Tasking Event-B: An Extension to Event-B for Generating Concurrent 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

The Event-B method is a formal approach for modelling systems in safety-, and business-critical, domains. Initially, system specification takes place at a high level of abstraction; detail is added in refinement steps as the development proceeds toward implementation. Our aim has been to develop a novel approach for generating code, for concurrent programs, from Event-B. We formulated the approach so that it integrates well with the existing Event-B methodology and tools. In this paper we introduce a tasking extension for Event-B, with Tasking and Shared Machines. We make use of refinement, decomposition, and the extension, to structure projects for code generation for multi-tasking implementations. During the modelling phase decomposition is performed; decomposition reduces modelling complexity and makes proof more tractable. The decomposed models are then extended with sufficient information to enable generation of code. A task body describes a task's behaviour, mainly using imperative, programming-like constructs. Task priority and life-cycle (periodic, triggered, etc.) are also specified, but timing aspects are not modelled formally. We provide tool support in order to validate the practical aspects of the approach.

1 Introduction

Event-B [3] can be used to model both single and multi-tasking software systems. The approach that we present here is greatly influenced by our previous experience [9, 10, 11] where we link Event-B with Java. We continue this section discussing our motivation. Section 2 provides an overview of Event-B. Section 3 describes the Tasking Event-B extension. Section 4 describes the preparation of a model for code generation. Section 5 describes the translation. In Section 6 we discuss our results and future work.

The work reported here has been undertaken as part of the EU DEPLOY [2] project, where our target domain is multi-tasking, real-time embedded systems. Our more general interest is in modelling concurrency in the application domain, with a view to automatic code generation. In previous work [9, 10, 11] we identified a problem with the large semantic gap between Event-B and the language that we used for specifying the implementation. We also encountered problems working with large models; our language introduced too much fine-grained atomicity, giving rise to a large number of proof obligations.

The undertaking, described here, was to develop an approach which integrates well with the existing Event-B methodology. One of our aims was to have just a few additions to the Event-B language; to have a small semantic gap between Event-B and the tasking specification. In essence we extend Event-B with just enough information to be able to derive an implementation. To address the problem of large model size we make use of Event-B's decomposition approach [6, 19]. After decomposition a machine can be further refined, and decomposed again if necessary. At a suitable point we introduce implementation specific constructs to the development and generate the code. We also use the extended Event-B model to generate a model of the implementation. We have developed a demonstrator tool [22] to validate the approach; the tool integrates with the existing Rodin platform [23].

machine AbstractBuffer
variables buff wVal ...
invariants
  buff $\in \mathbb{Z}$
  wVal $\in \mathbb{Z}$
    ...

event write
where
  buff $<$ 0
then
  buff := wVal
  sCount := sCount + 1
  wCount := sCount + 1
  wCount2 := wCount2 + 1
end

Figure 1: Example of Textual Event-B

2 Event-B

The Event-B method [3] was developed by J.R. Abrial, and uses set-theory, predicate logic and refinement to model discrete systems. The basic structural features of Event-B are contexts and machines. Contexts are used to describe the static features of a system using sets and constants, and the relationships between them are specified in the axioms. Machines are used to describe the variable features of a system in the form of state variables and guarded events which update state; system properties are specified using the invariants clause. Machines are able to *see* Contexts; the contents of a Context is visible and accessible to a machine. The invariants give rise to proof obligations, which are generated automatically by the tool; a large number of the proof obligations may be discharged without user intervention by the provers. Where auto-provers fail to discharge proof obligations the user guides the interactive prover. They proceed by suggesting strategies, and sub-goals in the form of hypotheses, in the endeavour to complete the proof.

2.1 An Event-B Model

A fragment of an Event-B specification is shown in Fig. 1. The specification has a number of variable declarations which are typed in the invariant clause. Additional predicates are added to the invariants clause to describe the desired safety properties; the invariants clause consists of a conjunction of predicates. The *write* event has a single guard clause (but may have more) following the *where* keyword, and a number of actions following the *then* keyword. The guard is a predicate, over the sets, constants, and variables of the system, describing a condition under which an event may occur. Each event guard may have a number of guard clauses which are conjoined. Action clauses consist of assignments to variables, and may be non-deterministic. The event action is a parallel composition of the action clauses.

3 Tasking Event-B

3.1 Extending Event-B Machines

Tasking Event-B introduces a number of new concepts to facilitate code generation. The most significant is the extension of Event-B with two new types of machine, namely Tasking and Shared Machines. Tasking Machines are related to the concept of an Ada [21] task (but we are not restricted to Ada implementations). Shared Machines are related to the concept of a protected resource, such as a monitor [14].

Event-B machines without any additional additional features may be implemented; but we wish to implement the machines in a particular way which necessitates the use of information in addition to

that available in a standard Event-B machine. We introduce specific tasking constructs to facilitate code generation for multi-tasking systems where the implemented tasks may have interleaving executions, but are restricted to communicate only with some protected resource in a mutually exclusive manner. The tasking features are an extension of standard Event-B machines, and machines to be implemented are characterised by one of the following attributes:

- *Auto Task Machine* or *Shared Machine*

Auto Tasks are tasks that will be declared and defined in the *Main* procedure of the implementation. The effect of this is that the *Auto Tasks* are created when the program first loads, and then activated (made ready to run) before the *Main* procedure body runs.

3.2 Tasking Specifics

3.2.1 Task Scheduling

The following extensions relate only to *Tasking Machines*, and provide implementation details; but note that timing aspects of periodic tasks, and scheduling, is not modelled formally.

- *TaskType* and *Priority*

the *TaskType* construct is used to define the scheduling, cycle and lifetime of a task. i.e. one-shot, periodic or triggered. The period of a task is specified in milliseconds. *Priority* is an integer value; the task with the highest value priority takes precedence when being scheduled.

3.2.2 Flow Control

Each Tasking Machine has a *task body* which contains the following flow control (algorithmic) constructs.

- *Sequence*, *Branch*, *Loop*, *EventSynch*, *Event*

The *Sequence* construct is used for imposing an order on events. *Branch* is choice between a number of mutually exclusive events. *Loop* specifies event repetition while its guard remains true. *EventSynch* is used to synchronize an event in a Tasking Machine with an event in a Shared Machine. The *EventSync* construct allows the updates in a Tasking Machine and Shared Machine to be viewed as an atomic update. Synchronization is implemented as a subroutine call, with atomic (with respect to an external viewer) updates. The updates in the protected resource are implemented by a procedure call to a protected object. It is important to note that tasks communicate through shared resources, and not directly with each other. The *EventSync* construct facilitates subroutine parameter declarations, and substitution in calls, by pairing ordered Event-B parameter declarations. The *EventSynch* construct may also be used with a single event, in which case it is implemented as a subroutine call with no parameters. The event may belong to the Tasking Machine (local), a Shared Machine (remote), or both.

3.3 Events

Events can play one of several roles in the mapping to the implementation as follows,

- *ProcedureSynch*, *ProcedureDef*, *Branch*, *Loop*

Events with the *ProcedureSynch* extension can take part in event synchronization, whilst *ProcedureDef* indicates implementation as a parameterless subroutine call. The remainder are self explanatory.

3.4 Tasking Constructs

Events that are local to a Tasking Machine only update the Tasking Machine state; conversely events that are remote only update the state of the Shared Machine. Synchronised events share parameters to facilitate communication between Tasking and Shared Machines.

To represent the combined updates on local and remote machines we introduce synchronized event composition. The synchronization of the two events is equivalent to a single atomic event, with the guards and actions of the individual events merged. We can write the guards and actions of the events as guarded commands [8]. The general case of event synchronization is shown in Equation (1) where a local event is written as $g_l \rightarrow a_l$, and g_l and a_l are local guards and actions. The remote event is $g_r \rightarrow a_r$, where g_r and a_r are remote guards and actions. The synchronization of one local and one remote event uses the event composition operator $\|_e$. The actions describing state updates are composed with the parallel update operator $\|$.

$$g_l \rightarrow a_l \,\|_e\, g_r \rightarrow a_r \triangleq g_l \wedge g_r \rightarrow a_l \,\|\, a_r \tag{1}$$

The merged guard is the conjunction of the guards of the local and remote events, and the merged action is the parallel composition of the actions of the local and remote events.

The general case of event synchronisation may lead to undesirable behaviour if our implementation followed Event-B semantics in an unrestricted manner. For example, an event guard prevents an update of state until the guard is true. This is implemented as a subroutine with a conditional critical region [13]. In an implementation the calling task should not block itself, so the local guard g_l is redundant; the task state is not visible externally so the task would remain blocked. Therefore the definition of our synchronized event s omits the local guard, as shown in Equation (2).

In the current version of our language we define the *Branch* and *Loop* constructs with the restriction that guards are defined for the local event only. The definition of a simple branch b is shown in Equation (3), which makes use of the alternative choice operator $[]$. The same restrictions are applied to the loop construct thus when we define a loop it has the same form as the simple branch shown.

$$s = a_l \,\|_e\, g_r \rightarrow a_r \tag{2}$$

$$b = g_l \rightarrow a_l \,\|_e\, a_r \quad [] \quad \neg g_l \rightarrow SKIP \tag{3}$$

4 Preparing a Model for Implementation

Fig. 2 illustrates keys features of the following section, and we begin with the *AbstractBuffer write* event of Fig. 1. The buffer *buff* is used to transmit natural numbers, so a reader sets *buff* to -1 when it has read and removed the value; this signals that the writer can write and the reader is blocked. The *write* event is enabled when $buff < 0$; the action writes *wVal* to the buffer. We also keep track of the number of updates to the buffer with *sCount*. *wCount* will be the writer's copy of *sCount*. We keep track of the count of the writer's writes with *wCount2*. The counts *wCount* and *wCount2* may differ if further writer tasks are added. Using the existing decomposition tools [6, 19] we separate the variables into separate components during decomposition. The *ReadWriteBuffer* is the first refinement, where parameters are added in preparation for decomposition. The abstract and refined *write* events are shown in Appendix A.1 for reference.

We now focus on the first refinement of the *write* event, see Appendix A.1. We introduce shared parameters *p1* and *p2*, define $p1 = wVal$ in the guard, and make the assignment to *buff* using *p1*. We do the same for *wCount* and *p2*. The use of shared parameters is necessary for decomposition, since *buff* and *wVal* will reside in different machines, as will *wCount* and *sCount*.

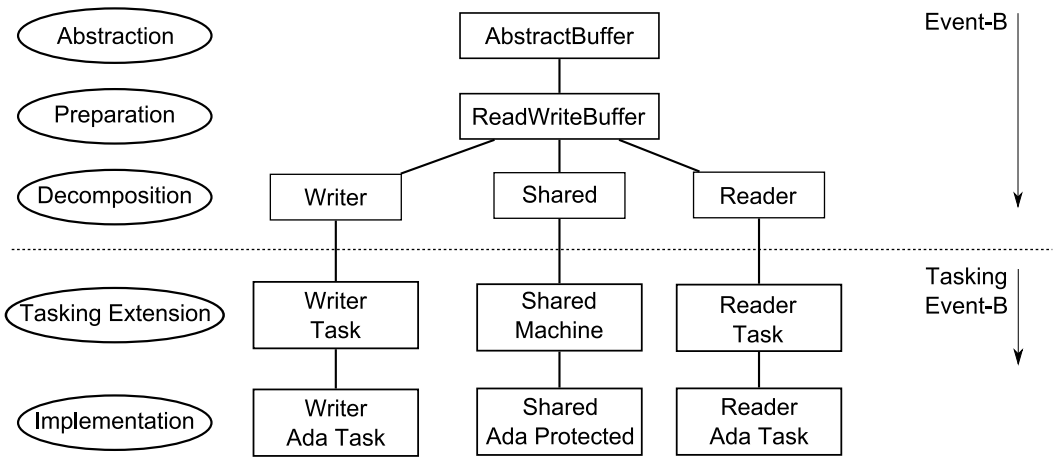

Figure 2: Linking Event-B to Code

We decompose into a writer, reader and shared machine using shared-event decomposition [6, 19]. We segregate the variables; *buff* and *sCount* become part of the shared machine, *wVal* and *wCount* become part of the writer machine. See Appendix A.2. Following decomposition, the *write* events in the writer and shared machines are not structurally linked in Event-B, but may synchronize in Tasking Event-B. The development, shown below the dashed line in Fig. 2, continues. We specify the task body, where we define the flow of control (see Appendix A.3). We have the *Sequence*, *Branch*, *Loop* and *EventSynch* constructs available to use; in our example we just use *Sequence* and *EventSynch*. Clause *w1* specifies that the writer's *write* event should synchronize with the shared machine's *write* event. This is sequentially composed with clause *w2*. The *write* event parameters are marked as *actualIn* and *actualOut* to indicate the type and direction of parameters in the implementation. The shared machine's *write* event is the same as its abstraction (thus not shown) except for the specification of formal in and out parameters.

5 Translation

We have developed tool support to translate the Tasking Event-B to Ada source, and a model of the implementation. The writer task body can be seen in Appendix A.4.

In the translation of the task body we map the tasking constructs to a Common Language Model. The Common Language Model is an abstraction of commonly used programming constructs, such sequence, loop, branch, subroutine call. It is useful to provide such an abstraction to simplify the task of translating to a number of different implementations. The tasking extension has both an Event-B semantics, and an operational semantics which makes use of the Common Language. The translator is then an implementation of the rule definitions. For instance a branch specification may be written

$$\begin{array}{l}\textbf{if } evt_1 \parallel_e evt_2\\ \textbf{else } evt_3 \parallel evt_4\\ \textbf{end}\end{array}$$

This is syntactic sugar for the following, where $evt_i = g_i \rightarrow a_i$ and $g_3 = \neg g_1$.

$$\begin{array}{l} g_1 \rightarrow a_1 \parallel_e a_2\\ []\\ g_3 \rightarrow a_3 \parallel_e a_4\end{array}$$

Then the definition of a branch translation to Common Language Model is as follows,

Event	Common Language Model
$g_1 \rightarrow a_1 \parallel_e a_2$	**if** g_1
[]	**then** a_1 ; a_2
$g_3 \rightarrow a_3 \parallel_e a_4$	**else** a_3 ; a_4
	end

6 Discussion

6.1 Correctness of the Translation

We have successfully used the tool to specify tasking developments, and generate code in several small case studies. The question of correctness of the generated code has not been formally addressed at present. However, we do have rules defining the operational semantics, and these are embodied in the translation to the Common Language Model. The tasking constructs are relatively simple, so we are able to check that the translator implementation corresponds to the rules. As an example we consider the atomic event which appears in a task body. This atomic event maps to a atomic subroutine call in the Common Language Model. In an Ada implementation we translate the atomic subroutine call to protected object call. The protected object acts as a monitor to enforce mutually exclusive access, and we assume that the Ada protected object correctly enforces this. The other constructs can be reasoned about in a similar manner, however, further work is being undertaken to address the issue of correctness of the translation.

Confidence in the correctness of the resulting code can be increased by the use of SPARKAda [1]. SPARKAda is an approach where Ada code is augmented with pre-, post, and assert conditions, and a verifier is used to prove that the code satisfies the conditions. Additional restrictions are placed on multi-tasking developments in accordance with the Ravenscar profile [5]. We would expect to be able to derive the SPARKAda annotations from the Tasking Event-B model, but this remains a task for the future.

6.2 Related Work

The motivation for this work was to facilitate the link between Event-B and multi-tasking implementations; with the specific aim of overcoming the shortcomings discovered in our previous work [11]. To our knowledge no other work has been undertaken to facilitate this. The closest comparable work is that of providing implementations for Classical-B [4] using the B0 implementation notation described in [7]. B0 is similar to a programming language, and consists only of concrete programming constructs that map to programming constructs in programming languages. B0 forms part of the Classical-B refinement chain, so the implementation level specification is shown to refine an abstract development. B0 is similar to our current work in that the final step between B0 specification and actual implementation code is verified by inspection rather than formal proof. It should be noted that, although B0 can be translated to various multi-tasking programming languages, there is no support for concurrency in B0.

In previous investigations we considered using a combined CSP [15] and Event-B approach to specify the order in which events occur and synchronize. In such a development, the specifications are combined so that Event-B events synchronize with CSP events with the same name. We considered this approach to be more complex than the approach we ultimately adopted, and we prefer a streamlined approach which uses only Event-B in combination with tasking extensions.

Another code generation approach for multi-tasking involves the use of CSP and Java, and is called JCSP [24, 17] and JCSProB [27]. JCSP links the OCCAM [18] subset of the CSP process algebra and the

Java programming language. The result is the ability to specify process behaviour in CSP, and translate to those to Java threads. The resulting Java is a message passing style implementation of communication between processes. This differs from the shared memory approach described in our work. JCSProB combines the CSP and classical-B formal methods, the *ProB* tool [16] can be used to provide a unified approach for specification and model checking. The most obvious difference between the *JCSProB* approach and Tasking Event-B is that our work is aimed at the more recent Event-B approach.

Other work involving JCSP is that of *Circus* [25], which is a combined approach using CSP and $Z - notation$ [20]. In a *Circus* specification the Z and CSP constructs are used to build a specification that is amenable to model checking using [26]. In this respect, *Circus* has more in common with *JCSProB* than Tasking Event-B since it is a combined approach using model-checking technology. *Circus* can be translated to Java as described in [12], making use of the JCSP library code.

There is also code generation for VDM++ which can be used to specify and implement multi-tasking systems. VDM++ is an object-oriented extension to VDM-SL formal specification language. Models can be described textually; or using a graphical interface using UML diagrams. The VDM++ Toolbox can be used to generate multi-tasking C++ and Java code. Conditional waiting can be specified using permission predicates; this corresponds to the Shared Machine guards used t specify blocking behaviour in Tasking Event-B.

7 Conclusions

We have developed an approach for generating source code from Event-B, specifically targeting multi-tasking, embedded, real-time systems. We have succeeded in achieving a small semantic gap between Event-B and the tasking extension through the use of a minimal set of constructs; this is achieved by restricting the use of event guards so that we can implement them using sequences, loops, branches, or procedure calls (with parameter passing). We keep models manageable using the decomposition approach; and allocating variables to machines during shared event decomposition automatically generates the parameters. We anticipate that the approach will be scalable because the decompositions can be performed repeatedly, whenever required, prior to application of the tasking extension. Development of each of the decomposed machines can then continue in isolation from the other components.

We extend the machine with a task body to define the control flow and generate source code from this, and an Event-B model of the implementation. Tasking Machines are implemented as Tasks, Shared Machines as protected objects, and event synchronizations as procedure calls. The approach has been applied to a multi-tasking read/write buffer, and a controller for a heating system.

In the current tool we are limited to integer and Boolean types; the tool is not yet fully integrated with the Rodin platform; and we have not yet implemented *triggered* tasks. The tool support will be developed further to add new types, and special sensing variables that allow sensing in tasks. We will investigate when is the best time in the development to apply the tasking extensions, and further formalize the approach.

References

[1] SPARKAda. Available at http://www.praxis-his.com/sparkada/index.asp.

[2] The DEPLOY Project Website. at http://www.deploy-project.eu/.

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

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

[5] Alan Burns, Brian Dobbing, and Tullio Vardanega. Guide for the use of the ada ravenscar profile in high integrity systems. *Ada Lett.*, XXIV:1–74, June 2004.

[6] M. Butler. Decomposition Structures for Event-B. In *Integrated Formal Methods iFM2009, Springer, LNCS 5423*, volume LNCS. Springer, February 2009.

[7] ClearSy System Engineering. *The B Language Reference Manual*, version 4.6 edition.

[8] E.W. Dijkstra. Guarded Commands, Non-determinacy and Formal Derivation of Programs. *Commun. ACM*, 18(8):453–457, 1975.

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

[10] A. Edmunds and M. Butler. Linking Event-B and Concurrent Object-Oriented Programs. In *Refine 2008 - International Refinement Workshop*, May 2008.

[11] A. Edmunds and M. Butler. Tool Support for Event-B Code Generation, 2010.

[12] A. Freitas and A. Cavalcanti. Automatic Translation from Circus to Java. In J. Misra, T. Nipkow, and E. Sekerinski, editors, *FM*, volume 4085 of *Lecture Notes in Computer Science*, pages 115–130. Springer, 2006.

[13] P.B. Hansen. Structured multiprogramming. *Commun. ACM*, 15:574–578, July 1972.

[14] C. A. R. Hoare. Monitors: An Operating System Structuring Concept. *Commun. ACM*, 17(10):549–557, 1974.

[15] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[16] M. Leuschel and M. Butler. ProB: A Model Checker for B. In *Proceedings of Formal Methods Europe 2003*, 2003.

[17] P.H.Welch, J.R. Aldous, and J. Foster. CSP Networking for Java (JCSP.net). In *Computational Science - ICCS 2002: International Conference*, 2002.

[18] SGS-Thomson Microelectronics Ltd. *Occam 2.1 Reference Manual*, 1995.

[19] R. Silva, C. Pascal, T.S. Hoang, and M. Butler. Decomposition Tool for Event-B. *Software: Practice and Experience*, 2010.

[20] J. M. Spivey. *The Z notation: A Reference Manual*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[21] T.S. Taft, R.A. Tucker, R.L. Brukardt, and E. Ploedereder, editors. *Consolidated Ada reference manual: language and standard libraries*. Springer-Verlag New York, Inc., New York, NY, USA, 2002.

[22] The Deploy Code Generation Wiki. at http://wiki.event-b.org/index.php/Code_Generation_Activity.

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

[24] P.H. Welch and J.M.R. Martin. A CSP Model for Java Multithreading. In *Software Engineering for Parallel and Distributed Systems*, 2000.

[25] J. Woodcock and A. Cavalcanti. A Concurrent Language for Refinement. In A. Butterfield, G. Strong, and C. Pahl, editors, *IWFM*, Workshops in Computing. BCS, 2001.

[26] Jim Woodcock, Ana Cavalcanti, and Leonardo Freitas. Operational Semantics for Model Checking Circus. In John Fitzgerald, Ian J. Hayes, and Andrzej Tarlecki, editors, *FM*, volume 3582 of *Lecture Notes in Computer Science*, pages 237–252. Springer, 2005.

[27] L. Yang and M. Poppleton. Automatic Translation from Combined B and CSP Specification to Java Programs. In J. Julliand and O. Kouchnarenko, editors, *B*, volume 4355 of *Lecture Notes in Computer Science*, pages 64–78. Springer, 2007.

A Appendix

A.1 Abstract and First refinement

event write
where
buff $<$ 0
then
buff := wVal
sCount := sCount + 1
wCount := sCount + 1
wCount2 := wCount2 + 1
end

event write **refines** write
any p1 p2
where
p1 = wVal
p2 = sCount + 1
buff $<$ 0
then
buff := p1
sCount := sCount + 1
wCount := p2
wCount2 := wCount2 + 1
end

A.2 Decomposed Machines

machine Writer
variables wVal wCount ...
invariants ...
events
event write
any p1 p2
where
p1 = wVal
then
wCount := p2
wCount2 := wCount2 + 1
end

machine Shared
variables buff sCount ...
invariants ...
events
event write
any p1 p2
where
p2 = sCount + 1
buff $<$ 0
then
buff := p1
sCount := sCount + 1
end
...

A.3 Tasking Machine Extension

machine WriterTsk **refines** Writer

tasktype periodic(250)
priority 5

taskbody is
w1: WriterTsk.write $\|_e$ Shared.write;
w2: ...

event write
is procedureSynch refines write
any
actualOut p1
actualIn p2
where
p1 = wVal
p1 $\in \mathbb{Z}$
p2 $\in \mathbb{Z}$
then
wCount := p2
end

A.4 Ada Task Code

```
task body WriterTsk is ...
loop
   t := clock;
   write;
   shared.write(wVal, wCount);
   ...
   delay until t + period;
end loop;
```

```
protected body Shared is
   entry write(p1: in Integer; p2: out Integer)
   when buff < 0 is
   begin
      p2 := sCount + 1;
      buff := p1;
      sCount := sCount + 1;
end;
```