Building on the DEPLOY Legacy: Code Generation and Simulation

A. Edmunds[1] and J. Colley and M. Butler

[1]University of Southampton, UK
ae2@ecs.soton.ac.uk

Abstract. The RODIN, and DEPLOY projects have laid solid foundations for further theoretical, and practical (methodological and tooling) advances with Event-B; we investigated code generation for embedded, multi-tasking systems. This work describes activities from a follow-on project, ADVANCE; where our interest is co-simulation of cyber-physical systems. We are working to better understand the issues arising in a development when modelling with Event-B, and animating with ProB, in tandem with a multi-simulation strategy. With multi-simulation we aim to simulate various features of the environment separately, in order to exercise the deployable code. This paper has two contributions, the first is the extension of the code generation work of DEPLOY, where we add the ability to generate code from Event-B state-machine diagrams. The second describes how we may use code, generated from state-machines, to simulate the environment, and simulate concurrently executing state-machines, in a single task. We show how we can instrument the code to guide the simulation, by controlling the relative rate that non-deterministic transitions are traversed in the simulation.

1 Introduction

Event-B [2], and supporting tools have been developed during the RODIN and DEPLOY [9] projects, and continues in ADVANCE [8]. Some industrial partners were interested in the formal development of multi-tasking, embedded control systems. We developed an approach for automatic code generation, from Event-B models, for these type of systems [3]. Event-B uses set-theory, predicate logic and refinement to model discrete systems. The basic structural elements of Event-B models are contexts and machines. Contexts describe the static aspects of a system, using sets, constants, and axioms. The contents of a Context can be made visible to a machine. Machines describe the dynamic aspects of a system, in the form of state variables, and guarded events, which update state. Required properties are specified using the invariants clause. The invariants give rise to proof obligations.

State-machine diagrams [1] can be added to a machine. Each contains an initial state, typically contains one or more transitions, one or more other states, and possibly a final state. A transition 'elaborates' one or more events; that is, a transition describes the atomic state updates that occur during the change from

one state to the next. We use an example of an automotive engine stop-start controller, loosely based on [5], to illustrate our approach. The system aims to save fuel by switching the engine off when the car is stationary. Fig. 1 is an example of a state-machine diagram, *EngMode*. Initially the state-machine is in the *ENG_OFF* state, and may go the *ENG_CRANKING* state via transitions *s1* or *userStart*, and so on. In the properties we define 'translation type' as *Enumeration*. The underlying Event-B model, uses a set-partition of the states, as shown below. The current state of the state-machine is recorded in a variable $EngMode \in EngMode_STATES$, where $EngMode_STATES$ is a partition of the states of the EngMode state-machine,

$$\begin{aligned} &partition(EngMode_STATES, \{ENG_STOPPING\}, \\ &\quad \{ENG_CRANKING\}, \{ENG_RUNNING\}, \{ENG_OFF\}) \end{aligned} \tag{1}$$

In this paper we describe how we extend the code generation work of DEPLOY;

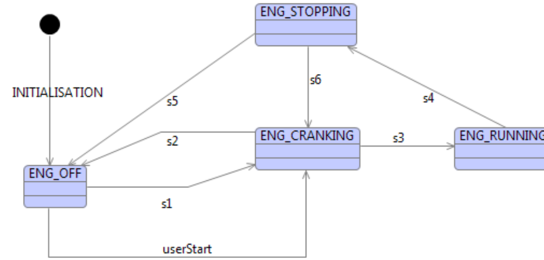

Fig. 1. EngMode State-machine

we add the ability to generate code from Event-B state-machine diagrams. We then describe how we may use code, generated from state-machines, to simulate the environment, and simulate concurrently executing state-machines, in a single task. We describe how we guide the simulation, that is, control the relative rate that non-deterministic transitions are traversed, using additional guards on the transition implementations.

1.1 Tasking Event-B

Tasking Event-B [3,4] is an extension to the Event-B; where Event-B elements are restricted to implementable types. If required we use decomposition [6,7] to separate the system into sub-components. At an appropriate stage we introduce implementation specific constructs to guide code generation. These constructs are underpinned by Event-B operational semantics; Tasking Event-B introduces three main constructs:- AutoTask, Environ, and Shared Machines. AutoTask Machines model controller tasks (in the implementation). Environ Machines model the environment, and Shared Machines provide a protected resource for sharing data between tasks.

Tasks bodies are specified using the syntax shown in Fig. 2. We can use (;) sequence , (if-elsif-else) branching, (do) looping, and text output to the console.

```
TaskBody ::=
  TaskBody ; TaskBody
  | if EventName
    (elseif EventName)*
     else EventName
  | while EventName
  | output String VariableName

EventName ::= String

VariableName ::= String
```

Fig. 2. Task Body Syntax

1.2 Translation of a Task Body

To simplify the discussion, our example uses a single tasking approach. We will not consider here the issue of multi-tasking. We therefore need only to give a brief overview of AutoTask Machine translation, since it will not be synchronized with a Shared Machine. Given an event $E \triangleq g \to a$, we map action a to a program statement $a\prime$, and guard g to a condition $g\prime$, if g exists. The guard should be $\top$ for events used in sequences, but may be any implementable predicate for use in branching and looping statements. An example translation of branching follows, where events $e_1 \triangleq g1 \to a_1$ and $e_2 \triangleq g_2 \to a_2$, are used in the task body,

$$\mathbf{if}\ \ e_1\ \mathbf{else}\ e_2\ \mathbf{endif}$$
$$\leadsto$$
$$\mathbf{if}\ g\prime_1\ \mathbf{then}\ a\prime_1\ \mathbf{else}\ a\prime_2\ \mathbf{end\ if};$$

The branching construct of the task body contains events e_1 and e_2, and translates to a branching construct in the program code. The guard $g\prime_2$ does not appear in the code, but a proof obligation can be generated to ensure that $g_2 = \neg g_1$. The code generator could be augmented to automatically generate proof obligations to show that branch guards are disjoint and complete.

2 The Automotive Stop-Start Model

A typical approach to multi-tasking in hybrid systems, relies on a *write-read-process* protocol. The shared variable store, shown in Fig. 3, is used by the various modules; to write to, then read from. In such a system, each task keeps a local copy of the parts of the state that it needs to deal with. In the *write-read-process* protocol, all tasks write to the store, all tasks then read from the store. Only when all tasks have updated their local copies of shared state, can processing take place. The task iterates these steps in a loop. In our tool we

simulate the concurrent implementation using *sequential* code generated from a single AutoTask Machine. The deployable modules of Fig. 3 can be implemented in a multi-tasking environment if the execution order of the protocol is preserved.

In our sequential simulation, we use a single AutoTask Machine, which contains both controller and environment state-machines; and define write and read behaviour in the machine's task-body construct. We have already seen the Stop-Start (SSE) system's *EngMode* state-machine, in Fig. 1. In addition to this we have Clutch, Gear and Steering environment state-machines. There are three controller modules, the SSE Module which decides whether to issue stop or start commands based on the engine state. It receives output from the HMI Controls module. HMI Controls monitors the clutch, gear, and steering controls to see if automatic stop or start should be enabled.

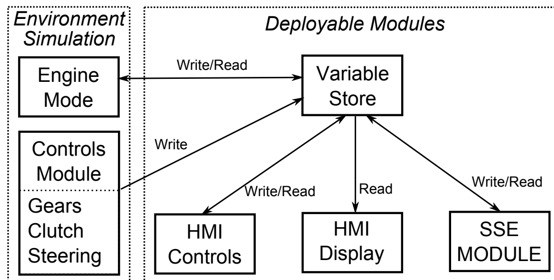

Fig. 3. Overview of the Stop-Start Architecture

2.1 The Task-body

We have defined the state-machines of the system and we can now specify the IO between the modules via the shared variable store. The store contains a copy of all of the variables involved in IO between modules. Each of the modules may send data and receive data from the variable store. If we take, as an example, the engine's IO, we output the engine state and speed to the shared variable store. All variables in the store are prefixed 'STO_', and variables in the engine module (other than state names) are prefixed 'ENG_', so the following event updates the shared variable store's copies of the engine state. Each state-machine has a send (*write*) and receive (*read*) event which has the state-machine name and *send* or *recv* as a suffix.

$$\begin{aligned} \mathit{Eng_send} \triangleq\ & STO_EngMode := EngMode \\ & \parallel \mathit{STO_EngineSpeed} := \mathit{ENG_EngineSpeed} \end{aligned}$$

2.2 Modelling Starting and Stopping the Engine

The *EngMode* state-machine keeps track of the engine mode, i.e. off, running, cranking, or stopping. The engine is initially in the *ENG_OFF* state. We model

the ultimate task of the SSE system, the automatic engine start, with the *s1* event. This is enabled after receiving an engine start order from the Stop-Start Controller module (the SSE Module's SSEMode state-machine, introduced later). The *s1* event follows,

$$s1 \triangleq \textbf{when } EngMode = ENG_OFF \wedge ENG_Start_Order = TRUE \\ \textbf{then } EngMode := ENG_CRANKING \\ \textbf{end}$$

The predicate and action involving *EngMode* are automatically generated in the translation from the state-machine diagram. The guard with *ENG_Start_Order* is added by the developer to indicate that the engine should enter the cranking state when a Start Order has been received. The engine may also be started manually, as modelled by the userStart event. When the engine is running at a sufficient rate $s3$ sets the engine state to *ENG_RUNNING*,

$$s3 \triangleq \textbf{when } EngMode = ENG_CRANKING \\ \wedge Eng_EngineSpeed >= Eng_Idle_Speed \\ \textbf{then } EngMode := ENG_RUNNING \\ \textbf{end}$$

When the engine is running it can be stopped automatically by the SSE module setting the guard of $s4$. The *HMI_Controls* module checks to see if it is in neutral gear, steering not-used, and clutch released. If it is, *HMI_Stop_EnaT* sets *HMI_Stop_Ena* to true. This eventually gets passed to the *SSEMode module* via the shared store.

$$HMI_Stop_EnaT \triangleq \\ \textbf{when } HMI_Gear = NEUTRAL \wedge HMI_Steer = NOT_USED \\ \wedge HMI_Clutch = RELEASED \wedge HMI_ControlsSM = HMI_OPERATION \\ \textbf{then } HMI_Stop_Ena := TRUE \parallel HMI_Strt_Req := FALSE \\ \textbf{end}$$

Event $t7$, of the SSE Module, copies its start command to the variable store if SSE_Stop_Ena, and the other guards, are satisfied. It is then read by the engine.

$$t7 \triangleq \\ \textbf{when } SSEMode = SSE_OPERATION \wedge SSE_Stop_Req = TRUE \\ \wedge SSE_EngMode = ENG_RUNNING \wedge SSE_Stop_Ena = TRUE \\ \textbf{then } SSEMode := SSE_STOPPING \parallel SSE_Stop_Order := TRUE \\ \parallel SSE_Start_Order := FALSE \\ \textbf{end}$$

2.3 The Task Body

We specify the sequence of events in the Task Body in the 'usual' Tasking Event-B style, seen in Fig. 4. We have specified that send events occur before the read events. This is necessary to ensure the latest state is made available for the state-machine evaluation. The Task Body is periodic, and generates a loop in the implementation. The order of processing is as follows: 1) Initialisation of state. 2) Evaluate state-machines. 3) Send updated values to the variable store. 4) Read updated values from the variable store; then go to 2, and repeat. The sequence {3,4,2}, in the task body, corresponds to the *write-read-process* protocol, which follows initialisation. Fig. 4 also shows the *output* clause, for text output to the console. The next section provides details of the translation to Ada code.

```
TASK BODY
ENG_send ;
CON_send ;
SSE_send ;
HMI_send ;
DIS_send ;
ENG_recv ;
SSE_recv ;
HMI_recv ;
DIS_recv ;
output ".ENG_Start_Order " ENG_Start_Order ;
output "..ENG_Stop_Order " ENG_Stop_Order ;
output "...EngMode        " EngMode ;
output "....SSE Lamp    " DIS_Info_Lamp
```

Fig. 4. The Task Body Specification

3 Translating State-Machines to Ada Code

To illustrate the translation process we show the Ada implementation, we have seen how state-machine states are modelled by an enumeration partition, and we use this in the implementation. The partition of Equation 1 is translated to the following Ada code.

$$\begin{array}{l} \textbf{package}\ StopStart01b_Globals\ \textbf{is} \\ \textbf{type}\ EngMode_STATES\ \textbf{is}(\\ \quad ENG_STOPPING,\ ENG_CRANKING, \\ \quad ENG_RUNNING,\ ENG_OFF); \ldots \end{array}$$

We create the package *StopStart01b_Globals* to store the global constants and types. The type *EngMode_STATES* is an enumeration of the state-machine states. Recall also, that we generate a state variable $EngMode$ which is typed as $EngMode \in EngMode_STATES$, to keep track of the state; it has the initial value *Eng_OFF*. We use the diagram and the initialisation event to generate the following code:

$$EngMode : EngMode_STATES := ENG_OFF;$$

The main program invokes the state-machine implementations in a loop, once per cycle. Each state-machine diagram maps to a procedure. State-machine procedures are called exactly once before the sends to, and reads, from the variable store. The evaluation of each state-machine procedure is independent of the others state-machines, since each keeps a local copy of the state, copied from the variable store. Each state-machine procedure has a state variable v, states s_i, and implemented actions a_i. To each state-machine procedure, we add to a *case* statement,

case v **is when** $s_1 => a_1$;
when $s_2 => a_2;\ \ldots$
when $s_n => null$;

Translation of our example gives rise to the following code,

```
procedure EngModestateMachine is
begin
case EngMode is
  when ENG_STOPPING =>
  if ((ENG_EngineSpeed = 0)) then
    EngMode := ENG_OFF;
  elsif ((ENG_Start_Order = true)) then
    EngMode := ENG_CRANKING;
  else null;
  end if;
  when ENG_CRANKING =>
  if ((ENG_EngineSpeed = 0)) then
    EngMode := ENG_OFF;
  elsif ((ENG_EngineSpeed >= Eng_Idle_Speed)) then
    EngMode := ENG_RUNNING;
  else null;
  end if;
  when ENG_RUNNING => ...
end case;
end EngModestateMachine;
```

We can see that each of the case's *when* statements contains a branching statement. This is because each state of the state-machine has at least two branches; a do-nothing transition, plus one or more outgoing transitions. The do-nothing transition is not explicitly shown on the diagram. A do-nothing transition can be added to each state, since adding a *skip* event is a valid refinement. It is implemented by the **else null;** branch. Other branches are translated from states with

more than one outgoing transition. This may be seen in the *ENG_STOPPING* case in the example. The branch conditions are mapped from the guards of the events ($s5$ and $s6$) that elaborate the outgoing transitions.

4 Manipulating State Machine Transitions

The generated code from our example is compiled to an executable file and run. In essence we have generated implementable code for the controller state-machines, and a simulation of the environment from the environment state-machines. When we run the code we find that most of the state remains unexplored, and this is due to the non-determinism in the state-machines. This section identifies how we can guide a simulation, by reducing the non-determinism in the state-machines.

For the controller state-machines, each state's outgoing transitions are disjoint and complete; in other words, a transition is always taken in the simulation. However, in the environment, it is unlikely that the clutch changes state so frequently. We do have the implicit *do-nothing* transition on environment state-machine states, but we need this to happen more often than the other transitions. We must have some control over the relative rate that non-deterministic transitions are traversed in the simulation. As it stands, any outgoing transition is equally likely to occur. To solve this in the simulation, we introduce an enabling variable $q \in 0 \mathrel{..} n$ and a random variable $r \in 0 \mathrel{..} n$, and use the random variable in a case-statement's branch conditions. Variable q is calculated once at the beginning of the simulation, but a new random variable r is calculated at each state-machine evaluation. The event $g \mapsto a$ in Event-B terms is implemented as a branch $g \wedge r = q \mapsto a$ in a case-statement.

We now suggest how we may generate, and use the variables q and r in simulation. This aspect is work in progress, but we believe the approach will be useful for generating test scenarios, and therefore will help to achieve full test coverage. By adding a guard to the branch condition we can influence the path taken through the code during simulation. In effect, we reduce the non-determinism in the state-machines, which allows us to guide the simulation, and therefore the exploration of the state-space.

One question is, how to choose a value of n? We could base it on the total number of outgoing transitions of the state involved, but this would not give a large enough value. A typical state may have four transitions so a random number $r \in 0 \mathrel{..} 4$ could be used. However, we wish to manipulate the probability of a branch being taken, so that a branch is very unlikely to be taken; therefore, a much larger value for n is required. So, we calculate n based on the number of tests that would be required, for test coverage of all transitions, in all states. Likewise, the value of q must be unique within the case-statement; we just allocate an arbitrary, but unique value, close to n. In future work we will investigate how we could modify n during simulation runs, and use this value to reduce the probability of a simulation traversing previously explored state. In the code fragment below, we add the probabilistic condition to the branch of

the case-statement, where $r = StartStop01b_random$ ($StartStop01b_random$ is a random variable in the implementation code) and $q = 3990$.

case $EngMode$ **is**

when $ENG_STOPPING =>$

if $((ENG_EngineSpeed = 0))$ *and* $(StopStart01b_random = 3990)$ **then**

$EngMode := ENG_OFF; \ldots$

Adding the branch condition gives us control over the likelihood that a particular transition from a state will be taken when the state-machine is evaluated. We manually modify the conditions, to affect the behaviour of the simulation. We may wish to focus on exploring the state in a particular region. For instance, to test an engine-stop scenario, we require that the engine is in the ENG_ RUNNING state, the gear is in NEUTRAL, the clutch is in the RELEASED state, and the steering NOT_USED. Fig. 5 shows that we want large probabilities of transitions leading to the states that we want, and small ones departing.

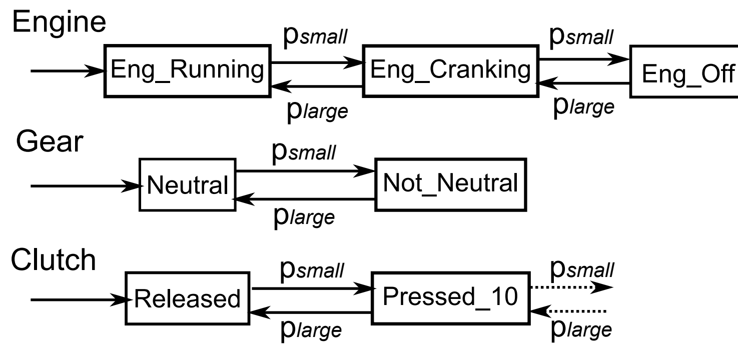

Fig. 5. Controlling the Simulation

For a given simulation run we can define *attracting* and *repelling* states. Here, ENG_RUNNING is an attracting state; that is, we want the state-machine to be in that state or moving towards it most of the time. To achieve this we can adjust the branch conditions, to increase the probability of the transitions that lead to that state, being taken. For instance to increase the probability of the engine going from ENG_OFF to ENG_CRANKING we can modify the statement to read (StopStart01b_random <= 3990). In addition to this, we propose to record the navigated transitions, for transition coverage analysis. So, we will be able to use the data also, to guide the simulation. We show two simulation runs here, with the text output defined in the Task Body, *Run1* uses the 'unmodified', generated code; it simply loops and never reaches the ENG_RUNNING state. With the branch conditions modified, as described, *Run 2* shows the simulation cycling from $ENG_RUNNING$ to ENG_OFF; and with the indicator lamp changing to inform the driver of the situation.

Run 1
.ENG_Start_Order FALSE ..ENG_Stop_Order FALSE ...EngMode ENG_OFF ....SSE Lamp OFF . . .

Run 2
.ENG_Start_Order FALSE ..ENG_Stop_Order TRUE ...EngMode ENG_RUNNING ....SSE Lamp OFF .ENG_Start_Order FALSE ..ENG_Stop_Order TRUE ...EngMode ENG_STOPPING ....SSE Lamp ORANGE_STOP .ENG_Start_Order FALSE ..ENG_Stop_Order TRUE ...EngMode ENG_OFF . . .

5 Conclusions

We have shown how we generate Ada code from State-machines, and illustrated the approach with a case study based on an automotive engine controller, automatic stop-start system. We describe how we simulate the environment, and a multi-tasking implementation. We gain an insight into how we adjust the conditions to provide meaningful simulation runs. In future work we intend to record the transition coverage, and feed this back to the simulator, to ensure all transitions are covered. We will also investigate the interaction between the generated code, environment simulations, and ProB.

References

1. Event-B State Machines. Details available at http://wiki.event-b.org/index.php/Event-B_Statemachines.
2. J. R. Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, 2010.
3. A. Edmunds and M. Butler. Tasking Event-B: An Extension to Event-B for Generating Concurrent Code. In *PLACES 2011*, February 2011.
4. A. Edmunds, A. Rezazadeh, and M. Butler. Formal Modelling for Ada Implementations: Tasking Event-B. In *Ada-Europe 2012: 17th International Conference on Reliable Software Technologies*, June 2012.
5. R. Gmehlich, F. Loesch, K. Grau, J.C. Deprez, R. de Landtsheer, and C.Ponsard. DEPLOY Deliverable D38, D1.2 Report on Enhanced Deployment in the Automotive Sector. Technical report, Robert Bosch GmbH and CETIC, 2011.
6. R. Silva, C. Pascal, T.S. Hoang, and M. Butler. Decomposition Tool for Event-B. *Software: Practice and Experience*, 2010.
7. Renato Silva and Michael Butler. Shared event composition/decomposition in event-b. In *FMCO Formal Methods for Components and Objects*, November 2010. Event Dates: 29 November - 1 December 2010.
8. The Advance Project Team. The Advance Project. Available at http://www.advance-ict.eu.
9. The DEPLOY Project Team. Project Website. at http://www.deploy-project.eu/.