Generating FMI-Standard Code for Co-simulation with Event-B

Andrew Edmunds and Michael Butler

University of Southampton

Abstract. FMI is a tool-independent standard that supports co-simulation of multiple models using different simulation tools. Recent work has enabled Event-B formal models, simulated using the ProB tool, to be co-simulated with models in other simulation tools. This means that discrete Event-B models may be co-simulated with continuous models in other languages. Existing code generation for Event-B supports generation of embedded controller implementations. This paper presents an adaption of this Event-B code generation to conform to the FMI interface. This adaptation allows controller code generated from Event-B models to be tested against continuous models of the controller environment.

Event-B; Code Generation; Co-simulation; Cyber-Physical Systems; Formal Modelling

1 Overview

Rodin is a tool platform [2] for the rigorous specification of critical systems, using the Event-B approach [1]. The tool was developed in the RODIN project [13], and experience with industry gained in the DEPLOY project [18]. Our work has been funded by the ADVANCE project [17], a continuation of the research effort, focussing now on co-simulation of Cyber-Physical Systems. We are exploring the interface between discrete, Event-B modelling, and simulation of continuous systems. However, this paper only describes our work on translations from discrete Event-B models, to discrete implementations conforming to the Functional Mock-up Interface standard (FMI) [20] for co-simulation. In FMI, a system is represented as a collection of executable slave components (FMUs) that communicate over connectors. FMUs may be a continuous, or discrete, simulation of the overall system, and is driven by a simulation master.

In the early stages of an Event-B development, co-simulation can be performed using discrete Event-B models, and continuous (FMU) models of the environment. At some point it may be desirable to replace the discrete Event-B models, with discrete (FMU) implementations. The implementations can then be used in the simulation as a more accurate representation of the deployed system, and they can be used to test the implementation with a continuous model of the environment. To target the FMI co-simulation framework, we generate code for an FMU from the Event-B model. An FMU is a compressed file, the contents of which is defined in the FMI standard. The FMU should contain an XML description of the model being simulated, and include the shared libraries required to run the simulation. The shared libraries are compiled from the C code that we generate from Event-B.

We have some experience of similar work with code generation, see [10,8,9]. The developments reported in this paper make use of this work, specifically *Tasking Event-B*. In Sect. 2 we provide an overview of Co-simulation and FMI. In Sect. 3 we give a brief overview of Event-B. In Sect. 4 we continue, with an overview of Tasking Event-B. In Sect. 5 we describe the role of Event-B, and our FMU translator, in producing code for use in Co-simulation. In Sect. 6 we show an example of code generation for an FMU. In Sect. 7 we present our conclusions.

2 Co-simulation and FMI

The FMI standard [4,5] is a tool-independent standard, that has been developed to facilitate the exchange, and re-use, of modelling components in the automotive industry. We see it as being of interest to a wider community (than just automotive) and are pursuing the goal of applying it in a more general sense, to cyber-physical systems in the ADVANCE project. In ADVANCE we are interested in co-simulation and verification. Our aim is to verify and simulate discrete systems, existing in a continuous environment. Event-B with ProB [11] can be used to model, verify and animate discrete systems; and FMI is an interface standard which can be used to facilitate co-simulation. The ProB animator is being changed to be used with the interface, and will allow Event-B animation, with external simulators including simulations of continuous systems, and also of discrete implementations generated from Event-B models. The continuous systems models may be developed and exported from any continuous modelling tool that supports FMI, of which there are a number, as detailed in [20]. A simulation master can then make API calls to co-ordinate any number of simulator slaves. In fact, the FMI standard only describes the slave simulators, called Functional Mock-up Units (FMUs). The master simulator is not defined in the FMI standard. But its job is to coordinate the simulation. It does this by obtaining output values from the slave components, after a specified time. It then passes these values to inputs, as described in a model description file. In the work described here we focus on simulation of two communicating slaves. More general simulations with multiple components is the subject of on-going work [14].

The standard defines two aspects, model-exchange and co-simulation; it makes use of an XML model description file to record this, and other information relevant to its use. For co-simulation we are required to define a model identifier, FMU version, generation tool, a time-stamp and model name etc. Specific information about communication across the interface also needs to be supplied. For instance a description of *modelVariables* should be specified, if visible externally (wrt. the FMU). The details include whether the variable is to be treated as an input or output, whether it is a constant, its type and so on. Other than the model description file, the FMI standard defines a C-based Application Programming Interface (API) which developers must adhere to. A simulator will consist of a master, and a number of slave simulators (these are the FMUs). The API is provided for life-cycle management of the FMUs, and FMU developers provide implementation details of instantiation, initialisation, simulation steps, and termination.

The FMI standard provides a powerful interface, but we will not need to use all of its features for our simulation (but we should provide API stubs to satisfy FMI conformance checking). We abstract away some detail from the simulation cycle, described in the FMI standard, and provide a cycle that is sufficient for our current needs. Firstly, the FMU is created, and initialised. Then, the following cycle is repeated until the end of the simulation (a user defined period). We identify two steps; a communication step, and a simulation step:

- The master retrieves values from any FMUs with output variables, and passes the values to any FMUs that are awaiting these as input.
- The master calls the *fmiDoStep* function of each slave. This will perform the simulation step.
- If the time is not expired, repeat.

In order to comply with the FMI standard, and perform this simulation cycle, we generate functions to instantiate (possibly multiples of) slaves, and initialize them (*fmiInstantiateSlave* and *fmiInitializeSlave* resp.). The retrieval from, and updates to, FMUs is performed by generating getter and setter functions; and the simulation behaviour of the FMU is described in a function called *fmiDoStep*. This information must be obtained from our Event-B model, which we introduce in the next section.

3 Event-B

The Event-B method [1] was developed by J.R. Abrial, and uses set-theory, predicate logic and mathematical constructs to model discrete systems. Event-B *machines* are used to describe dynamic properties of a system, and *contexts* describe fixed properties. Properties (such as safety-properties) are described in a machine's *invariants* and a context's *axioms*.

An example of an Event-B machine can be seen in Fig. 1, which shows an abstract model of the pump controller from our case study. We will use this model to describe some features of Event-B. But first we introduce the case study, which models a discrete *pumpController* for use in co-simulation. In this work we model the *continuous* environment discretely too. The model describes a system where the controller receives a value of the fluid level, and whether a user-request to turn the pump on has been detected. Based on the known conditions, 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. 1, we see that the machine *refines* another machine, we will return to this aspect soon. But next we see that *sees* clause has a context (and can have a number). The context may contain sets, constants, axioms and theorems. In fact, we have already introduced the constant integer *MIN*, defined in an axiom $MIN \in \mathbb{Z}$. Machines describe the dynamic aspects of a system, using state variables and guarded atomic *events*. Events can have parameters, which can model local variables, or incoming/outgoing parameter values. Variables are introduced in the *variables* clause, and typed in the *invariant* clause. The invariant also 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.

```
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 fmiGetBoolean_c
 ANY p
 WHERE p = c_pumpOnCmd ∧ p ∈ BOOL
 THEN m_pumpOnCmd := p
 END
...
```

Fig. 1. An Event-B Pump Controller Model

In the next section we look at the *Events* clause. Events describe state transitions, using action expressions; and the conditions under which they may occur, using guard predicates. An *initialisation* event is a special unguarded event, that takes place before all other events. It describes the initial state of the machine, and occurs before all other events. After initialisation, any event with all its guards evaluating *true* may occur. Actions (in the *then* clause) contain assignment expressions. They can be deterministic or non-deterministic, $:=$ or $:\in$ resp; or the clause may even be empty, and do nothing (*skip*). In the example, the *fmiGetBoolean_c* event refines the abstract *fmiGetBoolean_c* event. It declares a parameter *p*, which is typed in the guard (*where* clause). The other guard relates a parameter to a state variable $p = c\_pumpOnCmd$, which we use to model information passing between components in the system. Here the master recieves the contoller's command to turn the pump on. In the case study, variables prefixed *m_* identifies those of the master, *c_* the controller's, and *_e* the environment. Together, the guard and action refine $m\_pumpOnCmd = c\_pumpOnCmd$. In a later step we will decompose *m_pumpOnCmd* and *c_pumpOnCmd* into separate machines. In the next subsections we briefly introduce refinement and decomposition.

3.1 Refinement

Refinement is the process of adding detail to a development, as we move towards implementation. A refinement machine can introduce new variables, invariants, and events. New, and existing events, can modify new variables; but, there are restrictions on how existing variables are modified. Consistency in the relationship between an abstract machine and its refinements is maintained by discharging the automatically generated

proof obligations. Proof obligations are generated by the tool automatically. They represent the conditions that should be satisfied to demonstrate that the model is consistent with the specified properties. Discharging proof obligations demonstrates that the related properties hold. In many cases proof obligations are discharged by Rodin's automatic proof tools, but it is often necessary to perform interactive proof within Rodin. Interactive proof is undertaken by suggesting strategies, and sub-goals in the form of hypotheses.

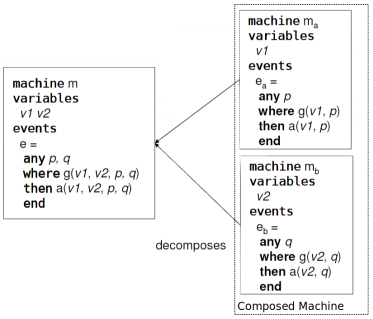

Fig. 2. Shared Event Decomposition

3.2 Decomposition

Shared Event-B decomposition [7,16] is a technique that we use to handle complexity; we are able to split a single machine specification into several. We begin with partitioning variables into machines. After decomposition the events that refer to them are shared between machines. A record of the composition is stored in the *Composed machine* Event-B component [15] as indicated in Fig. 2. The shared events are said to synchronize: i.e. they are only enabled when the guards of all events are true. The diagram shows an event e, which is decomposed, into e_a and e_b. In [9] we describe the synchronization of two events as being equivalent to a single, merged, atomic event, and then translated to a subroutine call, and subroutine definition. Event Synchronization may use shared parameters to facilitate communication between machines. As we move towards towards implementation, the decomposed artefacts reflect entities in the implementation; which assists with code generation.

4 Tasking Event-B

Tasking Event-B [9] is an extension to the Event-B language; an implementation-level, specification language. When annotations are added to a machine, it provides additional information, which is used to assist in code generation. When translating to code, it is usually necessary to work with a subset of implementable Event-B constructs. We consider *implementable constructs* to be those that are available in (or map well to) a

programming language. We would therefore usually not consider non-deterministic assignment to be implementable, for instance, and add a restriction; that these are 'refined out' of the implementation-level model. Annotations are added to both machines and contexts. The annotations are also used to generate an Event-B model of the implementation.

Machines can be implemented as task/thread-like constructs; shared, monitor-like constructs; or provide simulations of the environment. The machine *Type* annotations are *Autotask*, *Shared* and *Environ* respectively. In embedded systems, *autotask* Machines typically model *controller* tasks (of the implementation). We impose restrictions on the communication between these machines. The aim is to simplify the mapping to implementations that prevent interference in multi-threaded deployments. We stipulate that *autotask* machines cannot have synchronizing events, such as those arising from decomposition as described in Subsect. 3.2. If communication is required between tasks in an implementation then a shared machine must be used, to model a protected object. This approach was influenced by the Ravenscar profile [6] for safe multi-tasking.

We now describe some of the Tasking Event-B constructs. The main behaviour of a system's long-running task-like (or thread-like) processes are modelled by *autotasks*. An annotation is applied using the usual user interface, to a standard Event-B machine, to indicate that it models an *autotask*. An *autotask* machine has a task body which contains flow control (algorithmic) constructs; *Sequence*, *Branch*, *Loop*, *Event*, *EventSynch*, *Event*. The syntax of the *Task body* follows,

Task Body ::= TaskBody **;** TaskBody
  || **IF** Event [**ELSEIF** Event]* **ELSE** Event **END**
  || **DO** Event **END** || Event || EventSynch || output

These elements have program-related Event-B semantics. In fact we can generate a new Event-B model from the annotations, that models the implementation-level choices. However, in the work presented here we focus on relationship between the model and the generated code. The *Sequence* (**;**) construct is used for imposing an order on events, and maps to a sequence operator in programming languages. **IF** provides a choice, with optional sub-branches, between a number of events (it can only be used with events with disjoint guards, and where completeness must be shown). It maps to branching program statements, where guards are mapped to conditions and actions map to assignments. **DO** specifies event repetition while its guard remains true. It maps to a looping statement, with the loop condition derived from the event guard. *Event* is a single event, where just its action is mapped to a program statement (assignment), and guards are not permitted. *EventSynch* describes synchronization (as previously introduced) between an event in an *autotask* machine and an event in a *shared* machine. Synchronization must be implemented as an atomic subroutine call. The *EventSync* construct facilitates subroutine parameter declarations, and substitution in calls, by pairing ordered Event-B parameter declarations. Our code generators are able to produce Java, Ada and OpenMP-C, and now we add the ability to generate C for use in FMUs.

5 Machines for Co-simulation

To enable us to create an FMU for simulation of a discrete controller, we make use of the existing code generation methodology and much of the tooling. However, we provide a new translator to generate code for FMUs. Fig. 3 shows how we can use decomposition, and then refinement, to generate the discrete controller FMU from an implementation-level refinement. Later we can use the same model to generate the deployable *implementation* code. The diagram shows the partition between Event-B and

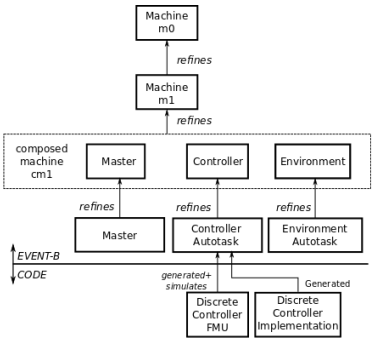

Fig. 3. Generating a discrete Controller FMU

code, using a horizontal line with arrows. Only controller code appears below the horizontal line; we do not generate code from the environment, or master models, since we can use the models themselves to test the generated code through FMI simulation. The environment model simulates the (possibly continuous) environment in a discrete model. To facilitate this new translation we remove one of the previous restrictions that we had in Tasking Event-B. In previous work *Autotask* machines could not synchronize with each other, this restriction was in place for compliance with the Ravenscar profile [6] for safe multi-threading. Now, due to the scheduling algorithm introduced by the master simulator, this issue is mitigated. Re-introduction of the problem in the deployable code is an issue for future work; we remain focussed on the generation of code for FMUs. In the remainder of the section we introduce the notion of FMUs that are modelled as a kind of *autotask* using a master's non-deterministic scheduler. We omit details of the master, where possible, and view it as a black-box that synchronizes with the FMU model, shown in Fig. 4.

On the left-hand side of the diagram we see the existing approach, where an *autotask* machine synchronizes with a *shared* machine, and the relationship between the two is consolidated in a *composed* machine. The order of the synchronizations between the *autotask* and *shared* machine are specified in the *TaskBody*. On the RHS of the diagram we see the new 'interpretation', where a black-box *master* simulator (with no *TaskBody*) synchronizes with an *autotask* machine. This *autotask* machine is a model of an FMU. As far as the FMU model is concerned, we do not worry about the ordering of the

synchronizations with the master. The master is an abstraction of some scheduling algorithm, we simply state that synchronizations are selected non-deterministically from all of the enabled synchronizations. This is the same interpretation of event 'firing', as in Event-B without synchronized events. The important aspect is to have a model, which is sufficiently well-defined, to generate code for an FMU implementation satisfying the API.

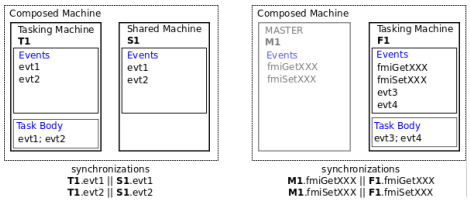

Fig. 4. Generating FMUs using a Master

In a typical Event-B development, communication between master and slave will be modelled using synchronization; and this model is the natural result of the decomposition process. It provides an easy route to formulate our translation approach. In future work we plan to omit this *master* model and use, instead, a diagram representing the connected components. This will provide the necessary information that, at the moment, is embodied in the master model.

In the diagram we see that the *master* has Events that synchronize with those of FMU model. The synchronizations model the API's getter and setter functions. Getters and setters perform the communication between master and FMU slaves. A getter and setter event must be added for each of the FMI types that are used in the model. The possible types, represented by XXX in the event name, are integer; Boolean; real; and string. For example, if an FMU exports integers, we have an *fmiGetInteger* event that models retrieval (by the master) of *all* integers from an FMU. If it imports integer values, we have an *fmiSetInteger* event that models the assignment of any incoming values (from the master).

The simulation *master* is free to implement the simulation cycle in any way, since it is not specified in the FMI standard. However, in its simplest form a master will perform a cycle of *reads*, *writes*, then request that the simulation *fmiDoStep* is performed. This is similar to the approach presented in [10]. For the purposes of translation to an FMU, the $fmiDoStep$ implementation can be modelled by one complete execution of an *Autotask*'s *Task Body*. In Fig. 4 this is where we see the sequence of events *evt3, evt4*; but it may contain any of the *TaskBody* constructs. Since the scheduling of the *autotask* is performed by the *master*, the FMU model is more like a *shared* machine construct, in that it is a 'passive'. It is not associated with its own thread (task/process etc.) during simulation, it acts merely as a slave to some thread via its subroutines.

6 Translation to Code

Event-B, and Tasking Event-B with its new interpretation, provides most of the information necessary for code generation. In addition to this, the translators make use of the theory plug-in [12]. The theory plug-in allows us to specify a mapping between Event-B mathematical operators, and their implementation counterparts. Fig. 5 shows the *controller* machine, after decomposition and refinement to the implementation level. The

```
MACHINE controllerImpl REFINES controller SEES ctx
EVENTS
  EVENT INITIALISATION ...
  EVENT fmiGetBoolean_c REFINES fmiGetBoolean_c ...
  EVENT fmiSetInteger_c REFINES fmiSetInteger_c ...
  EVENT fmiSetBoolean_c REFINES fmiSetBoolean_c ...
  EVENT filterOK REFINES filterOK ...
  EVENT filterWarn REFINES filterWarn ...
  EVENT doNothing REFINES doNothing ...
TASK BODY
  IF filterOK ELSEIF filterWarn ELSE doNothing END
```

Fig. 5. An Implementation-level Model of the Pump Controller

controller machine has three events that will be used to map to the FMI API functions, namely *fmiGetBoolean_c*, *fmiSetInteger_c*, and *fmiSetBoolean_c*. These synchronize with the master machine, and model functions (such as *fmiGetBoolean*) in the generated C code. The three events *filterOK*, *filterWarn* and *doNothing*, shown in Fig. 9, are referred to in the *TASK BODY*, and will be translated to the *fmiDoStep* API function.

6.1 Getters and Setters

We will look next at the *fmiSetInteger_c* events, as an example of communication between master and slave components. But, before we can understand the relationship between the model, and the code that is generated, we need to understand another aspect of the FMI standard; the *modelDescription* XML file. The example from the case study is shown in Fig. 6. The model description file is one of the artefacts generated from the Event-B model. It stores information about the FMU, and how it is intended to be used. Its XML attributes contain information, such as the fmi version, model name, a unique identifier for the model, and also a model description element. The model description includes elements to describe whether the model is for co-simulation, model exchange, or both. Most importantly for this work, it contains a description of the model variables. These are the variables that are visible to, or able to be manipulated by, the master via the API.

The figure shows three *ScalarVariables* from the model. Each *ScalarVariable* has a *ValueReference*, and is of a particular type (either integer, Boolean, real or a string). The *ValueReference* is used to identify the variables location in a zero-indexed array;

```
<fmiModelDescription fmiVersion="2.0" generationDateAndTime="2013-12-06..."
  generationTool="EB2FMU" guid="GUID_controllerImpl_..."
  modelName="controllerImpl" numberOfEventIndicators="0">
   <CoSimulation modelIdentifier="controllerImpl"/>
  <ModelVariables>
    <ScalarVariable name="c_level" valueReference="0"><Integer/></ScalarVariable>
    <ScalarVariable name="c_pumpOnReq" valueReference="0"><Boolean/></ScalarVariable>
    <ScalarVariable name="c_pumpOnCmd" valueReference="1"><Boolean/>
    ...
  </ModelVariables>
</fmiModelDescription>
```

Fig. 6. The *modelDescription* File

there is one array for each of the *ScalarVariable* types. In our example the first Boolean has *valueReference* zero, the next Boolean one, and so on. The value reference for each *ScalarVariable* type starts at zero; so we can see that the integer *ScalarVariable* also has an index of zero. The arrays, and the index values, are then used to resolve parameter values in calls to the API functions. To satisfy the FMI API, all variable values are passed to functions in arrays.

The use of value references is best explained with an example. In our generated code we declare a variable for each of the value references; *fmiValueReference c_level_ = 0*, and *fmiValueReference c_pumpOnReq_ = 0* and so on. We use the convention that the variable reference is related to the variable name with an underscore suffix. Here we see the use of the *fmiValueReference* type declared in the FMI standard, for which a header file is provided by the FMI organisation. Other types we will use are *fmiBoolean* and *fmiInteger*. We provide arrays to store values of each type such as an array of fmiIntegers *fmiInteger i[numberOfIntegers]*; and an array of Boolean, declared as *fmiBoolean b[numberOfBooleans]*. The array size, *numberOfIntegers* and so on, is calculated and sent by the master during the API call. The function signature for the integer setter follows,

```
fmiStatus fmiSetInteger(fmiComponent c,
                        const fmiValueReference vr[], size_t nvr,
                        const fmiInteger value[])
```

The function signature has the following parameters *fmiComponent c*, which refers to the particular 'instance' of an FMU. The parameter *fmiValueReference vr[]* refers to the array of *valueReference*s, of variables to be updated. The parameter *nvr* is the number of values in the array (number of variables to be updated). The *value* parameter is an array of updated values to which the variables should be set.

With this information we can take a look at the integer setter events *fmiSetInteger_c* which can be seen in Fig. 7. We will explain the relationship between the generated code of Fig. 8, and the event synchronization. In Fig. 7 The master *fmiSetInteger_c* event is shown on the left-hand side of the figure, and that of the controller is shown on the right-hand side. In Fig. 8 the master's variable declarations and the call are shown on the left hand side, and the controller's implementation of the setter function appears on the right. The master and controller events synchronize; they model, in an abstract

```
EVENT controllerImpl.fmiSetInteger_c
REFINES controller.fmiSetInteger_c
ANY IN p
WHERE p ∈ ℤ
THEN c_level := p
END
```

```
EVENT masterImpl.fmiSetInteger_c
REFINES master.fmiSetInteger_c
ANY OUT p
WHERE p ∈ ℤ ∧ p = m_level
THEN skip
END
```

Fig. 7. *fmiSetInteger* Synchronizing Events

```
fmiComponent c ...
size_t nvr = 1;
fmiValueReference vr[nvr] = {c_level_};
fmiInteger value[nvr] = {m_level};
fmiStatus fmiSetInteger(c, vr, nvr, value);
```

```
fmiStatus fmiSetInteger(fmiComponent c,
      const fmiValueReference vr[],
       size_t nvr , const fmiInteger value[]){
  int idx = 0;
  fmi_Component* mc = c;
  for(; idx < nvr; idx = idx + 1){
    mc->i [ vr[idx] ] = value[ vr[idx] ];
  }
  return fmiOK;
}
```

Fig. 8. *fmiSetInteger* Function Call and Function Definition

way, the assignment of integer values in the master, to controller variables. In this case there is just one value being passed between the two, but there could be more. We have *m_level* appearing in the guard in the *masterImpl* on the right *p = m_level*. Via the parameter *p*, its value is assigned to *c_level* in the *controllerImpl* action on the left. As part of the annotations, that are applied during the code generation process, we have added direction information to the parameters. The *controllerImpl* event has an incoming parameter, and the master an *outgoing* parameter. The name of the synchronizing events is, therefore, from the perspective of the master. In the implementation, the *valueReference*, to be assigned to, is stored in the value reference array *vr*. The value to be assigned is stored in the *value* array. Since there can be more than one 'instance' of an FMU a reference to the *fmiComponent* is passed to the function, together with the number of variables to be updated. In the setter function implementation, on the right, we can see that each value reference to be updated *vr[idx]*, is retrieved from the array, together with its corresponding value *value[idx]*, and set in the component's array of integers *i*[]. The function returns a value *fmiOK*, of type *fmiStatus*, to indicate that it has successfully completed, we do no error checking, so it it is the only possible return value.

6.2 Simulation Step: Models and Code

In the previous subsection we discussed events that model the communication between slaves and master. Next, we look at the part of the case study where we model the simulation step, and its implementation, *fmiDoStep* function. In our simulation code, a single 'execution' of the task body maps to the *fmiDoStep* function body. The task body of Fig. 5 refers to three events shown in Fig. 9; *filterOK*, *filterWarn* and *doNothing*. In it they appear in a branching construct which has the following semantics:

$$\text{IF } g_1 \rightarrow a_1 \text{ ELSEIF } g_2 \rightarrow a_2 \text{ ELSE } g_3 \rightarrow a_3$$

where g_i and a_i refer to the event's guards and actions. We map this to the program statement:

$$\text{if}(g_1)\{a_1;\} \text{ else if}(g_2)\ \{a_2;\} \text{ else}\{a_3;\}$$

It is the developers responsibility to ensure that the branch guards are disjoint and cover all cases, but it would be possible to generate a proof obligation to show this, in future work. The *filterOK* event describes the behaviour where a user's request has been

```
EVENT filterOK REFINES filterOK
WHERE c_level > MIN ∧ c_pumpOnReq = TRUE
THEN c_level_internal := c_level || c_pumpOnReq_internal := TRUE
        || c_pumpOnCmd := TRUE
END

EVENT filterWarn REFINES filterWarn
WHERE c_level ≤ MIN || c_pumpOnReq = TRUE || c_warn := TRUE
END

EVENT doNothing REFINES doNothing
WHERE ¬(c_level > MIN ∧ c_pumpOnReq = TRUE)
          ∧¬(c_level ≤ MIN || c_pumpOnReq = TRUE || c_warn := TRUE)
THEN skip
END
```

Fig. 9. Simulation Step Events

received to turn the pump on, and the level is greater than the minimum value. In the action we set an 'internal' value for the level, and record that a pump on request has been made. We also set a variable *pumpOnCmd* to TRUE, representing the command to turn the pump on, which is required to indicate, externally, that the pump should be turned on. The *filterWarn* event models the behaviour when a user's request to turn the pump on has been received, but the level is at or below its minimum. A variable *c_warn* representing is set so that some alarm can be raised externally. The *doNothing* event covers all cases where neither of the preceding event guards are true. Although it is not strictly necessary for code generation purposes, since we have all of the information that we require, we have stipulated (in Tasking Event-B) that all branches must have an *ELSE* clause for completeness.

The translation to code can be seen in Fig. 10. The guards and actions in *fmiOK* and *fmiWarn* is translated to conditions and program statements. They are renamed to suit the array-style assignment and lookup. For instance the guard of *fmiOK*: *c_level* > *MIN* translates to,

$$\text{mc -> i [c\_level\_controllerImpl\_] > MIN}$$

Once again *i* is the FMU *mc*'s array of integers, and *c_level_controllerImpl_* is the variable reference relating to its position in the array. Note that we have no guards in the *else* branch, and assume we have satisfied the conditions through showing completeness over the disjoint guards in the model.

```
fmiStatus fmiDoStep(fmiComponent c, fmiReal currentCommunicationPoint,
        fmiReal communicationStepSize, fmiBoolean noSetFMUStatePriorToCurrentPoint) {
    fmi_Component* mc = c;
    if ((mc->i[c_level_controllerImpl_] > MIN)
          && (mc->b[c_pumpOnReq_controllerImpl_] == fmiTrue)){
        mc->i[c_level_internal_controllerImpl_] = mc->i[c_level_controllerImpl_];
        mc->b[c_pumpOnReq_internal_controllerImpl_] = fmiTrue;
        mc->b[c_pumpOnCmd_controllerImpl_] = fmiTrue;
    } else if ((mc->i[c_level_controllerImpl_] <= MIN)
          && (mc->b[c_pumpOnReq_controllerImpl_] == fmiTrue)){
        mc->b[c_warn_controllerImpl_] = fmiTrue;
    } else { /*doNothing*/}
    return fmiOK;
}
```

Fig. 10. Simulation Step (fmiDoStep) Code

6.3 Creating the FMU

In this final section we give brief details of creation of an FMU. Until now we have just focussed on the generation of C code from the Event-B models. Generating the code from a correctly structured model should be a one-click effort. We do, however, need to package the generated code in a particular way, for use in an FMU. At present this a largely manual task, but it could be automated if necessary. The most minimal FMU, described by the FMI Standard, is a zip file containing the model description, and pre-compiled libraries. The FMI standard defines a directory, named *binaries*, where binaries for each particular platform are stored, in a folder of their own. It is also allowable for the source code to be stored in the zip file. It is the responsibility of the master to unzip the FMU, to obtain the information that it requires to execute the methods of the API. In some cases this may include compiling the sources provided.

7 Conclusions

In this paper we described our approach to generating code for use in Functional Mock-up Units (FMUs) from Event-B Tasking machines. We specifically target the discrete controllers in the development. We aim to use FMUs for co-simulation, and testing, of discrete implementations with a continuous model of the environment. We showed how synchronizing events model communication between the master and slaves; slaves can communicate with each other indirectly, via the master. We also showed how the non-synchronizing events of an *autotask* machine model the simulation step defined in the FMI standard. We then gave details relating the formal model to the code, and the model description file, required by the FMI standard.

Information about other work on formal reasoning and co-simulation of Cyber-Physical systems is scant. Ptolemy uses JFMI [21] to import FMUs, but relies on model-checking rather than theorem-proving for verification. DESTECS is a project that uses a co-simulation with VDM [19] but is not based on FMI. A more general appraisal of formal verification of Hybrid Systems is given in [3]; but we know of no other work

that links *theorem-proving* style formal verification to co-simulation, using the FMI Standard;

References

1. J. R. Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, 2010.
2. 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`.
3. Rajeev Alur. Formal verification of hybrid systems. In *Embedded Software (EMSOFT), 2011 Proceedings of the International Conference on*, pages 273–278. IEEE, 2011.
4. T. Blochwitz, M. Otter, J. Akesson, M. Arnold, C. Clauß, H. Elmqvist, M. Friedrich, A. Junghanns, J. Mauss, D. Neumerkel, et al. Functional Mockup Interface 2.0: The Standard for Tool Independent Exchange of Simulation Models. In *9th International Modelica Conference, Munich*, 2012.
5. T. Blochwitz, M. Otter, M. Arnold, C. Bausch, C. Clauß, H. Elmqvist, A. Junghanns, J. Mauss, M. Monteiro, T. Neidhold, et al. The Functional Mockup Interface for Tool Independent Exchange of Simulation Models. In *Modelica'2011 Conference, March*, pages 20–22, 2011.
6. A. Burns. The Ravenscar Profile. *Ada Lett.*, XIX:49–52, December 1999.
7. M. Butler. Decomposition Structures for Event-B. In *Integrated Formal Methods iFM2009, Springer, LNCS 5423*, volume LNCS. Springer, February 2009.
8. A. Edmunds and M. Butler. Linking Event-B and Concurrent Object-Oriented Programs. In *Refine 2008 - International Refinement Workshop*, May 2008.
9. A. Edmunds and M. Butler. Tasking Event-B: An Extension to Event-B for Generating Concurrent Code. In *PLACES 2011*, February 2011.
10. A. Edmunds, J. Colley, and M. Butler. Building on the DEPLOY Legacy: Code Generation and Simulation. In *DS-Event-B-2012: Workshop on the experience of and advances in developing dependable systems in Event-B*, 2012.
11. M. Leuschel and M. Butler. ProB: an automated analysis toolset for the B method. *STTT*, 10(2):185–203, 2008.
12. I. Maamria, M. Butler, A. Edmunds, and A. Rezazadeh. On an Extensible Rule-based Prover for Event-B. In *ABZ2010*, February 2010.
13. RODIN Project. at http://rodin.cs.ncl.ac.uk.
14. V. Savicks, M. Butler, J. Bendisposto, and J. Colley. Co-simulation of Event-B and Continuous Models in Rodin. In *4th Rodin User and Developer Workshop*, June 2013.
15. R. Silva and M. 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.
16. R. Silva, C. Pascal, T.S. Hoang, and M. Butler. Decomposition Tool for Event-B. *Software: Practice and Experience*, 2010.
17. The Advance Project Team. The Advance Project. Available at http://www.advance-ict.eu.
18. The DEPLOY Project Team. Project Website. at http://www.deploy-project.eu/.
19. The DESTECS Project. Design Support and Tooling for Embedded Control Software. Available at http://www.destecs.org/.
20. The Modelica Association Project. The Functional Mock-up Interface. Available at https://www.fmi-standard.org/.
21. The Ptolemy Project. JFMI - A Java Wrapper for FMI. Available at http://ptolemy.eecs.berkeley.edu/java/jfmi/index.htm.