Templates for Event-B Code Generation

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 how we can configure the code generator, for deployment on different target systems. In this paper, we describe how templates can be used to avoid hard-coding the 'boilerplate' code not directly associated with a formal model, and at the same time allowing integration of code generated from the formal model. We have developed a lightweight approach, where tags (i.e. tagged mark-up) can be inserted in source templates. The template-processors could be of use to other plug-in developers; they can be used with any text input file.

1 Introduction

Rodin is a tool platform [2] for the rigorous specification of critical systems, using the Event-B approach [1]. Tasking Event-B [4,5,6,7] is an extension to Event-B that facilitates generation of source code. We can generate Java [8], Ada [3], C for OpenMP [11], and C for the Functional Mock-up Interface [10] standard. The work reported in this paper has been undertaken during the ADVANCE project [9]; a continuation of the Event-B research effort, this time focussing on co-simulation of Cyber-Physical Systems. This paper introduces an approach that allows use of boilerplate code, in the form of templates with code injection.

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, system health monitoring, or task scheduling. But, this code is independent of the actual state and behaviour of the part of the system being formally modelled. We provide a simple Eclipse extension to facilate the use of templates with tagged mark-up; we can then insert code, generated from the formal model, into boilerplate templates. The use of templates should facilitate re-use of existing code, and importantly, avoid having to hard-code such details. The template creator will need to apply annotations to the boiler-plate code by adding annotations in the form of tags. The tags can define locations where other templates should be expanded. They also define code insertion points, and meta-data generators, linked to pre-configured code fragment-generators. This extension is suitable for use with any text-based source and target. To validate the approach we provided a C code generator for use with the Functional Interface Standard (FMI) [10]. We provide an overview of FMI and code generation with Tasking Event-B in Sect. 2. We introduce templates, and show an example of their use, in Sect. 3, and conclude in Sect. 4.

2 Background

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 the Functional Mock-Up Interface (FMI) standard [10]; 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, defining an interface for re-usable components, known as Functional Mock-up Units (FMUs). FMUs play the role of slave simulators in simulations that are coordinated by a simulation master. The master simulator is not defined in the FMI standard, but its job is to coordinate the simulation e.g. by stopping and starting slaves. It also manages the communication; where all the slaves' input and output values are communicated via the master, never directly between slaves. To target the FMI co-simulation framework, we generate code for an FMU from the Event-B model. An FMU is a compressed file containing an XML description of the model being simulated, and the shared libraries required to run the simulation. The shared libraries are compiled from the C code that we generate from Event-B. 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 wish to avoid hard-coding the translation where possible; so, templates provide a good solution.

Tasking Event-B [6] 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. Machines can be implemented as task/thread-like constructs; shared, monitor-like constructs; or provide simulations of the environment. The machine *Types* are *Autotask*, *Shared* and *Environ* respectively. In embedded systems, *autotask* Machines typically model *controller* tasks (of the implementation).

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 *autotask* machine has a task body which contains flow control (algorithmic) constructs. 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. 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.

Fig. 1 shows how an abstract model may be refined, decomposed, and then refined again to the implementation-level (i.e. above the horizontal line annotated with *Event-B*). The code generation phase (below this line) is a two-step

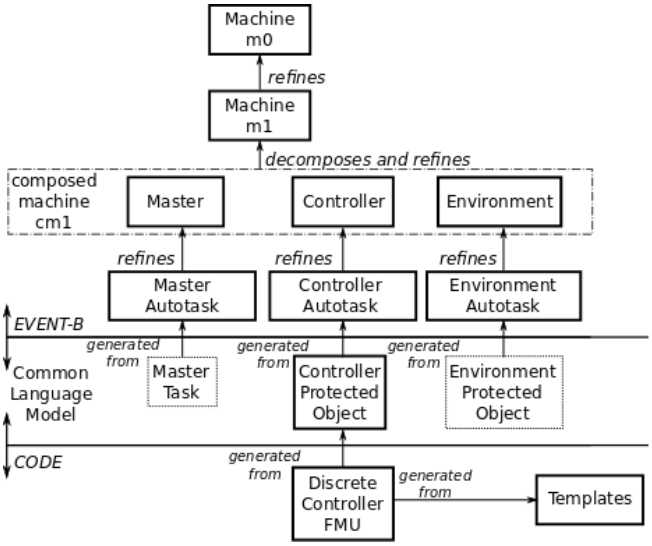

Fig. 1. The Code Generation Process

process; although only a single step is visible to the user. The first step is to translate the Event-B machine to a language-neutral model, the Common Language Model (CLM). During this second step, when the source code is being generated, the templates contribute to the generated code.

3 Using Templates

The ADVANCE project [9] is providing a way to co-simulate discrete Event-B models with continuous models of the environment; using the Functional Mock-Up Interface [10]. 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. 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. Templates provide a way to re-use this 'boiler-plate' code. An architectural overview of our template-driven approach can be seen in the diagram of Fig. 2. We see the artefacts involved in template processing; namely text-

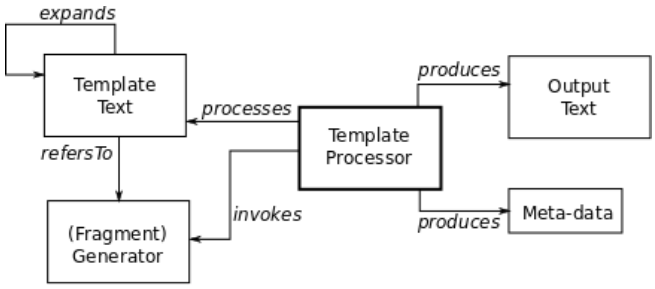

Fig. 2. The Template Processor and Artefacts

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; for instance, a fragment generator inserts the variable initialisations as specified in a template. We can see an example of this in Fig. 3. The template

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

Fig. 3. An Example Template

shows part of an implementation of the FMI API's *fmiInitializeSlave* function; the code in the template is common to all of the FMUs that we will generate for a particular target configuration. The tags accommodate variability between

models; e.g. FMUs keep track of state-variables, which may be different for each model. These state-variables correspond exactly to the variables of the system that have been modelled in Event-B. In the function shown, the first parameter is the *fmiComponent*, the 'instance' of the FMU that is to be initialised. The other parameters relate to the simulation life-cycle. In the template, we insert a place-holder (which we call a *tag*), where we want variable initialisation to occur. The tags in our example begin with the character string, //##. The line continues with an *identifier*, <*identifier*>. A tag is usually (but not always) used as an insertion point; its *identifier* can relate to another template (to be expanded and processed 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. 2). In the example we have three tags. The first tag *addToHeader* identifies a generator that creates meta-data, which are used at a later stage, for generation of a header file. 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, modellers and product-line implementers. The extension points allow the platform developer to provide template utilities for the other users. They can define new tags and fragment-generators. An overview of the templates and generators used in the FMI translation, can be seen in Fig. 4 (much of the detail is 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. The template-processor scans each line, and copies the output; or inserts

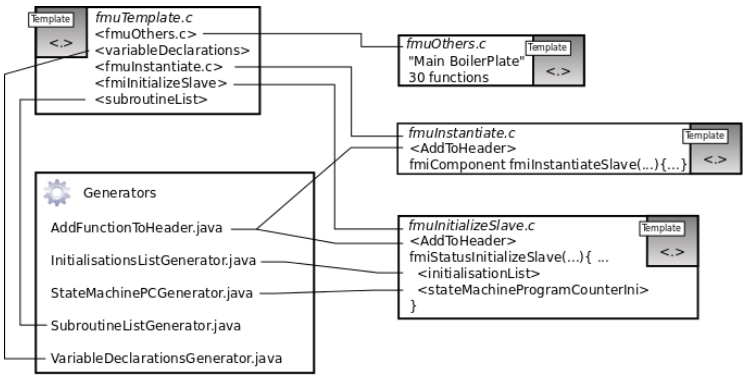

Fig. 4. The Templates and Generators in the FMI Code Generator

new text, or meta-data as required, until we reach a generator tag. 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. 5. The main steps

```
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. 5. An Example Fragment-Generator

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.

4 Conclusions

Using the approach that we have described in this paper, we are able to perform target configuration prior to code-generation; and re-use boilerplate code, without having to hard-code it. The template-processor reads each line of a template and copies the contents, verbatim, to a target file unless a template tag is encountered. A tag can refer to another template, which is processed by expanding it in-line, or a custom fragment generator. As part of an extensible approach, a platform developer can enrich the template language, by adding new template tags and associate them with custom fragment-generators. In this way complex code generation activities can be performed, to generate text output, or to generate meta-data in other formats. The meta-data is useful for downstream code

generation. We used the template-driven approach to implement part of a new code generator, translating Event-B models to FMI-C code.

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.
3. J. Barnes. *Programming in Ada 2005*. International Computer Science Series. Addison Wesley, 2006. ISBN-10: 0321340787 / ISBN-13: 9780321340788.
4. A. Edmunds. *Providing Concurrent Implementations for Event-B Developments*. PhD thesis, University of Southampton, March 2010.
5. A. Edmunds and M. Butler. Linking Event-B and Concurrent Object-Oriented Programs. In *Refine 2008 - International Refinement Workshop*, May 2008.
6. A. Edmunds and M. Butler. Tasking Event-B: An Extension to Event-B for Generating Concurrent Code. In *PLACES 2011*, February 2011.
7. 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.
8. J. Gosling, B. Joy, G. Steele, and G. Bracha. *Java Language Specification - Third Edition*. Addison-Wesley, 2004.
9. The Advance Project Team. Advanced Design and Verification Environment for Cyber-physical System Engineering. Available at http://www.advance-ict.eu.
10. The Modelica Association Project. The Functional Mock-up Interface. Available at https://www.fmi-standard.org/.
11. The OpenMP Architecture Review Board. The OpenMP API specification for parallel programming. Available at http://openmp.org/wp/.