From Event-B Models to Code: Sensing, Actuating, and the Environment

A. Edmunds[1], A. Rezazadeh[2] and M.J. Butler[3]

[1] ae2@ecs.soton.ac.uk
[2] ra3@ecs.soton.ac.uk
[3] mjb@ecs.soton.ac.uk

Abstract. The Event-B method is a formal approach for modelling systems in safety-, and business-critical, domains. We focus, in this paper, on multi-tasking, embedded control systems. Initially, system specification takes place at a high level of abstraction; detail is added in refinement steps as the development proceeds toward implementation. In previous work, we presented an approach for generating code, for concurrent programs, from Event-B. Translators generate program code for tasks that access data in a safe way, using shared objects. We did not distinguish between tasks of the environment and those of the controller. The work described in this paper offers improved modelling and code generation support, where we separate the environment from the controller. The events in the system can participate in actuating or sensing roles. In the resulting code, sensing and actuation can be simulated using a form of subroutine call; or additional information can be provided to allow a task to read/write directly from/to a specified memory location.

1 Introduction

Event-B [1] can be used to model both single and multi-tasking software systems. In previous work [2], we described our extension to Event-B for generating code for multi-tasking implementations, using Tasking Event-B. Our approach focussed on modelling the tasks and shared objects, and this gave rise to models that were difficult to implement. This paper describes the methodology and tool support that we introduced, to improve the approach. We chose Ada [3] as the target programming language, but other languages are also viable targets.

The work reported here has been undertaken as part of the EU DEPLOY [4] project, where our target domain is multi-tasking, real-time embedded systems. In our current approach, to comply with Ravenscar [5], we have found that restrictions on communication between tasks lead to designs that did not accurately reflect the interaction with the environment. The environment was forced to communicate with tasks through protected objects. We relax this restriction for the environment; we model the environment separately, and introduce new constructs to model sensing and actuating. Sensing and actuating events model the interaction with the environment, and are easy to map to an implementation.

2 Event-B

The Event-B method [1] was developed by J.R. Abrial, and uses set-theory, predicate logic and refinement to model discrete systems. The basic structural elements of Event-B models are contexts and machines. Contexts are used to describe the static aspects of a system, using sets and constants; the relationships between them are specified in the axioms clause. Machines are able to *see* Contexts; the content of a Context is visible and accessible to a machine. Machines are used to describe the dynamic aspects of a system, in the form of state variables, and guarded events, which update state. System properties are specified using the invariants clause. 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 auto-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. A fragment of an Event-B specification is shown in Fig. 1. The specification has variables, which are typed in the **invariant**. Invariants also describe desired safety properties. The event declares two parameters *tm1* and *tm2*. These are typed in the **where** clause. The third clause describes an enabling condition for the event. State updates are specified in the **then** (action) clause; they may do nothing (skip); or, contain deterministic or non-deterministic assignments.

Decomposition is a known technique, used to handle complexity; we make use of shared event decomposition [6,7]. Following decomposition, machines are identified as Tasking, or Shared. Event Synchronization uses shared parameters to facilitate communication between Tasking and Shared Machines. In [2] we describe the synchronization of the two events as being equivalent to a single, merged, atomic event. 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.

```
                                 event TurnON_Heat_Source
machine HCtrl_M0                 any tm1 tm2
sees HC_CONTEXT                  where
variables avt stm1 ...             tm1 ∈ ℤ
invariants                         tm2 ∈ ℤ
  avt ∈ ℤ                          tm1 < tm2
  stm1 ∈ ℤ                       then
    ...                            hsc := TRUE
                                 end
```

Fig. 1. Example of Textual Event-B

3 Tasking Event-B

Tasking Event-B [2] is an extension to the Event-B; but some Event-B elements are restricted to implementable constructs. Using decomposition, we partition the system into its components parts. We then introduce implementation specific constructs, and these are used to guide code generation. An Event-B operational semantics underpins the extension. We have a demonstrator tool [8] for generating Ada code, which integrates with the existing Rodin platform [9]. In this section, we discuss two types of machine extensions, namely AutoTask and Shared Machines. AutoTask Machines model *controller* tasks (in the implementation), and are related to the concept of an Ada [10] task (but we are not restricted to Ada implementations). Shared Machines are related to the concept of a protected resource, such as a monitor [11]. An example AutoTask Machine, from our case study [12,13], is shown in Fig. 2 (it is a descendant of the fragment shown in Fig. 1, after refinement and decomposition). Here we see the **taskbody** clause, with flow control operators: sequence, branch, and synchronization (";", **IF**, and $\|_e$). Synchronization arises from shared event decomposition; where updates in the AutoTask, and Shared, machines can be viewed as a single atomic update. This is implemented as a protected subroutine call.

Events can play one of several roles in the mapping to the implementation. Events can take part in event synchronization, parameterless subroutine call, part of a branch, or part of a loop. The Output construct is provided to allow text output to a console during simulation. Events labels may be specified by the developer (*tc*4 etc. in Fig. 2) to generate program counters, these can be used in invariants to specify system properties. Events of an AutoTask Machine only update the AutoTask Machine state; events of Shared Machines only update that machine's state. Synchronised events share parameters to facilitate communication between AutoTask and Shared Machines.

```
machine Temp_Ctrl_TaskImpl
is autoTask
refines Temp_Ctrl_Task

tasktype periodic(250)
priority 5
taskbody is
  ...
  tc4: TCGet_Target_Temperature2
       ||e SOGet_Target_Temperature2;
  tc5: IF TCTurnON_Heat_Source
       END IF
       ELSE TCTurnOFF_Heat_Source
       END ELSE;
  tc6: ...

event TCTurnOn_Heat_Source
is branch
refines TurnOn_Heat_Source
when
  avt < cttm2
then
  hsc := TRUE
end
```

Fig. 2. An Example AutoTask Machine

4 Adding the Environment Model to Tasking Event-B

We extend the previous approach [2] with *Environ* machines, to model the environment. We add *sensing* and *actuating* roles for events, to Tasking Event-B. *Environ* Machines are implemented as Ada tasks. The tasks communicate with *controller* tasks, using Ada's rendezvous mechanism, *entries*. We are able to identify *Environ* machines after refinement and decomposition of the abstract system. The system can be partitioned into AutoTask, Shared and Environ machines. AutoTask and Environ Machines model communication using the synchronized events that arise from decomposition. The developer determines which events take part in *sensing* and *actuating*, and applies the annotations. The translator generates environment tasks to simulate external changes of state in the environment. The translator also generates task entries; these can be called by the *controller* tasks. In the case of actuation, the task entries are used to modify the state of the environment. In the case of sensing, task entries are used to read values from the environment. Note that, in current work, we do not provide a formal semantics for the extension, due the small semantic gap between model and implementation. We have, however, done work to show that the model is correctly implemented, in terms of its updates to variables. Since AutoTask and Environ Machines share the same Tasking Event-B constructs, the semantics of the latter does not differ greatly from the former.

5 Writing Directly to Memory Locations

The discussion, so far, has focussed on a simulation of the environment whereby the task communicates with the environment using a subroutine call. In Ada, this is implemented as an entry call to the environment task. It may, however, be the case that the developer can specify some memory locations to read from, and write to, directly (corresponding to memory mapped I/O). To achieve this, our approach allows developers to annotate event parameters, and environment variables, with the address information. We have introduced the *addr* annotation, where we specify a memory location and its number base. In the Tasking Event-B specification, the parameter *t1* is given the address *ef14* and is in base 16 by writing,

```
any
  actualIn addr(16, ef14) t1
where...
```

The parameter *t1* is mapped to an integer variable declaration. The address of the variable has been set using the following statement:

```
t1: Integer;
for t1'Address use System'To_Address(16#ef14#);
```

In Ada this is known as an *address clause*. So, *t1* is used in the code shown in Fig. 3; using the variable defined in the address clause, the assignment reads

```
task body Temp_Ctrl_Task1Impl is
  stm1 : Integer := 0;
  t1: Integer;
  for t1'Address use System'To_Address(16#ef14#);
  . . .
  procedure TCSense_Temperatures is
  begin
    stm1 := t1;
    stm2 := t2;
  end;
  . . .
end Temp_Ctrl_Task1Impl;
```

Fig. 3. Implementation of Addressed Variables

from the memory location, and assigns the value to *stm1*. The task updates its value without a call to an external environment subroutine; so we can discard the environment task, when it comes to the deployment. In a similar translation to C, we would declare a pointer to integer type, **int*** t1 = (**int***) 0xef14, and use assignment stm1 = *t1 .

6 Conclusions

Code generation is a two-step process; firstly generating a common language model (CLM), from which a number of target implementations could be generated. Translation to a specific target language takes place in the second step. Our tool generates a pretty print of Ada code; in future, we will output Ada files. We successfully compiled and ran the generated code, with no code modifications.

In previous work [2] we developed an approach for automatically generating source code, from an extended Event-B model. In this paper, we describe an extension that enables us to separate the environment from the remainder of the system. The partitioning of the tasks into AutoTask, Shared, and Environ machines, using Event-B decomposition, is quite intuitive. The annotations facilitate automatic code generation. Event synchronization models the interaction between the environment tasks, and *controller* tasks. We produced a case study of heating controller during the DEPLOY project [4] available to see at [13].

There are a number of avenues that may be explored in future work. For instance, multiple Environ machines could be used to model each device in the environment; the task entries may provide the basis for a link with device driver APIs. We would like to investigate the use of Simulink to model the environment; and generating SPARKAda from Tasking Event-B would be interesting. We will also add support for interrupts.

In related work, the closest comparable work is that of Classical-B's code generation approach using B0 [14]. B0 consists of concrete programming constructs that map to programming constructs in target programming languages. B0 can be translated to Ada, but there is no support for concurrency. VDM++ [15] may be used to generate code. It has been used to model real-time systems, see [16]. They model time, and asynchronous communication. We do not address these issues in our work since the research is ongoing. Scade [17] is an industrial tool for formally modelling embedded systems. It provides a graphical approach to specification, and has a similar control flow approach to that of UML-B state machines [18]. We hope to investigate code generation from UML-B in the near future.

References

1. Abrial, J.R.: Modeling in Event-B: System and Software Engineering. Cambridge University Press (2010)
2. Edmunds, A., Butler, M.: Tasking Event-B: An Extension to Event-B for Generating Concurrent Code. In: PLACES 2011. (2011)
3. Barnes, J.: Programming in Ada 2005. International Computer Science Series. Addison Wesley (2006)
4. The DEPLOY Project Team: Project Website. (at http://www.deploy-project.eu/)
5. Burns, A., Dobbing, B., Vardanega, T.: Guide for the use of the Ada Ravenscar Profile in high integrity systems. Ada Lett. **XXIV** (2004) 1–74
6. Butler, M.: Decomposition Structures for Event-B. In: Integrated Formal Methods iFM2009, Springer, LNCS 5423. Volume LNCS., Springer (2009)
7. Silva, R., Pascal, C., Hoang, T., Butler, M.: Decomposition Tool for Event-B. Software: Practice and Experience (2010)
8. The Deploy Code Generation Wiki. (at http://wiki.event-b.org/index.php/Code_Generation_Activity)
9. The RODIN Project. (at http://rodin.cs.ncl.ac.uk)
10. Taft, T., Tucker, R., Brukardt, R., Ploedereder, E., eds.: Consolidated Ada reference manual: language and standard libraries. Springer-Verlag New York, Inc., New York, NY, USA (2002)
11. Hoare, C.A.R.: Monitors: An Operating System Structuring Concept. Commun. ACM **17**(10) (1974) 549–557
12. Edmunds, A., Rezazedah, A.: Event-B Wiki: Development of a Heating Controller System. (at http://wiki.event-b.org/index.php/Development_of_a_Heating_Controller_System)
13. Edmunds, A., Rezazedah, A.: Event-B Project Archives: Tasking Event-B Tutorial. University of Southampton. (at http://deploy-eprints.ecs.soton.ac.uk/304/)
14. ClearSy System Engineering: The B Language Reference Manual. (Version 4.6 edn.)
15. CSK Systems Corporation: (The VDM++ Language Manual)
16. Verhoef, M., Larsen, P.G., Hooman, J.: Modeling and Validating Distributed Embedded Real-Time Systems with VDM++. In Misra, J., Nipkow, T., Sekerinski, E., eds.: FM. Volume 4085 of Lecture Notes in Computer Science., Springer (2006) 147–162
17. Berry, G.: Synchronous Design and Verification of Critical Embedded Systems Using SCADE and Esterel. In Leue, S., Merino, P., eds.: FMICS. Volume 4916 of Lecture Notes in Computer Science., Springer (2007) 2
18. Snook, C., Butler, M.: UML-B: A Plug-in for the Event-B Tool Set. In Börger, E., Butler, M.J., Bowen, J.P., Boca, P., eds.: ABZ. Volume 5238 of Lecture Notes in Computer Science., Springer (2008) 344