Code Generation for Event-B

Andreas Fürst[1], Thai Son Hoang[1], David Basin[1], Krishnaji Desai[2], Naoto Sato[3], and
Kunihiko Miyazaki[3]

[1] Institute of Information Security, ETH-Zurich, Switzerland
{fuersta,htson,basin}@inf.ethz.ch
[2] Hitachi India Pvt. Ltd., India
krishnaji@hitachi.co.in
[3] Yokohama Research Lab, Hitachi Ltd., Japan
{naoto.sato.je,kunihiko.miyazaki.zt}@hitachi.com

Abstract. We present an approach to generating program code from Event-B models that is correct-by-construction. Correctness is guaranteed by the combined use of well-definedness restrictions, refinement, and assertions. By enforcing the well-definedness of the translated model, we prevent runtime errors that originate from semantic differences between the target language and Event-B, such as different interpretations of the range of integer values. Using refinement, we show that the generated code correctly implements the original Event-B model. We provide a simple yet powerful scheduling language that allows one to specify an execution sequence of the model's guarded events where assertions are used to express properties established by the event execution sequence, which are necessary for well-definedness and refinement proofs.

Keywords: Event-B, code generation, correct-by-construction.

1 Introduction

The Event-B modelling language [2] is a formal method that is well suited for developing embedded controllers that satisfy strong safety requirements. The advantage of Event-B and its notion of refinement is that we can express and prove safety properties on an abstract model of the system that includes both the controller and its working environment. Details of the system are afterwards gradually introduced into the formal models via refinement. Refinement in Event-B preserves the proved safety properties of the abstract model.

Once the system's model is sufficiently detailed, the controller part of the model can be extracted. This must afterwards be translated into a sequential program that runs on given hardware. We identify three main challenges for this translation. First, the Event-B model must be restricted to a well-defined subset in order to generate code for a particular programming language. Well-definedness for the sublanguage thereby reflects the available data types of the target language. For example, arithmetic operations that are valid in the Event-B model, but not well-defined for a target language, might result in overflows at runtime because of the different domains of the integer type. Second, Event-B's semantics is such that the event that is executed next is chosen non-deterministically from the set of enabled events. This non-determinism must be replaced by a schedule that defines an execution order on the events. As the scheduling

language becomes more sophisticated, one can generate more efficient program code. Finally, it is evident that the translation must preserve the safety properties of the Event-B model.

There has been extensive related work on code generation for Event-B [2, 4–6, 9, 10]. The different approaches have limitations including restricted scheduling languages [6, 9, 10], ignoring the differences between the mathematical notation of Event-B and the target languages [2, 4–6, 10], and missing formal justification of the approach's correctness [6, 9, 10]. More details on the limitations of the existing approaches are provided in Section 5.

To overcome these limitations, we present an approach to generating code from Event-B models that focuses on the translation's correctness. We therefore concentrate on a single target language, namely C. Furthermore, our approach provides a flexible scheduling language that is not only useful for encoding different scheduling strategies but also for proving that the specified schedules are valid, that is, they do not result in programs with behaviours that are not described by the original Event-B model.
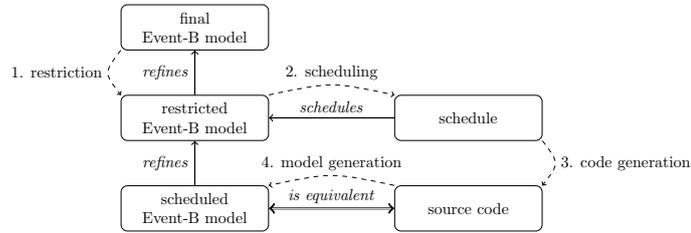

Fig. 1. Overview of our code generation approach

Our code generation approach has four steps. Figure 1 depicts the different entities involved and their relationships together with the corresponding step in which they are provided by the user or the code generator. The "final Event-B model" represents the final refinement step of an Event-B development and is the starting point for our code generation approach. First, we restrict the Event-B model via refinement to ensure that the variables are of suitable types and operations on them are well-defined. Second, we use a special scheduling language to specify a schedule for the restricted Event-B model, that describes the intended execution order on the events. Third, we execute our code generator with the schedule as input. Based on the schedule, the code generator translates the restricted model into a sequential program and thereby generates source code. Finally, our code generator also generates a scheduled Event-B model representing the semantics of the sequential program, and we prove that this scheduled model refines the restricted Event-B model.

The correctness of our translation relies on (i) the use of partial functions and well-definedness to ensure that the operations on the data types provided by the target language are valid, (ii) assertions that are annotated in the schedule and subsequently translated into invariants of the scheduled Event-B model, and (iii) the proof that the

scheduled model refines the restricted Event-B model, which relies on the automatically generated invariants.

Overall, our contribution is an approach to code generation from Event-B models that guarantees that generated programs correctly implement their Event-B specifications and therefore will not incur runtime errors such as arithmetic overflows. The novelty of our approach is the use of well-definedness in the restriction step to prevent runtime errors, a flexible scheduling language with assertions for specifying scheduling information during the second step, and the use of refinement in the fourth step of our approach to prove the generated program code's correctness. Based on our approach, we implemented a plug-in for the Rodin platform [3] and successfully generated code for industrial-scale case studies including an elevator control system and a train control system, both with strong safety properties. To make this paper self-contained, we illustrate our approach using a comparatively simple academic example from [2].

Structure. We briefly overview the Event-B modelling method in Section 2.1 and the "cars on a bridge" case study from [2] in Section 2.2. We use this example to illustrate the four steps of our approach to generating code from Event-B models in Section 3. In Section 4, we provide evidence for the general applicability of our approach. In Section 5, we compare our approach with the existing code generation tools for Event-B. We draw conclusions and discuss future work in Section 6.

2 Background

2.1 Event-B

Event-B [2] represents an extension as well as a simplification of the classical B-method [1], which has been focused around the general notion of *events*. Event-B has a semantics based on transition systems and simulation between such systems. We will not describe in detail the semantics of Event-B here; full details are provided in [2]. Instead, we will describe some Event-B modelling concepts that are important for the later presentation.

Event-B models are related by *refinement* and are organized in terms of the two basic constructs: *contexts* and *machines*. *Contexts* specify the static part of a model and may contain *carrier sets*, *constants*, *axioms*, and *theorems*. Carrier sets are similar to types. Axioms constrain carrier sets and constants, whereas theorems express properties derivable from axioms. The role of a context is to isolate the parameters of a formal model (carrier sets and constants) and their properties, which are intended to hold for all instances.

Machines specify behavioral properties of Event-B models. Machines may contain *variables*, *invariants*, *theorems*, and *events*. Variables v define the state of a machine, and are constrained by invariants $I(v)$. Theorems are properties derivable from the invariants. Possible state changes are described by events.

The term $e \mathrel{\widehat{=}} \mathbf{any}\ t\ \mathbf{where}\ G(t, v)\ \mathbf{then}\ S(t, v)\ \mathbf{end}$ represents an event e, where t is the event's *parameters*, $G(t, v)$ is the event's *guard* (the conjunction of one or more predicates), and $S(t, v)$ is the event's *action*. The guard states the condition

under which an event may occur, and the action describes how the state variables evolve when the event occurs. An event's action is composed of one or more *assignments* of the form $x := E(t, v)$, where x is a variable in v and $E(t,v)$ is an expression of the same type as x. Assignments in Event-B may also be nondeterministic. However, we ignore these assignments in our approach since we only translate deterministic assignments and force the user to first refine non-deterministic assignments into deterministic ones. All assignments of an action $S(t, v)$ occur simultaneously. A dedicated event without any parameter or guard is used for *initialisation*.

Refinement provides a means to gradually introduce details about the system's dynamic behaviour into formal models [2]. A machine **CM** can refine another machine **AM**. We call **AM** the *abstract* machine and **CM** the *concrete* machine. The states of the abstract machine are related to the states of the concrete machine by *gluing invariants* $J(v, w)$, where v are the variables of the abstract machine and w are the variables of the concrete machine. A special case of refinement (called superposition refinement) is when v is kept in the refinement, i.e. $v \subseteq w$. Intuitively, any behaviour of **CM** can be simulated by a behaviour of **AM** with respect to the gluing invariants $J(v, w)$.

Refinement can be reasoned about on a per-event basis. Each event e of the abstract machine is *refined* by one or more concrete events f. Simplifying somewhat, we can say that f refines e if f's guard is stronger than e's guard (*guard strengthening*), and the gluing invariants $J(v, w)$ establish a simulation of f by e (*simulation*).

2.2 Running Example

In this section, we describe the "cars on a bridge" example taken from [2, Chapter 2] that we use as a running example to illustrate our approach to code generation. The system's main functionality is to control the cars on a bridge between an island and the mainland. Due to the bridge's width, only traffic in one direction is allowed at a time. The system is equipped with four sensors to detect the presence of cars entering and leaving the bridge. The system controls the two traffic lights located at both ends of the bridge. Moreover, the maximum number of cars allowed on the island is limited. The Event-B model is gradually developed in four machines. The last refinement includes environment events modelling the movement of cars that triggers the sensors and controller events setting the traffic lights accordingly. For the purpose of illustrating our approach to generating code, we focus on the following events of the last refinement.

```
ML_out1 :              ML_out2 :              IL_tl_green :          ML_OUT_DEP :
when                   when                   when                   when
 ml_out_10 = TRUE       ml_out_10 = TRUE       il_tl = red            ML_OUT_SR = on
 a + b + 1 < d          a + b + 1 = d          0 < b                  ml_tl = green
then                   then                    a = 0                 then
 a := a + 1             a := a + 1             ml_pass = 1            ML_OUT_SR := off
 ml_pass := 1          ml_tl := red            ml_out_10 = FALSE     ml_out_10 := TRUE
 ml_out_10 := FALSE    ml_pass := 1            IL_OUT_SR = on        A := A + 1
end                    ml_out_10 := FALSE     then                  end
                       end                     il_tl := green
                                               ml_tl := red
                                               il_pass := 0
                                              end
```

The events ML_out1, ML_out2, and IL_tl_green are controller events and the event ML_OUT_DEP is an environment event. We omit other events for clarity. The constant

d represent the maximum number of cars allowed on the island. The variables a, b, c, ml_pass, il_pass are controller variables and the variable A is an environment variable. Other variables are shared variables representing the sensors (from the environment to the controller), i.e., ml_out_10, il_out_10, ml_in_10, il_in_10, ML_OUT_SR, and IL_OUT_SR, or the actuators (from the controller to the environment), i.e., ml_tl and il_tl. The interested reader can find the exact meaning of the variables in [2, Chapter 2].

3 A Code Generator for Event-B

In our approach, the code generator translates an Event-B model into C source code. As depicted in Figure 1, prior to generating code we must restrict the Event-B model and provide a schedule. Using the schedule as an input, the code generator then generates two outputs, C source code and an Event-B machine that we use to prove the correctness of the generated source code.

3.1 Well-definedness Restrictions

The final model of an Event-B development may still include parts that are not well-defined with respect to the target language. Using refinement, we restrict these remaining parts and thereby obtain a restricted model that is well-defined. Our plug-in checks that the model is restricted before generating source code for it. In the following, we describe the semantic differences between Event-B and C and describe our approach to establishing well-definedness.

Basic Types. The two basic types that our code generator supports are 32-bit integers and booleans. While the boolean type in C is equivalent to type BOOL in Event-B, the integer types have different ranges. We therefore define in Event-B the range of the C integer type as a constant C_INT ($C\_INT = -2147483648 .. 2147483647$) and require that every integer variable belongs to this set. C_INT can be seen as a restricted data type in the model and we say that a variable is of type C_INT whenever it belongs to the set described by the constant C_INT, i.e., $variable \in C\_INT$.

Arrays. We support one- and two-dimensional arrays for both basic types. Arrays are represented by total functions in Event-B. If a variable or constant is not of a basic type, then it must be of one of the array types in the table below, where k and l are natural numbers smaller than the maximum value in C_INT. We use $\mathcal{T}(x)$ to represent the translation of a string x that complies with Event-B syntax.

Event-B	C
$f \in 0 .. k \rightarrow C\_INT$	`int f[`$\mathcal{T}(k)$`+1]`
$g \in 0 .. k \rightarrow BOOL$	`bool g[`$\mathcal{T}(k)$`+1]`
$f \in 0 .. k \times 0 .. l \rightarrow C\_INT$	`int f[`$\mathcal{T}(k)$`+1][`$\mathcal{T}(l)$`+1]`
$g \in 0 .. k \times 0 .. l \rightarrow BOOL$	`bool g[`$\mathcal{T}(k)$`+1][`$\mathcal{T}(l)$`+1]`

The restriction on k and l guarantees that the size of a generated array is always positive and at most the maximum number in C_INT. However, the maximal allowed size of an array depends on the target system and its memory management. Hence, we cannot guarantee that the memory allocation at the beginning of the running program will succeed.

Arithmetic Operators. Careless use of arithmetic operators is the source of integer overflows in software. Since the integer type in Event-B does not have a lower or upper bound, the addition of two positive integer numbers always results in a positive integer number. In a C program, however, this result might be larger than the maximum integer number and cause a runtime error or be mapped to a negative number. Either way, the outcome of the computation is different from that in the Event-B model. Due to the restriction of integer variables to the type C_INT, assignments of the form $x := x + y$ are already checked for well-definedness when proving the preservation of the invariant $x \in C\_INT$. The intermediate results of multiple arithmetic operations and arithmetic operations in predicates, however, are not checked. To enforce the well-definedness of all arithmetic operations, we introduce special operators that are adapted to the integer type C_INT and we restrict the use of each arithmetic operator to just these.

$$c\_plus = \{\lambda a \mapsto b \cdot a \in C\_INT \wedge b \in C\_INT \wedge a + b \in C\_INT \mid a + b\}$$
$$c\_minus = \{\lambda a \mapsto b \cdot a \in C\_INT \wedge b \in C\_INT \wedge a - b \in C\_INT \mid a - b\}$$
$$c\_mul = \{\lambda a \mapsto b \cdot a \in C\_INT \wedge b \in C\_INT \wedge a * b \in C\_INT \mid a * b\}$$
$$c\_div = \{\lambda a \mapsto b \cdot a \in C\_INT \wedge b \in C\_INT \wedge b \neq 0 \wedge a \div b \in C\_INT \mid a \div b\}$$
$$c\_mod = \{\lambda a \mapsto b \cdot a \in C\_INT \wedge b \in C\_INT \wedge 0 \leq a \wedge 0 < b \mid a \bmod b\}$$

The result of an integer division in Event-B is always rounded towards zero as in the C99 standard. In C89 and C90, however, it is implementation dependant whether the result of an integer division is rounded towards zero or towards minus infinity. This difference is important when the integer division results is a negative number. When using a compiler compliant to C89 or C90, the definition of the c_div operation must be adapted to prevent negative results and a possibly inconsistent translation. No such action is required for the modulo operator since the domain of Event-B's modulo operation is already restricted to natural numbers.

Due to the use of lambda expressions in the operator's definition, arithmetic operations change from infix notation to function applications in the model. We keep this style in the translation to source code and define macros to replace the function calls during compilation by the standard operators.

```
#define c_plus(x,y)  (x+y)
#define c_minus(x,y) (x-y)
#define c_mul(x,y)   (x*y)
#define c_div(x,y)   (x/y)
#define c_mod(x,y)   (x%y)
```

Events. For the translation of events to source code, an event's parameters must be fixed to specific values. Theoretically, an event parameter that is fixed to a single value is not that useful as any occurrence of the parameter in guards and actions could just be replaced by its fixed value. For practical reasons, we support event parameters as local storage for computation results. If the result of a computation is used in more than one action, it is more efficient to do the computation only once and store the result.

A core concept of our approach is that the guards of the events are not translated, but their evaluation to true is guaranteed by the flow control structures of the schedule or more precisely by the specified branch conditions, loop conditions, and assertions. The only guards that are translated are those that specify the value of an event parameter. We require in the restricted Event-B model that for every event parameter there is exactly one guard of the form $parameter = \dots$, where the right-hand side of the equation must be an expression of type BOOL or C_INT.

Since an event's actions denote parallel assignments, the order of the actions does not matter in Event-B. This changes when we translate the actions into a sequence of single assignments. As a result, the right-hand side of the assignments in the source code cannot refer to the before-values of the variables. To overcome this issue, we restrict the actions of the event so that the right-hand side of an assignment does not refer to variables that already occurred on the left-hand side of a previous assignment. If this restriction is not guaranteed, the developer must either rearrange the actions where possible or introduce parameters as auxiliary variables to store the before-value of the conflicting variables. This task could be automated in a future version of our plug-in.

Expressions and Predicates. We restrict expressions and predicates to a subset of the Event-B syntax for which we provide the translation mappings presented in Table 2 in Section 3.3. In developments with arrays, there are often events with guards that contain quantifiers to express predicates on arrays. We therefore developed patterns for translating quantified predicates. Due to space restrictions we only present predicates with a single universal quantifier and omit translation patterns for existential quantification and combinations of multiple quantifiers. In our approach, the quantified predicate is translated to a function call of a dedicated function that evaluates the quantified predicate.

Assignments. In Table 3 in Section 3.3 we present the allowed assignments for updating variables in an event's action. The right-hand side of an assignment to a variable of type BOOL or C_INT must be an expression of the corresponding type. The update of arrays is slightly more difficult. We provide different translation rules for updating arrays at one or more positions and for overwriting an array with a set of index-value pairs. The bound variable used in the set comprehension is translated to the iteration variable of a for-loop.

Example. Returning to the "cars on a bridge" example, we first restrict the context of the development, i.e., the values for the constant d to C_INT. There are two options that we can take.

1. We apply generic instantation [7] to give d a concrete value (say $d = 20$) and prove that $d \in C\_INT$ as a theorem.
2. We add an axiom, i.e., $d \in C\_INT$ to further constrain d. In this way, d is left undefined and the user must define its concrete value within the program code. It is then the user's task to ensure that the concrete value satisfies the axioms.

In terms of safety guaranties, the first option is preferable as we prove that the values chosen for the constants imply the specified axioms. Hence, the current version of our plug-in follows this approach. The second option provides more flexibility as the definition of the constants' values can be written into a header file. The constants represent the parameters of the system and can easily be changed without generating new code. However, we have no practical way yet to enforce that the values in the header file are checked with respect to the model's axioms.

We also restrict the variables of the machine of the development. More precisely, integer variables of the machine (e.g., a, b, c) must be restricted to C_INT. This can

be done by proving the corresponding condition, i.e., $a \in C\_INT$, $b \in C\_INT$, and $c \in C\_INT$ as invariants or theorems of the machine. In our example, we can prove these conditions as theorems derivable from the restriction of d to C_INT, the fact that all variables are natural numbers, and the invariant $a + b + c \leq d$. Moreover, we replace all occurrences of arithmetic operators in the events' actions by their well-defined version. For example, events ML_out1 and ML_out2 are restricted as follows.

```
ML_out1 :                        ML_out2 :
when                             when
 ml_out_10 = TRUE                 ml_out_10 = TRUE
 a + b + 1 < d                    a + b + 1 = d
then                             then
 a := c_plus(a ↦ 1)               a := c_plus(a ↦ 1)
 ml_pass := 1                     ml_tl := red
 ml_out_10 := FALSE               ml_pass := 1
end                               ml_out_10 := FALSE
                                 end
```

Note that arithmetic operations used in event guards need not be restricted, except for those used to define parameters, since only parameter definitions are translated.

3.2 Scheduling the Model

To specify the execution order on the events of the restricted model, we provide the following scheduling language in our plug-in.

```
<schedule>  ::= <sequence>
<sequence>  ::= <sequence> ;{a} <sequence> | <block>
<block>     ::= event | <branch> | <loop>
<branch>    ::= if(c) <body> else <body> fi
<loop>      ::= do(c) <body> od
<body>      ::= "" | <sequence>
```

The symbols `a` and `c` represent a list of assertions and a loop or branch condition, respectively. The difference between a body and a sequence is that the body can be empty. For convenience, we can omit $\{a\}$ if there are no assertions required between two sequentially composed sequences. Furthermore, if there is no else part in the branch, we can just write `if(c) <body> fi`.

Example. The first part of our schedule is as follows. The numbers are automatically generated in the editor of our plug-in.

```
0:    if(ml_out_10 = TRUE)
1:      if(c_plus(c_plus(a↦b)↦1)<d)
2:        ML_out1
3:      else
4:        ML_out1
        fi
      fi;
      {ml_out_10=FALSE}
5:    if(il_tl=red ∧ 0<b ∧ a=0 ∧ ml_pass=1 ∧ IL_OUT_SR=on)
6:      IL_tl_green
      fi;
```

Note that arithmetic operations used in the branches must be restricted. Moreover, the assertion $\{ml\_out\_10 = FALSE\}$ before the branch at position 5: allows us to avoid checking this condition in the branch.

3.3 Translation to Source Code

The translation of the schedule is straightforward using if-else statements and while-loops. We omit the translation rules here and just give an example. Event blocks as well as branch and loop conditions are translated according to Tables 1-3. Note that we do not translate assertions, which are only used for the proof of correctness.

Table 1. Translation of Events

Event-B	C
evt-name	`{`
ANY s, t	`int s = `$\mathcal{T}(E_i(v,c))$`;`
WHERE	`bool t = `$\mathcal{T}(E_b(v,c,s))$`;`
$s = E_i(v,c)$	
$s \in C\_INT$ (theorem)	`vi = `$\mathcal{T}(E_i(v,c,s,t))$`;`
$t = E_b(v,c,s)$	`vb = `$\mathcal{T}(E_b(v,c,s,t))$`;`
$t \in BOOL$ (theorem)	`}`
THEN	
$vi := E_i(v,c,s,t)$	
$vb := E_b(v,c,s,t)$	
END	

The translation of the basic predicates and expressions is straight forward and similar to the translation mappings of the other approaches. Noteworthy is the possibility in our approach to translate quantified predicates, which are useful to express conditions in connection with arrays.

Table 2. Translation of Predicates and Expressions

Event-B	C	Event-B	C
$\neg x$	`!`$\mathcal{T}(x)$	$x \wedge \cdots \wedge y$	`(`$\mathcal{T}(x)$ `&& ... &&` $\mathcal{T}(y)$`)`
$\top$	`true`	$x \vee \cdots \vee y$	`(`$\mathcal{T}(x)$ `|| ... ||` $\mathcal{T}(y)$`)`
$\bot$	`false`	$x \Rightarrow y$	`(!`$\mathcal{T}(x)$ `||` $\mathcal{T}(y)$`)`
$a = b$	`(`$\mathcal{T}(a)$`==`$\mathcal{T}(b)$`)`	$x \Leftrightarrow y$	`((!`$\mathcal{T}(x)$ `||` $\mathcal{T}(y)$`) &&`
$a \neq b$	`(`$\mathcal{T}(a)$`!=`$\mathcal{T}(b)$`)`		`(!`$\mathcal{T}(y)$ `||` $\mathcal{T}(x)$`))`
$a < b$	`(`$\mathcal{T}(a)$`<`$\mathcal{T}(b)$`)`	$f(a)$	`f[a]`
$a \leq b$	`(`$\mathcal{T}(a)$`<=`$\mathcal{T}(b)$`)`	$f(a \mapsto b)$	`f(`$\mathcal{T}(a)$`,`$\mathcal{T}(b)$`)`
$a > b$	`(`$\mathcal{T}(a)$`>`$\mathcal{T}(b)$`)`	$c\_operator(a \mapsto b)$	`c_operator(`$\mathcal{T}(a)$`,`$\mathcal{T}(b)$`)`
$a \geq b$	`(`$\mathcal{T}(a)$`>=`$\mathcal{T}(b)$`)`		
		$\forall i \cdot i \in c\_upto(j \mapsto k)$	`eval_uid()`
$identifier$	`identifier`	$\Rightarrow P(v,c,i)$	

```
bool eval_uid() {
  for(int i=𝒯(j); i<=𝒯(k); i++) {
    if (!𝒯(P(v,c,i)))
      return false;
  }
  return true;
}
```

Event-B	C
$TRUE$	`true`
$FALSE$	`false`

Noteworthy in our translation of assignments are the different patterns for updating arrays. We can either update an array at one or more fixed positions or we can iterate through the array and use a predicate to evaluate at runtime which positions are updated.

Table 3. Translation of Assignments

Event-B	C
$vi := b$	`vi = `$\mathcal{T}(b)$`;`
$vb := r$	`vb = `$\mathcal{T}(r)$`;`
$f(a) := b$	`f[a] = `$\mathcal{T}(b)$`;`
$g(a \mapsto b) := r$	`g[a][b] = `$\mathcal{T}(r)$`;`
$f := f \mathbin{\lhd\mkern-9mu-} \{a1 \mapsto b1\} \mathbin{\lhd\mkern-9mu-} \ldots \mathbin{\lhd\mkern-9mu-} \{am \mapsto bm\}$	`f[`$\mathcal{T}(a1)$`] = `$\mathcal{T}(b1)$`;`
	`...`
	`f[`$\mathcal{T}(am)$`] = `$\mathcal{T}(bm)$`;`
$f := f \mathbin{\lhd\mkern-9mu-} \{i \cdot i \in c\_upto(j \mapsto k) \wedge P(v,c,i) \mid$	`for(int i=`$\mathcal{T}(j)$`; i<=`$\mathcal{T}(k)$`; i++){`
$\qquad\qquad\qquad E_1(v,c,i) \mapsto E_2(v,c,i)\}$	`  if(`$\mathcal{T}(P(v,c,i))$`)`
	`    f[`$\mathcal{T}(E_1(v,c,i))$`] = `$\mathcal{T}(E_2(v,c,i))$`;`
	`}`

Example. The C code generated corresponding to the above snippet of our schedule is as follows.

```
if (ml_out_10 == true){
  if (c_plus(c_plus(a,b),1) < d){
    a = c_plus(a,1);
    ml_pass = 1;
    ml_out_10 = false;
  }
  else{
    a = c_plus(a,1);
    ml_tl = red;
    ml_pass = 1;
    ml_out_10 = false;
  }
}
if ((il_tl == red && (0 < b && (a == 0 && (ml_pass == 1 && IL_OUT_SR == on))))) {
  il_tl = green;
  ml_tl = red;
  il_pass = 0;
}
```

3.4 Proving the Correctness of the Scheduled Model

To prove the correctness of the generated source code, we generate a scheduled model that includes the schedule encoded in the machine as follows. We introduce a new variable pc that represents the program counter and add events that simulate the update of the program counter according to the schedule. The controller events are refined by removing all guards except for parameter initialisations and adding the action $pc := pc+1$ to simulate the increment of the program counter. The additional events for the different blocks are as follows.

Branch. For every branch, we generate a set of events. The events differ slightly depending on whether the branch has an "else" part or not. The symbols s and m represent the block's start and middle position, respectively within the schedule and e is the next valid position in the schedule after the end of the branch. These numbers are automatically computed by the plug-in. The branch condition is represented by bc.

```
if_true :              if_false (long form) :   if_false (short form) :   if_exit (long form) :
when                   when                     when                      when
  pc = s                 pc = s                   pc = s                    pc = m
  bc                     ¬bc                      ¬bc                     then
then                   then                     then                        pc := e
  pc := pc + 1           pc := m + 1              pc := e                 end
end                    end                      end
```

Loop. For a loop we generate the following events. The symbols s and e represent the block's start and end position, respectively. Both numbers are automatically computed by the plug-in. The loop condition is represented by lc.

```
do_true :          do_false :         do_return :
when               when               when
  pc = s             pc = s             pc = e
  lc                 ¬lc              then
then               then                 pc := s
  pc := pc + 1       pc := e + 1      end
end                end
```

Example. Based on the (automatically generated) program counter associated with the statements, the scheduled Event-B model corresponding to the above schedule snippet is as follows. Most events are required for modelling the control flow of the schedule determined by the program counter pc.

```
if_ml_out_10_true :          if_ml_out_10_false :            if_ml_out_true :
when                         when                            when
  pc = 0                       pc = 0                          pc = 1
  ml_out_10 = TRUE             ml_out_10 = FALSE               c_plus(c_plus(a ↦ b) ↦ 1) < d
then                         then                            then
  pc := pc + 1                 pc := 5                         pc := pc + 1
end                          end                             end


ML_out1 :                    if_ml_out_exit :   if_ml_out_false :
when                         when               when
  pc = 2                       pc = 3             pc = 1
then                         then                 ¬c_plus(c_plus(a ↦ b) ↦ 1) < d
  a := c_plus(a ↦ 1)           pc := 5          then
  ml_pass := 1               end                  pc := 4
  ml_out_10 := FALSE                            end
  pc := pc + 1
end


ML_out1 :                    if_il_tl_green_true :
when                         when
  pc = 4                       pc = 5
then                           il_tl = red ∧ 0 < b ∧ a = 0 ∧ ml_pass = 1 ∧ IL_OUT_SR = on
  a := c_plus(a ↦ 1)         then
  ml_tl := red                 pc := pc + 1
  ml_pass := 1               end
  ml_out_10 := FALSE
  pc := pc + 1
end


IL_tl_green :        if_il_tl_green_false :
when                 when
  pc = 6               pc = 5
then                   ¬(il_tl = red ∧ 0 < b ∧ a = 0 ∧ ml_pass = 1 ∧ IL_OUT_SR = on)
  il_tl := green     then
  ml_tl := red         pc := 7
  il_pass := 0       end
  pc := pc + 1
end
```

In addition, we generate invariants to capture the effect of the control flow and the user-defined assertions.

invariants :
if_ml_out_Pre : $pc = 1 \Rightarrow ml\_out\_10 = TRUE$
ml_out1_Pre : $pc = 2 \Rightarrow ml\_out\_10 = TRUE \wedge c\_plus(c\_plus(a \mapsto b) \mapsto 1) < d$
if_ml_out_Post : $pc = 3 \Rightarrow ml\_out\_10 = FALSE$
ml_out2_Pre : $pc = 4 \Rightarrow ml\_out\_10 = TRUE \wedge \neg(c\_plus(c\_plus(a \mapsto b) \mapsto 1) < d)$
if_il_tl_green_Pre : $pc = 5 \Rightarrow ml\_out\_10 = FALSE$
IL_tl_green_Pre : $pc = 6 \Rightarrow ml\_out\_10 = FALSE \wedge$
$$(il\_tl = red \wedge 0 < b \wedge a = 0 \wedge ml\_pass = 1 \wedge IL\_OUT\_SR = on)$$

Notice how the invariants take into account the effect of the nested branches, e.g. when $pc = 2$, and of assertions, e.g. when $pc = 6$. Proving that the scheduled Event-B model refines the restricted Event-B model is straightforward with these invariants, except for the following problem regarding shared variables.

Shared Variables and Atomicity Our schedule imposes an atomicity assumption, captured by the scheduled Event-B model, representing the semantics of the program code. The atomicity is indicated by the values of the program counter pc. For example, we assume that the evaluation of conditions in branches and loops are atomic. Moreover, we also assume that the assignments of the original events (which are translated as sequential updates) are executed atomically. However, we break the atomicity assumption between checking the guards and executing the actions of the original events. In particular, the evaluation of the event guards is often distributed to different branch and loop conditions. For example, the guard of IL_tl_green is partially checked by the branch condition at $pc = 5$ and partially guaranteed (assertion $ml\_out\_10 = FALSE$) by the control flow before that. Since this atomicity assumption differs from the atomicity assumption of the restricted Event-B model, where the evaluation of an event's guards and the execution of its actions are assumed to be atomic, inconsistency can arise. This is in particular the case when shared variables are used in the event's guard. Since we schedule only controller events, the environment events in the scheduled Event-B model can updated the shared variable at any time. This is reflected by unprovable invariant preservation proof obligations of the scheduled Event-B model. In our example, variable ml_out_10 is assumed to be shared between the controller and the environment. More specifically, the environment can change the value of ml_out_10 with its event ML_OUT_DEP as follows.

ML_OUT_DEP :
when
 $ML\_OUT\_SR = on$
 $ml\_tl = green$
then
 $ML\_OUT\_SR := off$
 $ml\_out\_10 := TRUE$
 $A := A + 1$
end

An attempt to prove that ML_OUT_DEP maintains invariants like IL_tl_green_Pre will fail, since our model does not prevent the occurrence of ML_OUT_DEP between checking the branch condition at $pc = 5$ and executing IL_tl_green's action at $pc = 6$. To remedy the situation, we add a guard, e.g., $pc = 0$ to ML_OUT_DEP to prevent the environment event from occurring during the controller's execution. The meaning

of this guard is an *assumption* on the overall system such that when a car leaves the corresponding sensor, then the controller has finished processing the last message and is ready to process the next message. Notice that a similar assumption regarding the speed of the controller has also been made for the original development in [2]. In general, guarding the environment events might give rise to assumptions that are unrealistic. In this case, we must return to the original development and perform further refinement, e.g., to introduce a private copy of the shared variables for the controller. Essentially, we anticipate the possible interference of the environment and account for that earlier in the development.

4 Experience

As stated in the introduction, we applied our approach to different developments. In addition to the "cars on a bridge" example, we generated code for two more sophisticated case studies: an elevator control system and a train control system. While the elevator control system is a simplification of real elevator systems, we developed the core functionality of the train control system from a real specification [7].

Table 4. Statistics

	"cars on a bridge" controller	elevator control system	train control system
controller variables:	5	2	15
shared variables:	8	12	21
controller events:	8	30	34
refinement steps:	3	3	105
schedule lines:	18	88	90
invariants encoding the schedule:	14	67	69
events encoding the schedule:	18	99	92
POs for refinement:	454	9075	8757
lines of C code:	127	312	373

The numbers in Table 4 show that the elevator and train system are comparable in terms of the size of their schedules and the number of proof obligations required to prove refinement. This may be surprising since the train control system is substantially more complex than the elevator control system and required considerably more effort to develop, which is reflected by the large number of refinement steps. This is because the train control system's development includes several refinement steps that already account for the later restriction step and translation to code. Hence, both control systems are refined to a level where they are close to their final implementation and the translation to program code becomes straightforward. With the restriction of a model as the first step of our code generation approach, we force the developer to refine the

model to a concrete level and therefore keep the translation effort in check. For this reason our approach works equally well for small academic examples and large, complex industrial systems.

The number of proof obligations generated to prove that the scheduled Event-B model refines the restricted model is rather high. For every invariant/event pair, we have a proof obligation of the form $pc = M \vdash pc = N \Rightarrow \ldots$, where N and M are numbers. However, most of them are trivial as $N \neq M$ holds. The number of relevant proof obligations (i.e. $N = M$) is less than 180 for all three developments and at least 77% are automatically discharged. In fact, our plug-in can generate these relevant proof obligations directly from the schedule rather than having Rodin generating them together with all the irrelevant ones. The remaining task is to prove that this set of proof obligations indeed implies those proof obligations generated to prove the refinement relation. This proof can be done once at the meta-level and is valid for all translations.

5   Related Work

Here we discuss related work in more detail.

Merging Rules. Merging rules are introduced in [2] as a mechanism for synthesising sequential programs from Event-B models. There are two rules for creating branches and loops that constitute patterns for developing sequential programs. As a result, the form of the programs are limited and not every program can be synthesised from its Event-B model. For example, a sequential statement is only possible after a loop.

B2C Tool. The B2C tool [10] was developed to generate code for a specific Event-B model of an instruction set architecture. As a result, the plug-in supports only the translation of the Event-B syntax used in this particular model. The most significant shortcoming is that it does not support contexts and therefore cannot be used when constants and sets are used in a machine.

With the B2C tool, there is no possibility to specify a desired execution order of the events. For every event in the model, a C function is created that checks the guards before the actions are executed. In a function named "iterate", all these event functions are combined in a sequence of function calls equivalent to the event ordering in the Event-B machine. As soon as a function call is successful (the actions were executed) the iterate function returns. In the main function, first the initialisation function is called and then a while loop calls the iterate function as long as there is no deadlock in the system. The disadvantage of leaving the iterate function after a successful action execution of an event is that events at the bottom of the iteration sequence might never be executed.

EB2ALL Tool. EB2ALL [9] is a set of tools for generating code for different target languages. Currently there are four plug-ins included for translations to C, C++,C#, and Java respectively. The EB2ALL tool is based on the B2C tool. The authors of the tool argue that its correctness is justified by an observable equivalence between the Event-B model and the generated code together with some meta-proofs. It is not specified what notion of observational equivalence is intended and no details on the meta-proof

are provided. Furthermore, they state that the generated code usually must be altered manually after generation and that correctness is maintained if the manually added code is also verified in some way. Again, formal details are lacking.

In EB2ALL, the default scheduling is the same as in B2C. The tool provides an optimisation by automatically grouping events that have common guards. This is done by analysing the refinement relation of the events. Two events that both refine the same abstract event have the guards of the abstract event in common. Within the iterate function, these common guards are translated into an if-statement surrounding the function calls of the corresponding events. The intention is that if the guards of the abstract event evaluate to false, then there is no need to check the guards of the refining events. Unfortunately, this approach only works if the guards of the abstract event are all deterministic and translatable. Furthermore, this only produces more efficient code when many events refine one single abstract event. Otherwise, the overhead of additional if-statements may outweigh any efficiency gains.

Tasking Event-B. Tasking Event-B [6] is a tool developed for code generation from Event-B models into code with a special focus on concurrent processes. Currently, the tool supports translations to C, Ada and Java. As in EB2ALL, expressions with multiple arithmetic operators are supported. Currently it is not checked whether arithmetic operators maintain the lower and upper bound of the target data type; hence runtime overflows are possible.

Tasking Event-B is the most mature among the existing tools for code generation with respect to scheduling. It is the only tool that provides a scheduling language for user defined scheduling of events. Unfortunately, the language is very restrictive. The bodies of loops and branches are limited to single events. Hence, there is no support for schedules that include structures such as nested branches or a sequence of events within a loop.

Scheduling Patterns. We are unaware of any tool support for the scheduling patterns introduced in [4, 5]. The attractiveness of the work is the proof of correctness for the patterns done using set transformers. However, this reasoning must be done manually. Furthermore, our scheduling language is more expressive than the scheduling language defined in [4]. For example, nested branches are not possible in [4].

Classical B. Our scheduling language has features similar to (classical) B [1], for example conditional statements, sequential statements and loops. In B, the last model of a refinement chain is a special construction, the IMPLEMENTATION, from which program code can be generated. Variables of the IMPLEMENTATION must be either *concrete variables* (i.e., of some implementable datatype) or variables of some predefined libraries. Updates of the variables must be well-defined, which is captured by the preconditions of the corresponding assignments. However, loop and branch conditions do not have preconditions to enforce their well-definedness. As a consequence, they are restricted to predicates over simple expressions (e.g., no arithmetic operations are allowed). In our approach, we check all conditions for well-defineness, hence they can contain any expressions. Furthermore, our approach allows us to state assertions between two sequential blocks. As a result, proof obligations can be generated for each block sepa-

rately. In B, the effects of the sequentially composed statements are combined together, which often results in complicated proof obligations.

6   Conclusion

We presented our approach to generating program code from Event-B models. Our approach is correct-by-construction and relies on reasoning about well-definedness, assertions, and refinement. Although we presented only the translation to C source code, our approach is also applicable to other languages by adapting the notion of well-definedness and the restriction step to the corresponding target language.

As future work we would like to consider loop termination and liveness properties in general. The challenge here is to integrate standard loop variant reasoning into the scheduled Event-B model. Naturally, this will lead to reasoning about deadlock-freedom and event convergence properties as shown in [8].

Furthermore, as mentioned in Section 4, we have identified the set of relevant proof obligations which can be generated directly from the schedule. We are working on the meta-proof that this set of proof obligations indeed guarantees that the scheduled Event-B model refines the restricted Event-B model. Note that our approach is also correct without the meta-proof, but requires more proof obligations to be proved.

References

1. Abrial, J.R.: The B-book: Assigning Programs to Meanings. Cambridge University Press (1996)
2. Abrial, J.R.: Modeling in Event-B: System and Software Engineering. Cambridge University Press (2010)
3. 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. Software Tools for Technology Transfer 12(6), 447–466 (Nov 2010)
4. Boström, P.: Creating Sequential Programs from Event-B Models. In: Méry, D., Merz, S. (eds.) IFM. Lecture Notes in Computer Science, vol. 6396, pp. 74–88. Springer (2010)
5. Boström, P., Degerlund, F., Sere, K., Waldén, M.A.: Derivation of concurrent programs by stepwise scheduling of Event-B models. Formal Asp. Comput. 26(2), 281–303 (2014)
6. Edmunds, A., Butler, M.: Tasking Event-B: An Extension to Event-B for Generating Concurrent Code. In: 4th Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software (2011)
7. Fürst, A., Hoang, T.S., Basin, D., Sato, N., Miyazaki, K.: Formal System Modelling Using Abstract Data Types in Event-B. In: Ameur, Y.A., Schewe, K.D. (eds.) Abstract State Machines, Alloy, B, TLA, VDM, and Z. Lecture notes in computer science 5000 (2008) - 6999 (2012), vol. 8477, pp. 222–237. Springer, Berlin (2014)
8. Hoang, T.S., Abrial, J.R.: Reasoning about Liveness Properties in Event-B. In: Qin, S., Qiu, Z. (eds.) Formal Methods and Software Engineering. Lecture Notes in Computer Science, vol. 6991, pp. 456–471. Springer-Verlag, Durham, United Kingdom (Oct 2011)
9. Méry, D., Singh, N.K.: Automatic Code Generation from Event-B Models. In: Proceedings of the Second Symposium on Information and Communication Technology. pp. 179–188. SoICT '11, ACM, New York, NY, USA (2011)
10. Wright, S.: Automatic Generation of C from Event-B. In: Workshop on Integration of Model-based Formal Methods and Tools (February 2009)