Configuring Code Generated from Event-B

A. Edmunds

University of Southampton, UK

Abstract. The Event-B method, and its tools, provide a way to formally model systems; Tasking Event-B is an extension facilitating code generation. We have recently begun to explore the issue of conferability, of the generated code. In this work describe how templates can be used, which avoids the need to hard-code much of the code, that which is not directly associated with the formal model. We developed a simple approach, where tags (i.e. tagged mark-up) can be inserted in the template. Template processing involves copying code, verbatim, to a target unless a tag is encountered. Tags can be used to inject source code, or produce meta-data output, using custom code generators. This is an extensible approach, making use of the Eclipse extension-point mechanism. It can be applied to any text input file, and was used in the FMI C code generator, which we use as a running example in the paper.

1 Introduction

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 was gained in the DEPLOY project [18]. Tasking Event-B [8,9,10,11] is an extension to Event-B that facilitates generation of source code; code generators are currently available for Java [12], Ada [3], C for OpenMP [23], and C for the Functional Mock-up Interface [22] standard. The work reported here has been funded by the ADVANCE project [17], a continuation of the Event-B research effort, this time focussing on co-simulation of Cyber-Physical Systems. But, the initial discussion and some of the ideas for the work reported here, arose from an industrial collaboration with Thales Transportation Systems, Germany; during which time we undertook a case study, involving a top-to-bottom appraisal of the Event-B approach - from abstract specification to implementation. One of the main issues of concern was that of deploy-time target configuration, and how to tailor generated code. This paper reports on one approach that allows configuration, before code-generation, for a particular target environment. It is envisaged that this is just one part of such a methodology, that could be applicable in an industrial strength tool.

Often, when a software system is being implemented, there is a large amount of code that is common to the particular target implementation. Examples of this might be code for system life-cycle management, or system health monitoring. But, this code is independent of the actual state and behaviour of the part of the system being formally modelled. To achieve this we have provided

a simple Eclipse extension, to allow the use of templates. The approach has an injection mechanism, to allow insertion of 'boilerplate' code (copied verbatim), and also insertion of model-specific, generated code. At code generation time, the templates are placed in a *templates* directory, in the project being translated. The template-processor, that we developed, looks for the templates in this directory. The use of templates should facilitate re-use of existing code, and importantly, avoid having to hard-code such details. It can also provide a focal point for applying configuration criteria; something that can be explored in future work. It is envisaged that the template writer will be working on models nearing the end of the development process, close to the implementation-level. So they will, at least, require knowledge of the link between model and code. The template writer will need to apply annotations to the boiler-plate code by adding annotations in the form of tags. Of course, these details will need to be well documented to be useful. The tags can define template expansion points; and code insertion points, and meta-data generators, that are linked to pre-configured code fragment-generators. We discuss these features in Sect. 5. It should also be noted that the approach is suitable for use with any textual source and target file. In our work with code-generation, for the Functional Interface Standard (FMI) [4,5,22], we developed, and made use of the template-based generator; but it was our aim to make the template mark-up, and code fragment-generators, both customizable, extensible and suitable for use with other text based input.

2 An Overview of FMI

In order to provide a context for the use of templates, we base the discussion around the work undertaken on the Event-B-to-FMI translator. It is therefore necessary to provide some background on FMI. The Functional Mock-Up Interface standard [4,5,22] is a tool-independent standard, developed to facilitate the exchange, and re-use, of modelling components in the automotive industry. It is a C-based standard for interfaces, with re-usable components, known as Functional Mock-up Units (FMUs). These implement the FMI API, which facilitates modular composition; FMU slave simulators can be imported into a master simulation coordinator. 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]. As part of the ADVANCE project [17], we aim to provide tools and methods facilitate co-simulation of discrete Event-B models with continuous models of the environment in which they operate.

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. The contribution discussed in this paper arises from this code generation work. To conform to the FMI standard, FMU implementers must implement API functions for simulation life-cycle management, such as instantiating a slave, initialising a slave's variables, and terminating the slave. Many of the functions defined in the API are not dependent on the particular model being simulated, the code is the same for all models. We therefore wish to avoid hard-coding the translation, and it is here that we find that find the need for templates in our code generation approach.

3 An Overview of 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 ultimately use this model to generate code for use in an FMU; but we present the details here, only to illustrate our explanation of Event-B. We will not require in-depth understanding of the model itself, but we will refer to entities such as machines, variables, and a common language model (CLM) that we will explain in due course. But first we provide some details of the discrete *pumpController* model, that would like to use in a co-simulation. 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.

```
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

The state $pumpOnCmd = TRUE$ is set, if the level is OK, and a pump-on request has been received.

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 :∈ 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 receives the controllers command to turn the pump on. In the example, 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.

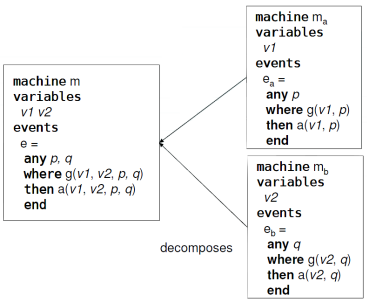

Fig. 2. Shared Event Decomposition

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 [10] 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 [10] 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. Now, when it comes to generating code for an FMU, we do not have the same restriction, since the FMI master coordinates the interactions in such a way that interference is not possible. So in this case, events of *autotask*'s may synchronize.

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 C for use in FMUs.

In order to understand the code generation process, we refer the reader to the diagram in Fig. 3. This show how an abstract model may be refined, and

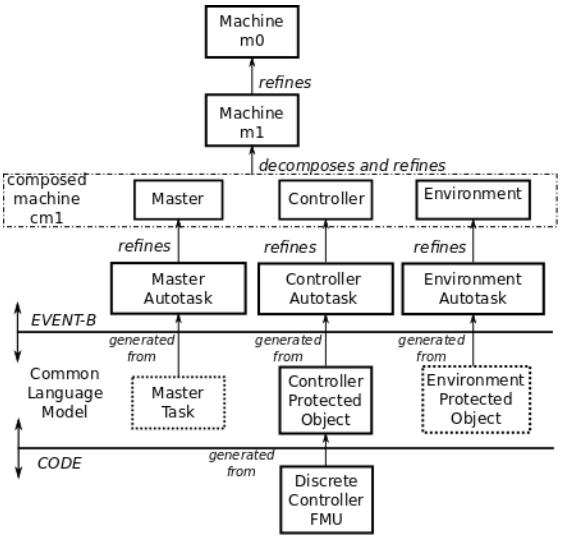

Fig. 3. The Code Generation Process

decomposed, and refined to the implementation-level (i.e. above the horizontal line annotated with *Event-B*). The code generation phase is a two-step process. The first step is to translate the Event-B machine to a language-neutral model, that we call the Common Language Model (CLM). The CLM contains all of the information necessary to implement the model, such as tasks, protected objects, variables, subroutines and program-style statements. These entities represent the 'structure' of the implementation; but the program-style, statements and conditions, retain the Event-B mathematical representation of expressions and predicates. These will be translated in the second phase of translation. It is during this second step of the code generation, when the source code is being

generated, that the template-processor is used. Notice that, in the diagram, the code generator produces a protected object, in the CLM, from the FMU Controller Autotask. It is also the case that no code is required from the master, or environment models; code generation for FMU-C is a slightly different approach from our previous work.

5 Templates: for Configuration and Re-use

We have already mentioned that the ADVANCE project [17] is providing a way to co-simulate discrete Event-B models with continuous models of the environment; using the Functional Mock-Up Interface [22]. In an extension to this work we developed a code generation approach; where discrete Event-B controller models can be translated into C code, for use in FMU simulation of discrete implementations. The FMUs can then be used for simulation and testing, with models of its continuous environment. We were able to re-use much of the generator code from the existing OpenMP generator, and just provide a new FMU-C specific generator.

When generating code from Tasking Event-B, we found that there was a large amount of boiler-plate code, from the FMI API, that we did not want to hard-code. We thought that this provided a good opportunity to explore the use of templates in code generation. The templates provide an opportunity to re-use this 'boiler-plate' code. It is also easily customised, so this is where we can make use of it for system configuration. In the first instance we surveyed the existing technology, such as Java Emitter Templates (JET) [20]. We found that this provided a very expressive solution, but was unnecessarily complex for our simple needs. We provide an architectural overview in the diagram of Fig. 4.

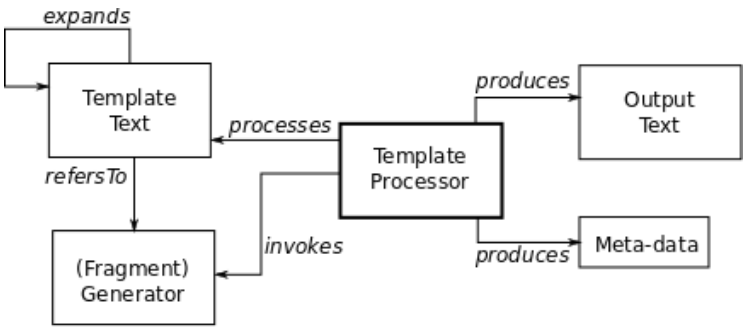

Fig. 4. Overview of Template Use

5.1 A Template Example

In Fig. 4 we can see that we have a number of artefacts involved in template processing; we have text-based templates, code-fragment generators, text output,

meta-data output and a template-processor that does the work. The templates may contain plain-text (which is copied verbatim to the target during processing) and tags. The tags may refer to other templates, or code-fragment generators. The code-fragment generators are hard-coded generators that relate to certain aspects of the final output. By way of example, the code FMI code generator has a fragment generator that inserts the variable initialisations into the template we can see its use in 5. The template is part of an implementation of the FMI

```
//## <addToHeader>
fmiStatus fmiInitializeSlave(fmiComponent c,
  fmiReal relativeTolerance, fmiReal tStart,
  fmiBoolean stopTimeDefined, fmiReal tStop){
    fmi_Component* mc = c;
    //## <initialisationsList>
    //## <stateMachineProgramCounterIni>
    return fmiOK;
}
```

Fig. 5. An Example Template

API's *fmiInitializeSlave* function. This function is part of the 'boiler-plate' C code for an implementation of an FMI slave, and the code in the example is common to all initialization functions. The first parameter of the function is the *fmiComponent*, it is the 'instance' of the FMU that is to be initialised; the other parameters relate to the simulation life-cycle. However, an FMU also keeps track of state-variables which will be different for each model. These state-variables correspond exactly to the variables of the system that have been modelled in Event-B. As part of the code generation approach we generate an intermediate model, the Common Language Model (CLM) which is independent of the target implementation language. The fragment-generator can use the CLM, in the translation to the C source code. In the template, we a place-holder (which we call a *tag*), where we want the variable initialisation to occur. The tags in our example begin with the character string, //##. This string is customizable, since we have provided an extension point, for users to define which characters to use as tags. This makes use of the Eclipse extension mechanism. The string we chose allows the tag to be treated as a comment; so, that the remainder of the code can be syntax checked if required. This line is continued with <*identifier*> where an *identifier* is supplied. A tag is usually (but not always) an insertion point; its *identifier* can relate to another template (to be expanded in-line); or the name of a fragment-generator. The fragment-generator is a Java class that can be used to generate code; or meta-data that is stored for later use, in the code generation process (see Fig. 4). In the example we have three tags. We will show how the template processor uses the *intialisationList* tag, as a code injection point to add variable initialisations, in subsection 5.3. The first tag

addToHeader identifies a generator that creates meta-data, this used at a later stage for generation of a header file. The last tag is not used in our example, since we have no state-machine diagrams in the model. But, if we did have state-machines in the FMU, we generate code to implement its initial state, in this location.

5.2 Template Tags and Generators

It is possible to categorize the users of Rodin into several types of users. One such type are the 'ordinary' modellers, using Event-B in smaller organisations. But for large scale use one may have meta-modellers (to develop product lines for instance), and another level of user may instantiate models (of the product line). There may also be platform developers, that provide platform tools for use by meta-modellers and modellers and product-line implementers. The extension points that we provide allows the platform developer to provide template utilities for the other users. The first extension point
org.eventb.codegen.templates.tag has been provided to allow them to specify the *tagCommentCharacters*. The second, *org.eventb.codegen.templates.generator* has been provided to allow them to add new *tag identifiers*, and *GeneratorClass*es that implement *IGenerator*. The *IGenerator* interface defines a single *generate* method which takes an instance of *IGeneratorData* as a parameter. The *IGeneratorData* is a container for a *List* of objects, used by the *generate* method, when generating the code fragment. The template-processor stores the link between *tag identifier* and *GeneratorClass* in a *GeneratorMap* (repository). By adding a new *tag identifier* and providing a new generator implementation, a Rodin-user (working at the implementation-level) has this new feature available for insertion into the template, to produce some specific output.

An overview of the templates and generator,s used in the FMI translation, can be seen in Fig. 6 (much of the detail omitted for brevity). The *root* template is *fmuTemplate.c*, from this we can navigate to all of the other templates, and generators. The root template generates variable declarations and the subroutines, and expands the main boilerplate functions in *fmuOthers.c*. The *fmuInstantiate* and *fmuInitialise* templates generate the corresponding FMI API function implementations. From the diagram we can see that these rely on generators to do some of the translation.

5.3 Code Injection

We now return to our explanation of how the *initialisationList* is processed, and refer back to the example shown in Fig. 5. The template-processor scans each line, and copies the output; or inserts new text, or meta-data as required, until we reach the line with the *initialisationList*. The template-processor finds the *initialisationList* in the *GeneratorMap*. It is linked to the *InitialisationListGenerator* class, by the map. The class's *generate* method is invoked, to begin the process of text insertion. A fragment of the *InitialisationListGenerator* class can be seen in Fig. 7. The class defines variables that are used to hold the data

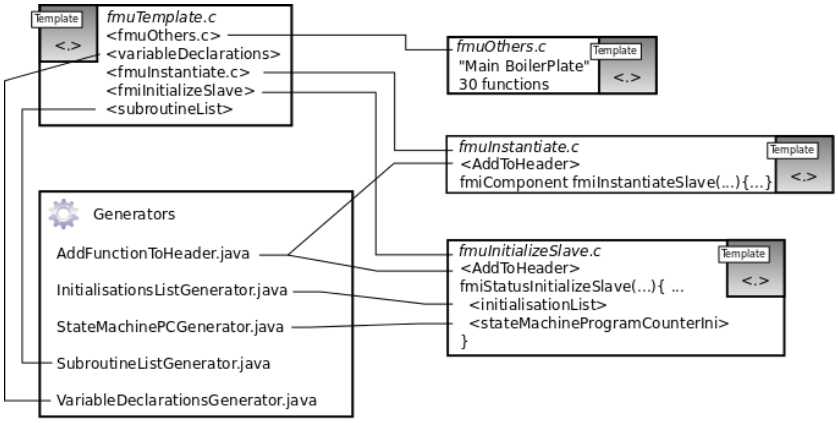

Fig. 6. The Templates and Generators in the FMI Code Generator

retrieved from the *IGeneratorData* object; *prot* and *tm* are objects that are required to process the initialisations. In this case, the declarations are retrieved from the (CLM's) *Protected* object, see Fig. 3, and translated in turn.

The main steps are highlighted using numbered comments in the code. In step 1, the data is un-packed; in step 2, the declarations are obtained from the *Protected* object; in step 3, the initialisation are translated, and add to an array of initialisation statements; in step 4, the initialisations are returned to the template-processor. The code that is generated is shown in Fig. 8. In this code we see that variable initialisations such as:

$$\text{mc->i[c\_level\_controllerImpl\_]} = 100;$$

have been generated, between the comments numbered (1) and (2). The initialisation, seen here, is very specific to the FMU approach. The variable *mc*, is of type FMI component, which is an instance of an FMU. Its integer variables are stored in an array $i[]$. A variable *c_level_controllerImpl* from the model, is initialised using an index into the array *c_level_controllerImpl_*. This is rather complex but should demonstrate the point that we can inject code of some complexity.

6 Conclusions

Using the approach that we have described in this paper, we are able to perform some deploy-time target configuration, tailoring the generated code to a specific target. To do this we provide an extensible mechanism, to process the templates. The template approach also has the advantage that we can re-use boiler-plate code, and do this without having to hard-code large sections of it. The template-processor that we provide iterates through the templates and copied the contents verbatim to a target file, unless a template tag is encountered. The default usage is for the tag to refer to another template, which is processed and expanded

```
public class InitialisationsListGenerator implements IGenerator {
 public List<String> generate(IGeneratorData data){
  List<String> outCode = new ArrayList<String>();
  Protected prot = null;
  IL1TranslationManager tm = null;
  //(1) Un-pack the GeneratorData
  List<Object> dataList = data.getDataList();
  for (Object obj : dataList) {
   if (obj instanceof Protected) {prot = (Protected) obj; }
   else if(obj instanceof IL1TranslationManager){
    tm = (IL1TranslationManager) obj;}}
 ...
 //(2) Get the Declarations
 EList<Declaration> declList = prot.getDecls();
 //(3) Process each Variable Declaration/Initialisation
 for (Declaration decl : declList) {
  ...
  String initialisation = FMUTranslator.updateFieldVariableName(...);
  outCode.add(initialisation);
 }
 // (4) return the new fragment
 return outCode;}}
```

Fig. 7. An Example Fragment-Generator

```
fmiStatus fmiInitializeSlave(fmiComponent c, fmiReal relativeTolerance,
    fmiReal tStart, fmiBoolean stopTimeDefined, fmiReal tStop) {
 fmi_Component* mc = c;
 // (1) Injected By InitialisationsListGenerator >>>
 mc->i[c_level_controllerImpl_] = 100;
 mc->b[c_pumpOnReq_controllerImpl_] = fmiFalse;
 mc->b[c_pumpOnCmd_controllerImpl_] = fmiFalse;
 ...
 // (2) <<< End of Injection
 return fmiOK;
}
```

Fig. 8. The fmiInitializeSlave Function with Injected Code

in-line. A platform developer is able to enrich the template language, by adding new template tags and associating them with custom fragment-generators. In this way complex code generation activities can be performed, to generate text output; or meta-data in other formats. The meta-data is useful for downstream code generation. This approach also provides some scope for applying configuration options, in other ways, since we could modify the template on-the-fly, during code generation. This is however left for future investigation.

During preliminary investigations we considered the following template processing approaches; Acceleo [19] , Xpand [21], and JET [20]. Each of the approaches are more complex than we really need. We have also found, in previous approaches, that too much reliance on external tools can lead to integration and maintenance problems. Our own code generation approach already has many dependencies on external plug-ins, we will only increase this number if absolutely necessary. It is also the case, with these template based processors, that complex code generation control structures are difficult to use; and this is why we developed the simple interface, as a priority. For those reasons we chose to implement our own lightweight, extensible template language and code-injection mechanism. We do recognize, however, that the burden of complexity has been shifted to the *IGenerator* implementers. But, we believe this should be an acceptable trade-off, when developing a product-line for instance.

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. J. Barnes. *Programming in Ada 2005*. International Computer Science Series. Addison Wesley, 2006.
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. *Providing Concurrent Implementations for Event-B Developments*. PhD thesis, University of Southampton, March 2010.
9. A. Edmunds and M. Butler. Linking Event-B and Concurrent Object-Oriented Programs. In *Refine 2008 - International Refinement Workshop*, May 2008.
10. A. Edmunds and M. Butler. Tasking Event-B: An Extension to Event-B for Generating Concurrent Code. In *PLACES 2011*, February 2011.

11. 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.
12. J. Gosling, B. Joy, G. Steele, and G. Bracha. *Java Language Specification - Third Edition*. Addison-Wesley, 2004.
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. Advanced Design and Verification Environment for Cyber-physical System Engineering. Available at http://www.advance-ict.eu.
18. The DEPLOY Project Team. Project Website. at http://www.deploy-project.eu/.
19. The Eclipse Foundation. Acceleo - transforming models into code. Available at http://www.eclipse.org/acceleo/.
20. The Eclipse Foundation. JET - Java Emitter Templates. Available at http://www.eclipse.org/modeling/m2t/?project=jet#jet.
21. The Eclipse Foundation. Xpand: A language specialized on code generation based on EMF models. Available at http://wiki.eclipse.org/Xpand.
22. The Modelica Association Project. The Functional Mock-up Interface. Available at https://www.fmi-standard.org/.
23. The OpenMP Architecture Review Board. The OpenMP API specification for parallel programming. Available at http://openmp.org/wp/.