Towards Generating SPARK from Event-B Models

Sanjeevan Sritharan and Thai Son Hoang(✉)

ECS, University of Southampton, Southampton, UK
{ss6n17,t.s.hoang}@soton.ac.uk

Abstract. This paper presents an approach to generate SPARK code from Event-B models. System models in Event-B are translated into SPARK packages including proof annotations. Properties of the Event-B models such as axioms and invariants are also translated and embedded in the resulting models as pre- and post-conditions. This helps with generating SPARK proof annotations automatically hence ensuring the correct behaviour of the resulting code. A prototype plug-in for the Rodin has been developed and the approach is evaluated on different examples. We also discuss the possible extensions including to generate scheduled code and data structures such as records.

Keywords: Event-B · SPARK · Code generation · Rodin platform

1 Introduction

Ensuring properties of safety- and security-critical systems is paramount. Event-B [1] is a formal modelling method which enables the design of systems, using mathematical proofs ensuring the conformity of the system to declared safety requirements. SPARK [4] is a programming language making use of static analysis tools which verify written code correctly implements the properties of the system as specified in the form of written proof annotations (e.g., pre- and post-conditions). SPARK has been used in many industry-scale projects to implement safety-critical software. However, manually writing SPARK proof annotations can be time-consuming and tedious.

Our motivation is to develop a tool-supported approach to translate an Event-B model into a SPARK package, including proof annotations and other structures, from which manually written SPARK code can be verified, hence ensuring the correct behaviour of the software. Event-B supports development via refinement, allowing details to be consistently introduced into the models. Properties of the systems such as invariances therefore are easier to be discovered compare to SPARK. One aim for our approach is to cover as much as possible the Event-B mathematical language that can be translated into SPARK.

Thai Son Hoang is supported by the HiClass project (113213), which is part of the ATI Programme, a joint Government and industry investment to maintain and grow the UK's competitive position in civil aerospace design and manufacture.

B. Dongol and E. Troubitsyna (Eds.): IFM 2020, LNCS 12546, pp. 103–120, 2020.
https://doi.org/10.1007/978-3-030-63461-2_6

Our contribution is an approach where Event-B sets and relations are translated as SPARK Boolean arrays. A library is built to support the translation. Furthermore, properties of the systems such as axioms and invariants are translated and embedded in SPARK as pre- and post-conditions. These properties, in particular invariance properties, are often global system properties ensuring the safety and consistency of the overall system, and are often difficult to be discovered. Using these conceptual translation rules, a plug-in was created for the Rodin platform [2] and was evaluated with several Event-B models. From the evaluation, we discuss different possible extensions including to generate scheduled code and records data structure.

The rest of the paper is structured as follows. Section 2 gives some background information for the paper. This includes an overview of Event-B, SPARK, and our running example. Our main contribution is presented in Sect. 3. We discuss limitation and possible extensions of the approach in Sect. 4. Section 5 reviews the related work. Finally, we summary and discuss future research direction in Sect. 6.

2 Background

2.1 Event-B

Event-B [1] is a formal method used to design and model software systems, of which certain properties must hold, such as safety properties. This method is useful in modelling safety-critical systems, using mathematical proofs to show consistency of models in adhering to its specification. Models consist of constructs known as machines and contexts. A *context* is the static part of a model, such as *carrier sets* (which are conceptually similar to types), *constants*, and *axioms*. Axioms are properties of carrier sets and constants which always hold. *Machines* describe the dynamic part of the model, that is, how the state of the model changes. The state is represented by the current values of the *variables*, which may change values as the state changes. *Invariants* are declared in the machine, stating properties of variables which should always be true, regardless of the state. *Events* in the machine describe state changes. Events can have *parameters* and *guards* (predicates on variables and event parameters); the guard must hold true for event execution. Each event has a set of *actions* which happen simultaneously, changing the values of the variables, and hence the state. Every machine has an initialisation event which sets initial variable values. An important set of proof obligations are invariant preservation. They are generated and required to be discharged to show that no event can potentially change the state to one which breaks any invariant, a potentially unsafe state.

An essential feature of Event-B, stepwise refinement, is not used within the scope of this project, which focuses on Event-B's modelling of a single abstraction level model. Further details on refinement can be found in [1,10]. In Sect. 2.3 we present the our running example including the Event-B model.

2.2 SPARK

SPARK [4] is a programming language used for systems with high safety standards. It includes tools performing static verification on programs written in the language. SPARK is a subset of another programming language, Ada [5], which is also used for safety-critical software. SPARK removes several major constructs from Ada, allowing feasible and correct static analysis.

SPARK includes a language of annotations, which are specifications for a SPARK program, clarifying what the program should do [13]. While program annotations focus on the flow analysis part of static analysis, focusing on things such as data dependencies, proof annotations support "assertion based formal verification". In particular, a specification for a SPARK procedure has the following aspects:

– Pre aspect: pre-conditions which are required to hold true on calling a subprogram, without which the subprogram has no obligation to work correctly.
– Post: post-conditions which should be achieved by the actions of a subprogram, provided the pre-conditions held initially
– Global aspect: specifying which global variables are involved in this subprogram, and how they are used.
– Depends aspect: which variables or parameters affect the new value of the modified variables

Proof annotations also involve loop invariants, which are conditions which hold true in every iteration of a loop.

This mix of proof and program annotations ensure that any implementation written in SPARK adheres to its specification, producing reliable, safe software.

2.3 A Running Example

To illustrate our approach, we use an adapted version of the example of a building access system from [6]. We only present a part of the model here. The full model and the translation to SPARK is available in [17].

The context declares the sets of PEOPLE and BUILDING with a constant maxsize to indicate the maximum number of registered users. Note that we have introduced axioms to constrain the size of our carrier sets and fix the value of the constant as it is necessary for our generated SPARK code. Normally, Event-B models are often more abstract, e.g., there are no constraints on the size of the carrier sets.

```
context c0
sets PEOPLE BUILDING
constants maxsize
axioms
 @finite−PEOPLE: finite(PEOPLE)
 @card−PEOPLE: card(PEOPLE) = 10
 @finite−BUILDING: finite(BUILDING)
```

```
  @card−BUILDING: card(BUILDING) = 4
  @def−maxsize: maxsize=3
end
```

The machine models the set of register users, their location and their permission for accessing buildings.

```
machine m0
variables register size location permission
invariants
  @inv1: register ⊆ PEOPLE
  @inv2: size ≤ maxsize
  @inv3: location ∈ register ⇸ BUILDING
  @inv4: permission ∈ register ↔ BUILDING
  @inv5: location ⊆ permission
events
  ...
end
```

Invariant @inv5 specifies the access control policy: a register user can only be in a building where they are allowed.

Initially, there are no users in the system, hence all the variables are assigned the empty set.

```
event INITIALISATION
begin
  @init−register: register := ∅
  @init−size: size := 0
  @init−location: location := ∅
  @init−permission: permission := ∅
end
```

We also consider two events RegisterUser and Enter. Event RegisterUser models the situation where a new user p registers with the system. Guard @grd2 ensures that the maximum number of registered users will not exceed the limit maxsize.

```
event RegisterUser
any p where
  @grd1: p ∈ PERSON \ register
  @grd2: size ≠ maxsize
then
  @act1: register := register ∪ {p}
  @act2: size := size + 1
end
```

Event Enter models the situation where a user p enters a building b given that they have the necessary permission.

```
event Enter
any p b where
 @grd1: p ∈ register
 @grd2: b ∈ building
 @grd3: p ∉ dom(location)
 @grd4: p ↦ b ∈ permission
then
 @act1: location(p) := b
end
```

In Sect. 3.2, we will use this example to illustrate our approach to translate Event-B models to SPARK.

3 Contributions

In this section, we first discuss about the translation of the Event-B mathematical language into SPARK, then present the translation of the Event-B models.

3.1 Translation of the Mathematical Language

In term of the translation of the Event-B mathematical language into corresponding constructs in SPARK, our aim is to cover as much as possible the Event-B mathematical language. Due to the abstractness of the Event-B mathematical language, we focus on the collection of often-used constructs, including sets and relations.

Translation of Types. The built-in types in Event-B, i.e., $\mathbb{Z}$ and BOOL, are directly represented as Integer and Boolean in SPARK. Note that there is already a mismatch as Integer in SPARK are finite and bounded while $\mathbb{Z}$ is mathematical set of integers (infinite). However, any range check, i.e., to ensure that integer value are within the range from Min_Int and Max_Int, will be done in SPARK. Other basic types in Event-B are user-defined carrier sets and they will be translated as enumerated type or sub-type of Integer (see Sect. 3.2).

Translation of Sets. With the exception of BOOL and enumeration, Event-B types are often represented as sub-types of Integer in SPARK. As a result, we can represent Event-B sets as SPARK arrays of Boolean value, indexed by the Integer range.

```
type set is array (Integer range <>) of Boolean;
```

As a result, a set S containing elements of type T can be declared as

```
S : set(T);
```

Subsequently, membership in Event-B, e.g., $e \in S$ can be translated as $S(e) = True$ in SPARK.

Translation of Relations. Similar to translation of sets, we use two-dimensional SPARK arrays of Boolean values to represent relations. The two dimensional arrays are indexed by two Integer ranges corresponding to the type of the domain and range of the relations.

$$\textbf{type}\ \text{relation}\ \textbf{is array}\ (\text{Integer}\ \textbf{range} <>, \text{Integer}\ \textbf{range} <>)\ \textbf{of}\ \text{Boolean};$$

Hence, a relation $r \in S \leftrightarrow T$ (where S and T are types) can be declared as

$$\text{r} : \text{relation}(\text{S}, \text{T})$$

For a tuple $e \mapsto f$, membership of a relation r, i.e., $e \mapsto f \in r$ will be translated as $r(e, f) = True$ in SPARK.

With this approach to represent sets and relations, these Event-B constructs can thus only have carrier sets (but not enumeration) or Integer type elements, not BOOL. In the future, we will add different translation construct involving enumerations and BOOL.

Translation of Predicates. For propositional operators, such as $\neg$, $\wedge$, $\vee$, $\Rightarrow$ and $\Leftrightarrow$, the translation to SPARK is as expected. In the following, for each formula F in Event-B, let EB2SPARK(F) be the translation of F in SPARK.

- $\neg$ P is translated as not EB2SPARK(P).
- P1 $\wedge$ P2 is translated as EB2SPARK(P1) **and then** EB2SPARK(P2)
- P1 $\vee$ P2 is translated as EB2SPARK(P1) **or else** EB2SPARK(P2)
- P1 $\Rightarrow$ P2 is translated as **if** EB2SPARK(P1) **then** EB2SPARK(P2)
- P1 $\Leftrightarrow$ P2 is translated as **if** EB2SPARK(P1) **then** EB2SPARK(P2) **else** (not EB2SPARK(P2))

For quantifiers, i.e., $\forall$ and $\exists$, we need to extract the type of the bound variable accordingly, i.e.,

- $\forall z \cdot P$ is translated as **for all** z **in** z_type = > EB2SPARK(P)
- $\exists z \cdot P$ is translated as **for some** z **in** z_type = > EB2SPARK(P)

Translation of Relational Operators. For relational operators such as $\subseteq$, $\subset$, etc., there are no direct corresponding construct in SPARK. We can translate according to their mathematical definition. For example $S \subseteq T$ can be translated as

$$\textbf{for all}\ \text{x}\ \textbf{in}\ \text{S'}\textbf{Range} = > (\textbf{if}\ \text{S(x)}\ \textbf{then}\ \text{x}\ \textbf{in}\ \text{T'}\textbf{Range and then}\ \text{T(x)})$$

(Note that S and T are translated as Boolean arrays in SPARK). To improve the translation process, we define a utility function isSubset as follows.

```
function isSubset (s1 : set; s2 : set) return Boolean is
   (for all x in s1'Range => (if (s1(x) then x in s2'Range and then s2(x))));
```

With this function $\mathsf{S} \subseteq \mathsf{T}$ can be simply translated as isSubset(S, T). Other relational operators are translated similarly.

The supporting definitions, e.g., definitions for sets and relations, and utility functions, are collected in a supporting SPARK package, namely sr.ads, that will be included in every generated files. The translation is described in details in [17].

3.2 Translation of Event-B Models

Each Event-B model (including the contexts and the machine) will correspond to a SPARK Ada package. We focus at the moment on the package specification. The package body, i.e., the implementation, can be generated similarly and is our future work.

```
with sr; use sr;
package m0 with SPARK_Mode is
-- code generated for m0 (including seen contexts)
end m0;
```

In particular, each Event-B event corresponds to a procedure where the guard contributes to the precondition and the action contributes to the post-condition. In the following, we describe in details the translation of the different modelling elements.

Translation of Carrier Sets. Carrier sets are types in Event-B and can be enumerated sets or deferred sets. An enumerated set S containing elements E1, ..., En in Event-B is defined as follows.

```
sets S
constants E1, ..., En
axioms
  @def-S: partition(S, {E1}, ..., {En})
```

It is straightforward to represent the enumeration in SPARK as follows.

```
type S is (E1, ..., En);
```

A deferred set in Event-B will be represented as an Integer subtype in SPARK. As a result, we require that the deferred set in Event-B to be finite and with a specified cardinality. That is, it is declared in Event-B as follows, where n is a literal.

```
sets S
axioms
  @finite-S: finite(S)
  @card-S: card(S) = n
```

In fact, a carrier set in Event-B provides two concept: a user-defined type and a set containing all elements of that type. As a result, there are two different SPARK elements that are generated:

– A type declaration S_type.
– A variable S corresponding to the set which a Boolean array containing True indicating that set contains all elements of S_type.

```
subtype S_type is Integer range 1 .. n;
S : set(S_type) := (others => True);
```

Example 1 (Translation of Carrier Sets). The carrier sets PEOPLE and BUILDING in the example from Sect. 2.3 are translated as follows.

```
subtype PEOPLE_type is integer range 1 .. 10;
PEOPLE : set (PEOPLE_type) := (others => True);
subtype BUILDING_type is integer range 1 .. 4;
BUILDING : set (BUILDING_type) := (others => True);
```

Translation of Constants. Event-B constants are translated constant variables in SPARK. Since constant variable declarations in SPARK require that the variable be defined with a value, an axiom defining the value of the constant is also required. As a result, only constants with axioms specifying their values are translated. For example, the following constant C is specified in Event-B as follows, where n is a integer literal.

```
constants C
axioms
  @def−C : C = n
```

The specification is translated into SPARK as follows.

```
C : constant Integer := n;
```

Example 2 (Translation of Constants). The constant maxsize in the example from Sect. 2.3 is translated as follows.

```
maxsize : constant Integer := 3;
```

Translation of Axioms. For each Event-B axiom, an expression function is generated. The name of the function is the axiom label and the predicate is translated according to Sect. 3.1. At the moment, we do not generate SPARK function for axioms about finiteness: all variables in SPARK are finite. We also do not generate SPARK function for axioms about cardinality: they are non-trivial to specify and reason about with arrays. For convenience, we also generate an expression function represent all axioms of the model; we call this expression function **Axioms**. We also include in this **Axioms** constraints about carrier sets, that is they contain all elements of the types.

Example 3. Translation of Axioms. The translation of axioms for the example in Sect. 2.3 is as follows

```
function def_maxsize return Boolean is (maxsize = 3);
function Axioms return Boolean is (
 isFullSet(PEOPLE) and then
 isFullSet(BUILDING) and then
 def_maxsize);
```

Here isFullSet is a function defined in sr.ads, ensuring that PEOPLE and BUILDING are arrays containing only True.

Translation of Variables. Each variable in Event-B corresponds to a variable in SPARK. For the variable declaration in SPARK, we need to extract the type of the Event-B variable. At the moment, we support variable types of either basic types (T), set of basic types ($\mathbb{P}$(T)), and relations between basic types ($\mathbb{P}$(T1 $\times$ T2)).

Example 4. Translation of Variables The translation of the variables for the example in Sect. 2.3 is as follows.

```
register : set (PEOPLE_type);
size : Integer;
location : relation (PEOPLE_type, BUILDING_type);
permission: relation (PEOPLE_type, BUILDING_type);
```

Translation of Invariants. Each invariant corresponds to an expression function (similar to axioms) and these invariant functions are used as pre- and post-conditions of every procedures. For convenience, we define an expression function, namely **Invariants** as the conjunction of all invariants.

Example 5 (Translation of Invariants). The translation of the invariants of the example in Sect. 2.3 is as follows.

```
function inv1 return Boolean is (isSubset(register, PEOPLE));

function inv2 return Boolean is (size <= maxsize);

function inv3 return Boolean is
 isPartialFunction(location, register, BUILDING);
```

```
function inv4 return Boolean
 is isRelation(permission, register, BUILDING);

function inv5 return Boolean is isSubset(location, permission);

function Invariants return Boolean is (
 inv1 and then
 inv2 and then
 inv3 and then
 inv4 and then
 inv5
)
```

Translation of Events. For each Event-B event, a procedure of the same name is generated. The Event-B event parameters corresponding to the SPARK procedure input parameters. The other aspects of the specification, i.e., Global, Depends, Pre and Post are computed accordingly. The following Event-B event

```
event e
any p where
 ...
then
 ...
end
```

is translated into a SPARK procedure with the following structure.

```
procedure e(p : in p_type) with
  Pre => Axioms and then Invariants and then event guards
  Post => Axioms and then Invariants and then event actions
  Global => Computed from the event actions,
  Depends => Computed from the event actions,
```

First of all, the Pre and the Post aspects contain both the axioms and invariants. Since SPARK does not provide notation for invariants, we just make the assertions in the pre- and post-conditions of all procedures (except for the one corresponding to the INITIALISATION, where assertions only appear in the post-condition). The translation of guards are the translation of the individual guard predicate as described in Sect. 3.1. For each action the corresponding SPARK post-condition is generated as follows.

- $v := E(p, v)$ is translated as $v = E(p, v\text{'Old})$
- $v :\in E(p, v)$ is translated as $\text{isMember}(v, E(p, v\text{'Old}))$
- $v :| E(p, v, v')$ is translated as $E(p, v\text{'Old}, v)$

Global aspect specifies the access to the global variables and it could be **In** (for variables that are read), **Out** (variables that are updated but not read), **In_Out** (for variables that are both read and updated) or **Proof_In** (variables that only used in Precondition, i.e., for proving). We generate variables **In**, **Out** or **In_Out** based on how they are used in the event actions. Any other variables will be **Proof_In** as the preconditions references all variables (since they include all axioms and invariants).

Depends aspect specifies the dependency between the Output variables and the Input variables. We generate the Depends aspects by inspecting individual assignment. Each individual assignment corresponds to an Depends aspects clause, where the left-hand side of the clause is the variable on the left-hand size of the assignment, and the right-hand size of the clause are all variables on the right-hand size of the assignment.

Example 6 (Translation of the INITIALISATION event). The INITIALISATION event in the example from Sect. 2.3 is translated as follows.

```
procedure INITIALISATION with
 Pre => Axioms,
 Post =>
  Axioms and then
  Invariants and then
  isEmpty(register) and then
  size = 0 and then
  isEmpty(location) and then
  isEmpty(permission),
 Global => (
  Out => (register, size, location, permission),
  Proof_In => (PEOPLE, BUILDING, maxsize)
 )
 Depends => (
  register => null,
  size => null,
  location => null,
  permission => null
 )
end INITIALISATION;
```

Example 7 (Translation of the Enter event). The Enter event in the example from Sect. 2.3 is translated as follows.

```
procedure Enter(p : in PEOPLE_type, b : in BUILDING_type) with
 Pre =>
  Axioms and then
  Invariants and then
  register(p) and then
```

```
    BUILDING(b) and then
    not (inDomain(p, location)) and then
    permission(p, b),
   Post =>
    Axioms and then
    Invariants and then
    (for all x in location'Range(1) =>
      if x /= p then (for all y in location'Range(2) =>
                        location(x, y) = location'Old(x,y))
      else (for all y in location'Range(2) =>
                      if y /= b then not location(x, y)
                      else location(x, y))
    ),
   Global => (
    In_Out => (location),
    Proof_In => (PEOPLE, BUILDING, maxsize, register, size, permission)
   )
   Depends => (
    location => (location, p, b),
   )
  end Enter;
```

Here the effect of updating a function is specified using universal quantifiers to ensure that only the location of person `p` is updated to be `b`.

4 Discussion

A prototype plug-in was developed for the Rodin platform [2]. The plug-in provide a context menu for Event-B machine to translate the machine to SPARK specification package. Since the Event-B to SPARK translator requires information such as types of variables, etc., the plug-in looks at the statically checked version of the machine then generate the SPARK specification according to the translation described in Sect. 3.

Beside the example of building access control system, we also generate SPARK code from other models, such as a room booking system, a club management system [10], controlling car on a bridge [1]. Note that the plug-in only generate the specification of the package at the moment. We have manually written the package body according to the Event-B model and prove that the model is consistent. More details about these examples can be found in [17].

4.1 Code Scheduling

At the moment, we only generate the SPARK code corresponding to individual events. Combination of these events according to some scheduling rules, such as [13] or some user-defined schedule, such as [8] will be our future work. To investigate the possibility, we also applied our approach to generate SPARK code for developing a lift system (the example used in [16]) and manually wrote the scheduled code in SPARK. The code corresponds to the Event-B model including events for controlling the door of the lift, controlling the lift motor, and changing the direction of travel. Some events relevant for our scheduling example are as follows.

– DoorClosed2Half_Up: to open the door from Closed to Half-closed while the lift travel upwards,
– MotorWinds: to wind the lift motor,
– ChangeDirectionDown_CurrentFloor: to change the lift travel direction to downward due to a request at the current floor to go down.
– ChangeDirectionDown_BelowFloor: to change the lift travel direction to downward due to a request below the current floor.

Our manually written scheduled code are as follows

```
if motor = STOPPED then
  case door is
    when CLOSED =>
      if direction = UP then
        if hasRequest_Up
        then
          DoorClosed2Half_Up;
        else
          if
            floorRequestAbove or
            upRequestAbove or
            downRequestAbove
          then
            MotorWinds;
          else
            if floor /= 0 and then down_buttons_array(floor) = TRUE then
              ChangesDirectionDown_CurrentFloor;
            elsif
              floorRequestBelow or
              upRequestBelow or
              downRequestBelow
            then
              ChangesDirectionDown_BelowFloor;
            end if;
          end if;
        end if;
      else -- direction = Down
        ...
      end if;
    when OPEN => ...
```

```
      when HALF => ...
    end case;
  else -- motor /= STOPPED
    ...
  end if;
```

In the above, hasRequest_Up, floorRequestAbove, upRequestAbove, etc. are local variables capturing the different requests for the lift. The manually written code invokes the different procedures generated from the Event-B model, e.g., MotorWinds, DoorClosed2Half_Up, ChangesDirectionDown_CurrentFloor, and ChangesDirectionDown_BelowFloor. SPARK generates verification conditions to ensure the correctness of our schedule, e.g., the preconditions of the procedures are met when the they are invoked. We plan to utilise the framework from [8] to allow users to specify the schedule and generate the SPARK scheduling code accordingly. The elevator model and the manually written SPARK code are available from https://doi.org/10.5258/SOTON/D1554.

4.2 Record Data Structures

At the moment, our main data structures for the generated SPARK code is Boolean arrays (one-dimensional arrays for sets and two-dimensional arrays for relations). Some modelling elements are better grouped and represented as record data structures in the code. To investigate the idea, we extend the lift example to a MULTI-lift system. The example is inspired by an actual lift system [18]. The systems allows multiple cabins running on a single shaft system vertically and horizontally. In our formal model, we have variables modelling the status of the different cabins in the lift system, e.g., the floor position (cabins_floor), the cabin motor status (cabins_motor), the door status (cabins_door), the current shaft of the cabin (cabins_shaft), and the cabin floor buttons (cabins_floor_buttons). The types of the variables are as follows.

```
invariants
  @typeof−cabins_floor: cabins_floor ∈ CABIN → 0 .. TOP_FLOOR
  @typeof−cabins_motor: cabins_motor ∈ CABIN → MOTOR
  @typeof−cabins_door: cabins_door ∈ CABIN → DOOR
  @typeof−cabins_shaft: cabins_shaft ∈ cabins → SHAFT
  @typeof−floor_buttons: floor_buttons ∈ cabins → ℙ(0 .. TOP_FLOOR)
```

With our current approach, the variables will be translated individually as Boolean arrays. It is more natural to use a SPARK record to represent the cabin status. For example, the following CABIN_Type record can be used to capture the different attributes of a cabin.

```
type CABIN_Type is record
  floor : Integer; -- The current floor of the cabin
  motor : MOTOR_Type; -- The current status of the cabin motor
  door : DOOR_Type;  -- The current status of the door
```

```
    shaft : SHAFT_Type; -- The current shaft of the cabin

    -- The current floor buttons status inside the cabin
    floor_buttons : array (Integer range 0 .. TOP_FLOOR) of Boolean;
  end record;
```

Recognising the record data structures from the Event-B model is one of our future research directions.

5 Related Work

Generating SPARK code from Event-B models has been considered in [13]. Their approach involves not only generating pre- and post-conditions, along with loop invariants, but also generates implementing SPARK code from Event-B models, using the merging rules described by [1], which describe how to generate *sequential programs from Event-B models*. However, the model used in [13] is fairly concrete, in particular in terms of the data structure used in the model. We aim to derive proof annotations from models where mathematically abstract concepts such as sets and relations are used. Given this, the merging rules used in [13] may not be applicable to very abstract models, as such an algorithm may not be represented or derivable. Furthermore, merging rules [1] can only be applied to model with a certain structure where the scheduling is implicitly encoded in the event guards. In our paper, we focus on the translation of the data structure. Furthermore, the translation rules from Event-B to SPARK assertions shown in [13] are limited, particularly in terms of set-theoretical constructs. This is an issue to address given Event-B is a set-theory-focused modelling tool.

Generating proof annotations from Event-B models has been investigated in [8]. Their work explores the mapping between Event-B and Dafny [12] constructs. This paper claims that a "direct mapping between the two is not straight forward". Due to the increased richness of the Event-B notation compared to Dafny, only a subset of Event-B constructs can be translated. Similar to [13], the authors of [8] suggest that a particular level of refinement must be achieved by the Event-B model, to reduce "the syntactic gap between Event-B and Dafny". However, the level of refinement required is needed to have a model containing only those mathematical constructs which have a counterpart in Dafny, not to obtain a model with a clear algorithmic structure present in its events. As such, this approach can still translate fairly abstract models. Their paper states the assumption 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". Their approach uses Hoare logic [11], by transforming events into Hoare triples, and deriving the relevant pre- and post-conditions.

Translation of Event-B models into Dafny is also the scope of [7]. The Dafny code generated is then verified using the verification tools available to Dafny. The translation is done so that Dafny code is "correct if and only if the Event-B refinement-based proof obligations hold". In other words, their approach allows users to verify the correctness of their models using a powerful verification tool. Specifically, their paper focuses on refinement proof obligations, showing that the concrete model is a correct refinement of the abstract model. While this is outside of the scope of our paper, it nevertheless introduces some translation rules which are relevant for us. For example, their paper demonstrates how invariants may be translated into Dafny and used in pre-conditions. It also shows an example of how relations in Event-B may be modelled in Dafny.

Another approach explored is the translation of Event-B to JML-annotated Java programs [14], which provides a translation "through syntactic rules". JML provides specifications which Java programs must adhere to, and so it is similar to contracts. Their approach generates Java code as well as JML specifications. Unlike the previous approaches, instead of grouping similar events, every single event is translated independently. This is perhaps not as efficient, as grouping similar events and using specific case guards in the post-conditions to differentiate between the expected outcomes can give an insight into how these events interact. Additionally, event grouping also saves space in the generated code by having fewer methods. This is only foreseen to be a problem when the translated model is concrete, and has several events representing different situational implementations of a single abstract event. Their paper demonstrates translation rules of machines and events to JML-annotated programs. The approach of deriving the JML specifications can possibly be adapted for our purpose, and can perhaps be considered an alternative approach to the one by [8]. However, an interesting point to note from their paper is that the approach given has the ability to generate code even from abstract models. The translation rules given can generate code from variables and assignments to variables in actions, in any level of abstraction or refinement. Hence, this approach of generating code can possibly be adapted for the generation of SPARK code.

6 Conclusion

In summary, we present in this paper an approach to generate SPARK code from Event-B models. We focus on covering as much as possible the Event-B mathematical language by representing sets and relations as Boolean arrays in SPARK. Each Event-B event is translated into a SPARK procedure with pre- and post-conditions, and aspects for flow analysis (i.e., Global and Depends aspects). Axiom and invariance properties of the models are translated into SPARK expression functions and are asserted as both pre- and post-conditions for the generated SPARK procedures. A prototype plug-in for the Rodin platform is developed and evaluated on different examples. We discuss the possible improvement of the approach including generating code corresponding to some schedule and using record data structure.

In term of translating sets and relations, we have also considered different approaches including using *functional sets* and *formal ordered sets* [3]. Our experiment shows that these representations have limited support for set and relational operators and did not work well with the SPARK provers.

For future work, we plan to include the generation of the procedure body with our prototype. The generation will base on the representation of sets and relations by Boolean arrays. We expect that this extension will be straightforward. As mentioned earlier, generating SPARK record data structures from Event-B models is another research direction. The challenge here is to recognise the elements in the Event-B models corresponding to records. With the introduction of records in Event-B [9], the mapping from Event-B elements to record data structures will become straightforward. Furthermore, we aim to develop a development process that starts from modelling at the system level using Event-B, gradually develop the system via refinement and generate SPARK code including event scheduling and data structure such as records.

During system development by refinement in Event-B, abstract variables can be replaced (data refined) by concrete variables. This allows (mathematically) abstract concepts to be replaced by concrete implementation details. Often, systems properties are expressed using abstract variables and are maintained by refinement. In this sense, abstract variables are similar to ghost variables in SPARK. We plan to investigate the translation of abstract variables in Event-B as ghost variables in SPARK.

Models in Event-B are typically system models, that is they contain not only the details about the software system but also the model of the environment where the software system operates. Using decomposition [15], the part of the model corresponding to the software systems can be extracted. Nevertheless, having a "logical" model of the environment will also be useful and it can be represented again using ghost code in SPARK.

Acknowledgements. Supporting material for this study is openly available from the University of Southampton repository at https://doi.org/10.5258/SOTON/D1554.

References

1. Abrial, J.-R.: Modeling in Event-B: System and Software Engineering. Cambridge University Press, Cambridge (2010)
2. Abrial, J.-R., Butler, M., Hallerstede, S., Hoang, T.S., Mehta, F., Voisin, L.: Rodin: an open toolset for modelling and reasoning in Event-B. Softw. Tools Technol. Transf. **12**(6), 447–466 (2010)
3. AdaCore. GNAT Reference Manual, 21.0w edition, July 2020. http://docs.adacore.com/live/wave/gnat_rm/html/gnat_rm/gnat_rm.html
4. Barnes, J.: High Integrity Software: The SPARK Approach to Safety and Security. Addison Wesley, Boston (2003)
5. Booch, G., Bryan, D.: Software Engineering with ADA, 3rd edn. Addison-Wesley, Boston (1993)
6. Butler, M.: Reasoned modelling with Event-B. In: Bowen, J., Liu, Z., Zhang, Z. (eds.) SETSS 2016. LNCS, vol. 10215, pp. 51–109. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-56841-6_3
7. Cataño, N., Leino, K.R.M., Rivera, V.: The EventB2Dafny rodin plug-in. In: Garbervetsky, D., Kim, S. (eds.) Proceedings of the 2nd International Workshop on Developing Tools as Plug-Ins, TOPI 2012, Zurich, Switzerland, 3 June 2012, pp. 49–54. IEEE Computer Society (2012)

8. Dalvandi, M., Butler, M., Rezazadeh, A., Salehi Fathabadi, A.: Verifiable code generation from scheduled Event-B models. In: Butler, M., Raschke, A., Hoang, T.S., Reichl, K. (eds.) ABZ 2018. LNCS, vol. 10817, pp. 234–248. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-91271-4_16
9. Fathabadi, A.S., Snook, C., Hoang, T.S., Dghaym, D., Butler, M.: Extensible data structures in Event-B (submitted to iFM2020)
10. Hoang, T.: Appendix A: an introduction to the Event-B modelling method. In: Romanovsky, A., Thomas, M. (eds.) Industrial Deployment of System Engineering Methods, pp. 211–236. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-33170-1_1
11. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12**(10), 576–580 (1969)
12. Rustan, K., Leino, M.: Developing verified programs with Dafny. In: Joshi, R., Müller, P., Podelski, A. (eds.) VSTTE 2012. LNCS, vol. 7152, p. 82. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-27705-4_7
13. Murali, R., Ireland, A.: E-SPARK: automated generation of provably correct code from formally verified designs. Electron. Commun. EASST **53**, 1–15 (2012)
14. Rivera, V., Cataño, N.: Translating Event-B to JML-specified Java programs. In: Cho, Y., Shin, S.Y., Kim, S.-W., Hung, C.-C., Hong, J. (eds.) Symposium on Applied Computing, SAC 2014, Gyeongju, Republic of Korea, 24–28 March 2014, pp. 1264–1271. ACM (2014)
15. Silva, R., Pascal, C., Hoang, T.S., Butler, M.J.: Decomposition tool for Event-B. Softw. Pract. Exp. **41**(2), 199–208 (2011)
16. Snook, C., et al.: Behaviour-driven formal model development. In: Sun, J., Sun, M. (eds.) ICFEM 2018. LNCS, vol. 11232, pp. 21–36. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-02450-5_2
17. Sritharan, S.: Automated translation of Event-B models to SPARK proof annotations. Technical report, University of Southampton (2020). https://eprints.soton.ac.uk/444034/
18. thyssenkrupp: MULTI - a new era of mobility in buildings. https://www.thyssenkrupp-elevator.com/uk/products/multi/. Accessed July 2020