves that the generation of simulations for validating Event-B models is feasible. We were able to use it successfully on several large models realized outside our group. JeB calls for news work on several direction. The efficiency of the simulations depends on the quality of the generation and of the run-time libraries. Efficient set-manipulation libraries and big integer lobraries are needed. The second direction is to provide safe-guards to users when they introduce hand-coded functions. It requires a formal of behaviours and observations. The last direction concerns the integration of validation activities into refinement-based methods. The issue is to connect the validation of each refinement with the validation of the final system.

References

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

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

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

[4] A. Mashkoor, J.-P. Jacquot, and J. Souquières, "Transformation Heuristics for Formal Requirements Validation by Animation," in *2nd IW SaFeSert*, York, UK, 2009.

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

[6] A. Lanoix, "Event-B Specification of a Situated Multi-Agent System: Study of a Platoon of Vehicles," in *2nd Int. Symposium on Theoretical Aspects of Software Engineering*, Nanjing, China, 2008.

[7] F. Yang and J.-P. Jacquot, "Scaling up with Event-B: A Case Study," in *The 3rd NASA Formal Methods Symposium*, LNCS, no. 6617. 2011, pp. 438–452.

[8] 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.

[9] M. Leuschel, J. Falampin, F. Fritz, and D. Plagge, "Automated Property Verification for Large Scale B Models with ProB," *FAC*, pp. 1–27, 2011.

[10] A. Mashkoor and J.-P. Jacquot, "Stepwise Validation of Formal Specifications," in *The 8th Asia-Pacific SEC*, Ho Chi Minh City, Vietnam, 2011.

[11] S. Wright, "Automatic Generation of C from Event-B," in *Workshop on Integration of Model-based Formal Methods and Tools*, 2009.

[12] D. Méry and N. Singh, "Automatic Code Generation from Event-B Models," in *Proc. Symposium on Information and Communication Technology*, Hanoi, Vietnam, 2011.

[13] F. Yang and J.-P. Jacquot, "An Event-B Plug-in for Creating Deadlock-Freeness Theorems," in The *14th Symposium on FM*, São Paulo, Brazil, 2011.