# From Event-B Models to Code

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

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

**Abstract**

Event-B is a formal approach to modelling systems, based on set theory, predicate logic and arithmetic. Various techniques are used during modelling, including refinement, and decomposition. This article describes an extension to the Event-B approach, which we call Tasking Event-B, that facilitates automatic generation of source code, from annotated Event-B models. 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.

# 1 Introduction

Event-B [1] 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* [2]. It is a modelling language, used with a supporting tool platform, Rodin [3]; so named from the project in which it was developed [10].

In this section we introduce Event-B to the reader, and compare Event-B with some other formal approaches. We discuss automatic code generation from formal models, and potential target programming languages.

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

## 1.1 Event-B

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 [2, 4, 7, 8] 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 [9], and inspiration was also drawn from action systems [5].

It is fair to say that Event-B is not just an entity; it is a notation, and a methodology. A mature core tool-set complements the methodology; the tool platform being named *Rodin*, after its development programme. The main modelling components of Event-B are contexts and machines. Contexts are used to model static features using sets, constants, and axioms. 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 [9], 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.

As we mentioned earlier, Event-B makes use of a technique called refinement, where a machine can be refined by another. During this process new variables, events and invariant properties can be added; or existing events can be modified 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. 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 withthe proof effort, we can link the parameters of the abstract event with their concrete counterpart using a $WITNESS$, this construct is a predicate describing the relationship between the event parameter of the abstraction and a corresponding variable in the refinement. 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. 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 [6].

In terms of the recommended methodology, Event-B development begins with the abstraction, and modelling, of the observable events occurring in a system. Event-B (in correspondence with its name) takes an event-based view of a system.

Event based modelling uses the notion of guarded events to describe the observable events. An event is said to be enabled when the guard is true, otherwise it is disabled. Typically when an enabled event fires some state update occurs, which is described by the event's action. The high level abstraction can be refined, possibly a number of times. At each refinement step new events and state information are added. The purpose of refinement in the Event-B method is to introduce more detail into the model and at the same time maintain the model's consistency. Eventually the model should describe the behaviour of the system at such a level of detail that an OCB model can be defined. The OCB model is subsequently transformed to an Event-B model that can be shown to refine the abstract development, and the target source code. An example of textual Event-B is shown in Figure 1. The context, named *exampleContext* has a set, $A$. Machine *exampleMachine* sees *exampleContext* to gain access to its contents. It has variables $b$ and $c$ which are typed in the invariant along with any additional constraints on the state. In this example $b$ is typed as a powerset of $A$ and $c$ is an Integer. The example shows an event named *inc* which increments the value of $c$. The event named *new* non-deterministically selects an element of set $A \setminus b$ and adds it to $b$, using set union. Events consist of guards and actions, *inc* has no non-deterministic parameters, so the guard is in the *WHEN* clause, and the actions are in the *THEN* clause. Where the event has non-deterministic parameters, as in *new*, the guard is contained in the *WHERE* clause. Guards are predicates which describe the conditions under which the event is enabled, actions are substitutions which describe the effects of the event. An event describes the transition from the 'before' state to the 'after' state which happens atomically; that is, there is no intermediate state visible. In a consistent model the guards of an event ensure that its actions do not violate the invariant. When developing a software system it can be useful to view the occurrence of state changes, in shared memory concurrent systems, as atomic events. We aim to relate the atomic state changes of such an implementation to atomic events described in an Event-B model, using OCB. This will simplify reasoning about the system under development since our abstraction does not include details of locking, unlocking and the implementation of conditional waiting. Event-B provides the formal semantics for the OCB notation which is introduced in subsequent chapters.

One important aspect of the Event-B approach is that *any* enabled event may occur, but only one of the enabled events may occur at any one moment. When modelling certain aspects of a system we may wish to impose an ordering of events. However, there is no sequence operator provided in the Event-B approach. It is therefore necessary to make appropriate use of guards and and state variables

**CONTEXT** exampleContext
**SETS** A

**MACHINE** exampleMachine
**SEES** exampleContext
**VARIABLES** b, c
**INVARIANT**
  $b \subseteq A \wedge c \in 0\,..\,10$
**EVENTS**
  INITIALISATION =
    $b := \varnothing \parallel c :\in 0\,..\,10$

  inc =
    **WHEN** $c{+}1 \in 0\,..\,10$
    **THEN** $c := c{+}1$
    **END**

  new =
    **ANY** x
    **WHERE** $x \in A \setminus b$
    **THEN** $b := b \cup \{x\}$
    **END**

Figure 1: Example of Textual Event-B

to model this aspect of a system. For example if we wish to impose an ordering on two events *evt1* and *evt2* so that $evt1$ occurs before $evt2$ we can use the following approach. Introduce an enumerated set $Grds = \{one, two, stop\}$ and a variable $step \in Grds$. Initially $step := one$; and we make use of $step$ in the event guards as follows,

$$evt1 = \textbf{WHEN}\ step = one\ \textbf{THEN}\ \ldots \parallel step := two\ \textbf{END}$$
$$evt2 = \textbf{WHEN}\ step = two\ \textbf{THEN}\ \ldots \parallel step := stop\ \textbf{END}$$

This ensures that initially $evt1$ is enabled and $evt2$ is disabled since $step = one$; only after $evt1$ has updated the $step$ variable to $two$ is $evt2$ enabled. At this time $evt1$ is no longer enabled since its guard is now false. Finally no events are enabled since $step = stop$ and all guards are false.

## 1.2 Related Approaches

– VDM
  – Z

## 1.3 Code Generation Rationale

## 1.4 Targets for Code Generation

- Ada
  - Java
  - FMI-C

# 2 More about Event-B

## 2.1 Refinement

## 2.2 Decomposition and Composition

## 2.3 Theories

## 2.4 ProB

# 3 Tasking Event-B

## 3.1 The Language and Semantics

## 3.2 Theories for TEB

## 3.3 State-machines

# 4 Tooling

## 4.1 The Rodin Platform and Eclipse

## 4.2 IL1/CLM

## 4.3 Templates

## 4.4 Interfaces

## 5 Conclusions

Conclusions

## References


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

[2] J.R. Abrial. *The B Book - Assigning Programs to Meanings*. Cambridge University Press, 1996.

[3] J.R. Abrial, M. Butler, S. Hallerstede, T.S. Hoang, F. Mehta, and L. Voisin. Rodin: An Open Toolset for Modelling and Reasoning in Event-B. *Software Tools for Technology Transfer*, 12(6):447–466, November 2010. `http://dx.doi.org/10.1007/s10009-010-0145-y`.

[4] J.R. Abrial and D. Cansell. Click'n Prove: Interactive Proofs within Set Theory. In *TPHOLs*, 2003.

[5] R.J.R. Back and K. Sere. Stepwise Refinement of Parallel Algorithms. *Science of Computer Programming*, 13(2-3):133 – 180, 1990.

[6] M. Butler. Incremental Design of Distributed Systems with Event-B, November 2008.

[7] D. Cansell. The Click_n_Prove interface. from http://www.loria.fr/ cansell/cnp.html.

[8] ClearSy. B4Free. from http://www.b4free.com.

[9] S. Hallerstede. Justifications for the Event-B Modelling Notation. In J. Julliand and O. Kouchnarenko, editors, *B*, volume 4355 of *Lecture Notes in Computer Science*, pages 49–63. Springer, 2007.

[10] RODIN Project. at http://rodin.cs.ncl.ac.uk.