Proving the Fidelity of Simulations of Event-B Models

Faqing Yang, Jean-Pierre Jacquot, Jeanine Souquières

Proving the Fidelity of Simulations of Event-B Models

Faqing Yang, Jean-Pierre Jacquot and Jeanine Souquières

Université de Lorraine – LORIA (UMR 7503)
F-54506 Vandœuvre lès Nancy, France
Email: {firstname.lastname}@loria.fr

***Abstract*—A major hindrance to the use of formal methods is the difficulty to validate the models, particularly at the early stages of the development. We propose to build simulations: programs automatically generated from the specifications but with user-provided implementations of the non-executable traits of the models. We present such a simulation. Of course, the question of the *fidelity* of the simulation to the model is raised in such a setting. We provide a formal definition of *fidelity* and the proof obligations that can be attached to each hand-coded element so that *fidelity* can be proven.**

***Keywords*—Formal methods, Event-B, Validation, Simulation, JavaScript, Proof obligation**

I. Introduction

Among the techniques required to develop high assurance systems, refinement-based formal methods have a strong appeal. Their foundational idea is that it is possible to develop pieces of software which are *correct by construction.* So, when the code is delivered, there should be no need to test it for discovering anomalies or to check its conformity to the specification. This is made possible by languages such as B [1] and Event-B [2] with their environments which make this paradigm an effective choice for developers.

For those "development methods," building a software is constructing a sequence of formal models with the following properties:

1) the first model is an abstraction of the specification of the software,
2) the last model is a concrete model, i.e. a model fully deterministic and implementable on current machines,
3) each model is formally consistent and "correct," and
4) each model, except for the first, is formally tied to the previous one by a "refinement" relation and all ties can be proven "correct."

"Correct" is perfectly defined above. In case 3, it is the preservation of an invariant; in case 4, it is the existence of some abstraction function between the models with the property to preserve the abstract invariant when the concrete invariant is preserved. The language used for expressing the formal models is designed in such a way that the correctness properties can be cast into sets of small theorems. It is also designed so the proofs of the theorems can be conducted with current software provers.

The weak point of refinement-based methods is actually the first case above. There are two reasons for this weakness. A minor one is the difficult to go from informal requirements toward formal models. Works on requirements elicitation with formal goal models [3], on transforming goal models into specifications [4], or on formal domain modeling in [5] for instance, show that the formalization question can be answered. The major reason lies in the fact that requirements evolve during many projects [6]. Therefore, it is unlikely that the very first formal model will contain the complete and final specification.

The obvious solution is to cycle developments, including new requirements as they are discovered. However, in practice, such cycle can work only if the new requirements are discovered early and if they can be included smoothly into the chain of refinements. The worst case scenario is when the discovery happens on the coding phase and requires to restart all the development from the initial model.

In this paper, we defend the point of view that all models, or at least, most of them, should be *validated* as well as verified. Validation ensures that the model is a description of the software we really want. To achieve this, we need specific tools and theories.

A usual technique is to run the model. Tools such as animators, ProB [7] for instance, or translators, EB2ALL [8] for instance, allow to execute some Event-B models, but they fail on most because of their high level of non-determinism. We propose a tool JeB, a JavaScript simulation framework for Event-B. This tool generates automatically JavaScript programs, combining ideas from animators and translators, but it allows users to add hand-coded elements to supply deterministic solutions when needed. Our case-studies show that the manual part amounts to only 1% of the size of the simulation [9]. On the one hand, this strategy allows us to execute a lot of models, for a small cost. On the other hand, it raises some questions about the correctness of the observations made during the executions.

Our answer is to propose a formal notion, called *fidelity,* which defines precisely the form of correctness we desire. From this definition, we can derive what must be proven in order to ensure that what is observed on a simulation is what is specified in the model.

The paper is organized as follows. We first present Event-B and the refinement-based process it supports. We present then a case-study to motivate the necessity and the difficulty of validating early steps in a development. The simulation of the models is discussed after. Last, we present the notion of *fidelity*

and the proof obligations that can be generated to assess that a particular simulation is correct.

II. Event-B

A. Language

A specification in Event-B consists in two parts: a state and events. The state maps names to values; it is constrained by an invariant. For practical purposes, models are split into *Contexts* and *Machines* which describe respectively the constant and the variable parts of the state. The values are build inductively from integers, boolean and symbols by using power sets and cartesian products. Special cases such as binary relations, partial/total functions, or injections enjoy specific notations. The typing system is equated with the "belongs to" relation between a value and a set.

The invariant is a first-order formula on the state. It is constructed as the conjunction of smaller formulas called *axioms* and *invariants* in contexts and machines.

Events model the evolution of the state. Formally, they are guarded generalized substitutions. Guards are first-order formulas on the state; they may contain free variables called *event parameters.* Substitutions change some values, all at the same time. An event is *enabled* when its guard is true, it can then be fired. The substitution uses the parameters' values which made the guard true.

The formal semantics of an Event-B model is based on the *feasibility* (*fis*) property expressed as two ideas:

- a legal state with actual values must exist, and
- the firing of any enabled event from a legal state leads to a legal state.

The specification language is designed so *fis* can be cast as a set of small logical formulas, called *proof obligations*. A model is correct when all its POs have been discharged. POs can be automatically generated; current provers can discharge automatically most of them.

B. Refinement

Event-B embodies a development method based on the association of two ideas: programs are constructed through stepwise refinements, and programs can be *correct by construction.* In this context, *correct* means "mathematically proven to meet its specification."

Mathematically, the refinement is a relation between models. A model M_r is a *refinement* of a model M_a, if:

- M_r is consistent (i.e., it defines one actual state, invariants are preserved through the actions),
- there is an abstraction function from the M_r state and events to the M_a state and events,
- M_r legal states are abstracted as M_a legal states,
- all events fired in M_r preserve the invariant of M_a.

These properties generate POs. Discharging all POs of all refinements in a chain guarantees that the most concrete model preserves the invariant of the most abstract model.

These definitions are embedded in the framework at two levels. The language provides users with a syntax to express the refinement relationship (refines or extends) and the abstraction function (with clause). The support environment generates the POs.

In Event-B, refining is an activity which adds information into a model. Users can see a refinement as one of four kinds:

- adding new invariants; this reinforces the properties of interest and the constraint on the domain,
- adding new variables and constants unconnected to previous ones; this introduces new concepts and properties,
- adding new variables and constants to implement previous ones; this is the usual data refinement which reifies abstract data structures,
- adding new events; this splits an abstract "large" event into several concrete "smaller" events.

Of course, it is possible to build a refinement belonging to several kinds, but this is bad practice [10].

C. Operational Semantics

Event-B has an intuitively simple operational semantics. It is a cycle of three steps :

1) compute the set of enabled events,
2) pick one event and one value for each of its parameters,
3) apply the substitution on the state.

The cycle starts with the INITIALISATION event and ends when no event is enabled. Depending on the system modeled, stopping can be a desired feature (termination) or not (deadlock).

The operational interpretation of an Event-B model relies on non-determinism at three levels: free variables can take any value which makes the guards true, substituted values can be described non-constructively by a property ("any value such that"), and any enabled event can be fired. So, the execution of Event-B specifications requires tools that can deal with non-determinism. In practice, this means that combinatorial explosion must be tamed.

III. Case-study

A. Platooning

New urban transportation systems could be build with small automatic electric public cars. For several reasons, such cars should be able to move in platoons. This moving mode involves several vehicles which aggregate and move like a train but which are connected by virtual rather than material links. The software which controls the cars, except for the platoon's leader, is typical of the kind for which formal methods where created. Modeling and developing control algorithms for platooning are an excellent case-study.

We have studied a local version of the Daviet-Parent algorithm [11]. It relies on vehicles being able to observe the position and speed of their front neighbor in the platoon and make decision on those values; no communication between vehicles are required. While not the most efficient control

algorithm, it is very robust and could then be used as a fall-back solution when sophisticated communication-based systems fail. In [12], a simplistic "linear" (or 1D) version of the Davier-Parent algorithm id modeled to assess the feasibility of using Event-B to develop a *proven* version; In [13], a more realistic 2D version is presented. Both versions end in a deterministic model which guarantees that no vehicles belonging to the same platoon may collide. The version presented thereafter has been slightly modified from [13].

B. First Two Models

Figure 1 shows the very first model of 2D platooning[1]. The safety property is expressed as an invariant: vehicles are required to maintain a minimum distance *dmin* between themselves. When the platoon moves, the invariant is preserved.

It should be noted that this model is totally abstract. In a sense, we don't define much more than a vocabulary. The distance function *dist*, the representation of the geometric surface *Plane*, the set of vehicles *Vehicles* and even the movement of the platoon *move* are abstractions. We assume only that they exist, they can be observed, and they have some basic obvious properties. For instance, the axiom labeled `axm9` states that there is enough space to contain one platoon.

Figure 2 is the first refinement. The movement of the platoon is decomposed into the movements of its members. From the perspective of validation, it is important to note the following additions to the initial model which are brought up by the first refinement:

1) two new events,
2) three new variables in the state, with their respective invariants, which represent an abstraction of "temporary" movements,
3) the reduction of the non-determinism of the event *move* by an abstraction function which lifts the concrete variable *tpos* as the abstract parameter *np*,
4) an implicit constraint, introduced by the guard labeled `forw`, which states that the vehicles move forward.

Those additions actually introduce several behaviors and properties into the model. Some behaviors may be explicit in the requirements; for instance, the "order" in which the individual vehicles move is modeled by the non-determinism in the events *fst_move* and *nth_move*. Some behaviors may be implicit or inferred; for instance, the invariant labeled `tpos_safety` requires that the temporary positions and the "observed" positions are all spaced apart by *dmin* at least. This is a stronger property than the invariant. Some behaviors may sneak in as emergent behaviors. For instance, the model may have deadlocks, or platoons may be unable to stop. A validation of the model is then strongly advised, both to check the correct formulation of the explicitly and implicitly required behaviors, and to exhibit and assess the other behaviors.

CONTEXT *space*
SETS *Plane Vehicles*
CONSTANTS *dist dmin*
AXIOMS
axm1: $finite(Vehicles)$
axm2: $finite(Plane)$
axm3: $dmin \in \mathbb{N}_1$
axm4: $dist \in Plane \times Plane \to \mathbb{N}_1$
axm5: $\forall x, y \cdot x \in Plane \land y \in Plane \Rightarrow dist(x \mapsto y) = dist(y \mapsto x)$
axm6: $\forall x \cdot x \in Plane \Rightarrow dist(x \mapsto x) = 0$
axm7: $card(Vehicles) \geq 2$
axm8: $card(Plane) > card(Vehicles)$
axm9: $\exists start \cdot start \in Vehicles \rightarrowtail Plane \land (\forall v1, v2 \cdot v1 \in Vehicles \land v2 \in Vehicles \land v1 \neq v2 \Rightarrow dist(start(v1) \mapsto start(v2)) > dmin)$
END

MACHINE *platoon* **SEES** *space*
VARIABLES *pos*
INVARIANTS
type_pos: $pos \in Vehicles \to Plane$
pos_safety: $\forall v1, v2 \cdot v1 \neq v2 \Rightarrow dist(pos(v1) \mapsto pos(v2)) > dmin$
EVENTS
INITIALISATION $\widehat{=}$
BEGIN
act1: $pos :| pos' \in Vehicles \to Plane \land \forall v1, v2 \cdot v1 \neq v2 \Rightarrow dist(pos'(v1) \mapsto pos'(v2)) > dmin$
END

move $\widehat{=}$ **STATUS ordinary**
ANY *np*
WHERE
type_np: $np \in Vehicle \to Plane$
inv_np: $\forall v1, v2 \cdot v1 \neq v2 \Rightarrow dist(np(v1) \mapsto np(v2)) > dmin$
THEN
act1: $pos := np$
END
END

Figure 1. Initial model of a platooning system

IV. Simulation of the Model

A. Executing the model

Any kid playing with dinky toys would be able to execute *platoon1* without any difficulty. Yet, no current execution tool could animate it or translate it into an executable program. On the one hand, the tools need to consider all the options opened by the non-deterministic features: all the possible realizations of the *Plane* space, all the possible realizations of the distance computation, all the possibilities to put the vehicles in their initial positions, etc. Even when the space of possible values can be automatically explored, it is often impractical because of combinatorial explosion. On the other hand, the kid has an approximate, but good enough, implementation of the elements of the model: a tabletop for *Plane*, a ruler for *dist*, some natural constraints on the movement (no jump, limited turning radius), etc. An interesting observation is that, even if the dinky toys cover only a tiny fraction of all the theoretically possible implementations of *platoon1*, they are in practice the most interesting to observe: the final system will have analogous constraints.

JeB, a JavaScript framework to simulate Event-B models, is based on the last observations. Automated tools and human can cooperate to build and run simulations of formal models. Most of the simulation programs can be automatically generated from the Event-B text. Specifiers have only to provide a few quasi deterministic resolutions of non-deterministic features which will be good approximations of the solutions to come.

[1]The specification presented here is a simplified version of the actual specification.

MACHINE *platoon1* **REFINES** *platoon* **SEES** *space*
VARIABLES *pos tpos iveh unmoved*
INVARIANTS
type_tpos: $tpos \in Vehicles \to Plane$
tpos_safety: $\forall v1, v2 \cdot v1 \neq v2 \Rightarrow dist(tpos(v1) \mapsto tpos(v2)) > dmin$
type_iveh: $iveh \in 1..card(Vehicles) \rightarrowtail Vehicles$
type_unmoved: $unmoved \in \mathbb{P}(1..card(Vehicles))$
EVENTS
INITIALISATION $\widehat{=}$
BEGIN
ini_pos: $pos :| pos' \in Vehicles \to Plane \wedge$
$\forall v1, v2 \cdot v1 \neq v2 \Rightarrow dist(pos'(v1) \mapsto pos'(v2)) > dmin$
ini_tpos: $tpos :| tpos' \in Vehicles \to Plane \wedge$
$\forall v1, v2 \cdot v1 \neq v2 \Rightarrow dist(tpos'(v1) \mapsto tpos'(v2)) > dmin$
ini_iveh: $iveh :\in 1..card(Vehicles) \rightarrowtail Vehicles$
ini_unmoved: $unmoved := 1..card(Vehicles)$
END

move $\widehat{=}$ **REFINES** *move*
WHERE
grd1: $unmoved = \varnothing$
WITH
np: $np = tpos$
THEN
act1: $pos := tpos$
act2: $unmoved := 1..card(Vehicles)$
END

fst_move $\widehat{=}$
ANY *np1*
WHERE
1_move: $1 \in unmoved$
to: $np1 \in Plane$
forw: $dist(np1 \mapsto pos(iveh(2))) <$
$dist(pos(iveh(1)) \mapsto pos(iveh(2)))$
safe: $\forall i.i \in 1..card(Vehicles) \wedge i \neq 1 \Rightarrow$
$dist(tpos(iveh(i)) \mapsto np1) > dmin$
THEN
act1: $tpos(iveh(1)) := np1$
act2: $unmoved := unmoved \setminus \{1\}$
END

nth_move $\widehat{=}$
ANY *v npv*
WHERE
type_v: $v \in 2..card(Vehicles)$
v_move: $v \in unmoved$
to: $npv \in Plane$
forw: $dist(npv \mapsto pos(iveh(v-1))) <$
$dist(pos(iveh(v)) \mapsto pos(iveh(v-1)))$
safe: $\forall i.i \in 1..card(Vehicles) \wedge i \neq v \Rightarrow$
$dist(tpos(iveh(i)) \mapsto npv) > dmin$
THEN
act1: $tpos(iveh(v)) := npv$
act2: $unmoved := unmoved \setminus \{v\}$
END

END

Figure 2. First refinement of the platoon model

We don't present the details of JeB in this paper and refer the interested readers to [9], [14]. Simulating a specification with JeB is a process with four steps:

1) the JeB translator generates a JavaScript program from the axioms, invariants and events of the system, and an HTML page;
2) users complete the program by providing JavaScript functions for generating values for non-deterministic event's arguments and assignments, for implementing carrier sets and functions defined by properties;
3) optionally, users realize a graphical display to adapt the visualization of the state to their purpose;
4) simulations are run, using the scheduler and the set library provided by JeB as a part of the simulator runtime.

Experiments with different specifications at all their refinement stages indicate that the above step (2) accounts for around 1% of the size of the simulation programs.

B. Moving Platoons

When JeB is activated on the specification, it generates the bulk of the code for two independent simulations[2]: one for *platoon* and one for *platoon1*. Those simulations cannot be run directly as the models contain several traits either non executable or executable with unpractical inefficiency. Four points require us to provide our own code.

1) Carrier sets: Neither *Plane* nor *Vehicles* are specified in enough details that an implementation can be derived automatically. *Vehicles* can be simply enumerated:

```
$cst.Vehicles = $B.SetExtension( 'Vehicle01', 'Vehicle02',
                                 'Vehicle03' );
```

Plane cannot be enumerated in a useful way. However, we can be fairly confident that a standard cartesian representation is adequate. In JavaScript, this is done with the definition of an object:

```
$cst.Plane = function(x, y) {
    var obj = {};
    obj.x = x;
    obj.y = y;
    obj.__proto__ = $cst.Plane.prototype;
    return obj;
};
$cst.Plane.finite = true;
$cst.Plane.card = function() {
    return $B('90000');
};
$cst.Plane.anyMember = function() {
    return $cst.Plane( $B.UpTo($B('0'), $B('299')).anyMember(),
        $B.UpTo($B('0'), $B('299')).anyMember() );
};
...
```

2) Function dist: The function *dist* is definite descriptively rather than constructively by the axioms. Again, a standard euclidean definition is a reasonable approximation of the function that will be implemented in future refinements. A possible implementation is:

```
$cst.dist = $B.SetExtension();
$cst.dist.concrete = false;
$cst.dist.functionImage = function( p1, p2 ) {
    var _dist;
    _dist = "" + jeb.util.integer( Math.sqrt( $B.plus(
        $B.multiply( $B.minus(p1.x, p2.x), $B.minus(p1.x, p2.x) ),
        $B.multiply( $B.minus(p1.y, p2.y), $B.minus(p1.y, p2.y) )
        ).literal.toJSValue() ) );
    return $B(_dist);
};
```

[2]The simulators can be found at http://dedale.loria.fr/misc/jeb/platoon-hase

3) Initialization: The starting position of the platoon can be set by a probabilistic computation. However, this has two disadvantages: we may have to wait long before an admissible value is found, and, more to the point, many random legal position are without any interest (vehicles spaced too far apart). The value of *iveh* is not very important, so, setting it to facilitate the reading of the state is a good idea. So, the non-deterministic assignments labeled `ini_pos`, `ini_tpos` and `init_iveh` are realized as assignments to predefined values:

```
$evt.init.action.a1.assignment = function( $arg ) {
    $var.pos._value = $B.SetExtension(
        $B.Pair('Vehicle01',$cst.Plane($B('30'),$B('30'))),
        $B.Pair('Vehicle02',$cst.Plane($B('20'),$B('20'))),
        $B.Pair('Vehicle03',$cst.Plane($B('10'),$B('10')))
    );
};

$evt.init.action.a2.assignment = function( $arg ) {
    $var.tpos._value = $var.pos._value;
};

$evt.init.action.a3.assignment = function( $arg ) {
    $var.iveh._value = $B.SetExtension(
        $B.Pair($B('1'),'Vehicle01'),
        $B.Pair($B('2'),'Vehicle02'),
        $B.Pair($B('3'),'Vehicle03')
    );
};
```

4) Generation of arguments: The generation of the arguments for the events leads to the same reflection as the one above. We can still get a probabilistic choice but on a restricted range for the parameters in *platoon1*:

```
var get_np1 = function (eventId) {
    return $cst.Plane(
        $B.plus($var.tpos.value.getImage('Vehicle01').x,
            $B.UpTo($B('-2'), $B('5')).anyMember() ),
        $B.plus($var.tpos.value.getImage('Vehicle01').y,
            $B.UpTo($B('-2'), $B('5')).anyMember())
    );
};

var get_npv = function (eventId) {
    var v = $B.functionImage($var.iveh.value,
                $evt.e3.parameter.v.value);
    return $cst.Plane(
         $B.plus($var.tpos.value.getImage(v).x,
             $B.UpTo($B('-2'), $B('5')).anyMember()),
         $B.plus($var.tpos.value.getImage(v).y,
             $B.UpTo($B('-2'), $B('5')).anyMember())
    );
};
```

The scheduler will call the two functions repetitively until a value makes the guards true or the number of calls exceeds a specified limit.

The total amount of code provide by us for the simulations of the two machines is around 150 lines, including a graphical display of the platoons (not shown here). The total amount of code of the simulation is around 5,000 lines, either generated or from the runtime.

C. Observations on the simulations

With our provided code, the simulators can run in "automatic" mode. Using this feature, we can execute the models without any further preparation. We ran the simulations many times. In most runs, the platoon behaved as expected, however, a few runs exhibited two unexpected behaviors.

Variable	Value
pos	{Vehicle01↦(x: 33, y: 34), Vehicle02↦(x: 28, y: 18), Vehicle03↦(x: 13, y: 18)}
tpos	{Vehicle01↦(x: 31, y: 38), Vehicle02↦(x: 27, y: 20), Vehicle03↦(x: 13, y: 18)}
iveh	{1↦Vehicle01, 2↦Vehicle02, 3↦Vehicle03}
unmoved	{3}

Figure 3. Deadlocked state (screen-shot)

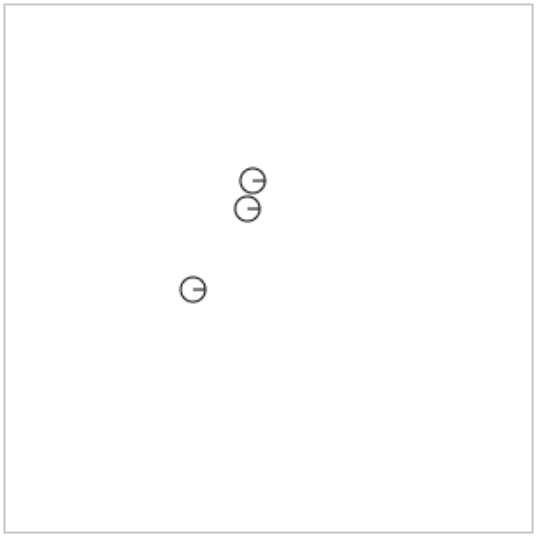

Figure 4. Outstretched platoon (screen-shot)

Figure 3 is a screen-shot of the GUI view which displays the state for *platoon1*. It was taken after the system deadlocked. In this case, we can not find a value for the parameter *npv* of the event *nth_move* to satisfy the guard `forw` and the guard `safe` at the same time, therefore no event can be fired.

Figure 4 shows another interesting behavior. It is a screen-shot of the graphical display of the state which shows the vehicles on the plane. We can see that the vehicles do not look like a platoon anymore. There is no requirement on the maximum distance between two vehicles in the models.

Whether those two behaviors should be considered wrong or not is left to the hypothetical stakeholder who ordered the platooning system. The point is that JeB should allow her to make decision on the model: for instance, to add a requirement about the longest distance, and to consider the temporal properties such as the deadlock-freeness at the early stages of the development.

The issue is of course: can we trust our simulations to make such decisions?

V. Fidelity of Simulations

JeB allows one to easily build simulators. Once a simulation is set up, the issue becomes whether it is "safe" to use it as an observation tool of the model. This, of course, depends on the kind of usage we want to put simulation in.

Table I. SYMBOLS AND NOTATIONS

Elements	Event-B	JavaScript
machine	M	P
sets	s	s^t
constants	c	c^t
axioms	$Axm(s,c)$	
variables	v	v^t
invariants	$Inv(s,c,v)$	$f_{Inv}()$
events	$Evts = \{E^1, E^2, ..., E^m\}$	$f_{Evts} = \{f_{E^1}, f_{E^2}, ...f_{E^m}\}$
initialization event	E^0, $E^0 \notin Evts$	f_{E^0}, $f_{E^0} \notin f_{Evts}$
actions	$Act_{E^0}(s,c,v)$	$f_{E^0}.f_{Act}()$
before-after predicate	$BAP_{E^0}(s,c,v')$	
a particular event	E^i, $E^i \in Evts$	f_{E^i}, $f_{E^i} \in f_{Evts}$
parameters	x_{E^i}	$x^t_{E^i}$
guards	$Grd_{E^i}(x_{E^i},s,c,v)$	$f_{E^i}.f_{Grd}(x^t_{E^i})$
actions	$Act_{E^i}(x_{E^i},s,c,v)$	$f_{E^i}.f_{Act}(x^t_{E^i})$
before-after predicate	$BAP_{E^i}(x_{E^i},s,c,v,v')$	
interpretation function	$bool$	$eval$
boolean values	$TRUE$ $FALSE$	$true$ $false$
equal operator	$=$	$==$
substitution	$x := y$	$x^t = y^t$
event occurrences	$e_j, e_j \in \{E^0\} \cup Evts$	$f_{e_j}, f_{e_j} \in \{f_{E^0}\} \cup f_{Evts}$

To be used for the validation of models, simulations must guarantee that any behavior of the simulation is a behavior specified in the model. We call this property *fidelity* of the simulation to the model. Casting this intuition into a formal frame requires us to define behaviors, observations and the relation between the Event-B code and the JavaScript code.

Let M be an Event-B machine, P be a translated JavaScript simulator of M, Table I summarizes the symbols and notations that we use. The prime ($'$) notation denotes the state after the substitution. The value of a predicate Ψ after the application of a substitution σ is noted as $[\sigma]\Psi$. The superscript (^{t}) notation denotes the translated expressions. The dot (.) notation is used to access JavaScript object properties, the semicolon (;) notation represents sequential execution of JavaScript statements.

We make the assumption that the translation of data types, values and expressions from Event-B to JavaScript is correct. Formally, this can be stated as the existence of a total function mapping JavaScript values to Event-B values ($f \in V_J \rightarrow V_B$) and that the evaluation of the translated predicates produces the same result as the evaluation of the original predicates. Let x be the collection of all free identifiers for predicate Ψ, x^t and f_Ψ are the translated identifiers and predicate function respectively, x_0 be the given values of x^t, then there exists an interpretation function $eval$, where

$$\begin{aligned} &eval(x^t = x_0);\, eval(f_\Psi()) == true \quad \Leftrightarrow \\ &\quad bool([x := f(x_0)]\Psi) = TRUE \\ &eval(x^t = x_0);\, eval(f_\Psi()) == false \quad \Leftrightarrow \\ &\quad bool([x := f(x_0)]\Psi) = FALSE \end{aligned}$$

Actually, Event-B syntax and semantics for expressions and predicates have been designed to be close to their programming equivalent. So, the assumption is reasonable.

A. Behaviors

A behavior of a machine M is a finite sequence of events

$$E^0; e_1; e_2; \ldots e_n$$

such that

$$\forall j.j \in 1..n \Rightarrow e_j \in Evts$$

and

$$\text{fis}(E^0; e_1; e_2; \ldots e_n) \Leftrightarrow true$$

where fis is the feasibility predicate of the sequence which is point-wise extension of the feasibility proof obligation for M:

$$Axm(s,c) \Rightarrow \exists v' \cdot BAP_{E^0}(s,c,v')$$

$$\bigwedge_{i=1}^{m} Axm(s,c) \wedge Inv(s,c,v) \wedge Grd_{E^i}(x_{E^i},s,c,v) \Rightarrow$$

$$\exists v' \cdot BAP_{E^i}(x_{E^i},s,c,v,v')$$

$Traces(M)$ denotes the set of all behaviors of the machine M.

B. Simulation Traces

A simulation trace is a finite sequence

$$f_{E^0}(); f_{e_1}(x_{f_{e_1}}); f_{e_2}(x_{f_{e_2}}); \ldots; f_{e_n}(x_{f_{e_n}})$$

of event firings such that:

$$\bigwedge_{j=1}^{n} eval(f_{e_j}.f_{Grd}(x_{f_{e_j}})) == true$$

$TraceExecutions(P)$ denotes the set of all simulation traces of the simulator P.

C. Fidelity of a simulation

Let $f_{Trace} \in f_{Evts} \rightarrow Evts$ be the function which relates the JavaScript events with their Event-B counterpart and map be the function which applies its functional arguments to all events in a sequence. The fidelity of the simulator P to the model M is defined as

$$\forall t \cdot t \in TraceExecutions(P) \Rightarrow \mathrm{map}(t, f_{Trace}) \in Traces(M)$$

In practice, we need to relate this definition to actual observations and properties on the code introduced by users.

- *Condition 1 (Correct Context Setup):*

$$eval(s^t = s_v); \quad eval(c^t = c_v); \quad eval(f_{Axm}()) == true$$
$$\Rightarrow$$
$$bool([s,c := f(s_v), f(c_v)]Axm(s,c)) = TRUE$$

where s_v and c_v are the actual values given to carrier sets and constants.

- *Condition 2 (Correct Initialization):*

$$eval(v_0 = f_{E^0}.f_{Act}()); \quad eval(f_{Inv}()) == true$$
$$\Rightarrow$$
$$\mathrm{bool}([s,c,v' := f(s_v), f(c_v), f(v_0)]BAP_{E^0}(s,c,v')) = TRUE \wedge$$
$$\mathrm{bool}([s,c,v := f(s_v), f(c_v), f(v_0)]Inv(s,c,v)) = TRUE$$

where v_0 is the value of variables v^t after initialization.

- *Condition 3 (Events' Enabledness):* For all events $E^i j$ in a simulation trace

$$eval(f_{E^i}.f_{Grd}(x_{E^i j})) == true$$
$$\Rightarrow$$
$$bool([s,c,v,x_{E^i} := f(s_v), f(c_v), f(v_j), f(x_{E^i j})]Grd_{E^i}(x_{E^i}, s, c, v)) = TRUE$$

where v_j is the current value of v^t and $x_{E^i j}$ the current value of $x^t_{E^i}$.

- *Condition 4 (Reachable States):*

$$eval(v_{j+1} = f_{E^i}.f_{Act}(x_{E^i j})); \quad eval(f_{Inv}()) == true$$
$$\Rightarrow$$
$$\mathrm{bool}([s,c,v,v',x_{E^i} := f(s_v), f(c_v), f(v_j), f(v_{j+1}), f(x_{E^i j})] BAP_{E^i}(x_{E^i}, s, c, v, v')) = TRUE \wedge$$
$$\mathrm{bool}([s,c,v := f(s_v), f(c_v), f(v_{j+1})]Inv(s,c,v)) = TRUE$$

where v_{j+1} is the next value of v^t, $x_{E^i j}$ the current value of $x^t_{E^i}$ and f_{E^i} a randomly chosen event which satisfies Condition 3.

We can then define the observation theorem:

Theorem 1: If the execution of a simulator P satisfies the condition 1, 2, 3 and 4, then the fidelity of the simulation is guaranteed.

The proof is straightforward.

D. Proof Obligations

Assessing the fidelity of a given simulator amounts to discharging a series of proof obligations generated from the user-provided values and code. Some of those POs can be expressed and discharged in Event-B, but others cannot and need to be discharged by classical semantic reasoning on JavaScript programs.

1) Constants:

PO 1 (Valuation of Constants – Event-B):

$$bool([s,c := f(s_v), f(c_v)]Axm(s,c)) = TRUE$$

2) Parameters:

PO 2 (Valuation of Events Parameters – Event-B):

$$\bigwedge_{j=1}^{n} bool([s,c,v,x_{e_j} := f(s_v), f(c_v), f(v_j), f(x_{f_{e_j}})] Grd_{e_j}(x_{e_j}, s, c, v)) = TRUE$$

3) Hand-coded Functions: The proof obligations associated with the hand-coded functions provided by the users depend on their role and place. There are four cases.

a) Parameter value generators: this kind of functions is just a facility to run the simulation. The values it produces are actually fed to the guard-function of the event; so, since we assume the translation is correct, only legal parameter values will make the guard true. So, the only requirement on parameter generators is that they produce fairly consistent and reasonable values efficiently.

b) Predicate in invariant and guard: let f_Ψ be a user implementation of a particular predicate Ψ. The basis of the PO is to show that f_Ψ is equivalent to a naive translation of Ψ. Discharging such PO requires us to reason at the level of JavaScript programs:

PO 3 (Invariant and Guard – JavaScript):

$$eval(f_\Psi()) == true \quad \Leftrightarrow \quad bool(\Psi) = TRUE$$

c) Value returned by an action: let f_{act} be an user implementation of a particular action act. The PO ensures that the set of computed values are admissible values:

PO 4 (Action – Event-B):

$$\{f(v_0) \mid v_0 = f_{act}()\} \subseteq \{v' \mid BAP_{act}(x_{E^i}, s, c, v, v')\}$$

d) Function defined by properties in a context: this situation requires us to transform each property axiom into a program whose correctness must be established. A property axiom defining the functional constant g has the form

$$\forall v_1, ..., v_n \cdot type(v_1) \wedge ... \wedge type(v_n) \Rightarrow \Psi(g, v_1, ..., v_n)$$

where Ψ contains several application of g. Let g_{user} be the user implementation of g, $m \in \mathbb{N}_1$ be the number of occurrences of g in Ψ, $l \in \mathbb{N}_1$ be the number of parameters of g, the transformation is sketched in the following algorithm:

```
Program Prog =
   for each call to g (i_th call, 1 ≤ i ≤ m)
      for each argument of the call (j_th parameter,
                1 ≤ j ≤ l)
         generate a fresh variable a_ij
         generate instruction
            a_ij = translation of j_th argument expression
      generate a fresh variable for result r_i
      generate r_i = g_user(a_i1, ..., a_il)
   translate the typing expressions to f_type(v_k)
   translate the predicate Ψ to f_Ψ, replacing each
      call of g by their corresponding immediate
      evaluation result r_i
```

The following PO ensures the correcttness of g_{user}:

PO 5 (Uninterpreted Function – JavaScript):

$$wp(Prog, f_\Psi) \Rightarrow \bigwedge_{k=1}^{n} f_{type}(v_k)$$

E. Assumptions

The formal definition of *fidelity* presented above focused on the original part of our contribution: the safe inclusion of hand-coded elements into a simulation. In designing the POs, we made four assumptions:

- A1 Event-B values can be represented in JavaScript,
- A2 Event-B expressions can be translated into an observationally equivalent JavaScript expression,
- A3 Event-B Abstract Syntax Tree (AST) can be syntactically mapped to library calls,
- A4 Event-B operational semantic cycle can be implemented into a scheduler in JavaScript.

SETL [15] shows that A1 is reasonable. Existing translators such as B2C [16] and EB2ALL [8] show that A2 makes sense. We have designed and implemented a set library whose APIs cover all Event-B notations, thus establishing the validity of A3. A4 grounds tools such as ProB [7] and Brama [17]. We are confident that the assumptions can be proven. The proof of correctness of the set library and of the translator can be achieved by using standard verification techniques.

VI. Conclusion

In this paper, we have shown how to integrate safely user-written pieces of code into simulations of abstract models in Event-B. Thus, we can validate all refinements in a development. We strongly believe that the inclusion of validation activities into formal refinement-based methods is necessary as it allows developers and stakeholders to assess the progress of the development toward its goal.

Of course, this work is only a first step. It provides us with the foundations for research in several directions. A first direction is technical. It concerns the integration of JeB into the Rodin environment and the improvement of the efficiency of simulations. A second direction is theoretical. It concerns the generation of the POs and their proof which must be automated in some way to become practical. A third direction is methodological. Current formal methods focus mostly on the issue of verification. Refinement is only defined in this respect. How to integrate validation into the refinement process is an open question.

References

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

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

[3] A. Dardenne, A. van Lamsweerde, and S. Fickas, "Goal-directed requirements acquisition," *Science of Computer Programming*, vol. 20, no. 1-2, pp. 3–50, 1993.

[4] A. Matoussi, F. Gervais, and R. Laleau, "A First Attempt to Express KAOS Refinement Patterns with Event B," in *Proc. of the Int. Conf. on ASM, B and Z (ABZ)*, ser. Lecture Notes in Computer Science. Springer-Verlag, 2008, pp. 12–14.

[5] A. Mashkoor and J.-P. Jacquot, "Guidelines for Formal Domain Modeling in Event-B," in *13th International High Assurance Systems Engineering Symposium (HASE'11)*, Boca Raton, FL, USA, 2011.

[6] T. Nakatani, T. Tsumaki, M. Tsuda, M. Inoki, S. Hori, and K. Katamine, "Requirements Maturation Analysis by Accessibility and Stability," in *Software Engineering Conference (APSEC), 18th Asia Pacific*, 2011, pp. 357–364.

[7] M. Leuschel, J. Falampin, F. Fritz, and D. Plagge, "Automated property verification for large scale B models with ProB," *Formal Aspects of Computing*, pp. 1–27, 2011, 10.1007/s00165-010-0172-1. [Online]. Available: http://dx.doi.org/10.1007/s00165-010-0172-1

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

[9] F. Yang, J. Jacquot, and J. Souquières, "The case for Using Simulation to Validate Event-B Specifications," in *Software Engineering Conference (APSEC), 19th Asia-Pacific*, vol. 1, 2012, pp. 85–90.

[10] J.-R. Abrial, "Formal methods in industry: achievements, problems, future," in *Proceedings of the 28th international conference on Software engineering*, ser. ICSE '06. New York, USA: ACM, 2006, pp. 761–768. [Online]. Available: http://doi.acm.org/10.1145/1134285.1134406

[11] 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.

[12] A. Lanoix, "Event-B Specification of a Situated Multi-Agent System: Study of a Platoon of Vehicles," in *2nd IFIP/IEEE International Symposium on Theoretical Aspects of Software Engineering (TASE 2008)*, France, 2008, pp. 297–304. [Online]. Available: http://hal.archives-ouvertes.fr/hal-00260577/en/

[13] F. Yang and J.-P. Jacquot, "Scaling Up with Event-B: A Case Study," in *The 3rd NASA Formal Methods Symposium (NFM'11)*, ser. LNCS, vol. 6617. California, USA: Springer Berlin / Heidelberg, 2011, pp. 438–452.

[14] F. Yang, "A Simulation Framework for the Validation of Event-B Specifications," Ph.D. dissertation, Université de Lorraine, 2013 (to appear).

[15] J. T. Schwartz, R. B. Dewar, E. Schonberg, and E. Dubinsky, *Programming with sets; an introduction to SETL*. New York, USA: Springer-Verlag, 1986.

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

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