Translating Event-B to JML-Specified Java programs

Victor Rivera
The University of Madeira
victor.rivera@m-iti.org

Néstor Cataño
University EAFIT
ncatano@eafit.edu.co

ABSTRACT

We present a translation from Event-B machines to JML-specified Java class implementations and the EventB2Java Rodin plug-in that automates the translation. Producing JML specifications in addition to Java implementations enables users to write bespoke implementations that can then be checked for correctness using existing JML tools. We have validated the proposed translation by applying the EventB2Java tool to various programs and systems.

Categories and Subject Descriptors

D.2.1 [Software Engineering]: Requirements/Specifications—methodologies, tools; D.2.4 [Software/Program Verification]: [validation, formal methods]

General Terms

automated translation, combined formal methods

Keywords


1. INTRODUCTION

Event-B provides a formal notation for the modeling of reactive systems [2]. The notation is supported by the Rodin platform [3], an open-source Eclipse IDE that provides a set of tools for working with Event-B models, e.g. an editor, a proof obligation generator, and provers. This paper presents a full-fledged translation from Event-B to Java and the EventB2Java Eclipse plug-in that implements such a translation. EventB2Java produces JML specifications [21] in addition to Java code. This enables users to write customised Java implementations that can be checked for correctness using existing JML tools [5]. A key feature of our translation from Event-B is that it can be applied to any abstract or refinement Event-B machine without additional user intervention. Previous approaches either can translate only implementation machines [23, 27], or require the user to write tasking bodies defining the flow of execution of machine events [15, 16]. Our approach to code generation of Event-B models gives the user the choice of when to transition to Java, rather than dictating the transition point.

The contributions of this work are (i) the definition of a translation from Event-B to JML-specified Java programs through syntactic rules, and (ii) the implementation of the translation as the EventB2Java Rodin plug-in. We have validated the proposed translation by using EventB2Java to produce JML-specified Java implementations for several Event-B models including various searching and sorting algorithms [1], a model for a social networking application adapted from [10], a social event planner [25] implemented in Android [18, 22], and a model for a massive transportation system [9], among others. Our work enables software developers to use different formal methods techniques and tools at different software development phases. For instance, a developer can use Event-B’s strong support for model design and verification during early stages of software development to produce a fully verified model of the application, and then transition to Java to run the code. The JML specifications produced further enable users to write customised implementations of the code produced by the tool. The user needs then to use existing JML-machinery to check the correctness of the customised code against the JML specifications.

2. PRELIMINARIES

2.1 Event-B

Event-B models are composed of machines and contexts. Three basic relationships are used to structure an Event-B model, namely, a machine sees a context or can refine another machine, and a context can extend another context. Machines contain the dynamic parts of a model (i.e. variables, invariants, events), and contexts the static part of a model (i.e carrier sets, constants). Events are composed of two parts, the guard (the where keyword) and the actions (the then keyword). The actions can only be executed if the guard holds. Each action determines how a machine variable evolves by modeling a variable assignment in Event-B (the := symbol).

A partial example of a social networking abstract Event-B model, adapted from the B model in [10], is depicted in Figure 1. The machine sees a context c (not shown here) that defines carrier sets PERSON (the set of all possible people in the network) and CONTENTS (the set of all possible images, text, ... in the network). The abstract machine declares vari-
variables persons contents owner pages
invariants
inv1 persons ⊆ PERSON
inv2 contents ⊆ CONTENTS
inv3 owner ∈ contents → persons
inv4 pages ∈ contents ↔ persons
inv5 owner ⊆ pages
events
initialisation
begin
    in1 persons, contents, owner, pages := ∅, ∅, ∅, ∅
end
edit_owned
any c1 p1 newc where
    grd1 c1 ∈ contents grd2 p1 ∈ persons
    grd3 owner(c1) = p1
    grd4 newc ∈ CONTENTS \ contents
then
    act1 contents := (contents \ {c1}) ∪ {newc}
    act2 pages := (c1) ∈ pages ∪ (newc) × pages[{c1}]
    act3 owner := (c1) ∈ owner ∪ (newc → p1)
end
end
end

machine permissions refines abstract sees c
variables persons contents owner pages viewp edtp
invariants
invr1 viewp ∈ contents ↔ persons
invr2 edtp ∈ contents ↔ persons
invr3 owner ∈ viewp invr4 owner ∈ edtp
invr5 edtp ∈ viewp invr6 pages ⊆ viewp
events
initialisation
begin
    inr1 persons, contents, owner, pages := ∅, ∅, ∅, ∅
    inr2 viewp, edtp := ∅, ∅
end
edit_owned extends edit_owned
then
    actr1 viewp := (c1) ∈ viewp ∪ (newc) × viewp[{c1}]
    actr2 edtp := (c1) ∈ edtp ∪ (newc → p1)
end
end

Figure 1: Event-B machine for social networking

ables persons (the set of people actually in the network), contents (the set of content actually in the network), owner (a total surjection mapping each content item to its owner), and pages (a total surjective relation indicating which content items are visible to which people). Invariant inv5 ensures that each content item is visible to its owner. Event-B provides notations ⇔ for a relation, → for a total surjective function, and ↔ for a total surjective relation.

The initialisation event ensures that all of these sets, functions and relations are initially empty. The symbol ∅ represents the empty set. The abstract machine further defines the event edit_owned that allows a user of the network to edit an owned content item. Events can be executed/triggered when their guards (the part after the where) hold. Hence, edit_owned can execute whenever there is a person p1 in the network that owns a content item c1 and there is at least another new content item newc. The meaning of an event is the meaning of the actions in its body (the part after the then). edit_owned event’s actions replace a content item c1 with a new content item newc with the same ownership and visibility. The symbol \ represents set difference, \{c1\} a pair of elements, ∈ domain subtraction, which removes from a relation all the pairs with a first element in some set, and × cross-product. The expression pages[{c1}] evaluates the relation pages on the set \{c1\}.

In Figure 1, the refinement machine permissions introduces permissions to the system. It adds variables viewp and edtp which respectively track which people have permission to view and edit each content item. edit_owned extends the abstract event so that the owner of c1 has edit and view permissions on the item (actions actr1 and actr2). The invariants further specify that the owner always has permission to view and edit an item, that a person must have view permission on an item in order to have edit permission, and that an item is not visible to a person who does not have view permission on the item. The machine includes events for editing not owned content, creating a user account, uploading content, manipulating the wall, and publishing content (not shown here).

2.2 JML

JML [21, 5] 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 of JML 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 these types are specified (in JML) and implemented as Java methods.

JML method pre- and post-conditions (indicated by keywords requires and ensures, respectively) are embedded as comments immediately preceding method declarations.

JML predicates are first-order assertions 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 equivalence ↔, 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; E declares a variable x of type T, and E is the expression that must hold for every (some) value of type T.

3. A TRANSLATION EXAMPLE

EventB2Java produces a JML-specified Java class implementation for each machine. Machine permissions in Figure 1 is translated as class permissions in Figure 2. The initialisation event of the machine translates as the class constructor, which allocates memory to the class attributes persons, contents, owner, etc. The post-condition of the constructor (the ensures part) states that all the machine variables are initially empty, the pre-condition (the requires part) imposes no condition on the caller, and the frame-condition (the assignable part) allows the constructor to assign to any variable. Classes BRelation and BSet are built-in to EventB2Java and implement mathematical relations and sets respectively [12]. Carrier sets are
translated to Java as elements of type BSet. As carrier sets contain all possible elements of type Integer, they are allocated memory to contain elements from MIN to MAX integer values. Additionally, JML history constraints (the JML constraint keyword) for carrier sets are produced that prevent any method from mutating them or accessing machine variables and synchronise over shared resources. Figure 3 presents the translation of event edit_owned¹, where m is a reference to the machine class implementation, and eventId is the event identifier. In Event-B, events can be triggered when their guards are satisfied. Event triggering results in the execution of the event actions. EventB2Java models this event behaviour by creating two methods: a guard method (e.g. guard_edit_owned) containing the translation of the event guard, and a run method (e.g. run_edit_owned) containing the translation of the event body. The run method should only be executed when the guard method returns true. We ensure this by overriding the standard run() Java Thread method to implement Lamport’s Bakery algorithm for the critical section [20]. The critical section is the set of event actions. The implementation of method run() respects Event-B semantics for the execution of events: the guards of two or more events can be evaluated concurrently, whereas only one event can execute (its critical section) at some point. Event actions are executed sequentially for the event in the critical section.

¹The actual code generated by EventB2Java uses getters and mutators for variables defined in m.
machine invariant is translated as a JML class invariant, and
Event-B is translated as a Java class attribute, an Event-B
EB2Java sees
grams is realised through syntactic rules. The translation is
4. TRANSLATION RULES
the effect of simultaneous assignment.

Figure 4: The translation of machine $M$, and the
context $C$ that $M$ sees.

 guard_edit_owned returns true whenever the guard
of the event edit_owned holds. The JML specification of
guard_edit_owned closely follows its implementation. The
specification of run_edit_owned uses two specification cases
(the also keyword). If the event guard holds, the method
implements the effect of replacing content $c$ with content
newc. Methods diff, domainSub, union and cross imple-
ment \setminus, $\sqcup$, union, and $\times$, respectively. We translate
each variable bounded by an any as a method parameter. For in-
stance $c1$, $p1$ and $newc$ in edit_owned are translated as pa-
rameters of guard_edit_owned and run_edit_owned.

In Event-B, the left-hand side of an assignment is always
a variable, and simultaneous assignments assign to differ-
ent variables. We implement the behaviour of simultane-
ous assignments by first calculating the value of the right
hand side of each assignment into a temporary variable (e.g.
contents_tmp), and then assigning the temporary to the
matching field (i.e. contents). Additionally, in the post-
conditions of JML specifications, the translations of all refer-
ences to machine variables on the right-hand side of Event-B
assignments are wrapped by the JML old operator to give
the effect of simultaneous assignment.

4. TRANSLATION RULES
Our translation from Event-B to JML-specified Java pro-
grams is realised through syntactic rules. The translation is
implemented with the aid of an EB2Prog operator that tran-
lates an Event-B machine and any context that it sees to a
JML-specified Java class implementation. EB2Prog relies
on EB2Java that produces only Java code and EB2Jml that
produces only JML specifications, e.g. a machine variable in
Event-B is translated as a Java class attribute, an Event-B
machine invariant is translated as a JML class invariant, and
constants in Event-B are translated as Java class attributes
which are constrained by a JML constraint specification.

Figure 4 presents Rule M, which translates a machine $M$
that sees context $C$ (not shown). We discharge all the ma-
chine proof obligations in Event-B before the machine is
translated\(^2\), so that proof obligations and closely associated
Event-B constructs, namely, witness and variant, need not be
considered in the translation. 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.

Machines are translated as a JML-specified Java class, and
the translation of a machine incorporates the context the
machine sees. Hence, the Java class includes the transla-
tion of carrier sets, constants, axioms and theorems declared in
the context, and the variables and invariants declared in the
machine. In Event-B, all components of a refined machine
are included in a refinement machine, either explicitly (vari-
ables, initialisations, guards and actions of a refining event
defined using refines) or implicitly (invariants, guards and
actions of a refining event defined using extends). Hence, a
refinement machine can be translated in exactly the same
manner as an abstract machine, as long as all included com-
ponents from refined machines are included and translated.

A machine initialisation event is translated as the con-
structor of the Java class obtained as translation of the ma-
chine. The constructor includes a JML post-condition that
accounts for the initial values of the class attributes. Every
other event is translated as a Java Thread that includes a
reference to the machine class implementation (see rule
Any below). Since event guards are executed concurrently;
once the guard of an event is true, that event might exe-
cute its actions. There can be just one event at the time
executing its actions. The event Java Thread implement-
ation includes three methods: a guard_evt method that
tests if the guard of the event evt holds, a run_evt method
that models the execution of evt, and a run() method that
overrides the corresponding Java Thread method to imple-
mant Port's Bakery algorithm for the critical section
[20]. Expression GuardValue<Type>.next() returns a value
of type Type that satisfies the guard event.

In rule Any, variables bounded by an any construct are
translated as parameters of the run_evt and guard_evt
methods. The helper operator Mod calculates the set of
variables assigned by the actions of the event. Operator
Pred translates an Event-B predicate or expression to its
JML counterpart. The JML specification of run_evt 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 mod-
ify the system state. Since the effective pre-condition of
a JML method with multiple specification cases (separated
by also in JML) is the disjunction of the pre-conditions
of each case, the pre-condition of a run_evt method is al-
ways true. Hence, even though we translate guards as pre-
conditions, no method in the translation result has a pre-
condition. Rather, the translation of the guard determines
\(^2\) However, EB2Java can translate models with undis-
charged proof obligations. This is quite useful when ex-
perimenting with an Event-B model under construction.
which behaviour the method must exhibit. Therefore, the
effective pre-condition of run_edit_owned is just true.

\[
\text{EB2Jml}(I(s, c, v)) = T \quad \text{(Inv)}
\]

\[
\text{EB2Jml}(\text{invariants } I(s, c, v)) =
/\!* \text{ public invariant } I;*
\]

As axioms are often used to specify properties of
constants, they are translated as invariants. In Event-B,
theorems should be provable from axioms, matching the
semantics of the invariant_redundantly clause in JML.

\[
\text{EB2Jml}(\text{axioms } X(s, c)) =
/\!* \text{ public invariant } X;*
\]

In Event-B, every event must maintain the machine
invariant. In JML, invariants state properties that must hold
in every visible system state, specifically after the execution
of the class constructor and after a method is invoked. This
is semantically equivalent to conjoining the invariant to the
post-condition of each method and the constructor. Since
is semantically equivalent to conjoining the invariant to the
post-condition of the class constructor and after a method is invoked. This

\[
\text{run}(\text{event } x) \quad \text{where } G(s, c, v, x)
\]

\[
\text{public class } \text{evt} \text{ extends } \text{Thread}
\]

\[
/\!* \text{ requires } true;\]

\[
\text{assignable } \text{\{}\text{private } M = \text{\}}
\text{int } \text{\{}\text{private } \text{int } \text{\{}\text{event}\text{Id};\text{\}}}
\]

\[
/\!* \text{ requires } true;\]

\[
\text{assignable } \text{\{}\text{\}} \text{\{}\text{private boolean } \text{\}}
\text{guard_evt}(\text{Type } x) \text{\{}\text{\}}
\text{return } G;
\]

\[
/\!* \text{ requires } \text{guard_evt}(x);\]

\[
\text{assignable } \text{\{}\text{D } \text{\{}\text{ensures } A;\text{\}}
\text{also}\]

\[
/\!* \text{ requires } \text{guard_evt}(x);\]

\[
\text{assignable } \text{\{}\text{\}} \text{\{}\text{ensures } true;\text{\}}
\text{private void } \text{runEvt}(\text{Type } x) \text{\{}\text{\}}
\text{if } (\text{guard_evt}(x)) \text{\{} B \text{\}}
\]

\[
/\!* \text{ requires } \text{run() \{}\text{\}}\]

\[
\text{while } (\text{true}) \text{\{}\}
\text{... }
\text{Type } x = \text{GuardValue}\text{Type}.\text{next}();
\text{if } (\text{guard_evt}(x)) \text{\{}\}
\text{m.util.lock(\text{event});}
\text{run_evt}(x);
\text{m.util.unlock(\text{event});}
\text{\}}
\]

Translation of constants follows a similar pattern to the
translation of carrier sets, except that constants values are
rather constrained by axioms. The helper operator TypeOf
translates the type of an Event-B variable or constant to the
Java representation of that type. Value<Type>.next() returns a value of type Type
that satisfies the axioms defined in the context the machine
sees.

Machine variables are translated to class attributes. The
JML keyword spec_public makes a protected or private
attribute or method public to any JML specification.

Deterministic and non-deterministic assignments are
translated via the operators EB2Java and EB2Jml respectively. 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 \( v : v' = v + w \).

\[
\text{EB2Jml}(\text{constant } c) =
/\!* \text{ public constraint } c.\text{equals}(\text{old}(c));*
\]

\[
\text{TypeOf}(c) = \text{Type } v = \text{Value}\text{Type}.\text{next}(\text{Cons})
\]

\[
\text{EB2Java(\text{constant } c) =
\text{public static final Type } c = v;}
\]

\[
/\!* \text{ spec_public } +/\!* \text{ private Type } v;\]

\[
\text{EB2Java(\text{variables } v) =
/\!* \text{ spec_public } +/\!* \text{ private Type } v;\]
\]

\[
/\!* \text{ public invariant_redundantly } T;\]

We translate carrier sets and constants as class attributes,
and restrict those attributes for verification purposes. Hence,
carrier sets are translated as class attributes with the addition
of a history constraint that prevents any change in their
values. As we have no type information about carrier sets,
they are simply translated as sets of integers.
Multiple actions in the body of an event are translated individually and the results are conjoined. For example, a pair of simultaneous actions $x := y$ and $y := x$ is translated to the JML post-condition $x == \text{old}(y) \land y == \text{old}(x)$ for variables $x$ and $y$ of type integer. This translation correctly models simultaneous actions as required by the semantics of Event-B. In Java, simultaneous assignments are implemented by first calculating the value of each right hand side of the assignment into a temporary variable.

In the following, we present all the helper operators used in the translation. 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 unifying the variables assigned by all contained actions.

$$\text{Mod}(v := E) = \{v\} \quad \text{Mod}(v : |P|) = \{v\}$$

The $\text{Pred}$ operator translates boolean, relational and arithmetic expressions in the natural way. Since types and operations in Event-B do not have a direct counter part in Java, to supply these types, and operations, we implemented (and specified in JML) classes $\text{BSet}$ and $\text{BRelation}$. Several of the rules defining $\text{Pred}$ that translate applications of Event-B operators to calls of the corresponding methods of classes $\text{BSet}$ and $\text{BRelation}$ are presented below. In these rules, the $s_i$’s are sets and $r$ is a relation.

$$\frac{\text{Pred}(s_1) = s_1 \quad \text{Pred}(s_2) = s_2}{\text{Pred}(s_1 \subseteq s_2) = s_1.\text{isSubset}(s_2)} \quad \text{(Subset)}$$

$$\frac{\text{Pred}(x) = x \quad \text{Pred}(s) = s}{\text{Pred}(x : s) = s.\text{has}(x)} \quad \text{(Has)}$$

$$\frac{\text{Pred}(r) = r \quad \text{Pred}(s) = s}{\text{Pred}(r(s)) = r.\text{image}(s)} \quad \text{(Image)}$$

Particular types of Event-B relations (total relations, functions, etc.) are translated as $\text{BRelations}$ with appropriate restrictions in the invariant, e.g. $\text{Pred}(r \in D \rightarrow R)$ for sets $D$ and $R$ equals: $r.\text{isFunction}() \land r.\text{inverse}().\text{isFunction}() \land r.\text{domain}().\text{equals}(D) \land r.\text{range}().\text{isSubset}(R)$, which is added to the invariant. We further define classes $\text{Enumerated}$, $\text{ID}$, $\text{INT}$, $\text{NAT}$, $\text{NAT1}$ and $\text{BOOL}$. For example, $\text{Pred}(i \in \text{N}) = \text{NAT}.\text{instance}.\text{has}(i)$, which restricts $i$ to be non-negative. The $\text{TypeOf}$ operator translates the type of Event-B variables and constants to the corresponding Java type. All integral types are translated as type $\text{Integer}$, all relations and functions are translated as type $\text{BRelation}$, and all other sets are translated as type $\text{BSet}$.

4The $\text{Mod}$ operator is based on a similar operator defined within the Chase tool [8].

### 4.1 Support for Event-B Model Decomposition

When modeling systems with Event-B, one usually starts with the design of a single closed machine that includes both the modeling of the system and the surrounding environment. The machine is then refined into a more concrete model of the system. Abstract machines usually include a few events and a few variables whereas (advanced) refinements include quite a few of them so that defining additional refinements becomes heavy and discharging proof obligations in Rodin becomes intricate. Hence, one would like to understand which variables and events are actually involved in the refinement steps one is undertaking. Therefore, a machine decomposition mechanism is needed, and the question arises on whether code generation of decomposed machines is feasible. In [4], J.-R. Abrial and S. Hallerstede propose a technique for machine decomposition based on shared variables in which each decomposed machines simulates the behaviour of other decomposed machines through the use of external events. In [6] M. Butler proposes a technique for machine decomposition by shared events in which decomposed machines include copies of all the variables events use. The latter technique is implemented in Code Generation [15]. Both machine decomposition techniques produce independent machines that include local copies of shared variables or local events that simulate the effect of other decomposed machines acting on the shared variables. Therefore, the EventB2Java can produce Java implementations of decomposed machines straightforwardly.

### 4.2 Support for Code Customisation

The JML specifications produced by EventB2Java enable users to write bespoke implementations of the Java code produced by EventB2Java. Thus, the user may customise the produced implementation and then use an existing JML tool such as OpenJML [13] to check whether or not the customised implementation verifies against the JML specification produced by the EventB2Java tool. Of course, for this to work, the translation from Event-B to JML needs to be sound. A soundness proof of this translation is presented in [7]. The 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. The soundness condition just described is stated as a theorem and proved interactively in Rodin.

### 5. IMPLEMENTATION

EventB2Java is implemented as a plug-in of the Rodin platform [3] and is available at http://poporo.uma.pt/Projects/favas/EventB2Java.html. It uses the Rodin API to collect all components of the machine to be translated (i.e. carrier sets, constants, axioms, variables, invariants and events) as well as all the necessary information (such as the gluing invariant) from the refined machines. All this information is stored in the Rodin database and can be accessed using the $\text{org.eventb.core}$ library. Event-B expressions and statements are parsed and stored as abstract syntax trees, which can be accessed and traversed using the AST library in the $\text{org.eventb.core.ast}$ package. The Rodin API also provides a library to traverse trees (a walker) and to attach information to tree nodes. The bulk of the implementation of EventB2Java is realised through the
implementation of a utility class to fetch machine invariants, contexts and context information, theorems, axioms, abstract and concrete machine variables, and events in general. We extended the visitor design pattern provided by Rodin to traverse the abstract syntax trees and generate Java code with the respective JML specifications. We also implemented a class that stores the type of variables. We implemented Event-B sets and relations as the BSet and BRelation Java classes.

### Table 1: Statistics of the Event-B Models

<table>
<thead>
<tr>
<th>Event-B Model</th>
<th>LOC</th>
<th>Mchs</th>
<th>Evts</th>
</tr>
</thead>
<tbody>
<tr>
<td>Social-Event Planner [25]</td>
<td>1328</td>
<td>9</td>
<td>35</td>
</tr>
<tr>
<td>MIO [9]</td>
<td>586</td>
<td>7</td>
<td>21</td>
</tr>
<tr>
<td>Heating Controller [17]</td>
<td>458</td>
<td>15</td>
<td>32</td>
</tr>
<tr>
<td>State Machine [26]</td>
<td>86</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>Binary Search [1]</td>
<td>101</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Linear Search [1]</td>
<td>54</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Minimum Element [1]</td>
<td>64</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>Reversing Array [1]</td>
<td>64</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Sorting Array [1]</td>
<td>137</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Array Size</th>
<th>Binary Search</th>
<th>Linear Search</th>
<th>Min Element</th>
<th>Rev Array</th>
<th>Sort Array</th>
</tr>
</thead>
<tbody>
<tr>
<td>100,000</td>
<td>111</td>
<td>2859</td>
<td>29</td>
<td>262</td>
<td>25</td>
</tr>
<tr>
<td>200,000</td>
<td>203</td>
<td>3078</td>
<td>28</td>
<td>248</td>
<td>28</td>
</tr>
<tr>
<td>300,000</td>
<td>174</td>
<td>3227</td>
<td>30</td>
<td>228</td>
<td>35</td>
</tr>
<tr>
<td>400,000</td>
<td>135</td>
<td>3130</td>
<td>31</td>
<td>471</td>
<td>88</td>
</tr>
<tr>
<td>500,000</td>
<td>147</td>
<td>3276</td>
<td>28</td>
<td>487</td>
<td>271</td>
</tr>
</tbody>
</table>

Table 2: Execution times (milliseconds) of the code produced by EventB2Java.

6. EXPERIMENTAL RESULTS

The EventB2Java tool generates an Eclipse project that contains the Java code and the JML specifications produced by the tool. We ran EventB2Java on the examples in Table 1. As a validation step, we used the OpenJML tool [13] to check the syntax of the JML specifications generated by EventB2Java. This process uncovered some issues related to the translation of the type of Event-B variables and the translation of invariants to JML. We addressed the issues and corrected the tool and generated code again. In Table 1, “LOC” stands for Lines of Code in Event-B, “N Mchs” for the number of machines that the Event-B model is composed, and “N Evts” for the number of events the model has. For instance, Binary Search consists of 101 lines of Event-B code, it is composed of 3 machines, and contains 3 events. EventB2Java produced Java code and JML specifications for all the examples listed in Table 1. The code is available at http://poporo.uma.pt/Projects/favas/EventB2Java.html. The Social-Event Planner [25] is an Event-B model of a planner of social events in which a user can create a social event and invite a list of people to join it. This model extends the Social Networking model introduced in Section 2. MIO [9] is an Event-B model of a massive transportation system which includes articulated buses following the main corridor routes of a city. The Heating Controller [17] model provides an interface to adjust and display a target temperature, to sense and display the current temperature, and to power on/off the heater, among other functionality. State Machine [26] is an Event-B model of state machines. The rest of the examples are sequential program developments written by J.-R. Abrial in [1]. Linear and Binary Search are the Event-B models of the respective searching algorithms. Minimum Element is an Event-B model for finding the minimum element of an array of integers. Reversing Array, and Sorting Array are Event-B models for reversing and sorting an array.

Table 2 shows the times in milliseconds that the produced Java code by EventB2Java for several examples in Table 1 takes to finish its execution, where “Array Size” is the number of elements of the array to be processed. Since EventB2Java does not generate values for constants (e.g. arrays in models depicted in Table 1), we manually generated for constants. We ran the code produced by EventB2Java on a Mac OS X laptop with an Intel Core i5 2.3 GHz processor.

8. FUTURE WORK

EventB2Java translates axioms as JML invariants. Axioms shape the values constants. Hence, to generate code for a machine that includes constants and axioms one needs to give each constant some initial value that lives up to the axioms (see Rule Cons in Section 4). The process of generating values for constants is not automated in EventB2Java. As future work, we plan on implementing an automated system that suggests values for Java constants that are constrained by axioms. One could use the jmle tool [19, 11] to execute specifications. jmle translates JML specs to Java programs that are executed via the Java Constraint Kit (JCK). Providing support for non-deterministic assignments in EventB2Java is also future work. We are further interested in benchmarking EventB2Java against other Java code generators such as Code Generation [15] and EB2J [23]. Finally, the expression GuardValue<Type>.next() (see Rule Any in Section 4) is not completely implemented yet. Its imple-
mentation is future work too.

9. CONCLUSION

We have defined a translation that bridges the gap between Event-B models and a programming language. The translation was made through a series of translation rules that parse an Event-B model and produce Java code. Additionally, the translation produces JML specifications so the code can be customised. We have developed the EventB2Java tool that automates the process of the translation. The EventB2Java tool produces JML-specified Java code from an (either abstract or concrete) Event-B model, the tool largely support the Event-B syntax, and it produces executable Java code (either for sequential developments or for multi-threaded ones). We have tested the tool with several Event-B models. Having a tool that translates refinement calculi models into a programming language makes formal techniques and tools more usable, as software developers can use the formal methods, tools and software development methodologies they are most familiar with while keeping strong connections with other formal approaches.

10. REFERENCES