JeB: Safe Simulation of Event-B Models in JavaScript
Faqing Yang, Jean-Pierre Jacquot, Jeanine Souquières

To cite this version:
Faqing Yang, Jean-Pierre Jacquot, Jeanine Souquières. JeB: Safe Simulation of Event-B Models in JavaScript. The 20th Asia-Pacific Software Engineering Conference (APSEC), Dec 2013, Bangkok, Thailand. hal-00908056

HAL Id: hal-00908056
https://inria.hal.science/hal-00908056
Submitted on 7 Dec 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.
JeB: Safe Simulation of Event-B Models in JavaScript

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—The validation of formal models is a challenge for formal methods. We propose JeB, a framework which generates and executes simulations of Event-B models, even highly non-deterministic ones. JeB allows users to safely insert pieces of code to supply deterministic computations where the automatic translation fails. We present how JeB translates Event-B model code to supply deterministic computations where the automatic deterministic ones. JeB allows users to safely insert pieces of and executes simulations of Event-B models, even highly non-formal methods. We propose JeB, a framework which generates JavaScript, Proof-Obligations it through proof-obligations.

Keywords-Formal methods, Event-B, Validation, Simulation, JavaScript, Proof-Obligations

I. INTRODUCTION

Formal methods promise the production of software which are correct, i.e., proven to meet their specification. Hence, getting the right formal specification is crucial.

The validation of formal models through execution is an old issue [1], [2], [3]. Actually, the main difficulty revolves around the level of determinism: good practice advocates the slow refinement of highly abstract and non-deterministic specifications [4] while execution tools require deterministic or quasi-deterministic models.

Event-B [5] is a modeling language of the B family [11] based on refinement; it is supported by the Rodin environment [6]. Event-B has a dynamic semantics which is implemented by the available compilers [7], [8] and animators [9], [10]. However, while the semantics accounts well for non-determinism, the current tools often fail on early models in developments. Hence, those cannot be validated.

JeB, a JavaScript framework for executing Event-B specifications, answers this issue. JeB consists of a translator which compiles the Event-B model into JavaScript code and generates enough “hooks” for developers to provide their own hand-coded resolution of non-determinism. Obviously, hand-coded functions could be “incorrect.” Fidelity formally expresses this notion of correctness. It is defined so we can derive proof obligations.

JavaScript looks an unlikely choice in a formal context, yet it has technical features which fit well with our purpose. Furthermore, the browsers running JavaScript allow to set up easily graphical displays.

We first present Event-B. Then, we discuss JeB’s design and the translator’s implementation. Last, we show the formal definition of fidelity and the associated proof obligations.

II. EVENT-B

A. Notations

A specification in Event-B consists of 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. The values are built inductively from integers, booleans and symbols by using power sets and cartesian products. Special cases such as binary relations, partial functions, or injections enjoy specific notations. The typing system is equivalent to set membership and inclusion. The invariant is a first-order formula on the state. It is a conjunction of smaller formulas called axioms in contexts and invariants in machines.

Events model the evolution of the state. Formally, they are guarded 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 formal semantics of an Event-B model is based on the feasibility (fis) property expressing two ideas:

- a legal state with actual values must exist, and
- firing enabled events leads from legal states to legal states.

The specification language is designed so that 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.

Figure 1 shows the first model of a platooning system. Vehicles are required to maintain a minimum distance \(d_{min}\) between themselves. At this level, the distance function \(dist\), the representation of the geometric surface Plane and the movement of the platoon move are abstractions.

B. Refinement

Event-B embodies two ideas about programs: they are elaborated by stepwise refinements, and they can be correct by construction. In this context, correct means “mathematically proven to meet their specification.”

A model \(M_e\) is a refinement of a model \(M_a\), if:

- \(M_e\) is consistent (i.e., it defines one actual state, invariants are preserved through the actions),
- there is an abstraction function from the \(M_e\) state and events to the \(M_a\) state and events,
- \(M_e\) legal states are abstracted as \(M_a\) legal states, and
- all events fired in \(M_e\) preserve the invariant of \(M_a\).
Non-determinism is present 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. As a practical consequence, execution tools must deal with non-determinism and the associated combinatorial explosion.

### III. JeB Design

#### A. Motivations

The claim that a proven Event-B model is therefore correct in the broad sense is too strong as the three cases below show.

A first case is the introduction of new axioms during a refinement. The new model may then become logically inconsistent, which is a non-decidable property. We must resort to test-like procedures like proving formulas such as true = false.

A second case concerns the limitation on provable properties. In Event-B, temporal properties are notoriously difficult to state and to prove. Therefore, some may be left unchecked.

A third case concerns the introduction of new events, that is, new behaviors. Even when requirement documents guide the specifiers, emergent behaviors may slip in and stay unnoticed until the model is actually executed.

So, the equivalence between “proven” and “correct” can only be claimed if all properties have been expressed in the initial models. Stated another way: we must have complete requirements before developing. Unfortunately, we know that we do not for most actual projects [12].

#### B. Target Language

We use JavaScript/HTML as the target language because:

- JavaScript [13] is embedded and interpreted in most Web browsers. Those environments provide efficient and lightweight tools to build generic and ad-hoc GUI.
- JavaScript supports sophisticated OOP techniques which helped solve two issues: the name conflicts and the traceability between the JavaScript and the Event-B codes. The generated code is clean and easy to read.
- JavaScript uses dynamic interpretation that allows users to switch between different implementations even at runtime. We use this feature to separate user hand-coded functions from the main simulator code.
- JavaScript has first-class functions, which eased the translation of nested expressions and quantifications.

The architecture of JeB reflects our philosophy that a simulation is the result of a cooperation between tools and humans. The former provides the efficiency and safety of automation, the latter provide the insights to overcome non-determinism. Humans are involved at three levels:

- specifiers provide hand-coded functions for some uninterpreted functions,
- experts provide values for constants and event parameters in different scenarios; they set up graphic visualizations,
- users run simulations, observe and analyze behaviors to check that an Event-B model meets their expectations.

---

**Figure 1. Initial model of a platooning system**

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.

Refinement is embedded in the framework at two levels. The language contains a syntax to express the refinement relationship (refines or extends) and the abstraction function (with clause). The support tools generates the POs.

In Event-B, refinements add 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, or
- adding new events; this splits an abstract “large” event into several concrete “smaller” events.

#### C. Event-B Operational Semantics

The operational semantics of Event-B is a three-step cycle:

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

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

<table>
<thead>
<tr>
<th>Namespace</th>
<th>Prefix</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>jeb</td>
<td>jeb</td>
<td>the top namespace</td>
</tr>
<tr>
<td>jeb.axiom</td>
<td>Sxmx</td>
<td>axioms</td>
</tr>
<tr>
<td>jeb.event</td>
<td>Sevt</td>
<td>machine events</td>
</tr>
<tr>
<td>Sxsevtebreb.org</td>
<td>Surg</td>
<td>event parameters</td>
</tr>
<tr>
<td>SB</td>
<td>SB</td>
<td>JavaScript library for Event-B</td>
</tr>
</tbody>
</table>

Table II

<table>
<thead>
<tr>
<th>Event-B</th>
<th>JavaScript/HTML</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONTEXT name</td>
<td>(1) name.js (context model)</td>
</tr>
<tr>
<td>EXTENDS context_names</td>
<td>(2) name.html (user interface)</td>
</tr>
<tr>
<td>SETS identifiers</td>
<td>list of $cst.identifier</td>
</tr>
<tr>
<td>CONSTANTS identifiers</td>
<td>list of $cst.identifier</td>
</tr>
<tr>
<td>AXIOMS predicates</td>
<td>list of jeb.lang.Axiom instances</td>
</tr>
<tr>
<td>END</td>
<td></td>
</tr>
</tbody>
</table>

### IV. JEB REALIZATION

Technically, a simulation consists in three parts: the generated code, a runtime library, and the hand-coded functions provided by the users.

#### A. Generated code

1) **Namespace**: Generated names must not conflict between themselves or existing names in the target environment, and should be traceable back to Event-B. With prefixes on global object properties, we can achieve the same effect as with namespaces; Table I shows some examples.

2) **Translation of Contexts**: Each context is translated into a file which contains the translation of the axioms as JavaScript objects. The overall structure of the file is derived from the structure of the Event-B code, as shown on Table II.

   a) **Sets and Constants**: They are translated as standard identifiers prefixed by $cst$. They are given values in the configuration files. Enumerated sets are instantiated by collecting elements already defined in the CONSTANTS clause. The JEB library supports the instantiation of a carrier set either as a collection of symbolic strings or as a class in the OOP sense.

   b) **Axioms**: JEB translates each axiom as an instance of jeb.lang.Axiom, prefixed by $axm$:

   ```javascript
   jeb.lang.Axiom = function( id, label ) {
   this.id = id;
   this.label = label;
   this.domNode = jeb.ui.$( id );
   this.predicate = function(){...}
   ...};
   ```

   where id is a unique reference; label is axiom’s name as written in the context; domNode links this instance to the HTML page to display its evaluation; predicate is a method to evaluate the axiom.

3) **Translation of Machines**: The translation of a machine generates three files: an executable version of the model, a configuration file (both JavaScript), and a simulation GUI (HTML). The configuration file contains stubs and default implementation of non-deterministic features.

   a) **Variables**: Each variable is translated as an instance of jeb.lang.Variable, prefixed by $var$:

   ```javascript
   jeb.lang.Variable = function( id ) {
   this.id = id;
   this.value = undefined;
   this._value = undefined;
   this.domNode = jeb.ui.$( id );
   ...};
   ```

   where id is a unique reference; value is the current value; _value is the primed value; domNode binds to an element in the HTML page which displays its value; updateView is the method used for updating the user interface. The variable value can be any Event-B mathematical object. It is checked by computing the invariant after each simulation cycle.

   b) **Invariants**: They are translated like axioms, using jeb.lang.Invariant instead of jeb.lang.Axiom.

   c) **Events**: JEB generates an independent simulation for each refinement, so the refinement links (refines and with) are ignored. The guards and actions are translated as two different functions. Events’ parameters require a special treatment described thereafter. Each event is translated as an instance of jeb.lang.Event:

   ```javascript
   jeb.lang.Event = function( id, label ) {
   this.id = id;
   this.label = label;
   this.domNode = jeb.ui.$( id );
   this.enabled = true;
   this.parameter = {};
   this.guard = {};
   this.action = {};
   this.bindArguments = function(){...};
   ...};
   ```

   where id, label, and domNode have the same role as for axioms. The property enabled contains the result of the evaluation of the guards. The properties parameter, guard, and action are namespaces for the event’s parameters, guards and actions, respectively. The four functions are helpers for implementing the simulation cycle: evaluateGuards and testGuards are used for determining parameter values, updateParametersView and doActions do what their names imply.

   d) **Event Parameters**: The so-called “event parameters” in Event-B are the variables introduced in the ANY clause
of events. They are of two kinds: deterministic variables are actually local variables, their value is automatically computed by JeB; non-deterministic variables are true parameters in the common programming sense.

Each true parameter par is associated with an argument generator function get_par which is bound to the input field in the HTML page. The JeB translator generates it into a configuration file as a function stub or, when possible, as a default implementation. Users should provide their own implementation for a realistic simulation in the auto-run mode of execution. Before each simulation cycle, the bindArguments method binds each parameters to a get_par function, which can then be changed “on the fly.”

The order in which the parameters are declared is irrelevant in Event-B, but not in JavaScript. When necessary, the JeB translator adjusts the order of parameters in the GUI and in the bindArguments function so they get a value before being used.

Each identifier declared in an ANY clause is translated as an instance of jeb.lang.Parameter:

```javascript
jeb.lang.Parameter = function(id, type, eventId) {
    this.id = id;
    this.type = type;
    this.domNode = jeb.ui.$(id);
    this.domNodeInput = jeb.ui.$(id + '.input');
    this.eventId = eventId;
    this.label = label;
    this.assignment = function() {};
    ...
};
```

where id is a unique reference; type indicates its nature (local variable or true parameter); domNode (resp. domNodeInput) binds to an element in the HTML page for display (resp. input). eventId references the event;

e) Event Guards: An event guard is a predicate formula. The enabled status of an event is the result of computing all its guards. Guards are translated like axioms with an added reference to the event.

f) Event Actions: Actions are assignments with two features: they occur simultaneously and they can involve non-deterministic values (“becomes such that” for instance).

The left-hand side of Figure 2 is a classical trap. The idea is to use only :: assignments and “primed values” as shown on the right-hand side. The order of execution becomes then irrelevant. Technically, the assignments are realized on the variables’ property _value and copied to the value property after all assignments have been made.

The non-deterministic assignments generate stubs in the configuration file which call default implementation in the runtime library. Users may override them to provide their own, deterministic, implementation.

Each action is coded as an instance of

```javascript
jeb.lang.Action = function(id, label, eventId) {
    this.id = id;
    this.label = label;
    this.eventId = eventId;
    this.assignment = function() {};
    ...
};
```

where assignment is the method to realize the assignment.

4) Translation of Formulas: Each operator is coded as a call to the runtime library. The API of the library is a one-to-one mapping of the Event-B mathematical notation. So, the translation algorithm is based on Event-B syntax and independent of the target language. Proving the correctness of the translation resolves into proving the library functions. Table III shows some translation rules.

B. Runtime Library

1) Simulation Scheduler: The operational semantics is implemented as follows:

1. setup contexts, prompt user if needed
2. trigger the INITIALISATION event
3. do
   1. save the state of variables
   2. update the variable view
   3. compute the invariants, stop if one is false
   4. build the set of enabled events, stop if empty
   5. pick an enabled event to trigger
   6. save the state of arguments
   7. execute the actions of the selected event

The step (3.3) is useless when executing a fully proven model, but it is a useful feature for the preliminary exploration of complex refinements. Step (3.4) is a computation of all the guards of the events based on the state values and the values chosen for the parameters, either through user’s input in the GUI or computation of the get_par functions.

2) JavaScript Library for Event-B: The computation core of the expressions is delegated to a library which implements set theory and first order logic. Its realization aims at two goals: a clear interface and efficient computations.

Technically, the APIs are modeled on the mathematical language of Event-B. The translation from Event-B to JavaScript is mostly syntactic. The APIs contain about 90 profiles.

The representation of numbers is an important issue. In Event-B, integers are unbounded; contrary to JavaScript primitive numbers. Discretization of computations, tiny units or large multiplicative constants are standard modeling tricks to model \( \mathbb{R} \) or functions such as \( \sin \). They induce very large numbers. The library implements arbitrary large integers.

3) Graphical User Interface: The simulation interface is implemented as an HTML page [14]. JeB provides a generic page with the code needed to control it.

C. Hand-Coded Add-Ons

An important feature of JeB is to allow users to provide safely pieces of code to improve the simulation.
1) **Graphical Display:** The generic GUI contains two stubs:
- `jeb.animator.init` to initialize the display area,
- `jeb.animator.draw` to draw the system’s state.

The second function is called at each cycle, the first one is called only once during initialization.

2) **Parameters of Events:** They are the main controls when running a simulation. The value of each “true parameter” is obtained by getting the content of the input fields in the GUI. Those fields can contain either a literal value or a function call. In the latter case, the execution mechanism looks for an implementation in the configuration files. Users can switch between values and functions at any time during a simulation.

V. **CORRECTNESS OF SIMULATION**

The notion of correctness is captured by the idea of **fidelity**.

A. **Definition of Fidelity**

The execution of an Event-B model $M$ is formalized as a **trace** [15]. Intuitively, a trace is a succession of instances of firings of events $1$, starting with the $E^0$ (**INITIALISATION**), which goes through a succession of legal states. Formally, $E^0e_1e_2\ldots e_n$ is a trace if

\[ \forall j, j \in 1..n \Rightarrow e_j \in Evts(M) \]

and

\[ fis(E^0, e_1; e_2; \ldots e_n) \iff true \]

where $fis$ is the feasibility predicate defined as the point-wise application of the feasibility PO of the machine $M$ [11]:

\[ Axm(s, c) \Rightarrow \exists v'. BAP_{E^0}(s, c, v') \]

$1$ lowercase $e_j$ denotes instances, uppercase $E^i$ denotes events

\[ A_{i=1}^m Axm(s, c) \land Inv(s, c, v) \land Grd_{E^i}(x_{E^i}, s, c, v) \Rightarrow \]

\[ \exists v'. BAP_{E^i}(x_{E^i}, s, c, v', v) \]

where $s, c, v$ are the Sets, Constants and Variables of the state, $Inv$ and $Axm$ the invariants and axioms, $BAP$ the Before-After predicate of a substitution, and $v'$ the After-value. $Traces(M)$ is the set of all traces of the machine $M$.

Intuitively, **fidelity** is seen as the inclusion of the set of traces of the executable model into $Traces(M)$. However, one set belongs to the JavaScript world, while the other belongs to the Event-B world. So, we introduce three abstraction-functions:

- $f \in V_j \rightarrow V_B$ which maps values in JavaScript to their equivalent in Event-B,
- $eval$ which evaluates the JavaScript translation $f_{\Psi}(\cdot)$ of the Event-B predicate $\Psi$. $eval$ is assumed such that:

\[ eval(\cdot) \equiv \begin{cases} true & \text{resp. false} \end{cases} \]

\[ bool([x := f(x_0)]\Psi) = \begin{cases} true & \text{resp. FALSE} \end{cases} \]

where $[a := b]\Psi$ is the substitution of $a$ by $b$ in $\Psi$.

- $f_{trace} \in f_{Evts(M)} \rightarrow Evts(M)$ which maps the JavaScript translation of each event to each Event-B event.

Assuming $f$ and $eval$ and noting $x_{f_{\Psi}}$ the event’s parameters, the set of “interesting” traces in a simulation is defined as

\[ f_{E^0}(\cdot); f_{\Psi_1}(x_{f_{\Psi_1}}); f_{\Psi_2}(x_{f_{\Psi_2}})\ldots; f_{\Psi_n}(x_{f_{\Psi_n}}) \]

and

\[ \forall t \in TraceSimulation(M) \Rightarrow map(t, f_{trace}) \in Traces(M) \]

1 | **Event-B Notation** | **Event-B Syntax** | **Translation** |
---|---|---|---|
True predicate | $\top$ | $SB.bTrue()$ | |
Implication | $P \implies Q$ | $SB.implementation(p,T^R, Q^R)$ | |
Universal | $\forall x_1, \ldots, x_n, P (n \geq 1)$ | $SB.\forallAxm([p^R], [Q^R], [x_1^R, \ldots, x_n^R])$ | |
Set membership | $c \in S$ | $SB.belong(S^R, \{c\}^R)$ | |
Focus identifier | $\chi$ | $1)$ Set $\chi$ if $\chi$ is a constant |
2) $\forall v_\chi$ if $\chi$ is a variable |
3) $\forall v_\chi$ if $\chi$ is a parameter |
Boolean TRUE | $true$ | $SB.TRUE$ | |
Boool expression | $bool(P)$ | $SB.bool(p^R)$ | |
Integer literal | $\alpha$ | $SB.intValue(\alpha)$ | |
Boool expression | $bool(f)$ | $SB.bool(f^R)$ | |
Subtraction | $m - n$ | $SB.minus(m^R, n^R)$ | |
Empty set | $\emptyset$ | $SB.EmptySet$ | |
Interval | $[m..n]$ | $SB.UpTo(m^R, n^R)$ | |
Natural numbers | $\phi$ | $SB.NATURAL$ | |
Set extension | $\{E_0, \ldots, E_n\} (n \geq 1)$ | $SB.SetExtension(E_0^R, \ldots, E_n^R)$ | |
Ordered pair | $E \rightarrow F$ | $SB.Pair(E^R, F^R)$ | |
Relations | $S \leftarrow T$ | $SB.Relations(S^R, T^R)$ | |
Partial functions | $S \rightarrow T$ | $SB.PartialFunctions(S^R, T^R)$ | |
Function image | $f(E)$ | $SB.functionImage(f^R, E^R)$ | |
Assignments | $x_1, \ldots, x_n : \{Q(x_1', \ldots, x_n')\} (n \geq 1)$ | $SB.becomesSuchThat([x_1^R, \ldots, x_n^R], [Q^R, [x_1^R', \ldots, x_n^R']])$ | |

\[ \forall v^M = 1 Axm(s, c) \land Inv(s, c, v) \land Grd_{E^i}(x_{E^i}, s, c, v) \Rightarrow \exists v'. BAP_{E^i}(x_{E^i}, s, c, v', v) \]

where $s, c, v$ are the Sets, Constants and Variables of the state, $Inv$ and $Axm$ the invariants and axioms, $BAP$ the Before-After predicate of a substitution, and $v'$ the After-value. $Traces(M)$ is the set of all traces of the machine $M$. Intuitively, fidelity is seen as the inclusion of the set of traces of the executable model into $Traces(M)$. However, one set belongs to the JavaScript world, while the other belongs to the Event-B world. So, we introduce three abstraction-functions:

- $f \in V_j \rightarrow V_B$ which maps values in JavaScript to their equivalent in Event-B,
- $eval$ which evaluates the JavaScript translation $f_{\Psi}(\cdot)$ of the Event-B predicate $\Psi$. $eval$ is assumed such that:

\[ eval(\cdot) \equiv \begin{cases} true & \text{resp. false} \end{cases} \]

\[ bool([x := f(x_0)]\Psi) = \begin{cases} true & \text{resp. FALSE} \end{cases} \]

where $[a := b]\Psi$ is the substitution of $a$ by $b$ in $\Psi$.

- $f_{trace} \in f_{Evts(M)} \rightarrow Evts(M)$ which maps the JavaScript translation of each event to each Event-B event.
B. Proof Obligations

The general definition can be broken down into local properties, which allow to associate a specific PO to each hand-coded element [16]. Six cases have been identified:

1) Valuation of constants. The PO checks the axioms hold.
   \[
   \text{bool}([s,c := f(s_j), f(c_j)])\text{Axm}(s,c) = \text{TRUE}
   \]

2) Valuation of parameters. The PO checks the guards hold.
   \[
   \bigwedge_{i=1}^{n}\text{bool}([x_{e_i} := f(x_{f_i})])\text{Grd}_{e_i}(x_{e_i},s,c,v) = \text{TRUE}
   \]

3) Parameter generator. The produced value is checked against the event’s guards, no proof is needed.

4) Predicate in an invariant or a guard. Let \( f_v \) be the user implementation of the predicate \( \Psi \), the PO is
   \[
   \text{eval}(f_v()) = \text{true} \iff \text{bool}(\Psi) = \text{TRUE}
   \]

5) Value for an action. Let \( f_{\text{act}} \) the user implementation. The PO ensures that computed values are admissible values.
   \[
   \{f(v_j) \mid v_j = f_{\text{act}}()\} \subseteq \{v' \mid \text{BAP}_{\text{act}}(s_E,s,c,v,v')\}
   \]

6) Function defined by non-constructive properties. Technically, this is the most difficult case. An axiom defining a function \( f \) has the form
   \[
   \forall v_1, ..., v_n : \text{type}(v_1) \land ... \land \text{type}(v_n) \Rightarrow \Psi(g,v_1,...,v_n)
   \]
   where \( \Psi \) contains several application of \( g \), and \( \text{type}(v_k) \) is a typing predicate for \( v_k \). \( \Psi \) is translated into a program, \( \text{Prog} \), using JeB rules for predicates and calls to the user’s implementation, \( g_{\text{user}} \), for each appearance of \( g \). Let \( f_v \) be the translation in JavaScript of \( \Psi \) using the definition of \( g \), then the PO is
   \[
   \text{wp}(\text{Prog}, f_v) \Rightarrow \bigwedge_{k=1}^{n} f_{\text{type}}(v_k)
   \]
   where \( \text{wp} \) is the usual weakest-precondition transformer. P0s 1, 2 and 5 can be discharged within the Event-B system; POs 4 and 6 must be discharged within JavaScript semantics.

C. Assumptions

The discussion of fidelity focused on our main contribution: the safe inclusion of hand-coded pieces of code. Discharging the POs proves the fidelity under four assumptions:

A1 Event-B values can be represented in JavaScript.

A2 Event-B expressions can be translated into an observational equivalent JavaScript expression.

A3 Event-B AST can be syntactically mapped to library calls.

A4 Event-B operational semantic cycle can be implemented into a scheduler in JavaScript.

SETL [17] shows that A1 is reasonable. Existing translators such as B2C [7] and EB2ALL [8] show that A2 makes sense. The design of our run library, whose API covers all Event-B notations, establishes the validity of A3. A4 grounds tools such as ProB [9] and Brama [10]. We are confident that the assumptions can be proven. Such proofs would use the well-established techniques of proving compilers.

VI. Conclusion

In [18] we showed how a formally correct model can be safely transformed into an “incorrect” animatable model which keeps the same behaviors. Here, we showed how to produce simulations which are guaranteed to preserve a subset of the behaviors. So, we have the theoretical framework necessary to validate most refinements in a development.

The design and implementations of support tools for proving fidelity is the next challenge: the mix of different formal frameworks raises interesting questions. Another challenge lies in the invention of techniques to create, manage and replay validation scenarios using simulations and animations.

REFERENCES


