

Configuring Code Generated from Event-B

A. Edmunds¹, K. Wiederaenders² and K. Reichl²

¹ University of Southampton, UK

² Thales Transportation Systems, Germany

Abstract. The Event-B method, and its tools, provide a way to formally model systems. Tasking Event-B is an extension facilitating code generation from Event-B. We have recently begun to explore the issue of re-usability, of generated code. In this work describe two features, interface generation and templates. To improve re-usability, we extract an interface from an Event-B model of the environment, and provide automatically generated simulation code. The provision of the interface allowed further experimentation with hand-crafted code. In a second enhancement to the approach, we introduce a method of using templates, for code generation. We avoid hard-coding code common to particular target; we add the common code to template and introduce tags for template expansion or code injection. Template expansion allows copying of code verbatim, and generator-tags allow code injection linked to custom code generators. This is an extensible approach, making use of the Eclipse extension-point mechanism. This approach can be applied to any text output, and was used in the FMI C code generator, from which we take our examples.

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 [9], and experience with industry was gained in the DEPLOY project [11]. Tasking Event-B [4,5,6,7] is an extension to Event-B that facilitates generation of source code; code generators are currently available for Java [8], Ada [3], C for OpenMP [14], and C for FMI [13]. The work reported here has been funded by the ADVANCE project [10], 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 re-use of generated artefacts. Until now the deploy-time issues of target configuration, and how to tailor the generated code for re-use has not been addressed.

It is certainly mainstream engineering practise to re-use artefacts, wherever possible. This provides benefits in terms of time and cost savings, and improved

reliability. In programming terms re-use of code often involves the use of patterns, sub-classing, and use of polymorphism. In the work that we report here, we too have aimed to improve re-use in this way. We first discussed this as future work, in [6], where we alluded to the fact that a generated environment simulation may be replaced by one that interfaces with the ‘actual’ drivers of the target system. In Sect. 3 we introduce interface generation, from Tasking Event-B, and present two short examples of how the interfaces can be used in the development process.

Often when a system is being implemented, there is a large amount of code that is common to the particular target implementation, perhaps 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. We have provided a simple extension to allow the use of templates; with an injection mechanism to allow insertion of such ‘boilerplate’ code (copied verbatim), and also insertion of model-specific, generated, code. The use of templates in the code generation approach should facilitate re-use of existing code, and importantly avoid having to hard-code such details. It will also provide a focal point for application of many configuration details, whilst at the same time permitting insertion of generated code at locations specified by the template writer. The tags that define insertion points, are linked to pre-configured code fragment-generators. We discuss this feature in Sect. 4. It should also be noted that the approach is suitable for use with any textual source and target file. In our work with FMI code-generation 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 and output.

2 Event-B

All about Event-B and Tasking Event-B

3 The Use of Generated Environment Interfaces

The case study

4 Templates: for Configuration and Re-use

The Functional Mock-Up Interface [13] is a C-based interface that has been devised to facilitate re-use of simulation models. The re-usable components, known as Functional Mock-up Units (FMUs) implement the FMI API, and allows modular composition where FMU slave simulators are imported into a custom master simulation coordinator. The ADVANCE project [10] is providing a way to co-simulate discrete Event-B models with continuous models of the environment. We have also 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) [12]. 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. 1.

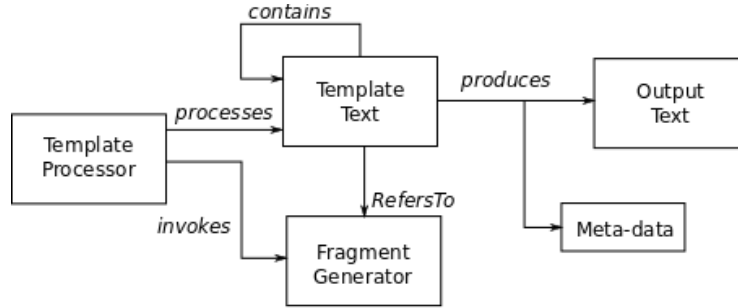


Fig. 1. Overview of Template Use

4.1 A Template Example

In this diagram we can see that we have only three artefacts; text templates, code-fragment generators, and the text output. The templates may contain text (which is copied verbatim to the target) and insertion points. The insertion points 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 2. The template is part of an implementation of the FMI API's *fmiInitializeSlave* function. This 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 template also contains some specialisations that depend on the model being translated. Each line of a template is scanned, and tags are processed when the *template processor* reads a line beginning with

```

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

```

Fig. 2. An Example Template

///##. These characters indicate to the template processor that it should treat the line as a tag, which is usually (but not always) an insertion point. The particular character string is customizable. We have provided an extension point, for users to define which characters to use as tags, using 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.

The tag, with its *identifier*, can define the name of another template (to be expanded in-line); or the name of a fragment generator. The generator can either be used to generate code, or meta-data for use later 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 4.3. 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.

4.2 How the Template Processor Works

It is possible to categorize the users of Rodin into several types of users. One such type are the meta-modellers, another user may instantiate models, a third is a system administrator/programmer that provides tool for the others. The extension points that we provide allows the system administrator/programmer 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* names, and *GeneratorClasses* that implement *IGenerator*. By adding a new tag and providing a new generator implementation, a user has this new feature available for insertion into the template, to produce some specific output.

The *IGenerator* interface contains When it comes across a tag - expands, or passes on to a processor.

Implements IGenerator, with IGeneratorData to hold data.
Generator Extensions.
Looks for name in templates file
Looks for name in generator extensions.

4.3 Code Injection

The example

5 Conclusions

Tags for Template Expansion

Tags for Code Insertion with fragment generators

Tags for processing meta-data

Re-use of boiler-plate code, avoids hard-coding too.

Provided configuration, either manually through changing the template code, or by making use of tags as insertion points, that can make use of configuration data.

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. 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. RODIN Project. at <http://rodin.cs.ncl.ac.uk>.
10. The Advance Project Team. The Advance Project. Available at <http://www.advance-ict.eu>.
11. The DEPLOY Project Team. Project Website. at <http://www.deploy-project.eu/>.
12. The Eclipse M2T Project. Java Emitter Templates. Available at <http://www.eclipse.org/modeling/m2t/?project=jet#jet>.

13. The Modelica Association Project. The Functional Mock-up Interface. Available at <https://www.fmi-standard.org/>.
14. The OpenMP Architecture Review Board. The OpenMP API specification for parallel programming. Available at <http://openmp.org/wp/>.