Electronic Communications of the EASST
Volume 53 (2012)

Proceedings of the 12th International Workshop on Automated Verification of Critical Systems (AVoCS 2012)

E-SPARK: Automated Generation of Provably Correct Code from Formally Verified Designs

Rajiv Murali and Andrew Ireland

15 pages

Guest Editors: Gerald Lüttgen, Stephan Merz
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

E-SPARK: Automated Generation of Provably Correct Code from Formally Verified Designs

Rajiv Murali[1] **and Andrew Ireland**[2]

[1] Rm339@hw.ac.uk [2] A.Ireland@hw.ac.uk
School of Mathematical and Computer Sciences,
Heriot-Watt University,
Edinburgh, EH14 4AS, UK.

Abstract: An approach to generating provably correct sequential code from formally developed algorithmic designs is presented. Given an algorithm modelled in the Event-B formalism, we automatically translate the design into the SPARK programming language. Our translation builds upon Abrial's approach to the development of sequential programs from Event-B models. However, as well as generating code, our approach also automatically generates code level specifications, i.e. SPARK pre- and post-conditions, along with loop invariants. In terms of the SPARK proof tools, having the loop invariants increases verification automation. A prototype, known as E-SPARK, has been implemented as a plugin for the Rodin Platform (Event-B toolkit), and tested on a range of examples, i.e. searching, sorting and numeric calculations.

Keywords: Event-B, Rodin, SPARK, Automatic Code Generation

1 Introduction

In terms of verification, critical software systems demand strong guarantees which formal methods can deliver via mathematical proof. However, without a high degree of proof automation the accessibility of a formal method will be limited. Here we consider proof automation within the context of *correctness-by-construction* (CxC) – software development where formal methods are used throughout the development process [JOW06].

Specifically we focus on the Event-B formalism [Abr10] and the SPARK Approach [Bar03] to developing critical software systems. Event-B promotes a refinement style of formally modelling, where verification involves proving properties of a design, as well as verifying the correctness of each refinement step. SPARK is a programming language derived from Ada, which is supported by a range of static analysis tools, including formal verification.

The approach we present here automatically generates provably correct sequential code, i.e. SPARK, from formally developed algorithmic designs modelled in the Event-B formalism. The translation builds upon Abrial's method for developing sequential programs from the Event-B models [Abr10]. However, as well as generating SPARK code, our approach also automatically generates code level specifications, i.e. SPARK pre- and post-conditions, along with loop invariants. A prototype of our approach, known as E-SPARK, has been implemented as a plugin for the Rodin Platform (Event-B toolkit).

As well as the productivity gains of automated code generation, a key benefit of our approach is that the specification effort at the design level (Event-B) is automatically reused at the code level (SPARK) in order to increase proof automation, e.g. without loop invariants, a significant level of proofs will fail.

The contribution of the paper are three fold:

- A mechanisation of Abrial's method for the SPARK programming language – implemented as a Rodin plugin.
- A translation from Event-B formalism into the SPARK proof annotation language.
- Initial experimental results, i.e. searching, sorting and numeric calculations.

The remainder of the paper is structured as follows. In Section 2, we present background material on SPARK and Event-B. This sets the scene for Section 3 where we describe the core of our approach. The implementation details of E-SPARK are outlined in Section 4 and the results are discussed in Section 5. Related and future work is described in Section 6 and our conclusions are given in Section 7.

2 Background

2.1 The SPARK Approach

The SPARK programming language [Bar03] was designed for safety and security critical applications, and has been used in developing software industrial scale projects for over twenty years [Bar11].

SPARK is derived from the Ada programming language. In order to make static analysis feasible, and applications robust, SPARK excludes many Ada constructs, e.g. pointers and aliasing. Moreover, it also restricts the way in which the language can be used, e.g. functions cannot have side-effects and recursion is excluded.

In support of static analysis, SPARK also includes a language of annotations, i.e. program and proof annotations. Both allow a programmer to specify the intended behaviour of their code. In the case of *program annotations*, specifications define *information flows* which can be checked automatically against the code using the SPARK Examiner - a static analysis tool. The *proof annotations* support traditional assertion based *formal verification*, and include pre- and post-conditions as well as loop invariants. In support of formal verification, the Examiner contains a *verification condition generator* (VCG) while proof tools are also provided in order to discharge *verification conditions* (VCs), i.e. the Simplifier (automatic) and the SPADE Proof Checker (interactive). The SPARK Approach supports two kinds of formal verification:

Partial correctness: proving that whenever the pre-conditions to a subprogram hold, and assuming that the subprogram terminates, then the post-condition will hold on termination[1].

Exception freedom: proving that no exceptions will be raised at run-time, e.g. buffer overflows and arithmetic overflows.

[1] Note that SPARK does not support *total correctness*, i.e. proving termination as well as partial correctness.

In our work we have focused for now on the proof annotations and formal verification.

Finally, we present in more detail some aspects of the SPARK languages that are relevant to the presentation of our work:

- Like Ada, a SPARK program is defined in terms of a *package specification* and a *package body* – representing the interface and implementation components respectively.

- Subprograms take the form of procedures and functions – currently we only deal with procedures. The interface to a procedure is defined in terms of *imported* and *exported* variables, which are denoted by global variables or formal parameters. In the case of formal parameters, the modes `in` and `out` and used to indicate imported and exported variables respectively. For instance, an `in` parameter is read-only, while an `out` parameter is unititialized and allows the corresponding actual parameter to be updated. In order for an actual parameter to be read and updated, the corresponding formal parameter is tagged as `in out`.

- A key feature of SPARK is the notion of a *subtype*, which allows a programmer to tightly constrain their program variables. For example, the range of integers $1 \ldots 10$ can be defined as a subtype of `Integer` as follows:

```
subtype T is Integer range 1 .. 10;
```

- Annotations are special comments, i.e. while “`--`” denotes the start of a comment, in SPARK “`--#`” denotes the start of an annotation. To illustrate, consider the following pre- and post-condition style specification for a procedure that safely increments the parameter `X` of subtype `T`:

```
--# pre   X > 0 and X < 10;
--# post  X > 0 and X <= 10;
```

  In the case of loop invariants, an assertion is included at the start of a loop construct, e.g.

```
...
loop
--# assert X > 0 and X <= 10  ;
    exit when X = 10 ;
    ...
loop end;
```

  Note that while SPARK supports a range of iterative constructs, we focus on the most general, i.e. the `loop` construct.

2.2 Event-B

Event-B is a refinement based formal method that is used in the design of discrete systems. Designs are represented as *models*, which are constructed from contexts and machines. A *context* models the static aspects of a system while a *machine* models the dynamic aspects. A context contains *constants* and associated *axioms* while a machine contains *variables* and *invariants*. The variables denote the state of the system, while properties of the system state are represented by invariants. Machines also contain *events*, which define the state transitions associated with the system. The most basic form of an event is given below:

$$< identifier > \triangleq \textbf{when} < guards > \textbf{then} < actions > \textbf{end}$$

The applicability of an event is determined by the *guards*. The *actions* take the form of assignments that update the state variables and are performed simultaneously. Where an event represents a refinement of another event, the keyword `refines` is used to indicate this relationship. In terms of formal verification, *proof obligations* (POs) are generated to ensure the correctness of each refinement step, as well as the internal correctness of each model, e.g. all events must preserve the invariants.

In our work we also make use of the `status` of an event, which is optional and by default is set to `ordinary`. An event can also have status `anticipated` or `convergent` – which relates to the termination of an event, i.e. the event will not always be applicable. If tagged as `anticipated`, then the developer is indicating their intention to refine an event so that it will terminate, where the subsequent terminating event is tagged as `convergent`. Both events require a *variant*, and in the case of a convergent event, a termination PO is generated.

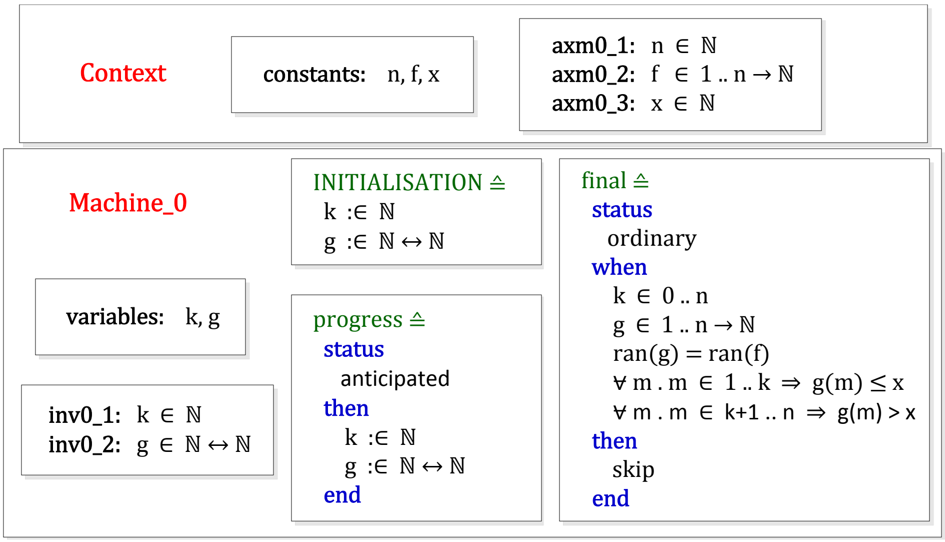
The Rodin Platform [ABHV06] is an industrial-strength tool-set that supports Event-B formalism. Rodin is Eclipse-based, so is extensible and provided significant support in the development of our E-SPARK plug-ins.

As mentioned above, we have focused on the development of sequential programs, following Abrial's method. Below we highlight the three phases of Abrial's method, drawing upon the Array Partition example given in Figures 1 and 2:

Specification Phase: At the start of a development process, we will have an *initialization* event, potentially some anticipated events, e.g. *progress*, but crucially the initial machine will contain a guarded event that specifies the post-condition of the algorithm. In the case of the Array Partition example, the event *final* plays this role (see Figure 1).

Development Phase: Events are incrementally introduced until the complete algorithm is represented. In the Array Partition example only one refinement step is required, which introduces the events *progress_1*, *progress_2* and *progress_3* . (see Figure 2).

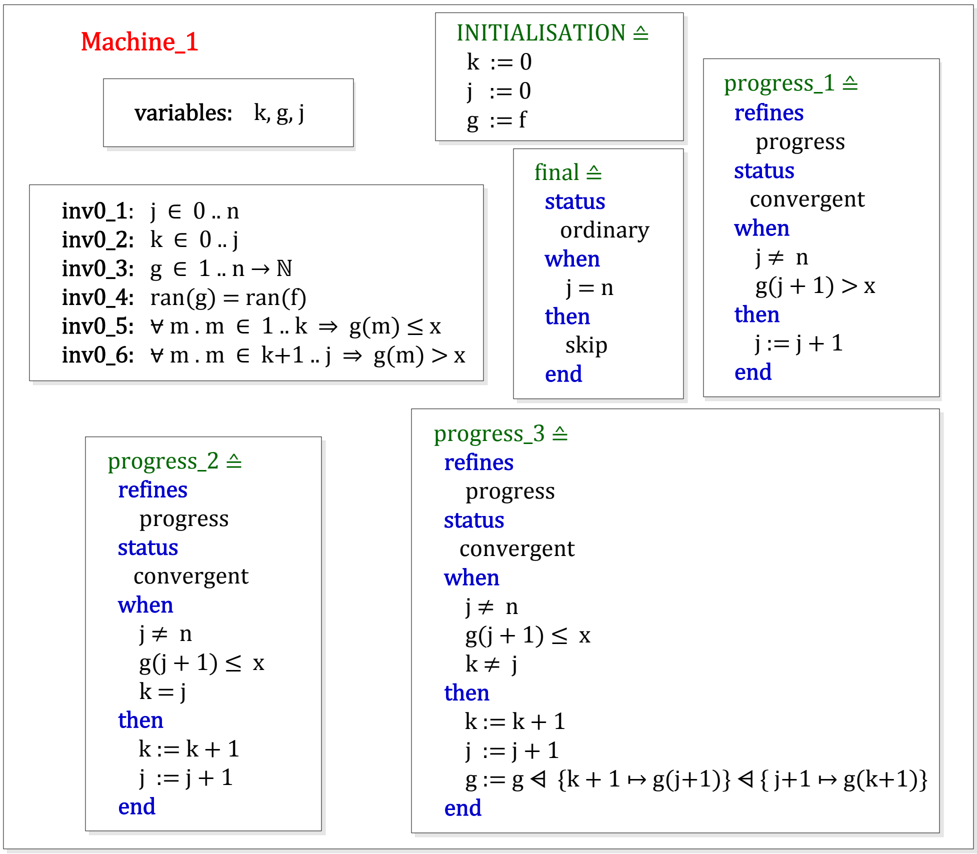
Merging Phase: The final phase involves merging the events, within the most concrete machine, into imperative code. This process is underpinned by Abrial's *merging rules*, i.e. M_IF, M_WHILE, M_ELSIF and M_INIT. We delay the presentation of the rules until Section 3, where we show how they have been tailored for the generation of SPARK code.



```
Context

constants:   n, f, x

axm0_1:  n ∈ ℕ
axm0_2:  f ∈ 1 .. n → ℕ
axm0_3:  x ∈ ℕ
```

```
Machine_0

variables:   k, g

inv0_1:  k ∈ ℕ
inv0_2:  g ∈ ℕ ↔ ℕ

INITIALISATION ≙
  k :∈ ℕ
  g :∈ ℕ ↔ ℕ

progress ≙
  status
    anticipated
  then
    k :∈ ℕ
    g :∈ ℕ ↔ ℕ
  end

final ≙
  status
    ordinary
  when
    k ∈ 0 .. n
    g ∈ 1 .. n → ℕ
    ran(g) = ran(f)
    ∀ m . m ∈ 1 .. k ⇒ g(m) ≤ x
    ∀ m . m ∈ k+1 .. n ⇒ g(m) > x
  then
    skip
  end
```

Figure 1: Array Partition: Event-B Initial Model



```
Machine_1

variables:   k, g, j

INITIALISATION ≙
  k := 0
  j := 0
  g := f

inv0_1:  j ∈ 0 .. n
inv0_2:  k ∈ 0 .. j
inv0_3:  g ∈ 1 .. n → ℕ
inv0_4:  ran(g) = ran(f)
inv0_5:  ∀ m . m ∈ 1 .. k ⇒ g(m) ≤ x
inv0_6:  ∀ m . m ∈ k+1 .. j ⇒ g(m) > x

final ≙
  status
    ordinary
  when
    j = n
  then
    skip
  end

progress_1 ≙
  refines
    progress
  status
    convergent
  when
    j ≠ n
    g(j + 1) > x
  then
    j := j + 1
  end

progress_2 ≙
  refines
    progress
  status
    convergent
  when
    j ≠ n
    g(j + 1) ≤ x
    k = j
  then
    k := k + 1
    j := j + 1
  end

progress_3 ≙
  refines
    progress
  status
    convergent
  when
    j ≠ n
    g(j + 1) ≤ x
    k ≠ j
  then
    k := k + 1
    j := j + 1
    g := g ⩤ {k + 1 ↦ g(j+1)} ⩤ { j+1 ↦ g(k+1)}
  end
```

Figure 2: Array Partition: Event-B Final Refinement

3 Translation: Event-B to SPARK

As discussed in Section 1 our aim is to produce provably correct SPARK code from Event-B models. The Array Partition model shown in Figures 1 and 2 is translated to the SPARK code seen in Listings 1 and 2. This translation is discussed in the following subsections, with the context translation in Subsection 3.2 and machine translation in Subsection 3.3.

Listing 1: Array Partition: SPARK Package Specification

```
package ArrayPartition
is
   --Note To User: Please reassign required value for constant n.
   n: constant Natural:= 10;

   subtype f_pointer_Type is Integer range 1 .. n;
   Type f_Array_Type is array (f_pointer_Type) of Natural;
   subtype j_Type is Integer range 0 .. n;

   procedure ArrayPartition_Procedure(x: in Natural; f: in f_Array_Type; g:
       out f_Array_Type; k: out j_Type);
   --# post (For all m in f_pointer_Type range 1..k => (g(m) <= x))
   --#  and (For all m in f_pointer_Type range k+1..n => (g(m) > x));

end ArrayPartition;
```

Listing 2: Array Partition: SPARK Package Body

```
package body ArrayPartition is
   procedure ArrayPartition_Procedure(x: in Natural; f: in f_Array_Type; g:
       out f_Array_Type; k: out j_Type)
   is
      j: j_Type;
      swap_g: Natural;
   begin
      k := 0; j := 0; g := f;  --INITIALISATION Event
      loop
         --# assert (For all m in f_pointer_Type range 1..k => (g(m) <= x))
         --#    and (For all m in f_pointer_Type range k+1..j => (g(m) > x))
         --#    and k>=0 and k<=j and j>=0 and j<=n;
         exit when not (j /= n); --Final Event
         if g(j+1) > x then   --Progress_1 Event
            j := j + 1;
         elsif k = j then   --Progress_2 Event
            k := k + 1; j := j + 1;
         else      --Progress_3 Event
            swap_g := g(j+1);
            g(j+1) := g(k+1);
            g(k+1) := swap_g;
            k := k + 1; j := j + 1;
         end if;
      end loop;
   end ArrayPartition_Procedure;
end ArrayPartition;
```

3.1 Basic Translations

Below we define some basic elements of our translation, which relate to both the context and machine components of a model.

Definition 1 (Translating Enumeration Sets) Enumerated sets are defined within the *context* of a model. An enumerated set S, with elements $E_1, E_2, \ldots, E_n$, is translated into the SPARK enumeration type as follows:

$$S = \{E_1, E_2, \ldots E_n\} \rightsquigarrow \textbf{type}\ \texttt{S\_type}\ \textbf{is}\ (E_1, E_2, \ldots, E_n)$$

Note that the subtype is translated as a global declaration that appears within the package specification. Note also that the name given to the enumerated type, i.e. `S_type`, is derived from the identifier associated with the enumerated set.

Definition 2 (Translating Set Membership) In Event-B the type of a constant or a variable is given via set membership, and may take the form of an axiom or an invariant. We translate the predefined sets $\mathbb{N}$, $\mathbb{Z}$ and $BOOL$ directly into the SPARK types `Integer`, `Natural` and `Boolean` respectively. Note that if the type corresponds to an enumeration set then Definition (1) is applicable.

Definition 3 (Translating Intervals) In Event-B, an *interval* takes the form $a..b$, where a and b denote the lower and upper bounds of the interval respectively. Intervals can be used to define constants and variables, as well as function domains (see Definition 4). Our translation of an interval makes use of the subtype, i.e.

$$a..b \rightsquigarrow \textbf{subtype}\ \texttt{n\_Type}\ \textbf{is}\ \texttt{BaseType}\ \textbf{range}\ \texttt{a..b;}$$

where `BaseType` is either `Integer` or `Natural` depending upon the context. Note that the bounds a and b must be constants or literal values. This restriction is imposed by SPARK, which requires that all memory must be statically allocated.

Definition 4 (Translating Total Functions) A total function in Event-B, where the domain takes the form $1..N$, is translated into a SPARK array. Note that the translation of the function domain gives rise to a subtype (see Definition 3). For example, given a function type of the form $f \in 1..b \rightarrow \mathbb{N}$, its translation into an array type is defined as follows:

$$f \in 1..b \rightarrow \mathbb{N} \rightsquigarrow \textbf{type}\ \texttt{f\_type}\ \textbf{is array}\ \texttt{S}\ \textbf{of}\ \texttt{Natural;}$$

where `S` denotes the subtype generated for the interval $1..b$. Currently we restrict the co-domain to be $\mathbb{N}$, $\mathbb{Z}$, $BOOL$ or an enumerated set.

A key feature of our approach is that we generate code level assertions, as well as code. Table 1 summarises the translations we use in mapping Event-B formulas onto SPARK assertions.

Event-B Formulas	SPARK Assertions
$n \in x..y$	`x ≤ n and n ≤ y`
$\forall m \cdot m \in x..y \Rightarrow (< predicate >)$	`For all m in <subtype_name> range x..y ⇒ (<predicate>)`
$\exists m \cdot m \in x..y \Rightarrow (< predicate >)$	`For some m in <subtype_name> range x..y ⇒ (<predicate>)`
$z = min(f[x..y])$	`For all m in <subtype_name> range x..y ⇒ (z ≤ f(m))`
$z = max(f[x..y])$	`For all m in <subtype_name> range x..y ⇒ (z ≥ f(m))`

Table 1: Translation of Event-B formulas into SPARK assertions

3.2 Context Translation

In this Section we discuss the context translation describing it first in the code level perspective and then the assertion level perspective in Subsections 3.2.1 and 3.2.2 respectively.

3.2.1 Code level perspective

Constants defined within the context of a model are either translated into SPARK constants or imported variables with respect to the procedure that is generated via the translation of the machine component. The associated type information is derived as described above in Section 3.1.

For instance, in the Array Partition example the context contains three constants, i.e. x, f and n. The constants x and f are translated as read-only formal parameters of the procedure `ArrayPartition_Procedure` (see line 10 of Listing 1), while n becomes a constant (see line 4 of Listing 1). In terms of type declarations:

- `x:Natural` follows from axm0_1: $x \in \mathbb{N}$ (Definition 2);
- The array type `f_Array_Type` follows from axm0_2: $f \in a..b \rightarrow \mathbb{N}$ (Definition 4), and gives rise to the subtype `f_pointer_Type` (Definition 3);
- `n:constant Natural:=10;` follows from axm0_1: $n \in \mathbb{N}$ (Definition 2), but requires user input in order to establish the value `10`, as will be explained in Section 4.

3.2.2 Assertion level perspective

Apart from providing type information, axioms specify conditions that act as hypothesis in all proof obligations in Event-B. We derive pre-condition annotation of the generated procedure from axioms that are translatable to SPARK proof context, as in Table 1. In the Array Partition example, there are no axioms that are translatable to SPARK proof annotations, and so the precondition is assumed `true`.

3.3 Machine Translation

In this Section we discuss the machine translation describing the code level perspective and event scheduling in Subsection 3.3.1 and Subsection 3.3.2 respectively, and finally the assertion level perspective in Subsection 3.3.3.

3.3.1 Code level perspective

As mentioned earlier, the events associated with a machine are translated into a single SPARK procedure. Note that all variables defined within a machine are by default translated into local variables in SPARK. However, variables that appear in the SPARK post-condition annotation (discussed in Subsection 3.3.3) are automatically translated into exported variables. These variables denote the expected result(s) of running an algorithm.

In the Array Partition example, the types for the program variables `j`, `g` and `k` are derived from the invariants inv0_1, inv0_2 and inv0_3 respectively:

- Machine variable g is a total function. Given that it has the same type as axm0_1, the translation uses the existing array type `f_array_type` in the declaration of g.
- Machine variable j is defined by the interval $0..n$, and is thus translated into the global subtype declaration `j_type` (Definition 3).
- Machine variable k is defined by the interval $0..j$. Note that the upper bound j is also a variable. Because bounds must be defined in terms of either constants or literal values, the upper bound of k is derived from the type of j (upper bound).

Note that the program variables `g` and `k` occur within the post-condition of procedure Array-Partition_Procedure. As a consequence, they translated into exported variables. The remaining variable, i.e. `j` is declared locally within the procedure.

Recall that actions represent simultaneous assignments. The translation of actions may therefore introduce the need for temporary program variables. For instance, in the Array Partition example, machine variable g in event `progress_3` is simultaneously updated, and therefore gives rise to a temporary variable, i.e. `swap_g` – see line 5 of Listing 2.

3.3.2 Event Scheduling

A key aspect of the translation involves determining how the events in the final refinement are mapped onto imperative constructs i.e. if-then, if-then-else and while statements. As mentioned earlier, we use Abrial's merging rules, i.e. M_IF, M_WHILE, M_ELSIF and M_INIT, adopted for SPARK. The SPARK versions of the M_IF and M_WHILE rules are given in Table 2. Note that the application of M_IF and M_WHILE to `Event A` and `Event B` generates *pseudo-events* `mergedAB_if` and `mergedAB_while` respectively.

Abrial associates the following side conditions with rules M_IF and M_WHILE in order to disambiguate their application:

- M_IF rule: requires that Event A and Event B are introduced at the same *level* within a development.

Event A	Event B	mergedAB_if Event	mergedAB_while Event
when	**when**	**when**	**when**
P	P	P	P
Q	$\neg Q$	**then**	**then**
then	**then**	`if` Q `then`	`loop`
S	T	S`;`	`--#assert` P`;`
end	**end**	`else`	`exit when not` Q`;`
		T`;`	S`;`
		`end if;`	`end loop;`
		end	T`;`
			end

Table 2: M_IF and M_WHILE merging rules [Abr10] tailored for SPARK program constructs

- M_WHILE rule: requires that Event A (loop body) is convergent at the level below which Event B was introduced.

Note that in order to apply the merging rules, the translation needs to keep track of the levels at which the events are introduced. Note that the M_ELSIF rule is treated the same as the M_IF rule, except that it requires one of the merging events, i.e. `A` or `B`, to be a pseudo-event, generated via a application of the M_IF rule.

These three rules are applied exhaustively until a single non-guarded event remains. At this stage, the M_INIT rule is applied, merging the initialisation event with the remaining non-guarded event to form the final program.

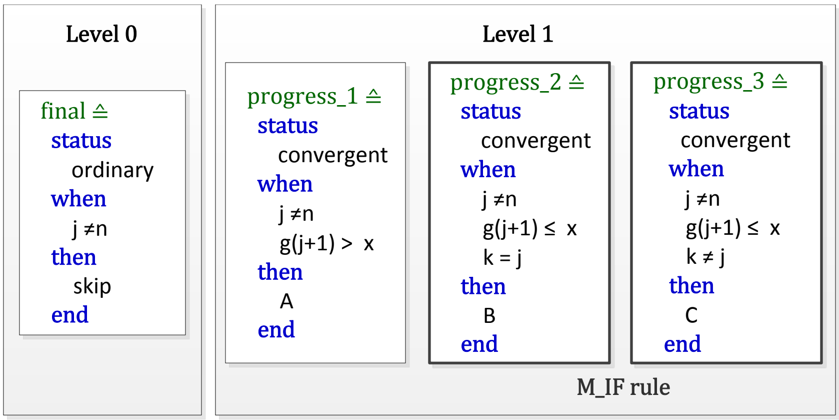


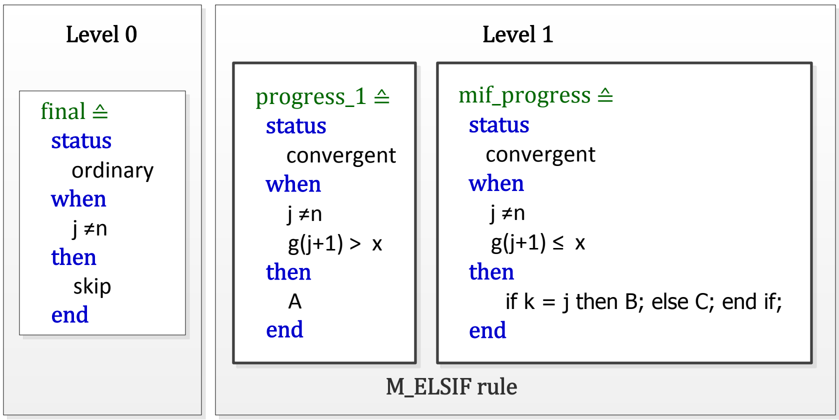
Figure 3: Concrete events in the final refinement of Array Partition Example

In the Array Partition example, the final machine is composed of `progress_1`, `progress_2`, `progress_3` and `final` (see Figure 3), excluding the initialisation event. The rules are applied in the following order M_IF, M_ELSIF, M_WHILE and finally M_INIT. The application to the Array Partition example is described below:

- M_IF Rule: Events `progress_2` and `progress_3` (see Figure 3) match the M_IF rule, where P denotes $j \neq n$ and $g(j+1) \leq x$ while Q denotes $k = j$. The side condition is satisfied as the events are both introduced at level 1. The events are merged to form the pseudo-event `mif_progress`.

  $$\texttt{mif\_progress} \triangleq \textbf{when } P \textbf{ then } \texttt{if Q then B; else C; end if;}\,\textbf{end}$$

  The merged pseudo-event `mif_progress` is introduced back into the same level (see Figure 4) as its parents. There are no more events that satisfy the application of the M_IF rule, the translation moves to the next rule M_ELSIF.



```
final ≜
  status
    ordinary
  when
    j ≠n
  then
    skip
  end
```

```
progress_1 ≜
  status
    convergent
  when
    j ≠n
    g(j+1) > x
  then
    A
  end
```

```
mif_progress ≜
  status
    convergent
  when
    j ≠n
    g(j+1) ≤ x
  then
     if k = j then B; else C; end if;
  end
```

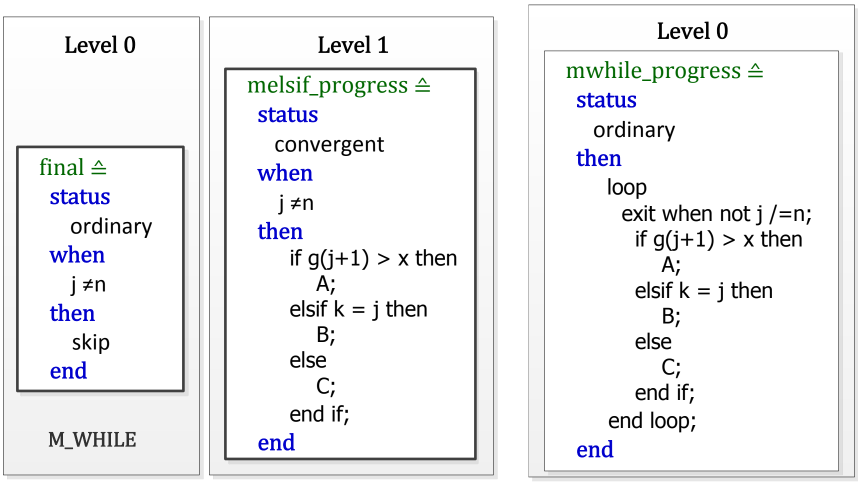
Figure 4: Remaining events after the application of the M_IF rule

- M_ELSIF Rule: The new merged pseudo-event `mif_progress` and `progress_1` match the M_ELSIF Rule, where P denotes[2] $j \neq n$ and Q denotes $g(j+1) > x$. The events are merged to form pseudo-event `melsif_progress`, which is introduced back into level of its parent events, as shown in Figure 5a. Note that in both M_IF and M_ELSIF merging rules, the status of the parent events dictates the status of the merged event. For instance, the merged event `mif_progress` has the status *convergent* of its parent events.

- M_WHILE Rule: The events `melsif_progress` and `final` are merged to produce `mwhile_progress`. Note that the events satisfy the side condition of the rule which requires the first event, `melsif_progress` (loop body) to be convergent at one refinement level below that of event `final`. Note also that there are no common guard P for the merged pseudo-events. The merged event is introduced at the level of the *non-convergent* parent event (see Figure 5b).

At this stage the M_INIT rule merges the initialisation event with the unguarded pseudo-event `mwhile_progress`, which gives the final program (see lines 7 to 21 of Listing 2).

[2] Note that we check the equivalence of formula, such as guards, semantically where matching is not possible.



```
Level 0

final ≙
  status
    ordinary
  when
    j ≠n
  then
    skip
  end

M_WHILE
```

```
Level 1

melsif_progress ≙
  status
    convergent
  when
    j ≠n
  then
      if g(j+1) > x then
        A;
      elsif k = j then
        B;
      else
        C;
      end if;
  end
```

```
Level 0

mwhile_progress ≙
  status
    ordinary
  then
      loop
        exit when not j /=n;
          if g(j+1) > x then
            A;
          elsif k = j then
            B;
          else
            C;
          end if;
      end loop;
  end
```

(a) Events left after M_ELSIF rule

(b) Event left after M_WHILE rule

Figure 5: Scheduling events using merging rules

3.3.3 Assertion level perspective

In terms of assertions, a machine provides us with the post-condition and loop invariants for an algorithm. While the post-condition is embedded within the initial machine, the loop invariants are obtained from the machine invariants given in the refined models. The translation of machine invariants into SPARK assertions are given in Table 1, while the resulting loop invariants are embedded within the corresponding `loop`-construct. In the Array Partition example, the invariants inv0_1, inv0_2, inv0_5 and inv0_6 (see Figure 2) are translated into the loop invariant shown in Listing 2 (see lines 9 to 11)

4 E-SPARK

Our mechanisation of Abrial's method, E-SPARK, is implemented as a Rodin plug-in. Being an Eclipse based platform, Rodin makes it relatively easy to access the details of an Event-B development. Once installed, the plug-in is executed via an extended user interface of the *machine file*, see Figure 6.

E-SPARK automatically accesses machine and context files for a given concrete machine. Each machine component has a *sees* and *refines* clause, which define the context and machine files that it is related to, apart from the initial model which may only have a *sees* clause. The tool uses this information to traverse through past refinements and collect all context and machine modelling elements accordingly. Note that the levels and status of events are also stored. This information is required in order to apply the merging rules, as explained in Subsection 2.2. Once the properties of the model are collected, the plug-in initiates the translation phase of the context and machine component accordingly. Context translation as discussed in Subsection 3.2

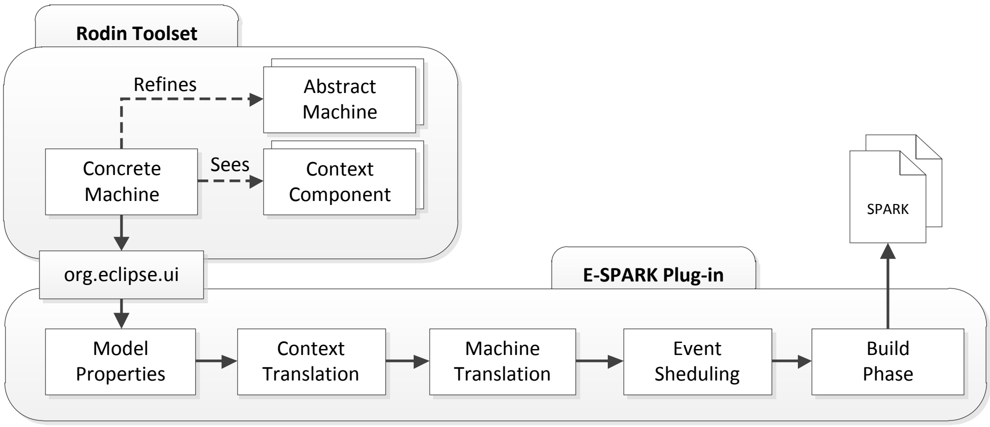


Figure 6: E-SPARK plug-in

derives the constants, their type declarations and associated pre-condition specification. Note constants in Event-B that are translated to SPARK constants have their value derived from the user during translation. The machine translation phase is similar, it derives the variables and their type declaration, including the invariant properties, as discussed in Subsection 3.3. The translation then proceeds with event scheduling, merging the events of the concrete machine to a single sequential SPARK program (procedure), as discussed in Subsection 3.3. E-SPARK also generates traceability comments during the event scheduling process – to help the user better understand the generated code. Code level specifications, i.e. SPARK pre- and post-conditions, along with loop invariants are then generated at the logical annotation phase. The final phase of translation provides a template that builds the translated components within package body and package specification files.

5 Results

The results of our initial experiments with E-SPARK are given in Table 3. We focused on searching and sorting algorithms, and numeric calculations. Given the relevant Event-B developments, E-SPARK successfully generated provably correct SPARK code for each of the examples. At the SPARK level, all the *verification conditions* (VCs) were discharged automatically using the SPARK proof tools, i.e. Simplifier and Victor. Note that in the results table we have separated the partial correctness (PC) VCs from the exception freedom (EF) VCs. Without the reuse of invariants, that E-SPARK facilitates, none of partial correctness verifications would succeed.

Event-B Model	No. of SPARK Constructs			SPARK VCs	
	if-then	if-then-else	loop	PC VCs	EF VCs
Array Partition	1	1	1	3	13
Array Reversing	1	0	1	3	7
Division Algorithm	0	0	1	3	4
Maximum Value	1	0	1	4	6
Polish Flag	1	0	1	5	10
Simple Sort	1	0	2	7	14
Summation	0	0	1	3	4

Table 3: Results of E-SPARK

6 Related Work and Future Work

A number of approaches to generating code from Event-B models have been explored. However, given that Event-B is not restricted to modelling sequential systems, most approaches have targeted concurrent code [Wri09, MS11, EB11]. In terms of the generation of sequential code, we do not know of any other mechanisation of Abrial's method, or any system that generates SPARK code from Event-B models. We are aware of work by Iliasov [Ili11] who argues for the value of certifying code generators, using the translation of Event-B to imperative code as an illustrative example – this work also relates to ClawZ and AutoCert discussed below.

In her AVoCS-11 keynote address [Bar11], Barnes reported on a number of industrial scale software development projects which combined specifications written in the Z notation [Spi89], with refinement to SPARK code. While this combination has proved very successful, there remains so called "semantic gaps" [Ros05] – what Barnes refers to as "taking small semantic steps between stages of the lifecycle" [Bar11]. Our approach has made progress towards bridging one of these gaps, i.e. mechanising the transformation between specification and code. Although this transformation has not been formally verified, its correctness is dependent upon a relatively small set of transformations from which the correctness of the generated code follows by proof.

We follow the approach taken by ClawZ [ACOS00], where one does not rely upon the correctness of the auto coder – instead a code level correctness proof is constructed each time the auto coder runs. In the case of ClawZ, the design level models are represented as Simulink diagrams and the generated code is SPARK. Event-B is a much richer modelling formalism than Simulink. Another significant example of this style of verification is the AutoCert [DF09] code analysis system developed at NASA. However, AutoCert works with requirements rather than design models, and is independent of the code generator.

The idea of proving the "correction of generated code", rather than proving the "code generator correct" has interesting implications for the safety-critical sector. That is, if one can independently verify that the generated code correctly implements design requirements, then the burden of tool qualification imposed by the safety-critical sector could be significantly reduced.

In terms of future work we have a number of plans:

- Further experimentation and testing of our E-SPARK plugin.
- Extend E-SPARK to include the generation of information flow annotations.

- Include traceability between formal model and generated code.

7 Conclusion

We have described our E-SPARK plugin, a mechanization of Abrial's method for translating sequential algorithms modelled in Event-B into imperative code. By targeting the SPARK programming language we were able to generate code level specifications from Event-B models. This reuse of Event-B specifications reduces the semantic gap between the formal modelling and the code. In addition, it gives rise to proof automation gains at the code level.

Acknowledgements

The first author was supported by an Industrial CASE studentship funded by EPSRC and BAE Systems (EP/J501992), while the second author was partially supported by EPSRC grant EP/J001058. We also thank the anonymous AVoCS-12 reviewers for their constructive feedback.

Bibliography

[ABHV06] J. R. Abrial, M. Butler, S. Hallerstede, L. Voisin. An open extensible tool environment for Event-B. 2006.

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

[ACOS00] R. Arthan, P. Caseley, C. O. O'Halloran, A. Smith. ClawZ; Control laws in Z. 2000.

[Bar03] J. G. P. Barnes. *High integrity software: the SPARK approach to safety and security*. Addison-Wesley Longman Publishing Co., Inc., 2003.

[Bar11] J. E. Barnes. Experiences in the Industrial use of Formal Methods. 2011.

[DF09] E. Denney, B. Fischer. A Verification-Driven Approach to Traceability and Documentation for Auto-Generated Mathematical Software. 2009.

[EB11] A. Edmunds, M. Butler. Tasking Event-B: An Extension to Event-B for Generating Concurrent Code. 2011.

[Ili11] A. Iliasov. Generation of certifiably correct programs from formal models. 2011.

[JOW06] C. B. Jones, P. O'Hearn, J. Woodcock. Verified Software: A Grand Challenge. 2006.

[KHCP00] S. King, J. Hammond, R. Chapman, A. Pryor. Is Proof More Cost Effective Than Testing? 2000.

[MS11] D. Méry, N. K. Singh. Automatic code generation from event-B models. 2011.

[Ros05] P. E. Ross. The exterminators [software bugs]. 2005.

[Spi89] J. M. Spivey. *The Z Notation*. 1989.

[Wri09] S. Wright. Automatic generation of C from Event-B. In *Workshop on integration of model-based formal methods and tools*. 2009.