A

A Machine-Checked Proof for a Translation of Event-B Machines to JML

Néstor Cataño, The University of Madeira
Camilo Rueda, Pontificia Universidad Javeriana
Tim Wahls, Dickinson College

We present a machine-checked soundness proof of a translation of Event-B to the Java Modeling Language (JML). The translation is based on an operator EB2Jml that maps Event-B machines to JML abstract class specifications, Event-B events to JML method specifications, and deterministic and non-deterministic assignments to method post-conditions. The translation has previously been implemented as the EventB2-Jml tool. We adopted a *taking our own medicine* approach in the formalisation of the proof so that Event-B as well as JML are formalised in Event-B and the proof is discharged in Rodin. For any Event-B substitution (whether an event or an assignment) and for the JML method specification obtained by applying Event-B2Jml to the substitution, we prove that the semantics of the JML method specification is *simulated* by the semantics of the substitution. Therefore, the JML specification obtained as translation from the Event-B substitution is a *refinement* of the substitution. Likewise, for any Event-B machine, and for the JML abstract class specification obtained as translation from the machine, we prove that the semantics of the machine simulates the semantics of the JML class specification. The proof includes machine invariants and the standard Event-B initialising event, but it does not include Event-B contexts. We assume that the semantics of the JML specification and the Event-B model operate on the same initial and final states, and we justify our assumption.

Categories and Subject Descriptors: F. Theory of Computation [**F.3 LOGICS AND MEANINGS OF PROGRAMS**]: F.3.1 Specifying and Verifying and Reasoning about Programs

General Terms: Languages, Verification

Additional Key Words and Phrases: Event-B, JML, EventB2Jml, Formal Proof, Translation, Rodin

ACM Reference Format:
ACM Trans. Softw. Eng. Methodol. V, N, Article A (January YYYY), 19 pages.
DOI = 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

1. INTRODUCTION

Software engineering researchers have developed an array of formal languages and associated tools for building provably correct software systems. These approaches can be divided into two broad categories: design-by-contract approaches as used with languages such as the Java Modeling Language (JML) [Leavens et al. 2006] and Spec# [Barnett et al. 2004] in which code is verified against a formal specification, and correctness-by-construction approaches such as B [Abrial 1996] and Event-B [Abrial 2010] in which an abstract model is translated into code via a series of provably correct refinement steps. Each approach has characteristic strengths and weaknesses. Many software developers prefer design-by-contract approaches, as there is always a tangible connection to code and the notation tends to be more intuitive. However, this relationship to code is also a weakness in that correctness-by-construction approaches often

start with a very abstract model of the system that is much more amenable to proofs of correctness, safety and security properties than a typical software contract would be. Conversely, the notation used in correctness-by-construction approaches and the general level of mathematical sophistication required can be intimidating, but the ability to reason over abstract models is often critically important.

In an earlier work [Cataño et al. 2012b], we have proposed a design methodology that attempts to combine the design-by-contract and correctness-by-construction approaches in a way that takes advantage of the strengths and avoids the weaknesses of each. In our approach, development begins with an abstract Event-B model which is proven to satisfy the properties necessary for the particular application. The model can be refined (and associated refinement proof obligations discharged) as needed to add more functionality, but the goal of these refinements is not to bring the model closer to code level as would normally be done in an Event-B development process. Rather, once a refinement that includes adequate functionality is achieved, it is translated to a JML class specification. This specification is then implemented and the implementation is verified against the specification. While our approach requires some degree of expertise in both Event-B and JML among the members of the development team, it has some distinct advantages over the use of either language in isolation:

— abstract models can be verified using the full suite of Event-B tools designed expressly for this purpose.
— the proof obligations generated for this type of refinement tend to be relatively easy to discharge, as the refinements are not moving toward a more concrete model.
— refinement chains tend to be relatively short. This is advantageous in that each successive refinement is often more difficult to verify than the last.
— members of the development team who do not have Event-B expertise can be charged with implementing the JML specification in Java.
— the kind of JML specification generated by the translation is relatively easy to implement, and verifying the implementation against this sort of specification is relatively straightforward.

We have validated our approach by using it to develop several Android [google Inc. 2012; Mejer 2012] applications, including a social-event planner [Rivera and Cataño 2012] and a car racing game [Perchy and Cataño 2012].

The cornerstone of our approach is the EventB2Jml tool.[1] EventB2Jml automatically translates Event-B machines to JML class specifications, and is implemented as a plugin for the Rodin platform [Butler et al. 2006]. While the translation could be performed manually, doing so would add significant time and cost to the development effort, and any errors made during the translation would result in code that was not a correct implementation of the original Event-B model. To ensure correctness, the JML specification would need to be verified against the model – again requiring significant time and expertise. Of course, automating the translation does not ensure that it is sound. However, it does provide an opportunity to prove the soundess of the translation (once), and then to use a tool implementing the translation (such as EventB2Jml) with confidence whenever it is advantageous to do so within a development effort. We present that proof in this paper.

The translation is based on an operator EB2Jml that maps Event-B machines to JML abstract class specifications, Event-B events to JML method specifications, and deterministic and non-deterministic assignments to JML method postconditions. Several of the rules that define EB2Jml are presented in Section 4. We have adopted a *taking our own medicine* approach in the formalisation of the proof so that Event-B as

[1]The tool is available at `http://poporo.uma.pt/~ncatano/Projects/favas/EventB2JML.html`

well as JML are formalised in Event-B and the proof is discharged in Rodin [RODIN 2011]. Our formalisation rests on a *shallow* embedding of Event-B and JML in the logic of the Event-B language. A shallow embedding approach (as opposed to a deep embedding) facilitates conducting and discharging a proof, but does not permit proofs of meta-properties about the language one is modelling. Hence, for any Event-B substitution (whether an event or an assignment) and for the JML method specification obtained by applying EventB2Jml to the substitution, we prove that the semantics of the JML method specification is *simulated* by the semantics of the substitution. Likewise, for any Event-B machine, and for the JML abstract class specification obtained as translation from the machine, we prove that the semantics of the machine simulates the semantics of the JML class specification. In this respect, the semantics of the JML specification is a *refinement* of the semantics of the Event-B model in that any behaviour in JML is matched by a behaviour in Event-B. The soundness proof considers Event-B machine invariants, machine variables, events and, in particular, the Event-B initialising event, but not machine contexts. To facilitate the proof we make a few assumptions (e.g. that the JML and Event-B models operate on the same initial and final states), and we justify the soundness of our assumptions.

The contributions of this paper are the following: $(i.)$ we present a machine-checked soundness proof for the key rules defining the EB2Jml operator. These rules are the core of the implementation of the EventB2Jml tool. $(ii.)$ we generalise our proof approach so that it can be extended to soundness proofs of translations of other formal languages.

2. RELATED WORK

Defining a shallow or a deep embedding of a language is not a new idea [Boulton et al. 1992; Gordon 1989]. Indeed, J. Bowen and M. Gordon in [Bowen and Gordon 1994] propose a shallow embedding for Z [Woodcock and Davies 1996] in HOL. Catherine Dubois *et al.* propose a deep embedding of B in the logic of Coq [Jaeger and Dubois 2007; Jacquel et al. 2011] for which B proof rules are formalized and proved correct. We opted for a shallow embedding approach instead, yet we consider Event-B, the successor of B. In a similar direction, Jean Paul Bodeveix, Mamoun Filali, and César Muñoz [Bodeveix et al. 1999] generalise the substitution mechanism of the B method in Coq and PVS as a shallow embedding.

Some other efforts relate to the automated verification of B and Event-B proof obligations. David Déharbe presents an approach to translate a particular subclass of B and Event-B proof obligations into the input language of SMT-solvers [Déharbe 2011]. Furthermore, we have implemented the EventB2Dafny Rodin plug-in [Cataño et al. 2012a] that translates Event-B proof obligations into Dafny, one of the front-end specification languages of the Boogie 2 tool [Barnett et al. 2006].

Other works relate to the translation of formal specification (and modeling) languages, and to the implementation of tools automating these translations. Thus, in an earlier work [Cataño et al. 2012c], the authors define a translation from *classical* B to JML implemented in the ABTools suite [Boulanger 2003].

Méry and Singh [Méry and Singh 2011] define a translation tool (implemented as a Rodin plug-in) that automatically translates Event-B machines into several different languages: C, C++, Java and C#. Wright [Wright 2009] defines a B2C extension to the Rodin platform that translates Event-B models into C code. However, this work considers only simple translations of formal concrete machines. Edmunds and Butler [Edmunds and Butler 2010; 2011] present a tasking extension for Event-B that generates code for concurrent programs (targeting multitasking, embedded and real-time systems). The main issue with these tools is that the user has to provide a final (or

at least an advanced) refinement of the system so that it can be directly translated to code.

Jin and Yang [Jin and Yang 2008] outline an approach for translating VDM-SL to JML. Their motivations are similar to ours in that they view VDM-SL as a better language for modelling at an abstract level (much the way that we view Event-B), and JML as a better language for working closer to an implementation level. In fact, they translate VDM variables to Java fields, thus dictating the fields of an implementation.

Bouquet et. al. have defined a translation from JML to B [Bouquet et al. 2005] and implemented their approach in the JML2B tool [Bouquet et al. 2006]. Although their translation is in the opposite direction from ours, their motivation is again quite similar – they view translation as a way to gain access to more appropriate tools for the task at hand, which in this case is verifying the correctness of an abstract model without regard to code. JML verification tools are primarily concerned with verifying the correctness of code with respect to specifications, while B has much stronger tool support for verifying models.

3. PRELIMINARIES

3.1. The Event-B Method

The B method, introduced by J.-R Abrial [Abrial 1996], is a strategy for software development in which an abstract model of a system is transformed into an implementation via a series of refinement steps, where the behaviour of each refinement is provably consistent with the behaviour of the previous step. Each refinement adds more details to the description of the system. A refinement generates proof obligations that must be formally verified in order to assert that a model M_{i+1} is indeed a refinement of a previous model M_i. These proof obligations are necessary and sufficient conditions to guarantee that, although at different levels of abstraction, both are models of the same system.

A derivative of the B method, also introduced by J.-R Abrial [Abrial 2010], is called Event-B. Event-B models are complete developments of discrete transition systems. They are composed of machines and contexts. Machines contain the dynamic parts of a model (e.g. variables, invariants, events). Contexts contain the static part of a model (e.g. carrier sets, constants).

A partial example of an event that creates an account for a social networking site (adapted from the B model in [Cataño and Rueda 2010]) is depicted in Figure 1. PERSON is a carrier set that represents the set of all possible people in the network, and CONTENTS the set of all possible images, text, and in general content, in the network. $persons$ is the set of people actually in the network, $contents$ the set of content actually in the network, $owner$ is a total surjection mapping each content item to its owner, and $pages$ is a total relation indicating which content items are visible to which people. The event $create\_account$ adds a new person and associated initial content to the network. Unlike the B Method in which operations are called, Event-B defines events that might be executed/triggered when the guard (the part between the where and the then) is true. Hence, this event can execute whenever there is at least one person and at least one content item that has not yet been added. The meaning of an event is the meaning of the actions in its body (the six parts between the then and the end), in this case, that the person $p1$ and content $c1$ are added to the network, that $p1$ owns $c1$, and that $c1$ is visible to $p1$. The symbol $\setminus$ represents set difference, $\mapsto$ a pair of elements, $\leftrightarrow$ a relation, $\rightarrow$ a total function, and $\nrightarrow$ a partial function.

$create\_account$
any $p1$ $c1$ where
grd1 $p1 \in \text{PERSON} \setminus persons$
grd2 $c1 \in \text{CONTENTS} \setminus contents$
then
act1 $contents := contents \cup \{c1\}$
act2 $persons := persons \cup \{p1\}$
act3 $owner := owner \cup \{c1 \mapsto p1\}$
act4 $pages := pages \cup \{c1 \mapsto p1\}$
act5 $viewp := viewp \cup \{c1 \mapsto p1\}$
act6 $editp := editp \cup \{c1 \mapsto p1\}$
end

Fig. 1. An event for creating an account for a social networking site.

3.2. JML

JML [Burdy et al. 2005; Breunesse et al. 2005; Leavens et al. 2012] is a model-based language designed for specifying the interfaces of Java classes. JML specifications are typically embedded directly into Java class implementations using special comment markers `/*@ ... */` or `//@`. Specifications include various forms of invariants and pre- and post-conditions for methods. The syntax is intentionally similar to that of Java so that it is less intimidating for developers. In particular, the mathematical types that are heavily used in other model based specification languages (sets, sequences, relations and functions) are provided in JML as classes, and the operations on those types are specified (in JML) and implemented as Java methods.

A simple JML specification for a Java class consists of pre- (`requires` in JML) and post-conditions (`ensures` in JML) added to its methods, and class invariants (`invariant` in JML) restricting the possible states of class instances. JML method pre- and post-conditions are embedded as comments immediately before method declarations. JML predicates are first-order logic predicates formed of side-effect free Java boolean expressions and several specification-only JML constructs. JML provides notations for forward and backward logical implications, `==>` and `<==`, for negation `!`, for non-equivalence `<=!=>`, and for logical or and logical and, `||` and `&&`. The JML notations for the standard universal and existential quantifiers are `(\forall T x; E)` and `(\exists T x; E)`, where `T x;` declares a variable `x` of type `T`, and `E` is the expression that must hold for every (some) value of type `T`.

Figure 2 presents the JML translation of the Event-B event in Figure 1. An Event-B event can be triggered (its actions can be executed) when its guard is satisfied. Actions are written between a then and an end keyword. This behaviour is modelled by creating two methods: a `guard` method containing the translation of the guard, and a `run` method containing the translation of the event body. The `run` method should only be executed when the `guard` method returns `true`. This is expressed as a pre-condition (`requires` in JML). The `guard_create_account` method ensures that the guard of the corresponding Event-B event (`create_account`) holds, while the method `run_create_account` specifies the event (effect) of creating a new account in the system. `\result` represents the result of a method call. The `assignable` clause is a frame condition specifying what locations may change from the pre-state to the post-state, e.g. all fields representing Event-B variables for this method. The pre-state is the state on entry to the method and the post-state is the state on exit from the method. Two special `assignable` specification exist, `assignable \nothing`, which specifies that the method modifies no location, and `assignable \everything`, which specifies that the method

```
/*@ assignable \nothing;
    ensures \result <==> (\exists Integer c1; (\exists Integer p1;
      (PERSON.difference(persons).has(p1) && CONTENTS.difference(contents).has(c1)))); */
public abstract boolean guard_create_account();

/*@ requires guard_create_account();
    assignable contents, persons, owner, pages, viewp, editp;
    ensures (\exists Integer c1; (\exists Integer p1;
       \old((PERSON.difference(persons).has(p1) && CONTENTS.difference(contents).has(c1)))
    && contents.equals(\old(contents.union((new BSet<Integer>(c1)))))
    && persons.equals(\old(persons.union((new BSet<Integer>(p1))))))
    && owner.equals(\old(owner.union((new BRelation<Integer,Integer>().singleton(c1,p1)))))
    && pages.equals(\old(pages.union((new BRelation<Integer,Integer>().singleton(c1,p1)))))
    && viewp.equals(\old(viewp.union((new BRelation<Integer,Integer>().singleton(c1,p1)))))
    && editp.equals(\old(editp.union((
         new BRelation<Integer,Integer>().singleton(c1,p1)))))));
 also
   requires !guard_create_account();
   assignable \nothing;
   ensures true; */
public abstract void run_create_account();
```

Fig. 2. A JML specification produced for creating an social networking account.

can modify any. The post-condition (`ensures` in JML) is expressing that in the state on exit from the method, person `p1` is added to the system with ownership, visibility and all permissions on content `c1`. Expressions within `\old` are evaluated in the pre-state, while all other expressions are evaluated in the post-state.

4. THE TRANSLATION FROM EVENT-B TO JML

The translation from Event-B to JML is defined with the aid of an EB2Jml operator that translates Event-B syntax to JML syntax. In the following, we present the EB2Jml rules for Event-B substitutions and for the translation of a machine.

Events in Event-B are translated to two JML methods: 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. In Rule Any below, variables bound by an any construct are existentially quantified in the translation, as any values for those variables that satisfy the guards can be chosen. The Pred operator translates Event-B predicates and expressions to JML. The translation of the guard is included in the precondition of the `run` method, and in the postcondition in order to bind these same variables, as they can be used in the body of the event. Translation of events uses an additional helper operator Mod, which calculates the set of variables assigned by the actions of an event (the JML `assignable` clause). The JML specification of each `run` method uses two specification cases. In the first case, the translation of the guard is satisfied and the post-state of the method must satisfy the translation of the actions. In the second case, the translation of the guard is not satisfied, and the method is not allowed to modify any fields, ensuring that the post-state is the same as the pre-state. This matches the semantics of Event-B – if the guard of an event is not satisfied, the event cannot execute and hence cannot modify the system state. Note that the effective pre-condition of a JML method with multiple specification cases (`also` in JML) is the disjunction of the pre-conditions of each case – so that in our translation, the pre-condition of a `run` method is always `true`. Hence, even though guards are translated as pre-conditions, no method in the translation result has a pre-condition. Rather, the translation of the guard determines which behaviour the method must exhibit.

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

```
/*@ assignable \nothing;
    ensures \result <==> (\exists Type x; G); */
public abstract boolean guard_evt();

/*@ requires guard_evt();
    assignable M;
    ensures (\exists Type x; \old(G) && A);
  also
    requires !guard_evt();
    assignable \nothing;
    ensures true; */
public abstract void run_evt();
```

The translation of ordinary and non-deterministic assignments via operator EB2Jml is presented below. The symbol $:|$ represents non-deterministic assignment. Non-deterministic assignments generalise deterministic assignments (formed with the aid of $:=$), e.g. $v := v + w$ can be expressed as $v :| v' = v + w$. If variable v is of a primitive type, the translation would use == rather than the `equals` method.

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

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

Multiple actions in the body of an event are translated individually and the results are conjoined. For example, a pair of actions:

$$\begin{array}{l}\textbf{act1}\ x := y \\ \textbf{act2}\ y := x\end{array}$$

is translated to: `x == \old(y) && y == \old(x)` for integer variables x and y, which correctly models simultaneous actions as required by the semantics of Event-B.

The Mod operator *collects* the variables assigned by Event-B actions. The cases of Mod for assignments are shown below. For the body of an event, Mod is calculated by unioning the variables assigned by all contained actions.

$$\mathsf{Mod}(v\ := E\ ) = \{v\} \qquad \mathsf{Mod}(v :|\ P) = \{v\}$$

Rule M below translates a machine M that sees a context C. We assume that all Event-B proof obligations of are discharged (e.g. in Rodin) before the machine is translated to JML, so that proof obligations and closely related Event-B constructs (namely, witnesses and variants) need not be considered in the translation and in the soundness proof. A witness contains the value of a disappearing abstract event variable, and a variant is an expression that should be decreased by all convergent events. An Event-B machine is translated to a single `abstract` JML annotated Java class, which can be extended by a subclass that implements the abstract methods. The translation of the context C is incorporated into the translation of machines that see the context.

$$\frac{\begin{array}{l}\text{EB2Jml(sets } s) = \texttt{S} \\ \text{EB2Jml(constants } c) = \texttt{C} \\ \text{EB2Jml(axioms } X(s,c)) = \texttt{X} \\ \text{EB2Jml(theorems } T(s,c)) = \texttt{T} \\ \text{EB2Jml(variables } v) = \texttt{V} \\ \text{EB2Jml(invariants } I(s,c,v)) = \texttt{I} \\ \text{EB2Jml(events } e) = \texttt{E}\end{array}}{\begin{array}{l}\text{EB2Jml(machine } M \text{ sees } C \\ \qquad \text{variables } v \\ \qquad \text{invariants } I(s,c,v) \\ \qquad \text{events } e \\ \quad \text{end}) = \\ \texttt{public abstract class } M \ \{ \\ \quad \texttt{S C X T V I E} \\ \}\end{array}}\ \text{(M)} \qquad \begin{array}{l}\text{context } C \\ \quad \text{sets } s \\ \quad \text{constants } c \\ \quad \text{axioms } X(s,c) \\ \quad \text{theorems } T(s,c) \\ \text{end}\end{array}$$

5. THE PROOF

Embedding a language in the logic of a proof assistant consists of encoding the semantics and syntax of the language in the logic. Two different ways of formalising languages in logic exist, namely, *deep* and *shallow* embedding [Boulton et al. 1992; Gordon 1989]. In the deep embedding approach, the syntax and the semantics of the language are formalised in logic as structures, e.g. as data-types. It is thus possible to prove meta-theoretical properties of the language, yet proofs are usually cumbersome. In the shallow embedding approach, the language is embedded as types or logical predicates. Proofs become simpler, although one cannot then prove meta-theoretical properties of the language itself.

Our soundness proof of the translation from Event-B to JML rests as a shallow embedding of Event-B and JML in the logic of the Event-B language. The proof allows for deterministic and non-deterministic assignments, events, including the initialising event, complete machines and machine invariants, yet contexts are not considered. Our embedding abstracts away the translation of Event-B predicates, and the correctness of the semantics of the whole translation assumes thus the correctness of the translation of these predicates. We abstract machine variables and events, that is, we consider the machine to be composed of a single variable and a (non-initialising) single event. Yet, the proof can be extended to rather consider a set of variables, and a set of events. Hence, the Event-B ability for non-deterministically triggering enabled events is not part of the soundness proof (or of the translation presented in Section 4). Notice that our soundness proof need not to include a proof of absence of deadlocks (the machine invariant entails the disjunction of the guards of the events) since we assume that all the proof obligations of the Event-B model are discharged before it is translated to JML.

Grosso modo, our soundness proof ensures that any state transition step of the JML semantics of the translation of some Event-B construct into JML can be *simulated* by a state transition step of the Event-B semantics of that construct. All steps in the proof are modelled in Event-B and implemented in Rodin [Butler et al. 2006], a platform that provides support to the writing and verification of Event-B models [RODIN 2011]. The soundness condition aforesaid is stated as a theorem and proved interactively in Rodin. The whole proof structure is represented by a collection of contexts in Rodin, with translation rules and semantics implemented as axioms.

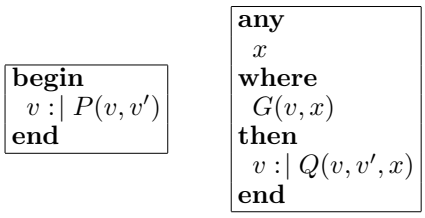

Fig. 3. Event-B substitutions.

Substitutions are of two forms as described in Figure 3. The substitution in the left non-deterministically selects a value that satisfies the predicate P and assigns this value to v. The symbol $:|$ 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 the right (an event) is parametrised by an event variable x. The event may only be executed (triggered) when the guard 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.

In formalising and conducting the proof, we assume the following:

(1) Translating an Event-B predicate produces a JML predicate whose semantics is the same as of the predicate it was translated from.
(2) Translating an Event-B machine invariant produces a JML invariant whose semantics is the same as of the invariant it was translated from.
(3) State definitions in the semantics of Event-B and JML are both the same.
(4) Event-B machines are composed of a single machine variable, an initialising event, and a single standard event. Machines do not *see* any context.

Additionally, we consider a simplified version of substitutions. Rule NAsg in Section 4 operates on substitution in left Figure 3, and the substitution in right Figure 3 is a simplified version of the one on which Rule Any in Section 4 operates, namely, the body of the event in right Figure 3 includes a single event action.

The structure of our proof consists of the following proof steps:

(1) To express Event-B and JML constructs as types in Event-B.
(2) To implement the EventB2Jml translator rules as type transducer rules.
(3) To define a semantics of Event-B types as state transducers.
(4) To define a semantics of JML types as state transducers.
(5) To prove that the semantics of JML is simulated by the semantics of Event-B.

5.1. The Types for Event-B and JML

For Event-B we define types $BPredType$, $AssgType$, and $AnyType$, representing the type of a predicate, the type of a non-deterministic assignment, and the type of an event, together with respective type *constructors* $BPred$, $Assg$, and Any as below. Similarly, we define types $MachineType$, $BInvType$, and $BInitType$ together with respective constructors $Machine$, $BInv$, and $BInit$, for machines, machine invariants, and the distinguished initialisation event. To provide a uniform presentation we assume that all the predicates in Event-B involve three identifiers, hence the reason for the arity of the constructor $BPred$. Thus, predicates P and G above will actually be represented by $BPred(v, v', v')$ and $BPred(v, x, x)$, respectively. Variable v' is the after-value of v. Substitution in left Figure 3 is written $Assg(v, v', BPred(v, v', v'))$ so $BPred(v, v', v')$ models the before-after predicate P. This non-deterministic substitution picks a value

that satisfies P and assigns it to v. In using $BPred$ to represent predicates involving three parameters, in general, the first parameter are the machine variables, the second one are their after-values, and the third one is the local event variable. If the third parameter is missing, as in the case of the substitution in left Figure 3, then we take the third parameter as the second one; if the second one is missing, as in the case of the event guard, then we choose the second parameter to be equal to the third one.

$$
\begin{array}{l}
BPred \in Id \times Id \times Id \rightarrow BPredType \\
Assg \in Id \times Id \times BPredType \rightarrow AssgType \\
Any \in Id \times BPredType \times Id \times Id \times BPredType \rightarrow AnyType \\
Machine \in BInvType \times BInitType \times AnyType \rightarrow MachineType \\
BInv \in Id \times Id \times Id \rightarrow BInvType \\
BInit \in Id \times Id \times BPredType \rightarrow BInitType
\end{array}
$$

We use types to abstract away unnecessary details in the Event-B to JML translation. Types defined for Event-B predicates, substitutions, and machines are the sets $BPredicate$, $BSubs$, and $BMachine$ respectively. The initialisation event is implemented as a non-deterministic assignment, $BInitType \subseteq AssgType$.

$$
\begin{array}{l}
\textsf{sets} \\
\quad BPredicate\ BSubs\ BMachine \\
\textsf{invariants} \\
\quad BPredType \subseteq BPredicate \\
\quad AssgType \subseteq BSubs \\
\quad AnyType \subseteq BSubs \\
\quad BInvType \subseteq BPredType \\
\quad BInitType \subseteq AssgType \\
\quad MachineType \subseteq BMachine
\end{array}
$$

Types for JML constructs are defined in a similar way as for Event-B, but many more components are involved. We consider JML predicate definitions involving,

— equality of identifiers, as in `x=y`,
— existentially (`\exists T x; ...`) quantified predicates in JML,
— JML boolean operators `&&` (logical and), `||` (logical or),
— constant predicates `true`, `false`,
— JML predicates with identifier values taken from the pre-state, as in `\old(P)`,
— JML predicates with identifier values taken from the post-state (the default).

These types pop up in the translation to JML of Event-B constructs. Hence, Event-B predicates are translated to semantically equivalent JML predicates, non-deterministic assignments are translated to existentially quantified predicates (rule NAsg in Section 4), and events to JML specified methods (rule Any). JML predicates appear in the `requires` (pre-condition) and `ensures` (post-condition) parts of JML-specified methods. For each predicate listed we define a type (a subset of $JmlPredicate$), e.g. $JmlExistsType$, $JmlAndType$, etc. and a constructor. Constructors are shown below. Notice that, as for Event-B predicates, a simple JML predicate represented by the $JmlPred$ constructor involves exactly three identifiers. $JmlMeth$ constructs a JML method specification from two predicates representing its pre- and post-condition. $JmlAlso$ combines two method specifications. $JmlOld$ matches the JML `\old` operator, and $JmlBecomes$ relates the value of machine variables in the pre-state with the value in the post-state.

$$
\begin{array}{|l}
\hline
JmlExists \in Id \times JmlPredicate \rightarrow JmlExistsType \\
JmlAnd \in JmlPredicate \times JmlPredicate \rightarrow JmlAndType \\
JmlTrue \in JmlTrueType \\
JmlFalse \in JmlFalseType \\
JmlBecomes \in Id \times Id \rightarrow JmlBecomesType \\
JmlNot \in JmlPredicate \rightarrow JmlNotType \\
JmlOld \in JmlPredicate \rightarrow JmlOldType \\
JmlPred \in Id \times Id \times Id \rightarrow JmlPredType \\
JmlMeth \in JmlPredicate \times JmlPredicate \rightarrow JmlMethType \\
JmlAlso \in JmlMethType \times JmlMethType \rightarrow JmlAlsoType \\
\hline
\end{array}
$$

5.2. The Event-B to JML Translation Rules

Translation rules are modelled as total functions transforming Event-B types into JML types and predicates. We consider five functions, one for translating Event-B predicates, one for translating simple assignments, one for standard events, one for initialising event, one for machine invariants and one for Event-B machines. These functions are shown below. An Event-B predicate is transformed into a JML predicate, a nondeterministic assignment is transformed into a JML predicate that is used within a JML postcondition specification, a standard event originates a JML method specification, an initialising event relates to the postcondition of a JML (Java) constructor, a machine invariant originates a JML class invariant, and a machine is translated to a JML abstract class specification.

$$
\begin{array}{|l}
\hline
BPred2Jml \in BPredicate \rightarrow JmlPredicate \\
Assg2Jml \in AssgType \rightarrow JmlPredicate \\
Any2Jml \in AnyType \rightarrow JmlMethType \\
BInit2Jml \in BInitType \rightarrow JmlPredicate \\
BInv2Jml \in BInvType \rightarrow JmlInvType \\
Machine2Jml \in MachineType \rightarrow JmlClassSpecType \\
\hline
\end{array}
$$

The translation rules for Event-B substitutions are expressed as axioms. Simple substitutions are presented in left Figure 3, where P is a before-after predicate involving variables v and v' (the machine variables before and after the assignment). The values after the assignment must satisfy P. Since this substitution is not guarded (meaning there is no condition required for the substitution to be executed), its translation amounts to the translation of P, as shown by Rule NAsg in Section 4. This is expressed by the following axiom, where $BPred(v, v', v')$ models P, $JmlOld$ evaluates P in the pre-state, and $JmlBecomes$ expresses how the values of the machine variables change from the prestate to the poststate.

$$
\begin{array}{|l}
\hline
\forall v, v'.\ (v \in Id \wedge v' \in Id \Rightarrow \\
\quad Assg2Jml(Assg(v, v', BPred(v, v', v'))) \\
\qquad = \\
\quad JmlExists(v', JmlAnd(JmlOld(BPred2Jml(BPred(v, v', v'))), JmlBecomes(v, v')))) \\
\hline
\end{array}
$$

Rule Any in Section 4 formalises the translation of the Event-B event shown in Figure 3. The event implements a guarded substitution. The translation of the event is axiomatised below. The axiom includes two method specification cases, one for each parameter of the $JmlAlso$ constructor. The first case is preconditioned by the `guard` method (see Section 4), constructed with $JmlExists$; the second case is preconditioned by its negation.

$$
\begin{array}{l}
\forall v, v', x.\ (v \in Id \wedge v' \in Id \wedge x \in Id \Rightarrow \\
\ Any2Jml(Any(x, BPred(v, x, x), v, v', BPred(v, v', x))) \\
\ \ = \\
\ JmlAlso(\\
\ \ JmlMeth(JmlExists(x, BPred2Jml(BPred(v, x, x))), \\
\ \ \ JmlExists(x, JmlAnd(JmlOld(BPred2Jml(BPred(v, x, x))), \\
\ \ \ \ \ \ \ \ \ \ JmlExists(v', JmlAnd(JmlOld(BPred2Jml(BPred(v, v', x))), \\
\ \ \ \ \ \ \ \ \ \ \ \ \ \ JmlBecomes(v, v')))))), \\
\ \ JmlMeth(JmlNot(JmlExists(x, BPred2Jml(BPred(v, x, x)))), JmlTrue) \\
\ \ \ ) \\
)
\end{array}
$$

5.3. Semantics

Event-B and JML semantics are defined as state transition systems. Hence, given any two states a, b and an Event-B substitution S, we show that if the pair (a, b) belongs to the state transition of the JML semantics of the translation of the type of S ($MachineType$, $AssgType$, $AnyType$, etc) into JML, then (a, b) also belongs to the Event-B semantics of the type of S. We assume that state definitions in the semantics of Event-B and in JML are both the same. For each construct in Event-B and JML, we define a predicate (a boolean function, more precisely) that holds whenever a given pair of states represents a valid transition for that construct. Similarly, the semantics of Event-B (JML) predicates is given by a predicate in the semantic domain that holds in a state whenever the B (JML) predicate is true in that state.

The types involved in the definition of the Event-B semantic functions are shown next. States are partial functions that associate some value to an identifier. Function $BPredSem$ furnishes a semantics to Event-B predicates, $BAssgSem$ to simple substitutions, $BAnySem$ to events, $BInitSem$ to the initialising event, and $MachineSem$ to machines. $BPredSem$ is parametrised by the state in which it is evaluated, the other types are all parametrised by two states, the pre- and the post-states, except for $MachineSem$ that rather takes the pre- and post-state of the standard event and the pre- and post-event of the initialising event. The semantics of $BAssgSem$, $BAnySem$, and $BInitSem$ depend on the machine invariant.

$$
\begin{array}{l}
State = (Id \nrightarrow Value) \\
BPredSem \in BPredicate \times State \rightarrow BOOL \\
BAssgSem \in AssgType \times State \times State \rightarrow BOOL \\
BAnySem \in AnyType \times BInvType \times State \times State \rightarrow BOOL \\
BInitSem \in BInitType \times BInvType \times State \times State \rightarrow BOOL \\
MachineSem \in MachineType \times State \times State \times State \times State \rightarrow BOOL
\end{array}
$$

Take the substitution in left Figure 3, a transition from state a to state b for the execution of the substitution exists if there is some post-state value y for v' such that the predicate $P(v, y)$ holds in state a, and the valuation of identifier v in state b is equal to y. This is asserted in the following axiom, where $a \Leftarrow \{z, w\}$ denotes state a modified such that the value of identifier z in the domain of a is equal to w.

$$
\begin{array}{l}
\forall v, v', a, b.\ (\ v \in dom(a) \wedge v' \in Id \wedge v' \notin dom(a) \wedge v' \neq v \wedge a \in State \wedge b \in State\ \Rightarrow \\
\quad (\\
\qquad BAssgSem(Assg(v, v', BPred(v, v', v')), a, b) \\
\qquad\quad \Leftrightarrow \\
\qquad \exists y.\ (\ y \in Value \wedge \\
\qquad\quad BPredSem(BPred(v, v', v'), (a \lhd\!\!\lhd \{v', y\})) \wedge \\
\qquad\quad b = (a \lhd\!\!\lhd \{v, y\})\ ) \\
\quad)\ )
\end{array}
$$

For the event in right Figure 3, a transition from some state a to state b is valid under the assumption that the machine invariants holds of the initial state a when (1) predicate $G(v, x)$ holds of a state a that maps some value y to variable x, and (2) there exists some value z for v' such that $P(v, v', x)$ holds of a state a modified so that v' has value z and x has value y, and (3) state b is equal to a but with the value of v set to z, and (4) the machine invariant holds of the final state b. Otherwise, if such a state transition does not exist, then states a and b remain the same. Notice that $a = b$ should actually further be conjoined with the negation of the existential property.

$$
\begin{array}{l}
\forall v, v', x, a, b.\ (\ v \in dom(a) \wedge v \in dom(b) \wedge v' \in Id \wedge x \in Id \wedge v' \notin dom(a) \wedge \\
\qquad v' \notin dom(b) \wedge x \notin dom(a) \wedge x \notin dom(b) \wedge a \in State \wedge b \in State \Rightarrow \\
\quad (\\
\qquad BAnySem(Any(x, BPred(v, x, x), v, v', BPred(v, v', x)), a, b) \\
\qquad\quad \Leftrightarrow \\
\qquad (\ BPredSem(BInv(v, vp, vp), a)\ \Rightarrow \\
\qquad\quad \exists y.\ (\ y \in Value \wedge \\
\qquad\qquad BPredSem(BPred(v, x, x), (a \lhd\!\!\lhd \{x, y\})) \wedge \\
\qquad\qquad \exists z.\ z \in Value \wedge BPredSem(BPred(v, v', x), (a \lhd\!\!\lhd \{x, y\} \lhd\!\!\lhd \{v', z\})) \wedge \\
\qquad\qquad b = (a \lhd\!\!\lhd \{v, z\}) \wedge \\
\qquad\qquad BPredSem(BInv(v, vp, vp), b)\ ) \\
\qquad)\ \vee \\
\qquad (\ a = b\ ) \\
\quad)
\end{array}
$$

$BInitSem$ is defined similarly to $BAssgSem$ (not shown here) yet it must further ensure that the machine invariant holds of the post-state, that is, the invariant holds of the state produced when creating the machine. The semantics of a machine must thus ensure the semantics of the initialisation and standard event for respective initial and final states c and d, and a and b.

$$
\begin{array}{l}
\forall v, vp, x, a, b, c, d.\ (\ v \in dom(a) \wedge v \in dom(b) \wedge v \in dom(c) \wedge v \in dom(d) \wedge \\
\quad vp \in Id \wedge vp \notin dom(a) \wedge vp \notin dom(b) \wedge vp \notin dom(c) \wedge vp \notin dom(d) \wedge \\
\quad x \in Id \wedge x \notin dom(a) \wedge x \notin dom(b) \wedge x \notin dom(c) \wedge x \notin dom(d) \wedge \\
\quad vp \neq v \wedge x \neq v \wedge a \in State \wedge b \in State \wedge c \in State \wedge d \in State\ \Rightarrow \\
\qquad MachineSem(\ Machine(BInv(v, vp, vp), \\
\qquad\qquad\qquad\qquad BInit(v, vp, BPred(v, vp, vp)), \\
\qquad\qquad\qquad\qquad Any(x, BPred(v, x, x), v, vp, BPred(v, vp, x))), a, b, c, d\ ) \\
\qquad\quad \Leftrightarrow \\
\quad (\ BInitSem(BInit(v, vp, BPred(v, vp, vp)), BInv(v, vp, vp), c, d) \wedge \\
\qquad BAnySem(Any(x, BPred(v, x, x), v, vp, BPred(v, vp, x)), BInv(v, vp, vp), a, b) \\
\quad) \\
)
\end{array}
$$

The semantics of JML methods is structured similarly, by predicates asserting valid state transitions. As for Event-B semantics, we define boolean functions for JML predicates and for JML methods. Their types are shown below. There are two semantic functions for predicates, $JmlPredSem$, which evaluates a predicate in a particular state, and $JmlPostPredSem$ that relates the semantics of a predicate through two states.

$$\begin{array}{l} JmlPredSem \in JmlPredicate \times State \to BOOL \\ JmlPostPredSem \in JmlPredicate \times State \times State \to BOOL \\ JmlMethSem \in JmlMethType \times JmlInvType \times State \times State \to BOOL \end{array}$$

Notice that $JmlPostPredSem$ takes two states as arguments. The first argument represents the pre-state, the second the post-state. This is needed since JML predicates can only refer to the pre-state explicitly in the post-condition of a JML specification (using the `\old` operator). How these states relate is best seen in the axioms below.

$$\begin{array}{l} \forall p, a, b.\ (\ a \in State \land b \in State \land p \in JmlPredicate\ \Rightarrow \\ \quad JmlPostPredSem(p, a, b) = JmlPredSem(p, b)\ ) \\ \\ \forall p, a, b.\ (\ a \in State \land b \in State \land p \in JmlPredicate\ \Rightarrow \\ \quad JmlPostPredSem(JmlOld(p), a, b) = JmlPredSem(p, a)\ ) \end{array}$$

The predicate $JmlBecomes$ describes how the value of a variable can change from the pre-state to the post-state of a substitution. Hence, $JmlPostPredSem$ holds of two states a, b whenever the post-state state b is the same pre-state a where identifier v' has been removed and the value it had is given to identifier v. That is, the value of variable v' in the pre-state is represented by the value of v in the post-state. The Event-B symbol $\lhd$ is called domain substraction, hence "$\{v'\} \lhd a$" removes all the pairs from a whose first element is v'.

$$\begin{array}{l} \forall v, v', a, b.\ (\ v \in Id \land v' \in Id \land a \in State \land b \in State \land \\ \qquad\qquad v \in dom(a) \land v \in dom(b) \land v' \notin dom(b) \land v' \neq v\ \Rightarrow \\ \quad (\ JmlPostPredSem(JmlBecomes(v, v'), a, b) \\ \qquad \Leftrightarrow \\ \quad\ b = (\{v'\} \lhd a) \lhd\!\!+ \{v, a(v')\}\ )\ ) \end{array}$$

In Section 5.1, types are given for several JML predicates. We define axioms relating the semantics of ones in terms of the others, as shown below.

$$\begin{array}{l} \forall p, q, s1, s2.\ (\ s1 \in State \land s2 \in State \land p \in JmlPredicate \land q \in JmlPredicate\ \Rightarrow \\ \quad (\ JmlPostPredSem(JmlAnd(p, q), s1, s2) \\ \qquad \Leftrightarrow \\ \quad\ JmlPostPredSem(p, s1, s2) \land JmlPredSem(q, s1, s2)\ \ ) \\ )\end{array}$$

Other boolean operators are axiomatised similarly. The axiom for the existential operator is show below. It says that the semantic predicate of the existential $JmlExists(x, p)$ holds of two states $s1$ and $s2$ whenever there is some value y such that the semantic predicate of p holds of the states $s1$ modified so that x has value y and $s2$.

$$\begin{array}{l} \forall p, x, s1, s2.\ (s1 \in State \land s2 \in State \land p \in JmlPredicate \land \\ \quad x \in Id \land x \notin dom(s1) \land x \notin dom(s2)\ \Rightarrow \\ \qquad (\ JmlPostPredSem(JmlExists(x, p), s1, s2) \\ \qquad\quad \Leftrightarrow \\ \qquad\quad \exists y.\ (\ \ y \in Value \land JmlPostPredSem(p, st \lhd\!\!+ \{x, y\}, s2)\ ) \\ \qquad)\ ) \end{array}$$

The semantics of a JML method states that two states a and b define a valid transition whenever the conjunction of the machine invariant inv and the method post-condition ens holds of both states, given that the conjunction of the machine invariant and the method pre-condition req holds of the pre-state a. This is formalised by $JmlMethSem$ below. We further define the semantics $JmlConstructorSem$ of the class constructor (not shown here) similar to $JmlMethSem$ except that the machine invariant not need to hold in the pre-state. And, $JmlAlso$ that combines two $JmlMethSem$ syntaxes.

$$
\begin{array}{l}
\forall req, ens, inv, a, b.\ (\ a \in State \land b \in State \land \\
\quad req \in JmlPredicate \land ens \in JmlPredicate \land inv \in JmlInvType\ \Rightarrow \\
\quad\ JmlMethSem(JmlMeth(req, ens), inv, a, b) \\
\qquad \Leftrightarrow \\
\quad\ (\ JmlPredSem(JmlAnd(req, inv), a) \\
\qquad \Rightarrow \\
\quad\ \ JmlPostPredSem(JmlAnd(ens, inv), a, b) \\
\quad\ ) \\
\ )
\end{array}
$$

The semantics of a whole JML class specification incorporates the semantics of the constructor, the specified method, and the machine invariant as shown below.

$$
\begin{array}{l}
\forall a, b, c, d, rc, ec, req, ens, inv.\ (\ a \in State \land b \in State \land c \in State \land d \in State \land \\
\quad req \in JmlPredicate \land ens \in JmlPredicate \land \\
\quad rc \in JmlPredicate \land ec \in JmlPredicate \land inv \in JmlInvType\ \Rightarrow \\
\quad JmlClassSem(JmlClassSpec(inv, \\
\qquad\qquad\qquad JmlConstructor(rc, ec), \\
\qquad\qquad\qquad JmlMeth(req, ens)), a, b, c, d) \\
\qquad \Leftrightarrow \\
\quad (\ JmlConstructorSem(JmlConstructor(rc, ec), inv, c, d) \land \\
\quad\ JmlMethSem(JmlMeth(req, ens), inv, a, b) \land \\
\quad\ JmlPredSem(inv, a) \\
\quad)\ )
\end{array}
$$

5.4. The Proof Statement

The semantics of Event-B substitutions and the semantics of JML are related as shown in Figure 4. Two states valid for an Event-B substitution should also be valid when the substitution is translated JML. We assume that translation of predicates is correct. The expression of the relation shown in the figure is achieved through three theorems, one for Event-B simple substitutions, one for events, and one for machines.

THEOREM 5.1. *The translation of a simple substitution is sound:*

$$
\begin{array}{l}
\forall v, v', a, b.\ (\ v \in Id \land v' \in Id \land a \in State \land b \in State \land \\
\qquad v \in dom(a) \land v \in dom(b) \land v' \notin dom(a) \land v' \notin dom(b) \land v' \neq v \Rightarrow \\
\quad (\ JmlPostPredSem(Assg2Jml(Assg(v, v', BPred(v, v', v'))), a, b) = TRUE \\
\qquad \Rightarrow \\
\quad\ BAssgSem(Assg(v, v', BPred(v, v', v')), a, b) = TRUE \\
\quad) \\
\ )
\end{array}
$$

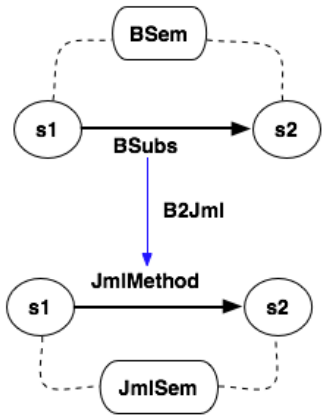

Fig. 4. Proof structure

THEOREM 5.2. *The translation of an event is sound:*

$$
\begin{array}{l}
\forall x, v, v', a, b.\ (\ x \in Id \wedge v \in Id \wedge v' \in Id \wedge a \in State \wedge b \in State \wedge \\
\quad v \in dom(a) \wedge v \in dom(b) \wedge v' \notin dom(a) \wedge v' \notin dom(b) \wedge \\
\quad v' \neq v \wedge x \notin dom(a) \wedge x \notin dom(b) \wedge x \neq v \wedge x \neq v' \Rightarrow \\
(\ JmlMethSem(Any2Jml(Any(x, BPred(v, x, x), v, v', BPred(v, v', x))), a, b) = TRUE \\
\quad \Rightarrow \\
\quad BAnySem(Any(x, BPred(v, x, x), v, v', BPred(v, v', x)), a, b) = TRUE \\
\ )\\
)
\end{array}
$$

THEOREM 5.3. *The translation of a machine is sound:*

$$
\begin{array}{l}
\forall x, v, vp, s1, s2, s3, s4.\ (s1 \in State \wedge s2 \in State \wedge s3 \in State \wedge s4 \in State \wedge \\
\quad v \in dom(s1) \wedge v \in dom(s2) \wedge v \in dom(s3) \wedge v \in dom(s4) \wedge \\
\quad vp \notin dom(s1) \wedge vp \notin dom(s2) \wedge vp \notin dom(s3) \wedge vp \notin dom(s4) \wedge \\
\quad x \notin dom(s1) \wedge x \notin dom(s2) \wedge x \notin dom(s3) \wedge x \notin dom(s4) \wedge \\
\quad vp \neq v \wedge x \neq v \wedge x \neq vp \ \Rightarrow \\
\quad (\ JmlClassSem(\ Machine2Jml(Machine(BInv(v, vp, vp), \\
\qquad\qquad\qquad\qquad BInit(v, vp, BPred(v, vp, vp)), \\
\qquad\qquad\qquad\qquad Any(x, BPred(v, x, x), v, vp, BPred(v, vp, x)))), s1, s2, s3, s4\ ) \\
\qquad\qquad \Rightarrow \\
\quad MachineSem(\ Machine(\ BInv(v, vp, vp), \\
\qquad\qquad\qquad BInit(v, vp, BPred(v, vp, vp)), \\
\qquad\qquad\qquad Any(x, BPred(v, x, x), v, vp, BPred(v, vp, x)) \\
\qquad\qquad\qquad), s1, s2, s3, s4\ ) \\
\quad)\ )
\end{array}
$$

Notice that we state a weaker form of semantic correctness in the above three theorems. As mentioned before, we want to guarantee that any valid transition of the JML method target of the translation must also be a valid transition of the source Event-B substitution. That is, an Event-B substitution must be capable of simulating any valid transition of its JML method counterpart. The JML translation then constitutes a sort of "refinement" of the Event-B specification.

The theorems above are discharged, under the assumptions below. The translation of an Event-B predicate (machine invariant) on some given variables produces the same predicate (class invariant) as in JML.

$$\forall t, u, z.\ (\ t \in Id \land u \in Id \land z \in Id\ \Rightarrow \\ \quad BPred2Jml(BPred(t,u,z)) = JmlPred(t,u,z)\ )$$

$$\forall t, u, z, s.\ (\ t \in Id \land u \in Id \land z \in Id \land s \in State\ \Rightarrow \\ \quad JmlPredSem(JmlPred(t,u,z), s) = BPredSem(BPred(t,u,z), s)\ )$$

$$\forall t, u, z.\ (\ t \in Id \land u \in Id \land z \in Id\ \Rightarrow \\ \quad BInv2Jml(BInv(t,u,z)) = JmlInv(t,u,z)\ )$$

$$\forall t, u, z, s.\ (\ t \in Id \land u \in Id \land z \in Id \land s \in State\ \Rightarrow \\ \quad JmlPredSem(JmlInv(t,u,z), s) = BPredSem(BInv(t,u,z), s)\ )$$

6. DISCUSSION

Formalising a proof in a tool is often a trade between the level of detail the proof should provide and the feasibility of discharging it in an automated tool. In our earlier work on the definition of the EB2Jml operator [Cataño et al. 2012b], carrier sets and constants in Event-B are translated to JML class specification fields. A bespoken JML class specification `BSet` is used to represent Event-B sets and operations over sets, and, in general, mathematical types such as relations, sequences and functions, which are heavily used in Event-B, are represented in JML as specification classes. The soundness proof presented in Section 5 relies on the correctness of the representation of Event-B mathematical types by respective JML types, and, indeed, attempting to represent those in the logic of Event-B would largely increase the complexity of the proof. Nonetheless, another approach can be attempted instead. One can use existing JML machinery [Burdy et al. 2005; Breunesse et al. 2005] to verify a range of JML specified properties that one is certain the Event-B mathematical types fulfil.

The five steps enumerated in the beginning of Section 5 outlines a general approach to prove the soundness of the translation between two formal languages, a host and a target, based on transition systems, using an automated tool. And, Figure 4 in Section 5.4 suggests a granularity level for those transitions in the underlying logic.

First, the syntax 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 between a deep and a shallow embedding). Second, the translation from the host to the target language is modelled in logic as taking syntax constructs from one language to the other. Third and fourth, one provides a type semantics for the the host and the target language, and defines type constructors to build elements of those types. Type states are defined as state transducers. Fifth, a soundness proof is enunciated as relating state transition in the target language with state transitions in the host language. The level of granularity of those transitions effects the level of complexity of the proof. And, to simplify the proof, one makes valid assumptions on how the states in the transition system for the host and the target language are represented. Choosing different representations for the states will certainly force you to represent the program memory of both languages in the logic of the automated tool, and to establish the relation of both program memories, which would add up to the complexity of the proof.

We have discussed a couple of simplifications and assumptions made to our formal model of Event-B machines in the beginning of Section 5. We considered that the body

of an event is composed of a single assignment, yet, in general, in Event-B, multiple assignments can be executed in parallel using the || operator. The semantics of parallel composition of assignments is amenable to a way of a sequential array of assignments by incorporating a temporal variable. Hence, the parallel composition $x := E \mid\mid y := F$ can be expressed as the sequence of assignments $temp := F \ ; \ x := E \ ; \ y := temp$. Our semantics of Event-B can easily adapted to incorporate these sequences.

7. CONCLUSION

REFERENCES

ABRIAL, J. R. 1996. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, New York, NY, USA.

ABRIAL, J. R. 2010. *Modeling in Event-B: System and Software Design*. Cambridge University Press, New York, NY, USA.

BARNETT, M., CHANG, B. E., DELINE, R., JACOBS, B., AND LEINO, K. R. M. 2006. Boogie: A modular reusable verifier for object-oriented programs. In *FMCO 2005*. LNCS Series, vol. 4111. Springer, Amsterdam, The Netherlands.

BARNETT, M., LEINO, K. R. M., AND SCHULTE, W. 2004. The Spec# programming system: An overview. In *CASSIS*. LNCS Series, vol. 3362. Springer, Marseille, France, 49–69.

BODEVEIX, J.-P., FILALI, M., AND MUÑOZ, C. 1999. 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)*. Springer, Toulouse, France, 33–49.

BOULANGER, J.-L. 2003. ABTools: Another B tool. In *Proceedings of ACSD (Application of Concurrency to System Design)*. IEEE Computer Society, Guimaraes, Portugal.

BOULTON, R. J., GORDON, A. D., GORDON, M. J. C., HARRISON, J., HERBERT, J., AND TASSEL, J. V. 1992. Experience with embedding hardware description languages in HOL. In *International Conference on Theorem Provers in Circuit Design (TPCD)*. North-Holland, Nijmegen, The Netherlands, 129–156.

BOUQUET, F., DADEAU, F., AND GROSLAMBERT, J. 2005. Checking JML specifications with B machines. In *Proceedings of ZB*. Lecture Notes in Computer Science Series, vol. 3455. Springer Verlag, Guildford, U.K, 435–454.

BOUQUET, F., DADEAU, F., AND GROSLAMBERT, J. 2006. JML2B: Checking JML specifications with B machines. In *Proceedings of B: Formal Specification and Development in B*, J. Julliand and O. Kouchnarenko, Eds. Lecture Notes in Computer Science Series, vol. 4355. Springer Berlin / Heidelberg, Besanĉon, France, 285–288.

BOWEN, J. AND GORDON, M. 1994. Z and HOL. In *Z User Workshop, Cambridge 1994*. Workshops in Computing. Springer-Verlag, Cambridge, U.K, 141–167.

BREUNESSE, C.-B., CATAÑO, N., HUISMAN, M., AND JACOBS, B. 2005. Formal methods for smart cards: An experience report. *Science of Computer Programming 55,* 1-3, 53–80.

BURDY, L., CHEON, Y., COK, D., ERNST, M., KINIRY, J., LEAVENS, G. T., LEINO, K. R. M., AND POLL, E. 2005. An overview of JML tools and applications. *International Journal on STTT 7,* 3, 212–232.

BUTLER, M. J., JONES, C. B., ROMANOVSKY, A., AND TROUBITSYNA, E., Eds. 2006. *Rigorous Development of Complex Fault-Tolerant Systems [FP6 IST-511599 RODIN project]*. LNCS. Springer.

CATAÑO, N., RIVERA, V., AND LEINO, R. 2012a. The EventB2Dafny rodin plug-in. In *Proceedings of 2nd Workshop on Developing Tools as Plug-ins (TOPI)*. IEEE Xplore, Zurich, Switzerland, 49–54.

CATAÑO, N. AND RUEDA, C. 2010. Matelas: A predicate calculus common formal definition for social networking. In *Proceedings of ABZ 2010*, M. Frappier, Ed. Lecture Notes in Computer Science Series, vol. 5977. Springer Berlin Heidelberg, Québec, Canada, 259–272.

CATAÑO, N., WAHLS, T., RIVERA, V., AND RUEDA, C. 2012b. Translating Event-B machines to JML specifications. Submitted to ASE.

CATAÑO, N., WAHLS, T., RUEDA, C., RIVERA, V., AND YU, D. 2012c. Translating B machines to JML specifications. In *27th ACM Symposium on Applied Computing, Software Verification and Testing track (SAC-SVT)*. ACM, Trento, Italy.

DÉHARBE, D. 2011. Integration of SMT-solvers in B and Event-B development environments. *Science of Computer Programming*. Article in Press.

EDMUNDS, A. AND BUTLER, M. 2010. Tool support for Event-B code generation. In *WS-TBFM2010*. John Wiley and Sons, Québec, Canada.

EDMUNDS, A. AND BUTLER, M. 2011. Tasking Event-B: An extension to Event-B for generating concurrent code. In *PLACES 2011*. Springer, Saarbrucken, Germany.

GOOGLE INC. 2012. The android platform. http://developer.android.com/design/index.html.

GORDON, M. 1989. Mechanizing programming logics in higher order logic. In *in Current Trends in Hardware Verification and Automated Theorem Proving*. Springer-Verlag, New York, NY, USA, 387–439.

JACQUEL, M., BERKANI, K., DELAHAYE, D., AND DUBOIS, C. 2011. 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. Springer-Verlag, Berlin, Heidelberg, 253–268.

JAEGER, É. AND DUBOIS, C. 2007. Why would you trust B? In *Proceedings of LPAR*. LNCS Series, vol. 4790. ACM Digital Library, Armenia, USSR, 288–302.

JIN, D. AND YANG, Z. 2008. Strategies of modeling from VDM-SL to JML. In *International Conference on ALPIT*. IEEE Computer Society, Liaoning, China, 320–323.

LEAVENS, G., BAKER, A., AND RUBY, C. 2006. Preliminary design of JML: A behavioral interface specification language for Java. *ACM SIGSOFT 31,* 3, 1–38.

LEAVENS, G., POLL, E., CLIFTON, C., CHEON, Y., RUBY, C., COK, D., MÜLLER, P., KINIRY, J., AND CHALIN, P. 2012. JML reference manual. http://www.eecs.ucf.edu/∼leavens/JML/jmlrefman/jmlrefman_toc.html.

MEJER, R. 2012. *Professional Android 4 Application Development*. Wrox, Indiana, USA.

MÉRY, D. AND SINGH, N. K. 2011. Automatic code generation from Event-B models. In *Proceedings of the Second SoICT*. SoICT '11. ACM, Hanoi, Vietnam.

PERCHY, S. AND CATAÑO, N. 2012. The Racing Car Game. Available at http://cic.javerianacali.edu.co/~ysperchy/formal-game.

RIVERA, V. AND CATAÑO, N. 2012. The Event-B Planner. Available at http://poporo.uma.pt/Projects/favas/Social-Event_Planner.html.

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

WOODCOCK, J. AND DAVIES, J. 1996. *Using Z: Specification, Refinement, and Proof*. International Series in Computer Science. Prentice-Hall, Inc., New Jersey, USA.

WRIGHT, S. 2009. Automatic generation of C from Event-B. In *Workshop on IM_FMT*. Springer-Verlag, Nantes, France.