SEB-CG: Code Generation Tool with Algorithmic Refinement Support for Event-B

Mohammadsadegh Dalvandi[1], Michael Butler[2], and Asieh Salehi Fathabadi[2]

[1] University of Surrey
`m.dalvandi@surrey.ac.uk`
[2] University of Southampton
`{mjb, asf08r}@soton.ac.uk`

Abstract. The guarded atomic action model of Event-B allows it to be applied to a range of systems including sequential, concurrent and distributed systems. However, the lack of explicit sequential structures in Event-B makes the task of sequential code generation difficult. Scheduled Event-B (SEB) is an extension of Event-B that augments models with control structures, supporting incremental introduction of control structures in refinement steps. SEB-CG is a tool for automatic code generation from SEB to executable code in a target language. The tool provides facilities for derivation of algorithmic structure of programs through refinement. The flexible and configurable design of the tool allows it to target various programming languages. The tool benefits from xText technology for a user-friendly text editor together with the proving facilities of Rodin platform for formal analysis of the algorithmic structure.

Keywords: Automatic Code Generation · Event-B · Program Verification.

1 Introduction

Event-B [1] is a general purpose formal modelling language based on set theory and predicate logic. It has been successfully applied in a wide range of systems including sequential, concurrent and distributed systems. The language is supported by a tool called Rodin [2]. Rodin is an extensible Eclipse-based platform which facilitates modelling and verification of Event-B models. Event-B in its original form does not support code generation. There have been a number of attempts to provide Event-B and Rodin with a code generation tool [6,8,9]. However, the lack of explicit control flow in Event-B made these tools suffer from usability problems. Other issues like the lack of clear and formal relationship between the generated code and the high level formal model decreased the confidence in the code generated by those tools.

This work is a fresh attempt to provide the Event-B toolset with facilities required for formal development and generation of sequential programs. The tool described in this paper is built on our empirical experience with existing Event-B code generation tools in particular [6]. In developing SEB-CG, we have tried to address shortcomings of existing tools and also build on our previous theoretical work on derivation of algorithmic structures and verifiable code generation from Event-B models [3,4]. In designing SEB-CG, we have had the following principles in mind:

- **Extensibility:** The tool should be designed in a way that it is straightforward to extend it to accommodate new target languages.
- **Customisability:** The output of the tool should be highly customisable so that it can be used for generating programs for different domains.
- **Self-Sufficiency:** The tool should be self-sufficient for its core functionalities and its dependency on other Rodin plugins should be minimal.
- **Usability:** The tool should be designed in way that it is intuitive, useful and easy to use.

The above principles have been realised in SEB-CG in various ways. We have provided interfaces for extending the tool and adding support for new programming languages in a clear way. Also it is straightforward to add new translation rules for new target languages. The output of the tool is defined using templates and can be customised by modifying the templates. For instance, the way that a program is structured in terms of procedures and classes can be defined by the user. Unlike some of the previous works that were heavily dependent on other Rodin plugins for some of their core functionalities (e.g. translation rule definitions), SEB-CG has minimal dependency on other Rodin plugins and it has native support for its core functionalities. The scheduling language of SEB-CG is implemented using xText[3]. The xText editor provides a user-friendly environment for writing schedules. We have implemented a number of validation rules using the xText validator which provide the user with live and useful feedbacks including error and warning messages and tips on how to resolve the problem.

The tool and the instructions on how to install and use are provided in `http://dalvandi.github.io/SEB-CG`. The rest of this paper is devoted to details of the tool and its implementation.

2 Scheduled Event-B

SEB-CG implements the approach introduced in [3,4] which augments Event-B models with explicit control structures. We provided a scheduling language that allows the modeller to specify the control flow of events explicitly. In our approach, starting from the most abstract specification, the modeller provides a schedule associated with each machine. As Event-B refinement continues, the schedule associated with each refinement model should also refine the abstract schedule. We also provided a number of schedule refinement rules that direct the modeller in deriving a concrete program structure from the abstract schedule through refinement. In [4] we provided a number of translation rules for generating code and contracts (logical assertions) from a scheduled Event-B model. Our

[3] https://www.eclipse.org/Xtext/

translation rules transform a concrete scheduled Event-B model to executable code and also generate a number of assertions that allow static verification of code properties.

3 Tool Overview

The SEB-CG tool is implemented as a Rodin plugin. The tool consists of a UI and a code generation core. The UI provides the user with a text editor for writing schedules. It also extends the Rodin explorer to include Schedule elements in a Rodin project folder and also provides a handle to the schedule-specific proof obligation generated by the tool.

The code generation machinery of the tool is depicted in Figure 1. As shown, the SEB-CG tool receives four inputs: Schedule, Model, Program Template, and Translation Rules. A brief description of these inputs follows.

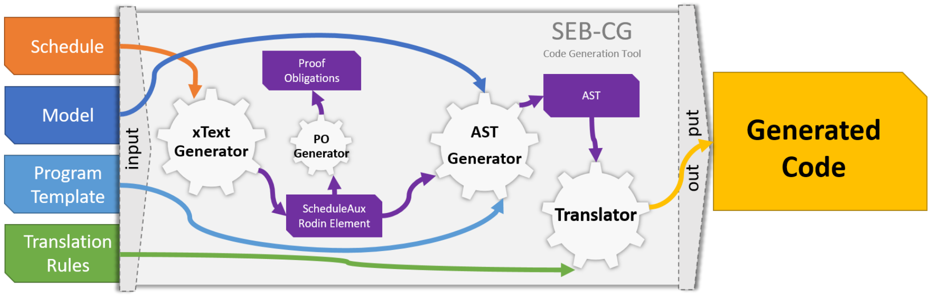


Fig. 1. A high-level overview of SEB-CG

Schedule and Model: A *Schedule* is a text file written in the Scheduled Event-B (SEB) language and has *.seb* extension. The SEB language supported by the tool is presented in Appendix A. The schedule file contains a reference to a machine whose events it schedules. This is specified using the `machine` keyword. The name of the schedule is defined using the `schedule` keyword and it should be the same as the schedule file name. The schedule may refine another schedule. This is defined using `refines` keyword. As an example see the schedule presented in Figure 2 (1). The schedule name is $s3$ and it refines the abstract schedule $s2$. It is scheduling machine $m3$. Once the SEB-CG is invoked on a schedule for a target language, then the tool takes the schedule and the specified machine as inputs.

Program Template: We mentioned earlier that extensibility and customisability are two of the principles of SEB-CG. To realise these principles we have introduced *program templates*. A program template is a convenient feature of the SEB-CG that allows the user to specify and customise the output of the tool without the need for making changes to the implementation. It is expected that each new language that the tool is extended with, is provided with a program template. The program templates are not expected to be modified by non-expert users as this may make the output of the tool invalid. Template files are XML files that describe how different elements of the program are ordered and placed in the final generated code. We have defined a simple language for templates. The grammar of our Program Template Language (PTL) is given in Appendix B.

Translation Rules: Translation rules define the way in which a scheduled Event-B model is translated to code in a target language. Instead of hard-coding the rules in the implementation of the tool, SEB-CG provides a flexible way for defining translation rules. Each target language has a translation rule file in XML format. The grammar of the syntax of the Translation Rule Language (TRL) is given in Appendix C.

4 Tool Components

Figure 1 provides a high level view of the tool core machinery. As can be seen there are four main components in the tool: xText Generator, PO Generator, AST Generator and AST Translator. The work flow of the tool is also depicted in the figure. The rest of this section provides details of various components.

xText Component: We have leveraged the power of xText [7] in the implementation of our tool. Specifically, we have used xText to define the grammar of the scheduling language. xText also provides us with other useful facilities like text editor and a basic validator out of the box. We extended the xText validator with schedule refinement rules so that concrete schedules are checked to be valid refinements of the abstract ones. We have also used the xText generator to translate the textual representation of the schedule to a newly defined Rodin element called ScheduleAux. This translation is performed in order to be able to use the Rodin proof obligation generator easier. ScheduleAux is an internal element and is hidden from the Rodin user.

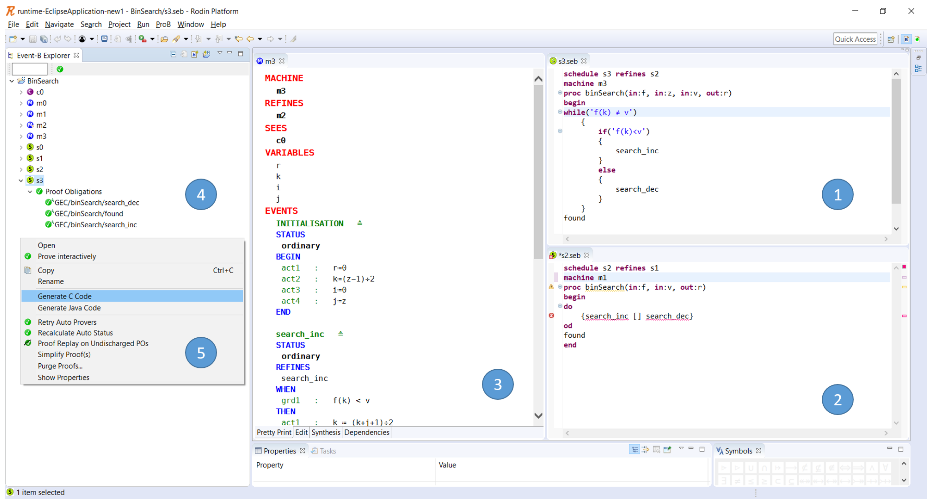
PO Generator: As explained in [3], there are a number of proof obligations (i.e. guard elimination POs) that a schedule must satisfy. We have extended the Rodin proof obligation generator to generate the required proof obligations based on the schedule (ScheduleAux) and the model (machine and context) that the schedule is referring to.

AST Generator: In order to translate a model to code in a target language, the tool first generates an abstract syntax tree (AST). The AST is generated based on the program template, schedule and the model. The AST represents the overall structure of the program and the hierarchical order of different parts of the program, e.g. classes, procedures, program body, etc. Sequentialisation of event actions [4] are also done by the AST generator as part of the event AST generation.

AST Translator: Once the AST is generated, it is translated to the code in the target language by the AST translator. The AST translator receives translation rules and the generated AST as its inputs and traverses the AST recursively and matches its sub-trees with appropriate rules and outputs the program text.

5 Tool Usage

SEB-CG is designed to be an easy-to-use tool. The GUI is intuitive and consists of a simple text editor. The text editor has syntax colouring and highlighting support and provides live feedback on syntactical warnings/errors. The schedule is also checked in the background to ensure conformance to the refinement rules of [3]. Schedules appear in the Event-B project explorer of Rodin alongside other project elements e.g. machines and contexts. A schedule can only refer to machines in the same project. Since schedules and their respective proof obligations are stored separately from the Event-B model, modifying a schedule does not change its associated model or its proofs.



```
MACHINE
   m3
REFINES
   m2
SEES
   c0
VARIABLES
   r
   k
   i
   j
EVENTS
   INITIALISATION
   STATUS
    ordinary
   BEGIN
    act1   :   r≔0
    act2   :   k≔(z−1)÷2
    act3   :   i≔0
    act4   :   j≔z
   END

   search_inc
   STATUS
    ordinary
   REFINES
    search_inc
   WHEN
    grd1   :   f(k) < v
   THEN
    act1   :   k ≔ (k+j+1)÷2
```

```
schedule s3 refines s2
machine m3
proc binSearch(in:f, in:z, in:v, out:r)
begin
while('f(k) ≠ v')
    {
        if('f(k)<v')
        {
            search_inc
        }
        else
        {
            search_dec
        }
    }
found
```

```
schedule s2 refines s1
machine m1
proc binSearch(in:f, in:v, out:r)
begin
do
    {search_inc [] search_dec}
od
found
end
```

Fig. 2. A screen-shot of the tool

Figure 2 is a screen-shot of the tool. (1) is a schedule editor showing the concrete schedule $s3$ and (2) is another editor showing the abstract $s2$ schedule. Note that $s3$ refines $s2$. We intentionally injected an error into $s2$ by referencing to a wrong machine ($m1$ instead of $m2$) so that the text highlighting is demonstrated. (4) is the Event-B explorer showing schedules $s0, .., s3$ and proof obligations related to $s3$. (5) is the menu that allows the invocation of the code generator for any of the available target languages. Finally (3) is an Event-B machine $m3$, which is scheduled by $s3$, shown using the standard Event-B machine editor.

The recommended practice for using the tool is to start introducing the schedules from the abstract level where the abstract machine is defined and then refine it alongside the machine refinement. The abstract schedule usually contains only abstract scheduling constructs (i.e. choice and iteration). As the refinement continues, the abstract constructs are replaced with concrete ones (i.e. if-branches and while-loops). Although it is possible to define the concrete schedule for the concrete model directly without going through schedule refinement steps, it is a discouraged practice since it is more likely to result in guard elimination POs that cannot be discharged.

Once the refinement has reached a concrete level, both for the model and schedule, the user can invoke the code generator by right-clicking on the concrete schedule element and select the desired target language from the list of available target languages. It is at this time that the tool starts building the AST with respect to the program template, schedule and model. The generated AST together with the translation rules are then fed into the translator and the code is generated. If during the translation phase, the translator does not find a match between a sub-tree and the provided rules, an exception will be thrown and the user will be provided with the pattern of the rule that it was unable to find.

6 Conclusion

In this paper we presented a tool for automatic code generation from scheduled Event-B models. The tool is customisable and extensible and can potentially accommodate a wide range of target languages. Currently the tool has out of the box support for C and Java code generation and it can be extended to include other languages.

As far as we are aware, the only other code generation tool for Event-B that allows introduction of explicit program order is Tasking Event-B [6]. However, comparing to SEB-CG, the Tasking Event-B tool has a restrictive scheduling language (e.g. no support for nested control structures or explicit loop/branch conditions) and has no support for schedule refinement. There exist other code generation tools for Event-B which do not allow introduction of algorithmic structure of the model by the modeller [8,9]. The generated code by these tools may not be optimised and depends entirely on the implementation of the tool and not on a verified algorithmic structure provided by the modeller.

In future, we would like to extend the tool to also generate code contracts (i.e. assertions and pre/post-conditions) as described in [5,4]. The generated contracts will allow the verification of some properties of the generated code (e.g. sequentialisation) using a static program analyser. Extension of the scheduling language to support procedure calls is another feature for the future. We are also interested in further development of our tool to support concurrent program generation.

Acknowledgments. This work was supported by the EPSRC PRiME Project (EP/K034448/1), www.prime-project.org. The first author is currently supported by EPSRC grant EP/R032556/1.

References

1. Abrial, J.R.: Modeling in Event-B: system and software engineering. Cambridge University Press (2010)
2. Abrial, J.R., Butler, M., Hallerstede, S., Hoang, T., Mehta, F., Voisin, L.: Rodin: an open toolset for modelling and reasoning in Event-B. International Journal on Software Tools for Technology Transfer **12**(6), 447–466 (2010)
3. Dalvandi, M., Butler, M., Rezazadeh, A.: Derivation of algorithmic control structures in Event-B refinement. Science of Computer Programming **148**(Supplement C), 49 – 65 (2017). https://doi.org/https://doi.org/10.1016/j.scico.2017.05.010, `http://www.sciencedirect.com/science/article/pii/S016764231730120X`, special issue on Automated Verification of Critical Systems (AVoCS 2015)
4. Dalvandi, M., Butler, M., Rezazadeh, A., Fathabadi, A.S.: Verifiable code generation from scheduled event-b models. In: International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z. pp. 234–248. Springer (2018)
5. Dalvandi, M., Butler, M.J., Rezazadeh, A.: Transforming Event-B models to Dafny contracts. ECEASST **72** (2015), `http://journal.ub.tu-berlin.de/eceasst/article/view/1021`
6. Edmunds, A., Butler, M.: Tasking Event-B: An extension to Event-B for generating concurrent code (February 2011), `https://eprints.soton.ac.uk/272006/`, event Dates: 2nd April 2011
7. Eysholdt, M., Behrens, H.: Xtext: implement your language faster than the quick and dirty way. In: Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion. pp. 307–309. ACM (2010)
8. Méry, D., Singh, N.K.: Automatic code generation from Event-B models. In: Proceedings of the second symposium on information and communication technology. pp. 179–188. ACM (2011)
9. Wright, S.: Automatic generation of C from Event-B. In: Workshop on integration of model-based formal methods and tools. p. 14. Citeseer (2009)

Appendix A Scheduling Language

```
⟨Schedule⟩     ::= schedule ⟨ScheduleName⟩, [refines ⟨ScheduleName⟩],
                   machine ⟨MachineName⟩,
                   {⟨Procedure⟩}

⟨Procedure⟩    ::= proc ⟨ProcName⟩(⟨ProcPars⟩)
                   begin
                   ⟨ScheduleBody⟩
                   end

⟨ScheduleBody⟩ ::= ⟨Expression⟩, {⟨Expression⟩}

⟨Expression⟩   ::= Event
                 | ⟨ScheduleBody⟩, {[] ⟨ScheduleBody⟩}
```

```
                  |  do ⟨ScheduleBody⟩ od
                  |  if(⟨Cond⟩){⟨ScheduleBody⟩},
                     {elseif(⟨Cond⟩){⟨ScheduleBody⟩}},
                     [else{⟨ScheduleBody⟩}]
                  |  while(⟨Cond⟩){⟨ScheduleBody⟩}

⟨Cond⟩          ::=  Predicate

⟨ScheduleName⟩  ::=  String

⟨MachineName⟩   ::=  String

⟨ProcName⟩      ::=  String

⟨ProcPars⟩      ::=  ⟨ProcPar⟩,{, ⟨ProcPar⟩}

⟨ProcPar⟩       ::=  in : ⟨Par⟩
                  |  out : ⟨Par⟩

⟨Par⟩           ::=  Event-B variable or constant
```

Appendix B Program Template Language

```
⟨Template⟩      ::=  <file name=", ⟨Name⟩, ">,
                     {⟨TemplateElement⟩}
                     </file>

⟨Class⟩         ::=  <class name=", ⟨Name⟩ ,">,
                     ⟨ClassElements⟩,
                     </class>

⟨Procedures⟩    ::=  <procedures>,
                     {⟨Procedure⟩},
                      </procedures>

⟨Procedure⟩     ::=  <procedure name=",⟨Name⟩,
                     " inpar=",⟨Bool⟩," outpar=",⟨Bool⟩,
                     " return=",⟨Bool⟩,">,
                     ⟨ProcedureBody⟩,
                     </procedure>

⟨VarDecl⟩       ::=  <vardecl/>

⟨Init⟩          ::=  <init/>

⟨Bool⟩          ::=  true | false
```

```
⟨Name⟩          ::= ⟨Char⟩ | ⟨Ref⟩, {⟨Char⟩ | ⟨Ref⟩}

⟨Char⟩          ::= a..z | A..Z | 0..9 | _ | - | .

⟨Ref⟩           ::= #SCHEDULENAME
                  | #MACHINENAME
                  | #PROCNAME
```

Appendix C Translation Rule Language

```
⟨Translations⟩ ::= <translations language=", ⟨LangName⟩, ">,
                   {⟨Rule⟩},
                   </translations>

⟨Rule⟩          ::= <rule type=", ⟨RuleType⟩, ">,
                   [⟨MetaVars⟩],
                   ⟨Source⟩,
                   ⟨Target⟩,
                   </rule>

⟨MetaVars⟩      ::= <metavariables>,
                   {⟨Var⟩},
                   </metavariables>

⟨Var⟩           ::= <var>,
                   <id>,
                   $, ⟨VarName⟩,
                   </id>,
                   <type>,
                   ⟨VarType⟩,
                   </type>,
                   </var>

⟨Source⟩        ::= Source Expression/Structure

⟨Target⟩        ::= Translation

⟨LangName⟩      ::= Name of the target language

⟨RuleType⟩      ::= class | procedure | constructor | sequence | inpar
                   | outpar | return | ifbranch | elseifbranch |
                   elsebranch | loop | vardecl | identifier | type |
                   ctype | operator

⟨VarName⟩       ::= Valid Event-B identifier name
```

$\langle \mathit{VarType} \rangle$	::=	*Valid Event-B type*