Fredrik Degerlund | Richard Grönblom | Kaisa Sere

Code Generation and Scheduling of Event-B Models

Turku Centre *for* Computer Science

TUCS Technical Report
No 1027, December 2011

Code Generation and Scheduling of Event-B Models

Fredrik Degerlund
Åbo Akademi University, Dept. of Information Technologies,
Joukahainengatan 3-5, 20520 Åbo/Turku, Finland
fredrik.degerlund@abo.fi
Richard Grönblom
Åbo Akademi University, Dept. of Information Technologies,
Joukahainengatan 3-5, 20520 Åbo/Turku, Finland
Kaisa Sere
Åbo Akademi University, Dept. of Information Technologies,
Joukahainengatan 3-5, 20520 Åbo/Turku, Finland
kaisa.sere@abo.fi

TUCS Technical Report
No 1027, December 2011

Abstract

Event-B is a formal method for full system modelling, and the RODIN platform provides tool support for it. The method can be used for stepwise development of parallel programs, but there are different approaches to code generation and execution of the resulting code. In this paper, we demonstrate how C++ code can be generated using a separate plug-in for the RODIN tool, and how the resulting code can be scheduled concurrently using a dedicated tool. While our approach is related to animation in preserving the event nature, it supports execution over several processors or a network using the MPI (Message Passing Interface) framework.

Keywords: Formal methods, Event-B, RODIN, scheduling, parallelism, MPI

TUCS Laboratory
Distributed Systems Design Laboratory

1 Introduction

Event-B [1] is a state-based formal method for full-system modelling, and it can be used for stepwise development of software. The RODIN platform [18] provides tool support for Event-B, and different functionality can be achieved through custom plug-ins. Software production calls for code generation, and, in the case of concurrent programs, a means of co-ordinating execution of the resulting code. In this paper, we suggest one approach to code generation in Event-B. We have written a plug-in for the RODIN platform that translates models into C++ [20] code, which can then be compiled into object code. The generated code consists of C++ methods, which can be executed in parallel using a separate scheduler that we have developed. Scheduling depends on a behavioural semantics, which for our tool is inherited from the Action Systems formalism [2], and which allows for a parallel interpretation of programs. We support parallel execution of independent events by using the MPI (Message Passing Interface) framework [16], which allows events to be distributed over several cores or processors, as well as over a network. The work presented in this paper is based on the master's thesis of Grönblom [10].

The rest of the paper is structured as follows. In section 2, we shortly discuss Event-B, related formalisms and generation of code. In section 3, we focus on our code generation plug-in that can translate a certain class of Event-B models into C++ code. Next, in section 4, we present a scheduling tool designed to execute the code produced by the plug-in on a topology consisting of one master and several slave nodes. We also discuss scheduling policies as well as parameters that can be given to control what events are executed on which node. We then give a practical example in section 5, in which our approach is used to generate code from a factorisation model and execute it on processors in a cluster. Finally, we sum the paper up in section 6, where we also discuss related work.

2 Event-B

2.1 Background and code generation

Event-B has its roots in the B method and the Action Systems formalism. The B method was designed for correct-by-construction development of software, it is based on refinement, and tool support is provided by Atelier B and the B toolkit. The Action Systems formalism is also based on refinement and the correct-by-construction paradigm, but unlike the B method, it exhibits an event-based behavioural semantics. This approach has later been applied on the B method in the form of B Action Systems [21], whereby the established B tools can, to a certain degree, be used to prove correctness. Event-B is a descendant of B Action Systems, but with tool support of its own [18]. It does, however, use a modified modelling language, and code generation was initially not supported by the tool.

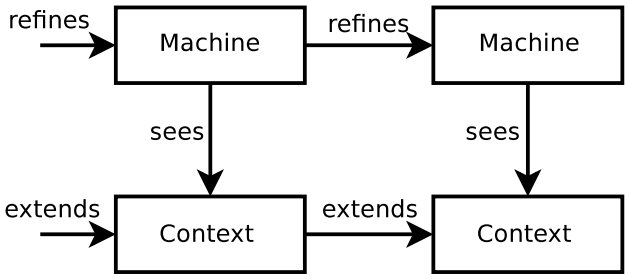

Figure 1: Hierarchy of machines and contexts in Event-B.

One strategy for code generation was proposed by Wright [22], particularly for use with a virtual machine framework. The method we propose is based on his work, as well as on an approach previously developed [6] for B Action Systems.

2.2 Event-B syntax and semantics

In Event-B, models are expressed in the form of *machines* and *contexts*. Machines contain *variables*, *invariants* and *events*, whereas contexts consist of *carrier sets*, *constants* and *axioms*. Machines can be declared to *see* contexts, whereby they are free to make use of the values contained in the context. Contexts can *extend* another context, whereas machines can *refine* each other. This results in a chain of machines and contexts, in which models are developed in a stepwise manner. The modelling hierarchy can be illustrated [1] as in figure 1, where the most concrete versions are shown to the right.

Machine structure. The general structure of a machine is shown in figure 2. The clause MACHINE contains the name of the machine. If it refines another machine, the name of that machine is given in the REFINES clause. All contexts that are seen by the machine are listed in SEES. The variables of the machine are listed in the VARIABLES clause, and their types are given in INVARIANTS, together with other properties that should always hold in the model. The VARIANT clause is used under certain circumstances when convergence has to be shown. Finally, the events of the machine are given in the EVENTS clause. Events model the behaviour of the machine in terms of state transitions on the state space made up of the variables.

Events. Events can be written in the following form [12]:

E = **when** $G(v)$ **then** $v :| S(v,v')$ **end**

An event consists of two parts: a *guard* $G(v)$ and an *action* $v :| S(v,v')$. The guard states a necessary condition for the action to take place, and when the guard eval-

Figure 2: The structure of an Event-B machine.

uates to true, the event is said to be *enabled*. The action describes the relationship between the value of the variables before the action takes place (v) and right after it has occurred (v'). The expression $v :| S(v,v')$ can intuitively be understood so that the variables of the machine are assigned new values in such a way that the action relation $S(v,v')$ holds. Since there may be several possible combinations of new values satisfying the relation, the assignment can also be non-deterministic. Each machine also contains a special initialisation event that has no guard, and the action of which can be expressed as $v :| A(v')$. This event is intended to take place at the time the machine is initialised, before any other events are executed. Since it initialises the state space, its action $v :| A(v')$ does not depend on previous values of the variables. Correctness properties are in Event-B expressed as proof obligations that have to be discharged for each model. Such proof obligations include properties such as invariant preservation, but also refinement correctness is expressed in the form of proof obligations.

Behavioural semantics. Strictly speaking, Event-B has no fixed behavioural semantics, and any behavioural semantics that is compatible with the Event-B proof obligations can be applied [12]. However, the same semantics as in the Action Systems formalism is typically used. This semantics will be assumed in this paper, and it intuitively works as follows. When a machine starts executing, its initialisation event is run to assign initial values to the variables. The rest of the events are then considered to be inside a loop where enabled events are chosen for execution in a non-deterministic order. Events may enable and disable each other, as part of the execution of their action, and the machine terminates only when none of the events is enabled any more. This behavioural semantics also has a parallel interpretation. Events are assumed to be atomic, whereby two or more events that do not interfere with each other can be executed in parallel. Interference freedom can be guaranteed if the events have no variables in common, or if the variables they have in common are only read, but never written, by the events in question.

3 Code generation

Event-B in its basic form does not support translation of models into executable code. We now show a means of translating a model into object code that can be executed as a computer program. In our approach, translation is made from the last refinement step, also called the ultimate refinement of a model. Other attempts at generating code from Event-B models are summarised in section 6, where we compare them to our framework.

3.1 Approach summary

To achieve code generation, we have developed a plug-in for the RODIN platform. The plug-in accepts a certain subset of Event-B models, and translates them into C++ code, which is based on the C language, but possesses additional high-level features. However, C++ specific features are mostly not used by the code generator, and it could easily be modified to generate standard C code instead. Since both C and C++ are widely used and their compiled code is generally fast, we consider both of them to be suitable as a target language for model translation, which is important in parallel computing.

The code generator is written in Java as a plug-in for the RODIN platform. The Eclipse IDE, which RODIN's interface is based on, provides good support for plug-in development. It provides an application programming interface (API) for creating the user interface of a plug-in. A plug-in can access models that have been created with RODIN, from a database. Event-B components are fetched through an interface to the database. For example, one can fetch the invariants of a machine through a function call. This interface is used by the code generator to fetch the components of a model.

Our code generator does not only translate code on a one-to-one basis, but also adds components needed for parallel execution in the scheduler, such as the dependency matrices that we discuss in section 4. Since such computations would be cumbersome to do by hand, we consider tool support mandatory for our framework. The output of the code generator consists of a model converted to a C++ class, in which the events of the model are represented as methods. The idea is that our scheduler (discussed in section 4) executes and accesses the class through an *interface* that we have defined. An interface consists of a set of operations that can be executed by the clients. The interface of the model contains several different operations on the model. For example, it contains operations for executing an event, checking a guard and accessing a variable.

3.2 Event-B0

In most cases, an Event-B model cannot be elementarily translated into computer code of some programming language. This is because specification languages are

very different from programming languages. They both serve a different purpose and there is no one-to-one mapping between them. In our opinion, the best way to handle this difference is to define a subset of Event-B that consists of components having a direct equivalent in the target language. We have defined such a "concrete" subset, called Event-B0. This decision was inspired by a very similar set, called B0, defined in classical B [5], which contains a subset of classical B with some additional constructs. B0 contains only concrete data types and operations on these, but not any abstract components that would be non-trivial to translate. It is equivalent to a limited programming language that can be converted into programming languages such as Ada or C. Components included in B0 are, among others, *integers*, *arrays*, *enumerated sets* and *arithmetic operations*. Atelier B and the B Toolkit both include a code generator that converts B0 models into computer code. For this process, both tools have a separate B0 checker that checks if a model can be translated.

Ranges of variables. The components that we have allowed in Event-B0 are those that have a low level equivalent in C++. They are very general and can be expressed in most programming languages. Another restricting factor of Event-B0 is the fact that computer memory is limited, so the size of every type and data set has to be assigned a limited size. Otherwise, it would be possible for variables to *overflow*, which is a situation when a variable is assigned a value outside its available storage space on the computer. These situations normally lead to incorrect behaviour. Integer overflow in the C programming language causes undefined behaviour [14], so we have to prevent situations where it can occur.

Most programming languages have minimum and maximum integer values, often denoted INT_MIN and INT_MAX, respectively. Our target computer uses signed 32-bit integers with a range between -2^{32} and $2^{32}-1$, from -2147483648 to 2147483647, in C++ programs. If the value of a variable goes beyond this range, the program will not be valid. The size of enumerated sets and arrays can neither be of infinite size. A maximum size has to be defined in an invariant.

To prevent overflowing, we impose restrictions on the arithmetical operations in Event-B0. We only allow operations that have two factors. If there are more than two factors, the whole expression can be valid, even though a part of the expression overflows. For instance, the arithmetic operation (INT_MAX+1)-1 overflows in the first part. The maximum integer value plus one cannot be represented by the storage space of the integer. However, the whole expression is of legal range, as the result is INT_MAX.

However, there are techniques for handling integer overflow. It is common that *wrap around* is used in situations of this type. If an arithmetic expression overflows, it continues the operation from the opposite extreme. This is similar to how the modulo operator works. For example, if an arithmetic addition overflows with 5, the result will be evaluated to INT_MIN+4. By using wrap around, the

operation (INT_MAX+1)-1 would be evaluated correctly on a computer. First, (INT_MAX+1) would overflow so the value will be in the opposite extreme, i.e. INT_MIN. The second part of the expression, INT_MIN-1, will also overflow and be evaluated in INT_MAX. Hence, the correct result is achieved. Arithmetic operations in Event-B0 would be less strict by using a target language that supports wrap around.

Invariants. Invariants are used in Event-B0 for assigning types and restrictions to variables. Every variable has to be assigned a type of either integer, Boolean, enumerated set or an array of one of these types. Event-B0 has no other restrictions on the invariant, as they only concern the verification part of the model, but not the functionality. Therefore, they will not be needed in the executable version, and do not have to be translated.

3.3 Formal presentation of Event-B0

We now present Event-B0 formally by using simplified *production rules* in *Backus-Naur Form*. Variables are defined in an identifier and assigned a type in an invariant. The type is assigned by using the "belongs to" operator, denoted by the symbol $\in$. Arrays are defined using the function operator, denoted "$\rightarrow$," from a numerical interval, 0 to *n*, that maps to one of the three Event-B0 types. Type assignments are defined in the following way:

TypeAssignment ::=
Identifier $\in$ (*BOOL*|*integer*|*enumerated set*)
ArrayTypeAssignment ::=
Identifier $\in$ 0..*N* $\rightarrow$ (*BOOL*|*integer*|*enumerated set*)

Variable assignments have to be written in a very simple form in Event-B0. On the left-hand side we have a single variable or array element and on the right-hand side a variable, a value or a two-part arithmetic operation. One important thing to note is that events are atomic, which means that all the substitutions of an event are executed at once. This has the effect that variable assignments are updated only after the whole event has been executed. This is not the case in C++, where any variable assignments instantly take effect. Hence, we have to forbid events that first update a variable and then use it in a later substitution, as it would potentially lead to incorrect C++ code. Assignments are formally expressed as follows:

Assignment ::=
Identifier := (*Identifier*|*Value*|*ArithmeticOperation*|*BOOL*)

The arithmetic operations allowed in Event-B are the four basic ones: *addition*, *subtraction*, *multiplication* and *division*, including the *modulo* operation. They

can have one value or one identifier on each side:

ArithmeticOperation ::=
(*Value*|*Identifier*) (+ | - | * | ÷ | mod) (*Value*|*Identifier*)

Event guards are predicates that have to be true for the statements of the event to be executed, i.e. event preconditions. Event parameters are defined in the same section as the guards. In Event-B, parameters can be assigned a type implicitly. For instance, a parameter occurring in an arithmetic predicate will be assigned a numerical type automatically. This is similar to implicit type casting in dynamic languages. However, in Event-B0, every parameter has to be assigned a type explicitly, as in static programming languages. Guards can be expressed as follows:

RelationalExpression ::=
Expression RelationalOperator Expression
Expression ::=
(*Identifier* | *Value* | *ArithmeticOperation*)

The relational operators can be of six different types in Event-B0. Integers can be used with all operators. Booleans and enumerated sets can only be used with the “equals” and “not equals” operators. The following relational operators are allowed:

RelationalOperator ::=
(= | ≠ | < | ≤ | > | ≥)

3.4 Operation of the code generator

The plug-in starts by fetching the most refined machine in a chosen Event-B model, constituting the most concrete version in the refinement chain. This machine needs to be in Event-B0 form in order to be translatable by the code generator. One problem that we encountered with the refinement aspect of Event-B is that some elements can only be found in abstract machines from earlier steps of the refinement chain. With components scattered across several machines and contexts, one would have to first merge all machines into one concrete machine. However, this is not supported by our code generator, as we have not focused on model merging. Therefore, the code generator demands that all components that are used in the most refined machine be situated in it. The code generator performs translation and Event-B0 checking simultaneously. It checks on the fly if components are of Event-B0 form and if they are, it proceeds to convert them to C++ code. If an illegal component is encountered, the plug-in terminates with an error code describing why the component could not be translated.

Event-B	C++
MACHINE m	`int var1`
VARIABLES	`bool var2`
var1	`int var3[N]`
var2	`bool var4[N]`
var3	
var4	
INVARIANTS	
$var1 \in$ -2147483648..2147483647	
$var2 \in BOOL$	
$var3 \in 0..N \rightarrow$ -2147483648..2147483647	
$var4 \in 0..N \rightarrow BOOL$	

Figure 3: Translation of type assignments.

Every event of a machine is translated into two separate functions: one that contains the guard and one that contains the assignments. This design decision was taken due to the fact that the guard checking is done by the master node before an event is run, and the slaves only execute the substitutions of the event.

3.5 Translation rules of the code generator

This section presents the rules of how the components of an Event-B model are translated into C++ code. In Event-B machines, program variables are defined in two steps. The variable first has to be defined by a unique name in the VARIABLES clause. The variable is then assigned a type in an invariant. It is also possible to add other restricting invariants to variables. However, such invariants will not be converted to code because they concern the correctness of the model, not the functionality. Translation rules for type definitions are defined in figure 3.

In every Event-B model, all variables have to be assigned initial values in the *INITIALISATION* event. Integers, Booleans and enumerated sets always have to be given a deterministic value. However, we have not implemented a way of initializing arrays, as large arrays would require hundreds of substitutions in the initialisation. The code will automatically initialise the elements to values that the compiler has chosen. However, for correctness, RODIN demands that all variables be initialised. Arrays have to be initialised to some “dummy” values in a non-deterministic assignment. It would be possible to add some mechanism for initialising arrays that can be converted to C++ code.

Constants can be defined in contexts, by an identifier in the CONSTANTS field and a value assignment in an axiom. Constants are useful in parallel programming as they can be accessed simultaneously by several processes. Constants are translated into code as described in Figure 4.

Enumerated sets are defined in a context, by a set name in the SETS clause,

Event-B	C++
CONTEXT c **CONSTANTS** *const1* **AXIOMS** *const1 = value*	`const int const1 = value`

Figure 4: Translation of constants.

Event-B	C++
CONTEXT c **SETS** *Enum1* **CONSTANTS** *const1* *const2* **AXIOMS** *Enum1* = {*const1*,*const2*} *const1* $\neq$ *const2*	`enum Enum1 {const1, const2}`

Figure 5: Translation of enumerated set definitions.

where elements of the set are defined as constants. The set is then defined in an axiom that states which elements belong to the set. We also need to denote that all the elements in the set are different, in order to be able to distinguish them from each other. Figure 5 describes how enumerated sets are translated into C++.

A machine can use an enumerated set as defined above, if the context is listed in the SEES clause of the machine. Variables in the machine can then be type assigned to the enumerated set, either as a single set or an array of enumerated sets. Figure 6 describes translation of type assignment of enumerated sets.

The guards of an event are situated in the WHERE clause of an event. The

Event-B	C++
MACHINE m **SEES c** **VARIABLES** *var1* *var2* **INVARIANTS** *var1* $\in$ *Enum1* *var2* $\in$ 0..N $\rightarrow$ *Enum1*	`Enum1 var1` `Enum2 var2[N]`

Figure 6: Translation of enumerated set type assignments.

Event-B	C++
EVENT e	
WHERE	
guard1: *value1* $\neq$ *value2*	`value1 != value2`
guard2: *value1* $=$ *value2*	`value1 == value2`
guard3: *value1* $<$ *value2*	`value1 < value2`
guard4: *value1* $\leq$ *value2*	`value1 <= value2`
guard5: *value1* $>$ *value2*	`value1 > value2`
guard6: *value1* $\geq$ *value2*	`value1 >= value2`

Figure 7: Translation of guard predicates.

Event-B	C++
EVENT e	
WHERE	
guard_1: *value_11* $\bigoplus_1$ *value_12*	`(value11` $\bigoplus_1$ `value12 && ...`
$\vdots$	`&& value_n1` $\bigoplus_n$ `value_n2)`
guard_n: *value_n1* $\bigoplus_n$ *value_n2*	

Figure 8: Translation of multiple guards.

six different predicates allowed in Event-B0 are translated to code as described in figure 7, where "value" denotes a variable or a constant value. Multiple guards, in an event, are combined with *logical and*, represented by the operator "&&" in C++. In this way, all the guards are situated in a single predicate that has to be true in order for the event to be executed. Multiple guards are translated to code as described in figure 8, where $\bigoplus_1, ..., \bigoplus_n$ denotes the relational predicates in figure 7 and "$value_{ij}$" denotes a variable or constant value.

Substitutions are very similar in Event-B and C++. They are both denoted by an assignment operator. The left-hand side can contain a variable or an array element. The right-hand side consists of a variable, a value or an arithmetic operation. We do not allow non-deterministic substitutions in Event-B0. Figure 9 describes translation of variable and array substitutions.

Events in Event-B are translated into methods in the C++ class. They are implicitly indexed in the scheduler, so that the first event will have index 0 and

Event-B	C++
EVENT e	`variable = value`
THEN	`array(index) = value`
variable := *value*	
array[*index*] := *value*	

Figure 9: Translation of substitutions.

Event-B	C++

EVENT ev1
WHERE
guard_1
guard_2
⋮
guard_n
THEN
assignment_1
assignment_2
⋮
assignment_m
	`int Machine_Class::ev1_guard(void) {`
	`  if(guard_1 && guard_2 && ...  & guard_n)`
	`  return true;`
	`  else return false;`
	`}`
	`int Machine_Class::ev1(void) {`
	`  assignment_1;`
	`  assignment_2;`
	`  ⋮`
	`  assignment_m;`
	`}`

Figure 10: Translation of events.

the next event index 1, etc. The guard and the assignments are translated into two separate functions. This structure is required by the scheduler, as the master node checks the guards and the substitutions are executed on the slave nodes. The guard function has all the guards in a single *if-case*. If all guards are true, the function will return true, otherwise it will return false. Event translation is defined in figure 10.

Event parameters in Event-B have a parameter name defined in the ANY clause and its type defined in the WHERE clause. Every event parameter is represented by a separate machine variable in the C++ class. This decision was taken because the master node first creates the parameter and then sends it to a slave node. Parameters have a naming scheme associated with the event, in the form "eventname_parametername." Upon executing a parametrised event, the parameters have to be assigned new values that are either fetched or randomised. The code generator automatically creates an external function for the parameters in the Environment class, which is used for defining how the values should be created. Parameters are translated to code as described in figure 11.

It is also possible to include additional restrictions on the parameters. For example, the guard "$var1 < 10$" states that $var1$ also has to be less than 10. However, the parameter randomiser does not automatically generate a value that fulfils all the guards. It only determines the type of the parameter. Since a parameter can be restricted in an infinite amount of ways, it would require quite a sophisticated algorithm to automatically determine the range of values that the guards impose. Therefore, users have to define the parameter functions manually.

If a model is in Event-B0 form, then the code generator will translate it into two different C++ files that the scheduler can execute. The machine and the contexts are translated into the header file "Machine.h" and the source code file "Machine.cpp." The C++ class that represents the model is defined in the source code

Event-B	C++

EVENT ev1
ANY
*var*1
*var*2
*var*3
WHERE
*var*1 ∈ -2147483648..2147483647
*var*2 ∈ *BOOL*
*var*3 ∈ *enum*
	`int ev1_var1`
	`bool ev1_var2`
	`enum ev1_var3`

Figure 11: Translation of event parameters.

file. The interface to the model is also located in this file. Enumerated sets and constants that are defined in a context are situated in the header file. The code generator also generates two files for handling parameters: the header, "Environment.h," and the source code in the "Environment.cpp" file. Custom functions that create parameter values can be added to the latter file.

4 Scheduling

In order to execute the code generated by the plug-in, we have also developed a scheduling platform. This tool has its roots in a method given by Degerlund et al. [6] of correctly scheduling an action system in a parallel environment. Scheduling in this sense means assigning computational work to processes. As the Action Systems formalism targets parallel and distributed systems, its structure is readily suitable for parallel execution. The only rule that has to be followed is that events that have no variables in common can be executed in parallel. To execute an action system in parallel, we have to fulfil this criterion for correct behaviour. Degerlund et al. achieve this by applying mutual exclusion to the variables, so that they cannot be accessed by two or more events simultaneously. Events that need to access a variable that is currently in use by another event have to wait until it is freed up. Note that the terminology used in action systems theory is somewhat different from that of Event-B. For example, the word *action* has a different meaning in action systems (corresponding to *events* in Event-B) as compared to how it is used in Event-B. For clarity, we will stick to Event-B terminology, even when discussing theory with its roots in the Action Systems formalism.

Scheduling tool support. Degerlund et al. developed the proof-of-concept program *ELSA* that implements the described scheduling method for classical B models. As most of the research in the field has been theoretical, focus was put on implementation issues. ELSA can schedule classical B models of B0 form that

have been translated to computer code. Models are created with the Atelier B tool and then converted to C code by the built-in code generator. ELSA is written in C++ and utilises the MPICH2 communication library [17] for communication between nodes in a computer cluster. MPICH2 is an implementation of the Message Passing Interface (MPI) [16], which is the most dominant communication protocol used for parallel programming [9]. The MPI standard has been widely use in computer clusters and supercomputers. MPICH2 provides an interface for communication between computers by *message passing*.

The Event-B scheduler is based on the code of ELSA, but contains changes and added functionality needed for Event-B and our framework. The scheduler is written in C++ code and utilises the MPICH2 communication library for parallel computing. This set-up was considered to be suitable for our requirements. The scheduler takes as input an Event-B0 model, which has been translated to C++ code by the code generator. Upon execution, it schedules the events in parallel on a computer cluster or a computer with multiple computational cores or processors.

4.1 Background theory

The scheduler is based on the parallel interpretation of actions systems / Event-B stating that events that share no variables can be executed simultaneously. To schedule an event, two properties must hold:

1. The guard of the event is true.
2. No other events that share variables are currently being executed.

To avoid interference, we utilise mutual exclusion to prevent variables from being accessed simultaneously by several events. This is implemented by using a locking mechanism. An event in execution locks all variables involved in the event, so that no other events can access them. The variable locks are implemented by Boolean variables. If a variable is locked, then a corresponding Boolean locking variable will have the value true, and if the variable is not in use, the lock will instead have the value false. We only need to implement this mechanism on the master node, as it takes all scheduling decisions. The locks are implemented using a matrix that is created in two steps.

Variable-event matrices. When executing an action system or an Event-B model in parallel, we have to prevent several events that share variables from being executed simultaneously. To achieve this, we first generate the *variable-event* matrix, *ve*, of the size *n* * *m*, where *n* is the number of global variables and *m* is the number of events in the system. For every element ve_{ij}, where $i \in \{1..n\}$ and $j \in \{1..m\}$, in the matrix, we will have a Boolean value. If variable *i* is involved in event *j*, the element will have the value true and if it is not involved, it will have the value false. By "involved," we denote that an event accesses a variable. This matrix describes the dependencies between the events and the variables.

Event-event matrices. After the variable-event matrix has been created, we can create a scheme of the dependencies between events, with respect to the variables. This is described in the *event-event* matrix, *ee*, of size *m* * *m*, where *m* is the number of events in the system. This matrix also contains Boolean values and is derived from the *ve* matrix. Element ee_{ij} is true if there exists a $k \in \{1..n\}$, where ve_{ki} = ve_{kj} = true, otherwise it will have the value false. This means that if the events *i* and *j* both use variable *k*, they have a variable dependency and cannot be executed simultaneously.

Event-location matrices. We also have a third matrix, el (*event-location*), for specifying which events are allowed to run on the processors. An action system can be partitioned with respect to either variables or events [19], dividing them into disjoint sets. By partitioning a system with respect to events, every processor has a set of events that can be executed on it. This scheme is represented by the *el* matrix of the size *m* * *p*, where *m* is the number of events in the action system and *p* is the amount of processors used on a computer cluster. If element el_{ij}, where $i \in \{1..m\}$ and $j \in \{1..p\}$, is true, then event *i* is allowed to be executed on processor *j*. If el_{ij} is false, then it is not allowed to be executed on processor *j*. Every processor should have at least one event that is allowed to be executed on it, and each event should be executable on at least one processor. This matrix can, for example, be used to reserve a faster processor for a compute-intensive event.

Scheduler considerations. We have implemented the three above-mentioned matrices for our Event-B scheduler. The code generator calculates the two dependency matrices upon model translation. The *el* matrix can be defined in a configuration file of the scheduler. This matrix is optional for the scheduler. If it is not defined, all events can be executed on any processor.

The scheduler tries to schedule events to the nodes in the cluster in a *round-robin* fashion, as the execution order of an Event-B model is non-deterministic. It first checks that the guard of an event is true and that it is not currently being executed. Then it checks whether the event shares any variables with other events currently being executed. If this is the case, the event cannot be scheduled. Finally, it checks if there are any idle nodes that can execute the event. After a slave node has executed an event, it returns the updated values of all variables involved in the event to the master node, after which the variables can once again be accessed by other events.

When all guards of a machine are false, the scheduler is considered to be *deadlocked*. This is often a desired termination state. However, some systems will always have a guard evaluating to true and can therefore execute infinitely. This is often the case in reactive systems that are designed to execute forever. A system of this type has to be terminated manually. An Event-B machine is said to be deterministic if only one guard is true at any time. Such machines will not gain any speed from parallel execution, as only one event can be executed at once.

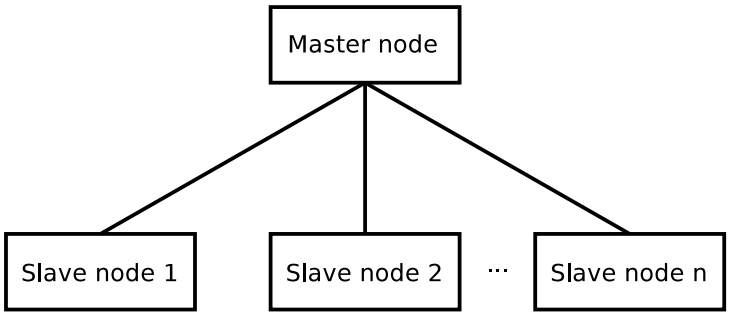

Figure 12: Relationship between master and slave nodes.

Fairness of events concerns the order of event execution. It is possible that an event can be blocked by other events in execution, either because of mutual exclusion or because all processors are constantly busy. In the former case, we have implemented some fairness, by always checking the guards in different order. Every time the scheduler iterates through the guards, it starts from a different one. Otherwise, an event could constantly disable other events by always being executed first. It is also possible to exercise manual control by editing the *el* matrix. For example, a specific processor can be reserved for a given event.

4.2 Operation of Scheduler

In our parallel execution of Event-B models, we have a central master process that schedules events to be executed on slave nodes, as illustrated in figure 12. The master node has the most up-to-date state of the Event-B model and it takes all scheduling decisions. Upon scheduling an event to a slave node, the master node sends all the variables involved in the event to it. The slave proceeds to execute the event and once finished, it sends back all the variables to the master node. Then the master updates the master state of the model and frees up the variables that were involved in the event.

4.3 Scheduler algorithms

The master node constantly schedules events to the nodes in a loop. It schedules any event that satisfies all the conditions described above. The main scheduling algorithm is expressed in pseudocode in figure 13.

After the scheduler has looped through all events once, it waits for a slave node that has finished execution to send back the results. If no events are being executed, then the program terminates. This algorithm is described in pseudocode in figure 14.

The slave nodes have a different algorithm that they constantly loop through. All the slave nodes continuously wait to be scheduled an event. After receiving an index of an event to execute, they receive the values of the variables involved in

```
WHILE any guard true
    DO iterate through events 1..n with i
    IF event i is already running
        THEN stop current iteration
    IF variables involved in event i in use
        THEN stop current iteration
    IF the guard event i is false
        THEN stop current iteration
    IF event i has parameters
        THEN randomise parameters
    IF idle nodes available
        THEN schedule event i to free node
    send variables involved in event i to the node
    IF event i has parameters
      THEN send parameters to the node
```

Figure 13: Master node scheduling algorithm.

```
IF any events are currently in execution
  THEN
  wait until any node has finished execution
  get index of executed event
  receive variables involved in the event
  free up variables involved in the event
ELSE send termination signal to all nodes
```

Figure 14: Master node receive results algorithm.

```
DO
  receive an index of scheduled event
  IF index equals termination signal
    THEN terminate
  receive the variables involved in the event
  IF event has parameters
    THEN receive parameters
  execute event
  signal master node that execution has finished
  send back the variables involved in the event
```

Figure 15: Slave node algorithm.

the event and any parameters it might have. It then proceeds to execute the event by using the variables it received. After finishing, it returns the updated variables to the master node and idles until it is scheduled another event. This algorithm is described in figure 15.

4.4 Scheduler logging facilities

We have implemented logging facilities for the scheduler, which can be used to write a log file of how the scheduling occurred. Degerlund et al. developed a log analyser in Java, which provides a graphical representation of the execution. We have, however, not implemented any tools for log analysing. The scheduler has the possibility to log the following events:

- Execution initialised / Amount of Nodes / Amount of events
- Event *i* scheduled / To node *j* / At iteration *x*
- Node *j* finished / Event number *i* executed successfully
- Program termination

By logging how many events that are scheduled for every iteration, it is possible to analyse the parallel properties of the model. If the system seldom has several events that can be executed in parallel, it cannot gain any significant speed improvement by parallel execution.

5 Example: Factorisation

We now show how our method can be used in practice using an integer factorization example. The scenario is that we have an integer *n*, of which we want to find the lowest factor (greater than 1). If *n* is a prime number, the result produced

should be *n* itself. The algorithm we use is trial division, which is in itself ineffective, but simple enough to show how our method works. The problem being *embarrassingly parallel*, it is also easy to share the tasks between different processors. The factorisation algorithm is performed by checking whether $n \bmod i$ is zero, for values of i starting from 2 counting up to $\sqrt{n}$. If a factor has not been found by then, it can be shown mathematically that n is a prime number. Since Event-B does not support square root, we instead use $n/2$ as our limit, which is correct, albeit not as efficient. As soon as a match is found, the algorithm terminates and the corresponding value of *i* is returned.

The parallel version of the algorithm works in the same way, except that it performs several trial divisions concurrently. Every process has a different set of divisors, with the first one performing the modulo operation with i and the second one with $i + 1$, the third one with $i + 2$ etc. Instead of increasing the i variable by one, we increase it with the process amount. In this way, every event uses a different set of values for i. By defining n as a constant in the context, it can be used by all events. Race conditions can not occur when accessing constant values concurrently. The algorithm would normally be performed by having the same function executed on every node, but with different parameters. Such algorithms cannot be modelled in Event-B, so instead, several different events that perform the same computations can be created. One drawback with this is that a dynamic number of threads cannot be used to execute the algorithm.

Test case modelling. In our test case, we have modelled parallel trial division with three different events that perform modulo operations simultaneously. For every event, there are three associated variables: an $i_{process}$ variable that is used in the modulo operation, a $result_{process}$ variable for storing the result of the modulo operation, and a Boolean variable called $continue_{process}$ for controlling program execution. The $process$ subscript denotes a process number, which in our model is 1, 2 or 3. The variable $i_{process}$ is increased by 3 after every trial division, so that all events have a different set of values for $i_{process}$. The $continue_{process}$ variable has to be true for the trial division to be carried out, and after execution, it is set to false by the event. A fourth event, called $check$, sets all the $continue$ Booleans to true if no divisor to n has been found, and if is there is still a possibility to find one. The program continues until the smallest divisor to n has been found, or until each counter $i_{process}$ has exceeded $n/2$. In the latter case, n is a prime, as it has no other factors other than 1 and itself.

Results. We executed the model successfully, with different values for n, on a computer with eight cores. The algorithm was modelled as one machine and one context, both in Event-B0 form. The model was automatically translated to C++ code by our code generator. By computing and examining the *event-event* matrix of the code, we can see that events 1 to 3, the trial division events, share no variables. Hence, they can be executed in parallel. Event number 4 is the event

that checks the result. It is involved with the variables of events 1 to 3 and can therefore not be scheduled while any other events are running. A slightly amended version of the model can be found in appendix A.

6 Conclusions and future work

In this paper, we have proposed an approach to code generation using the Event-B formalism, as well as scheduling and execution of the resulting code. Software can be modelled and refined in the established Event-B tool (the RODIN platform) in the standard way, but the last refinement step has to comply with Event-B0, which is a subset of the Event-B language. Event-B0 is inspired by the B0 language of the Atelier B tool used for the classical B method, an it only contains constructs that can be easily translated into C++. While the correspondence between the two languages has not been formally proven, Event-B0 has been carefully designed to contain only constructs that have a very close correspondence to C++ code. Events are translated to C++ methods, and the constructs allowed in the events are restricted. Translation from Event-B0 to C++ has been implemented as a plug-in for the RODIN platform.

We have also developed a scheduler, written in C++, that is used to execute the generated events (methods). Execution adheres to the commonly used behavioural semantics of Event-B, in which enabled events are non-deterministically chosen for execution, and the program terminates when all events are disabled. The scheduler makes use of the fact that events are assumed to be atomic, and it is therefore possible to schedule events in parallel, given that all events executed in parallel have no variables in common. The scheduler uses the MPI (Message Passing Interface) framework to schedule events on different processors in a network, or on different cores of a multi-core processor.

Related and future work. Code generation for Event-B and related formalisms has also been studied elsewhere. The Atelier B tool for the classical B method defines a B0 language [5], from which its code generator is able to produce C/C++ or Ada code. The intermediary B0 language serves the corresponding purpose as Event-B0 in our work, and Event-B0 is in fact inspired by B0. Atelier B does not take a stand on a behavioural semantics, and the operations (cf. events in Event-B) are simply translated into functions in the target language, but have to be explicitly called upon by the programmer. Consequently, Atelier B does not provide a scheduler, nor does it explicitly take a stand on concurrent execution. A scheduler that can be used with (slightly modified) code generated from Atelier B has, however, been developed by Degerlund et al. [6]. This scheduler is intended for use in B Action Systems, and it also constitutes the code base for the scheduler of this paper.

A code generator for Event-B similar to ours has been developed by Wright

[22]. It was, however, developed for the purpose of a virtual machine project, and was not intended to be an all-round tool. Our code generator can be seen as an extension, taking Wright's work one step further towards a general tool. A similar tool, EB2ALL, with C, C++, C# and Java code generation has been developed at Loria [15]. To our knowledge, it focuses strictly on code generation, and does not take a stand on scheduling. Edmunds has suggested an Event-B code generation approach [7] in which the developer can express control flow information in a language called OCB (Object-oriented Concurrent-B). The target language is Java, and concurrent execution is supported. This method gives the developer more control over the execution of the final program, which is sometimes a desired feature. It is, however, a different design philosophy as compared to ours, where we rely on the established Event-B behavioural semantics and let the scheduler automatically take care of the scheduling in such a manner that adheres to the semantics. This approach has also been further developed by Edmunds and Butler and adapted to ADA code generation [8]. Another approach to Event-B scheduling has been proposed by Boström [3]. Boström's work, which focuses on sequential programs, relies on explicit schedules given in a scheduling language, and proposes a pattern-based approach to showing the correctness of imposing a given schedule on an Event-B model. Related methods have been suggested by Iliasov [13] and Hallerstede [11]. An approach that introduces support for concurrent programs has also been suggested by Boström et al. [4].

The automated "on-the-fly" scheduling of our approach has the advantage that it very closely preserves the usual behavioural semantics of the models. It also facilitates for the developer, since the execution order of the events is automatically decided by the scheduler during run-time. On the other hand, the lack of explicit control flow also poses a challenge. Since the events are scheduled one by one on different processors/cores, the communication overhead is sometimes large. Future work includes evaluating how this affects the practical use of our method, and also investigating means of (automatically) scheduling groups of events, or repetitive execution without involving the scheduler. The challenge constitutes achieving this in a way that adheres to the standard behavioural semantics and, thus, would not require the introduction of explicit control flow structures.

References

[1] J.-R. Abrial. *Modeling in Event-B - System and Software Engineering*. Cambridge University Press, 2010.

[2] R.J.R. Back and R. Kurki-Suonio. Decentralisation of process nets with centralised control. In *Proc. of the 2nd ACM SIGACTS-SIGOPS Symp. on Principles of Distributed Computing*, pages 131–142, 1983.

[3] P. Boström. Creating sequential programs from event-b models. In *Proceedings of the 8th international conference on Integrated formal methods*, IFM'10, pages 74–88, Berlin, Heidelberg, 2010. Springer-Verlag.

[4] P. Boström, F. Degerlund, K. Sere, and M Waldén. Concurrent scheduling of event-b models. In *Proceedings 15th International Refinement Workshop*, pages 166–182, June 2011.

[5] ClearSy. B language reference manual version 1.8.6. http://www.atelierb.eu/ressources/manrefb1.8.6.uk.pdf.

[6] F. Degerlund, M. Waldén, and K. Sere. Implementation issues concerning the action systems formalism. In *Proceedings of the Eighth International Conference on Parallel and Distributed Computing, Applications and Technologies*, PDCAT '07, pages 471–479, Washington, DC, USA, 2007. IEEE Computer Society.

[7] A. Edmunds. *Providing Concurrent Implementations for Event-B Developments*. PhD thesis, University of Southampton, 2010.

[8] A. Edmunds and M. Butler. Tasking Event-B: An extension to Event-B for generating concurrent code. In *PLACES 2011*, 2011.

[9] T. Gangadharappa, M. Koop, and D. K. Panda. Designing and evaluating mpi-2 dynamic process management support for infiniband. In *Proceedings of the 2009 International Conference on Parallel Processing Workshops*, ICPPW '09, pages 89–96, Washington, DC, USA, 2009. IEEE Computer Society.

[10] R. Grönblom. A framework for code generation and parallel execution of event-b models. Master's thesis, Åbo Akademi University, 2009.

[11] S. Hallerstede. Structured Event-B models and proofs. In *Abstract State Machines, B and Z*, volume 5977 of *LNCS*, pages 273–286. Springer-Verlag, 2010.

[12] S. Hallerstede. On the purpose of event-b proof obligations. *Form. Asp. Comput.*, 23:133–150, January 2011.

[13] A. Iliasov. On Event-B and control flow. Technical Report CS-TR-1159, School of Computing Science, Newcastle University, 2009.

[14] B. W. Kernighan and D. Ritchie. *The C Programming Language*. Prentice-Hall, 1978.

[15] Loria. EB2ALL. http://eb2all.loria.fr/.

[16] Message passing interface forum. http://www.mpi-forum.org/.

[17] MPICH2 website. http://www.mcs.anl.gov/research/projects/mpich2/.

[18] RODIN platform website. http://www.event-b.org.

[19] K. Sere. *Stepwise Derivation of Parallel Algorithms*. PhD thesis, Åbo Akademi University, 1990.

[20] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3rd edition, 2000.

[21] M. Waldén and K. Sere. Reasoning about action systems using the b-method. *Form. Methods Syst. Des.*, 13:5–35, May 1998.

[22] S. Wright. Using eventb to create a virtual machine instruction set architecture. *Abstract State Machines, B and Z*, pages 265–279, 2008.

A Factorisation model

MACHINE TrialDiv

SEES TrialDiv_Context

VARIABLES

i_1

i_2

i_3

result_1

result_2

result_3

continue_1

continue_2

continue_3

INVARIANTS

inv1 : $i\_1 \in -2147483648 \mathinner{..} 2147483647$

inv2 : $i\_2 \in -2147483648 \mathinner{..} 2147483647$

inv3 : $i\_3 \in -2147483648 \mathinner{..} 2147483647$

inv4 : $result\_1 \in -2147483648 \mathinner{..} 2147483647$

inv5 : $result\_2 \in -2147483648 \mathinner{..} 2147483647$

inv6 : $result\_3 \in -2147483648 \mathinner{..} 2147483647$

inv7 : $continue\_1 \in BOOL$

inv8 : $continue\_2 \in BOOL$

inv9 : $continue\_3 \in BOOL$

inv10 : $n \geq 4$

inv11 : $n \leq 2147483647$

inv12 : $n/2 + 3 \leq 2147483647$

EVENTS

Initialisation

begin

act1 : $i\_1 := 2$

act2 : $i\_2 := 3$

act3 : $i\_3 := 4$

act4 : $result\_1 := -1$

act5 : $result\_2 := -1$

act6 : $result\_3 := -1$

act7 : $continue\_1 := TRUE$

act8 : $continue\_2 := TRUE$

act9 : $continue\_3 := TRUE$

end

Event *process1* $\widehat{=}$

when

grd1 : $continue\_1 = TRUE$

grd2 : $i\_1 \leq n/2$

then

act1 : $result\_1 := n \ mod \ i\_1$

act2 : $i\_1 := i\_1 + 3$

act3 : $continue\_1 := FALSE$

end

Event *process2* $\widehat{=}$

when

grd1 : $continue\_2 = TRUE$

grd2 : $i\_2 \leq n/2$

then

act1 : $result\_2 := n \ mod \ i\_2$

act2 : $i\_2 := i\_2 + 3$

act3 : $continue\_2 := FALSE$

end

Event *process3* $\widehat{=}$

when

grd1 : $continue\_3 = TRUE$

grd2 : $i\_3 \leq n/2$

then

act1 : $result\_3 := n \ mod \ i\_3$

act2 : $i\_3 := i\_3 + 3$

act3 : $continue\_3 := FALSE$

end

Event *check* $\widehat{=}$

when

grd1 : $result\_1 \neq 0$

grd2 : $result\_2 \neq 0$

grd3 : $result\_3 \neq 0$

grd4 : $i\_1 \leq n/2 \vee i\_2 \leq n/2 \vee i\_3 \leq n/2$

then

act1 : $continue\_1 := TRUE$

act2 : $continue\_2 := TRUE$

act3 : $continue\_3 := TRUE$

end

END

CONTEXT TrialDiv_Context

CONSTANTS

n

AXIOMS

axm1 : $n = 479001599$

END

Turku
Centre *for*
Computer
Science

Joukahaisenkatu 3-5 B, 20520 Turku, Finland | www.tucs.fi

University of Turku
- Department of Information Technology
- Department of Mathematics

Åbo Akademi University
- Department of Information Technologies

Turku School of Economics
- Institute of Information Systems Sciences

ISBN 978-952-12-2685-4
ISSN 1239-1891