Electronic Communications of the EASST
Volume XXX (2015)

Proceedings of the
15th International Workshop on
Automated Verification of Critical Systems (AVoCS 2015)

# Transforming Event-B Models to Dafny Contracts

Mohammadsadegh Dalvandi, Michael Butler, Abdolbaghi Rezazadeh

15 pages

Guest Editors: Gudmund Grov, Andrew Ireland
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

# Transforming Event-B Models to Dafny Contracts

**Mohammadsadegh Dalvandi**[1], **Michael Butler**[2], **Abdolbaghi Rezazadeh**[3]

School of Electronic & Computer Science, University of Southampton

[1 2 3] md5g11,mjb,ra3@ecs.soton.ac.uk

**Abstract:** Our work aims to build a bridge between constructive (top-down) and analytical (bottom-up) approaches to software verification. This paper presents a tool-supported method for linking two existing verification methods: Event-B (constructive) and Dafny (analytical). This method combines Event-B abstraction and refinement with the code-level verification features of Dafny. The link transforms Event-B models to Dafny contracts by providing a framework in which Event-B models can be implemented correctly. The paper presents a method for transformation of Event-B models of abstract data types to Dafny contracts. Also a prototype tool implementing the transformation method is outlined. The paper also defines and proves a formal link between property verification in Event-B and Dafny. Our approach is illustrated with a small case study.

**Keywords:** Formal Methods, Hoare Logic, Program Verification, Event-B, Dafny

## 1 Introduction

Various formal methods communities [CW96, HMLS09, LAB+06] have suggested that no single formal method can cover all aspects of a verification problem therefore engineering bridges between complementary verification tools to enable their effective interoperability may increase the verification capabilities of verification tools. We distinguish two major approaches to software correctness based on their target phases in the development cycle: the *constructive approach* and the *analytical approach*. The constructive approach focuses on the early stages of the development and aims at formal modelling of the intended behaviour and structure of a system in different levels of abstraction and verifying properties of models. The analytical approach focuses on the code level and its target is to verify properties of the final program code. A wide range of verification tools exist to support both the approaches provided by formal methods communities worldwide. A high level look at these two approaches suggests that the constructive and analytical approaches should complement each other well. Nevertheless, our understanding and experience of how these approaches can be combined at a large scale is very limited. This represents a wasted opportunity, as these approaches are not benefiting from each other effectively.

We have chosen Event-B [Abr10] and Dafny [Lei10] as examples of constructive and analytical approaches respectively. Event-B is a formal approach for modelling and verifying software systems. An Event-B model is built through a number of successive refinement steps starting from an abstract representation of the system and proceed towards a concrete level. Event-B is supported by an open platform called Rodin [ABH+10]. Dafny is a programming language and verifier. Given a program code and its formal specification, the Dafny tool [LW14] (which is an

SMT-based verifier) can verify the program against its contract. In Dafny, contracts are annotations within the code. Event-B in its initial form does not have any support for the final phase of the development (implementation phase). On the other hand, Dafny has very little support for abstraction and refinement. In this work we present a tool-supported development method by linking two verification technologies: Rodin and Dafny. Our combined methodology is beneficial for both Event-B and Dafny users. It makes the abstraction and refinement of Event-B available for generating Dafny specifications which are correct with respect to a higher level of abstract specification in Event-B and provides a framework in which Event-B models can be implemented and verified in a programming language.

We provide a method for transforming Event-B models to Dafny code contracts (method pre- and post-conditions). In this paper we use the terms code contracts and annotations interchangeably. Transformation of Event-B formal models to annotated Dafny method declarations is achieved by defining a set of transformation rules. Using this set of transformation rules, one can generate code contracts from Event-B models but not implementations. The generated code contracts must be seen as an interface that can be implemented. The implementation can be verified later against the generated annotations using an automatic verifier to prove the correctness of the implementation with regards to the high level Event-B specification. We also developed a tool that is based on these translation rules for generating Dafny code contracts from Event-B models. The transformation rules are validated by being applied to a number of case studies including a map, a queue, and a stack abstract data types. This paper extends our previous short paper [DBR14] by providing full details of our transformation method and its proof of correctness.

The organisation of the rest of the paper is as follows: in Section 2, background information on Event-B and Dafny is given. Section 3 and Section 4 explain the methodology, transformation rules, and the formal basis of our work. A small example of transformation of Event-B models to Dafny contracts is presented in Section 5. Section 6 provides an overview to the tool support for our method. In Section 7 related and future works are discussed and finally Section 8 contains the conclusions.

## 2 Background

### 2.1 Event-B

Event-B is a formal modelling language for system level modelling based on set theory and predicate logic for specifying, modelling and reasoning about systems, introduced by Abrial [Abr10]. Modelling in Event-B is facilitated by an extensible platform called Rodin [ABH+10]. A model in Event-B consists of two main parts: *contexts* and *machines*. The static part(types and constants) of a model is placed in a context and is specified using carrier sets, constants and axioms. The dynamic part (variables and events) is specified in a machine by means of variables, invariants and events. An *event* models the state change in the system. Each event may have a number of assignments called actions. Each event may also have a number of guards. Guards are predicates that describe the necessary conditions which should be true before an event can occur. An event may have a number of parameters. Event parameters are considered to be local to the event. Figure 1 illustrates machine *m0* with two events *Add* and *Remove*. KEYS and VALUES are carrier sets in context *c0* which is not shown here.

**Machine** *m0* **Sees** *c0*
**Variables** map
**Invariants** map $\in$KEYS $\rightarrowtail\!\!\!\!\!\mapsto$VALUES
**Initialisation** map $:=\emptyset$

**Event** Add
**any** *k,v*
**where**
*grd1*: k $\in$KEYS
*grd2*: v $\in$VALUES
**then**
*act1*: map(k) $:=$ v

**Event** Remove
**any** *k*
**where**
*grd1*: k $\in$dom(map)
**then**
*act1*: map $:=$ {k}$\lhd\!\!\!-$map

Figure 1: Machine *m0*: the Most Abstract Level of Map ADT Model

Modelling a complex system in Event-B can largely benefit from refinement. Refinement is a stepwise process of building a large system starting from an abstract level and proceeds towards a more concrete level by a series of successive steps in which new details of functionality are added to the model in each step [But13]. The abstract level represents key features and the main purpose of the system. It is essential to prove the correctness of refinement steps in Event-B. Refinement of a model may consist of refining existing events, adding new events, and adding new variables and invariants. The new events must not diverge. This means that they should not run for ever. Each refinement may involve introducing new variables to the model. This usually results in extending abstract events or adding new events to the model. It is also possible to replace abstract variables by newly defined concrete variables. Concrete variables are connected to abstract variables through *gluing invariants*. A gluing invariant associates the state of the concrete machine with that of its abstraction. All invariants of a concrete model including gluing invariants should be preserved by all events. All abstract events may be refined by one or more concrete events.

The built-in mathematical language of the Rodin platform is limited to basic types and constructs like integers, boolean, relations and so on. The *Theory Plug-in* [BM13] has been developed to make the core language extension possible. A theory, which is a new kind of Event-B component, can be defined independently from a particular model and it is the means by which the mathematical language and mechanical provers may be extended.

## 2.2 Dafny

Dafny [Lei10] is an imperative sequential programming language. A program in Dafny usually contains two parts, namely implementation and specification. Dafny supports generic classes with some basic features such as method definitions, dynamic allocation, and inductive types for implementation. A method in Dafny is a piece of imperative, executable code. The verification power of Dafny originates from its specification constructs. A program behaviour can be specified in Dafny using constructs such as methods' pre- and post-conditions, framing constructs and termination metrics. The specification language also offers updatable ghost variables, recursive functions, sets, sequences and some other features. The Dafny verifier which is based on an SMT-solver called Z3 [DB08] uses specification to verify the implementation.

As mentioned above Dafny specification supports *functions*. A function has a very similar concept to mathematical functions and cannot write to memory and is defined by an expression. Functions are required to have only one unnamed return value. A special form of function which returns a boolean value is called *predicate*. Dafny uses the `ensures` keyword for post-condition declarations. A post-condition is always a boolean expression. Each method can have more than one post-condition which can be either joined with boolean *and* (&&) operator or defined separately using the `ensures` keyword. To declare a pre-condition the `requires` keyword is used. Like post-conditions, multiple pre-conditions are allowed in the same style. Pre- and post-conditions are placed after method declarations and before method bodies. Dafny does not have any specific construct for specifying *class invariants*. A work around is to place all class-level invariants in a predicate and then include this predicate in the pre- and post-conditions of all the class methods. In this we enforce the verifier to check if each method preserves all invariants.

## 3 Transforming Event-B Machines to Dafny Classes

Event-B supports a richer mathematical language than Dafny. For this reason, before an Event-B model is transformed to Dafny contracts, it must be refined to a level where the data types and operators have a Dafny counterpart. For example, relations do not have any Dafny counterpart so they are refined to a more concrete data structure (e.g. sequences) before transformation takes place. This is essential for reducing the syntactic gap between Event-B and Dafny. When the aforementioned point in the refinement process is reached then a machine and its elements (e.g. variables, invariants,...) are translated to a Dafny class. The translation of variables, generic types, and invariants is almost one-to-one. Transformation of events to method contracts is discussed in the next section. Generic types in an Event-B model, as mentioned earlier, are defined using carrier sets in a context. In Dafny, generics are declared in angle brackets after the name of a class. The following example shows how generic types are defined in Dafny:

`class` *class_name* <$T_1$, $T_2$,...,$T_n$>{ *...class body...*}

When a machine is translated to a Dafny class, all carrier sets which are defined in the context that is seen by that machine are translated as Dafny generics. Note that it is assumed at the moment that the context of the model only contains carrier sets. In an Event-B model variables are declared in the variables part of a machine and their types are specified using *typing invariants* which are defined separately in the invariants part of the machine. All machine's variables are translated as class variables in Dafny. Event-B invariants can be categorised as follows:

- **Typing invariants** that declare the types of a variable.
- **Model invariants** that express the properties of a model.
- **Gluing invariants** that relate the concrete variables to abstract variables.

As explained earlier, typing invariants are used for variable declarations. Preservation of typing invariants are checked implicitly by the Dafny type system. Gluing invariants are not translated to Dafny because at the moment we assume that the machine that is being translated is a data refinement of the abstract machine and none of the abstract variables are present in the refined machine. Preservation of gluing invariants must be proved in Event-B. Only model invariants are translated to Dafny. The conjunction of all model invariants are placed in a Dafny predicate called *Invariants()*. It is explained later how the predicate *Invariants()* is used in pre-condition generation.

## 4 Transforming Events to Annotated Method Declarations

In the previous section we discussed how the declaration elements of an Event-B machine should be translated to Dafny class members. In this section we present the way in which machine events are transformed to annotated Dafny method declarations.

### 4.1 Constructor Statement

Machine events are translated to Dafny methods. In Event-B, each event has three main parts: *parameters*, *guards*, and *actions*. Parameters may have different implicit roles (e.g. input or output) in an event. During the translation process, the role of each parameter must be made explicit by the modeller. There might be cases where the modeller decides to merge several events to form a single Dafny method. This is because the Event-B style is to represent different cases of some conceptual operation as separate events. Events that are going to be merged together and the target Dafny method should also be made explicit by the modeller. To cater for this we have have extended the underlying representation of Event-B machines with a new element called *constructor statement*. Constructor statements are used to make the parameters' roles and the merging events explicit. A constructor statement has the following form:

$$\texttt{method}\; \mathit{mtd\_name}(\mathit{in}_1, \mathit{in}_2,...)\; \texttt{returns}\; (\mathit{out}_1, \mathit{out}_2,...)\; \{\mathit{Evt}_1, \mathit{Evt}_2,...\}$$

Each constructor statement has four parts: name of the target method (*mtd_name*), a comma separated list of the method's input arguments ($in_1, in_2,...$), a comma separated list of the method's output arguments ($out_1, out_2,...$) and a comma separated list of events placed between braces ($Evt_1, Evt_2,...$). A constructor statement may or may not have input and/or output arguments.

### 4.2 Method Contract Generation

As mentioned before, to transform a group of events to a single method contract, constructor statements are used. Because Dafny uses Hoare logic [Hoa69] as the basis for verification, each defined constructor statement gives rise to generation of a Hoare triple. Pre- and post-conditions of each Hoare-triple are generated from listed Event-B events in the constructor statement. Each generated Hoare triple then is translated to Dafny method contracts. Any implementation that satisfies the generated contracts would be considered as a correct implementation of the Event-B model.

Assume there is a model with $n$ events, a set of variables $v$, invariants $I(v)$ and a constructor statement as follows:

$$\mathrm{Evt}_1 \triangleq \textbf{any}\ \mathrm{x}_1\ \textbf{where}\ \mathrm{P}_1(\mathrm{x}_1,\mathrm{v})\ \textbf{then}\ \mathrm{v} := \mathrm{E}_1(\mathrm{x}_1,\mathrm{v})\ \textbf{end}$$

$$\vdots$$

$$\mathrm{Evt_n} \triangleq \textbf{any } \mathrm{x_n} \textbf{ where } \mathrm{P_n(x_n,v)} \textbf{ then } \mathrm{v := E_n(x_n,v)} \textbf{ end}$$

$$\texttt{method}\, EVT(x)\ \texttt{returns}(y)\ \{Evt_1, ..., Evt_n\} \tag{1}$$

As mentioned earlier, one of the purposes of having a constructor statement is to assign a role to each event parameter. For the purpose of generating contracts, event parameters can be categorised with regards to the constructor statement that the event is listed in as follows:

- **Input parameter (x)**: the parameter has input behaviour (receives a value from the environment of the machine) and is listed as an input parameter in the constructor statement
- **Output parameter (y)**: the parameter has output behaviour (returns a value to the environment of the machine) and is listed as an output parameter in the constructor statement
- **Internal parameter (z)**: the parameter is a local variable to the event and is not listed as input/output parameter in the constructor statement

All input and output parameters that are listed in a constructor statement must exist in all listed events. They are used as a method's input or return arguments. Parameters that are not listed in the constructor statement are treated as internal parameters. Internal parameters are local variables to events. It is explained later how internal parameters are dealt with. Events $Evt_1 ... Evt_n$ can be represented based on constructor statement 1 as follows:

$$\mathrm{Evt_1} \triangleq \textbf{any } \mathrm{x,y,\ z_1} \textbf{ where } \mathrm{P_1(x,y,\ z_1,v)} \textbf{ then } \mathrm{v := E_1(x,y,\ z_1,v)} \textbf{ end}$$

$$\vdots$$

$$\mathrm{Evt_n} \triangleq \textbf{any } \mathrm{x,y,\ z_n} \textbf{ where } \mathrm{P_n(x,y,\ z_n,v)} \textbf{ then } \mathrm{v := E_n(x,y,\ z_n,v)} \textbf{ end}$$

In the above events, the union of $x$, $y$, and $z$ is equal to the set of all parameters of the respective event.

A number of pre-conditions may be defined for each method in Dafny to specify the conditions which must be true before a method is called. Pre-conditions are generated from invariants and some of the event guards. As was mentioned previously, conjunction of all model invariants are translated to a Dafny predicate. This predicate should be a pre-condition for all generated Dafny methods. The reason for this is that from the Event-B model, it is expected that invariants are true before execution of each event therefore it can be expected that invariants are true before execution of each method as well.

Event guards are used in both pre- and post-conditions depending on the role that they play in the event. Guards of each listed event in a constructor statement can be categorised as follows:

- **Typing guard (GT):** a guard that declares the type of an event's input or output parameters
- **Method guard (GP):** a guard that is being shared between all listed events in a constructor statement and only refers to input parameter and variables and not to output or internal parameters and is not a typing guard
- **Output guard (GO):** a guard that determines the value of an output parameter

- **Internal guard (GI):** a guard that refers to internal parameters and is not an output guard
- **Case guards (GC):** a guard that makes the enabling condition of its respective listed event distinct from other listed events and only refers to input parameters and machine variables

Each guard can only fall into one of the above categories. For generating pre-conditions from event guards we only consider method guards. If there is only one listed event then there would not be any case guards. With regards to the above categorisation, listed events in constructor statement 1 can be represented as follows:

```
$Evt_1 \triangleq$                    $Evt_n \triangleq$
any $x, y, z_1$                        any $x, y, z_n$
where                                  where
  $GT(x)$                                $GT(x)$
  $GT(y)$                                $GT(y)$
  $GP(x, v)$                             $GP(x, v)$
  $GC_1(x, v)$             ...           $GC_n(x, v)$
  $GI_1(x, z_1, v)$                      $GI_n(x, z_n, v)$
  $GO_1(x, y, z_1, v)$                   $GO_n(x, y, z_n, v)$
then                                   then
  $v := E_1(x, y, z_1, v)$               $v := E_n(x, y, z_n, v)$
end                                    end
```

To form method pre-conditions based on a constructor statement, model invariants ($I$), method guards (GP), and typing guards (GT) of input parameters are used. From Event-B model we expect that all invariants are true before execution of each event hence invariants are pre-conditions of the method. Method guards are conditions that are shared by all listed events and they must hold before execution of each of the listed events thus method guards are also pre-conditions of the method. Typing guards of input parameters are also a pre-condition of the method to guarantee that the input value is a valid one. Based on this, the following predicate is the pre-condition for the method that is generated based on the given constructor statement (1):

$$I(v) \land GT(x) \land GP(x, v) \tag{2}$$

For generating post-conditions from events we use case guards, internal guards, output guards and before-after predicates of event actions. A before-after predicate denotes the relation that exists between the value of a variable just before and just after the execution of an action. As mentioned before, a case guard makes the enabling condition of its respective listed event distinct from the other listed events. Case guards are used to determine which case is enabled at each time and therefore what is the expected outcome of the method. Internal parameters are used to determine the outcome of an event. An output parameter is treated as a free variable whose value is determined by the body of the method in Dafny. The value of an output parameter in Event-B is determined by output guards. Before-after predicate of actions specify the value of variables after the execution of an event. Due to this, internal guards, output guards and before-after predicates of each listed event are used to form post-conditions of the method. Therefore, each listed event in constructor statement 1 gives rise to generation of a predicate as follows where $i \in 1..n$:

$$GC_{\mathrm{i}}(x,v) \implies GT(y) \wedge (\exists z_{\mathrm{i}}.GI_{\mathrm{i}}(x,z_{\mathrm{i}},v) \wedge GO_{\mathrm{i}}(x,y,z_{\mathrm{i}},v) \wedge v\prime = E_{\mathrm{i}}(x,y,z_{\mathrm{i}},v)) \tag{3}$$

By convention, all primed variables appearing in a before-after predicate refer to the value of the variables after execution of an event and all unprimed variables refer to the value of the variable before the execution. It was discussed before that input and output parameters are treated as constants and free variables, respectively. To determine the value of internal parameters we existentially quantify over them. As mentioned before, if there exists only one listed event in a constructor statement then there is no case guard. In this case, only one post-condition will be generated in the following form:

$$GT_{\mathrm{out}} \wedge (\exists z \,.\, GI \wedge GO \wedge v\prime = E) \tag{4}$$

Given the constructor statement 1, the following Hoare triple is generated:

$$\{I \wedge GT_{\mathrm{in}} \wedge GP\} \; impl \; \{GT_{\mathrm{out}} \wedge ((GC_1 \implies (\exists z_1.GI_1 \wedge GO_1 \wedge v\prime = E_1)) \\ \wedge \cdots \wedge (GC_{\mathrm{n}} \implies (\exists z_{\mathrm{i}}.GI_{\mathrm{n}} \wedge GO_{\mathrm{n}} \wedge v\prime = E_{\mathrm{n}})))\} \tag{5}$$

where *impl* is a placeholder for the (yet to be constructed) correct implementation of the method. The above Hoare triple can be translated to annotations of a Dafny method which is generated based on constructor statement 1:

```
method Evt (x : T) returns(y : R)
requires Invariants()
requires GP
ensures GC1 ==> ∃ z1 :: GI1 && GO1 && v == E1
⋮
ensures GCn ==> ∃ zn :: GIn && GOn && v == En
```

Note that all non-typing and non-gluing invariants are translated to *Invariants* predicate in Dafny. Preservation of typing invariants and guards are checked by the Dafny type system, hence, they are implicitly part of the translated annotations. $T$ and $R$ in the above method declaration are the type of input and output parameters and are determined by typing guards.

## 4.3 New Proof Obligations

To ensure that the translation is sound and the generated code contracts are implementable a number of proof obligations should be discharged. These proof obligations are discussed in this subsection.

### 4.3.1 Internal and Output Parameter Feasibility

We should make sure that the specification of an internal or an output parameter is feasible by showing that there exists a value that satisfies internal and output guards:

$$\mathrm{I, GT, GP, GC_i} \vdash \exists y, z_\mathrm{i}.\ GI_\mathrm{i} \ \wedge\ GO$$

We split the above sequent to have two separate proof obligations for internal parameter feasibility and output feasibility. For internal parameter feasibility, if there are $n$ listed events in a constructor statement and $I$ is the conjunction of typing and model invariants and $GC_i$ and $GI_i$ where $i \in 1..n$ are the conjunction of case guards and the conjunction of internal guards of $i$th listed event respectively, then we can generate $n$ proof obligations with the following form:

$$\mathrm{I, GT, GP, GC_i} \vdash \exists z_\mathrm{i}.\mathrm{GI_i}$$

Similar to internal parameter feasibility proof obligation, if there are $n$ listed events in a constructor statement and $I$ is the conjunction of typing and model invariants and $GC_i$, $GI_i$ and $GO_i$ where $i \in 1..n$ are the conjunction of the case guards, the conjunction of the internal guards and the conjunction of the output guards of $i$th listed event respectively, then we can generate $n$ proof obligations with the following form:

$$\mathrm{I, GT, GP, GC_i, GI_i} \vdash \exists y.\mathrm{GO_i}$$

### 4.3.2 Disjointness

As explained in previous sections on generating post-conditions, if there is more than one listed event in a constructor statement then for each listed event (case) a predicate that specifies the behaviour of that event is generated and the conjunction of all the generated predicates would form the post-condition of the generated method. If there are situations where more than one of the cases are available then the generated Dafny specification would not be implementable. To avoid this, the specifier must make sure that case guards of all listed events are disjoint.

In principle, we could deal with non-disjoint events. But that would mean that the post-conditions corresponding to the separate events would need to be combined through disjunction. For pragmatic reasons we prefer to generate a separate Dafny post-condition for each listed event in an constructor statement. Since separate post-conditions are implicitly conjoined, the case guards need to be disjoint. This means that we remove any non-determinism arising from overlapping event guards prior to translation to Dafny contracts.

To prove the disjointness of the case guards a number of proof obligations must be discharged. If there are $n$ events listed in the constructor statement and $I$ is the conjunction of typing and model invariants and $GC_i$ where $i \in 1..n$ is the conjunction of all case guards of $i$th event then $n$ sequent can be generated with the following form:

$$\mathrm{I, GT, GP, GC_i} \vdash \neg\mathrm{GC_1} \wedge \ldots \wedge \neg\mathrm{GC_{i-1}} \wedge \neg\mathrm{GC_{i+1}} \wedge \ldots \wedge \neg\mathrm{GC_n}$$

The number of proof obligations can be reduced by simplifying the above sequent:

$$\mathrm{I, GT, GP, GC_i} \vdash \neg\mathrm{GC_{i+1}} \wedge \ldots \wedge \neg\mathrm{GC_n}$$

### 4.3.3 Completeness

As explained in previous sections, when there is more than one event listed in a constructor statement, case guards are one of the forming components of each generated post-condition. There might be situations in which the generated post-conditions do not specify the intended behaviour of the method for all possible values specified by the method's pre-conditions. This problem can be referred to as an incompleteness issue i.e. the specification is not complete. If there are $n$ events listed in the constructor statement and $I$ is the conjunction of typing and model invariants and $GC_i$ where $i \in 1..n$ is the conjunction of all case guards of $i$th event and $GP$ is the conjunction of method guards, to avoid incompleteness issue, the following sequent should be proved:

$$\mathrm{I, GT, GP} \vdash \mathrm{GC_1} \vee ... \vee \mathrm{GC_n}$$

Completeness is a desirable but not a must-have property and can be ignored by the modeller. If a method is executed in a state that satisfies the pre-condition but is not covered by any of the cases (i.e., all post-conditions are trivially satisfied), then any outcome for the method is allowed.

## 4.4 Invariant Preservation Proof

The validity of our transformation scheme is based on the fact that invariants of the Event-B model are also invariants of any Dafny implementation that satisfies the generated contract. Here we outline the proof of this for the case where a method contract is defined by one event. The proof easily generalises to multiple events as the cases are separate.

Assume we have a model **M** with variable $v$ and invariant $I(v)$ and event $Evt$:

$$\mathrm{Evt} \triangleq \textbf{any}\ \mathrm{x}\ \textbf{where}\ \mathrm{P(x,v)}\ \textbf{then}\ \mathrm{v := E(x,v)}\ \textbf{end}$$

Consider a method specified from an Event-B machine as follows:

$$\texttt{method}\ \mathrm{Evt(x)}\ \texttt{returns}()\ \{\mathrm{Evt}\}$$

The following Hoare triple characterises the correctness of the implementation of the contract generated from this method specification:

$$\{I(v) \wedge P(x,v)\}\ impl\ \{v = E(x, old(v))\} \tag{6}$$

where $old(v)$ refers to the value of variable $v$ before execution of $impl$. We have the following rule with regards to the operator $old$ [PM99]:

$$\frac{\{P(v)\}\ C\ \{\top\}}{\{P(v)\}\ C\ \{P(old(v))\}} \tag{7}$$

This rule says that if the pre-conditions of the triple hold for the value of the variable $v$ before the execution, then they still hold for the old value of the variable $v$ after the execution. By applying these new rules to Hoare triple 6, it can be rewritten as follows:

$$\{I(v) \wedge P(x,v)\}\ impl\ \{I(old(v)) \wedge P(x, old(v)) \wedge v = E(x, old(v))\} \tag{8}$$

The invariant preservation proof within Event-B guarantees the following:

$$I(old(v)) \wedge P(x, old(v)) \wedge v = E(x, old(v)) \implies I(v) \tag{9}$$

Finally, based on 8 and 9, we have that *impl* preserves the invariants:

$$\{I(v) \wedge P(x,v)\}\ impl\ \{I(v)\} \tag{10}$$

## 5 Example: Map Abstract Data Type

Our method and tool for transforming Event-B models to Dafny code contracts have been validated through a number of case studies including a map, a stack, and a queue abstract data type. Due to space limitation, we only present the map ADT case study in this paper.

A map (also called associative array) is an abstract data type which associates a collection of unique keys to a collection of values. The abstract level of the map which is shown Figure 1 is modelled as a partial function which links a key to a value. Types KEYS and VALUES are defined in the context as sets. The variable *map* is the only variable in this level and initialised with empty.

The abstract model (machine *m0*) is refined by machine *m1* (Figure 2). Event *Add* is refined by two events *Add1* and *Add2* to deal with two different cases. Event *Add1* will prepend a new key *k* to the sequence *keys* and value *v* to the sequence *values* . Event *Add2* modifies the value associated with an existing key *k*, in sequence of values. Event *Remove* is refined in this level to be able to remove an existing key *k* and its associated value from both the sequences. Now, machine *m1* has only those Event-B constructs that have a Dafny counterpart, hence it can be transformed to Dafny contracts. To do this, two constructor statements are provided:

```
method Add(k, v) returns () {Add1, Add2}
method Remove(k) returns () {Remove}
```

Given the above constructor statements and the presented transformation approach in Section 4, the following method contracts are generated:

```
method Add (k : KEYS, v: VALUES) returns()
requires Invariants()
ensures k !in keys ==> keys == [k] + old(keys) && values == [v] + old(values)
ensures k in keys ==> ∃ i :: i in (set k0| 0<=k0 && k0<=|old(keys)| - 1)
                          && old(keys)[i] == k
                          && values == old(values)[i:=v] && keys == old(keys)

method Remove (k : KEYS) returns()
requires Invariants()
requires k in keys
ensures ∃ i :: i in (set k0| 0<=k0 && k0<=|old(keys)| - 1)
           && old(keys)[i] == k && keys == old(keys)[..i] + old(keys)[i + 1..]
           && values == old(values)[..i] + old(values)[i + 1..]
```

```
Machine m1 Refines m0 Sees c0
Variables keys, values
Invariants
 inv1: keys ∈seq(KEYS)
 inv2: values ∈seq(VALUES)
 inv3: seqSize(keys) = seqSize(values)
 inv4: ∀i,j·i∈0··seqSize(keys)−1 ∧ j∈0··seqSize(keys)−1 ∧ i≠j ⇒ keys(i)≠keys(j)
 g_inv1:  map={i·i∈0··seqSize(keys)−1|keys(i)↦values(i)}
Initialisation keys :=∅, values :=∅
 Event Add1 refines Add                    Event Add2 refines Add
  any k,v                                  any k,v,i
  where                                    where
   grd1: k ∈KEYS                             grd1: k ∈KEYS
   grd2: v ∈VALUES                           grd2: v ∈VALUES
   grd3: k∉ran(keys)                         grd3: k ∈ran(keys)
 then                                        grd4: i∈0··seqSize(keys)−1
   act1: keys:=seqPrepend(keys,k)            grd5: seqElemAccess(keys,i)=k
   act2: values:=seqPrepend(values,v)      then
                                            act1: values:=seqElemUpdate(values,i,v)
 Event Remove refines Remove
  any k,i
  where
   grd1: k ∈KEYS
   grd2: i∈0··seqSize(keys)−1
   grd3: seqElemAccess(keys,i)=k
 then
   act1: keys := seqSliceToN(keys,i) seqConcat seqSliceFromN(keys,i+1)
   act2: values := seqSliceToN(values,i)  seqConcat seqSliceFromN(values,i+1)
```

Figure 2: Machine *m1*: Refinement of Abstract Model of Map

We have constructed Dafny implementations of the methods by hand and used the Dafny verifier to verify these against the generated contracts.

When the abstract model of the map is refined, the correctness of the refined event *Add2* can be proved without guard *grd3*. After the introduction of the method *Add* constructor statement, a number of proof obligations are generated. As discussed in 4.3, one of the proof obligations that should be discharged before the transformation takes place is the *internal parameter feasibility* PO. Internal parameter feasibility for event *Add2* without *grd3* has the following form:

I(v), k∈KEYS ∧ v∈VALUES
⊢
∃ i. i∈0..seqSize(keys)-1 ∧ seqElemAccess(keys,i) = k

*I(v)* denotes the model invariants. This proof obligation is not provable. A counter-example for this PO is a key that is not already in the sequence of the keys. To be able to prove this PO, *grd3* should be introduced. The new PO is as follows and can be discharged trivially:

I(v), k∈KEYS ∧ v∈VALUES, k∈ran(keys)
⊢
∃i. i∈0..seqSize(keys)-1 ∧ seqElemAccess(keys,i) = k

## 6 Tool Support

A Rodin plug-in has been developed to facilitate the automatic transformation of Event-B models to annotated Dafny method declarations. The plug-in extends Event-B with a new machine element for storing constructor statements. Whenever a new constructor statement is added to a machine, the plug-in will generate a number of new proof obligations based on the constructor statement (See 4.3). After discharging all PO's in Rodin the user can invoke the contract generator to generate a Dafny class including variable declarations, predicate *Invariants* and annotated Dafny method declarations. The tool has been validated by being applied to a number of small case studies.

## 7 Related and Future Work

As far as we are aware no attempt has been made to generate annotated Dafny programs from Event-B models and there is little research on linking the constructive approach to the analytical approach in the literature.

EventB2Dafny [CLR12] is a Rodin plug-in for translating Event-B proof obligations to Dafny code to use Dafny verifier as an external theorem prover for proving Event-B proof obligations. A Rodin plug-in called EventB2JML [RC14] has been developed to translate Event-B models to Java JML-specified code. EventB2JML implement Event-B models by producing a Java thread implementation for each event and does not impose any control flow on events. Also at the code level invariants need to be verified again using an static verifier. Mery and Monahan in [But09] proposed a transformation technique from an Event-B specification to an executable algorithm. In their approach the specification of the algorithm is provided at the start of the development in form of Spec# [BLS05] pre- and post-conditions and the algorithm is modelled in Event-B with regards to those code contracts. At the end the generated code from the Event-B model is verified against the code contracts in Spec#. Tasking Event-B [EB11] is a code generator that generates code from Event-B models to a target language but it does not support verification of the generated code.

Our current transformation rules allow us only to transform Event-B models of abstract data types to Dafny contracts. In the future we want to be able to transform Event-B model of more complex algorithms to Dafny contracts. One possible way is to use Event Refinement Structure (ERS) [FBR14]. By using ERS we will be able to impose algorithmic structures at Event-B level. This will ease the contract generation for more complex structures like loops.

## 8 Conclusion

We have presented a tool supported method for transforming Event-B models to Dafny code contracts. Using this method, Dafny users will enjoy the abstraction and refinement power of Event-B for building specifications that are correct with regards to an abstract specification. This approach provides a framework in which Event-B models can be implemented correctly in a sequential programming language. Our method also provides a way for merging Event-B events in order to generate contracts for a single method which implements different cases. We have also proved that if generated contracts are satisfied by an implementation in Dafny then it also satisfies the invariants of the abstract model and there is no need to reprove invariant preservation in the Dafny level. A tool in the form of a Rodin plug-in has been developed in order to implement the link. Given a machine and a number of constructor statements, the tool automatically generates relevant code contracts. A number of extra proof obligations (discussed in 4.3) should be discharged in order to guarantee the soundness of the generated contracts.

**Acknowledgements:** This work was funded in part by a Microsoft Research 2014 Software Engineering Innovation Foundation Award.

## Bibliography

[ABH+10] J.-R. Abrial, M. Butler, S. Hallerstede, T. Hoang, F. Mehta, L. Voisin. Rodin: an open toolset for modelling and reasoning in Event-B. *International Journal on Software Tools for Technology Transfer* 12(6):447–466, 2010.

[Abr10] J.-R. Abrial. *Modeling in Event-B: system and software engineering*. Cambridge University Press, 2010.

[BLS05] M. Barnett, K. Leino, W. Schulte. The Spec# Programming System: An Overview. In Barthe et al. (eds.), *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*. Lecture Notes in Computer Science 3362, pp. 49–69. Springer Berlin Heidelberg, 2005.

[BM13] M. Butler, I. Maamria. Practical theory extension in Event-B. In *Theories of Programming and Formal Methods*. Pp. 67–81. Springer, 2013.

[But09] M. Butler. Incremental design of distributed systems with Event-B. *Engineering Methods and Tools for Software Safety and Security* 22:131, 2009.

[But13] M. Butler. Mastering System Analysis and Design through Abstraction and Refinement. 2013.
http://eprints.soton.ac.uk/349769/

[CLR12] N. Catano, K. R. M. Leino, V. Rivera. The EventB2Dafny Rodin plug-in. In *Developing Tools as Plug-ins (TOPI), 2012 2nd Workshop on*. Pp. 49–54. 2012.

[CW96] E. M. Clarke, J. M. Wing. Formal methods: state of the art and future directions. *ACM Comput. Surv.* 28(4):626–643, 1996.

[DB08] L. De Moura, N. Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*. Pp. 337–340. Springer, 2008.

[DBR14] M. Dalvandi, M. Butler, A. Rezazadeh. From Event-B models to Dafny code contracts. 2014.
http://eprints.soton.ac.uk/373440/

[EB11] A. Edmunds, M. Butler. Tasking Event-B: An Extension to Event-B for Generating Concurrent Code. *Programming Language Approaches to Concurrency and Communication-cEntric Software*, p. 1, 2011.

[FBR14] A. S. Fathabadi, M. Butler, A. Rezazadeh. Language and tool support for event refinement structures in Event-B. *Formal Aspects of Computing*, pp. 1–25, 2014.

[HMLS09] C. Hoare, J. Misra, G. T. Leavens, N. Shankar. The verified software initiative: A manifesto. *ACM Comput. Surv* 41(4):1–8, 2009.

[Hoa69] C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Commun. ACM* 12(10):576–580, Oct. 1969.

[LAB+06] G. T. Leavens, J.-R. Abrial, D. Batory, M. Butler, A. Coglio, K. Fisler, E. Hehner, C. Jones, D. Miller, S. Peyton-Jones et al. Roadmap for enhanced languages and methods to aid verification. In *Proceedings of the 5th international conference on Generative programming and component engineering*. Pp. 221–236. 2006.

[Lei10] K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning*. Pp. 348–370. Springer, 2010.

[LW14] K. R. M. Leino, V. Wüstholz. The Dafny integrated development environment. *arXiv preprint arXiv:1404.6602*, 2014.

[PM99] A. Poetzsch-Heffter, P. Müller. A programming logic for sequential Java. In *Programming Languages and Systems*. Pp. 162–176. Springer, 1999.

[RC14] V. Rivera, N. Cataño. Translating event-B to JML-specified Java Programs. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing*. SAC '14, pp. 1264–1271. ACM, New York, NY, USA, 2014.