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 [26]. Further work was undertaken in the DEPLOY [36] 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, as the platform stabilized, where we targeted multi-tasking, embedded control systems. Initially, we chose Ada [9] as a basis for our 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 properties relating to the state, that are important for system safety. In current work, in the Advance project [35], 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.

1.1 Some History

Initial development of the code-generation approach began from an object-oriented perspective, with the implementation-level notation, object-oriented, concurrent-B (OC-B) [14]. The focus of this work was to use Event-B to model and generate code for safe multi-tasking with object-oriented languages. We initially targeted the Java [18] language. We found, however, that this approach gave rise to a notation where the semantic gap between Event-B and the OC-B was quite large. This was not optimal for developers of Event-B models, who then wanted to write implementation-level specifications, from which code could be generated. We also found that 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 [30,31], 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

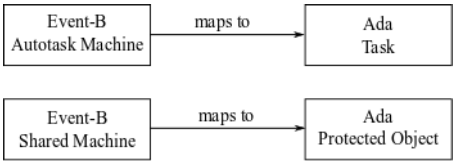

Fig. 1 Mapping between Event-B and Ada

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 Sects. 3.4 and **??**.

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, 11, 12] 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 [19], 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 [19], 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 of the fluid level, and a boolean value representing a user-request to turn the pump. Based on the inputs to the controller, a command to turn the pump on may be issued, or a warning is issued if a minimum level *MIN* has been reached. In Fig. 2, we see that machine *M1* refines another machine *M0*; we 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

```
MACHINE m1 REFINES m0 SEES ctx
VARIABLES m_level, c_level, e_level, m_pumpOnReq, c_pumpOnReq, e_pumpOnReq,
   m_pumpOnCmd, c_pumpOnCmd, e_pumpOnCmd, m_warn, c_warn,
e_warn,
  c_level_internal, c_pumpOnReq_internal
INVARIANTS
  (c_level_internal ≤ MIN ∧ c_pumpOnReq_internal = TRUE ⇒ c_warn
= TRUE)
   ∧ (c_level_internal > MIN ∧ c_pumpOnReq_internal = TRUE
      ⇒ c_pumpOnCmd = TRUE)
   ∧ (c_level_internal ∈ ℤ)
   ∧ (c_pumpOnReq_internal ∈ BOOL) . . .
EVENTS
INITIALISATION c_level := 100 ∥ m_level := 80 ∥ c_pumpOnReq :=
FALSE ∥ . . .
EVENT fmiSetBoolean_c REFINES fmiSetBoolean_c
  ANY p
  WHERE p = c_compound ∧ p ∈ BOOL
  THEN m_pumpOnCmd := p
  END
. . .
```

Fig. 2 An Event-B Pump Controller Model

of the controller, and invariants 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 re-

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

Fig. 3 An Example Context

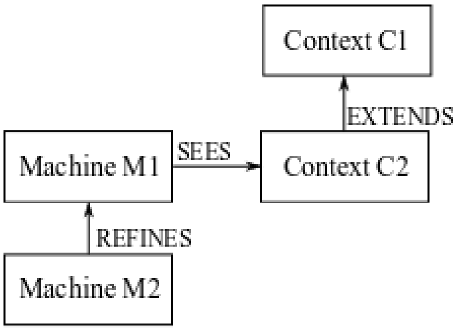

Fig. 4 Refinement and Extension

lationship. 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 system 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 [10]. 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 [18], C [22] 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 [30] 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. It keeps track of the refinement relationship between the abstract machine and the decomposed machines. In the decomposition, it is the synchronization of events, in different 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 [24]. 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.
2. Datatypes: Used to define simple data types, which can be added using a type constructor and element constuctors.

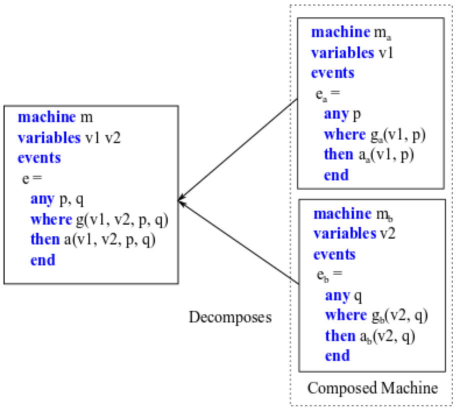

Fig. 5 Decomposition

3. Operators: The operators section can be used to define polymorphic operators, such as the sequence described as an ordered list [24].
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.

3.4 Targets for Code-Generation

In this subsection, we provide details of the target languages that we wish to generate code for. Ada provides the basis for the code-generation approach as described in [15]. Ada tasks are processing units, known to the run-time

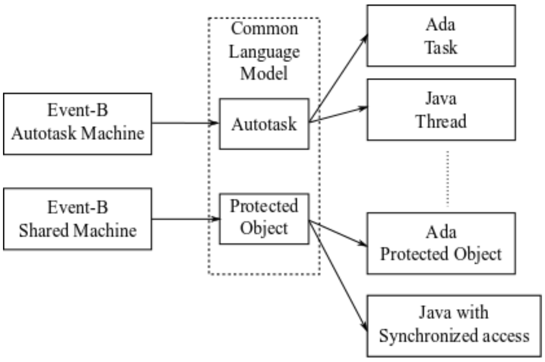

Fig. 6 The Common-Language Model

system. Protected objects provide an encapsulation mechanism, that prevents interference. 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 [38] 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 [35]. In this approach we generate a specialized form of C code, conforming to the FMI standard for model co-simulation [37]. 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; with the first step translating to a common-language model, containing 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 discussion about 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, as we approach the implementation-level specification, we will consider how the behaviour of a task-like implementation construct can be modelled in Event-B, and we will introduce a notation for the purpose of ordering of events. In the following discussion, we distinguish between the behaviour of a task, such as it's life-cycle (which is not modelled formally) and the updates to state (which are modelled formally). In our approach we intend that the implementation-level development be a refinement of the abstract development. We will therefore introduce notations that assist with the formal and non-formal aspects of the specification. The non-formal aspects of the notation are required in the case that we want to generate code for periodic tasks, for instance.

4.1 Flow Control for Events

To enable us to impose an order on events we introduce the *task body*, a textual extension to standard Event-B. The first notion we will look at is sequential ordering, for which we introduce a 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.

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

4.2 Introducing sequences

We will now define some translation functions for translating Tasking Event-B to Event-B. To add the guards and actions modelling a program step, seen in Fig. 7, we define a translation TEB_step. It takes, as parameters, a TEB construct (at this stage, either an event or sequence operator) and the next program counter name. It returns a set of events. In the first instance, the syntax of TEB ::= EVENT | SEQ, where the concrete syntax of SEQ is the semi-colon ';'. We assume that events have no guards other than the program counter guards.

$$TEB_{step} \in TEB \times Name \rightarrow POW(EVENT)$$

We use a function *pcName* to extract an event name from an event, for use as a program counter variable. When applied to a sequence it returns two events.

$$\begin{aligned} &TEB_{step}(e_1; e_2, x) = \{ \\ &TEB_{step}(e_1, pcName(e_2)), \\ &TEB_{step}(e_2, x)\} \end{aligned}$$

The TEB_{step} function is applied to each event in the returned set. In the following discussion we represent an event, with guards and actions, using the notation $g \rightarrow a$; where g is the guard, and a is the action.

$$\begin{aligned} &TEB_{step}(e_1, x) = \\ &e_1 \triangleq pc = pcName(e_1) \rightarrow a_1 \parallel pc = x \end{aligned}$$

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

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

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

To resolve the mapping from parallel to sequential we define a function which generates program statements from the event $ACTIONS$.

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

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 expressions 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 applied to the an event is therefore defined as a further translation,

$$\begin{aligned} &TEB_{clm}(e_1) = \\ &TEB_{iniValSubs}(a_1) \end{aligned}$$

As an example, a_1 may be a parallel assignment, where $x := y \parallel y := x$. To translate this we have,

$$\begin{aligned} &TEB_{iniValSubs}(x := y \parallel y := x) = \\ &y_{ini} := y;\ x_{ini} := x;\ x := y_{ini};\ y := x_{ini} \end{aligned}$$

4.3 Introducing branching

To the CLM: In many programming languages we see the semi-colon ';' used as a delimiter, to terminate a program statement. A sequence of program statements is constructed by appending semi-colon delimited statements. In formal languages we often use the semi-colon as an operator. When thinking about a sequence of events we need to consider the relationship between our notion of sequence, and the delimiter used in these languages. An event is an atomic update; so, in a multi-tasking environment, it should be implemented with the necessary protection mechanisms. The CLM isolates us from this.

4.4 Theories for TEB

4.5 State-machines

5 Tooling

5.1 The Rodin Platform and Eclipse

5.2 IL1/CLM

5.3 Templates

5.4 Interfaces

6 Conclusions

6.1 Discussion

6.2 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 [13]. 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 [22], and *High Integrity Ada* (based on *SPARKAda* [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 [32] is an object-oriented approach to development using Z. A route to implementation is described using a translation to *PerfectDeveloper* [16] in [34]. *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 [20,27] 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 [25, 39]. A hybrid CSP and Classical-B approach [28,29] 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 [23] 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 [41] approach, which combines CSP and Classical-B, and has a code generator inspired by JCSP. Another combined approach, amenable to model-checking, is *Circus* [40], which is combination of CSP and *Z-notation* [33]. CSP is used to order the Z operations. It can be translated to Java as described in [17] 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. Butler, M.: Incremental Design of Distributed Systems with Event-B (2008). URL http://eprints.ecs.soton.ac.uk/16910/
11. ClearSy: B4Free. from http://www.b4free.com
12. Clearsy System Engineering: The Atelier B Tool. URL http://www.atelierb.eu/en/
13. ClearSy System Engineering: The B Language Reference Manual, version 4.6 edn.
14. Edmunds, A.: Providing concurrent implementations for event-b developments. Ph.D. thesis, University of Southampton (2010). URL http://eprints.soton.ac.uk/141688/
15. 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/
16. Escher Technologies: PerfectDeveloper. Available at http://www.eschertech.com
17. 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)
18. Gosling, J., Joy, B., Steele, G., Bracha, G.: Java Language Specification - Third Edition. Addison-Wesley (2004)
19. Hallerstede, S.: Justifications for the Event-B Modelling Notation. In: Julliand and Kouchnarenko [21], pp. 49–63
20. Hoare, C.: Communicating Sequential Processes. Prentice Hall (1985)
21. 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)
22. Kernighan, B., Ritchie, D.: The C Programming Language, Second Edition. Prentice-Hall (1988). URL http://cm.bell-labs.com/cm/cs/cbook/
23. 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
24. Maamria, I.: Towards a practically extensible event-b methodology (2013). URL http://eprints.soton.ac.uk/347887/
25. P.H.Welch, Aldous, J., Foster, J.: CSP Networking for Java (JCSP.net). In: Computational Science - ICCS 2002: International Conference (2002)
26. RODIN Project: at http://rodin.cs.ncl.ac.uk
27. Roscoe, A.W., Hoare, C.A.R., Bird, R.: The Theory and Practice of Concurrency. Prentice Hall PTR, Upper Saddle River, NJ, USA (1997)

28. 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)
29. Schneider, S., Treharne, H.: CSP Theorems for Communicating B Machines. Formal Asp. Comput. **17**(4), 390–422 (2005)
30. 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
31. 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/`
32. Smith, G.: The Object-Z specification language. Kluwer Academic Publishers, Norwell, MA, USA (2000)
33. Spivey, J.M.: The Z notation: A Reference Manual. Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1989)
34. Stevens, B.: Implementing Object-Z with Perfect Developer. Journal of Object Technology **5**(2), 189–202 (2006)
35. The Advance Project Team: Advanced Design and Verification Environment for Cyber-physical System Engineering. Available at http://www.advance-ict.eu
36. The DEPLOY Project Team: Project Website. at http://www.deploy-project.eu/. URL `http://www.deploy-project.eu/`
37. The Modelica Association Project: The Functional Mock-up Interface. Available at https://www.fmi-standard.org/
38. The OpenMP Architecture Review Board: The OpenMP API specification for parallel programming. Available at http://openmp.org/wp/
39. Welch, P., Martin, J.: A CSP Model for Java Multithreading. In: Software Engineering for Parallel and Distributed Systems (2000)
40. Woodcock, J., Cavalcanti, A.: A Concurrent Language for Refinement. In: A. Butterfield, G. Strong, C. Pahl (eds.) IWFM, Workshops in Computing. BCS (2001)
41. Yang, L., Poppleton, M.: Automatic Translation from Combined B and CSP Specification to Java Programs. In: Julliand and Kouchnarenko [21], pp. 64–78