Noname manuscript No.
(will be inserted by the editor)

Tasking Event-B: A Code-Generation Extension for Event-B

Andrew Edmunds · Michael Butler

Received: date / Accepted: date

Abstract Event-B is a formal set-theoretic approach to modelling systems in the safety-critical, and business-critical, domains. This article describes an extension to the Event-B approach, called Tasking Event-B, which facilitates automatic generation of source code from annotated Event-B models. Tasking Event-B allows specification of multi-tasking implementations. We believe that automatic code-generation makes a useful contribution to the Rodin tool-set, by contributing a link in a coherent tool-chain. Automatically generating code from the Event-B model can be seen as a productivity enhancement. It removes a source of errors, that of manually coding for each development. To validate the approach we have undertaken case studies and taken part in an industrial collaboration. We present a number of case-studies to illustrate our work, in this article.

Keywords Formal Methods · Event-B · Code Generation · Concurrency · Embedded Systems

Andrew Edmunds
School of Electronics and Computer Science,
University of Southampton, UK
E-mail: ae2@ecs.soton.ac.uk

Michael Butler
School of Electronics and Computer Science,
University of Southampton, UK
E-mail: mjb@ecs.soton.ac.uk

1 Introduction

Event-B [6] is one of a number of formal methods that may be used to model systems where a high degree of reliability is required. Event-B was inspired by its predecessor, *Classical-B* [3]. It is a modelling language, used with a supporting tool platform, Rodin [4]; so named from the project in which it was developed [29]. Further work has been undertaken in the DEPLOY [39] project, to assess the approach with a number of the industrial partners. To derive the greatest benefit from the formal modelling approach, it is desirable to use the formal modelling artefacts to generate implementations for, at least, some parts of the system being modelled. During the latter stages of the DEPLOY project, automatic generation from Event-B models began to evolve. The work presented here describes how we we link Event-B to some frequently used programming languages, with the aim of introducing automatic code generation for modellers using Event-B and Rodin. In particular we target multi-tasking, embedded control systems, since this is relevant to the evolution of the project work.

We chose the Ada [9] programming language as a basis for the code generation approach. This was not only because of it's suitability for the application domain, but also because the programming constructs are well-considered programming abstractions. Ada maps well to Event-B modelling elements, which is easy for modelling, and simplifies the translation to code. We do not attempt to model all aspects of the system, e.g. time; but we model the evolution of the system state, and are able to specify the properties, relating to the state, that are the most important for system safety. In the *Advance* project [38], we are concerned with modelling and co-simulation of Cyber-physical systems. We can, again, make use of automatic code-generation techniques to provide implementations for use in co-simulation.

We believe that our code generation approach provides a lightweight/streamlined solution for adding implementation-level detail (such as that required for describing atomic state-updates with procedure calls, and communication between 'tasks' via monitor-style protected objects). We describe it as lightweight, since we have a small language extension, that describes the additional, required implementation details; and streamlined, since it fits well with the existing Event-B tools and methodology. In the approach, we make use of a number of plug-in features. Such as a modular decomposition approach for, handling complexity, and representing communication between the various elements in the implementation. We also use a Theory extension, that allows the Event-B language to be extended with new data types, operators, and target translations.

1.1 Previous Work

Initially, development of code-generation from Event-B began from an object-oriented perspective, with an implementation-level notation for object-oriented,

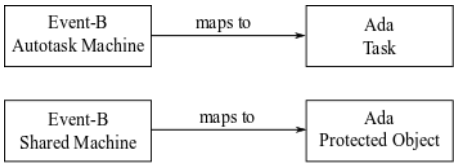

Fig. 1 Mapping between Event-B and Ada

concurrent-B (OC-B) [17]. The focus of this work was to use Event-B to model and generate code for safe multi-tasking with object-oriented languages. We targeted the Java [21] language. We found, however, that this approach gave rise to a notation where the semantic gap between Event-B and the OC-B was large. This was not ideal for developers of Event-B models, who then wanted to write implementation-level specifications from which code could be generated. We found that the models quickly became large and intractable; we recognized the need for decomposition of models into smaller units. At that time, Event-B tools were at an early stage of development, and machine decomposition was still in development.

We decided to modify the approach, to use the newly developed decomposition approach [33,34], and use an implementation-notation more closely related to Event-B (than OC-B, which was an object-oriented notation). The focus, then, became generating safe concurrent implementations, rather than targeting object-oriented technology, per se. This gave rise to our approach, known as Tasking Event-B. Our interest in safe, concurrent implementations drove us towards the Ada tasking model. At the implementation-level, an Event-B model is effectively a detailed model of an implementation. We found that the Ada [9] programming language provides clearly defined constructs, that map well to Event-B, see Fig. 1. Ada tasks and protected objects can be modelled by Event-B machines in a one-one mapping. We distinguish between the two entities by annotating the machines with the *autotask* and *shared* keywords. We describe the mapping in more detail in Sect. 4.5.

In Sect. 2 we introduce the core Event-B features to the reader, and compare Event-B with some other formal approaches. In Sect. 3 we provide an overview of some additional Event-B features required for code generation, and discuss some of the potential target programming language targets.

In Sect. **??** we... In Sect. **??** we...

2 Event-B Modelling

The formal methods related to the work presented here can be categorized as state-based formal methods. Alternative, but not unrelated, approaches are categorized as process-based methods. Classical-B [3,7,13,14] and its successor, Event-B are said to be state-based, since they focus on modelling the changes of state, not the behaviour of processes. In Classical-B, state updates are modelled by guarded operations, where the operation is an analogue of

a procedure call in a programming language. In Event-B, state updates are modelled by guarded events, providing a more abstract view of the way a system evolves. Event-B can be used to model systems at an abstract level; and by adding more detail (using a technique called refinement) it can model the software aspects of systems too. Both methods are set theoretic modelling approaches that incorporate a notion of proof to show that important system properties are maintained. The former is primarily an approach to software systems development, the latter more widely applicable to system-modelling. In an effort to make modelling and proof easier, Event-B was developed to overcome some of the difficulties encountered when using in Classical-B. The main differences between Classical and Event-B are highlighted in [22], and inspiration was also drawn from action systems [8].

It is fair to say that Event-B is not just a formal modelling language; the name is used to describe both a notation, and a methodology. In addition to this a mature tool-platform called *Rodin*, named after its development programme, complements the methodology. The main modelling components of Event-B are contexts and machines. Contexts are used to model static features using sets, constants, axioms, and theorems. Machines are used to model variable state using *variables*. A third, more recent addition, is the Theory component; where a developer can augment the bundled mathematical language, and rule-base, with new (inference and re-write) rules, data types, and operators. During the modelling process, changes to the components result in automatic generation of proof obligations, which must be discharged in order to show that the development is consistent. The proof obligations generated in classical-B are often complex, the Event-B approach results in simpler proof obligations as described in [22], since Event-B consists of a simplified action syntax, giving rise to simpler proof obligations. A further simplification was made by adopting an event-based approach, where each atomic event has a predicate guard and an action consisting only of assignment statements. Events correspond to operations in the B-method; operation specification was more expressive, and included constructs for specifying operation preconditions (as part of its Design by Contract approach), operation calls, return parameters, and more complex structures for branching and looping. These constructs are not features of Event-B. Due to these simplifications (and more efficient proof tools) a large number of the proof obligations may be discharged automatically, by the automatic provers. Where un-discharged proof obligations remain, the user has, at their disposal, an interactive prover. Various techniques can be applied, to discharge the proof obligations, such as adding hypotheses; or making use of the hyperlink-driven user interface, for rule and tactic application. In the early stages of development with Event-B, a developer will begin by abstracting, and modelling, the observable events occurring in a system. Event-B, as the name suggests, takes an event-based view of a system; where events occur spontaneously from the choice of enabled events. An event is said to be enabled when the guard is true, and the state updates, described in the event actions, can take place; otherwise it is disabled, and none of its updates can occur.

2.1 An Event-B Example

An example of an Event-B machine can be seen in Fig. 2. It shows an abstract model of a pump controller, used in one of the case studies. We will use this model to describe some features of Event-B. But first we introduce the case study, which models a discrete *pumpController*. The model describes a system where the controller receives a value for the level of fluid in a tank. The variable *e_level* represents the value in the environment, and *c_level* models the value at a port in the controller. The value at the port is stored internally in *c_level_internal*. We adopt the prefix *e_* and *c_* throughout, to model environment and controller variables resp. and the suffix *_internal* to represent the controllers view of the environment. Reactive interactions between controller and environment are described as requests or commands. A Boolean value *e_pumponReq* represents an operator's request to turn the on pump. Based on the inputs to the controller, a command to turn the pump on may be issued, or a warning issued (and no command issued) if a minimum level *MIN* is not satisfied. In Fig. 2, we see that machine *M1* refines another machine *M0*; we

```
MACHINE m2 REFINES m1 SEES ctx
VARIABLES                c_level, e_level, c_pumpOnReq, e_pumpOnReq,
  c_pumpOnCmd, e_pumpOnCmd, c_warn, e_warn,
  c_level_internal, c_pumpOnReq_internal, c_pumpOnCmd_internal
INVARIANTS
 (c_level_internal ≤ MIN ∧ c_pumpOnReq_internal = TRUE ∧
    commit = TRUE ⇒ c_warn_internal = TRUE)
 ∧ (c_level_internal > MIN ∧ c_pumpOnReq_internal = TRUE ∧
    commit = TRUE ⇒ c_pumpOnCmd_internal = TRUE)
 ∧ (c_level_internal ∈ ℤ)
 ∧ (c_pumpOnReq_internal ∈ BOOL) ...
EVENTS
INITIALISATION c_level := 100 ‖ e_level := 90 ‖ c_pumpOnReq :=
FALSE ‖ ...
EVENT getLevel_eAPI REFINES getLevel_eAPI
 ANY p1
 WHERE p1 = e_level
 THEN c_level := p1
 END
...
```

Fig. 2 An Event-B Pump Controller Model

will discuss refinement in Subsect. 2.2. It also has a *SEES* clause; this makes the contents of a context visible to a machine. Contexts may contain sets, constants, axioms and theorems, and example context can be seen in Fig. 3. There are variables representing the internal state of the controller, and in-

```
CONTEXT ctx
CONSTANTS MIN
AXIOMS MIN = 10
```

Fig. 3 An Example Context

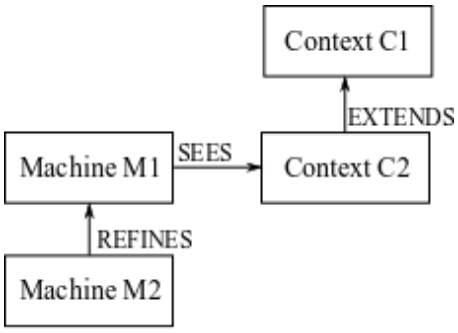

Fig. 4 Refinement and Extension

variants providing type information for variables. Invariants are also used to describe the safety properties of the system. This describes a required safety property, that if the level is at or below *MIN*, and a user's pump-on request is detected, then a warning will be issued. Also, if the level is OK and a pump-on is requested, then the state $pumpOnCmd = TRUE$ is set. Following the *INVARIANTS* clause are the model's *Events*. The *Initialisation* event is special event, since it has no guards. The initialisation event of a machine must occur before any other event in the machine is enabled. The event in the figure has a parameter p, in the *ANY* clause. Parameters can be used to represent information flow, in and out of events, or they can represent a *local* variable within the scope of the event. The event guard is defined in the *WHERE* clause, in the example, where p is typed as a Boolean. The guard relates the parameter to a machine variable $c\_pumpOnCmd$, in the predicate $p = c\_compound$. The event action appears in the *THEN* clause, where the parameter is assigned to the variable $m\_pumpOnCmd$, in the expression $m\_pumpOnCmd := p$.

2.2 Refinement and Extension

As we mentioned earlier in the section, Event-B makes use of a technique called refinement, where a machine can be refined by another. Fig. 4 shows this relationship. The refined machine is augmented with state variables, events and invariants, to provide a more detailed specification satisfying the properties, specified in the invariants, of the abstract specification. The counterpart to refinement of machines, is extension of contexts. It is then possible to build upon pre-existing contexts, using the *EXTENDS* clause, by adding more sets, constants, axioms and theorems. When a machine *SEES* a context, the contexts that it extends are also accessible to the machine. In a refinement, new variables, events and invariant properties can be added. Existing events can be modified, but in a restricted manner. Machine refinement is transitive and leads to a hierarchical structure. Refinements are related to their more abstract counterparts in such a way that, a valid refinement always satisfies the specifications higher in the refinement hierarchy. In this way, important sys-

tem properties can be specified at a high level of abstraction, and maintained down through the refinement chain. The Event-B tools are responsible for generating the proof obligations relating to refinement; these must be discharged in a similar way to those generated for proof of machine consistency. It is often necessary to specify a linking invariant, to describe the relationship between the variables of the abstract and refinement machines. Inspection of the proof obligations can assist in this task since some of the un-discharged proof obligations provide information about this link. In some cases we may model entities in an abstraction that are defined in the event parameters; and in the refinement these entities may be introduced to the model as machine variables. To assist with the proof effort we have a slightly different strategy to that of refining abstract variables with concrete variables. We link the parameters of the abstract event, with their concrete counterpart, using a $WITNESS$. This construct is a predicate describing the relationship between an event parameter (that disappears) from an abstract model, and the corresponding (refining) variable in the concrete model. Another feature of Event-B is the ability to refine one atomic event with a number of events, thus breaking the atomicity, as described in [12]. Eventually, at the end of a refinement chain the models are detailed enough to accurately describe an implementation. But Event-B is a modelling language, and there is a disjunction between the description of the system in Event-B, and commonly used programming languages, such as Java [21], C [25] and Ada [9]. Addressing the semantic gap between Event-B and programming constructs is a contribution of this article.

3 More Event-B Features

Event-B is based on the Rodin, open-source, project. There are two distinct sets of plug-ins: firstly there are a set of core plug-ins, which is mostly maintained, and coordinated, by a commercial organization; and there are a number of open-source plug-ins, some maintained by the commercial organization, others by academic institutions, or jointly. Section. 1 gave details of Event-B features provided by the core plug-ins. In this section we describe some important plug-in from the second category, namely Composition, Decomposition and the Theory plug-ins features.

3.1 Decomposition and Composition

Decomposition and composition are two related approaches, that we use to partition a system, to allow us to work on smaller, manageable sub-models. Figure 5 illustrates the shared-event decomposition approach [33] which we make use of in our code generation approach. An alternative shared-variable approach is described in [5]. *v1* and *v2* are disjoint sets of variables, and p and q are disjoint sets of parameters. g and a are guards and actions that range over the variables and parameters. In the shared-event decomposition approach the system is partitioned so that each variable is allocated to a single machine. In Fig. 5, the variables of machine m are partitioned into the sets $v1$ and $v2$ and decomposed into m_a and m_b respectively. The decomposed events have guards and actions which involve their respective variable partitions. The *composed machine* construct contains references to the decomposed machines, and the synchronizing events. It keeps track of the refinement relationship between the abstract machine and the decomposed machines; and the abstract events and their refinements in the decomposition. It is the synchronization of the refining events, across multiple decomposed machines, that refines a single abstract event.

The main purpose of using this form of event decomposition is that it reduces the size of the models, therefore making modelling and proof easier. The decomposed machines can be refined without restriction. For code generation, the synchronization of events provides a suitable basis for modelling procedures and procedure calls.

3.2 Theories

The theory plug-in provides a mechanism for extending the Event-B proof capabilities, and addition of new mathematical types [27]. Proof obligations will be generated to verify the soundness of the augmented prover. We can make use of the following sections in the theory plug-in.

1. Type Parameters: A theory can define type parameters to be used as polymorphic types.

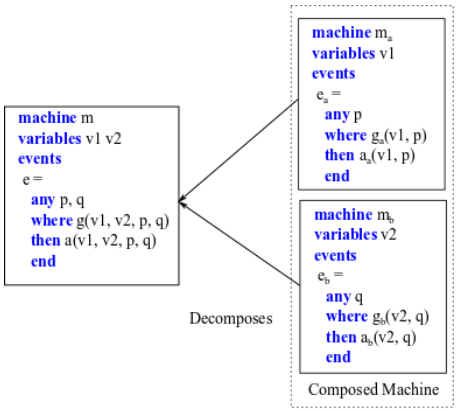

Fig. 5 Decomposition

2. Datatypes: Used to define simple data types, which can be added using a type constructor and element constuctors.
3. Operators: The operators section can be used to define polymorphic operators, such as the sequence described as an ordered list [27].
4. Axiomatic Definitions: These are defined to produce types, when no suitable type constructors or datatypes can be used as a basis for construction.
5. Theorems: Polymorphic theorems can be used to assist with the proof of newly introduced definitions.
6. Rewrite rules: Rewrite rules define how to rewrite formulas to equivalent forms. When a rewrite rule is defined, the author specifies whether the rule should be applied automatically, or during an interactive proof session.
7. Inference rules: Rules are matched against sequent goals. If a match is found, a backward proof step is performed. The rule may also match a hypotheses of a sequent, where a forward proof step is performed.

3.3 Background for Code-Generation

In the section so far we have looked at a number of other features of Event-B that will be used in the code generation approach. Decomposition provides the basic structuring of a development, that makes an Event-B development amenable to our code generation approach. We also make use of the theory plug-in by extending it to allow definition of translations from the Event-B mathematical language, to programming language mathematical expressions and conditions. We can also introduce more concrete data types, such as arrays, and provide translation details for these.

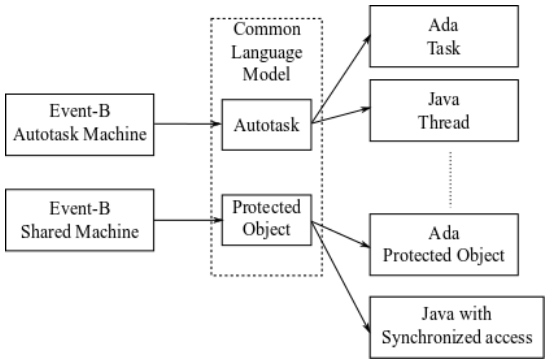

Fig. 6 The Common-Language Model

3.4 Targets for Code-Generation

In this subsection, we provide details of the target languages that we wish to generate code for. Ada provided the original basis for the code-generation approach described in [18] because its programming constructs are well defined and map well to Event-B. Ada tasks are processing units, known to the run-time system. Protected objects provide an encapsulation mechanism, that prevents interference in situations where data must be shared between concurrently executing tasks. In Java the processing unit is provided by Thread class, and encapsulation is provided by synchronized methods and blocks. In C, there is some choice about the method of thread implementation, such as OpenMP [41] threads, or POSIX threads[1]. In its most basic form, the encapsulation mechanism for protected regions must be coded explicitly. Our final code-generation target arises from our work on co-simulation of Cyber-Phsyical systems in the Advance project [38]. In this approach we generate a specialized form of C code, conforming to the FMI standard for model co-simulation [40]. This can be used to simulate the software running on an embedded controller, in a simulation of its environment. In order to provide some commonality in the code generation process we formulated a two-stage code-generation process, as shown in Fig. 6, which is a modified from Fig. 1; where the first step is a translation to a Common Language Model (CLM), which is an abstraction of the required implementation. The second stage is a back-end processor, translating to the various target implementation languages. The semantics of the common-language model are formulated using Event-B, and provides a formal basis for the translation. We will discuss the semantics of Tasking Event-B in Sect. 4.

4 Tasking Event-B

In this section we begin our presentation of Tasking Event-B. We begin by describing the Event-B semantics of the notation. An Event-B model is an abstraction of a system, the evolution of the state is described using events. The guard of an event describes the conditions which must hold prior to an event being enabled. The actions describe the values of the state variables after the event has occurred. Proof obligations are generated, to ensure that the actions update the state with values that satisfy invariant. As a result, a system that is consistent, can be described using events that do not impose an ordering: since in Event-B any enabled event can occur.

Now, in an implementation-level specification wish to reason about task-like behaviour, with implementation constructs, such as sequencing. So we introduce a notation for the purpose of ordering of events. The notation can be used to assist with code generation, since it allows the developer to introduce an ordering where none might exist. In the case that such an ordering does exist in the abstraction, the developer can generate a new Event-B model from the notation, and show that it a refinement of the more abstract one. When considering translation to code, it is usually necessary to work with a subset of implementable Event-B constructs. We consider *implementable constructs* to be those that are available in (or map well to) a programming language. We would therefore usually not consider non-deterministic assignment to be implementable, for instance, and add a restriction; that these are 'refined out' of the implementation-level model. Annotations are added to both machines and contexts. As well as assisting with code generation, the annotations are used to generate an Event-B model of the implementation. In Subsect. 4.1 we introduce the notation for specifying control flow, and provide Event-B, and implementation, semantics for this. In Subsect. 4.5 we discuss how we relate tasks, and protected objects, to Event-B machines.

In the following discussion, we distinguish between a task's behaviour, such as it's life-cycle (which is not modelled formally) necessary for implementation; and the updates to state (which are modelled formally). Our notation allows specification of both, formal and non-formal, aspects of a specification. An example of a non-formal aspect, is the task type which might be periodic or one-shot. Specifying that a task should repeat every so often, or happen just once.

4.1 Flow Control for Events

To enable us to impose an order on events we introduce *Tasking Event-B*, a textual extension to standard Event-B, with the following syntax for a task

body t,

$$
\begin{aligned}
t ::= & \\
& event \\
& \mid t \ ; \ t \\
& \mid IF \ event \ [ELSEIF \ event \ ]^* \ ELSE \ event \ END \\
& \mid WHILE \ event \ END
\end{aligned}
\tag{1}
$$

To be well-formed, events may only be referred to once in a task body. The semi-colon performs the role of a sequence operator, and we have clauses for branch and loop respectively. The notation $[\ ]^*$ represents an option of zero or more; in this case referring to sub-branches.

The first notion we will look at is sequential ordering using the semi-colon ';' sequence operator. In a model with two events *evt1* and *evt2*, we can write *evt1;evt2* to specify a sequential ordering. We can provide Event-B semantics for the ordering, by introducing an abstract program counter to the model. An enumeration of constants models the abstract program counter values; one per event. An abstract program counter variable models the currently enabled event. Guards enforce the ordering, and actions update the abstract program counter. An example can be seen in Fig. 7, where the program counter constants are *evt1* and *evt2*, *pc* has type *pcValues*, and the program counter enumeration is defined by a partition, using the Event-B partition operator, so $partition(pcValues, \{evt1\}, \{evt2\}, \{term\})$. The partition means that the values of the enumeration are distinct, i.e. $evt1 \neq evt2$ and so on. *term* is label indicating a final state where no event is enabled. Initially $evt1$ is enabled and $evt2$ is disabled, since $pc = evt1$. The update action A_1 occurs, with the program counter being set *evt2*. This, in turn, enables $evt2$, and so on.

4.2 Translating Sequences of Events to Event B

We will now define some translation functions for translating Tasking Event-B to Event-B. These functions effectively add program counter guards, and actions to the event. When translating sequences of events, we assume that events have no guards other than the program counter guards. To add the guards and actions modelling a program step, seen in Fig. 7, we define a translation TEB_pc. It takes, as parameters, a Tasking Event-B construct, and the next program counter name. It returns a set of events. The TEB_{pc}

Variables evt1, evt2
Invariant pc $\in pcValue$
Initialisation = pc := evt1
evt1 = **WHEN** pc = evt1 **THEN** A_1 ‖ pc := evt2 **END**
evt2 = **WHEN** pc = evt2 **THEN** A_2 ‖ pc := term **END**

Fig. 7 Sequence

function is polymorphic on its inputs, with a different function application for each Tasking Event-B construct.

$$TEB_{pc} \in TEB \times Name \to POW(EVENT)$$

We use a function *pcName* to extract an event name from an event; the name is used as a program counter variable. When TEB_{pc} is applied to a sequence, it returns two events, with TEB_{pc} applied to the individual events.

$$\begin{array}{l} TEB_{pc}(e_1; e_2, x) \rightsquigarrow \\ \{\ TEB_{pc}(e_1, pcName(e_2)), \\ \ \ TEB_{pc}(e_2, x)\ \} \end{array}$$

The TEB_{pc} function is applied to each event in the returned set. We now represent an event, with guards and actions, using the notation $g \to a$; where g is the guard, and a is the action.

$$\begin{array}{l} TEB_{pc}(e_1, x) \rightsquigarrow \\ \{\ e_1 \triangleq pc = pcName(e_1) \to (a_1 \parallel pc := x)\ \} \end{array}$$

This returns an updated event, with the additional guards and actions modelling a program pc.

4.3 Branching

To introduce branching to Tasking Event-B we consider the simple case first, where we have two events $e_1 = g \to a_1$ and $e_2 = \neg g \to a_2$. We add to the syntax, IF e_1 ELSE e_2 END. The translation function takes the branch as an argument, adds the program counter information, and returns the set of updated Events. We assume that a proof obligation is generated, whereby we can show that the guards are disjoint and cover all cases.

$$\begin{array}{l} TEB_{pc}(IF\ e_1\ ELSE\ e_2\ END,\ x) \rightsquigarrow \\ \{\ e_1 \triangleq (g_1 \wedge pc = pcName(e_1)) \to (a_1 \parallel pc := x), \\ \ \ e_2 \triangleq (\neg g_1 \wedge pc = pcName(e_1)) \to (a_2 \parallel pc := x)\} \end{array}$$

In the branching case, both events share the same enabling program counter. Instead the ordering is determined by the guard g_1 or $\neg g_1$, and both branches take the same next program counter value. In the case of additional sub-branches, we may have $i \in 2 \ldots n$ sub-branches,

$$\begin{array}{l} TEB_{pc}(IF\ e_1\ [ELSEIF\ e_i]^*\ ELSE\ e_{n+1}\ END,\ x) \rightsquigarrow \\ \{\ e_1 \triangleq (g_1 \wedge pc = pcName(e_1)) \to (a_1 \parallel pc := x), \\ \ \ [e_i \triangleq (\neg(g_{1\ldots i-1}) \wedge g_i \wedge pc = pcName(e_1)) \to (a_i \parallel pc := x),]^* \\ \ \ e_{n+1} \triangleq (\neg(g_{1\ldots n}) \wedge pc = pcName(e_1)) \to (a_m \parallel pc := x)\} \end{array}$$

For each of the sub-branches containing an event i, we require that the preceding guards are negated, and only the guard of g_i is true. We write $\neg(g_{1\ldots i-1})$ to denote a conjunction of the preceding guards. The action a_i can make state updates for this branch, and the program counter is set to the next value x.

4.4 Looping

The Tasking Event-B while loop is written *WHILE e1 END*. This simple loop repeats *e1* while the guard is true. It has the following Event-B semantics for which we provide the translation,

$$
\begin{aligned}
&TEB_{pc}(WHILE\ e_1\ END,\ x) \rightsquigarrow \\
&\{\ e_1 \triangleq (g_1 \wedge pc = pcName(e_1)) \rightarrow (a_1), \\
&\quad e_2 \triangleq (\neg g_1 \wedge pc = pcName(e_1)) \rightarrow (pc := x)\}
\end{aligned}
$$

We will not explicitly give details of the translation of Tasking Event-B constructs from the CLM to the implementations in target languages, since we anticipate that the CLM constructs represent implementation constructs in an 'obvious' way.

4.5 Tasks and Shared Machines

In our implementations we wish to describe two kinds of behaviour. The first behaviour is the task-like behaviour of the *tasking machine* machine. We impose an ordering on the events of a machine, to describe the activities that take place when the task is active. We model this in an extension to Event-B, using a *task-body* and the Tasking Event-B constructs introduced earlier in the section. The task-body, with its tasking notation, provides programming-style specification in Event-B. This is readily translated to the CLM and to Event-B. The second behaviour is that of the *shared* machine type, which provides a means to share information between the tasks. *Shared* machines have no flow control specification of their own (i.e. no event ordering). They rely on the task-body, and synchronization of events. The synchronizations arise from the decomposition process, and are recorded in the composed machine structure. The events in the *shared* machine synchronize with the corresponding events in tasks; it is through this that we translate to a procedure, and its call, in the implementation.

There are two types of tasking machines, one describing the controller tasks in the system; the other describing the environment. These are known as *autotask* and *environ* machines respectively. The main difference between a model of a controller, and a model of the environment, is that the environment model need only provide sufficient detail to implement a simulation of the environment. Whereas, a controller model can be refined to a level of detail

that makes it possible to generate code for an implementation which can be deployed on the target system. The environment model can also be used to generate a Java-style interface; or, with Ada, procedure bodies that can be modified, with calls to software driver APIs. This will allow a controller to interact with hardware in the real environment. The following table relates the Tasking Event-B machine constructs to their implementable types.

Tasking Event-B	CLM Type
Shared Machine	Protected Object
Autotask Machine	Deployable Task
Environ Machine	Environ Simulation Task

Table 1 Mapping from Machines to CLM Types

Now we will summarize the main constituents of the CLM representation of shared objects, and tasks, as attributes of their CLM types.

CLM Type	Attributes
ProtectedObject	name, variableDecls, constantDecls, subroutines
(Deployable) AutoTask	name, variableDecls, constantDecls, task-body
(Simulation)EnvironTask	name, variableDecls, constantDecls, task-body, subroutines

Table 2 CLM Type Attributes

5 Translating to the CLM

The translation to code for a sequence is relatively straightforward. We simply map the concrete sequence operator to a statement delimiter, expand the event actions, and add a terminating delimiter in the appropriate place. First we define a polymorphic translation function that maps from Tasking Event-B to program statements in our common language abstraction. TEB can be any of the Tasking Event-B annotations that we have introduced.

$$TEB_{clm} \in TEB \rightarrow ProgramStatements \tag{2}$$

5.1 Branching

The translation to the common language model, is as follows.

$$TEB_{clm}(IF\ e_1\ [ELSEIFe_n]^*\ ELSE\ e_m) \rightsquigarrow$$
$$if(g_1)\{a_1\}$$
$$[elseif(g_n)\{a_n\}]^*$$
$$else\{a_m\}$$

where * represents the possibility of zero or more branching statements.

5.2 Looping

The translation to the common language model, is as follows.

$$TEB_{clm}(WHILE\ e_1) \rightsquigarrow$$
$$while(g_1)\{a_1\}$$

6 Generating Tasks from Event-B Machines

So far we have just dealt with the Tasking Event-B language that we introduced. We now discuss how we translate the machines map to protected objects and tasks. We begin with an overview of the issues involved. In the CLM we have a 'generic' *autotask* element, generated from the Tasking Event-B *autotask*. The final implementation depends on the selected target language. To define the semantics of the CLM artefacts we use Ada programming semantics, since it is well defined [9]. The *main* Ada program may contain static task definitions in it declarative part. The tasks defined here, start after elaboration of the declarative part. This behaviour is modelled by the Event-B model, where each static task is modelled by an *autotask* machine. There is no explicit ordering imposed on their start-up, and the number of tasks is fixed, since it is determined by the number of machines in the development. The *autotask* can also be implemented as a Java thread, or a POSIX pthread in C; in this case the implementation should implement the Ada semantics.

The *autotask* construct, in the CLM, has a task-body containing an representation of the tasking constructs, obtained from the application the previously introduced translation rules. The task body may contain sequences, and branches, etc. but still contains Event-B predicates and expressions. Translation of these will take place in the second step, which involves the use of the theory plug-in. *Autotasks* are used to model a system's controllers, and it is intended that their translations will be deployed on the target system.

In the same way as *autotasks* are related to Ada tasks, *shared* Machines are related to Ada protected objects. The protected objects provide mutually exclusive access to private data. In our translation all machine variables are

translated to private variables, so it is necessary to use the object's procedures to access the data. Protected objects are owned by a task, and used to share information between tasks. Implementations in other languages may need to provide their own mutual exclusion mechanisms, such as using Java's synchronized constructs.

Environ Machines model the environment, and their implementation is similar to an *autotask*'s, with the exception that we allow direct communication between *autotasks*, and *environ* tasks. We prohibit inter-task communication between *autotasks*, for reasons given in the Ravenscar profile [10,11]. But here it is possible; because we use the Ada tasks for simulating the environment. It is never intended that the code be deployed, so the *environ* tasks can have Ada entries, for use with rendezvous style communication. Rendezvous allows two tasks to agree on a communication point in the code. The first task to reach that point will wait until the other is ready, and then proceed with the communication when both are ready. This allows controller implementations to poll, and set, environment values, using an entry call. Ada is unusual in that it has entries, for rendezvous, as a first class language construct. Similar implementations should be achievable in other languages, but have to be programmed explicitly.

6.1 Machine Translation

We summarize the remaining aspects of machine translation. Translation of task bodies has been described earlier in the section, with the exception of the implementation of communication between tasks and shared objects, and tasks and the environment. We look at communication between tasks in Sect. 7.

6.1.1 Translation of Variables, Constants, Invariants

In this subsection we look at the translation of variables, constants and invariants. In general, invariants are ignored in the final translation to implementation. That is, unless they are required downstream for further verification. However, typing invariants are used in the translation of variable declarations. Each variable v in a machine, is typed in an invariant i, and initialised in an initialisation action t. The function to translate the Event-B variable, type, and initial value, to a variable declaration is typed as follows.

$$TEB_{varDecl} \in Variable \times Invariant \times Action \rightarrow VariableDeclaration$$

where Variable, Invariant and Action are types of Event-B Elements. The variable declaration, in the CLM, has three attributes; the variable name, its type, and its initial value. We describe the translation from Event-B to a variable declaration, as follows,

$$TEB_{varDecl}(v,\ v \in t,\ v := x) \rightsquigarrow$$
$$VariableDeclaration :\ name = v,\ type = t,\ initialValue = x$$

A similar rule exists to translate constants into a constant declarations with typing and initialisation derived from axioms.

It is also possible that the constants are used in a partition, using the Event-B partition operator. This can be mapped to an enumeration declaration, a special kind of *constantDecl*. In an Event-B context, a partition is written as

$$partition(S, \{a\}, \{b\}, \ldots)$$

We define the following translation rule for translating the constants involved in a partition, we need only pass the axiom to the translator.

$$TEB_{partition} \in Axiom \rightarrow EnumDeclaration$$

The axiom describes the Set to be partitioned, and the elements, which are constants in singleton sets. We define the translation as follows,

$$TEB_{partition}(partition(S, \{a\}, \{b\}, \ldots)) \rightsquigarrow$$
$$EnumDeclaration : name = S, elements = a, b, \ldots$$

where the EnumDeclaration has a *name* attribute, and a list of unique *elements* representing the enumerated values.

6.1.2 Translation of Machines

For each of the machine types, we translate to the CLM types, introduced in Table 2,

The translation function to map a *shared* machine to its CLM representation is typed as follows,

$$TEB_{shared} \in Name \times \mathbb{P}(Variables) \times \mathbb{P}(Constants) \times$$
$$\mathbb{P}(Invariants) \times \mathbb{P}(Events) \rightarrow CLM$$

We translate a machine with name n, a set of variables V, a set of invariants I, a set of seen constants C, and E is the set of events. In the functions we only pass the relevant data, i.e. in $TEB_{varDecls}$, for each of the variables in the set, i is the typing invariant involving v, and t is the initialisation action. In $TEB_{constDecls}$, x is the axiom that types c

$$TEB_{shared}(n, V, C, I, E) \rightsquigarrow$$
$$ProtectedObject : name = n,$$
$$variableDecls = \{v \in V \mid TEB_{varDecl}(v, i, t)\},$$
$$constantDecls = \{c \in C \mid TEB_{constDecl}(c, x)\},$$
$$subroutines = \{e \in E \mid TEB_{subroutine}(e)\}$$

Next we define a translation function to map an *autotask* Event-B machine to its CLM representation. It is the same as a *shared* machine, but has the additional task body T to translate.

$$TEB_{autotask} \in Name \times \mathbb{P}(Variables) \times \mathbb{P}(Constants) \times \\ \mathbb{P}(Invariants) \times \mathbb{P}(Events) \times TaskBody \rightarrow CLM$$

The *autotask* machine translates to an *autotask* CLM construct. The task body t is translated as described in Sect. 5.

$$TEB_{autotask}(n, V, C, I, E, t) \rightsquigarrow \\ AutoTask : name = n, \\ variableDecls = \{v \in V \mid TEB_{varDecl}(v, i, t)\}, \\ constantDecls = \{c \in C \mid TEB_{constDecl}(c, b)\}, \\ taskBody = TEB_{clm}(t)$$

The translation of the *environ* machine is similar to the *autotask* machine, except that an *environ* machine may have subroutines. These can be called by an *autotask* to poll for information, or to set values in the environment.

6.1.3 Additional Information for Tasks

When defining *autotask* and *environ* machines, it is possible to specify two additional implementation features. The *task type* annotation allows a developer to specify which kind of task to implement, that is one of *periodic*, *repeating*, or *one-shot*. In the case of a periodic task, the period of the cycle may be specified in milliseconds. The second feature is the task priority, which should be set according to implementation needs. In typical use, safety related tasks can be given a higher priority than visual display related tasks. But we emphasize that its use is contingent on support in the underlying implementation.

7 Communication between machines

We now consider how communication between machines, with event synchronization, is translated. In a synchronization there is a local event e_l and a remote event e_r. Local and remote are with respect to a task. We say that a local event is the event in the synchronization that belongs to the task; the remote event belongs to the *shared* machine. Synchronization occurs between *composed* events. In our case we compose an event from a Tasking Machine with one from a *shared* Machine, or an *environ* Machine. The synchronization of the two composed events is equivalent to a single atomic event. We add a syntactic extension to the language defined in 1, so we now have *event* [$\|_e$ *event*]: i.e. optional synchronization.

We describe the event composition operator $\|_e$ using the guarded command language [16] to show the conjunction of local and remote guards, and parallel

combination of local and remote actions. In the following, p_{in} and p_{out} are lists of input, and output, parameters respectively. These parameters embody the direction of the information flow, which is an extension to the standard Event-B interpretation; but as in Event-B, parameter visibility is limited to event scope. The local event is written,

$$e_l(p_{in}, p_{out}, \top \rightarrow a_l)$$

where $\top$ is the true guard, and a_l are local actions accessing the local machine variables, and parameters. The local guard must be true. In this interpretation, of synchronizing events, the blocking of the calling task by the event guard makes no sense. The remote event is, however allowed to block, and is written

$$e_r(p_{in}, p_{out}, g_r \rightarrow a_r)$$

where g_r and a_r are remote guards and actions accessing only remote machine variables, and parameters.

We can describe the synchronization of two events in terms of its guards and actions,

$$a_l \parallel_e (g_r \rightarrow a_r) \triangleq g_r \rightarrow (a_l \parallel a_r)$$

Since the Ada programming language contains the notion of the *in* parameters and *out*, and our common language abstraction is based on Ada, we can easily incorporate the direction information. But, we need to extract the information from the Event-B model. This is relatively straightforward, since during the decomposition process, *in* parameters appear on the left hand side of assignments in an action (and are typed in the guard). *out* parameters only appear in event guards; one for typing, the other as an equality with some machine variable. Furthermore, an annotation is automatically added to the parameters in an Event-B model, to clarify the direction of the information flow.

In order to define the translation more precisely we use a translation rule of the type defined in Eq. 2. This rule translates an event synchronization to a call in an *autotask*, and a procedure in a *shared* object.

$$TEB_{clm}(e_l(p_{in}, p_{out}, a_l), e_r(p_{in}, p_{out}, g_r, a_r)) \rightsquigarrow$$
$$a_l; TEB_{call}(e_l(p_{in}, p_{out}))$$

We do not translate a_l at this stage, since translation of the mathematical language takes place in second stage. A procedure call is created by TEB_{call}. The name of the procedure call information is derived from the shared machine name, and event name. The parameters are derived from the equality in the guard, for outgoing parameters; and assignment for incoming parameters. It will be easier to illustrate translation of parameters using an example, which follows in the next subsection.

MACHINE $m1$
VARIABLES x, y, v, w
INVARIANTS
$x \in T \land y \in U, \ldots$
EVENTS
EVENT e
THEN $x := v \parallel w := y$
END
...

Fig. 8 Parameters: The Abstraction

MACHINE $m2$ REFINES $m1$
VARIABLES x, y, v, w
INVARIANTS
$x \in T \land y \in U, \ldots$
EVENTS
EVENT e REFINES e
ANY p_v, p_y
WHERE $p_v = v \land p_y = y$
THEN $x := p_v \parallel w := p_y$
END ...

Fig. 9 Parameters: The Refinement

7.1 An Example of Parameter Translation

We describe the translation of Event-B parameters using an example shown in Fig. 8 showing an abstract machine with variables x, y, v and w. We decompose the abstract machine into two machines, to demonstrate translation of parameter passing. In the decomposition, x and y will be local, v and w remote. In Fig. 9 we take the necessary step of introducing parameters into the model, to facilitate information flow between the decomposed machines. The assignments now have parameters, instead of variables, on the right-hand side. This model allows a decomposition where the value v is sent to the local machine from the remote machine via parameter p_v, and the value y is sent from the local machine to the remote machine via parameter p_y. During decomposition, the variables are distributed between machines, and some parameters are given typing guards. After decomposition we have two machines, and a composed machine which keeps track of the synchronizations. The events are shown in Fig. 11 where e_l and e_r are events of the local machine m_l and remote machine m_r. We have also added the direction annotation, ? for input and ! for output. during translation there will be a subroutine generated,

PERHAPS EXPLAIN WHAT HAPPENED IN THE TRANSLATION OF PARAMETERS HERE

A further translation is used to create a subroutine definition from the remote event. The translation TEB_{proc} is applied to events of shared machines, which are necessarily used as the remote events in a synchronization.

$$TEB_{proc} \in Event \rightarrow Subroutine$$

MACHINE m_l
VARIABLES x, y
...
EVENT e_l
  ANY $?p_v$, $!p_y$
  WHERE $p_v \in T \wedge p_y = y$
  THEN $x := p_v$
END
...
MACHINE m_r
VARIABLES v, w
EVENT e_r
  ANY $!p_v$, $?p_y$
  WHERE $p_v = v \wedge p_y \in U$
  THEN $w := p_y$
END

Fig. 10 Parameters: The Decomposed Events

TASK m_l
VARIABLES x, y
TASKBODY $m_r.e_r(x,\ y)$

SHARED_OBJECT m_r
VARIABLES v, w
SUBROUTINE e_r(out $p_v \in T$, in $p_y \in U$)
  BODY $p_v := v\ ;\ w := p_y$
END

Fig. 11 Parameters: The Subroutine and Call

Taking a shared machine event, as a parameter, the translation produces a subroutine in a shared object.

$$
\begin{aligned}
&TEB_{proc}(e_r(p_{in}, p_{out}, g_r, a_r)) \rightsquigarrow \\
&Subroutine: \\
&name = name(e_r) \\
&parameters = p_{in}, p_{out} \\
&when = g_r \\
&actions = a_r
\end{aligned}
$$

8 From CLM to Implementation

One further consideration is the translation of parallel assignments to sequential, where care must be taken to translate variables on the right hand side of assignments to a local variable representing the initial values of those variables. During the final translation step from CLM to programming language, guards and actions are translated using the $TEB_{iniValSubs}$ function, typed as follows,

$$TEB_{iniValSubs} \in ACTIONS \rightarrow ProgramStatements$$

In the $TEB_{iniValSubs}$, function we translate assignments of the form $l := E(V)$ to program assignments. Here, l is a single variable identifier on the lhs of the assignment, and $E(V)$ represents an expression involving a set of variables V on the right hand side. For each variable $v \in V$ we insert in the translated statements, a local variable of the same type as v.It is initialised so that $v_{ini} := v$. We replace occurrences of v in E(V) with v_{ini}. The translation function is applied to an event as follows,

$$TEB_{clm}(e_l) \rightsquigarrow \\ TEB_{iniValSubs}(a_l)$$

As an example, a_l may be a parallel assignment, where $x := y \parallel y := x$. To translate this we have,

$$TEB_{iniValSubs}(x := y \parallel y := x) \rightsquigarrow \\ y_{ini} := y;\ x_{ini} := x;\ x := y_{ini};\ y := x_{ini}$$

8.1 Theories for TEB

8.2 State-machines

9 Tooling

9.1 The Rodin Platform and Eclipse

9.2 IL1/CLM

9.3 Templates

9.4 Interfaces

10 Conclusions

10.1 Discussion

10.2 Related Work

As we mentioned in the introduction, the main driver for this work has been to derive the greatest benefit from the formal modelling approach, by making use of the formal modelling artefacts to generate implementations. There are many formal notations and many have support for automatic code-generation.

The Classical-B [3] approach made use of an implementation-level notation called *B0*, described in [15]. B0 is similar to a programming language, and consists only of concrete programming constructs. These constructs map to programming constructs in a number of programming languages. *B0* forms part of the Classical-B refinement chain, so the implementation-level specification refines the abstract development. Translators are available targeting programming languages such as C [25], and *High Integrity Ada* (based on *SPARK Ada* [2]).

The *Z-notation* is a state-based specification notation (actually a distant ancestor of Event-B).

The VDM-SL is a state-based formal specification language, related to *Z*, and has tool support for automatic code generation.

VDM++ is an object-oriented extension to Models can be described textually; or using a graphical interface using UML diagrams, in much the same way as UML-B does for B and Event-B. VDM++ can be used with the VDM++ Toolbox to generate C++ and Java code. VDM++ can be used to model and implement developments with concurrently executing processes, using threads.

Object-Z [35] is an object-oriented approach to development using *Z*. A route to implementation is described using a translation to *PerfectDeveloper* [19] in [37]. *PerfectDeveloper*'s approach is to use verified-Design By Contract, where verification conditions are generated from a specification using constructs such as pre and postconditions, class and loop invariants, and assertions. The verification conditions must be shown to hold in order to show the specified contracts are satisfied by the implementation. They are generated for each method entry to show that the precondition holds, for each method exit to show that the postcondition holds, and wherever an assertion appears. *PerfectDeveloper* provides automatic, and semi-automatic, translations to Java and C++ but appears not to support concurrent processing.

There are some combined formal approach where code-generation has been investigated. CSP [23,30] is a process algebraic approach to system specification in which the ordering of events occurring in a system play a major role. CSP specifications have been translated into Java code using JCSP [28, 42]. A hybrid CSP and Classical-B approach [31,32] combines the benefits of modelling, using both process-based and state-based techniques. In this approach, called $CSP \parallel B$, specifications are used to constrain the order that the state-changing operations may occur; and to specify points at which the

processes may interleave. The B operations synchronize with CSP events with the same name, and provide an ordering of the occurrence of B operations. ProB [26] is an animator and model checker for Classical-B and Event-B, and can be used with the $CSP \parallel B$ combination. It is used in the JCSProB [44] approach, which combines CSP and Classical-B, and has a code generator inspired by JCSP. Another combined approach, amenable to model-checking, is *Circus* [43], which is combination of CSP and *Z-notation* [36]. CSP is used to order the Z operations. It can be translated to Java as described in [20] and makes use of the JCSP library.

References

1. POSIX.1c, Threads extensions, IEEE Std 1003.1, 2004 Edition
2. SPARKAda. URL http://www.adacore.com/sparkpro/
3. Abrial, J.: The B Book - Assigning Programs to Meanings. Cambridge University Press (1996)
4. Abrial, J., Butler, M., Hallerstede, S., Hoang, T., Mehta, F., Voisin, L.: Rodin: An Open Toolset for Modelling and Reasoning in Event-B. Software Tools for Technology Transfer **12**(6), 447–466 (2010). URL http://dx.doi.org/10.1007/s10009-010-0145-y
5. Abrial, J., Hallerstede, S.: Refinement, Decomposition, and Instantiation of Discrete Models: Application to Event-B. Fundam. Inform. **77**(1-2), 1–28 (2007)
6. Abrial, J.R.: Modeling in Event-B: System and Software Engineering. Cambridge University Press (2010)
7. Abrial, J.R., Cansell, D.: Clickn prove: Interactive proofs within set theory. In: D. Basin, B. Wolff (eds.) Theorem Proving in Higher Order Logics, *Lecture Notes in Computer Science*, vol. 2758, pp. 1–24. Springer Berlin Heidelberg (2003). DOI 10.1007/10930755_1. URL http://dx.doi.org/10.1007/10930755_1
8. Back, R., Sere, K.: Stepwise Refinement of Parallel Algorithms. Science of Computer Programming **13**(2-3), 133 – 180 (1990). DOI http://dx.doi.org/10.1016/0167-6423(90)90069-P. URL http://www.sciencedirect.com/science/article/pii/016764239090069P
9. Barnes, J.: Programming in Ada 2005. International Computer Science Series. Addison Wesley (2006). ISBN-10: 0321340787 / ISBN-13: 9780321340788
10. Burns, A.: The Ravenscar Profile. Ada Lett. **XIX**, 49–52 (1999). DOI http://doi.acm.org/10.1145/340396.340450. URL http://doi.acm.org/10.1145/340396.340450
11. Burns, A., Dobbing, B., Vardanega, T.: Guide for the use of the Ada Ravenscar Profile in high integrity systems. Ada Lett. **XXIV**, 1–74 (2004). DOI http://doi.acm.org/10.1145/997119.997120. URL http://doi.acm.org/10.1145/997119.997120
12. Butler, M.: Incremental Design of Distributed Systems with Event-B (2008). URL http://eprints.ecs.soton.ac.uk/16910/
13. ClearSy: B4Free. from http://www.b4free.com
14. Clearsy System Engineering: The Atelier B Tool. URL http://www.atelierb.eu/en/
15. ClearSy System Engineering: The B Language Reference Manual, version 4.6 edn.
16. Dijkstra, E.: Guarded Commands, Non-determinancy and a Calculus for the Derivation of Programs. In: F. Bauer, K. Samelson (eds.) Language Hierarchies and Interfaces, *Lecture Notes in Computer Science*, vol. 46, pp. 111–124. Springer (1975)
17. Edmunds, A.: Providing concurrent implementations for event-b developments. Ph.D. thesis, University of Southampton (2010). URL http://eprints.soton.ac.uk/141688/
18. Edmunds, A., Rezazadeh, A., Butler, M.: Formal Modelling for Ada Implementations: Tasking Event-B. In: Ada-Europe 2012: 17th International Conference on Reliable Software Technologies (2012). URL http://eprints.soton.ac.uk/335400/
19. Escher Technologies: PerfectDeveloper. Available at http://www.eschertech.com
20. Freitas, A., Cavalcanti, A.: Automatic Translation from Circus to Java. In: J. Misra, T. Nipkow, E. Sekerinski (eds.) FM, *Lecture Notes in Computer Science*, vol. 4085, pp. 115–130. Springer (2006)
21. Gosling, J., Joy, B., Steele, G., Bracha, G.: Java Language Specification - Third Edition. Addison-Wesley (2004)
22. Hallerstede, S.: Justifications for the Event-B Modelling Notation. In: Julliand and Kouchnarenko [24], pp. 49–63
23. Hoare, C.: Communicating Sequential Processes. Prentice Hall (1985)
24. Julliand, J., Kouchnarenko, O. (eds.): B 2007: Formal Specification and Development in B, 7th International Conference of B Users, Besançon, France, January 17-19, 2007, Proceedings, *Lecture Notes in Computer Science*, vol. 4355. Springer (2006)
25. Kernighan, B., Ritchie, D.: The C Programming Language, Second Edition. Prentice-Hall (1988). URL http://cm.bell-labs.com/cm/cs/cbook/
26. Leuschel, M., Butler, M.: ProB: an automated analysis toolset for the B method. STTT **10**(2), 185–203 (2008). URL http://dblp.uni-trier.de/db/journals/sttt/sttt10.html#LeuschelB08

27. Maamria, I.: Towards a practically extensible event-b methodology. Ph.D. thesis, University of Southampton (2013). URL http://eprints.soton.ac.uk/347887/
28. P.H.Welch, Aldous, J., Foster, J.: CSP Networking for Java (JCSP.net). In: Computational Science - ICCS 2002: International Conference (2002)
29. RODIN Project: at http://rodin.cs.ncl.ac.uk
30. Roscoe, A.W., Hoare, C.A.R., Bird, R.: The Theory and Practice of Concurrency. Prentice Hall PTR, Upper Saddle River, NJ, USA (1997)
31. Schneider, S., Treharne, H.: Communicating B Machines. In: D. Bert, J.P. Bowen, M.C. Henson, K. Robinson (eds.) ZB, *Lecture Notes in Computer Science*, vol. 2272, pp. 416–435. Springer (2002)
32. Schneider, S., Treharne, H.: CSP Theorems for Communicating B Machines. Formal Asp. Comput. **17**(4), 390–422 (2005)
33. Silva, R., Butler, M.: Shared Event Composition/Decomposition in Event-B. In: FMCO Formal Methods for Components and Objects (2010). URL http://eprints.soton.ac.uk/272178/. Event Dates: 29 November - 1 December 2010
34. Silva, R., Pascal, C., Hoang, T., Butler, M.: Decomposition Tool for Event-B. Software: Practice and Experience (2010). URL http://eprints.ecs.soton.ac.uk/21714/
35. Smith, G.: The Object-Z specification language. Kluwer Academic Publishers, Norwell, MA, USA (2000)
36. Spivey, J.M.: The Z notation: A Reference Manual. Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1989)
37. Stevens, B.: Implementing Object-Z with Perfect Developer. Journal of Object Technology **5**(2), 189–202 (2006)
38. The Advance Project Team: Advanced Design and Verification Environment for Cyber-physical System Engineering. Available at http://www.advance-ict.eu
39. The DEPLOY Project Team: Project Website. at http://www.deploy-project.eu/. URL http://www.deploy-project.eu/
40. The Modelica Association Project: The Functional Mock-up Interface. Available at https://www.fmi-standard.org/
41. The OpenMP Architecture Review Board: The OpenMP API specification for parallel programming. Available at http://openmp.org/wp/
42. Welch, P., Martin, J.: A CSP Model for Java Multithreading. In: Software Engineering for Parallel and Distributed Systems (2000)
43. Woodcock, J., Cavalcanti, A.: A Concurrent Language for Refinement. In: A. Butterfield, G. Strong, C. Pahl (eds.) IWFM, Workshops in Computing. BCS (2001)
44. Yang, L., Poppleton, M.: Automatic Translation from Combined B and CSP Specification to Java Programs. In: Julliand and Kouchnarenko [24], pp. 64–78