Verifiable Code Generation from Scheduled Event-B Models

MohammadSadegh Dalvandi, Michael Butler, Abdolbaghi Rezazadeh, and Asieh Salehi Fathabadi

University of Southampton, Southampton, United Kingdom
{md5g11, mjb, ra3, asf08r}@ecs.soton.ac.uk

Abstract. Scheduled Event-B (SEB) augments Event-B with a scheduling language to make the control flow in an Event-B model explicit and facilitate derivation of algorithmic structure in Event-B refinement. A concrete SEB model has a concrete algorithmic structure associated with it. Although this structure reduces the difficulty of code generation, there is still some gap between the model and executable code. This work formulates the translation of SEB models to a programming language called Dafny and proposes an approach in which a number of assertions are generated from the model that allows the verification of the generated code in a static program verifier.

1 Introduction

Event-B is a general purpose formal method which is designed to target a set of different domains including distributed systems, sequential programs, and embedded systems. This generality is achieved by not fixing the behavioural semantics of Event-B models [11]. Although this approach provides a great degree of freedom in using the method, the process of using Event-B in some domains (e.g. sequential program development) remains underdeveloped and not always easy to follow. In our previous work [6] we introduced Scheduled Event-B (SEB). SEB augments Event-B with a scheduling language and provides a number of refinement rules to facilitate derivation of algorithmic structure in Event-B refinement. It allows the modeller to introduce the algorithmic structure using the scheduling language from the very abstract level. The model, together with its algorithmic structure, is then refined towards a concrete level. The final refinement step results in a concrete Event-B model (i.e. a model with concrete data structures and no non-determinacy) and a concrete algorithmic structure (i.e. a deterministic algorithmic structure). It is assumed that this final refinement level is the closest possible model to the final implementation, i.e. a one to one mapping between the model constructs and the target language constructs exists. The most basic building block of a SEB model is an event. An event may have multiple actions (i.e. assignments) which model state changes and are executed simultaneously. If we consider assignments to be the most basic executable building blocks of an executable program, then each event should be

broken down to a number of assignments in the target language. Since event actions are considered to be executed at the same time, the syntactic ordering between them is not important. However, in a programming language, the order in which the assignments are sequentially executed may change the final state of the program. Due to this fact, when an event is sequentialised (i.e. its actions are translated to sequentially composed assignments), then the imposed ordering on assignments should be verified to prove that the sequential execution of the assignments will change the state in the same way that the execution of the atomic event changes it. This verification task can be carried out at the Event-B level. However, the problem with doing this in Event-B is a huge overhead caused by the introduction of new auxiliary variables, program counters, new invariants and events required for modelling and verification of the sequential execution of the actions of a single event. To avoid the aforementioned overhead, we can delegate this verification task to a program verifier which is much more sequential composition friendly than Event-B. This can be achieved by placing assertions in the program generated from an Event-B model in a way that proving the assertions implies that the sequentialised assignments change the state in the same way that the original atomic event does.

The above proposed approach takes advantage of abstraction and refinement offered by Event-B in developing an algorithm correctly and also benefits from modern and powerful program verifiers for verification of low level properties of the final implementation in order to prove that the final generated program implements the Event-B model correctly. In this paper we use the Dafny programming language and its verifier as our target language for implementing Event-B models.

This paper has two main contributions. First, it provides a set of rules for transforming a SEB model to an executable code in Dafny. Second, it introduces an approach for sequentialisation of atomic events and verifying its correctness using Dafny verifier. The rest of this paper is organised as follows: Section 2 provides background information required for understanding this work including an introduction to Event-B, Scheduled Event-B, and Dafny. Section 3 provides a set of transformation rules for transforming a SEB model to Dafny code. Section 4 discusses the verification of sequentialised model using Dafny verifier. Finally Section 5 discusses the future work and concludes the paper.

2 Background

2.1 Event-B

Event-B is a formal modelling language based on set theory and predicate logic for modelling and reasoning about systems, introduced by Abrial [2]. Event-B is greatly inspired by Action Systems [4] and the B-Method [1]. Modelling in Event-B is facilitated by an extensible platform called Rodin [3]. A model in Event-B usually has two main parts: a *context* and a *machine*. A context is the static part (types and constants) of a model which is specified using carrier sets, constants and axioms. A machine is the dynamic part (variables and events)

of a model which is specified 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 which are executed simultaneously. 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 can be parametrised by means of event parameters. A general Event-B event has the following form:

$$\boxed{\text{Evt} \triangleq \textbf{any } t \textbf{ when } P(t,v) \textbf{ then } S(t,v) \textbf{ end}}$$

where Evt is the name of the event, t is a set of parameters, v is the set of model variables, $P(t, v)$ is a set of guards and $S(t, v)$ is a set of actions. Modelling a complex system in Event-B can largely benefit from abstraction and refinement. Refinement is a stepwise process which starts from an abstract level and continues 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 [5]. The abstract level models the general purpose of the system by specifying *what* the system is supposed to achieve. Each refinement level adds more details on *how* the goal of the system can be achieved. It is essential that the correctness of each refinement is proved, i.e. proving that each refinement "displays the same behaviour" as the abstract one [15].

Refinement of an Event-B model may consist of refining existing events and/or adding new events, variables and invariants. The new events must not diverge. This means that they should not be enabled 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 (*data refinement*). Concrete variables are related 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 event.

2.2 Scheduled Event-B

In Event-B the control flow between events are implicitly encoded using event guards. Whenever the guards of an event are true, the event is considered to be enabled and can be executed. The lack of explicit control flow in Event-B can make algorithm and sequential program development difficult. To deal with the problem of control flow, in our previous work we introduced Scheduled Event-B (SEB) [6]. SEB augments Event-B with an explicit control flow construct called a *schedule*. Each refinement level has an associated schedule. A schedule provides the modeller with a set of abstract and concrete programming-like control constructs and allows the introduction of the control flow to a model from the very abstract level. SEB also provides a number of rules for schedule refinement. The rules allow the modeller to refine the abstract schedule along with the abstract model to a more concrete level. The final level of refinement will result in

a concrete algorithm with only deterministic control constructs left in it. Figure 1 shows the abstract and concrete control structures provided by SEB.

⟨*Schedule*⟩ ::= *Event*
| ⟨*Schedule*⟩ ; ⟨*Schedule*⟩
| ⟨*Schedule*⟩ □ ⟨*Schedule*⟩
| ⟨*Schedule*⟩*
| **if**(⟨*Cond*⟩){⟨*Schedule*⟩}, {**elseif**(⟨*Cond*⟩){⟨*Schedule*⟩}},[**else**{⟨*Schedule*⟩}]
| **while**(⟨*Cond*⟩){⟨*Schedule*⟩}

⟨*Cond*⟩ ::= *Predicate*

Fig. 1. The Scheduling Language. The language is presented in EBNF [18].

The simplest form of a schedule is a single event. *Event* denotes an event in the schedule. A schedule may contain one or more Event-B events. A sequential order can be imposed by the sequential composition operator (;). Non-deterministic choice ($S_1 \square S_2$) and iteration (S^*) are the abstract control structures. Iteration is required to be finite. This is enforced by proving convergence of events. The aforementioned control structures allow us to retain the event structure (guards and actions together) so that data refinement reasoning is localised to pairs of corresponding abstract and refining events using the standard definition of the Event-B refinement. The concrete control structures include deterministic `if..else` branches and `while` loops with explicit conditions (*Cond*). The branch and loop conditions should be valid Event-B predicates as defined in [2]. Non-deterministic choices and iterations can be refined to deterministic branches and loops, respectively. Schedule refinement rules are defined in [6]. Figure 2 depicts how a schedule is refined alongside with the Event-B model. To illustrate scheduled Event-B, we use the binary search algorithm presented in [6] here. We only provide the most concrete Event-B model of the search algorithm:

Machine $m3$ **refines** $m2$ **Sees** $c0$
Variables r, k, i, j
Initialisation $r := 0, k := (n-1)/2, i := 0, j = n-1$

Event *search_inc*
refines *search*
where
grd1: $f(k) < v$
then
act1: $k := (k+j+1)/2$
act2: $i := k+1$
End

Event *search_dec*
refines *search*
where
grd1: $f(k) > v$
then
act1: $k := (i+k-1)/2$
act2: $j := k-1$
End

Event $found$
refines $found$
where
grd1: $f(k) = v$
then
act1: $r := k$
End

In the above model, f represents an array modelled as a total function in Event-B ($f \in 0..n-1 \rightarrow \mathbb{Z}$). The concrete schedule associated with the above model is as follows:

$$\begin{array}{c} initialisation; \\ \textbf{while}(f(k) \neq v)\{\textbf{if}(f(k)<v)\{search\_inc\}\ \textbf{else}\{search\_dec\}\}; \\ found \end{array}$$

The above schedule defines the control flow of the Event-B model. The schedule contains a `while` loop and an `if..else` branch with explicit conditions. The explicit conditions in the schedule allow the guards in the events to be eliminated when generating the code (see Section 3). Variables r, k, i, and j are of type integer. f is a sorted array defined in the context $c0$. The above model does not include the context $c0$ or any of the abstract machines.

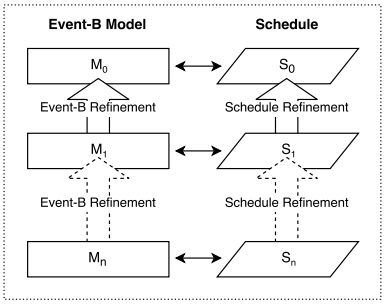

Fig. 2. Event-B and Schedule Refinement

2.3 Dafny

Dafny is an imperative, class-based language [13], which allows both strong and weak typed variables. Dafny implements the verification method of Hoare logic where a program can be specified with pre- and post-conditions. In the Dafny language, pre- and post-conditions are influenced by the Eiffel language [16] and the concept of *design-by-contract* [17]. Dafny is an object-oriented programming language with generic classes and allows creation of objects which gives rise to pointers [14]. Despite the fact that Dafny is a class-based language, it does not support subclasses and inheritance. However, there is a built-in **object** type that is a super-type of all class types. Dafny supports inductive datatypes and has its own specification constructs. Standard pre- and post-conditions, framing constructs and termination metrics are included in the specifications. In this paper we call these specification constructs, *code contracts*. The language also offers recursive functions, sets, sequences and some other features to support specification. Dafny allows the definition of **ghost** variables. A ghost variable is a variable that is used by the Dafny verifier and ignored at run time. A ghost variable is used for specification purposes only and does not appear in any part of the implementation. Specifications and ghost variables are omitted by the compiler and are used just during the verification process.

The Dafny verifier attempts to verify different parts of a program locally (modular verification) and infer the correctness of the whole system from those

locally verified parts. The Dafny verifier translates a Dafny program to an immediate verification language known as Boogie 2 [12]. This is done in a way that the correctness of the generated Boogie program implies the correctness of the Dafny program. First-order verification conditions then are generated by the Boogie tool and passed to the Z3 SMT solver [8].

3 Translating Concrete SEB Models to Dafny

A scheduled Event-B model, in its final refinement level, has a concrete schedule (i.e. the schedule has only events, `;`, `while` and/or `if..else`) associated with it. It is assumed that all constructs in the model are refined to a concrete level and all non-deterministic assignments are replaced with deterministic ones. The concrete schedule is assumed to be a correct refinement of the abstract one with respect to the refinement rules introduced in [6]. This section explains how a SEB model is translated to Dafny implementation. We will use the model of the binary search algorithm presented in Section 2.2 as an example to illustrate the translation. To formulate the translation of SEB models to Dafny, we define a function called $SEB2DFY$. The function accepts an Event-B model (M)(consisting of a machine and the context it sees) and a schedule (S) and returns generated code and contracts:

$$SEB2DFY(M,S) \triangleq \ SEB2DFY_{\text{class}}(M,S) \tag{1}$$

Function $SEB2DFY_{\text{class}}$ defines a class including a method implementing the algorithm. They are discussed in the following sections. The input model M and schedule S are expected to be refined to a concrete level as explained earlier.

3.1 Dafny Method Generation

The focus of SEB is on development and verification of sequential algorithms. SEB does not yet cover concepts like method calls or recursions. With this in mind, for the purpose of code generation, it would be an appropriate decision to map a SEB model to a class with a method implementing the algorithm based on the provided schedule S. Based on this decision, function $SEB2DFY_{\text{class}}$ will return a class with a single method with the same name as the model which was passed to it:

$$\begin{aligned} SEB2DFY_{\text{class}}(M,S) \triangleq \ & \texttt{class}\ mchn\{ \\ & \quad SEB2DFY_{\text{mtd}}(M,S) \\ & \} \end{aligned} \tag{2}$$

Function $SEB2DFY_{\text{mtd}}(M,S)$ defines the way that the method should be generated:

$$
\begin{aligned}
SEB2DFY_{\text{mtd}}(M,S) \triangleq\ & \texttt{method}\ mchn(SEB2DFY_{\text{args}}(M)) \\
& SEB2DFY_{\text{pre}}(M) \\
& \{ \\
& \quad SEB2DFY_{\text{var}}(v_1, inv_{v1}) \\
& \quad SEB2DFY_{\text{var}}(v_2, inv_{v2}) \\
& \quad \vdots \\
& \quad SEB2DFY_{\text{var}}(v_{\text{n}}, inv_{vn}) \\
& \quad SEB2DFY_{\text{alg}}(M,S) \\
& \}
\end{aligned} \tag{3}
$$

In the above class and method *mchn* is a placeholder for the name of the machine being translated. If there is a value that the algorithm needs to receive in order to perform a specific task on it such as an unsorted array to be sorted, it is usually declared and specified in the model context using constants and axioms. In this case the constant is mapped to an input argument which is passed to the method and the axioms specifying it are transformed to method pre-conditions. Functions $SEB2DFY_{\text{args}}(M)$ and $SEB2DFY_{\text{pre}}(M)$ are used to generate the method's input arguments and its necessary pre-conditions. Assume that we have a model containing machine *mchn* and a context with constants $a_1, ..., a_{\text{k}}$ where each constant is of type $T_1, ..., T_{\text{k}}$, respectively. Function $SEB2DFY_{\text{args}}(M)$ has the following definition:

$$
SEB2DFY_{\text{args}}(M) \triangleq\ a_1 : T_1, ..., a_{\text{k}} : T_{\text{k}} \tag{4}
$$

If the context of model M has n axioms specifying input arguments then function $SEB2DFY_{\text{pre}}(M)$ has the following definition:

$$
\begin{aligned}
SEB2DFY_{\text{pre}}(M) \triangleq\ & \texttt{requires}\ SEB2DFY_{\text{pred}}(axm_1) \\
& \vdots \\
& \texttt{requires}\ SEB2DFY_{\text{pred}}(axm_{\text{n}})
\end{aligned} \tag{5}
$$

The `requires` keyword is used in Dafny to declare method pre-conditions. The function $SEB2DFY_{\text{pred}}$ transforms an Event-B predicate to its Dafny equivalent. Function $SEB2DFY_{\text{var}}$ gives rise to generation of variable declarations including typing invariants. Finally, $SEB2DFY_{\text{alg}}(M,S)$ generates the implementation and necessary contracts. This function will be discussed in detail in the rest of this paper.

3.2 Algorithm Generation

An important step in transforming a SEB model to Dafny code is the generation of the code implementing the algorithm. Function $SEB2DFY_{\text{alg}}(M,S)$ formulates this step. Schedule S contains key information about the algorithmic structure of model M. A schedule is usually comprised of a number of subschedules (which are either a control structure or a single event) ordered using sequential composition operator:

$$S \triangleq S_1 \,;\, ... \,;\, S_{\mathrm{n}}$$

If a schedule is comprised of a number of sub-schedules like the above, then function $SEB2DFY_{\mathrm{alg}}(M, S)$ is defined as follows:

$$\begin{array}{rl} SEB2DFY_{\mathrm{alg}}(M, S) \triangleq & SEB2DFY_{\mathrm{alg}}(M, S_1) \\ & \vdots \\ & SEB2DFY_{\mathrm{alg}}(M, S_{\mathrm{n}}) \end{array} \tag{6}$$

As mentioned before, a sub-schedule may be a control structure (branch or loop) or an event. The general form of a branch sub-schedule is as follows:

$$S_{\mathrm{i}} \triangleq \texttt{if(}c_1\texttt{)\{ }s_1\texttt{ \} elseif(}c_2\texttt{)\{ }s_2\texttt{ \} ... else\{ }s_{\mathrm{n}}\texttt{ \}}$$

where $c_1, ..., c_{\text{n-1}}$ are branch conditions (in the form of Event-B predicates) and $s_1, ..., s_{\mathrm{n}}$ are schedules. In this case the definition of $SEB2DFY_{\mathrm{alg}}(M, S_{\mathrm{i}})$ is as follows:

$$\begin{array}{rl} SEB2DFY_{\mathrm{alg}}(M, S_{\mathrm{i}}) \triangleq & \mathrm{if}(SEB2DFY_{\mathrm{pred}}(c_1))\{ \\ & \quad SEB2DFY_{\mathrm{alg}}(M, s_1) \\ & \} \\ & \mathrm{elseif}(SEB2DFY_{\mathrm{pred}}(c_2))\{ \\ & \quad SEB2DFY_{\mathrm{alg}}(M, s_2) \\ & \} \\ & \vdots \\ & \mathrm{else}\{ \\ & \quad SEB2DFY_{\mathrm{alg}}(M, s_{\mathrm{n}}) \\ & \} \end{array} \tag{7}$$

If sub-schedule S_{j} is a loop then it has the following general form:

$$S_{\mathrm{j}} \triangleq \texttt{while(}c\texttt{)\{ }s\texttt{ \}}$$

where c is the loop condition and s is a schedule representing the body of the loop. The definition of $SEB2DFY_{\mathrm{alg}}(M, S_{\mathrm{j}})$ is as follows:

$$\begin{array}{rl} SEB2DFY_{\mathrm{alg}}(M, S_{\mathrm{j}}) \triangleq & \mathrm{while}(SEB2DFY_{\mathrm{pred}}(c))\{ \\ & \quad SEB2DFY_{\mathrm{alg}}(M, s) \\ & \} \end{array} \tag{8}$$

Now that we defined $SEB2DFY_{\mathrm{alg}}$ for branches and loops, we need one more definition for the case that a (sub-)schedule is a single event. This case will be discussed in the next section in detail.

3.3 Events to Sequential Statements

The most basic component of a schedule is an event. Event-B events usually have a number of guards and actions. In [6] we showed that a correct schedule allows us to eliminate event guards because guards should follow explicit schedule guards. Elimination of event guards is facilitated through a number of guard propagation and elimination rules. These rules allow us to propagate explicit schedule guards (loop or branch conditions) to events and eliminate event original guards safely. As an example consider the following schedule:

$$\textbf{while}(a)\{\textbf{if}(b)\{evt\}\}$$

where a and b are predicates and evt is an event. If the control reaches event evt then the schedule guarantees that the following condition holds right before the execution of evt:

$$a \ \wedge b$$

Guards of evt can be eliminated safely in the program if the following condition holds:

$$a \ \wedge b \ \Rightarrow \ grd(evt)$$

where $grd(evt)$ denotes evt guards. Guard propagation and elimination rules are discussed in detail in [6].

If we eliminate event guards then we are left with event actions. Since event actions are assumed to be executed simultaneously in Event-B, no ordering is assumed between them. Translation of an event to code involves sequentialisation of event actions and imposing a suitable sequencing on execution of them using sequential composition.

As explained before, since Event-B events are executed atomically, the syntactic ordering between the actions are not important. However when actions of an event are translated to a series of assignments in a programming language, the order in which they appear in the program can change the outcome. For instance, consider the following event from the model of binary search algorithm introduced earlier:

Event *search_inc*
refines *search*
where
grd1: $f(k) < v$
then
act1: $k := (k+j+1)/2$
act2: $i := k + 1$
End

If the actions of the event are translated to sequentially composed assignments in Dafny with the same order that they have in the event, the resulting program will change the state in a different way than the event. This is due to the fact that the right-hand side of action *act2* is dependent on variable k whose value is being updated by action *act1*. In this case the problem disappears

if we re-order the actions since act1 is independent of variable i. However this is not a general solution since action may be mutually dependent. We can use auxiliary variables to make the right-hand side of actions independent from the left-hand side of the other actions. To do this, one auxiliary variable should be introduced for each variable that is being modified and used by the event. The auxiliary variable needs to be initialised with value of its associated variable. All the occurrences of the left-hand side variables in the right-hand side expressions of the actions should then be replaced by the auxiliary variables. For instance, the actions of the above event should be translated to the following code:

```
var aux_k := k;
k := (aux_k + j + 1) / 2;
i := aux_k + 1;
```

As can be seen in the above code, the ordering between third and fourth assignments, with the help of auxiliary variables, does not matter any more. To formulate this, assume that we have the following general event:

Event *evt*
where
    $G(v)$
then
    act1: $v_1 := E_1(v)$
    $\vdots$
    actn: $v_n := E_n(v)$
End

The definition of $SEB2DFY_{\text{alg}}(M, S)$ when S is a single event evt ($S \triangleq evt$) is as follows:

$$\begin{aligned} SEB2DFY_{\text{alg}}(M, evt) \triangleq\ & SEB2DFY_{\text{ghost}}(M, evt) \\ & SEB2DFY_{\text{aux}}(v_1) \\ & \vdots \\ & SEB2DFY_{\text{aux}}(v_n) \\ & SEB2DFY_{\text{act}}(v_1 := E_1[v \backslash aux\_v]) \\ & \vdots \\ & SEB2DFY_{\text{act}}(v_n := E_n[v \backslash aux\_v]) \\ & SEB2DFY_{\text{post}}(M, evt) \end{aligned} \tag{9}$$

where v and aux_v are sets of model variables and auxiliary variables, respectively. $E[v \backslash aux\_v]$ is the result of substituting aux_v for all occurrences of v in E. Functions $SEB2DFY_{\text{ghost}}$ and $SEB2DFY_{\text{post}}$ which appeared on the first and last lines of the above definition, are used for contract generation purposes which will be discussed in the next section. Function $SEB2DFY_{\text{aux}}$ receives a variable and generates an auxiliary variable declaration and initialisation:

$$SEB2DFY_{\text{aux}}(v) \triangleq\ var\ aux\_v := v; \tag{10}$$

Function $SEB2DFY_{\text{act}}$ receives an action (a) in the form of $v := E(v)$ and has the following definition:

$$SEB2DFY_{\text{act}}(a) \triangleq \ v \ := SEB2DFY_{\text{exp}}(E); \tag{11}$$

$SEB2DFY_{\text{exp}}$ transforms an Event-B expression to Dafny based on a set of translations rules for translating Event-B expressions to Dafny. Due to space limitation, we omit expression and predicate translation rules here.

4 Verification of Event Sequentialisation

In the previous section we discussed the sequentialisation of an event in detail. This section discusses how we can prove its correctness. In order to be able to verify the sequentialisation, along with the translation of the actions of each event, we generate assertions representing the expected behaviour of the program based on before-after predicate of those actions.

Before we continue to explain our approach for verifying the correctness of event sequentialisation, we justify why this step is done at Dafny level. Although it is possible to sequentialise an event in Event-B and to impose a sequential order on the execution of its actions and prove its correctness, it involves the overhead of adding a number of new events, guards, and program counters and also extending the scheduling language and refinement rules proposed in [6] to accommodate sequentialisation. Due to this, performing event sequentialisation in a programming language designed for development of sequential programs seems to be a more appropriate choice than trying to sequentialise actions in Event-B level which involves the aforementioned overhead.

Previously, we discussed how an event is translated to a number of of sequential statements in Dafny. A program (or part of a program) in Dafny may be specified (annotated) using code contracts (method's pre- and post-conditions and assertions). The Dafny verifier checks an annotated program text against its specification in order to prove that the program behaves as intended. In order to prove that an event is correctly transformed to Dafny code (sequentialised correctly), a number of code contracts should be generated from the event in the form of assertions.

The way that the state is changed by an Event-B event can be expressed by a before-after predicate. By transforming an event's before-after predicate to Dafny assertions, we will be able to verify the correctness of event sequentialisation. Functions $SEB2DFY_{\text{ghost}}$ and $SEB2DFY_{\text{post}}$ will generate the necessary ghost variables and assertions for verification of sequentialisation.

To illustrate the generation of required assertions, recall event *search_inc* from the model of the binary search algorithm introduced in the previous sections:

Event *search_inc*
refines *search*
where
grd1: $f(k) < v$
then
act1: $k := (k+j+1)/2$
act2: $i := k + 1$
End

After the execution of the above event, the value of variables k and i are changed in the following way:

$$k' = (k + j + 1)/2 \ \wedge \ i' = k + 1 \tag{12}$$

where k' and i' are the value of variables k and k after the execution of the event and k and i are the value of variable k and i before the execution of *search_inc*. A block of code implements event *search_inc* correctly, if it has the same behaviour as the event, i.e. it changes the state in the same way. If we want to verify that a block of code sequentialises the event actions correctly, then we need to generate assertions like (12) in Dafny based on event before-after predicates.

The challenge here is how to refer to the before and after values (unprimed and primed variables) in an assertion in Dafny. A variable in a Dafny assertion always refers to the current value of the variable. So to be able to transform a before-after predicate like (12) to an assertion, we need to have access to the value of variables before execution of the block of code implementing the event. We transform the event *search_inc* and its before-after predicate to Dafny code and assertions using ghost variables for storing the before values (unprimed variable) of variables k and i:

```
1 ghost var old_k = k;
2 ghost var old_i = i;
3 var aux_k := k;
4 var aux_i := i;
5 k := (aux_k + j + 1) / 2;
6 i := aux_k + 1;
7 assert k == (old_k + j + 1) / 2;
8 assert i == old_i + 1;
```

Variables *old_k* and *old_i* are ghost variables and are used to keep the before value of variables k and i, respectively. A ghost variable is a variable that is used by the Dafny verifier and ignored at run time. Function $SEB2DFY_{\text{ghost}}$ facilitates the generation of the required ghost variables. It receives the model and a specific event, and works directly on the event before-after predicate and generates one ghost variable for every unprimed variable that appeared in the event before-after predicate and initialises it with the value of the unprimed variables. For practical reasons, we decided to generate one `assert` statement per each action. The assertions are yielded by replacing unprimed variables in the before-after predicate with their ghost counterparts and primed variables with

the original variables. Function $SEB2DFY_{\text{post}}$ generates assertions required for verification.

In the above code, lines 1-2 and 7-8 are required for verification only. Lines 3-6 are the code implementing the event. The guard of the event is not used here. This is because the guard is available to the Dafny verifier since it would be a condition in the `if` statement generated based on the schedule given in Section 2.2. Any code that can satisfy the assertions (lines 7-8) can replace lines 3-6.

Apart from the verification of sequentialisation, having assertions in the generated code is useful for another purpose as well. The embedded assertions make it possible to verify further amendments to the implementation at the code level to prove that the new code complies with its abstract specification (event). For instance in the above code, any implementation that satisfies the assertions (lines 7-8) can replace the code implementing the abstract event (lines 3-6).

5 Conclusion and Future Work

In this paper, we proposed an approach for generation of verifiable implementation from Scheduled Event-B [6] models. The paper outlined the necessary rules for translating the algorithmic structure of a model together with rules for sequentilisation of Event-B atomic events in Dafny. We also introduced a way for generating Dafny assertions that allows us to verify the correctness of the sequentialisation phase. Overall, our approach benefits from combining the verification power of Dafny together with abstraction and refinement offered by Event-B. We have applied this approach to a number of examples, including the model of Schorr-Waite algorithm introduced in [6] and generated code and contracts required for verification.

In our previous work [7], we introduced another approach for generating Dafny code contracts from Event-B models. The proposed approach generates Dafny method pre- and post-conditions from a group of atomic Event-B events in a way that any implementation that satisfies the generated pre- and post-conditions is considered to be a correct implementation of the Event-B abstract model. There are two main differences between the approach presented in this paper and the one presented in our previous work. First, the previous approach only focuses on contract (method's pre- and post-conditions) generation and no implementation is generated while in this work, both implementation and code contracts (i.e. assertions) are generated. The second difference is the granularity of generated contracts. In the previous work we generated contracts at method level in a way that the overall behaviour of the method was annotated, while in this work, the contracts are generated in a lower granularity where the behaviour of small blocks of code (sequentially composed assignments) inside a method are annotated.

In [10], we extended Event-B code generation tool [9] and applied it to an Event-B model of a learning-based RTM (Runtime Management system) in embedded system design to generate C implementation from the model. The code generation tool supports portability of the platform-independent model from

which platform-specific implementations are automatically generated. However, our experience shows that there are a number of limitations with the current Event-B code generation tool. The first limitation is that the algorithmic structure of the program can only be introduced at the final level of refinement and the modeller cannot benefit from refinement in derivation of algorithmic structure. The other limitation is that the current structuring language is too restrictive and does not allow the modeller to define nested programs. Another limitation is that the verification is only done at the Event-B level and no verification is performed on the generated code.

In future, we want to mechanise the process of generation of the code and contracts from scheduled Event-B models. We also envisage to apply the approach presented in this paper to other case studies including the Event-B model of RTM introduced in [10] to further validate our approach.

Acknowledgments. This work was funded in part by the EPSRC PRiME Project (EP/K034448/1), www.prime-project.org.

References

1. J. R. Abrial, M. K. O. Lee, D. S. Neilson, P. N. Scharbach, and I. H. Sørensen. The B-method. In Søren Prehn and Hans Toetenel, editors, *VDM '91 Formal Software Development Methods*, volume 552 of *LNCS*, pages 398–405. Springer Berlin Heidelberg, 1991.
2. Jean-Raymond Abrial. *Modeling in Event-B: system and software engineering*. Cambridge University Press, 2010.
3. Jean-Raymond Abrial, Michael Butler, Stefan Hallerstede, ThaiSon Hoang, Farhad Mehta, and Laurent 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.
4. R. J. R. Back and F. Kurki-Suonio. Distributed cooperation with Action Systems. *ACM Trans. Program. Lang. Syst.*, 10(4):513–554, October 1988.
5. Michael Butler. Mastering system analysis and design through abstraction and refinement. 2013.
6. Mohammadsadegh Dalvandi, Michael Butler, and Abdolbaghi Rezazadeh. Derivation of algorithmic control structures in Event-B refinement. *Science of Computer Programming*, 148(Supplement C):49 – 65, 2017. Special issue on Automated Verification of Critical Systems (AVoCS 2015).
7. Mohammadsadegh Dalvandi, Michael J. Butler, and Abdolbaghi Rezazadeh. Transforming Event-B models to Dafny contracts. *ECEASST*, 72, 2015.
8. Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.
9. Andrew Edmunds and Michael Butler. Tasking Event-B: An extension to Event-B for generating concurrent code. Event Dates: 2nd April 2011, February 2011.
10. Asieh Salehi Fathabadi, Michael J. Butler, Sheng Yang, Luis Alfonso Maeda-Nunez, James Bantock, Bashir M. Al-Hashimi, and Geoff V. Merrett. A model-based framework for software portability and verification in embedded power management systems. *Journal of Systems Architecture*, 82:12 – 23, 2018.

11. Stefan Hallerstede. On the purpose of Event-B proof obligations. In Egon Börger, Michael Butler, JonathanP Bowen, and Paul Boca, editors, *Abstract State Machines, B and Z*, volume 5238 of *LNCS*, pages 125–138. Springer Berlin Heidelberg, 2008.
12. K Rustan M Leino. This is Boogie 2. *Manuscript KRML*, 178:131, 2008.
13. K Rustan M Leino. Dafny: An automatic program verifier for functional correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 348–370. Springer, 2010.
14. K. Rustan M. Leino and Rosemary Monahan. Dafny meets the verification benchmarks challenge, 2010.
15. Grant Malcolm and Joseph A Goguen. *Proving correctness of refinement and implementation*. Oxford University. Computing Laboratory. Programming Research Group, 1994.
16. Bertrand Meyer. *Eiffel: The Language*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992.
17. Bertrand Meyer. *Design by contract*. Prentice Hall, 2002.
18. Niklaus Wirth. Extended Backus-Naur Form (EBNF). *ISO/IEC*, 14977:2996, 1996.