The EventB2Dafny Rodin Plug-In

Néstor Cataño  
The University of Madeira  
Funchal, Portugal  
ncatano@uma.pt

K. Rustan M. Leino  
Microsoft Research  
Redmond, WA, USA  
leino@microsoft.com

Víctor Rivera  
The University of Madeira  
Funchal, Portugal  
victor.rivera@m-iti.org

Abstract—This paper presents a translation of Rodin proof-obligations into the input language of Dafny, and the implementation of the translation as the EventB2Dafny Rodin plug-in. Rodin is a platform that provides support for Event-B. The paper uses a simplified Event-B model for social-networking to illustrate the translation and to describe the generated Dafny model. EventB2Dafny supports the full Event-B syntax and its full source code is available online.

Keywords—Boogie; Dafny; Event-B; Proof Obligations; Refinement Calculus; Rodin

I. INTRODUCTION

In the refinement calculus strategy for software development using Event-B [1], systems are first modelled from a very abstract point of view, which comprises fundamental observations, and then details are added to the system to describe its full behaviour. Refinement steps generate proof obligations that ensure that the system works correctly and refined models live up to more abstract models.

On the other hand, formal program notations such as Eiffel [2], JML [3], [4], Spec# [5] or Dafny [6], provide support for the idea of design-by-contract [7], in which the construction of software systems is regarded as the implementation of contractual relations between software modules. Dafny is one of the front-end specification languages of the Boogie2 tool [8], [9]. The Dafny verifier translates Dafny programs into the intermediate verification language Boogie, and the Boogie tool then generates verification conditions for the SMT solver Z3 [10] (among other theorem provers). The Dafny language is class based, and thus has features akin to those found in languages like C#.

This paper presents a translation of Event-B proof-obligations to Dafny and its implementation as the EventB2Dafny Rodin plug-in. That is, although Dafny is a programming language, the target Dafny program of this translation is not meant to be executed. Rather, since Dafny has an automatic verifier, the translation from Event-B produces Dafny code snippets that are correct if and only if the Event-B refinement-based proof obligations hold. The Rodin platform [11] is an Eclipse IDE for Event-B that provides support for program refinement and mathematical proof. EventB2Dafny supports the full Event-B syntax. As far as Rodin is concerned, EventB2Dafny and its use of Dafny looks like just another theorem prover back-end. Full source code for EventB2Dafny and instructions are available in [12].

The proposed translation of Rodin proof-obligations into the input language of Dafny and the use of the EventB2Dafny plug-in tool are useful for a number of reasons, both for the Event-B refinement calculus community and for the users of Dafny and Boogie. EventB2Dafny brings refinement calculus analysis into Boogie. It further allows people working on refinement calculus based approaches to have access to an ample set of formal techniques (e.g. runtime checking and static analysis) and tools (Boogie’s analysers and verifiers), providing quick feedback about the correctness of a modelled Event-B program.

We begin this paper by introducing the Event-B approach to software development (Section II) and the Dafny language (Section III). We then describe the type of proof obligations generated by the Rodin platform (Section IV), and present our translation of Rodin proof-obligations to Dafny programs (Section V). We finally present the implementation of the EventB2Dafny Rodin plug-in (Section VI), mention some related work (Section VII) and conclude and discuss future work (Section VIII). We use a simplified Event-B model for social networking from [13] to illustrate our approach to the translation of Event-B proof-obligations to Dafny. The Event-B model introduces network content, people, and operations for editing, adding, removing, and publishing content. It further models network privacy policies with the aid of access permissions.

II. REFINEMENT CALCULUS AND THE B METHOD

In the B method for software development, systems go through a series of stages. Each successive stage is a description of the system with a higher level of detail. A successive model is said to refine a model in a previous stage. A refined model must keep a palpable behavioural relation with its abstraction. This relation is modelled through a “gluing invariant” property. Event-B models (called machines) are composed of a static part (a state): relations, functions, and invariants; and a dynamic part: events, which describe how...
the system evolves. An event is composed of a guard (a conjunction of predicates) and actions that may be triggered when the guard holds. Refinement can be achieved by adding new variables to the system (new observations), or new actions to an event, or by strengthening the guard of an event. Refinement steps generate proof obligations so as to ensure that the refinement machine does not invalidate the refined machine. Events must maintain the machine invariants. The Event-B language for stating properties (essentially predicate logic plus set theory) and the language for specifying the dynamic behaviour of machines are seamlessly integrated.

Figure 1 presents a simplified Event-B model for social networking from [13]. Machine SocialNetwork sees a context that declares two sets, PERSON and RAWCONTENT, representing the set of all possible persons and the set of all possible content (text, video, photographs, etc.) in a social network, respectively. Variables person and rawcontent are the sets of all persons and content that are actually in the network, content is a relation mapping page-content to people, and owner is a total function that maps content to their owner.

A common operation in social networking websites is publishing content to people in the network. The event transmit is used to publish the raw-content rc from the page of the owner of rc to the page of pe. In the definition of transmit, $rc \rightarrow pe$ represents the pair of elements $(rc, pe)$. The construct any models unbounded choice substitution: it gives the implementer of the event the opportunity to choose any values for the bound variables rc and pe that satisfy the guards grd1, grd2, and grd3.

```plaintext
machine SocialNetwork sees ctx
variables
    person, rawcontent, content, owner
invariants
    inv1 person \subseteq \text{PERSON}
    inv2 rawcontent \subseteq \text{RAWCONTENT}
    inv3 content \in\text{rawcontent} \rightarrow \text{person}
    inv4 owner \in\text{rawcontent} \rightarrow \text{person}

events
transmit
    any rc pe
    where
        grd1 rc \in\text{rawcontent}
        grd2 pe \in\text{person}
        grd3 rc \rightarrow pe \notin content
    then
        act1 content := content \cup \{rc \rightarrow pe\}
    end
end
```

Figure 1. An Event-B machine for social networking

### III. Dafny

Dafny is a programming language that provides support for the annotation of program contracts as method pre- and post-conditions. The method `Init` below creates and returns an object of type `Cell`. The pre-condition of the method (the requires clause) states that it must be called with a non-null object, and its post-condition (the ensures clause) expresses that the new object is non-null and has a data value that is equal to the one of the parameter of the method.

```plaintext
function Init(x: Cell): Cell
    requires x != null;
    ensures Init(x) != null \&\&
        Init(x).data == x.data;
```

Dafny provides support for abstract specifications through the definition and use of ghost variables and methods. A ghost var $x$ variable declaration introduces a specification-only variable $x$, which is interpreted by Dafny as a concrete variable, except that it is not allowed to be written into a concrete variable. A ghost method declaration tells Dafny that the method will be used in the verification of the program but not at run-time, so that the compiler does not need to allocate space for creating the method.

Dafny provides support for the definition and specification of mathematical functions (see `isPositive` below). Functions are specification-only constructs; they exist for verification-only purposes and are ghost by default. The requires specification of the function may be used to define its domain (partial functions). The reads part declares the function’s frame condition, and the decreases part states the termination metrics of the function.

```plaintext
function isPositive(s: Cell): bool
    requires P; reads R; decreases D;
```

Dafny provides support for generic types. For instance, one can declare a class `Wallet<T>` that uses a generic type $T$. In the style of ML, Dafny also provides support for datatypes that use constructors and accessors. Dafny provides support for the use of universal and existential quantification of predicates. The reader is invited to consult [6] for a complete presentation of Dafny.

### IV. Rodin Proof Obligations

The Rodin proof-obligation generator automatically generates proof obligations based on the machine and context texts. Given the abstract event `evt0` and the concrete event `evt` below, and given an abstract and a concrete machine declaring the events respectively, Rodin generates several proof obligations to ensure that the machines are models of the same system yet at a different level of abstraction. Variables $s$ and $c$ are the sets and constants “seen” by the abstract and concrete machines, $v$ is the set of abstract variables, $w$, and
which includes \( v \), is the set of concrete variables, predicates \( G \) and \( H \) are the abstract and concrete guards, \( BA_0 \) and \( BA \) are before-after predicates that relate the state of variables before and after actions occur\(^1\).

The symbol \( \| \) represents non-deterministic assignment. Non-deterministic assignments generalise deterministic assignments (formed with the aid of \( := \)), e.g. the deterministic assignment \( x := x + z \) can be expressed as the non-deterministic assignment \( x :\| x' = x + z \).

The abstract machine declares (not shown here) an abstract invariant \( I \). The concrete machine declares an invariant \( J \) (also called “gluing” invariant) that depend on the context and the local machine variables respectively. Contexts can further declare a set of theorems and axioms.

\[
\begin{align*}
\text{evt}_0 & \quad \text{any } x \text{ where } G(s, c, v, x) \\
& \quad \text{then} \\
& \quad \text{act } v :\| BA_0(s, c, v, x, x') \\
& \quad \text{end} \\
\text{evt} & \text{ refines } \text{evt}_0 \\
\text{any } y \text{ where } H(y, c, w) & \quad \text{then} \\
& \quad \text{act } w :\| BA(s, c, w, y, w') \\
& \quad \text{end}
\end{align*}
\]

Rodin generates invariant preservation proof obligations (INV) for every abstract (concrete) event expressing that, given the axioms, the abstract (gluing) invariant, the guard of the event, and the before-after predicate, then the abstract (the concrete) invariant holds in the after state. Rodin generates a guard strengthening proof obligation (GRD) for every event expressing that the guard of the concrete event must be as least as strong as the guard of the abstract event. It generates a feasibility proof-obligation (FIS) for the action of every event stating that a solution to the before-after predicates exists. For every event, Rodin generates a simulation proof obligation (SIM) that ensures that abstract actions are correctly simulated by the concrete actions. That is, the result produced by the abstract action does not contradict the result produced by the abstract action. Rodin also generates numeric and finite proof obligations to ensure that variable declarations are well-defined (WD) and sets are finite (FIN).

In Rodin, one may declare a machine variant positive integer expression \( n \) that must be decreased by all new machine events to ensure that they are not indefinitely enabled (NAT). Machine events can be declared anticipated (convergent), and Rodin generates proof obligations expressing that the modified variant (evaluated after executing the event action) should be (strictly) less than or equal to the variant evaluated before the event action occurs (VAR).

One can declare a witness of a refinement event with the aid of the with clause of Rodin. A witness expression relates bounded variables of an abstract event with bounded variables of the concrete event, or abstract machine variables with concrete machine variables, e.g. we could have added the expression \( \text{with } x : x' = y \) to the declaration of the concrete event \( \text{evt} \) above, meaning that variable \( x \) in the abstract event is renamed \( y \) in the concrete event. Rodin generates a witness proof obligation (WFIS) for every event witness expressing that a solution for the witness expression exists.

Theorems must be provable from contexts or machines (THM). In Event-B, Theorems are used to simplify complex proof-obligations.

\[5. \text{ Rodin proof obligations in Dafny}\]

The Event-B language for expressing properties is based on predicate logic and set theory. We formalise sets, relations, and other Event-B structures as datatypes in Dafny, and operations over these structures as functions. A relation \( \text{Relation}<D,R> \) between a set of type \( D \) and a set of type \( R \) is a constructed type in Dafny, formalised with the aid of a Rel type constructor, which has three parameters, domain of type \( D \), range of type \( R \), and a map between the domain and the range, formalised as a set of pairs. For the Rel type constructor, in the style of other languages like Objective Caml or the PVS language [14], Dafny implicitly declares a \( \text{Rel}? \) predicate that returns \text{true} of any constructed element formed with the type constructor \( \text{Rel} \).

Function \( \text{image} \) below models the image of a relation \( r \) applied to an element \( a \) of its domain. Definition of function \( \text{image} \) is axiomatic. Instead of calculating the result of the function applied to an element of its domain, we wrote a function post-condition (the \( \text{ensures} \) clause) that defines the function. Hence, for all objects \( p \) of type \( \text{Pair}<D,R> \), \( p \) is an element of \( r \) and the first entry \( p.x \) of the pair \( p \) is equal to \( a \), if and only if the second entry \( p.y \) of \( p \) is part of the image of \( r \) applied to \( a \). We provided axiomatic definitions for all functions over sets and relations in Dafny.

Modelling relations and sets as datatypes rather than as classes has two main advantages in Dafny. First, instances of classes require new allocations, and second, their fields would need method frame declarations (the \( \text{modifies} \) clause of Dafny), which can degrade the performance of Dafny/Boogie/Z3. However, note that an Event-B relation can be used anywhere that a set can appear (a relation is a set of pairs), but unfortunately datatypes cannot be inherited. Therefore, our translation from Event-B to Dafny makes sure that operations are called on the right datatype.

\(^1\)Primed variables refer to after-states.

\(^2\)Dafny infers the type of \( p \) from its usage in the post-condition.
datatype Pair<S,T> = Pr(x: S, y: T);

datatype Relation<D,R> =
  Rel(domain: set<D>, range: set<R>,
       map: set<Pair<D,R>>);

static function image<D,R>(r: Relation<D,R>, a: D)
  : set<R>
ensures forall p :: p
     in r.map && (p.x == a)
     ==> (p.y in image(r,a));

We present below an Event-B machine context and its translation to Dafny. The context introduces the permissions view and edit. As we do not have information about carrier sets, e.g. PERMISSION below, we model them as set of integers. As Dafny does not include constants or axioms, we model constants as 0-ary integer functions and axioms as boolean functions with a post-condition that introduces the axiom. Constants are assumed in the translation of a proof obligation. Theorems are translated similar to axioms, but they are checked (the clause assert) instead.

<table>
<thead>
<tr>
<th>constants</th>
<th>view edit</th>
</tr>
</thead>
<tbody>
<tr>
<td>sets PERMISSION</td>
<td></td>
</tr>
<tr>
<td>axioms</td>
<td></td>
</tr>
<tr>
<td>axiom1 view ∈ PERMISSION</td>
<td></td>
</tr>
<tr>
<td>axiom2 edit ∈ PERMISSION</td>
<td></td>
</tr>
<tr>
<td>axiom3 view ≠ edit</td>
<td></td>
</tr>
</tbody>
</table>

var PERMISSION: Set<Integer>;

function view(): Integer
function edit(): Integer

function axiom1(): bool
  ensures PERMISSION.has(view());

function axiom2(): bool
  ensures PERMISSION.has(edit());

function axiom3(): bool
  ensures view().notEquals(edit());

Invariants (see below) are modelled as boolean functions with postconditions that can be assumed or asserted. Event-B variables, constants, and carrier sets are implicitly non-null. This non-nullness condition has to be made explicit in Dafny and assumed by all machine invariants.

<table>
<thead>
<tr>
<th>invariant</th>
</tr>
</thead>
<tbody>
<tr>
<td>inv person ⊆ PERSON</td>
</tr>
</tbody>
</table>

function nonNullnessCond(): bool
  { person != null && PERMISSION != null }

function inv1(): bool
  requires nonNullnessCond();
  ensures person.subSet(PERSON);

Depending on the type of proof obligation generated by Rodin, e.g. INV, GRD, FIS, SIM, etc., a ghost method is declared that might or might not assume local invariants, theorems or axioms.

We present below an event refinement of the transmit event in Figure 1 that includes an additional action actr1 that extends the “visible” content of the social network to contain the pair rc → pe. This action generates an INV proof-obligation transmit_invr1_INV for the invariant invr1 of the refinement machine (among other proof obligations). The ghost method assumes the invariants of the abstract and concrete machines, the non-nullness axiom, and the guards of the refined event. The method finally asserts the result of the translation of the action of the refinement event.

<table>
<thead>
<tr>
<th>variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>visible</td>
</tr>
<tr>
<td>invariants</td>
</tr>
<tr>
<td>invr1 visible ∈ rawcontent ⇔ person</td>
</tr>
<tr>
<td>event transmit extends transmit</td>
</tr>
<tr>
<td>then</td>
</tr>
<tr>
<td>actr1 visible := visible ∪ {rc ⇔ pe}</td>
</tr>
<tr>
<td>end</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ghost method</th>
</tr>
</thead>
<tbody>
<tr>
<td>transmit_invr1_INV()</td>
</tr>
<tr>
<td>{ // assume non-nullness condition, invariants, and axioms</td>
</tr>
<tr>
<td>assume rawcontent.has(rc);</td>
</tr>
<tr>
<td>assume person.has(pe);</td>
</tr>
<tr>
<td>assume Pair(rc,pe).notIn(content);</td>
</tr>
<tr>
<td>assert visible.union(Set({Pair(rc,pe)}))</td>
</tr>
<tr>
<td>.isRelation() &amp;&amp;</td>
</tr>
<tr>
<td>visible.union(Set({Pair(rc,pe)}))</td>
</tr>
<tr>
<td>.domain.equals(rawcontent) &amp;&amp;</td>
</tr>
<tr>
<td>visible.union(Set({Pair(rc,pe)}))</td>
</tr>
<tr>
<td>.range.equals(person);</td>
</tr>
<tr>
<td>}</td>
</tr>
</tbody>
</table>

Rodin well-definedness (WD) proof-obligations are generated to ensure the correct definition and usage of variables, functions, guards, actions, variants, and witness expressions. If the refinement transmit event above is extended with the action actr2 below that gives user pe view-permission on the page content of the owner of the raw-content rc, then Rodin generates a WD proof obligation to ensure that rc belongs to the domain of function owner (see inv4 in Figure 1). The proof obligation is modelled in Dafny as the transmit_actr2_WD() method below that includes an assert expression doing the verification of the WD condition.

| actr2 viewpermission := viewpermission ∪ |
| {owner(rc) ⇒ pe} |

3 viewpermission is a new variable of the refinement machine.
The proposed translation. The proof obligations have been from [13], used in this paper to illustrate concepts of and refinement machines for the social-networking model by using it to translate the proof obligations of the abstract
anism backing the Rodin platform.

The direct lies on the soundness of the proof-generation mech-
as generated by Rodin, so the soundness of EventB2Dafny
machine. EventB2Dafny directly works on proof obligations
with machine variables and invariants, which might be conjoined
information about a machine context, e.g. sets and axioms, or
into a Dafny program. The proof obligation can include
obligation syntax and generates Dafny code.

The EventB2Dafny tool parses Rodin proof obligations
into a Dafny program. The proof obligation can include
information about a machine context, e.g. sets and axioms, or
machine variables and invariants, which might be conjoined
with machine variables and invariants from a refinement
machine. EventB2Dafny directly works on proof obligations
as generated by Rodin, so the soundness of EventB2Dafny
directly lies on the soundness of the proof-generation mech-
anism backing the Rodin platform.

We have validated our implementation of EventB2Dafny
by using it to translate the proof obligations of the abstract
and refinement machines for the social-networking model
from [13], used in this paper to illustrate concepts of the proposed translation. The proof obligations have been
discharged with Boogie.

VII. RELATED WORK

In a previous work [15], the first and third authors define
a translation from classical B to JML (Java Modelling
Language), implemented in the ABTools suite [16]. We are
currently working on the implementation of the EventB2Jml
Rodin plug-in that translates Event-B machines to JML
specifications. EventB2Dafny and EventB2Jml share most
libraries to manipulate Event-B machines, contexts and
events. We plan to integrate the two plug-ins in a single
proof environment.

In [17], David Déharbe presents an approach to translate
Rodin platform proof-obligations to the input language of the
SMT-solvers. The approach handles proof obligations that
include boolean expressions, integer arithmetic expressions,
and basic sets and relations. The EventB2Dafny plug-in
works in a similar direction. Yet, by generating bespoke
Dafny/Boogie proof obligations, we can improve the perfor-
ance of the Z3 SMT solver [10] on which Boogie works.

In [18], David Mentré, Claude Marché, Jean-Christophe
Filliâtre and Masashi Asuka present the bpo2why tool that
translates proof-obligations generated by the Atelier B suite
into Why [19] programs. Therefore, proof-obligations can be
discharged using the Krakatoa tool [20] or other automatic
provers like Z3 [10]. The EventB2Dafny plug-in works in
a similar direction as the bpo2why tool, but our target
language is Dafny rather than Why.

VIII. CONCLUSION

Discharging Event-B proof obligations for complex mod-
els is labour intensive and users of refinement calculus
tools like Rodin need to learn predicate calculus, whose
mathematical aspects are not always intuitive for mainstream
programmers. The EventB2Dafny Rodin plug-in gives Rodin
expert users the alternative of using a proof-engine that
is backed by the power of the Z3 SMT solver. In Rodin,
the EventB2Dafny plug-in acts as another theorem prover.
EventB2Dafny also gives non-expert Rodin users the alterna-
tive of using a programming language with a syntax similar
to mainstream languages, e.g. .NET or Java to manipulate
and understand proof obligations.

Currently, proof obligations generated by EventB2Dafny
are manually fed into Dafny. We are planning to integrate
EventB2Dafny to Dafny and Microsoft Visual Studio. We
also want to investigate and to characterise the type of proof-
obligations for which Boogie outperforms existing Rodin
proof-engines.

Rodin provides a lasso functionality whereby users can
select or deselect hypotheses of a proof obligation having
common variables with the variables of the goal. We plan
to implement a similar functionality for EventB2Dafny.

We are currently working on the implementation of
the EventB2Jml tool that translates Event-B models into
JML specifications. We plan to integrate Event2Jml and
EventB2Dafny as part of the same formal methods envi-
ronment.

Besides translating Event-B components and proof obli-
gations, we also plan to generate frame conditions. To assist
in the generation of frame conditions, we plan to define an
operator Mod to calculate the set of variables modified by

ghost method transmit_actr2_WD()
{
    // assume non-nullness condition and machine invariants
    assert owner.domain.has(rc) &&
        owner.isFunction() &&
        owner.domain.equals(rawcontent) &&
        owner.range.equals(person);
}

VI. IMPLEMENTATION

that provides support for developing systems by program
refinement and includes functionality for generating and
manipulating proof obligations. The EventB2Dafny tool has
been integrated to Rodin as an Eclipse plug-in. Full source
code for EventB2Dafny and instructions for installing and
using it are available in [12].

Rodin provides an API for the data model and the
 persistence layers that link plug-ins with Event-B compo-
 nents. The data model is composed of a series of Java
 interfaces to manipulate the parts of Event-B components.
The persistence layer (called the Rodin database) uses XML
 files to store Event-B components. Rodin uses abstract
 syntax trees (AST) that store Event-B machines’ mathe-
 matical declarations for variables, predicates, sets, relations,
 functions, events, etc. It also provides a library to traverse
trees (a walker) and to attach information to tree nodes.
Rodin also provides functions for accessing and manipu-
lating proof obligations. The bulk of the implementation
of EventB2Dafny is realised through the implementation of
a utility machine to fetch machine invariants, contexts and
context information, theorems, axioms, abstract and concrete
machine variables, and events in general. We extended
an Event-B tree walker. The walker traverses the proof-
 obligation syntax and generates Dafny code.

The EventB2Dafny tool parses Rodin proof obligations
into a Dafny program. The proof obligation can include
information about a machine context, e.g. sets and axioms, or
machine variables and invariants, which might be conjoined
with machine variables and invariants from a refinement
machine. EventB2Dafny directly works on proof obligations
as generated by Rodin, so the soundness of EventB2Dafny
directly lies on the soundness of the proof-generation mech-
anism backing the Rodin platform.

We have validated our implementation of EventB2Dafny
by using it to translate the proof obligations of the abstract
and refinement machines for the social-networking model
from [13], used in this paper to illustrate concepts of the proposed translation. The proof obligations have been
discharged with Boogie.

VII. RELATED WORK

In a previous work [15], the first and third authors define
a translation from classical B to JML (Java Modelling
Language), implemented in the ABTools suite [16]. We are
currently working on the implementation of the EventB2Jml

bpo2why tool that
the action of an event. The definition of Mod will be based on the syntactic rules backing the analysis performed by the Chase tool in [21]. Hence, the only variables modified by an action are those variables modified by assignments within the action.

EventB2Dafny has been successfully applied to a social networking model written in Event-B, but it still needs to be tested on a wider variety of Event-B models. We plan to undertake a case study on comparing the efficiency of the Boogie tool against the standard Event-B provers to understand the kind of proof obligations for which the former outperforms the latter.

REFERENCES


