Soundness Proof of EventB2Java

Néstor Cataño
Innopolis University
Tatarstan, Russia
Universitetskaya St, 1, Innopolis, 420500
Email: n.catano@innopolis.ru

Shigeo Nishi
Pontificia Universidad Javeriana
Cali, Colombia
Calle 18, No 118-250
Email: nishi@javerianacali.edu.co

***Abstract*—The EventB2Java tool generates JML-specified Java implementations for Event-B models. Code generation is based on the definition of some syntactic rules. This paper presents a soundness proof for the translation encoded by those rules. This proof is important as Event-B is typically used to model safety critical systems, and hence we want to increase our trust on that the code generated by EventB2Java is correct. We conduct our proof in Coq. The proof relies on the definition of a state transition semantics for Event-B and JML in Coq. The soundness proof condition states that any transition step in the semantics of JML in Coq is matched by a transition step of the semantic of the Event-B construct from which the JML construct was translated.**

***Index Terms*—Code Generation; Coq; Event-B; EventB2Java; Formal Methods; Java; JML; Soundness Proof.**

I. Introduction

Event2Java is a code generator for Event-B which translates Event-B models into JML-specified Java programs. The EventB2Java tool [7], [17] has been used to generate the core functionality of several Android applications, including a social-event planner and a car racing game [18]. It has also been used to test the Tokeneer safety critical system [19], and to generate code for the control system of a massive transportation system [9]. Event-B is a formal language for reactive systems [2], and JML (Java Modeling Language) [14] a behavioural specification language for Java. The translation from Event-B to Java and JML at the core of EventB2Java relies on operators EB2Java and EB2Jml, which port Event-B syntax into Java and JML syntax, respectively. This paper presents a soundness proof of the translation encoded by EB2Jml. The proof is conducted using the Coq proof assistant [3]. Soundness proofs are specially important for code generators like EventB2Java since Event-B is typically used to model reactive critical systems and to express safety properties that the code generated by the tool must satisfy. Hence, if the code does not satisfy those properties, the tool is hardly useful.

Our soundness proof relies on a *shallow* [12] embedding of Event-B and JML in Coq. We do not provide a deep semantics for those languages as we are not interested in proving meta-theoretical properties of either language. We use a state transition semantics for modeling Event-B, covering constructs like variables, constants, sets, relations, events, and machines. We make some assumptions on the semantic definition of Event-B to simplify our proof, and we justify our assumptions, e.g. we assume that JML and Event-B semantics operate on the same initial and final states, and that the notion of state is the same for Event-B as for JML. We do not provide a full state transition semantics for the whole JML syntax (we don't need to), but only for the JML constructs that can be obtained as a translation from an Event-B construct via the EB2Jml operator. Our soundness proof states that any state transition step of the semantics of the translation into JML of some Event-B construct can be matched by a state transition step of the Event-B semantics of that construct. The five steps enumerated in Section V outline a general approach to prove the soundness of a translation between two formal languages, in particular, when the semantics of both languages are expressed via transition systems.

II. Background

A. Event-B

Event-B is a formal modelling language for reactive systems [1] that allows the modelling of complete critical systems (software plus hardware devices). Event-B models are composed of *contexts* and *machines*. Contexts define constants, uninterpreted sets and their properties expressed as axioms, while machines define variables and their properties, and state transitions expressed as events. Event-B's language is based on predicate calculus plus set theory. Event-B's models are called *machines*, which are composed of *events*. An event is composed of a *guard* and an *action*. The guard (written between keywords where and then) represents conditions that must hold in some state for the event to be executed. The action (written between keywords then and end) computes new values for state variables. A system that reaches a state where no event guard holds is said to have *deadlocked*. There is no requirement that the system should halt, and indeed, most Event-B models represent systems that run forever. If the system reaches a state where the guards of more than one event hold, the system is said to be *non-deterministic*: Event-B semantics allows any of the events whose guards are satisfied to be executed. The execution of an event is atomic.

Figure 1 presents a simplified version of an Event-B model for social networking (adapted from [10]). In this case, the invariants simply define the type of each variable. Variable "owner" is a surjective total function from "contents" to "persons", and variable "pages" a relation between "contents" and "persons". The initialisation event gives initial values to the state (machine) variables. Two further events are shown:

```
context snctx
 sets  PERSON  CONTENTS  end

machine snEvB  sees snctx
 variables persons  contents  owner pages
 invariants
  inv1  persons ⊆ PERSON
  inv2  contents ⊆ CONTENTS
  inv3  owner ∈ contents ↠ persons
  inv4  pages ∈ contents ↔ persons

 events
  event  initialisation
   then
    act1  persons := ∅     act2  contents := ∅
    act3  owner := ∅       act4  pages := ∅
  end
```

```
event  upload
 any  c1  p1
 where
  grd1  c1 ∈ CONTENTS \ contents
  grd2  p1 ∈ persons
 then
  act1  contents := contents ∪ {c1}
  act2  owner(c1) := p1
  act3  pages := pages ∪ {c1 ↦ p1}
end
```

```
 event  hide
  any  c1  p1
  where
   grd1  c1 ∈ contents
   grd2  p1 ∈ persons
   grd3  c1 ↦ p1 ∈ pages
   grd4  owner(c1) ≠ p1
  then
   act1  pages := pages \ {c1 ↦ p1}
  end
end
```

Fig. 1. A simplified social networking machine in the Event-B language.

one that is triggered when any user uploads a new content item (the "upload" event), and the other triggered when a content item is to be hidden from some user page (the "hide" event). The "upload" event uploads a content item c1 to the account of person p1. c1 should be a fresh item content so that c1 $\notin$ contents. The "hide" event hides content item c1 from the pages of person p1 (different from the owner of c1). Notation $a \mapsto b$ is used for a pair (a, b). The construct:

$$\text{any } x \text{ where } G(s,c,v,x) \text{ then } v := A(s,c,v,x) \text{ end}$$

specifies a *non-deterministic* event that can be triggered in a state where the guard $G(s,c,v,x)$ holds for some sets s, constants c, machine variables v, and parameter x. When the event is triggered, a value for x satisfying $G(s,c,v,x)$ is non-deterministically chosen and the event action $v := A(s,c,v,x)$ is executed with x bound to that value. The correctness condition of the event requires that, for any value chosen for x, the new values of the state variables computed by the action of the event maintain the invariant properties of the machine. In Figure 1, predicates on different lines are implicitly conjoined and actions on different lines are executed simultaneously. The "\" symbol is used for set difference.

B. JML

JML [14] is an interface specification language for Java that is designed for specifying the behaviour of Java classes. JML's type system includes all built-in Java types and additional types representing mathematical sets, sequences, functions and relations. JML includes notations ==> for logical implication, **`\exists`** for existential quantification, and **`\forall`** for universal quantification. JML class specifications can include **`invariant`** specifications (assertions that must be satisfied in every visible state of the class). JML provides pre-post style specifications for Java methods describing software contracts. JML uses keywords **`requires`** for method pre-conditions, **`ensures`** for normal method post-conditions, **`signals`** for method exceptional post-conditions, and **`assignable`** for frame conditions (lists of locations whose values may change from the pre-state to the post-state of a method).

In a JML **`ensures`** or **`signals`** clause, the keyword **`\old`** is used to evaluate expressions in the pre-state of the method – all other expressions are evaluated in the post-state. Figure 2 presents a partial output given by EventB2Java on the event "hide" in Figure 1. The details of the translation algorithm are presented in Section III. Variables "persons" and "contents" are fields of type `BSet`, and pages and owner are fields of type `BRelation` (line 4)[1]. The methods of these classes implement Event-B set and relation operations. EventB2Java translates an event as two methods, one for the guard and one for the action. The `guard_hide` method is the translation of the guard of the hide event. The condition stated by method `guard_hide` is a precondition of method `run_hide`. The exceptional post-condition ensures that no exception is thrown.

C. Coq

Coq [3] combines both a higher-order logic and a strongly typed functional programming language. The abstract syntax of a language like Event-B can be represented in Coq with the aid of functional type definitions and datatypes. Coq has been used in several software verification projects, most notably for the Compcert (formally verified) C compiler [16].

III. The Translation from Event-B to JML

EventB2Java translates Event-B models into Java and JML syntax. The translation is based on three operators EB2Prog, EB2Java and EB2Jml, defined through syntactic rules. The primary operator is EB2Prog, which translates Event-B to JML-specified Java programs. It uses EB2Java to obtain the Java part of the translation and EB2Jml to obtain the JML part. For example, Event-B invariants are translated only as JML specifications, and hence EB2Jml has a rule for invariants,

[1]`BSet` and `BRelation` are EventB2Java implementations of Event-B sets and relations, respectively.

```
public abstract class hide {
  // local declaration of variables contents, pages, owner, etc.

 /*@ public behavior
  requires true; assignable \everything;
  ensures true; signals(java.lang.Exception e) false; */
 public abstract hide(snEvB m);

  /*@ public behavior
      requires true; assignable \nothing;
      ensures \result <==> persons.has(p1) &&  contents.has(c1) &&
         pages.has(new Pair<Integer,Integer>(c1,p1)) && !owner.apply(c1).equals(p1);
  public abstract boolean guard_hide(Integer c1, Integer p1);

  /*@ public behavior
      requires guard_hide(c1,p1); assignable pages;
      ensures pages.equals(\old(pages.
         difference(new BRelation<Integer,Integer>(new Pair<Integer,Integer>(c1,p1)))));
      signals(java.lang.Exception e) false; */
  public abstract void run_hide(Integer c1, Integer p1);
}
```

Fig. 2. Part of the JML translation of the event hide in Figure 1.

whereas EB2Java does not. The soundness proof presented in this paper (Section VI) relates to the translation encoded by the EB2Jml operator.

The rest of this section presents EB2Jml. The full translation behind Event2Java as encoded by EB2Prog can be consulted in [17]. Rule M below presents the translation of an Event-B machine to JML. A machine is translated to a single JML annotated Java class that includes the translation of the machine components (variables, invariants and events). Machine M includes the declaration of machine variables v, invariants I, a distinguished initialising event (called initialisation), and some events e. Machine variables are translated to Java as class attributes via the operator EB2Java.

$$\begin{array}{l}
\text{EB2Java(variables } v) = \mathtt{V} \\
\text{EB2Jml(invariants } I(s,c,v)) = \mathtt{I} \\
\text{EB2Prog(events } e) = \mathtt{E} \\
\text{EB2Jml(event initialisation then } A(s,c,v) \text{ end}) = \mathtt{B}
\end{array} \quad \text{(M)}$$

```
EB2Prog(
  machine M
    variables v
    invariants I(s,c,v)
    event initialisation then A(s,c,v) end
    events e
  end) =

E

public class M {
  I V

  /*@ public normal_behavior
      requires true;
      assignable \everything;
      ensures B; */
  public M();
}
```

The EB2Jml operator naturally translates Event-B invariants as JML invariants.

$$\frac{\text{Pred}(I(s,c,v)) = \mathtt{I}}{\text{EB2Jml(invariants } I(s,c,v)) = \texttt{//@ public invariant I;}} \quad \text{(Inv)}$$

Event-B machines include a specialised initialisation event that gives initial values to state (machine) variables. This event is translated by EB2Jml as the post-condition of the (only) constructor for the Java class resulting from the translation of the machine (see Rule M above).

$$\frac{\text{EB2Jml}(A(s,c,v)) = \mathtt{B}}{\text{EB2Jml(event initialisation then } A(s,c,v) \text{ end}) = \mathtt{B}} \quad \text{(Init)}$$

EventB2Java translates each event into two JML-specified Java methods (see Rule Any below): a `guard` method that determines when the guard of the corresponding event holds, and a `run` method that models the execution of the corresponding event. Variables x in the guard of the event evt are translated by EventB2Java as parameters of both methods. Operator Pred translates Event-B predicates and expressions to predicates and expressions in JML[2].

The normal post-condition of the `guard` method (the **`ensures`** clause) holds if and only if the event guard holds. The exceptional post-condition specification (the **`signals`** clause) ensures that the method will never throw an exception object whose type is a subclass of `java.lang.-Exception`. The event guard is a pre-condition of the `run` method, and the method must satisfy the translation of the event actions. This matches the semantics of Event-B – if the event guard is not satisfied, the event cannot execute and hence cannot modify the system state. Translation of events uses a helper operator Mod, which *collects* the set of variables assigned by Event-B actions[3]. Operator TypeOf translates the type of an Event-B variable or constant to the Java representation of that type.

[2]Rule Any relies on the translation of sets and constants. We encode both of them directly in Coq using native libraries.

[3]Mod's rules largely follow the syntactic rules of the Chase tool [6].

$$\frac{\begin{array}{l}\mathsf{EB2Jml}(A(s,c,v,x)) = \texttt{A}\\ \mathsf{Pred}(G(s,c,v,x)) = \texttt{G}\\ \mathsf{Mod}(A(s,c,v,x)) = \texttt{D}\\ \mathsf{TypeOf}(x) = \texttt{Type}\end{array}}{\begin{array}{l}\mathsf{EB2Prog}(\mathsf{event}\ evt\\ \quad \mathsf{any}\ x\ \mathsf{where}\ G(s,c,v,x)\\ \quad \mathsf{then}\ A(s,c,v,x)\ \mathsf{end}) \ =\end{array}}\ (\mathsf{Any})$$

```
  /*@ public behavior
      requires true;
      assignable \nothing;
      ensures \result <==> G;
      signals(Exception e) false; */
  public boolean guard_evt(Type x);

  /*@ public behavior
      requires guard_evt(x);
      assignable D;
      ensures A;
      signals(Exception e) false; */
  public void run_evt(Type x);
}
```

An event body consists of potentially many non-deterministic and deterministic assignments. In Event-B, the symbol $:|$ represents non-deterministic assignment. Non-deterministic assignments generalise deterministic assignments (formed with the aid of $:=$). For example, $v := v + w$ can be expressed as $v :| v' = v{+}w$, where v' is the value of v after the assignment. Rules NAsg and Rule Asg below translate non-deterministic and deterministic assignments to JML. They are used within JML method post-conditions.

The JML translation of a non-deterministic assignment $v :| P$ is a JML existentially quantified expression. The JML expression **\old**(P) ensures that P is evaluated in the method pre-state. This matches the Event-B semantics for assignments, in which the left-hand side is assigned the value of the right-hand side evaluated in the pre-state. The expressions `v.equals(v')` and `v.equals(`**\old**`(E))` ensure that the v' value of a variable v in the post-state is properly characterised.

$$\frac{\mathsf{Pred}(P(s,c,v,v')) = \texttt{P} \quad \mathsf{TypeOf}(v) = \texttt{Type}}{\begin{array}{c}\mathsf{EB2Jml}(v :| P) = \\ (\backslash\textbf{exists}\ \texttt{Type v'; }\backslash\textbf{old}\texttt{(P) \&\& v.equals(v'))}\end{array}}\ (\mathsf{NAsg})$$

$$\frac{\mathsf{Pred}(E(s,c,v)) = \texttt{E}}{\mathsf{EB2Jml}(v\ := E\,) = \texttt{v.equals(}\backslash\textbf{old}\texttt{(E));}}\ (\mathsf{Asg})$$

Simultaneous assignments in the body of an event are translated individually and the results are conjoined. For example, a pair of simultaneous actions $x := y \,||\, y := x$ is translated to the JML post-condition `x == `**\old**`(y) && y == `**\old**`(x)` for variables x and y of type integer.

$$\frac{\begin{array}{c}\mathsf{EB2Jml}(A1) = \texttt{A1}\\ \mathsf{EB2Jml}(A2) = \texttt{A2}\end{array}}{\mathsf{EB2Jml}(A1 \,||\, A2\,) = \texttt{A1 \&\& A2}}\ (\mathsf{ParAsg})$$

IV. The Proof Embedding

Embedding a language in the logic of a proof assistant consists of encoding the semantics and syntax of the language in the logic. Two approaches, *deep* and *shallow* embeddings [12], have been proposed to formalise languages in logic. In the deep embedding approach, the syntax and the semantics of the (guest) language are formalised in logic as structures, e.g. as data-types. It is thus possible to prove meta-theoretical properties of the language, but proofs are usually cumbersome. In the shallow embedding approach, the language is embedded as types or logical predicates. Proofs are simpler, although one cannot hence prove meta-theoretical properties of the guest language. Our soundness proof is a shallow embedding of Event-B and JML (the guest logics) in the Coq language (the host logic). We intentionally decided to provide a shallow embedding of Event-B and JML in Coq as our intention is not to prove meta-theoretical properties of these languages, but to increase our confidence in that the EventB2Java tool generates code properly with respect to the syntactic defintion of the EB2Jml operator. Our embedding does not cover the full syntax of JML. We only provide an axiomatisation to those JML constructs that can be obtained as a translation from Event-B via the EB2Jml. Therefore, we provide Coq definitions for JML and Event-B predicates, and we include semantic functions that translate both Event-B and JML to Coq, but we do not include a Coq representation of its memory model or define a Coq operational semantics for Event-B.

Our soundness proof comprises Event-B features such as deterministic and non-deterministic assignments, events (including the initialising event), machines, and machine invariants. Our Coq embedding abstracts away the translation of Event-B predicates, and thus the soundness proof assumes the correctness of the translation of these predicates. We abstract events. That is, we consider machines to be composed of an initialising and a single event. Our proof of soundness can naturally be extended to consider a set of of events, but we have not done so here. Therefore, the ability of Event-B to non-deterministically trigger enabled events is not part of the soundness proof (or of the translation presented in Section III). Our soundness proof does not include a proof of invariant satisfaction or absence of deadlocks (the machine invariant entails the disjunction of the guards of the events). We assume that all the proof obligations of the Event-B model, including absence of deadlocks, are discharged in, e.g. Rodin [20], before EventB2Java is used to translate any Event-B model to JML.

V. Proof Sketch

The body of an event is composed of actions that are executed simultaneously. Events are atomic, thus concurrency in Event-B is modelled through the interleaving semantics of the execution of the event actions. Event-B implements actions as substitutions. A substitution evaluates the value of a right-hand side in the pre-state (the state before the execution of an event) and assigns that value to a left-hand side expression of the substitution. Event-B substitutions are of two forms

begin
$v :\mid P(v, v')$
end

any
x
where
$G(v, x)$
then
$v :\mid Q(v, v', x)$
end

Fig. 3. Event-B substitutions

as described in Figure 3. The substitution on the left non-deterministically selects a value that satisfies the predicate P and assigns this value to v. The symbol "$:\mid$" stands for non-deterministic assignment. The predicate P is a *before-after* predicate that depends on the value of the machine variables before (v) and after (v') the assignment. The substitution in right Figure 3 (an event) is parameterised by an event variable x. The event may only be executed (triggered) when the guard G holds. If so, the implementation of the event may select appropriate values for v, v' and x that make the before-after predicate Q true. Our soundness proof implements a state transition semantics of Event-B and JML in Coq. The proof assumes the following conditions:

1) States are defined in the same way in the semantics of both Event-B and JML. States are defined in Coq as sets of variable-value pairs. This is a pragmatic decision that simplifies our proof in Coq.
2) Translating an Event-B predicate produces a JML predicate whose semantics is the same as that of the predicate it was translated from. And, the semantics of Event-B and JML predicates in Coq are the same.
3) Translating an Event-B machine invariant produces a JML invariant whose semantics is the same as of the invariant it was translated from. This is a logical consequence of the previous assumption and Rule Inv as presented in Section III.
4) Event-B machines are composed of a set of machine variables, an initialisation event, and a single standard event.

In addition to the previous assumptions, we consider a simplified version of Event-B substitutions. Rule NAsg (Page III) operates on the substitution in left Figure 3, but the substitution in right Figure 3 is a simplified version of the one on which Rule Any (Page 4) operates, namely, the body of the event in the right consists of a single event action.

Our soundness proof proceeds via the following five sequence of steps that are explored in detail in Section VI.

1) Defining a suitable notion of state in Coq.
2) Implementing Event-B and JML constructs as inductive types in Coq.
3) Implementing the semantics of Event-B and JML in Coq.
4) Implementing the EB2Jml translation rules in Coq.
5) Proving that the semantics of the JML translation of Event-B constructs is simulated by the Event-B semantics of those constructs.

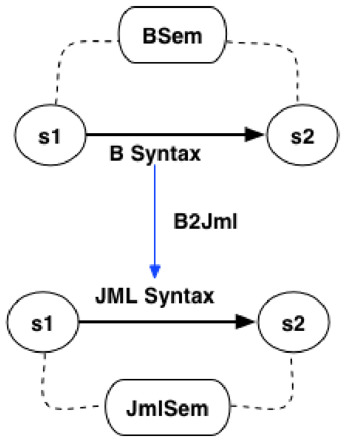

Fig. 4. Proof structure

The semantics of JML and Event-B relate as described in Figure 4. Hence, for any pair of states (a, b), and for any JML construct obtained via translation from an Event-B construct, if the pair of states is an element of the semantics of the JML construct, then it is also an element of the semantics of the Event-B construct that was translated. The expression of the relation shown in the figure is achieved through theorem `eb2jml_machine_sound` in Section VI-F.

VI. The Soundness Proof

A. State Representation

We define the type `state` as a partial function (**`option`** in Coq) from identifiers (the `Id` type) to values (the `A` type). The function `update` adds a pair composed of an identifier and a value to a state. In Coq, lambda expressions are introduced with the aid of the construct **`fun`**. Predicate `beq_id` checks if two identifiers are equal. Inductive predicate `beq_state` (not shown here) implements an equivalence relation over two states.

```
Inductive Id: Type :=
 | Identifier : nat → Id.

Definition total_state (A: Type): Type :=
 Id → A.

Definition state (A: Type): Type :=
 Id → option A.

Definition total_update (s: total_state A)
          (x: Id)(v: A) :=
 fun y ⇒ if beq_id y x then v else s y.

Definition update (s: state A)(x: Id)(v: A) :=
 total_update s x (Some v).
```

We introduce the following notations:

- `x` ϵ `W`: checks if identifier `x` is an element of the set of identifiers `W`.
- `x` ε `s`: checks if identifier `x` is an element of the domain of state `s`.

- `W` $\subseteq$ `s`: checks if the set of identifiers `W` is included in the domain of state `s`.
- `s[x]`: returns the value of identifier `x` in state `s`.
- `s[x` $\Rightarrow$ `v]`: overrides the state `s` with a map from identifier `x` to value `v`.
- $s_1 \doteq s_2$: states s_1 and s_2 are equivalent (they represent the same mapping).

B. Implementing Event-B and JML in Coq

We define `BPredicate` as an inductive type representing an Event-B predicate in Coq. `BPredicate` is implemented with the aid of a `CoqPred`, a function that takes a set of identifiers and a state of integers, and returns **`Prop`** (a Coq predicate). We use `BPredicate` to encode invariants, event guards and, in general, any Event-B predicate[4].

```
Definition CoqPred :=
  Ensemble Id → (state Z) → Prop.

Inductive BPredicate :=
| BPred : CoqPred → Ensemble Id → BPredicate.
```

Inductive type `Assignment` implements Event-B non-deterministic assignments. It is parametrised by an Event-B predicate, and by two variables `v` and `v'`, denoting the before and after value of the machine (state) variables (also see Figure 3). We introduce the `Init` predicate to denote the initialisation event.

```
Definition Guard := BPredicate.

Inductive Assignment :=
| Assg : BPredicate → Id → Id → Assignment.

Definition Init := Assignment.
```

Inductive type `Event` models events. Constructor `Any` is parametrised by an event variable, an event guard, and a non-deterministic assignment (the action part of the event). Type `Machine` models Event-B machines as composed of a machine invariant, the initialising event, the set of state variables, and a (standard) event.

```
Inductive Event :=
| Any : Id → Guard → Assignment → Event.

Inductive Machine :=
| Mach : BPredicate → Init → Ensemble Id →
       Event → Machine.
```

Inductive types for JML constructs are defined in a similar fashion as for Event-B. The encoding of JML in Coq includes definitions for `JmlPredicate`, `JmlBehaviour`, `BeforeAfter`, `JmlMethod`, `JmlConstructor` and `JmlClass`, which implement the definition of JML predicates, JML method specifications, JML before-after predicates, JML class constructors, and JML class specifications, respectively. `JmlPredicate` appears in the translation to JML of Event-B predicates and are used in JML method specifications. Notice that, as for Event-B predicates, a simple JML predicate represented by the `JPred` constructor involves a `CoqPred` and a set of identifiers.

```
Inductive JmlPredicate :=
| JPred : CoqPred → Ensemble Id → JmlPredicate.
```

`BeforeAfter` appears in method post-conditions, and it models the JML **`\old`** operator. **`\old`**`(E)` refers to the before value of expression `E` (its pre-state value). `JmlBehaviour` appears in the translation of an event. It encodes a JML specification (the JML **`behavior`** clause) as composed of a predicate representing the method pre-condition (the **`requires`** clause), an identifier representing a frame condition (the set of variables a method may modify - the JML **`assignable`** clause), and a `BeforeAfter` predicate representing the method-postcondition (the **`ensures`** clause).

```
Inductive BeforeAfter :=
 | JBeforeAfter : JmlPredicate → Id → Id →
      BeforeAfter.

Inductive JmlBehaviour :=
| JBehaviour: JmlPredicate → Ensemble Id →
             BeforeAfter → JmlBehaviour.
```

Type `JmlMethod` implements a JML method specification as composed by a method parameter and a `JmlBehaviour`. The parameter corresponds to the translation of an event guard parameter as given by Rule Any in Section III. `JmlConstructor` appears in the translation of the initialising event. `JmlClass` appears in the translation of complete Event-B models. A `JmlClass` specifications is composed of a class invariant, a class constructor - implemented as a method, a set of fields, and a method. The JML invariant is obtained as translation of the Event-B machine invariant (Rule Inv), the constructor is obtained by translation of the initialisation event (Rules M and Init), the fields are the translation of the machine state variables, and the method is the `run_evt` JML specification part of Rule Any. Section VI-E presents the implementation in Coq of the translation from Event-B to JML.

```
Inductive JmlMethod :=
| JMeth: Id → JmlBehaviour → JmlMethod.

Inductive JmlConstructor :=
| JCons: JmlBehaviour → JmlConstructor.

Inductive JmlClass :=
| JClass: JmlPredicate → JmlConstructor →
      Ensemble Id → JmlMethod → JmlClass.
```

We introduce the further following notations:

- `s` $\uparrow$ `p`: evaluates Event-B predicate `p` in state `s`.
- `s` $\upuparrows$ `asg`: evaluates Event-B predicate assignment `asg` in state `s`.
- `s` $\Uparrow$ `p`: evaluates JML predicate `p` in state `s`.
- `s` $\Uparrow\Uparrow$ `ba`: evaluates JML before-after predicate `ba` in state `s`.

[4]"Ensemble" is the French word for the English word "Set".

C. *Semantics of Event-B in Coq*

Type `BPredSem` below implements the semantics of Event-B predicates. The semantics of a predicate holds if its embedded Coq predicate holds.

```
Definition BPredEval(p: BPredicate)
                (s: state Z) : Prop :=
  match p with
    | BPred coq W ⇒ (coq W s)
  end.

Inductive BPredSem (p: BPredicate)
              (s: state Z) : Prop :=
| CPredSem : (BPredEval p s) → (BPredSem p s).
```

The remaining Event-B constructs are defined over state transitions from some state `a` (the pre-state) to some state `b` (the post-state) as in the case of `AssgSem`, which implements the semantics of non-deterministic assignments. Assignments occur within events. `AssgSem` is parametrised by a non-deterministic assignment, the machine invariant, the Event-B machine variables, and the pre- and post-state of the state transition definition. In short, the semantics of an assignment `asg` holds of states `a` and `b` if (1) machine variable v is different to v', (2) the invariant holds in the prestate, and (2) there exists a value z such that the assignment holds of the pre-state extended with a mapping from v' to z, the post-state equals the pre-state extended with a mapping from the machine variables$v$ to z, and the invariant holds of the post-state.

```
Inductive AssgSem: Assignment → BPredicate →
  Ensemble Id → state Z → state Z → Prop :=
| CAssgSem : forall(asg: Assignment)
                  (invariant: BPredicate)
                  (W U: Ensemble Id)
                  (v v': Id)
                  (a b: state Z),
          W = (getBPredVars invariant)
          → U = (getBPredVars (getAssgPred asg))
          → v = (getAssignedVar asg)
          → v' = (getPrimedVar asg)
          → (Setminus Id U W) = (Singleton Id v')
          → (W ⊆ a)
          → (W ⊆ b)
          → v' ≠ v
          → (v ϵ W)
          → not(v'ε a)
          → not(v'ϵ W)
          → (a ↑ invariant)
          → (exists(z: Z),
                (a[v'⇒ z]) ↑↑ asg ∧
                (a[v ⇒ z]) ≐ b ∧
                (b ↑ invariant))
           → (AssgSem asg invariant W a b).
```

`EventSem` defines the semantics in Coq of an event. `EventSem` is parametrised by the event, the machine invariant, the set of machine variables, and the pre- and post-states. The semantics of an event holds if the invariant holds in the pre-state and some value for the event variable exists such that if the event guard holds for that value, then the semantics of the event action holds; otherwise, if the event guard does not hold, the event action does not execute, leaving the state unchanged, and so the post-state equals the pre-state. A definition for `BInitSem` (the semantics of the initialising evnet) exists similar to `EventSem` in wich the pre-state `a` is the `empty_state`.

```
Inductive EventSem : Event → BPredicate →
  Ensemble Id → state Z → state Z → Prop :=
| AnySem : forall (any: Event)
                  (asg: Assignment)
                  (guard: BPredicate)
                  (invariant: BPredicate)
                  (W: Ensemble Id)
                  (v v' x: Id)
                  (a b: state Z),
          asg = (getEventAssg any)
          → guard = (getEventGuard any)
          → W = (getBPredVars invariant)
          → v = (getAssignedVar asg)
          → v' = (getPrimedVar asg)
          → x = (getEventPar any)
          → (W ⊆ a)
          → (W ⊆ b)
          → v' ≠ v
          → x ≠ v
          → (v ϵ W)
          → not(v'ε a)
          → not(v'ϵ W)
          → not(x ε a)
          → (a ↑ invariant)
          → ( exists(y: Z),
                (a[x ⇒ y] ↑ guard) →
                 ( exists(z: Z),
                    ((a[x ⇒ y])[v'⇒ z]) ↑↑ asg ∧
                    (a[x ⇒ y][v ⇒ z]) ≐ b ∧
                    (b ↑ invariant) ) )
          → (EventSem any invariant W a b).
```

The semantics of an Event-B machine holds if either the initialisation or the standard event executes.

```
Inductive MachineSem : Machine → state Z →
   state Z → Prop :=
| CMachineSem : forall(m: Machine)
                    (invariant: BPredicate)
                    (init: Init)
                    (W: Ensemble Id)
                    (any: Event)
                    (a b: state Z),
      init = (getMachineInit m)
      → invariant = (getMachineInvariant m)
      → W = (getMachineVars m)
      → any = (getMachineEvent m)
      → ((InitSem init invariant W empty_state b) ∨
         (EventSem any invariant W a b))
      → (MachineSem m a b).
```

D. *Semantics of JML in Coq*

As with the semantics of Event-B, we define several inductive predicates that implement the semantics of JML in Coq. Predicate `JmlPredSem` (not shown here) implements the semantics of a predicate in JML. `JmlPredSem`'s definition is similar to `BPredSem`. Inductive predicate `JmlMethSem` below implements the semantics of a JML method. It takes as parameters a JML method, a JML class invariant, the set of JML class fields, and two states. Variable `x` is the `run_evt` method parameter obtained as translation of the event guard variable using Rule Any in Section III. Predicates `requires` and `invariant` are the method pre-condition and the class invariant, respectively. Predicate `requires` does not depend on the after-value of the machine variable, and `invariant` only depends on the machine variables. A transition from state `a` to state `b` in the semantics of a JML method specification is valid if (1) `v` but not `v'` or `x` exist in `a`, `v` exists in `b`, `v` is different to `v'` and to `x`, (2) `invariant` holds in `a` (3), `requires` holds in an extension of `a` that maps `x` to `y`, (4) there exists some value `z` such that `ensures` holds in an extension of `a` that maps the machine variable's after-value to `z` and `x` to `y`, and `b` equals to `a` extended with a mapping from the machine variable to `z`, and (5) `invariant` holds in `b`.

```
Inductive JmlMethSem : JmlMethod →
JmlPredicate →
  Ensemble Id → state value →
  state value → Prop :=
  | JMethSem : forall(method: JmlMethod)
                 (jml: JmlBehaviour)
                 (invariant requires: JmlPredicate)
                 (ensures: BeforeAfter)
                 (W: Ensemble Id)
                 (v v' x: Id)
                 (a b: state value),
    jml = (getJmlSpec method)
    → requires = (getRequires jml)
    → ensures = (getEnsures jml)
    → W = (getJPredVars invariant)
    → v = (getBeforeVar ensures)
    → v' = (getAfterVar ensures)
    → x = (getMethodPar method)
    → (W ⊆ a)
    → (W ⊆ b)
    → v' ≠ v
    → x ≠ v
    → (v ϵ W)
    → not(v' ε a)
    → not(v' ϵ W)
    → not(x ε a)
    → (a ⇑ invariant)
    → ( exists(y: value),
        (a[x⇒y] ⇑ requires) →
        ( exists(z:value),
            (a[x⇒y][v'⇒ z] ⇑⇑ ensures) ∧
            (a[x ⇒ y][v ⇒ z] ≐ b) ∧
            (b ⇑ invariant) ) )
    → (JmlMethSem method invariant W a b).
```

We have also written a semantics for a JML class constructor (not shown here) as the inductive predicate `JmlConsSem` whose definition is similar to `JmlMethSem` except that its pre-state is the `empty_state`. `JmlClassSem` below implements the semantics of a JML class specification. Hence, the semantics of a JML class specification is valid for a transition from state `a` to state `b`, if the class invariant holds in `a` and `b`, and either the class constructor or the method obtained as translation of the machine event are valid for the pair of states.

```
Inductive JmlClassSem : JmlClass → state value →
                        state value → Prop :=
| JClassSem : forall (class: JmlClass )
              (method: JmlMethod)
              (c: JmlConstructor)
              (invariant: JmlPredicate)
              (jml: JmlBehaviour)
              (W: Ensemble Id)
              (a b: state value),
   method = (getMethod class)
  → c = (getConstructor class)
  → jml = (getJmlSpec method)
  → W = (getClassVars class)
  → invariant = (getClassInv class)
  → ((JmlConsSem c invariant W empty_state b) ∨
      (JmlMethSem method invariant W a b) )
  → (JmlClassSem class a b).
```

E. The Event-B to JML Translation Rules

Translation rules in Section III are implemented in Coq as total functions porting Event-B types into JML types and predicates. Six main functions are considered: for translating Event-B predicates, for machine invariants, for non-deterministic assignments, for the initialising event, for standard events, and for full machines. Function `BPred2Jml` below translates a `BPredicate` into a `JmlPredicate`. Since invariants are implemented as predicates, function `BPred2Jml` is also used to implement the translation of machine invariants (Rule Inv in Section III). Function `Assg2Jml` below translates non-deterministic assignments into JML. `Init2Jml` translates the initialisation event into a class constructor encoded as an element of type `JmlConstructor` (Rules M and Init). `Event2Jml` below translates an event as a `JmlMethod` that is composed of a method parameter and a JML method specification. Auxiliary function `Guard2Jml` (not shown here) translates an Event-B guard into an element of type `JmlPredicate`. Finally, function `Machine2Jml` translates an Event-B machine into a JML class specification (Rule M).

```
Definition BPred2Jml(bp: BPredicate)
    : JmlPredicate :=
match bp with
| BPred coq W ⇒ JPred coq W
end.

Definition Assg2Jml(asg: Assignment)
    : BeforeAfter :=
match asg with
| Assg (BPred coq W) v v' ⇒
JBeforeAfter (JPred coq W) v v'
end.

Definition Init2Jml(init: Init): JmlConstructor :=
(JCons
  (JBehaviour
    (JTrue) (* requires part *)
    (Assg2Jml init) (* ensures part *)
  )).

Definition Event2Jml (e: Event) : JmlMethod :=
match e with
| Any x guard assg ⇒
  (JMeth
    x
(JBehaviour (Guard2Jml guard) (* requires part *)
   (Assg2Jml assg)) ) (* ensures part *)
end.

Definition Machine2Jml (m: Machine) : JmlClass :=
match m with
| match invariant init W any ⇒
  JClass
    (BPred2Jml invariant)
    (Init2Jml init)
    W
    (Event2Jml any)
end.
```

F. The Soundness Proof

Theorem eb2jml_machine_sound below presents the main soundness theorem of the translation from Event-B to JML that is in the core of the EventB2Java tool. The semantics of JML and Event-B relate as described in the following. For any pair of states a and b, and for any JML class specification obtained via translation from an Event-B machine, if the pair of states is an element of the semantics of the JML class, then it is also an element of the semantics of the Event-B machine that was translated. Notice that we state a weak form of semantic correctness here. We want to provide guarantee that any valid transition of the JML class specification produced by the translation must also be a valid transition of the source Event-B machine. That is, an Event-B machine must be capable of simulating any valid transition of its JML class counterpart.

```
Theorem eb2jml_machine_sound:
  forall(m: Machine)
     (init: Init)
     (any: Event)
     (binv: BPredicate)
     (class: JmlClass)
     (W: Ensemble Id)
     (a b: state value),
    init = (getMachineInit m) ∧
    any = (getMachineEvent m) ∧
    binv = (getMachineInvariant m) ∧
    class = (Machine2Jml m) ∧
    W = (getMachineVars m)
    → (JmlClassSem class a b)
    → (MachineSem m a b).
```

Theorem eb2jml_machine_sound is discharged under the assumptions in Section V. Discharging the theorem requires (i) using Coq inversion tactic to unfold constructive types used in the hypotheses, (ii) using Coq constructor tactic to unfold constructive types in the goal, (iii) iterating on steps (i) and (ii) for constructive types exposed by the unfolding, and (iv) instantiating existential quantifiers, and using tactics left or right to choose a branch of a disjunction or conjunction of a proof. The whole Coq formalisation consists of 5 non-inductive types, 21 inductive types, 51 functions, 15 lemmas, 3 auxiliary theorems, and 1082 lines of code in Coq. The proof relies on several results. Operator $\Uparrow$ is isomorphic to $\uparrow$, and $\Uparrow\Uparrow$ is isomorphic to $\uparrow\uparrow$. The Event-B initialisation event plays a similar role as a JML class specification constructor. Assignments occurring within an event determine the shape of JML method postcondition specifications. An Event-B guard plays the same role as a JML method preconditon specification.

VII. Related Work

Coq has extensively been used in the formalisation of the semantics of various programming languages as well as for conducting various kinds of correctness proofs, noticeably, Coq's been used by Xavier Leroy et al. [16] in the formalisation and formal verification in Coq of a compiler from a subset of the C programming language to PowerPC assembly code. This proof is important in the context of critical software as well as it is ours.

Defining a shallow or a deep embedding of one language in another is not a new idea [12]. J. Bowen and M. Gordon in [5] propose a shallow embedding for Z in HOL. Catherine Dubois *et al.* propose a deep embedding of B in the logic of Coq [13] for which B proof rules are formalised and proved correct. Jean Paul Bodeveix, Mamoun Filali, and César Muñoz [4] generalise the substitution mechanism of the B method in Coq and PVS as a shallow embedding. In this current work, we consider a shallow embedding of Event-B (the successor of B) into Coq. We are not interested in a deep soundness proof.

Some other efforts relate to the automated verification of B and Event-B Proof Obligations (POs). David Déharbe presents an approach to translate a particular subclass of B and Event-B POs into the input language of SMT-solvers [11]. The first author has implemented the EventB2Dafny Rodin plug-in [8] which translates Event-B proof obligations into Dafny.

In [15], Hermann Lehnner presents a full formalisation of JML in Coq. We did not target for a full formalisation of Event-B in Coq, but for a rapid sound way to validate the code generated by the EventB2Java code generator. Additionally, we formalised in Coq only those JML constructs that can be translated from Event-B via the EB2Jml operator.

VIII. Limitations of Our Work

We have discussed several simplifications and assumptions made in our formal model of Event-B machines in Section V. Formalising a proof in a tool is often a trade off between the level of detail the proof provides and the feasibility of accomplishing it in an automated tool. The soundness proof presented in this paper relies on the correctness of the representation of Event-B mathematical types, e.g. sets and relations, by respective JML types, and, indeed, attempting to represent those in Coq would greatly increase the complexity of our proof. However, our objective has been to validate the translation from Event-B to JML and therefore increase our confidence in that EventB2Java generates code properly.

Our proof considers Event-B machines as composed of one single standard event. Considering Event-B machines with multiple events requires a degree of extra machinery in our proof, but would not change our fundamental approach or conclusions. EventB2Java implements events through two `run_evt` and `guard_evt` methods as shown in Figure 2. Method `run_evt` is atomic, it is executed within `lock` and `unlock` instructions from the Java `concurrent` library. EventB2Java relies on the correctness of the implementation of the Java locking/unlocking mechanism, so we did not need to undertake proof of properties like mutual exclusion, absence of deadlocks or starvation, in Coq.

IX. Conclusion

The soundness proof stated in Section VI is one-way. We are interested in proving that state transitions of a JML-specified program obtained as translation of an Event-B program are matched by state transitons in the semantics of the Event-B program. The way around proof is not interesting to us and it even might be impossible to discharge: abstract state transitions in Event-B do not necessarily match concrete state transitions in JML or Java.

The five steps enumerated at the beginning of Section V outline a general approach for using an automated tool to prove the soundness of a translation between two formal languages, particularly when the semantics of the languages are expressed via transition systems. Figure 4 in that section suggests a granularity level for those transitions in the underlying logic. First, the syntactic constructs of both languages are represented in the logic of the automated tool. Choosing the right representation for those constructs and the right level of detail will determine the level of complexity of the proof together with the number of proof-obligations that are to be discharged (the trade off between a deep and a shallow embedding). Second, the translation from the source to the target language is modelled in logic as axioms taking syntactic constructs from one language to the other. Third and fourth, the type semantics for the source and the target language are provided, and type constructors to build elements of those types are defined. Type states are defined as state transducers. Fifth, a soundness proof is enunciated as relating state transitions in the target language with state transitions in the source language. The level of granularity of those transitions affects the level of complexity of the proof. To simplify the proof, it may be useful to make (valid) assumptions about the representation of states in the transition system for the source and the target language. Choosing different representations for the states will certainly require constructing representations of both in the logic of the automated tool, and establishing a relationship between these representations. This would significantly increase the complexity of the proof.

References

[1] Jean-Raymond Abrial. *Modeling in Event-B: System and Software Design*. Cambridge University Press, New York, NY, USA, 2010.

[2] Jean-Raymond Abrial and Stefan Hallerstede. Refinement, decomposition and instantiation of discrete models: Application to Event-B. *Fundamentae Informatica*, 77(1,2):1–24, 2007.

[3] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer Verlag, 2004.

[4] Jean-Paul Bodeveix, Mamoun Filali, and César Muñoz. A formalization of the B-method in Coq and PVS. In *Electronic Proceedings of the B-User Group Meeting at the World Congress on Formal Methods (FM'99)*, pages 33–49, Toulouse, France, 1999. Springer.

[5] Jonathan Bowen and Mike Gordon. Z and HOL. In *Z User Workshop, Cambridge 1994*, Workshops in Computing, pages 141–167. Springer-Verlag, Cambridge, U.K, 1994.

[6] Néstor Cataño and M. Huisman. Chase: A Static Checker for JML's Assignable Clause. In L. Zuck, P. Attie, A. Cortesi, and S. Mukhopadhyay, editors, *Verification, Model Checking, and Abstract Interpretation*, volume 2575 of *Lecture Notes in Computer Science*, pages 26–40, New York, NY, USA, January 9-11 2003. Springer-Verlag.

[7] Néstor Cataño and Víctor Rivera. EventB2Java: A code generator for Event-B. In *Nasa Formal Methods (NFM)*, volume 9690 of *LNCS*, pages 166–171, Minneapolis, MN, USA, June 7–9 2016. Springer.

[8] Néstor Cataño, Víctor Rivera, and Rustan Leino. The EventB2Dafny rodin plug-in. In *Proceedings of 2nd Workshop on Developing Tools as Plug-ins (TOPI)*, pages 49–54, Zurich, Switzerland, June 3 2012. IEEE Xplore.

[9] Néstor Cataño and Camilo Rueda. Teaching formal methods for the unconquered territory. In J. Gibbons and J.N. Oliveira, editors, *Proceedings of FME Conference on Teaching Formal Methods (TFM)*, volume 5846 of *Lecture Notes in Computer Science*, pages 2–19, Eindhoven, the Netherlands, November 2009.

[10] Néstor Cataño and Camilo Rueda. Matelas: A predicate calculus common formal definition for social networking. In M. Frappier, editor, *Proceedings of ABZ 2010*, volume 5977 of *Lecture Notes in Computer Science*, pages 259–272, Québec, Canada, 2010.

[11] David Déharbe. Integration of SMT-solvers in B and Event-B development environments. *Science of Computer Programming*, 2011. Article in Press.

[12] Michael Gordon. Mechanizing programming logics in higher order logic. In *Current Trends in Hardware Verification and Automated Theorem Proving*, pages 387–439, New York, NY, USA, 1989. Springer-Verlag.

[13] Mélanie Jacquel, Karim Berkani, David Delahaye, and Catherine Dubois. Verifying B proof rules using deep embedding and automated theorem proving. In *Proceedings of the 9th international conference on Software Engineering and Formal Methods (SEFM)*, SEFM'11, pages 253–268, Berlin, Heidelberg, 2011. Springer-Verlag.

[14] G.T. Leavens, A.L. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for Java. *ACM SIGSOFT*, 31(3):1–38, 2006.

[15] Hermann Lehner. *A Formal Definition of JML in Coq and its Application to Runtime Assertion Checking*. PhD thesis, ETH Zurich, Switzerland, 2011.

[16] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, July 2009.

[17] Víctor Rivera and Néstor Cataño. Translating Event-B to JML-specified Java programs. In *ACM Symposium on Applied Computing, Software Verification and Testing track (SAC-SVT)*, South Korea, March 26-30 2014.

[18] Víctor Rivera, Néstor Cataño, Tim Wahls, and Camilo Rueda. Code generation for Event-B. *International Journal on Software Tools for Technology Transfer (STTT)*, pages 1–22, 2015.

[19] Víctor Rivera and Sukriti Bhattacharya Néstor Cataño. Undertaking the Tokeneer challenge in Event-B. In *Proceedings of the 4th FME Workshop on Formal Methods in Software Engineering (FormaliseSE)*, ACM and IEEE Digital Library, pages 8–14, Austin, Texas, USA, May 2016.

[20] Rigorous Development of Complex Fault-Tolerant Systems. Accessed September 2012. http://sourceforge.net/projects/rodin-b-sharp/, (2011).