Integrating Semantically Configurable State-machine Models in a C Programming Environment

by

Zhaoyi Luo

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2015

© Zhaoyi Luo 2015

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Model-driven engineering is a popular software-development methodology, which requires suitable domain-specific modelling languages (DSLs) to create models. A DSL requires flexible semantics depending on the domain knowledge. Among DSLs, Big-Step Modelling Languages (BSML) is a family of state-machine modelling languages that vary semantically. In BSML, a model can respond to an environmental input with a big-step which comprises a sequence of small-steps, each of which represents the execution of a set of transitions. The semantics of BSMLs are decomposed into mostly orthogonal semantic aspects with a wide range of semantic options. With configurable semantics, the modeller is able to choose the proper option for each semantic aspect, thus to fulfil their per domain/model semantic requirements.

In this thesis we present BSML-mbeddr, a state-machine modelling language with hierarchical states, concurrent regions and configurable semantics, which has implemented a large subset of BSML within the mbeddr C programming language environment. mbeddr is a DSL workbench which provides a tool suite that supports the incremental construction of modular DSLs on top of C, together with a set of predefined DSLs. By implementing on mbeddr, BSML-mbeddr is integrated into mbeddr-C that supports programs made with heterogeneous languages, including a combination of programming language and modelling language.

Acknowledgements

For the past years as a graduate student, Professor Joanne M. Atlee has been all I could have asked for as a supervisor. She kindly gave me all the support I could have probably asked. I would like to thank my thesis readers, Nancy and Krzysztof to spend time reading my thesis and giving me very constructive feedback. Lastly, I would like to thank all the people who made this thesis possible.

Table of Contents

List of Tables x

List of Figures xi

1 Introduction 1

1.1 Contributions 4

1.2 Organization 5

2 Background on mbeddr 6

2.1 Language Workbench and Projectional Editor 6

2.2 MPS 7

2.2.1 Structure 8

2.2.2 Constraint 9

2.2.3 Behaviour 9

2.2.4 Type System 10

2.2.5 Editor 10

2.2.6 Generator 11

2.3 mbeddr 12

3 Background on BSML 13

3.1 BSML Syntax 13

3.2 BSML Semantics 14
3.2.1 Execution Semantics 14
3.2.2 Big-step Maximality 16
3.2.3 Concurrency and Consistency 17
3.2.4 Event Lifeline 20
3.2.5 Enabledness Memory Protocol 23
3.2.6 Assignment Memory Protocol 25
3.2.7 Order of Small-steps 26
3.2.8 Priority 27
3.2.9 Combo-step Maximality 28

4 BSML-mbeddr 30

4.1 BSML-mbeddr Syntax 30
4.1.1 Example-based Demonstration 30
4.1.2 State-machine Elements in BSML-mbeddr 31
4.1.3 Language Features 33
4.1.4 Interaction with Environment 35
4.1.5 State-Region Hierarchy 36
4.2 BSML-mbeddr Semantics 37
4.2.1 Implemented Semantic Options 37
4.2.2 Priority 40
4.2.3 Modified Execution Semantics 41
4.2.4 Present in Same and Negation of Triggers 45
4.2.5 External Event 45
4.2.6 Granularity of Semantic Configuration 46

5 Implementation **47**

5.1 Syntax Implementation 47

5.1.1 Interface Concepts 47

5.1.2 Concrete Concepts 49

5.1.3 Other Language Aspects 52

5.2 Semantics Implementation 53

5.2.1 Code Layout 53

5.2.2 Template-based Generator 55

6 Validation **57**

6.1 Correctness 57

6.2 Expressiveness 58

6.2.1 Ground Traffic Control 59

6.2.2 Dialler System 64

6.2.3 State-Machine Factory 67

7 Discussions **69**

7.1 Designing Data Structure of Generated Code 69

7.2 Evolving Semantic Configuration 71

7.3 Computational Complexity 71

7.4 Language Usability 72

7.5 Semantics of Added Language Features 72

7.6 Event with Multiple Instances 73

7.7 Big-step Semantics 74

8 Related Work **76**

8.1 Semantically Configurable Code Generator 76

8.2 Code-model Co-development 77

9 Conclusion **79**

APPENDICES **81**

A Implementation: Editor, Constraint, Type System and Behaviour **82**

A.1 Editor 82

A.2 Constraint 83

A.3 Type System 85

A.4 Behaviour 86

B Implementation: Template-based Generator **88**

B.1 Mapping configuration 88

B.2 weave_Common 91

B.3 weave_StateMachine 91

B.4 reduce_StateMachine 93

B.5 reduce_Region 96

B.6 reduce_Transition 98

B.7 Miscellaneous 98

References **103**

List of Tables

3.1 Big-step Maximality Semantic Options [7] 17

3.2 Concurrency and Consistency Semantic Options [7] 18

3.3 Event Lifeline Semantic Options [7] 21

3.4 External Input/Output Event Semantic Options [7] 22

3.5 Interface Event Semantic Options [7] 23

3.6 GC Memory Protocol Semantic Options [7] 24

3.7 Interface Variable in GC Semantic Options [7] 25

3.8 Order of Small-steps Semantic Options [7] 26

3.9 Priority Semantic Options [7] 27

3.10 Combo-step Maximality Semantic Options [7] 29

4.1 BSML Semantic Aspects/Options. Semantic options implemented in BSML-mbeddr are indicated with check marks. 39

8.1 Comparison between BSML-mbeddr and mbeddr.statemachine 78

List of Figures

2.1 Illustration of how a projectional editor works compared to a traditional parser. . . . 6
2.2 MPS Structure . . . 8
2.3 MPS Constraint . . . 9
2.4 MPS Behaviour . . . 10
2.5 MPS Typesystem . . . 10
2.6 MPS Editor . . . 11
2.7 MPS Generator . . . 11

3.1 Process of a BSML Big Step [7]. The name of the associated semantic aspect of each stage is shown in a parenthetical clause. . . . 15
3.2 Preemption Example. t interrupts t' in both cases. . . . 19
3.3 Example of assigning priority by Negation of Triggers. . . . 28

4.1 Illustration of an example model and its corresponding state hierarchy. . . . 31
4.2 Code for Example Model and Environment . . . 32
4.3 Illustration of HIERARCHICAL Priority. . . . 41
4.4 Comparison of flowgraph of BSML and BSML-mbeddr. . . . 42
4.5 Example model demonstrating unintended behaviour when options PRESENT IN SAME and NEGATION OF TRIGGERS are selected together. . . . 45

5.1 BSML-mbeddr Interface Hierarchy. mbeddr's built-in concepts are highlighted in color. . . . 48

5.2 Example Behaviours of Interface Concepts 49
5.3 BSML Syntax. mbeddr's built-in concepts are highlighted in color. 50
5.4 Structure Example, **Transition**. 51
5.5 Example of Structural Code Layout 53
5.6 Example of Behavioural Code Layout 55

6.1 GTC Case Study [26] 59
6.2 GTC Models 61
6.3 GTC Testing Environment 63
6.4 Model and Semantic Configuration of the Dialler System 65
6.5 Dialler Code 66
6.6 State-Machine Factory Model 67
6.7 State-Machine Factory Code 68

7.1 Information Flow during Code Generation and Run-time Execution. 70

A.1 Editor Example. 83
A.2 Constraint Example. 84
A.3 TypeSystem Example. 85
A.4 Behaviour Example, mbeddr.**ICallLike**. By implementing this interface, benefits such as argument type checking are gained for **EventCall** and **SMGenEvent**. 87

B.1 Mapping Configuration 89
B.2 Mapping Configuration (continue) 90
B.3 weave_Common 92
B.4 weave_Common (continue) 93
B.5 weave_StateMachine 94
B.6 weave_StateMachine (continue) 95
B.7 weave_StateMachine (continue) 96

B.8 reduce_StateMachine . . . 97
B.9 reduce_Region . . . 98
B.10 reduce_Transition . . . 99
B.11 reduce_EventCall . . . 101
B.12 reduce_SMStart . . . 101
B.13 reduce_SMTrigger . . . 102
B.14 reduce_SMTerminate . . . 102

Chapter 1

Introduction

Model-driven engineering (MDE) is a popular software-development methodology which requires suitable domain-specific languages (DSL) to create models. A DSL is a language dedicated to a specific domain, which allows domain experts to build models efficiently based on their domain knowledge and without concerning the underlying implementation details [27]. Unlike a general-purpose language (GPL), a DSL requires a domain-specific syntax and semantics since each domain has its own vocabulary and language for description and definition.

Among DSLs, state-machine modelling languages are widely applied to interactive and reactive systems in various domains, such as network protocols, and control systems of vehicles, elevators, and medical devices. However, modellers cannot agree on a single semantics for the state-machine modelling language – there is ample evidence that suggests the modellers want to use a wider set of notations and semantics [26]. Moreover, depending on the characteristics of the domain, it can be significantly more concise and understandable to model some behaviours in one semantics than in another [9]. Thus, modellers need to be able to choose language features, especially semantic features on a domain-by-domain or model-by-model basis.

Big-Step Modelling Languages (BSML [8]) is a family of state-machine modelling languages (UML StateMachines [24], Argos [21], Statecharts [17], Stateflow [6], etc.) that vary semantically. In BSML, a model responds to an environmental input with a *big-step*, which comprises a sequence of *small-steps*, each of which represents the execution of a set of transitions. At the end of a *big-step*, the output of the model is delivered to its environment. In the previous work of Esmaeilsabzali and Day [9], the variations of BSML semantics have been systematically decomposed into several high-level, mostly orthogo-

nal aspects, each of which offers multiple semantic options. As a typical example of a semantic aspect, Event Lifeline denotes how long a generated event shall be present in the state-machine execution, to trigger other transitions. If option PRESENT IN REMAINDER is chosen, the generated event shall be present during the rest of the *big-step*. Whereas if option PRESENT IN NEXT is chosen, the generated event shall be present only during the next *small-step*. By configuring semantic aspects with predefined options, the combination of options can create a large design space of BSML semantics that covers a wide range of domain-specific requirements. Under the framework of BSML[1], we have developed BSML-mbeddr, an implementation of BSML with configurable semantics, allowing the modeller to choose the proper option for each semantic aspect and fulfil their per-domain or per-model semantic requirements[2].

Our work is built on MPS and mbeddr. The Meta Programming System (MPS) is a projectional language workbench which provides a suite of language tools that support efficient definition, extension and use of DSLs [25]. In MPS, languages to be created are decomposed into various *language aspects*[3] including *structure*, *editor*, *constraint*, *type system*, *generator* as well as other aspects for supporting sophisticated IDE functionality. By defining the *generator* of a DSL, the DSL can be transformed into lower-level DSLs with Java as the lowest-level DSL (i.e., the base language) in MPS, that generates plain-text Java code.

mbeddr [35] provides C (*mbeddr-C*) as another base language on MPS. mbeddr provides a tool suite that supports the incremental construction of modular DSLs on top of C, together with a set of predefined DSLs. mbeddr allows us to write programs with a combination of low-level C code (e.g., embedded software) as well as high-level abstractions (e.g., coordination code or safety-kernel) in heterogeneous languages. For example, the control system of an elevator is an embedded system normally developed in a low-level language such as C, which requires tedious conditional checking that is both error-prone and non-intuitive. For safety reasons, parts of the code might need to be model checked, requiring significant work for programmer to abstract the code into an model suitable for model checking. With mbeddr, the programmer is able to write normal C code mixed with interoperable state-machine models. The mbeddr program (mbeddr-C and state-machine

[1] In the rest of the thesis, we use "BSML" to indicate the family of semantically deconstructed state-machine modelling languages, with various semantic aspects and options.

[2] In BSML-mbeddr, a *semantic configuration* is associated to a *mbeddr program*, which may contain the definition and usage of multiple state-machine types, whose execution semantics follow the same semantic configuration. Details are discussed in Section 4.2.6.

[3] Please distinguish between semantic aspects and language aspects: semantic aspects in BSML are the semantic variation points that result from the semantic deconstruction of BSML; language aspects in MPS describe different language constructs of a language to be created.

model) can be transformed into plain-text C code for execution, whereas the state-machine model can be transformed into NuSMV for verification [28]. mbeddr provides support for basic state-machine models that have simple execution semantics, whereas BSML-mbeddr extends mbeddr to support a large subset of the BSML family, including state machines with hierarchical states, concurrent regions, and configurable semantics.

Our choice of the language (BSML) and platform (mbeddr) results in several advantages for our work:

- **By implementing within mbeddr, BSML-mbeddr allows creating programs with a combination of code and model, which may interact with each other.** A DSL built with traditional compiler technologies supports only the defined high-level domain-specific notation. Whereas with BSML-mbeddr, the modeller can define high-level state-machine models surrounded by interoperable mbeddr-C code which can, for example, send environmental inputs to the state-machines.
- **With mbeddr, DSLs are highly extensible and modular so that the language creator can easily evolve the language or build corresponding tools for analysis.** DSLs in mbeddr/MPS are divided into mostly orthogonal language aspects. For example, the *editor* aspect (concrete syntax) can be changed without changing the *structure* aspect (abstract syntax). We can easily change the concrete representation for the logical *and* operator from "&" to "&&", or change the concrete representations of binary operations from infix style to prefix style; multiple generators can be created for the same source language, to generate various target languages, without modifying aspects other than the *generator*. DSLs in mbeddr/MPS can be created as language modules or extensions, and analysis tools can be built based on the shared language module, which are accessible in all the languages that reuse the shared module.
- **Configurable semantics of BSML allow the modeller to choose suitable semantic options.** As we discussed earlier, configurable semantics is critical for BSML-mbeddr. It has been observed that modellers may abandon an existing modelling language and create their own simply because the existing language does not fit their semantic requirements [26]. BSML-mbeddr allows the modeller to configure the execution semantics of their models to fulfil their per-domain or per-model requirements.
- **BSML-mbeddr supports sophisticated state-machine constructs that are available in real-world notations used by professionals.** Specifically, BSML-mbeddr supports language features such as hierarchical states, concurrent regions,

event binding, static variables, entry blocks, cross-hierarchy transitions, which make the language more usable and make it easier for the modeller to construct real-world state-machine models.

Thesis Statement:

It is feasible to integrate multiple kinds of state-machine models into a programming environment, thereby creating a programming environment where a developer can create a program that intermixes C code with the developer's choice of state-machine models. A state-machine type can be semantically configurable so that it may exhibit various behaviour semantics through easy-to-change semantic parameters. It is feasible to support the evolution of the semantic configuration, where the modeller is allowed to select and change semantic options, thereby changing the execution semantics of a state-machine model. State-machine models with sophisticated language features, along with the configurable semantics, can be transformed into a low-level programming language such as C for execution.

1.1 Contributions

The contributions of this thesis are as follows:

- We have built BSML-mbeddr, a proof-of-concept implementation of BSML within mbeddr. BSML-mbeddr supports powerful state-machine modelling features including hierarchical states, concurrent regions and configurable semantics. To the best of our knowledge, this is the first attempt that validates that the semantically deconstructed BSML is implementable.

- Our work is a non-trivial extension of the mbeddr eco-system that allows one to create sophisticated state-machine models within mbeddr. We have shown that it is feasible to seamlessly integrate state-machine models with configurable semantics into a low-level C programming environment.

- We have evaluated the expressiveness of BSML-mbeddr by conducting several case studies. Each case study exercises one or more features of BSML-mbeddr, including: 1) the big-step semantics; 2) hierarchical states and cross-hierarchy transitions; 3) concurrent regions and inter-region communication; 4) configurable semantics; 5) code-model integration and interaction.

1.2 Organization

The rest of the thesis is organized as follows. Chapter 2 introduces background knowledge on MPS and mbeddr. Chapter 3 introduces background knowledge on BSML. Chapter 4 illustrates the syntax and semantics of BSML-mbeddr. Chapter 5 describes the implementation and code generation of BSML-mbeddr. Chapter 6 describes how BSML-mbeddr has been tested and evaluated, including case studies that validates the big-step semantics, hierarchical states, concurrent regions, configurable semantics and code-model interaction of BSML-mbeddr. Chapter 7 discusses the challenges we encountered during the implementation. Chapter 8 summarizes related work and Chapter 9 concludes the thesis.

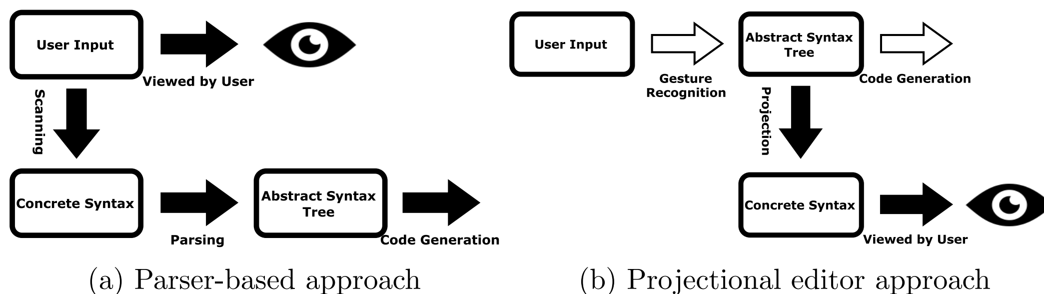
Chapter 2

Background on mbeddr

In this chapter, we provide a light overview of the background knowledge on mbeddr, the language workbench within which we implement BSML. Readers who are interested in more details about mbeddr may look at the mbeddr user guide [22].

2.1 Language Workbench and Projectional Editor

A language workbench is a platform and tool suite for efficient creation, composition, evolution and use of DSLs [33]. With the help of a language workbench, new DSLs and accompanying analysis tools can be created with a high degree of language modularization, reusability and reduction of redundant work. Without understanding the underlying implementation details, the language creator can simply wield the tools provided by the language



(a) Parser-based approach (b) Projectional editor approach

Figure 2.1: Illustration of how a projectional editor works compared to a traditional parser.

workbench to describe the DSLs to be built. According to Martin Fowler [14], a DSL is defined in a language workbench in three main parts: *schema*, *editor* and *generator*. The *schema* defines the abstract syntax of the language. The abstract syntax of a model that obeys the *schema* is persisted, often using XML or a database [34]. The *editor* defines how the schema (abstract syntax) shall be *projected* to graphical or textual representations for visualization (concrete syntax). The *generator* resembles the code-generation phase of a traditional compiler – it transforms the abstract syntax into low-level textual code.

The mechanism described above is known as *projectional editing*. The most important characteristic of a projectional editor is that the editor (concrete syntax) is separated and derived from the schema (abstract syntax). In a traditional compiler (Figure 2.1a), a program's concrete syntax is retrieved from the source code and is parsed into an abstract syntax; parsing is followed by code generation. In contrast, no textual code or concrete syntax is stored for a projectional editor – when the user types, the abstract syntax of the model is created and stored; the abstract syntax is projected to user-readable concrete representations (Figure 2.1b). There are several advantages to projectional editing [34]: a) no grammar or parser is used, which releases the language syntax from the limitations of a parser (e.g., the limitation that a LR(1) parser can only parse a syntax with unambiguous, context-free grammars); b) the model can be mapped to graphical, as well as textual representations. This makes the visualization more flexible and intuitive, especially for mathematical formulas; c) code auto-completion, error checking and syntax highlighting is provided by defining the editor aspect. This is automatically done by the language workbench (IDE) without input from the language creator; d) since projectional editing separates the concrete syntax from the abstract syntax, it eases evolution of the editor, and enables the construction of multiple projections without any changes to the abstract syntax of the model.

One main challenge to projectional editing is to simulate textual editing. When using a projectional editor, the user cannot modify the visualized model in arbitrary ways since arbitrary changes may introduces incomplete or inconsistent information. When the user performs mouse and keyboard operations on the model, the IDE will "guess" the user's purpose and make valid changes to the abstract syntax of the model, which requires understanding users' interaction patterns with textual code.

2.2 MPS

JetBrains MPS [25] is an open-source language workbench based on projectional editing, providing a suite of language tools that support efficient definition, extension and use of

```
concept FunctionCall extends    Expression
                     implements IModuleContentRef
                                IStepIntoable
                                ICallLike

  instance can be root: false
  alias: <no alias>
  short description: --

  properties:
  << ... >>

  children:
  actuals : Expression[0..n]

  references:
  function : FunctionSignature[1]
```

Figure 2.2: MPS Structure

DSLs. Following Martin's definition of a language workbench, MPS divides the definition of a DSL into *structure* (schema), *editor* and *generator* aspects. In addition, MPS provides language aspects such as *constraint*, *type system*, *behaviour*, as well as other aspects for supporting sophisticated IDE functionality such as *intention* and *action*. MPS provides language tools for the language creator to define each aspect of a DSL; these aspects are described in greater detail in the following subsections.

2.2.1 Structure

The first step to creating a new DSL is to define its abstract syntax, which is done by defining *concept*s that act as types of nodes in the abstract syntax tree. Concepts can be defined hierarchically, similarly to Java's class hierarchy; there are interface concepts and abstract concepts that resemble Java's interfaces and abstract classes. In this way, the abstract syntax can evolve and be extended with no invasive changes made to existing language concepts [34]. As an example, Figure 2.2 shows the *structure* of the **FunctionCall**[1] concept. The **FunctionCall** concept extends the **Expression** concept and implements several interface concepts. It contains zero-to-many **Expression** concepts which are the actual arguments of the call. It contains a reference to a **FunctionSignature** node which is the function declaration. In addition, a concept can have properties with types such as string (e.g., denoting the name of a variable) or boolean (e.g., denoting whether a variable is declared as static).

[1]In the rest of the thesis, we make the names of concepts in **bold** to ease the presentation.

```
concepts constraints FunctionCall {
  can be child <none>

  can be parent <none>

  can be ancestor <none>


  <<property constraints>>

  link {function}
    referent set handler:<none>
    scope:
      (exists, referenceNode, contextNode, containingLink, linkTarget, operationContext, enclosingNode, model, position,
        enclosingNode.ancestor<concept = IVisibleElementProvider, +>.visibleContentsOfType(concept/FunctionSignature/).
            select({~it => it : FunctionSignature; });
      }
    validator:
      <default>
    presentation :
      (exists, referenceNode, contextNode, containingLink, linkTarget, operationContext, enclosingNode, model, position,
        parameterNode.name;
      }
  ;
```

Figure 2.3: MPS Constraint

2.2.2 Constraint

The *constraint* aspect imposes constraints on relationships between nodes – we can prescribe whether a given node can be a child/parent/ancestor of the current node, and add constraints on node properties or the search scope of a reference node. Figure 2.3 shows the constraint aspect of **FunctionCall**, in which the search scope of the reference node $function$ is defined to be all visible **FunctionSignature** nodes under the scope of the nearest enclosing **IVisibleElementProvider** node.

2.2.3 Behaviour

The *behaviour* aspect, which is analogous to Java methods, defines methods that retrieve meaningful information from a node. Abstract methods without a method body can be defined in the behaviour of an interface concept, and a concrete concept that implements the interface must override them with concrete methods. Figure 2.4 shows the behaviour aspect of the **FunctionCall** concept. Two methods $rebindToProxy()$ and $referencedModuleContent()$ are defined, which override abstract methods from the interface concept **IModuleContentRef**.

```
concept behavior FunctionCall {

  constructor {
    <no statements>
  }

  public void rebindToProxy(node<> proxyElement)
    overrides IModuleContentRef.rebindToProxy {
    this.function = proxyElement : FunctionSignature;
  }

  public node<IModuleContent> referencedModuleContent()
    overrides IModuleContentRef.referencedModuleContent {
    return this.function;
  }
```

Figure 2.4: MPS Behaviour

```
checking rule check_FunctionCall {
  applicable for concept = FunctionCall as fc
  overrides false

  do {
    if (fc.function.hasEllipsis) {
      if (fc.actuals.size < fc.function.arguments.size) {
        error "wrong number of arguments; expecting " + fc.function.signatureInfo() -> fc;
      }
    } else {
      if (fc.actuals.size != fc.function.arguments.size) {
        error "wrong number of arguments; expecting " + fc.function.signatureInfo() -> fc;
      }
    }
  }
}
```

Figure 2.5: MPS Typesystem

2.2.4 Type System

MPS's type system is based on unification. The *type system* aspect defines a set of declarative type rules for concepts which are used by the MPS type solver to derive the type of each node. The automated typing guarantees the absence of type mismatches; otherwise a type error is thrown. The type system can be extended by adding new type rules without changing existing rules.

In addition, MPS supports the definition of non-typesystem rules that can be checked by the MPS type checker. Basically, such rules impose extra constraints on type conformity. Figure 2.5 shows the non-typesystem rule of **FunctionCall** that checks whether the number of actual parameters matches the number of formal parameters as declared.

2.2.5 Editor

In a traditional compiler, a program's concrete syntax is retrieved from the source code, and then parsed to an abstract syntax. In contrast, in MPS the abstract syntax of the

```
<default> editor for concept FunctionCall
  node cell layout:
    [- ( % function % -> { name } ) ( FE(- % actuals % /empty cell: * R/O model access * -) ) -]
```

Figure 2.6: MPS Editor

```
reduction rules:
  [concept     ItExpression]  --> content node:
  [inheritors  false       ]      void dummy() {
  [condition   <always>    ]        int8 __it;
                                    <TF [__it] TF>;
                                  } dummy (function)

  [concept     ForEachStatement] --> content node:
  [inheritors  false           ]     void dummy() {
  [condition   <always>        ]       int8[12] x;
                                       <TF [for ($COPY_SRC$[int64] __c = 0; __c < $COPY_SRC$[10]; __c++) { ] TF>
                                             $COPY_SRC$[int8] __it = $COPY_SRC$[x][__c];
                                             $COPY_SRCL$[int8 x; ]
                                           } for
                                     } dummy (function)
```

Figure 2.7: MPS Generator

model is stored, and transformed to concrete representations defined by the *editor* aspect. The *editor* consists of *cell*s, as shown in Figure 2.6. Each cell can contain a combination of literal text, symbols, and values of properties. In the figure, the **FunctionCall** node is projected to the name of reference node *function*, followed by a literal "(", followed by a list of actual arguments, followed by another literal ")". Then the editor of each actual argument is applied, to project the argument node onto its concrete representation.

2.2.6 Generator

The generation process (semantics) in MPS is basically a model-to-model transformation. The *generator* aspect defines how to transform the model either to a model in the base language or to a model in an intermediate language; the *generator* of the intermediate language is then applied to generate the corresponding model in the base language. Figure 2.7 shows the generator of a **ForEachStatement**, which transforms a *for-each* statement into an equivalent *for* statement. In MPS, the transformation process is defined independently from the language syntax, so that multiple generators can be defined to generate models in different target languages from the same source model.

2.3 mbeddr

mbeddr [22][35] provides support for C on MPS by implementing C as a base language (*mbeddr-C*). In addition, mbeddr provides a tool suite that supports incremental construction of modular DSLs on top of C, together with a set of predefined DSLs such as components, physical units and state machines. These DSLs, which greatly extend the ability of the C developer to program from an abstract perspective, are transformed into lower-level DSLs with *mbeddr-C* as the lowest-level DSL (i.e., the base language). The state-machine modelling language of mbeddr is described in detail in the related work (Chapter 8).

Chapter 3

Background on BSML

Big-Step Modelling Language (BSML) [7][8][9] is a family of state-machine modelling languages (UML StateMachines [24], Argos [21], Statecharts [17], Stateflow [6], etc.) that vary semantically. In this chapter we give a light overview on BSML including its syntax, basic execution semantics, and configurable semantics that are from the PhD thesis of Esmaeilsabzali [7]. Although BSML-mbeddr is an implementation of BSML, it varies from BSML in several ways. In Chapter 4 we will discuss the syntax and semantics of BSML-mbeddr, and how it differs from BSML.

3.1 BSML Syntax

A BSML *state machine* contains *control states*. A *control state* has a name and a *type*, which is either a *simple* state or a *composite* state. A *composite* state is either a *And* state or a *Or* state. The set of control states of a model forms a *hierarchy tree*. A leaf node of a hierarchy tree is a *simple* control state, whereas an *And* or an *Or* control state is a non-leaf node of a hierarchy tree. If a model resides in an *And* control state, it resides in all of its children. If a model resides in an *Or* state, it resides in one of its children, which is by default its *initial state*.

Two control states *overlap* if they are the same or one is an ancestor of the other. The *least common ancestor* of two control states is the lowest control state (closest to the leaves of the hierarchy tree) that is an ancestor of both. Two control states are *orthogonal* if neither is an ancestor of the other and their least common ancestor is an *And* control state. The *scope* of a transition is the least common ancestor of its source and target

control states. The *arena* of a transition is the lowest *Or* control state in the hierarchy tree that is the ancestor of both the source and target control states of the transition.

A transition has a name, a *source* and a *target* control state, and four optional parts: 1) a conjunction of *triggering events*, some of which may be negated; 2) a *guard condition* which is a boolean expression over the set of state-machine variables; 3) a set of assignments; and 4) a set of generated events.

3.2 BSML Semantics

In this section, we first describe the basic execution semantics of BSML, and we briefly explain how semantic variation points (configurable semantics) reside in the process of the basic execution semantics. Next, in the following sub-sections, we introduce each semantic *aspect* and its *options*.

3.2.1 Execution Semantics

The execution semantics describes how a state-machine model handles an environmental input, responds by executing transitions, and communicates its outputs. The basic unit of handling an environmental input is a *big-step*. A big-step begins with the state machine accepting a single environmental input from its environment, and ends with delivering outputs, as a result of executing a sequence of small-steps. The process of a big-step can be deconstructed into the stages described in Figure 3.1.

A big-step starts by accepting an environmental input. Then a small-step is started by identifying transitions enabled by events and variables. Only transitions that satisfy certain ordering constraints can be determined as enabled. Next, the maximality of combo-step and big-step is determined. If the maximal big-step is reached, then the big-step ends and environmental outputs are delivered. Otherwise, all maximal, consistent sets of transitions are identified from the set of enabled transitions as candidates of the current small-step. One of the identified sets with highest priority is chosen to be executed, which means that variables in the RHS of assignments of the chosen transitions are evaluated and the new status of the state machine is calculated.

Each stage in Figure 3.1 is associated with a semantic variation point (i.e., *semantic aspect*, which is shown in a parenthetical clause in the figure) in the basic execution semantics of a state machine. A semantic aspect may be decomposed into some semantic sub-aspects.

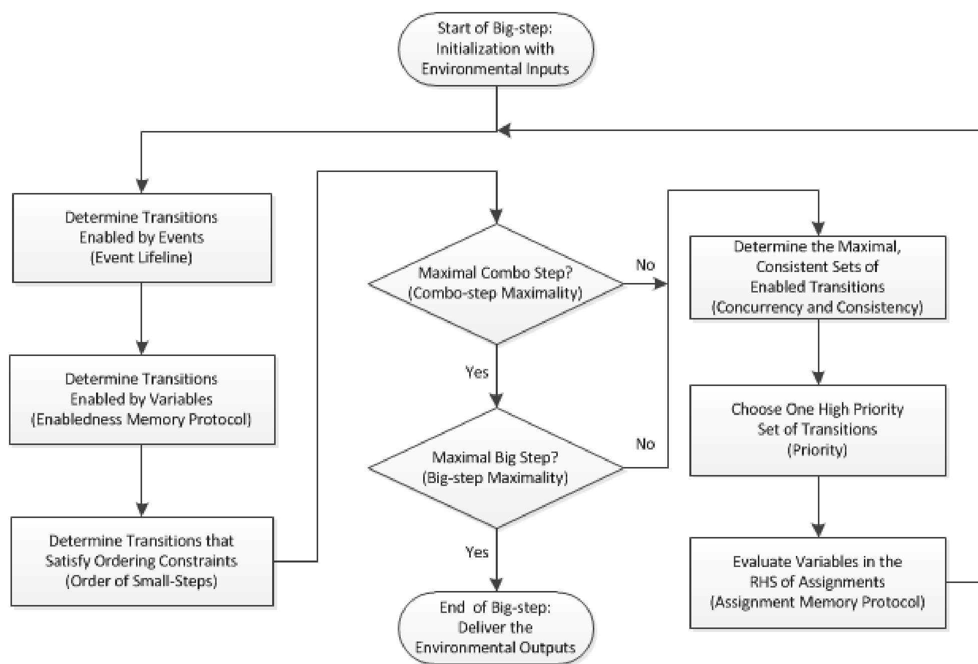


Figure 3.1: Process of a BSML Big Step [7]. The name of the associated semantic aspect of each stage is shown in a parenthetical clause.

Each semantic aspect or sub-aspect has several semantic options to choose from. In the rest of the thesis, we use font Sans Serif for the name of semantic aspects, and we use font SMALL CAP for the name of semantic options. The Big-step Maximality semantic aspect determines when a big-step ends, at which point environmental outputs are delivered and a new big-step starts by sensing new environmental inputs. The Combo-step Maximality semantic aspect specifies when a combo-step ends, at which point a new combo-step starts by committing the changes of values of variables and statuses of events in the current combo-step. The Event Lifeline semantic aspect specifies how far within a big-step a generated event can be sensed as present to trigger a transition. The Enabledness Memory Protocol semantic aspect specifies the snapshot from which the values of variables are read to enable the guard condition of a transition. The Assignment Memory Protocol semantic aspect specifies the snapshot from which the value of a variable in the right-hand side of an assignment is read. The Order of Small-steps semantic aspect describes options for constraining the order of transitions that execute within a big-step. The Concurrency and Consistency semantic aspect specifies which enabled transitions can be executed together in the same small-step. Lastly, the Priority semantic aspect defines priorities among transitions, which is used to choose from among the maximal, consistent sets of enabled transitions to execute.

3.2.2 Big-step Maximality

The semantic aspect Big-step Maximality determines the extent of a big-step. A big-step begins by accepting an environmental input, and Big-step Maximality determines when it ends. The least restrictive option is TAKE MANY: a big-step ends when no more transitions can be executed. TAKE ONE imposes an additional constraint that whenever a transition is executed, no transition with an overlapping arena can subsequently be executed in the same big-step. SYNTACTIC is appropriate when the modeller is able to syntactically tag a state as being **stable**. With this option, whenever an executed transition lands in a stable state, no transitions with an overlapping arena can subsequently be executed in the same big-step. The advantages of the options TAKE ONE and SYNTACTIC are that they are simple, but their constraints that prevent overlapping transitions from executing in the same big-step might be too restrictive for some models. The option TAKE ONE guarantees that a big-step eventually terminates while the other two options do not.

Option	Definition	Pros and Cons
TAKE MANY	Small-steps continue until there are no more enabled transitions.	(+) Expressive (–) Non terminating big-step is possible
TAKE ONE	No two transitions with overlapping arenas can be taken in the same big-step.	(+) Simple (+) Terminating big-step is guaranteed (–) Limited
SYNTACTIC	No two transitions with overlapping arenas that enter designated "stable" state can be taken in the same big-step.	(+) Syntactical scope for big-step (–) Non terminating big-step is possible

Table 3.1: Big-step Maximality Semantic Options [7]

3.2.3 Concurrency and Consistency

Concurrency

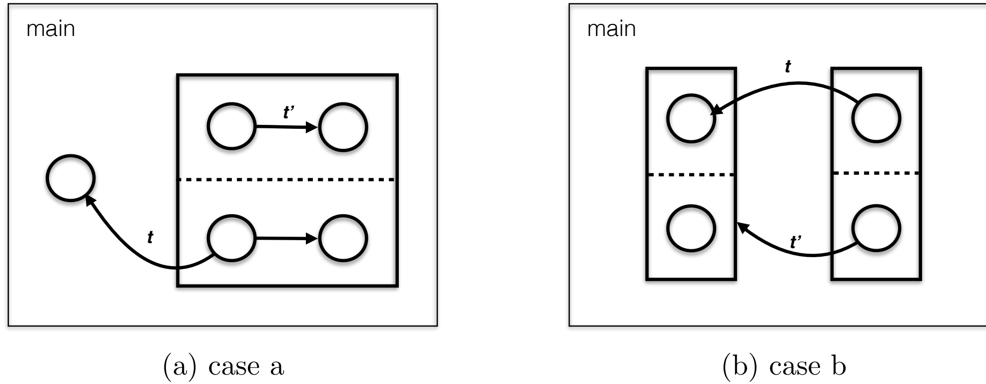
The semantic aspect Concurrency defines whether concurrent execution is allowed – that is, whether a small-step can comprise the execution of multiple transitions. Option SINGLE allows only one transition to be executed in a small-step, whereas option MANY allows multiple transitions to be executed in a small-step. The semantics of the SINGLE option are simple and easy to understand, but might result in a nondeterministic model. Because the MANY option allows transitions to be executed concurrently, it is possible for a model's execution to have a race condition (e.g., the execution of multiple transitions in the same small-step might write to the same share variable).

Consistency

For two enabled transitions where neither is an *interrupt* of the other and the semantic aspect Concurrency is MANY, the Consistency aspect determines whether they can be executed in the same small-step. The option ARENA ORTHOGONAL requires the transitions' arenas to be orthogonal in order for them to execute in the same small-step, whereas the option SOURCE-TARGET ORTHOGONAL requires that the transitions' source states and target states are pairwise orthogonal in order for them to execute in the same small-step.

Option	Definition	Pros and Cons
Concurrency		
SINGLE	Only one transition can be executed in a small-step.	(+) Simple (−) Nondeterministic
MANY	Multiple transitions can be executed in a small-step.	(+) Low chance of nondeterminism (−) Race conditions
Consistency		
ARENA ORTHOGONAL	Two transitions whose arenas are orthogonal can be executed in the same small-step.	(+) Simple (−) More restrictive
SOURCE-TARGET ORTHOGONAL	Two transitions whose source states and target states are pairwise orthogonal can be executed in the same small-step.	(+) Less restrictive (−) Complex
Preemption		
NON-PREEMPTIVE	Interrupter and interruptee can be executed in the same small-step.	(+) Support of "Last wish"
PREEMPTIVE	Interrupter and interruptee can not be executed in the same small-step.	(+) No "Last wish"

Table 3.2: Concurrency and Consistency Semantic Options [7]



(a) case a (b) case b

Figure 3.2: Preemption Example. t interrupts t' in both cases.

Note that option Source-Target Orthogonal is less restrictive than option Arena Orthogonal since all pairs of transitions that are arena orthogonal are also source-target orthogonal.

Preemption

When two transitions are enabled in the same small-step and the semantic aspect Concurrency is Many, there is a special case called *interruption* where Consistency does not apply. The semantic aspect Preemption governs the semantics of interruption, which is useful when the execution of a transition that exits its enclosing control state needs to interrupt the execution of transitions in orthogonal control states. Formally, we say that transition t *interrupts* transition t' if their source states are orthogonal, and either a) the target state of t' is orthogonal with the source state of t, and the target state of t is not orthogonal with the source states of both transitions (Figure 3.2a); or b) the target states of both transitions are not orthogonal with the source states of both transitions, and the target state of t is descendant of the target state of t' (Figure 3.2b)). To simplify our descriptions below, we say that the transition that does the interrupting is the *interrupter*, whereas the transition being interrupted is the *interruptee*.

Preemption has two options: Non-Preemptive and Preemptive. The Non-preemptive option allows interrupter and interruptee to be executed in the same small-step, which means that both transitions' actions are executed, but the state machine lands in the target state of the interrupter as if only the interrupter is executed. When Preemptive is chosen, interrupter and interruptee cannot be executed together in the same small-step.

Intuitively, the Preemption aspect is useful in situations where the execution of a high-priority transition (e.g., when a system error, an exception, or some high-priority event occurs) needs to abort the execution of lower-priority transitions. The difference between NON-PREEMPTIVE and PREEMPTIVE is that the NON-PREEMPTIVE option allows the lower-priority transition to finish its "last wish" actions, whereas the PREEMPTIVE option does not. Note that the semantics of Preemption itself does not impose any bias towards the interrupter. To impose a bias towards the interrupter, the modeller may use language constructs such as negated triggering events or explicit priority.

3.2.4 Event Lifeline

The semantic aspect Event Lifeline determines how long a generated event remains *present* in the big-step, and thus how long the event is able to trigger transitions. Table 3.3 shows the five Event Lifeline semantics: 1) in the PRESENT IN WHOLE option, a generated event is present throughout the big-step, from the beginning of the big-step; 2) in the PRESENT IN REMAINDER option, a generated event is present in the snapshot after it is generated and persists until the end of the big-step; 3) in the PRESENT IN NEXT COMBO option, a generated event is present only during the next combo-step; 4) in the PRESENT IN NEXT SMALL option, a generated event is present only during the next small-step; and 5) in the PRESENT IN SAME option, a generated event is present only during the small-step in which it is generated.

If option PRESENT IN NEXT SMALL or PRESENT IN REMAINDER is chosen, then a big-step is *causal*, which means any executed transition in a small-step must be triggered by events that are generated in some earlier small-step, whereas a big-step containing the execution of transitions triggered by rendezvous events (which applies in option PRESENT IN SAME) may not be causal. The semantics of PRESENT IN REMAINDER lacks the *orderedness* property: if event e_1 is generated earlier than event e_2, it need not be the case that transitions triggered by e_1 are executed earlier than traditions triggered by e_2. The PRESENT IN NEXT COMBO was devised to alleviate this problem by having a "rigorous causal ordering" between combo-steps, while being insensitive to the order of event generation within a combo-step [7]. The PRESENT IN NEXT SMALL semantics is ordered: a transition triggered by an internal event can be executed only if the internal event is generate by a transition in a previous small-step.

The PRESENT IN REMAINDER semantics can produce a *globally inconsistent* big-step, when the big-step includes a transition that generates an event and a transition triggered by the *absense* of that event. *Global inconsistency* is undesired because an

Option	Definition	Pros and Cons
Present in Whole	A generated event in a big-step is assumed to be present throughout the same big-step.	(+) Modularity (+) Global consistency (−) No causality
Present in Remainder	A generated event in a big-step is sensed as present in the same big-step after it is generated.	(+) Causality (−) No orderedness (−) Global inconsistency
Present in Next Combo	A generated event can be sensed as present only in the next combo-step after it is generated.	(+) Causality (−) Partial orderedness
Present in Next Small	A generated event can be sensed as present only in the next small-step after it is generated.	(+) Causality (+) Orderedness
Present in Same	A generated event can be sensed as present only in the same small-step it is generated in.	(+) Instantaneous communication (−) No causality

Table 3.3: Event Lifeline Semantic Options [7]

event is sensed both as absent and present in the same big-step. Present in Whole is globally consistent, and the other options are globally inconsistent but by design. The Present in Whole option is *modular*: an event generated during a big-step can be conceptually considered the same as an environmental input event because it is present from the beginning of the big-step. All other options except Present in Whole are non-modular.

External Event

In BSML, we may choose distinct Event Lifeline options for environmental input event (in-event), environmental output event (out-event), and internal event. The determination of an in-event (out-event) depends on the semantic aspect External Input Events (External Output Events). An event that is neither an in-event nor an out-event is treated as an internal-event.

Option Syntactic for External Input Events is appropriate when the modeller syntactically tags an event as being an in-event. If option Received in First Small is chosen, then any event can be generated by the environment, but only those events that are re-

Option	Definition	Pros and Cons
SYNTACTIC	In-events or out-events are determined syntactically.	(+) Simple (−) Syntax burden to modeller
RECEIVED IN FIRST SMALL/ GENERATED IN LAST SMALL	An event received in the first small-step is determined to be an in-event. An event generated in the last small-step is determined to be an out-event.	(+) Treats external and internal events uniformly (−) No boundary between model and environment
HYBRID	An event that is received at the beginning of a big-step and is never generated in the model is determined to be an in-event. An event that is generated in the last small-step and is not a triggering event for any transition in the model is determined to be an out-event.	(+) No syntax burden to modeller (−) Complex

Table 3.4: External Input/Output Event Semantic Options [7]

ceived at the beginning of a big-step (i.e., generated by the environment, or received from the input queue) are determined to be in-events. In the HYBRID option, an event that is received at the beginning of a big-step and is never generated in any transition or entry block is determined to be an in-event.

The SYNTACTIC option for External Output Events regards any event that is syntactically bound to a function as an out-event. If the semantic option GENERATED IN LAST SMALL is chosen, only events that are generated in the last small-step are determined to be out-events. In the HYBRID option, an event that is generated in the last small-step and is not a triggering event for any transition in the model is determined to be an out-event.

Interface Event

In BSML, a state-machine model may be structured as a set of *components*, each of which is a composite control state. *Components* communicate with each other only through interface events or variables. Table 3.5 lists the three options of interface events for inter-component communication. In the STRONG SYNCHRONOUS EVENT option, a generated interface event is sensed as present throughout the big-step in which it is generated, from

Option	Definition	Pros and Cons
STRONG SYNCHRONOUS EVENT	A generated interface event of a big-step is sensed as present from the beginning of the big-step.	(+) Modularity (−) No causality
WEAK SYNCHRONOUS EVENT	A generated interface event of a big-step is sensed as present in the snapshot after it is generated.	(+) Causality (−) Globally inconsistency
ASYNCHRONOUS EVENT	A generated interface event of a big-step is sensed as present in the next big-step after it is generated.	(+) Modularity (−) Previous big-step affects current big-step

Table 3.5: Interface Event Semantic Options [7]

the beginning of the big-step (similar to PRESENT IN WHOLE). In the WEAK SYNCHRONOUS EVENT option, a generated interface event is present in the big-step in which it is generated, but only after it is generated (similar to PRESENT IN REMAINDER). In the ASYNCHRONOUS EVENT option, a generated interface event is present in the next big-step, from the beginning of the big-step.

3.2.5 Enabledness Memory Protocol

The values of variables that a transition reads for its guard condition (GC) are determined by the GC Memory Protocol (i.e., *enabledness memory protocol*) semantic aspect. During the execution of a BSML state-machine model, three snapshots of variable values are kept to be read: *snapshot_big* retains the values that variables had at the beginning of the big-step, *snapshot_combo* retains the values that variables have at the beginning of the current combo-step, and *snapshot_small* retains the values that variables have at the beginning of the current small-step. In addition, we have *snapshot_cur* that holds the new values being computed in the current small-step which will overwrite the values in *snapshot_small* at the end of the current small-step, and overwrite the values in *snapshot_combo* at the end of the current combo-step. The GC BIG STEP, GC COMBO STEP, and GC SMALL STEP options use the variable values stored in *snapshot_big*, *snapshot_combo*, and *snapshot_small* when evaluating expressions in guard conditions, respectively.

The GC BIG STEP option is $non\text{-}interfering$, which means an earlier small-step of a big-step does not affect the read value of a later small-step. Non-interference relieves the

Option	Definition	Pros and Cons
GC Big Step	Values of variables are read from the snapshot at the beginning of the big-step when evaluating expressions in guard conditions.	(+) Non-interference (−) Non-sequentiality
GC Small Step	Values of variables are read from the beginning of the current small-step when evaluating expressions in guard conditions.	(+) Sequentiality (−) Interference
GC Combo Step	Values of variables are read from the beginning of the current combo-step when evaluating expressions in guard conditions.	(+) Some interference (+) Some interference

Table 3.6: GC Memory Protocol Semantic Options [7]

modeller from considering the accumulative effects of assignments to a variable during a big-step. In contrast, the GC Small Step is *sequential*, which means assignments to variables are able to subsequently affect the execution of the following small-steps. Sequentiality enables the modeller to decompose the computation process of the final output into multiple stages, where each is carried out by a separate small-step.

Interface Variables in Guard Conditions

Components can communicate among each other through not only interface events but also through interface variables. Table 3.7 lists the possible options for the Interface Variables in GC semantic aspect, which determines when a change to an interface-variable value becomes the value returned by a read of that variable in a guard condition. In the GC Strong Synchronous option, either an interface variable is not written to during a big-step, or all of its reads happen after it has been written to and the newly assigned value is returned by a read of that variable. In the GC Weak Synchronous option, a write to an interface variable can be read after the variable is written to, but the variable can also be read before it is written to, in which case its value from the previous big-step is returned by a read of that variable (similar to GC Small Step). In the GC Asynchronous option, a write to an interface variable can be read by the guard condition of any transition in the next big-step (similar to GC Big Step).

Option	Definition	Pros and Cons
GC STRONG SYNCHRONOUS	Either an interface variable is not written to during a big-step, or all of its reads happen after it has been written to and it returns the newly assigned value.	(+) Modularity (−) Blocking read and requiring dataflow analysis
GC WEAK SYNCHRONOUS	An interface variable can be read before or after it is written to; in the latter case, it returns the newly assigned value.	(+) Non-blocking read (−) Stale values for interface variables
GC ASYNCHRONOUS	The value written to an interface variable during a big-step can be read in the next big-step.	(+) Non-blocking read (+) Modularity (−) Delayed read

Table 3.7: Interface Variable in GC Semantic Options [7]

The GC STRONG SYNCHRONOUS option blocks a read operation if there exists a write operation in the same big-step. Thus, there should exist a dataflow order to ensure that the value of an interface variable is read only after it has been assigned. In the GC WEAK SYNCHRONOUS option, a read operation on a variable never blocks, but a *stale value* from the previous big-step may be returned by a read of the variable. In the GC ASYNCHRONOUS option, a read operation on a variable never blocks, but the communication among components is delayed.

3.2.6 Assignment Memory Protocol

The values of variables that a transition reads when evaluating the right-hand side (RHS) of an assignment are determined by the RHS Memory Protocol (i.e., *assignment memory protocol*) semantic aspect. Exactly the same semantic options as those of the enabledness memory protocol (Section 3.2.5) exist: RHS BIG STEP, RHS SMALL STEP, and RHS COMBO STEP. The modeller is allowed to select distinct options for assignment memory protocols from enabledness memory protocol. The same advantages and disadvantages as the semantic options for enabledness memory protocols apply to corresponding semantic options for assignment memory protocols.

Option	Definition	Pros and Cons
NONE	Small-steps are not ordered.	(+) Simplicity (−) Nondeterminism
EXPLICIT	Execution of small-steps is ordered syntactically.	(+) Control over nondeterminism (−) Possible unintended ordering
DATAFLOW	Small-steps are ordered so that an assignment to a variable happens before it is being read.	(+) Control over nondeterminism (−) Possible cyclic orders

Table 3.8: Order of Small-steps Semantic Options [7]

Interface Variables in RHS

Similar to the usage of interface variables in the guard condition (GC) of transitions, as described in Section 3.2.5, the semantics of interface variables are regulated by the Interface Variable in RHS semantic aspect. Exactly the same semantic options as those for Interface Variable in GC exist: RHS STRONG SYNCHRONOUS, RHS WEAK SYNCHRONOUS, and RHS ASYNCHRONOUS. The same advantages and disadvantages as the semantic options for Interface Variable in GC apply to corresponding semantic options for Interface Variable in RHS.

3.2.7 Order of Small-steps

The Order of Small-steps semantic aspect is introduced in BSML to reduce the number of enabled transitions within a small-step, thereby increasing the understandability of the model. The Order of Small-steps semantic aspect specifies a precedence relationship among transitions, which means that each transition has a set of *preceding* transitions. A transition is enabled only if each of its preceding transitions either is disabled or has already been executed in the current big-step. Table 3.8 lists all of the possible options for semantic aspect Order of Small-steps. In the NONE option, no such precedence relationship exists among transitions. In the EXPLICIT option, each transition is explicitly and syntactically associated with a set of preceding transitions. In the DATAFLOW option, the set of preceding transitions of each transition is determined by dataflow analysis: a transition t' is a preceding transition of a transition t if and only if the execution of t' includes an assignment

Option	Definition	Pros and Cons
HIERARCHICAL	The priority of transition is implicitly determined by the positions of the source and target control states in the state hierarchy.	(+) Simplicity (−) Implicit prioritization
EXPLICIT	Explicit priority is assigned to each transition.	(+) Exhaustive prioritization (−) Tedious to use
NEGATION OF TRIGGERS	A transition is given higher priority than another by strengthening the event trigger of the second transition such that it is not enabled when the first transition is enabled.	(+) Exhaustive prioritization (−) Tedious to use

Table 3.9: Priority Semantic Options [7]

to a variable which is read by t. The DATAFLOW option guarantees that an assignment to a variable happens before it is being read in a big-step. The EXPLICIT and DATAFLOW options can be used to avert undesired nondeterminism by disallowing the execution of the small-steps that do not satisfy the ordering constraints. The EXPLICIT option can be difficult to use because a modeller may introduce an unintended order of transitions. The DATAFLOW semantics can be difficult to use because an unintended cyclic dataflow order might be introduced by the modeller.

3.2.8 Priority

For each small-step, all maximal, consistent sets of transitions are calculated as candidates for execution in the current small-step. The Priority semantic aspect determines which candidate to choose to execute as the current small-step. The comparison of transitions' priorities is transitive. Table 3.9 lists semantic options for assigning a priority to a transition to avert nondeterminism. A set of transitions T has a higher priority than T' if there is a transition in T whose priority is higher than or equal to the priority of every transition in T'.

The HIERARCHICAL option determines the priority of transitions implicitly by the positions of the source and target control states of transitions in the state hierarchy of the model. The semantics of HIERARCHICAL priority is defined by its sub-aspect Basis

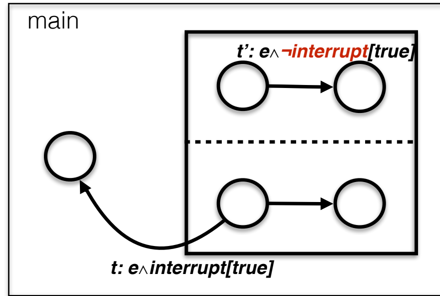


Figure 3.3: Example of assigning priority by Negation of Triggers.

which is one of SOURCE, TARGET, SCOPE, and by its sub-aspect Scheme which is either PARENT or CHILD. For example, our default options SCOPE-PARENT give a higher priority to a transition whose scope is the parent (ancestor) of the scope of the other transition. The EXPLICIT option is appropriate when the modeller is able to syntactically assigning a priority to a transition (e.g., by assigning numbers to transitions). The HIERARCHICAL option imposes no syntactic burden to the modeller but the priority is implicitly decided which might be error-prone, whereas the EXPLICIT option is the opposite. Note that the order of precedence and priority in BSML are both partial orders.

The NEGATION OF TRIGGERS option is not an independent way to assign priority, but uses the notation of negated triggering events to assign priorities. For example, in Figure 3.3 transition t is enabled by $e \wedge interrupt$ and transition t' is enabled by e. When both e and $interrupt$ are triggered, either t or t' can be executed which might be undesirable. In order to assign a higher priority to t, we change the triggering events of t' to be $e \wedge \neg interrupt$, so that t' is enabled only when event $interrupt$ is not present in the set of generated events.

3.2.9 Combo-step Maximality

The Combo-step Maximality semantic aspect specifies the extent of a contiguous segment (i.e., a *combo-step*) of a big-step where computation is carried out based on the statuses of events and values of the variables at the beginning of the segment. Table 3.10 lists semantic options for the Combo-step Maximality semantic aspect. These options are similar to the options for the Big-step Maximality semantics, but specify the scope of a combo-step instead of a big-step.

Option	Definition	Pros and Cons
TAKE MANY	Combo-steps continue until there are no more enabled transitions.	(+) Expressive (−) Non terminating combo-step is possible
TAKE ONE	No two transitions with overlapping arenas can be taken in the same combo-step.	(+) Simple (+) Terminating combo-step is guaranteed (−) Limited
SYNTACTIC	No two transitions with overlapping arenas that enter a designated "combo stable" state can be taken in the same combo-step.	(+) Syntactical scope for combo-step (−) Non terminating combo-step is possible

Table 3.10: Combo-step Maximality Semantic Options [7]

Chapter 4

BSML-mbeddr

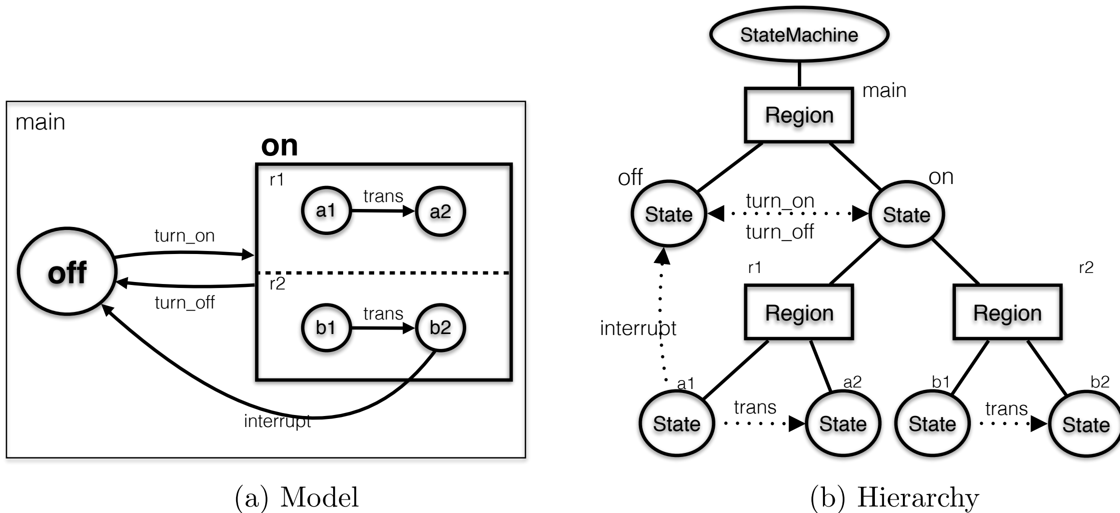
4.1 BSML-mbeddr Syntax

We first give a light introduction to state-machine elements that are used in the following example, whereas details are introduced in Section 4.1.2. A state-machine model may contain state-machine *element*s, including *state*s, *event*s, *region*s and *transition*s. A *state* can be *simple* if it has no internal structure, or *composite* if it contains any sub-*region*. A *region* contains states, events, transitions, and a reference to a contained state to denote its *current state*. A *transition* comprises a source state, a target state, and a triggering event. An *event* may *trigger* a transition, which means when an event is *generated*, a transition whose source state is a current state of the machine's execution and who is triggered by this event can be *executed*. The execution of a transition makes the current state switch from its source state to its target state.

4.1.1 Example-based Demonstration

Shown in Figure 4.1a, our example state-machine model contains a *main* region, within which there are two states – state off and state *on*. States off and *on* can transition to each other by triggering events *turn_on* and $turn_off$, respectively. State off is a simple state with no internal structure, whereas state *on* is a composite state with two concurrent regions $r1$ and $r2$, each of which contains two states as well as a transition triggered by event *trans*.

Figure 4.1b depicts the state hierarchy of the example model, which is formed by interleaved layers of states and regions. The source or target state of a transition may cross



(a) Model

(b) Hierarchy

Figure 4.1: Illustration of an example model and its corresponding state hierarchy.

the boundary of a region (called a *cross-hierarchy transition*), such as the transition from state $b2$ to state off triggered by $interrupt$.

4.1.2 State-machine Elements in BSML-mbeddr

Figure 4.2a illustrates the code for the example model in Figure 4.1. A *state machine* (Line 1) is the root node of a state-machine model; it contains a *main* region. A *region* (Line 2) is a concurrent component of the full state-machine model. It comprises a sub-machine that executes concurrently with other sub-machines; it contains one or more states, and zero or more events, transitions, variables and other utility elements. Each region must designate an *initial state* (Line 2) which is a reference to one of the contained states. A *state* contains zero or more regions: a state containing no region is a *simple* state (Line 17), otherwise it is a *composite* state (Line 18). Such a state hierarchy forms a tree with interleaved layers of states and regions (Figure 4.1b) – simple states are leaves in the hierarchy, whereas regions and composite states are internal nodes in the hierarchy.

Two states (regions) *overlap* if they are the same or one is the ancestor of the other. The *lowest common ancestor* of two states (regions) in the state hierarchy is the lowest node that is an ancestor of both states (regions). Two states (regions) are *orthogonal* if they do not overlap and their least common ancestor is a state, not a region. The *scope* of a transition is the lowest common ancestor of the source and target state. The *arena* of a

```
1  statemachine SM {
2    region main initial = off {
3      in event turn_on();
4      in event turn_off();
5      in event interrupt();
6      in event trans(double arg);
7      event out(string msg) => handle_out;
8      boolean guard = true;
9      double cur_speed = 0.0;
10     static int count_on = 0;
11     Status status = ON;
12     SM instance;
13     transition turn_on[guard] off -> on;
14     transition turn_off[true] on -> off;
15     transition interrupt[true] b2 -> off {
16       out("interrupt"));}
17     state off {  };
18     state on {
19       entry {count_on = count_on+1;}
20       region r1 initial = a1 {
21         state a1 { };
22         state a2 { };
23         transition trans && ¬interrupt[true] a1 ->
               a2 {
24         cur_speed = compute(arg);
25         guard = false;
26         out("normal trans"); }}
27       region r2 initial = b1 { ... }}}}
```

(a) Code of Model

```
28 int main() {
29   SM* m1 = sm_start(SM);
30   trigger_events(m1);
31   sm_trigger(m1, turn_on());
32   SM* m2 = sm_start(sm);
33   SM var = *m2;
34   trigger_events(m2);
35   sm_trigger(&var, trans(2.0), interrupt());
36   sm_terminate(&var);
37   sm_terminate(m1);
38   return 0;
39 }
40
41 void trigger_events(SM* arg) {
42   sm_trigger(arg, turn_on());
43   sm_trigger(arg, trans(2.0));
44   sm_trigger(arg, turn_off());
45 }
46
47 void handle_out(string arg) {
48         printf("%s", arg);
49 }
50 state-machine function:
51 double compute(double speed) {
52   //This function is called in a action to conduct
         computation or query status of environment.
53 }
```

(b) Code of Environment

Figure 4.2: Code for Example Model and Environment

transition is the lowest region that is an ancestor of both the source and target states.

An *event* is defined within a region (Line 3-7). The structure of an event in BSML-mbeddr is similar to that of a function declaration – an event has a name and zero or more arguments. Additionally, an event can have an optional binding to a function that is defined in the environment. There are three types of events: *in-event* and *out-event* determined depending on semantic aspects External Input/Output Event (Section 4.2.5), and *internal-event* that is neither in-event nor out-event. An in-event (Line 3-6) is expected to be triggered from the environment of the state-machine; an out-event (Line 7) is supposed to be bound to a function which is called when the out-event is generated, acted as a way to delivers the state-machine output to its environment; an internal-event is used for private communication inside the model.

A *transition* (Line 13-15) contains a conjunction of triggering events, a guard condition, a source state, a target state, and an optional code block called *action*. A triggering event refers to a visible event declaration, and a *guard condition* is an expression with boolean type. A transition is enabled if each of its trigger is present (or absent if trigger is negated) and the guard condition is evaluated as true. An *action* is a list of statements which is executed when the transition is executed. In an action, the modeller is allowed to manipulate local variables, read arguments of triggering events, call functions, query the

state of the environment, or generate events (Line 24-26).

4.1.3 Language Features

In this section we highlight several language features of BSML-mbeddr. For each language feature, we refer to the line number in Figure 4.2 in which the feature is used.

- **Event with Arguments** An event may have arguments (Line 6) of primitive type (e.g., boolean, int, double) or compound type (e.g., struct, enum, state-machine type). When the event is generated, actual arguments must be provided, and their types must match the declared types. Arguments of a generated event can be used in the guard condition or the action of a transition triggered by the *presence* of the event.

- **Event Binding** An event can be bound to a function that is called when the event is generated and is determined to be an out-event (Line 7). The number and types of arguments in the event and in the bound function must match. The bound function might be an imported library function (e.g., $printf$, $memcpy$, $free$).

- **Negation of Triggers** A transition may be triggered not only by the *presence* of events but also by the *absence* of events; the latter is specified by tagging a triggering event with a negation symbol "¬". For example, the transition on Line 23 is triggered when event *trans* is present and negated event *interrupt* is absent.

- **Transition with Multiple Triggers** A transition may have a conjunction of multiple triggering events, so that the transitions is enabled only if all of its non-negated triggering events are present, and all of its negated triggering events are absent (Line 23).

- **Entry Block** A state or region may optionally contain an entry block[1] of actions that modify values of state-machine variables, call functions, query the state of the environment, or generate events. An entry block is executed every time its associated state or region is entered (Line 19).

- **Cross-hierarchy Transition** We call a transition *local* if its target state has the same parent region as its source state. In other words, a local transition does not cross

[1] We have decided not to implement the exit block in the current version of BSML-mbeddr, considering the complexity to implement it.

the boundary of a region. In addition to local transitions, BSML-mbeddr also handles *cross-hierarchy* transitions that cross the boundaries of regions, including the proper execution of the transitions' actions and entry blocks of states or regions that the transition enters. Specifically, when a transition is executed, all states and regions along the way from the scope of the transition (exclusive) to the target state (inclusive) are entered; before a region is entered, all its sibling regions are entered cascadingly; the target state is entered cascadingly at last.

- **Big-step Start (End) Block** Inspired by the $constructer()$ ($finalizer()$) function of a Java class which is executed at the beginning (end) of the life cycle of a Java object, we have introduced the concept of a *big-step start* (*end*) *block* to BSML-mbeddr to improve integration of state-machine models and their C-code environment. Each state machine may contain an optional big-step start (end) block of statements, which is allowed to modify values of state-machine variables in addition to conduct any operations that are allowed in the environmental code. A big-step start (end) block is executed immediately before a big-step begins (after a big-step ends). Unlike an entry block or an action of a transition, which is regulated by the execution semantics of a big-step, a big-step start (end) block belongs to the environment and its execution takes effect instantaneously.

- **(Static) Variable** A Variable that is defined inside a state-machine is accessible only within the state machine (e.g., inside entry blocks, guard conditions, actions), and variables defined in the environmental code are not accessible inside the state machines. The types of variables can be primitive (e.g., boolean, int, double) or compound (e.g., struct, enum, state-machine type) types. A state-machine variable can be *static*, meaning that it is initialized when the state-machine instance is created, and its value persists as the execution re-enters the state or region where the variable is declared. In contrast, a non-static variable is initialized every time its enclosing state or region is entered. For example, static variable *count_on* (Line 10) is initialized with value 0, and incremented by 1 each time state *on* is entered (Line 19), thus to count the number of times state *on* is entered.

- **Function Call** A function that does not change the status of the environment can be tagged as a *state-machine* $function$ and can be called inside a state machine (e.g., inside a region, action, entry block, or guard condition). For example, such functions can be used to query the current values of variables in the environment, or they can be used as helper functions within more complex computations (Line 24).

- **Name Scoping** For better modularity, each state and region defines a local-name scope

for the contained state-machine elements and variables. This is achieved by assigning a fully qualified name to each state-machine element and variable, and by defining an appropriate search scope for variable references. For example, on Line 19, we can access local variables that are defined in the entry block or in the *main* region, whereas variables defined in other entry blocks, actions and orthogonal regions are not accessible. We may also define a local variable *count_on* in the entry block without conflicting with a similarly named variable *count_on* defined the in *main* region (Line 10).

- **Multiple Instances of State Machine** The modeller is able to create multiple instances of the same state-machine model running concurrently (Line 29 and 32), and may send environmental inputs to each of them without the machines interfering with each other if the user keeps bound functions thread-safe (Line 31 and 35).
- **Input with Multiple Events** An environmental input may contain instances of multiple in-events (Line 35) that simulate a “combo” action (e.g., a passenger of elevator pushes to multiple buttons at the same time). However, an environmental input is not allowed to generate multiple instances of the same in-event.

4.1.4 Interaction with Environment

A state-machine model is defined similarly to a C struct/enum. The definition of a state-machine is surrounded by *environmental code* (or simply the *environment* of the state machine), including definitions of global variables, structs, enums and functions.

Figure 4.2b illustrates a possible case of environmental code that surrounds a state machine. As shown, a local variable (Line 29) is defined with a pointer type pointing to a SM type (the state-machine type defined on Line 1), which is initialized with a $sm_start(sm_ref)$ expression that creates an instance of a SM and returns a pointer to it. A $sm_trigger(sm_handle, event, ...)$ statement (Line 42-44) takes as arguments a pointer to a state-machine type *sm_handle*, and a set of in-events; it is used to generate an environmental input with in-event instances, and put the environmental input into a queue that the state machine is listening to, and where all environmental inputs are sequentialized. A $sm_terminate(sm_handle)$ (Line 36-37) statement safely terminates a state-machine instance after all pending environmental inputs in the input queue are processed.

A variable of a state-machine type is implemented as a first-class citizen, which means it can be assigned to other variables (Line 33), returned from a function, or passed as an argument (Line 34). BSML-mbeddr imposes strict constraints and type-checking rules

which ensure that: 1) variables whose types are different kinds of state-machine types cannot be assigned to each other, nor can they be assigned to variables of non-state-machine types; 2) $sm_start(sm_ref)$ can be assigned only to a variable of a pointer to the same kind of state-machine type as sm_ref; 3) in $sm_trigger(sm_handle, event, ...)$, sm_handle must be a pointer to a state-machine type, and all *event* arguments must be events declared in the state-machine type that sm_handle points to; 4) arguments to in-events, if any, must be provided and their types must match the declared types; 5) sm_handle in $sm_terminate(sm_handle)$ must be a pointer to a state-machine type.

Delivering output from a state machine to its environment is achieved through event binding. For example, event *out* is bound to function $handle_out()$ on Line 7, and $handle_out()$ is called whenever event *out* is generated and determined to be an out-event[2].

In BSML-mbeddr, multiple instances of the same state-machine type can be declared (Line 29 and 32). When sm_start is executed, a thread representing the state-machine instance is launched and an input queue is created. In order to guarantee that multiple state-machine instances can run concurrently without interfering with each other (Section 7.7), it is the user's responsibility to keep functions bound to out-events thread-safe.

4.1.5 State-Region Hierarchy

The state hierarchy of BSML comprises *control states*, including *And* state, *Or* state, and *Simple* state (Section 3.1). In BSML-mbeddr, we have changed the state hierarchy to an state-region alternation structure that is used in existing state-machine modelling languages such as FORML [29]. Essentially, a region in BSML-mbeddr corresponds to an *Or* state (when a region is entered, one of its sub-state is entered), whereas a state in BSML-mbeddr corresponds to an *And* state (when a state is entered, all of its sub-regions are entered). In addition, our change of form imposes two extra constraints on the hierarchy: a) it forms interleaved layers of states and regions so that a state's parent or child must be a region, and vice versa; b) The source state and target state of each transition is only a state, and not a region. This simplifies the implementation of BSML-mbeddr by saving the process of checking whether a child, the parent, or a sibling of a given control state is an *And* state or an *Or* state: 1) given a transition, it is known that its source or target state is a *state* (an *And* state in BSML, correspondingly); and 2) given a region (an *Or* state in BSML, correspondingly), it is known that its parent and children are *state*s, and its siblings are *region*s; and 3) given a state, it is known that its parent

[2]Whether an event is determined to be an out-event (i.e., is to be communicated to the state machine's environment) is specified in the semantic aspect External Output Events, which is discussed in Section 4.2.5.

and children are *regions*, and its siblings are *states*. For example, with our state-region hierarchy, when computing the sequence of entry blocks (Section 4.1.3) to be executed for a transition and a region is to be entered, we are able to know that its parent is a state (*And* state), and its siblings are regions (*Or* states) who should be entered cascadingly before the current region is entered.

4.2 BSML-mbeddr Semantics

In this section, we introduce the semantics of BSML-mbeddr. Basically, BSML-mbeddr implements the semantic aspects and options of BSML as described in Section 3.2, so they are not repeated here. Instead, we describe in this section how our implementation deviates from the semantics of the original BSML. To ease the presentation of descriptions, we denote language syntax in **bold** font, semantic aspects in font Sans Serif, and semantic options in font SMALL CAP. In section 4.2.1 we give an overview on which options are implemented and which options are not implemented in BSML-mbeddr, and we explain why we have decided not to implement semantic options that require dataflow analysis, or that are related to combo-steps or components. In Section 4.2.2, we present the modified version of the Priority semantic aspect, where nondeterminism is resolved by turning priorities among transitions from a partial order into a total order. In Section 4.2.3, we present the modified process of a big-step. Based on the total ordering of priorities among transitions, we are able to compute a valid small-step with drastically reduced time complexity. We also explain how the modified process of a big-step leads to the decision of not implementing options within the Order of Small-steps semantic aspect. Lastly, in Section 4.2.4, we introduce a changed way of implementing the PRESENT IN SAME and NEGATION OF TRIGGERS options due to the potential hazard for the original way of implementation. Semantic aspects and options that are not discussed in this section are consistent with those in the original BSML.

4.2.1 Implemented Semantic Options

The collection of semantic aspects and options in BSML-mbeddr semantics is a subset of the semantic aspects and options defined in the original BSML [7][8][9]. Table 4.1 lists all of the BSML semantic aspects and options and shows with check marks which ones are implemented in BSML-mbeddr. Considering the limit of our effort and the fact that BSML-mbeddr is a proof-of-concept implementation to study configurable semantics, we

did not implement semantic options that requires dataflow analysis, or that are related to combo-steps or components.

We did not implement options that would require dataflow analysis because the dataflow analysis would have been costly in terms of computational complexity and in terms of implementation effort, compared to the knowledge we would gain from their implementation. Affected options include option PRESENT IN WHOLE within aspect Event Lifeline, option DATAFLOW within aspect Order of Small-steps, and option STRONG SYNCHRONOUS within aspects Interface Event Lifeline, Interface Variable in GC, and Interface Variable in RHS.

A *combo-step* defines a coherent sequence of small-steps within a larger big-step, such that a big-step comprises a sequence of combo-steps and a combo-step comprises a sequence of small-steps. There are combo-step semantic options that specify how statuses of events and variable values propagate across combo-steps, including option PRESENT IN NEXT COMBO within aspect Event Lifeline, option GENERATED IN LAST COMBO within aspect External Output Events, option GC COMBO STEP within aspect Internal Variable in GC, and option RHS COMBO STEP within aspect Internal Variable in RHS. In addition, there is a semantic aspect Combo-Step Maximality that defines the termination conditions of a combo-step. In general, the combo-step options are similar to corresponding big-step and small-step options. For example, option GC COMBO STEP within aspect Internal Variable in GC is similar to GC BIG STEP and GC SMALL STEP; and the options for Combo-Step Maximality are similar to the options for Big-Step Maximality. Therefore, we did not believe we would learn much from their implementation, so we left them unimplemented.

Components are used to decompose a state-machine model into encapsulated submachines that communicate with each other only through interface events or variables. Communication among components through interface events or variables is similar to communication among the regions of a machine through internal events or variables. For example, an internal event (or, correspondingly, an interface event) generated in one region (component) can trigger transitions in another region (component), whose presence is regulated by Internal Event Lifeline (Interface Event Lifeline). Although components add another abstraction layer of encapsulation which allows distinct semantic options to be selected for inter-component communication, we did not believe that we would learn much from their implementation. Affected semantic options include all options for aspects Interface Events, GC Memory Protocol: Interface Variables, and RHS Memory Protocol: Interface Variables.

Semantic Options		Semantic Options	
Big-step Maximality		Interface Event Lifeline	
Take One	✓	Strong Synchronous Event	
Take Many	✓	Weak Synchronous Event	
Syntactic	✓	Asynchronous Event	
Concurrency		GC Memory Protocol: Internal Variables	
Single	✓	GC Big Step	✓
Many	✓	GC Small Step	✓
Small-step Consistency		GC Combo Step	
Source-Target Orthogonal	✓	GC Memory Protocol: Interface Variables	
Arena Orthogonal	✓	GC Strong Synchronous	
Preemption		GC Weak Synchronous	
Non-Preemptive	✓	GC Asynchronous	
Preemptive	✓	RHS Memory Protocol: Internal Variables	
Internal Event Lifeline		RHS Big Step	✓
Present In Whole		RHS Small Step	✓
Present In Remainder	✓	RHS Combo Step	
Present In Next Combo		RHS Memory Protocol: Interface Variables	
Present In Next Small	✓	RHS Strong Synchronous	
Present In Same	✓	RHS Weak Synchronous	
External Input Events		RHS Asynchronous	
Syntactic	✓	Order of Small-steps	
Received In First Small	✓	None	
Hybrid	✓	Explicit	
Input Event Lifeline		Dataflow	
same as Internal Event Lifeline	✓	Priority	
External Output Events		Explicit	✓
Syntactic	✓	Hierarchical	✓
Generated In Last Combo		Negation of Triggers	✓
Generated In Last Small	✓	Combo-step Maximality	
Hybrid	✓	Combo Syntactic	
Output Event Lifeline		Combo Take One	
same as Internal Event Lifeline	✓	Combo Take Many	

Table 4.1: BSML Semantic Aspects/Options. Semantic options implemented in BSML-mbeddr are indicated with check marks.

4.2.2 Priority

The Priority semantic aspect specifies the relative priorities among transitions. The priority in BSML-mbeddr deviates from BSML in a way that the partial priority order of transitions in BSML is resolved to a total order, so that nondeterminism is resolved.

Option EXPLICIT in BSML-mbeddr is appropriate when the modeller is able to indicate priority by syntactically assigning a positive integer to a transition. An unannotated transition effectively has a priority value of infinite, indicating the lowest priority. If two transitions have the same priority, we use the textual order of their declarations to resolve nondeterminism – that is, the transition that is declared earlier in the mbeddr program has higher priority.

Option HIERARCHICAL in BSML-mbeddr determines the priority of transitions implicitly by the state hierarchy of the model. The semantics of HIERARCHICAL priority is defined by its sub-aspect Basis which is one of SOURCE, TARGET, SCOPE, and by its sub-aspect Scheme which is either PARENT or CHILD. For example, our default options SCOPE-PARENT give a higher priority to a transition whose scope is the parent (ancestor) of the scope of the other transition. If the Basis (which is source, target, or scope) of one transition is neither a descendant nor an ancestor of the Basis of another transition, we resolve the nondeterminism as follows: a) if the transitions' Basis states (or regions) are siblings, we prioritize the transitions according to the textual order in which their Basis states (or regions) are declared – the transition whose Basis is declared first has higher priority; b) otherwise, we compute the lowest common ancestor of the transitions' Basis; and then we prioritize the transitions according to the textual order in which the ancestors of their Basis states (or regions) (i.e., the child of the lowest common ancestor) are declared.

For example, in Figure 4.3 the scopes of t_1, t_2, and t_3 are a, $r1$ and b, respectively. Let's assume that the semantic aspect Priority is HIERARCHICAL and its sub-aspects are SCOPE and PARENT, and let's assume that a node's left child in the state hierarchy is textually declared earlier than its right child; then 1) t_1 has higher priority than t_2 because the scope of t_1 is an ancestor of the scope of t_2; and 2) t_1 has higher priority than t_3 because the scopes of t_1 and t_3 are siblings, and the scope of t_1 is declared earlier than the scope of t_3; and 3) t_2 has higher priority than t_3 because a, which is the ancestor of the scope of t_2 that is a sibling of the scope of t_3, is declared earlier than the scope of t_3 (which is b).

The technique to specify priorities among transitions by negation of triggers still exists in BSML-mbeddr, but is not implemented as an semantic option. This is explained in Section 4.2.4.

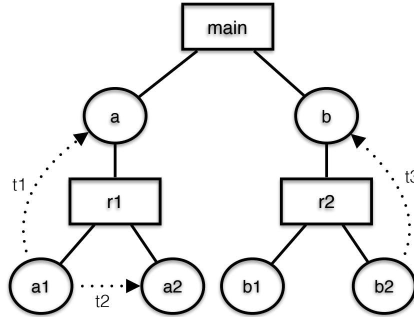


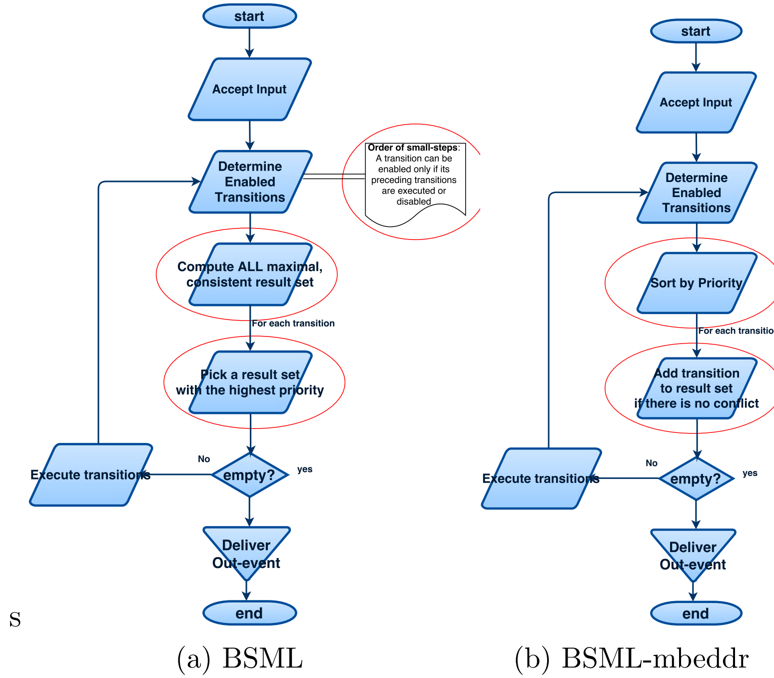
Figure 4.3: Illustration of HIERARCHICAL Priority.

4.2.3 Modified Execution Semantics

The execution semantics describes how a state-machine model handles an environmental input, responds by executing transitions, and communicates its outputs. As shown in Figure 4.4b, in BSML-mbeddr, a big-step starts by accepting an environmental input comprising a list of in-event instances and activating the in-events so that they are sensed as being *present*. Then a small-step is started by identifying enabled transitions. Next, the enabled transitions are sorted according to their priority, and a maximal, consistent subset of enabled transitions (called *result set*) with highest priority is deduced through a greedy approach. A small-step ends by executing all transitions in the result set of the current small-step and by calculating the new status of the state machine. At the end of a small-step, if the result set is not empty then a new small-step starts repeatedly where the previous result set is emptied and a new result set is calculated, otherwise the big-step ends and outputs of the big-step are delivered to the environment.

Figure 4.4 compares the process of a big-step in BSML with the process for a big-step in BSML-mbeddr. As shown in Figure 4.4a, BSML deduces all maximal, consistent subsets (*result sets*) of the set of enabled transitions, of which one with the highest priority is picked to execute as the current small-step. The aspect Order of Small-steps is introduced in BSML to reduce the number of enabled transitions, and thus the number of result sets within a small-step, thereby increasing the understandability of the model. The aspect Order of small-steps specifies a precedence relationship among transitions, which means that each transition has a set of *preceding* transitions. A transition is enabled only if each of its preceding transitions either is disabled or has already been executed in the current big-step. Note that the order of precedence and priority in BSML are both partial orders.

In contrast, in BSML-mbeddr, nondeterminism in deciding which enabled transitions are included in the result set is resolved by the textual order in which model elements are



(a) BSML (b) BSML-mbeddr

Figure 4.4: Comparison of flowgraph of BSML and BSML-mbeddr.

declared, so that priority is a total order (Section 4.2.2). As shown in Figure 4.4b, BSML-mbeddr constructs a single result set of the highest-priority enabled transitions using a greedy process: firstly, enabled transitions are sorted by their priority, and an empty result set is initialized; then each enabled transition, in decreasing order of priority, is considered for inclusion in the result set – if the result set with the transition included remains consistent, then the transition is added to the result set; otherwise it is not included. Given an arbitrary set of enabled transitions, the result set constructed by BSML-mbeddr is guaranteed to be one of the result sets with highest priority constructed by the BSML big-step process. The proof is as follows:

Definition 4.2.1. For two transitions t_1 and t_2 in the same state-machine model, we say that $t_1 <_m t_2$ if t_1 has higher priority than t_2 in BSML-mbeddr; similarly, we say that $t_1 <_b t_2$ if t_1 has higher priority than t_2 in BSML. A transition can never have a higher priority than itself.

Lemma 1. *$<_m$ is a topological order of $<_b$. That is, for any two transitions t_1 and t_2, $t_1 <_b t_2 \Rightarrow t_1 <_m t_2$.*

Proof. We prove it for semantic options Explicit and Hierarchical respectively. For

EXPLICIT, $t_1 <_b t_2$ means that t_1 is assigned a smaller integer than t_2, which in BSML-mbeddr also indicates t_1 has higher priority than t_2 (Section 4.2.2), thus, $t_1 <_m t_2$. For HIERARCHICAL, $t_1 <_b t_2$ means that t_1 is an ancestor or descendant node of t_2, depending on whether the sub-aspect Scheme is PARENT or CHILD. In BSML-mbeddr t_1 being an ancestor or descendant node of t_2 also indicates t_1 has higher priority than t_2 (Section 4.2.2), thus, $t_1 <_m t_2$. ■

Definition 4.2.2. Predicate $C_r(t)$ is true if and only if t can be executed according to the semantic aspect Big-step Maximality. Commutative predicate $C_s(t_1, t_2)$ is true if and only if t_1 and t_2 do not conflict with each other according to semantic aspects Concurrency, Consistency, and Preemption. The *consistency checking criteria* $C(T)$ checks whether a set of transitions T is *consistent*, which is defined as: $C(T) \Leftrightarrow (\forall t \in T : C_r(t)) \wedge (\forall t_1, t_2 \in T : t_1 \neq t_2 \Rightarrow C_s(t_1, t_2))$.

Lemma 2. *Given a transition t, a set of transitions T, and consistency checking criteria C, it holds that $C(T) \wedge C_r(t) \wedge (\forall t' \in T : C_s(t, t')) \Rightarrow C(T \cup \{t\})$.*

Proof. To prove $C(T \cup \{t\})$ holds, let us prove that (1) $\forall t' \in T \cup \{t\} : C_r(t')$, and (2) $\forall t_1, t_2 \in T \cup \{t\} : C_s(t_1, t_2)$, according to Definition 4.2.2.

(1) According to Definition 4.2.2, $C(T) \Rightarrow (\forall t' \in T : C_r(t'))$. Then $C_r(t) \wedge (\forall t' \in T : C_r(t')) \Rightarrow (\forall t' \in T \cup \{t\} : C_r(t'))$. (2) According to Definition 4.2.2, $C(T) \Rightarrow (\forall t_1, t_2 \in T : t_1 \neq t_2 \Rightarrow C_s(t_1, t_2))$. Then $\forall t_1, t_2 \in T \cup \{t\} : t_1 \neq t_2 \Rightarrow C_s(t_1, t_2))$ holds if $t_1 \neq t$ and $t_2 \neq t$. Next, let us consider the situation that $t_1 = t, t_2 \neq t$ (or $t_2 = t, t_1 \neq t$, equivalently): $\forall t_2 \in T : C_s(t, t_2)$ holds by assumption. Therefore, $C(T \cup \{t\})$ holds. ■

Lemma 3. *Given a set of transitions T, and consistency checking criteria C, it holds that $T' \subseteq T : C(T) \Rightarrow C(T')$.*

Proof. According to Definition 4.2.2, $\forall T' \subseteq T : C(T) \Rightarrow (\forall t \in T : C_r(t)) \wedge (\forall t_1, t_2 \in T, t_1 \neq t_2 : C_s(t_1, t_2)) \Rightarrow (\forall t \in T' : C_r(t)) \wedge (\forall t_1, t_2 \in T', t_1 \neq t_2 : C_s(t_1, t_2)) \Rightarrow C(T')$. ■

Definition 4.2.3. For two sets of transitions T_1 and T_2, we call $T_1 \lessdot T_2$ if T_1 has higher priority than T_2 in BSML. T_1 has higher priority than T_2 if $\exists t_1 \in T_1, \forall t_2 \in T_2 : t_1 <_b t_2$ (Section 3.2.8).

Theorem 4. *Given a set of enabled transitions T and consistency checking criteria C, let $R_m \subseteq T$ be the result set constructed by BSML-mbeddr; let $\mathbb{R}$ be the set of result sets constructed by BSML, where $\forall R \in \mathbb{R} : R \subseteq T$. It holds that (1) $R_m \in \mathbb{R}$ and (2) $\forall R \in \mathbb{R} : \neg(R \lessdot R_m)$.*

Proof. (1) With the same consistency checking criteria and the same set of transitions, BSML constructs all maximal, consistent result sets $\mathbb{R}$ while BSML-mbeddr constructs a single consistent result set R_m by a greedy process. First let us prove by contradiction that R_m is maximal. Assume that $\exists R'_m \subseteq T : R_m \subset R'_m$. Then $\exists t' : t' \in R'_m \wedge t' \notin R_m$. According to Definition 4.2.2, $(t' \in R'_m) \wedge C(R'_m) \Rightarrow C_r(t') \wedge (\forall t'' \in R'_m : C_s(t', t'') \Rightarrow C_r(t') \wedge (\forall t'' \in R_m : C_s(t', t'')$. Then according to Lemma 2, $C(R_m) \wedge C_r(t') \wedge (\forall t'' \in R_m : C_s(t', t'') \Rightarrow C(R_m \cup \{t'\})$. Then according to Lemma 3, $\forall R''_m \subseteq R_m : C(R''_m \cup \{t'\})$. This contradicts with the fact that when t' is considered for inclusion in a subset of R_m, it is not chosen to be included. Thus R_m is maximal. Therefore, $R_m \in \mathbb{R}$.

(2) Assume that $\exists R_b \in \mathbb{R}$ such that $R_b \lessdot R_m$. Then $\exists t_b \in R_b, \forall t \in R_m : t_b <_b t$ according to Definition 4.2.3. According to Lemma 1, $\forall t \in R_m : t_b <_m t$. Because BSML-mbeddr considers transitions for inclusion in the result set in order of decreasing priority, t_b would be considered before any transition t in R_m. Thus, the result set would be empty when t_b is considered for inclusion. According to Definition 4.2.2, $(t_b \in R_b) \wedge C(R_b) \Rightarrow C_r(t_b)$, which further indicates that $t_b \in R_b$ would be added to the empty result set when considered for inclusion, i.e., $t_b \in R_m$. This contradicts the previous proposition $\forall t \in R_m : t_b <_m t$ because a transition can never have a higher priority than itself (Definition 4.2.1). Therefore, $\forall R \in \mathbb{R} : \neg(R \lessdot R_m)$ holds. ■

BSML is a theoretical work where all possible maximal, consistent sets of enabled transitions should be considered and explored. In particular, analysis of a state-machine model should examine all possible result sets since all are acceptable responses to the environmental input. However, BSML-mbeddr is a concrete implementation of BSML for the purpose of execution. It is sufficient for BSML-mbeddr to construct a single acceptable response to the environmental input. Given that all of the transition sets in BSML's result sets are equally acceptable, BSML-mbeddr constructs just one – using the textual order of definitions to choose among equally acceptable choices. Our resolution strategy helps to ensure that the modeller can predict what the result set will be and can use definition order to effect some control on the result set. BSML-mbeddr does not employ the Order of Small-steps aspect in its big-step process because after introducing a total priority order, there is no obvious need to introduce Order of Small-steps to impose a partial precedence order on the enabled transitions. Moreover, a particular Order of Small-steps might cause confusion for the modeller. For example, if the modeller explicitly assigns a higher priority to a transition than its preceding transitions, and if the transition's triggering events are present and its guard condition is true, the transition still cannot be enabled if any of its preceding transitions are enabled, because Order of Small-steps applies before Priority. This scenario might be counter-intuitive and unwanted for the modeller.

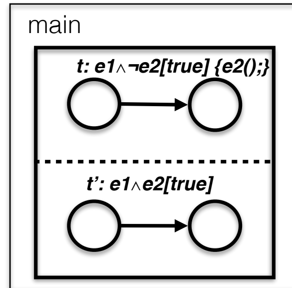


Figure 4.5: Example model demonstrating unintended behaviour when options PRESENT IN SAME and NEGATION OF TRIGGERS are selected together.

4.2.4 Present in Same and Negation of Triggers

In BSML, option PRESENT IN SAME within aspect Event Lifeline indicates that any generated event is present only in the small-step in which it is generated, so that it can enable other transitions in the same small-step. The option NEGATION OF TRIGGERS within aspect Priority allows transitions to be triggered by the absence of events. However, we find that the two options, when selected together, may cause confusion in a scenario where multiple transitions that are triggered by either the presence or the absence of the same event are enabled in the same small-step. For example, in Figure 4.5 if event e_1 is present, event e_2 is absent, and PRESENT IN SAME is chosen, then transition t is executed that generates e_2, which further triggers t' in the same small-step. This results in an unintended scenario where t and t', triggered by the presence and the absence of e_2, respectively, being enabled and executed in the same small-step. In BSML-mbeddr, PRESENT IN SAME and NEGATION OF TRIGGERS are supported, but not as semantic options. Trigger by the absence of an event is a language feature that is always enabled. Similarly, PRESENT IN SAME is a language feature that is always enabled but only for a special type of event called *rendezvous* event, which is syntactically specified. Thus, PRESENT IN SAME applies automatically when a transition has a rendezvous event as its triggering event.

4.2.5 External Event

Our implementation of semantic options within the External Event semantic aspect is consistent with those in BSML (Section 3.2.4). However, BSML does not specify the concrete notation to syntactically determine in-events and out-events. In BSML-mbeddr, the modeller is able to tag an event as **in**' event, whereas any event with event binding is syntacti-

cally determined to be an out-event. If option SYNTACTIC is chosen for the External Input Events semantic aspect, only events that are syntactically determined to be in-events can be triggered from the environment. For a generated event that is bound to a function and determined to be an out-event, the resulting function call (called *event-binding call*) is executed at the end of a big-step; the event-binding call of an event that is not determined to be an out-event is ignored.

4.2.6 Granularity of Semantic Configuration

It is not specified in BSML whether the execution semantics shall be configured per group of state-machine types, per state-machine type, or per state-machine instance. In BSML-mbeddr, a *semantic configuration* is associated to a *mbeddr program*, which may contain the definition and usage of multiple state-machine types whose semantics follow the same semantic configuration. It is theoretically possible to assign the semantic configuration per state-machine type or per state-machine instance. However, mbeddr support only one instance of such a *configuration* (from language module *mbeddr.buildconfig*) per mbeddr program.

Chapter 5

Implementation

In this chapter we discuss the implementation of BSML-mbeddr. BSML is implemented by defining each language aspect within mbeddr, the background of which is introduced in Chapter 2. In Section 5.1, we present the implementation of the BSML-mbeddr syntax. We first discuss the interface and concrete concepts of the *structure* aspect, and then we briefly talk about the usage of the other aspects including *constraint*, *editor*, *type system* and *behaviour*[1]. In Section 5.2, we present the implementation of the BSML-mbeddr semantics. We first illustrate by example the layout of the generated code, and then we briefly introduce the template-based *generator* aspect that transforms the source model into C-code fragments.

5.1 Syntax Implementation

5.1.1 Interface Concepts

We have defined a set of interface concepts for expressing the abstract behaviour and common properties of state-machine *elements*. This approach separates the abstract definitions of behaviours from their concrete implementations, so that the language definition is modular and reusable.

Figure 5.1 shows the interface hierarchy of BSML-mbeddr (mbeddr's built-in concepts are highlighted in color). The most basic interface concept is **ISMElement**, which rep-

[1]IDE-related aspects such as *intention* and *action* are not discussed in this thesis but their source code can be found in our github repository `https://github.com/z9luo/BSML-mbeddr`

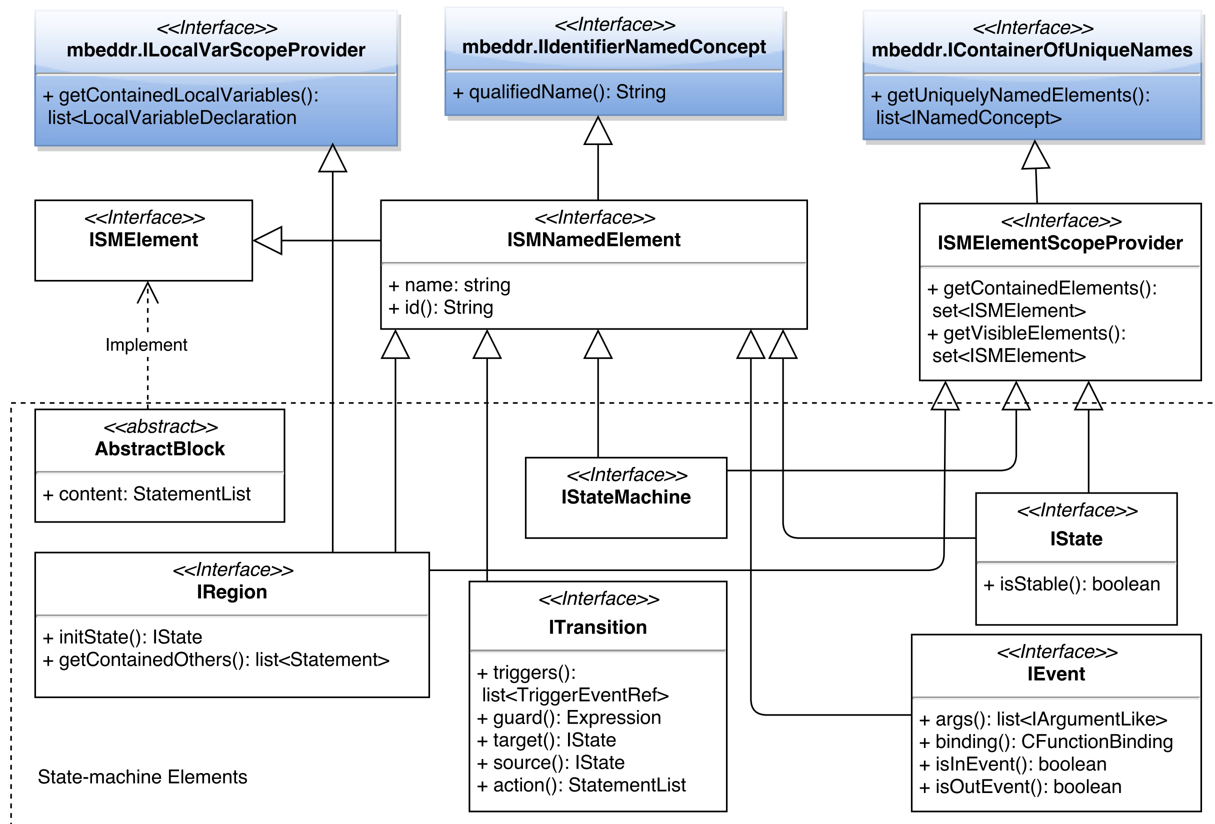


Figure 5.1: BSML-mbeddr Interface Hierarchy. mbeddr's built-in concepts are highlighted in color.

resents all state-machine elements. **ISMNamedElement** extends **ISMElement** and **IIDentifierNamedConcept**, to represent all named state-machine elements. It inherits method $qualifiedName()$ from **IIdentifierNamedConcept** so that every state-machine element is assigned a globally unique name (Figure 5.2a). Method $id()$ in **ISMNamedElement** transforms a unique name in the model into a legal C variable name (Figure 5.2b). **IStateMachine**, **IRegion**, **IState**, **ITransition** and **IEvent** all extend **ISMNamedElement**; whereas **AbstractBlock** implements **ISMElement** because entry blocks are not named elements.

IRegion, **IStateMachine** and **IState** all extend **ISMElementScopeProvider** (Figure 5.2c), denoting that they each provide a container of **ISMElement** (retrievable through calls to method $getContainedElements()$), and a local scope for the contained elements (retrievable through calls to method $getVisibleElements()$). **ISMElementScopeProvider**

```
public virtual string qualifiedName() {
  node<IIdentifierNamedConcept> anc = this.ancestor<concept = IIdentifierNamedConcept>;
  if (anc != null) { return anc.qualifiedName() + "." + this.name; }
  return this.name;
}
```

(a) **IIdentifierNamedConcept**, $qualifiedName()$.

```
public virtual string id() {
  string full_name = qualifiedName();
  return full_name.replaceAll("\\.", "__");
}
```

(b) **ISMNamedConcept**, $id()$.

```
public virtual abstract sequence<node<ISMElement>> getContainedElements();

public virtual sequence<node<ISMElement>> getVisibleElements() {
  if (this.ancestor<concept = ISMElementScopeProvider>.isNotNull) {
    return getContainedElements().union(this.ancestor<concept = ISMElementScopeProvider>.getVisibleElements());
  }
  return getContainedElements();
}
```

(c) **ISMElementScopeProvider**, $getContainedElements()$ and $getVisibleElements()$

Figure 5.2: Example Behaviours of Interface Concepts

extends **IContainerOfUniqueNames**, so that conflicts among global names are detected automatically. **IRegion** extends **ILocalVarScopeProvider** indicating that local variables can be defined within a region.

5.1.2 Concrete Concepts

Figure 5.3 shows the structure of BSML-mbeddr concrete concepts as well as the extension points where a state-machine model is integrated into the mbeddr program (mbeddr's built-in concepts are highlighted in color). The root node of the mbeddr program is **ImplementationModule**, which contains a list of **IModuleContent**. **GlobalVariableDeclaration** and **Function** implement interface **IModuleContent**, so that they can be created under **ImplementationModule** to declare functions and global variables. A **Function** node contains a **StatementList** as its function body. Finally, **LocalVariableDeclaration** extends **Statement**, which provides additional properties such as name and type for a local variable.

As for the extension points, **SMGlobalDeclaration**, which is the root node of a state machine, implements **IModuleContent**, so that it can be created along with other global variables, functions, enum and struct types in the mbeddr program. **RegionLocalDeclaration** and **StateLocalDeclaration** both extend **LocalVariableDeclaration**, whereas

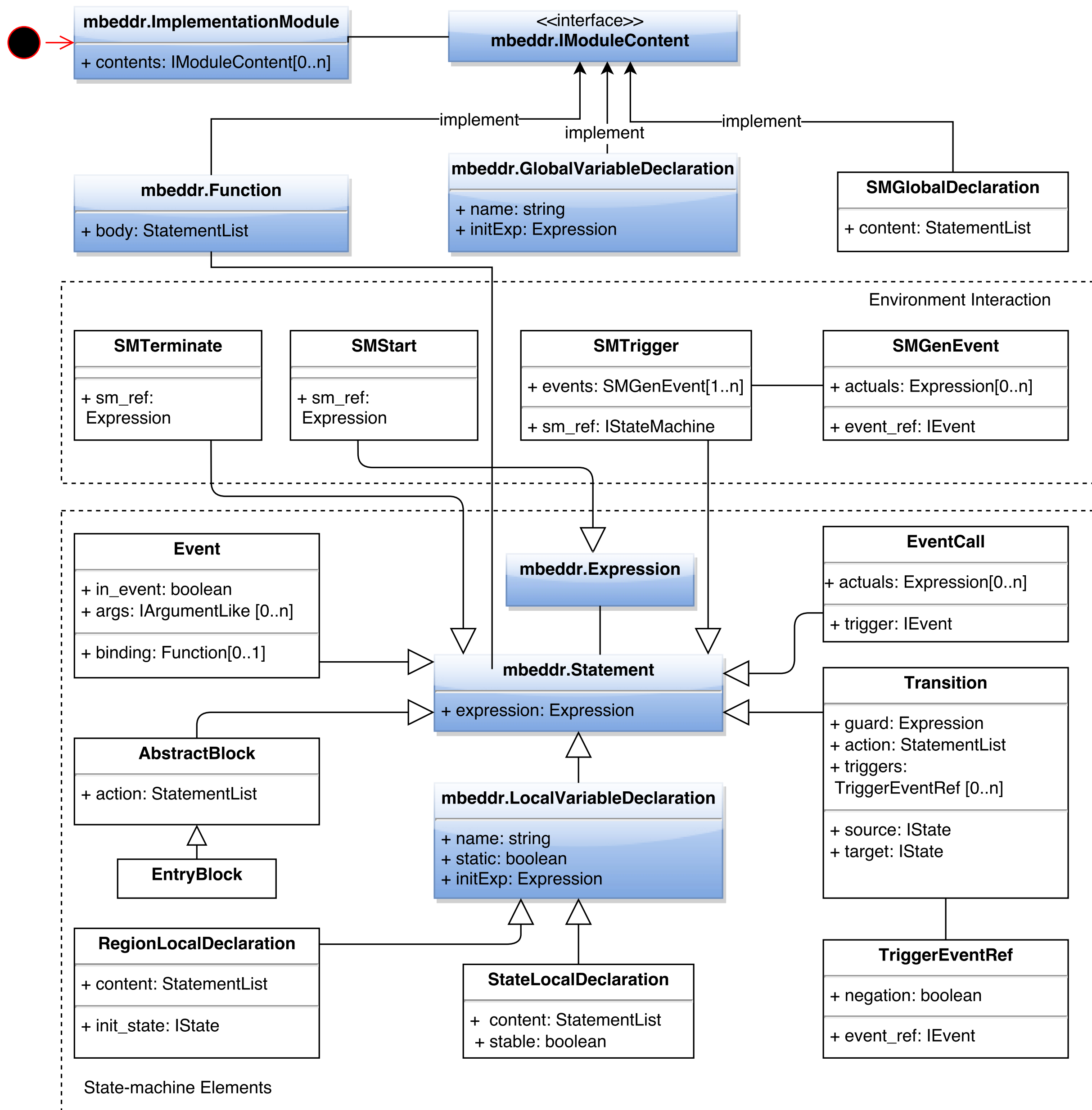


Figure 5.3: BSML Syntax. mbeddr's built-in concepts are highlighted in color.

```
concept Transition extends     Statement
                   implements ITransition

  instance can be root: false
  alias: transition
  short description: <no short description>

  properties:
  << ... >>

  children:
  guard    : Expression[0..1]
  action   : StatementList[0..1]
  triggers : TriggerEventReference[0..n]

  references:
  source : IState[1]
  target : IState[1]
```

Figure 5.4: Structure Example, **Transition**.

the other state-machine elements extend Statement. Each state-machine element container (state machine, state, region) contains a **StatementList**, so that state-machine elements can be defined within it. In the *constraint* aspect, we specify the types of state-machine elements that can be contained in each container.

From Figure 5.1 and Figure 5.3, we highlight the points where language features in Section 4.1.3 are implemented:

- **ISMNamedElement** provides qualified global names and **ISMElementScopeProvider** provides the local scope for a state-machine element (Name Scoping).

- **RegionLocalDeclaration** implements **ILocalVarScopeProvider** that may contain **LocalVariableDeclaration**; **AbstractBlock** and the *action* attribute of **Transition** both contain **StatementList** where **LocalVariableDeclaration** can be referred (Variables).

- **SMTrigger** and **SMTerminate** both extend **Statement**, whereas **SMStart** extends **Expression**. They are used in the environmental code to start and terminate a state-machine instance, and to trigger environmental inputs (Interaction with Environment, Section 4.1.4).

- Event contains a list of **IArgumentLike** (Event with Arguments) and an optional reference to a Function (Event Binding); it has a tag to denote whether the event is used locally or externally (External Event).

- Because **FunctionCall** is a subtype of **Statement**, a function call can be made within an action or an entry block (Function Call).

- **EntryBlock** extends **AbstractBlock**, which extends **Statement**. It allows an entry block that contains a **StatementList** to be defined within a region or state (Entry Block).
- **Transition** contains a list of **TriggerEventRef** (Multiple Triggers). Each **TriggerEventRef** contains a reference to an **IEvent** and a boolean variable indicating whether the trigger is negated (Negation of Triggers).

5.1.3 Other Language Aspects

In this section, we briefly discuss the roles of the language aspects of *editor*, *constraint*, *type system* and *behaviour* in BSML-mbeddr. More detailed descriptions of these can be found in Appendix A.

- **Editor** It defines how the abstract syntax of a model is projected to concrete representations. Additionally, it plays as a key role in hiding and recovering syntax when certain syntax needs to be disabled or re-enabled when the semantic configuration is changed by the modeller. For example, the **stable** tags on states are visible only when Big-step Maximality is SYNTACTIC.
- **Constraint** It is used to achieve two goals: a) restrict which node can be a parent/child/ancestor of another node, and b) specify the search scope of a reference node, if it has any. For example, we impose constraints on **RegionLocalDeclaration** that a) restrict the types of containing statements to **LocalVariableDeclaration**, **IEvent**, **ITransition**, **IState** and **AbstractBlock**; and b) define the search scope of its initial state to be all the contained states within the region.
- **Type System** It is used to derive or check types of concepts. A state-machine model is defined similarly to a C struct, so that variables can be of a state-machine type. We provide type derivation rules to make sure that variables of a state-machine type are resolved correctly. We also provides type-checking rules that check whether in event binding the declared types of arguments in the event match those in the bound function, and that check whether there is a name conflict in an element container, and so on.
- **Behaviour** It is used to define abstract methods in an interface to abstract away the implementation detail, or to define concrete methods in a concrete concept; a concrete method may implement an abstract method or override another concrete method. For

example, interface **ICallLike** defines several abstract methods in its behaviour, and any concrete concept that implements **ICallLike** as well as its abstract methods gains the benefit of argument type checking automatically.

5.2 Semantics Implementation

5.2.1 Code Layout

```
1  enum SM_StateEnum{
2    sm_main_off,
3    sm_main_on,
4    sm_main_on_r1_a1,
5    sm_main_on_r1_a2,
6    sm_main_on_r2_b1,
7    sm_main_on_r2_b2
8  };
9  enum SM_EventEnum {...};
10 enum SM_RegionEnum {...};
11 enum SM_TransEnum {...};
12 struct SM_SMStruct {
13   SM_StateEnum sm_main_cur_state;
14   SM_StateEnum sm_main_on_r1_cur_state;
15   SM_StateEnum sm_main_on_r2_cur_state;
16   //Collecting event bindings to functions,
17   //and execution at the end of a big-step.
18   GPtrArray* bindings;//Array of BindingCall*
19   //Other local variables
20   boolean sm_main_guard;
21   int sm_main_countOff;
22   Status sm_main_status;
23   ...
24 };
25 struct BindingCall {
26   void (*func)(void** args);
27   void** args;
28 };
```

(a)

```
29 struct SM_Transition {
30   SM_TransEnum trans_enum;
31   SM_StateEnum*[] _cur_states;
32   SM_StateEnum[] new_cur_state_value;
33   ActionRef action_ref;
34   BlockRef[] entry_refs;
35   SM_RegionEnum arena;
36   bool enter_stable_state;
37   int priority;
38   boolean is_interrupted
39   ...
40 };
41 void action_main_on_r1_t1(...) {
42 //body of transition action.
43 }
44 void on_entry_main_on_r1_a1(...) {
45 //re-initialize non-static variables; and execute
        entry block.
46 }
47 struct Event {
48   uint32_t type;
49   void** args;
50 };
51 typedef GPtrArray as EnvInput;//Array of Event*
52 struct SMHandle {
53   GThread* instance;
54   GAsyncQueue* queue;
55 };
```

(b)

Figure 5.5: Example of Structural Code Layout

Figure 5.5 and Figure 5.6 illustrate by way of an example the layout of the structural and behavioural code, that is generated from the state-machine model as well as the environmental code in Figure 4.2.

Figure 5.5 shows snippets of the structural code that is generated for the state-machine model in Figure 4.2a. An enum type SM_StateEnum (Line 1) is generated for the state-machine type SM that lists all the state names as enum values. Similarly, enum types are generated for the names of regions, events and transitions (Line 9-11). A struct SM_SMStruct (Line 12) is generated for SM that stores all of the run-time information for the state-machine instance. State machine SM can have multiple instances running

concurrently, each of which has its own SM_SMStruct instance. A state-machine's run-time information includes the current state of each region (Line 13-15), the values of state-machine variables (Line 20-22), and an array of function calls of events bindings for generated out-events (i.e., BindingCall) (Line 18). Struct BindingCall (Line 25) contains a pointer to the bound function and a pointer to the actual arguments. A struct SM_Transition (Line 29) is generated for SM to store information about a declared transition and for holding some run-time information. Specifically, a Transition struct instance records a transition's enum value, its priority, a pointer to its action, a sequence of entry blocks and some run-time information such as the memory addresses of the *cur_state* variables that the execution of the transition will affect as well as the new *cur_state* values. Each action and entry block is transformed into a function (Line 41 and 44). A struct Event (Line 47) that is generated once in an mbeddr program, contains an integer denoting its type and a pointer to its arguments, storing a generated event instance; an environmental input (i.e., EnvInput, Line 51) struct that is generated once in an mbeddr program, is a list of Event instances. An instance of SMHandle (Line 52), containing a pointer to a thread and an input queue, is generated for each state-machine instance. The SMHandle variables are used in the environmental code to start and terminate the corresponding state-machine instance, or to generate an environmental input and put it into the input queue.

Figure 5.6a shows snippets of the behavioural code that is generated for the state-machine model in Figure 4.2a. A state-machine instance is launched by executing function *sm_start*() (Line 1). *sm_start*() first instantiates a SMStruct to store all of the run-time information associated with the instance (Line 2-3). Then it keeps listening on the input queue that was passed in by argument, retrieving environmental inputs and calling *execute_big_step*() for each environmental input (Line 4-9). In a big-step (Line 10-31), small-steps are executed in a while-loop (Line 15-27) until no more transitions can be executed. During a small-step, enabled transitions are identified, and a maximal, consistent result set is derived; transitions in the result set are executed and event-binding calls are collected. A big-step ends when no more small-steps can be executed. Afterwards, event-binding calls of out-events are executed (Line 29-31). The predicate *is_consistent*() is used within the execution of a big-step to check whether a given transition can be added to the result set without violating any semantic criteria (Line 32-35).

Figure 5.6b shows snippets of the environmental code that is generated for the source environmental code in Figure 4.2b, which helps illustrate the translation for **SMStart**, **SMTerminate** and **SMTrigger**. To start a state-machine instance, a SMHandle struct is instantiated (Line 3-8) with an input queue and a thread executing *sm_start*(). To trigger an environmental input (Line 20-32), first the actual arguments of the generated events are wrapped in a pointer array; then the event instances are created and stored in an EnvInput

```
1  void sm_start(GAsyncQueue* in_queue) {
2    SMStruct snapshot_big;
3    init_snapshot(&snapshot_big);
4    while(true) {
5    EnvInput* in=g_async_queue_pop(in_queue));
6    for (Event* e : in)
7      snapshot_big.present_events[e->type]=e;
8    execute_big_step(&snapshot_big);
9  }}
10 void execute_big_step(SMStruct* snapshot_big) {
11   SMStruct* snapshot_small=copy(snapshot_big);
12   SMStruct* snapshot_cur=copy(snapshot_big);
13   Transition** enabled_transitions;
14   //small-step
15   do {
16   nested switch-cases.//identify enabled
         transitions
17   //calculate the result set
18   sort(enabled_transitions);//according to priority
19   result_set={};
20   for (Transition trans : enabled_transitions)
21     if(is_concsistent(result_set, trans, snapshot)
22           result.add(trans);
23   //execute transitions and
24   //collect binding of generated out-events
25   for (Transition trans : result_set)
26     handle_transition(trans, snapshot->bindings);
27   } while (result_set is not empty)
28   //end big-step
29   for(BindingCall* call in snapshot->bindings)
30     call->func(call->args);
31 }
32 boolean is_consistent(Transition** result_set,
         Transition* trans, SMStruct* snapshot) {
33   \\check whether adding trans in result_set will
         results in a consistent result_set
34   ...\\checking against each semantic aspect
35 }
```

(a) State-machine Behavioural Code

```
1  int main(int argc, char* argv[]) {
2    SMHandle* m1;
3    {  //generated code for \concept{SMStart}
4    SMHandle* ret=(SMHandle*)(malloc(sizeof(SMHandle)
         );
5    ret->queue=g_async_queue_new();
6    ret->instance=g_thread_new(&sm_start, ret->queue)
         ;
7    m1=ret;
8    }
9    trigger_events(m1);
10   { //generated code for SMTerminate
11     SMHandle* cur=m1;
12     ... //send a "terminate" event
13     //join its thread and free memory
14     g_thread_join(cur->instance);
15     g_async_queue_unref(cur->queue);
16     free(cur);
17   }
18 }
19 void trigger_events(SMHandle* m1) {
20   {  //generated code for SMTrigger
21   //wrap actual arguments
22   void** args_0 = 0;
23   args_0 = (void**)(malloc(1 * sizeof(void*)));
24   int8_t* arg0 =(int8_t*)(malloc(sizeof(int8_t)));
25   *arg0 = 22; //the actual argument
26   args_0[0] = arg0;
27   //trigger environmental input
28   EnvInput* input = g_ptr_array_new();
29   g_ptr_array_add(input,
30     create_event(EventEnum_e1, args_0));
31   g_async_queue_push(m1->queue, input);
32   } ...
33 }
```

(b) Environmental Code

Figure 5.6: Example of Behavioural Code Layout

array; lastly the EnvInput instance is pushed onto an input queue. To terminate a state machine (Line 10-17), a special "terminate" event is sent to the state-machine instance which will be terminated after all pending environmental inputs in the input queue are processed; afterwards the thread is joined with the main thread and the input queue is released.

5.2.2 Template-based Generator

In this section, we briefly talk about the templates that reduce each element in the source model to a code fragment in the target language. Detailed descriptions about the template-based generator can be found in Appendix B.

- **reduce_StateMachine** It maps a **SMGlobalDeclaration** element to the struct and

enum types shown in Figure 5.5 and the functions shown in Figure 5.6a. In the generated *execute_big_step*() function, reduce_StateMachine calls template reduce_Region for each contained region, which generates code to collect enabled transitions. Then reduce_StateMachine generates code to calculate and execute the result set in a small-step, finally generating code to execute the delayed event-binding calls at the end of a big-step.

- **reduce_Region** It maps a **RegionLocalDeclaration** element to a **StatementList** that contains a switch-case statement within which reduce_Region is recursively called for each sub-region within sub-states of the current region, and another switch-case statement that generates code to collect enabled transitions in the current region.

- **reduce_EventArgRef** Event arguments of the triggering event can be referred in the action or guard condition of a transition, . However in the generated code, the type of event argument is not an argument: it is a struct member in an instance of Event. reduce_EventArgRef resolves an **EventArgRef** to the corresponding **StructMemberRef**.

- **reduce_LocalVarRef** It resolves a **LocalVarRef** in a state machine onto a **StructMemberRef** in a SMStruct instance.

- **reduce_EventCall** It reduces an **EventCall** (event generation inside a state machine) to a **StatementList** with code that wraps the actual arguments in a pointer array, creates an Event instance, and collects event-binding calls.

- **reduce_SMStart/SMTrigger/SMTerminate** These templates reduce **SMStart**, **SMTrigger**, and **SMTerminate** to **StatementList** with corresponding code as shown in Figure 5.6b.

- **reduce_SMType** It reduces a **SMType** to a SMHandle struct.

Chapter 6

Validation

We have presented BSML-mbeddr, for building semantically configurable state-machine models in mbeddr's C programming environment. In this chapter we investigate the correctness and expressiveness of BSML-mbeddr.

6.1 Correctness

To demonstrate that our implementation of BSML matches its specification [7][9], we have designed test suites that cover all implemented semantic options (Section 3.2 and Table 4.1) and all language features (Section 4.1.3). For each test case, we identify the semantic option or language feature it should exercise, and we design the state-machine model and semantic configuration that forms a suitable context for the testing. Next, we group test cases that use the same semantic configuration and that use the same state-machine model, and we merge similar state-machine models used by different groups of test cases. Eventually, our designed test forms a set of test suites, each of which has a structure as follows:

- A semantic configuration associated with an mbeddr program
- An mbeddr program comprising:
  - One or more state-machine models
  - Environmental code containing a set of test-case functions, each of which tests the functionality of a single semantic option or a language feature. A test-case function contains:

* Statements that create an instance of a state-machine model and trigger environmental inputs.
* Statements that check whether the output matches the expectation.

In addition to test cases that test the functionality of a single semantic option or language feature, we also have test cases with complex state-machine hierarchy and execution behaviour that test the overall functionality, although not systematically. We did not cover all valid combinations due to the huge search space.

A language feature or semantic option might be tested multiple times in different contexts. For example, cross-hierarchy transitions are tested in the situation where its target state is the ancestor or descendant of the source, as well as in the situation where the target state and source state are in orthogonal regions; variables are tested in situations where they are used in guard conditions, entry blocks, actions, and where they are evaluated in expressions and assigned to; the semantic option ARENA ORTHOGONAL is tested when two transitions are: 1) arena orthogonal, 2) not arena orthogonal but source-target pairwise orthogonal, and 3) neither source-target pairwise orthogonal nor arena orthogonal.

Our test cases use the *mbeddr.unittest* language extension, so that performing a test case simply entails including the test in an **ExecuteTest** expression in the *main* function. *mbeddr.unittest* allows the tester to include *assertion*s in a test-case function, so that an error will show when the assertion is not true. BSML-mbeddr state machines run asynchronously whereas *mbeddr.unittest* requires that assertions of results are performed in the same test-case function where the environment input is triggered; thus, we use synchronization techniques to make sure the processing of a big-step has finished before accessing and asserting the returned results.

6.2 Expressiveness

We have conducted several case studies to exercise the expressiveness of BSML-mbeddr. Our motivation is to assess the applicability and integrability of BSML-mbeddr into mbeddr's C programming environment, and to check that one can use BSML-mbeddr to build real-world state-machine models with various semantic requirements. We categorize our intentions of exercising BSML-mbeddr into five categories: 1) big-step execution semantics, 2) hierarchical states and cross-hierarchy transitions, 3) concurrent regions and inter-region communication, 4) configurable semantics, and 5) code-model interaction and integration.

We have conducted three case studies with BSML-mbeddr: 1) a Ground Traffic Control (GTC) system [26] which exercises concurrent regions and big-step semantics; 2) a Dialler

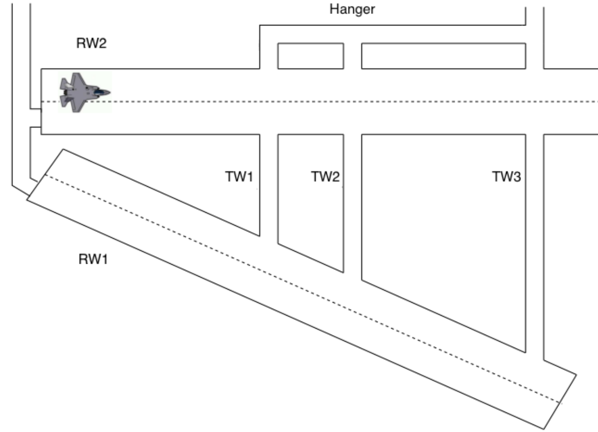


Figure 6.1: GTC Case Study [26]

System case study, adopted from example models in BSML [9], which exercises configurable semantics, big-step semantics, concurrent regions, and cross-hierarchy transitions; and 3) a State-Machine Factory case study that we created ourselves, and exercises the model-environment interactions and integration. The third case study demonstrates an approach to implement the *synchrony hypothesis* in BSML-mbeddr.

To conveniently visualize our model, we introduce some graphical notations in the following figures: the notation "$t_1 : (e_1 \wedge e_2)[guard]/action;$" denotes that a transition t_1 is enabled by the presence of two triggering events e_1 and e_2, whose guard condition is *guard* and action is *action*; the initial state of a region is pointed by an arrow with black dot; a stable state is notated by a check mark $\checkmark$.

6.2.1 Ground Traffic Control

Our Ground Traffic Control (GTC) case study is adopted from a work by Prout [26], originally developed by Bultan and Yavuc-Kahveci [37]. GTC simulates an airport control system that receives and sends signals to schedule airplanes to exclusive access to runways and taxiways that interconnect runways. We select GTC as one of our case studies because of its complex logical inter-region communications through shared variables and events, that helps to exercise the expressiveness of BSML-mbeddr.

Shown in Figure 6.1, GTC simulates an airport control system that schedules the usage of two runways RW1 and RW2, three taxiways TW1, TW2 and TW3, and a hanger. The

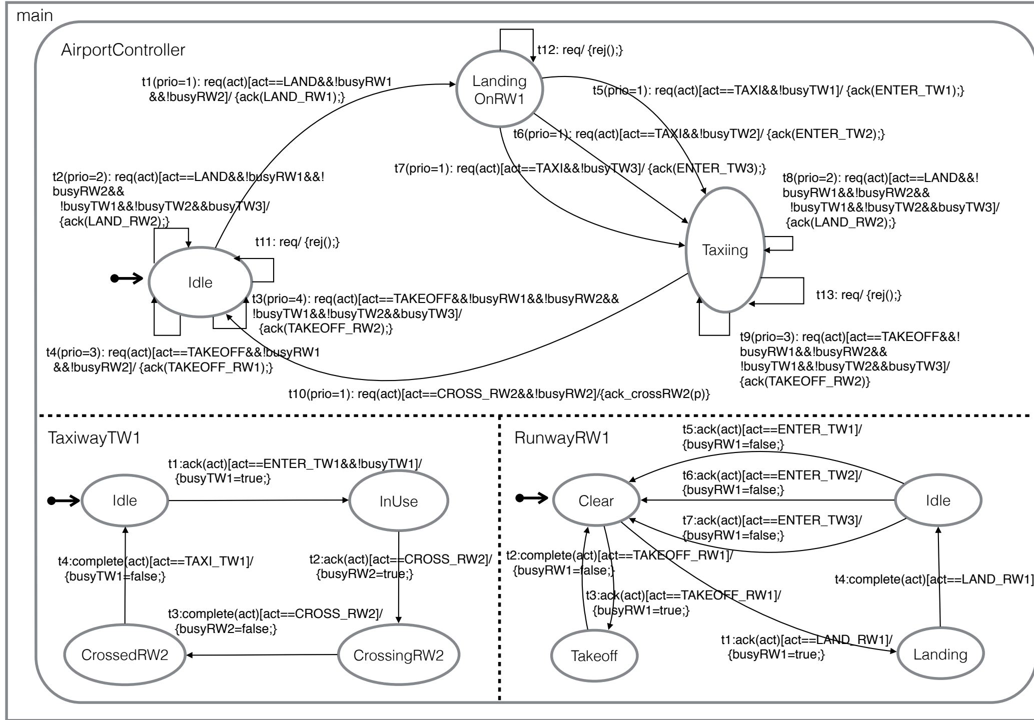
airport may be used by an arbitrary number of airplanes that may take off or land on either runway. Arriving airplanes landing on runway RW1 must taxi on a taxiway to reach a hanger, during which the airplane needs to cross runway RW2. The following properties must hold for the system:

1. Only one airplane can use a runway at a time.
2. Only one airplane can use a taxiway at a time.
3. An airplane can use runway RW1 (RW2) only if no airplane is using RW2 (RW1).
4. An airplane on a taxiway can only cross runway RW2 if no airplane is using it.
5. An airplane can land or take off on RW2 only if no airplane is on a taxiway.

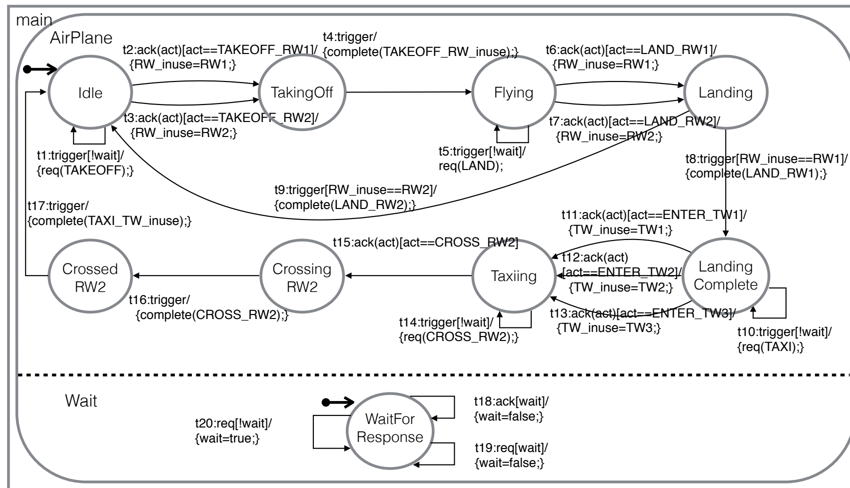
As shown in Figure 6.2a, we model GTC as a state machine with several concurrent regions: an Airport Controller, a Taxiway Controller for each taxiway, and a Runway Controller for each runway. The regions for taxiways TW2, TW3 and runway RW2 are not shown in the figure, but their structures are similar to those for TW1 and RW1; all six regions are modelled in the BSML-mbeddr case study. The Airport Controller receives a *req*(*act*) event with an argument *act* indicating the requested action (e.g., request to take off, land, enter a taxiway) from an airplane, and generates an *ack*(*act*) event with an argument *act* indicating the granted action (e.g., granted to take off on RW1, land on RW2, enter TW1) if the request is safe to be granted. The generated *ack* event causes updates to the status of a taxiway or a runway over the course of several small-step; at the end of the big-step, the airplane is notified to take the requested action. A Taxiway Controller receives *ack* events from the Airport Controller and *complete* events from airplanes, to update the status of a taxiway. Similarly, a Runway Controller receives *ack* events from the Airport Controller and *complete* events from airplanes, to update the status of a runway.

We modelled the airplane as a state machine AirPlane (see Figure 6.2b) with an arbitrary number of instances that interact with GTC through bound functions. An AirPlane instance maintains its current mode of operation (e.g., flying, landing, taxiing), and receives an in-event *trigger* from the tester to move a step forward. It updates its status when receiving *ack* events from the GTC to perform an action and generates *complete* events when the action has been performed. A successful take-off/landing cycle of a plane in our model is as follows:

1. Generate *req*(TAKEOFF) to request to take off.



(a) GTC Model



(b) AirPlane Model for Testing

Figure 6.2: GTC Models

2. Upon receiving *ack*(TAKEOFF_RW1/RW2), take off on RW1/RW2 and change status from Idle to TakingOff.
3. Generate *complete*(TAKEOFF_RW1/RW2) to notify GTC the completion of taking off and change status to Flying.
4. Generate *req*(LAND) to request to land.
5. Upon receiving *ack*(LAND_RW1/RW2), take off on RW1/RW2 and change status to Landing.
6. Generate *complete*(LAND_RW1/RW2) to nodify GTC the completion of landing. Change status back to Idle if the airplane uses RW1 for landing, otherwise change status of LandingComplete if the airplane uses RW1 for landing and continue the following steps.
7. Generate *req*(TAXI) to request to enter a taxiway.
8. Upon receiving *ack*(ENTER_TW1/TW2/TW3), enter the corresponding taxiway and change status to Taxiing.
9. Generate *req*(CROSS_RW2) to request to cross runway RW2, in order to reach the hanger.
10. Upon receiving *ack*(CROSS_RW2), cross the runway RW2 and change status to CrossingRW2.
11. Generate *complete*(CROSS_RW2) to notify GTC the completion of using RW2.
12. Generate *complete*(ENTER_TW1/TW2/TW3) to notify GTC the completion of taxiing and change status back to Idle.

Shown in Figure 6.3, in the environmental code, we write test cases that instantiate multiple airplanes, and simulate their concurrent interaction with GTC. For verification, we express the properties described previously as assertions that must hold; any property violation is reported as error. For example, to verify property 3, in the entry blocks of states Landing and Takeoff within region RunwayRW1, we check whether runway RW2 is in use; an error is reported if so. Additionally, properties 3, 4, and 5 are verified at the end of each big-step, to make sure that they hold when the state machine is in a stable status.

We have made several unsubstantial changes to the original GTC model discussed as follows.

```
#constant MAX_TEST_ROUND = 1000000;
#constant PLANE_NUM = 16;
int main() {
  GTC gtc=sm_start(GTC);
  AirPlane*[PLANE_NUM] planes;
  for (i ++ in [0..PLANE_NUM])
    plane[i]=sm_start(AirPlane);
  for (i ++ in [0..MAX_TEST_ROUND]) {
    //randomly pick an airplane to trigger
    int64 rand=random() & 017;
    sm_trigger(planes[rand],trigger(planes[rand]));
  }
  sm_terminate(gtc);
}
```

Figure 6.3: GTC Testing Environment

First, the original GTC lacks building of the environment, whereas we have created an AirPlane state machine that simulates the interaction between GTC and multiple AirPlane instances, as described above.

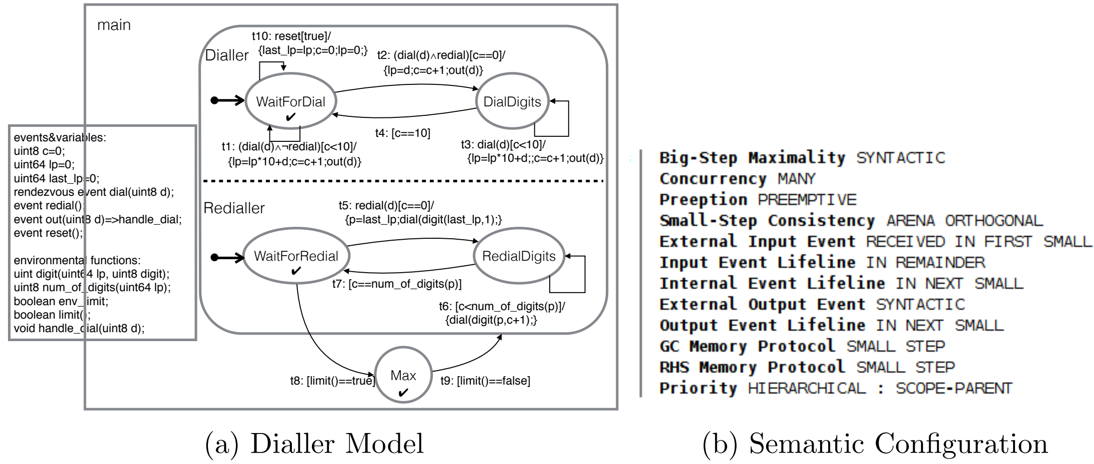
Second, since the original model lacks an interacting environment, rejected requests by airplanes are simply discarded by GTC, which might cause problem in our environment – repeated requests sent by an airplane could be granted multiple times by GTC (e.g., GTC might grant the airplane to land on RW1 and RW2 simultaneously), which case cannot be handled correctly by the original model. Therefore, we introduce a *rej* event to communicate rejected requests to airplanes. Specifically, if a *req* event does not trigger any transition in GTC, then a *rej* event is generated indicating the denial of the requested action; in AirPlane, after sending a *req* event, the airplane cannot send more requests until it receives response from GTC (*ack* or *rej*); if the request is rejected, the airplane sends the same request again until it is granted. This is achieved by adding a low-priority transition that generates *rej* to each state in Airport Controller (Figure 6.2a), and by adding a region $Wait$ in AirPlane, where the value of a boolean variable *wait* that indicates whether the airplane can generate *req* events is maintained (Figure 6.2b).

Third, the original model uses *rendezvous* events for inter-region communication, to guarantee that the statuses of taxiways and runways are updated instantly, to ensure that the state machine is always in consistent status at the end of every big-step. In contrast, BSML-mbeddr uses normal events for inter-region communication, which are sensed as present from the next small-step. We only need to make sure the status of a state machine at the end of a big-step is consistent because inconsistent statuses in between small-steps are not observable by the environment. In a second version of GTC, we also modelled inter-region communications using rendezvous events in BSML-mbeddr. Specifically, we changed the *ack* event in the above model to a *rendezvous* event, which results in a model that works correctly as well.

Lastly, we left a property (called *priority* property) in the original model – stating that landing requests have higher priority than take-off requests – unimplemented. Because an input queue is used for each state-machine instance to sequentialize all inputs, the *priority* property cannot be enforced by assigning higher priority to transitions that handle landing requests than transitions that handle take-off requests, as did in the original model. It is meaningless in the context of the environment we build because each big-step can have only one request from an AirPlane instance as input, whichever arrived first. However, there are approaches to simulate the situation where landing requests are processed with a higher priority than take-off requests, by prioritize requests in the environment. For example, we could create another state machine that collects requests from AirPlane instances; requests arriving during a certain period of time are treated as "requesting arriving simultaneously"; a high-priority request (i.e., a landing request, if any) in the collected requests is selected and sent to GTC, while the rest requests are rejected. To implement the *priority* property in such way would complicate our case study whereas the expressiveness of BSML-mbeddr would not be further explored. Therefore, we decide to left this property unimplemented.

6.2.2 Dialler System

Our Dialler System case study is adopted from Esmaeilsabzali's thesis [7] to exercise big-step semantics, hierarchical states, and inter-region communication of BSML-mbeddr. In addition, we explain at the end of the section how the selection of different semantic options distinctly affects the execution semantics of the model and how it affects the model's correctness. The user of the Dialler System is able to dial the digits of a phone number, or simply redial the previously dialled number. In addition, if the maximum number of concurrent calls is reached, the dialling process should be interrupted. Shown in Figure 6.4a, the Dialler System contains a state with two regions Dialler and Redialler, and a state Max for checking whether the limit of concurrent calls is reached. Region Dialler receives an in-event $dial(d)$ when the user dials a digit d, thereby by triggering transition t_1, which transmits out-event $out(d)$ to the environment (e.g., the phone system) to establish a phone connection. More digits than the first 10 dialled digits are ignored until the user hangs up the phone. When the user hangs up the phone, $reset()$ is generated that triggers t_{10} to reset the status of Dialler and save the previously dialled number in $last_lp$. When in-event $redial()$ is received, the Dialler System dials all digits of the previous dialled number in a single big-step with multiple small-steps. Specifically, region Redialler reacts to in-event $redial()$ by executing t_5 and t_6 that generates $dial(d)$ for each digit d in $last_lp$. Then the generated rendezvous event $dial(d)$ triggers transition t_2 or t_3 in the same small-step which further generates $out(d)$. Lastly, the Dialler System returns to WaitForDial and



(a) Dialler Model (b) Semantic Configuration

Figure 6.4: Model and Semantic Configuration of the Dialler System

WaitForRedial states after the redialling is done. In each small-step, the environmental variable *limit* is checked, and t_8 is executed if the maximum number of concurrent calls is reached, which interrupts the dialling/redialling process. For thread safety, we use a mutex to protect the access of environmental variable *limit*.

We delicately select semantic options that make the model work correctly, as shown in Figure 6.4b, and as explained as follows:

- SYNTACTIC is chosen for Big-step Maximality. State WaitForDial, WaitForRedial, Max are tagged as stable states.

- Event *dial* is a *rendezvous* event that has a PRESENT IN SAME event lifeline. If generated in a small-step, *dial* is enabled in the same small-step and might trigger more transitions, which makes sure the generated *dial* in t_5 (t_6) will trigger corresponding t_2 (t_3) in the same small-step.

- Concurrency is MANY, which allows multiple transitions to be executed in the same small-step, in order to support the rendezvous semantics.

- External Input Events is RECEIVED IN FIRST SMALL. Since *dial* is generated both from the environment and inside the state machine, we want to be sure that the *dial* events that are generated inside the state machine not determined to be in-events. An alternative is to choose SYNTACTIC for External Input Events and PRESENT IN NEXT SMALL for Input Event Lifeline, but this will restrict other in-events from being

```
1  statemachine SM_Dialing {
2    region main initial = Dialing {
3      uint8 c = 0;
4      uint64 lp = 0;
5      uint64 last_lp = 0;
6      rendezvous event dial(uint8 d);
7      event redial();
8      state Dialing {
9        region Dialer initial = WaitForDial {
10         event out(uint8 d) => handle_dial;
11         event reset();
12         (stable) state WaitForDial {  };
13         state DialDigits {  };
14         t1: on dial &&  redial[c < 10]
               WaitForDial -> WaitForDial {
15           c = c + 1;
16           lp = lp * 10 + d;
17           out(d);};
18         t2: on dial && redial[c == 0]
               WaitForDial -> DialDigits {
19           lp = d;
20           c = 1;
21           out(d);};
22         t3: on dial[c < 10] DialDigits ->
               DialDigits {
23           lp = lp * 10 + d;
24           c = c + 1;
25           out(d);};
26         t4: on [c == 10] DialDigits ->
               WaitForDial;
```

```
27         t10: on reset[true] WaitForDial ->
                WaitForDial {
28           last_lp = lp;
29           c = 0;
30           lp = 0;};
31       };
32       region Redialer initial = WaitForRedial
             {
33         uint64 p;
34         (stable) state WaitForRedial {  };
35         state RedialDigits {  };
36         t5: on redial[c == 0] WaitForRedial ->
                 RedialDigits {
37           p = last_lp;
38           dial(digit(last_lp, 1));};
39         t6: on [c < num_of_digits(p)]
               RedialDigits -> RedialDigits {
               dial(digit(p, c + 1)); };
40         transition t7: on [c == num_of_digits(
               p)] RedialDigits ->
               WaitForRedial;
41       };
42     };
43     (stable) state Max {  };
44     t8: on [limit()] WaitForRedial -> Max;
45     t9: on [!limit()] Max -> Dialing;
46   };
47 };
```

(a) (b)

Figure 6.5: Dialler Code

present in the remainder of the big-step. Note that even though the internal-event *dial* is a rendezvous event, the in-event *dial* that is received at the beginning of a big-step does not have a PRESENT IN SAME event lifeline because the event is not generated in a small-step.

- Preemption is PREEMPTIVE, so that when t_8 is enabled, any other enabled transitions are interrupted and the dialling/redialling process is aborted.
- Priority is HIERARCHICAL, with sub-options SCOPE-PARENT. This ensures that t_8 has higher priority than any transitions inside state Dialler, so that t_8 can interrupt them. An alternative is to choose Explicit priority and assign higher priority to t_8 than the other transitions.

We have designed test cases that validate that the dialling, redialling, and limit checking functionality work as expected.

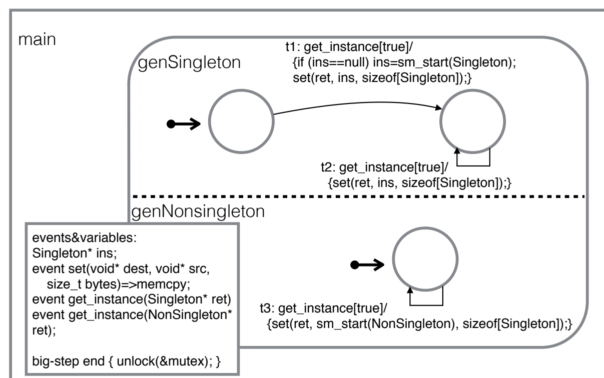


Figure 6.6: State-Machine Factory Model

6.2.3 State-Machine Factory

Lastly, we have modelled a State-Machine Factory which exercise the way that state machines interact with their environment. This case study also demonstrates how to support the *synchrony hypothesis* in BSML-mbeddr, such that a reaction (big-step) of the state machine is considered to be atomic. Shown in Figure 6.6, a state machine SMFactory is responsible for creating instances for state machines Singleton and NonSingleton, through in-events *get_singleton_instance*() and *get_nonsingleton_instance*(), respectively. SMFactory keeps an instance of Singleton and delivers the reference of it to the user upon request, whereas it creates a new instance of NonSingleton upon request.

As shown in Figure 6.7a, in the environment, the user creates an instance of SMFactory, and triggers *get_singleton_instance*() and *get_nonsingleton_instance*() to get instances of Singleton and NonSingleton (Line 6). Note that an environmental input may contain multiple in-events, and their instances are received and processed in the same big-step. The user must provide the address of variables of state-machine types (Line 3-4) as arguments. In order to make sure that SMFactory acts atomically (i.e., that the user proceeds only when SMFactory finishes processing a big-step), we use a mutex to synchronize interactions between the state machine and environment. A mutex is locked (Line 5) before a *sm_trigger*() statement. The mutex is also locked (Line 7) before the first time the returned state-machine instance is accessed. This technique ensures that SMFactory reacts atomically so that the environment proceeds only when the state machine has finished processing the environmental input.

In SMFactory, reference to pre-existing Singleton machine is returned upon request. Out-event *set_instance*() is bound to library function *memcpy*() (Line 14) and is issued

```
1   int32 main(int32 argc, string[] argv) {
2     SMFactory* sm = sm_start(SMFactory);
3     Singleton ret_single;
4     NonSingleton ret_multi;
5     g_mutex_lock(&mutex);
6     sm_trigger(sm, get_singleton_instance(&ret_single),
          get_nonsingleton_instance(&ret_multi));
7     g_mutex_lock(&mutex);
8     sm_trigger(&ret_single, e());
9     sm_trigger(&ret_multi, e());
10    return 0;
11  } main (function)
12  statemachine SMFactory {
13    region main initial = main_state {
14      event set_instance(void* dest, void* const src, size_t
            bytes) => memcpy;
15      event unlock_mutex(GMutex* mutex) => g_mutex_unlock;
16      state main_state {
17        region genSingleton initial = off {
18          Singleton* instance = null;
19          event get_singleton_instance(Singleton* ret);
20          state off {  };
21          state on {  };
22          t1: on get_singleton_instance[true] off -> on {
23            instance = sm_start(Singleton);
24            set_instance(ret, instance, sizeof[Singleton]);};
25          t2: on get_singleton_instance[true] on -> on {
26            set_instance(ret, (void*)instance, sizeof[
                  Singleton]);}; };
```

(a)

```
27        region genNonsingleton initial = any {
28          event get_nonsingleton_instance(
                NonSingleton* ret);
29          state any {  };
30          t1: on get_nonsingleton_instance[true]
                any -> any {
31            set_instance(ret, sm_start(
                  NonSingleton), sizeof[
                  NonSingleton]);
32            // unlock_mutex(&mutex)
33          };};};
34    big-step end { g_mutex_unlock(&mutex); };
35  };};
36  statemachine NonSingleton {
37    region main initial = AnyState {
38      event e();
39      state AnyState {  };
40      t1: on e[true] AnyState -> AnyState;
41    };};
42  statemachine Singleton {
43    region main initial = AnyState {
44      event e();
45      state AnyState {  };
46      t1: on e[true] AnyState -> AnyState;
47    };};
```

(b)

Figure 6.7: State-Machine Factory Code

for each request, ensuring that the actual assignment is performed at the end of a big-step. Then SMFactory releases the mutex to unblock the user, after processing a big-step (Line 34). The generalized usage of this technique is to block the user depending on the status of the state machine. For example, imagine a debugger state machine that processes two kinds of environmental inputs: commands from the console as well as callback messages from the running program. We might want the user to be prevented from issuing commands unless the target program has finished running. This can be achieved by locking a mutex before a command is issued by the user in the environment, and unlocking the mutex when the debugger enters a certain state in the state machine that grants the user to issue commands.

Chapter 7

Discussions

In this chapter we address several issues we have encountered from our experience during implementing BSML-mbeddr, which could guide people who work on similar language implementations.

7.1 Designing Data Structure of Generated Code

Considering the big gap between the low-level target language such as C, and the high-level source language such as a state-machine modelling language, we believe that one important issue is to delicately design the data structure of generated code so that the required run-time information is efficiently accessible.

Figure 7.1 illustrates the information flow during code generation and run-time execution of BSML-mbeddr. During code generation, information is retrieved from state-machine models and converted to proper data format in the generated code, such as structs Transition and SMStruct. Functions based on templates are generated for basic behavioural semantics that execute big-steps, identify enabled transitions, check semantic consistency, execute transitions, and so on. Information on semantic configuration is retrieved that resolves variation points in the execution semantics of a state machine, such as the behaviour of consistency checking among transitions, and of retrieval of variable values from a snapshot; information on semantic configuration is resolved during code generation and thus not stored in the generated code. Information that can be determined statically is identified and computed during code generation, instead of being pushed to run-time to be computed. For example, the sequence of entry blocks that a transition need to execute

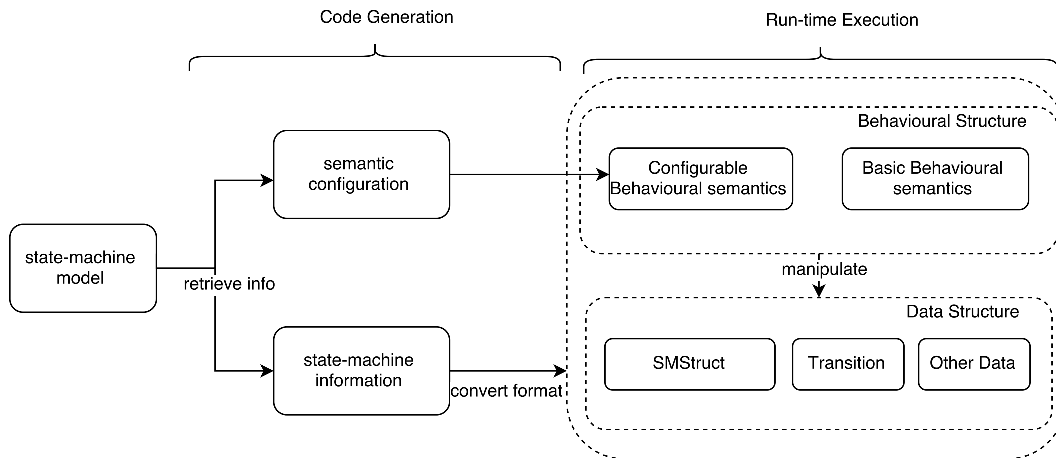


Figure 7.1: Information Flow during Code Generation and Run-time Execution.

is computed during code generation, and such an array of function pointers is stored in the Transition struct as a constant. During run-time execution, functions that represent the behavioural semantics of state machines are executed, which manipulate static and run-time information stored in our designed data structure.

Reckless decisions on the data structure design may cause required run-time information being inaccessible. For example, we may generate a function for each state machine and store its run-time status as static variables of the function. However this approach allows only one instance per state machine because multiple instances running simultaneously have to share the same set of static variables. A more flexible solution as we adopted, is to create a struct SMStruct to store all the run-time information, which allows multiple instances of the same state machine running concurrently (Section 5.2.2).

Reckless decisions on the data structure design may also cause required run-time information being inefficiently accessed. For example, we may store the state hierarchy of a state machine as a tree structure in the generated code, and query for relations between two given nodes in the state hierarchy by tree searching. However, we find that at run-time we only need the following information for a state hierarchy: 1) given two regions, whether they are orthogonal; 2) given two transitions, whether one transition interrupts the other; 3) if semantic aspect Priority is HIERARCHICAL, given two transitions, which one has higher priority. Therefore, we have decided to store the state-hierarchy information as bitmaps (e.g., a two-dimension bitmap *orthogonal*, where $orthogonal[a][b]$ is true if regions a and b are orthogonal.) in the generated code that returns the required run-time information in constant time complexity.

7.2 Evolving Semantic Configuration

An obstacle to implement configurable semantics is to consider dependencies between syntax and semantics – certain syntax must be enable or disabled when the semantic configuration is change by the modeller, which requires extra caution to handle.

This issue is alleviated on mbeddr – with the *projectional editor* natural of mbeddr, we are able to preserve syntactic information in the model which is neither projectioned to concrete notations nor resulted in the generated code. For example, when semantic aspect Big-step Maximality is SYNTACTIC, states can be tagged as **stable**. If the modeller change the option from SYNTACTIC to TAKE MANY, then all the **stable** tags should be removed. Instead of removing the syntax from the model, we hide **stable** tags from the user and ignore them during the code generation. If Big-step Maximality is changed back to SYNTACTIC, the hidden **stable** tags will show up again and take effect during code generation. It saves the model from information loss during evolving semantic configuration.

When a certain syntax is enabled but its value is absent, normally the modeller is responsible to fill all absent values. For example, if the modeller changes the semantic aspect Big-step Maximality from TAKE MANY to SYNTACTIC, the modeller shall specify every state to be either stable or non-stable. In order to easy the burden from the modeller, we impose default values for syntax whose values are absent, so that the modeller only need to fix the syntax when the default value is incorrect. In the above example, all states are non-stable states by default, and only states that are stable need to be tagged.

7.3 Computational Complexity

In BSML, the process of a small-step includes deriving all maximal, consistent result sets of enabled transitions, which incurs high computational cost. In BSML-mbeddr, nondeterminism is resolved by the textual order in which model elements are declared, so that priority is a total order based on which we are able to apply a greedy process to calculate a single result set for execution (Section 4.2.3). Our approach reduces the computation complexity drastically, whereas it is proved that given an arbitrary set of enabled transitions and consistency checking criteria, the result set constructed by BSML-mbeddr is one of the result sets with highest priority constructed by the BSML big-step process (Theorem 4).

7.4 Language Usability

Some functionality introduced by new language features of BSML-mbeddr has equivalence in BSML, but presented in a complicated way. For example, the functionality of the new feature *entry block* can be achieved in BSML by appending statements in the entry block to the actions of all transitions that enter the state or region. Similarly, language feature *static variable* can be achieved by defining these static variables as non-static, but in the *main* region; variables defined within the *main* region are initialized only when the state-machine instance is created. We add these language features to BSML-mbeddr not to extend the expressiveness of BSML but to makes it easier for the modeller to use.

7.5 Semantics of Added Language Features

When adding a new language feature to BSML-mbeddr, we must make sure to give it appropriate syntactic and semantic meaning that neither confuses the modeller nor incurs conflict with existing syntax and semantics of BSML. With configurable semantics, we also need to decide whether its semantics shall be fixed or configurable[1].

For example, the language feature *event binding* is originally introduced to deliver outputs from a state-machine model to its environment. Before adding this feature, we ask ourselves the following questions: a) shall event binding apply to all types of events or just out-event? b) if event binding applies to all types of events, then what semantic meaning shall be given to bindings to internal-events and in-events? c) if event binding applies to out-event only, how to deal with the case that semantic aspect External Output Events is not SYNTACTIC, where whether an event is an out-event is determined at runtime?

A reasonable potential usage for event binding to in-events is to trigger environmental inputs. That is, an in-event is triggered whenever the bound function is called. However, we are not able to implement this if there exists functions imported from a C library due to the limitation of mbeddr. Another issue is that, if semantic aspect External Output Events is SYNTACTIC, we are able to syntactically tag an event as out-event, which might confuse the modeller because only event bindings to out-events take effect, otherwise are ignored.

To resolve the above issues, our solution is to let event binding be the syntactic notation that determines whether an event is an out-event. If External Output Events is SYNTACTIC,

[1]For consistency with the semantic deconstruction of the original BSML, all our added language features are given fixed semantics.

then any event-binding call is executed because an event with a binding is always an out-event. If External Output Events is not SYNTACTIC, then only event-binding calls whose event is determined to be an out-event at run-time are executed. This semantics for event binding are both consistent with BSML's External Output Events semantic aspect, and make sense for the modeller to use.

Fortunately, we did not encounter such problems for many of the added languages features which have quite straightforward syntax and semantics.

7.6 Event with Multiple Instances

BSML-mbeddr allows an event to be defined with arguments so that each generated event instance is distinct. The semantics of multiple generated instances of the same event need to be carefully resolved.

For in-events, multiple environmental inputs containing instances of the same in-event can be generated and put into an input queue, each of which is processed by the state machine with a big-step. For out-events, all event-binding calls of generated out-events are collected and executed at the end of a big-step. However, it is tricky to resolve multiple instances of the same internal-event, with consideration of the Internal Event Lifeline semantic aspect. Internal Event Lifeline regulates how long a generated instance of internal-event is sensed as present, so that it is possible that an internal-event instance disappears after a small-step without being processed by any transition, or remains to be present after being processed by a transition. Therefore, the semantics "processed exactly once" does not apply to BSML-mbeddr's internal-events. Our solution is to treat internal-events in the same way as to internal variables: the execution of a transition may write to a shared variable which may trigger transitions in another region and its value may be read in some other executed transitions. If the shared variable is written several times by multiple executed transitions, then later values will overwrite earlier values. Similarly, the execution of a transition may generate an internal-event, that enables transitions in another region, and the values of its arguments may be read in some other executed transitions. If the internal-event is generated multiple times in a big-step, then later arguments will overwrite earlier arguments.

Esterel [3], a member of BSML family, allows multiple instances of the same out-event being generated at the same time, because the generation of out-events in Esterel takes effect instantaneously. Esterel allows the modeller to associate an associative, commutative *combination function* with each out-event, so that simultaneous generation of multiple

instances of the same out-event are combined into a single instance by combining each argument of the out-event. Because BSML-mbeddr collects all instances of out-events and calls their bound functions at the end of a big-step, BSML-mbeddr is able to support Esterel semantics by storing arguments of each out-event instance (e.g., by storing them in static or global variables of array type) in the bound function and combine them in the same way as in a *combination function*.

7.7 Big-step Semantics

In this section we address several issues considering the regulation of the big-step semantics.

First, BSML requires that the consequence of a big-step is not observable by the environment until the end of a big-step. We use the following three strategies to achieve this goal:

- The bound function of a generated out-event is not called immediately. Instead, we collect all the event-binding calls and delay their execution to the end of a big-step.
- In the state machine, we banned any operation that might change the status of the environment, including global variable reference on the left-hand-side (LHS) of an assignment, pointer dereference on LHS, self incremental or decremental operation on global variables, etc.
- A function can be called in a state machine only if it is tagged as a *state-machine function*. A state-machine function must not change the status of the environment, neither can it call any function that is not a state-machine function.

Second, the question raised that whether it should be allowed to call functions in a state machine that is imported from a C library. Since the source code of a library function is not accessible, we are not able to check whether the function will change the status of environment or not. Thus, allowing to call functions in a library may conflict with the semantics of a big-step, so we banned calls in a state machine to any function that is imported from a library.

Third, it is a desirable property to let multiple instances of the same state machine running concurrently without the machines interfering with each other. We address the obstacles and our solutions to achieve this goal as follows:

- Multiple instances of the same state-machine might have different run-time statuses. As a solution, we store all run-time information of a state machine instance in a SMStruct struct, and each state-machine instance has its own SMStruct instance.

- Multiple state-machine instances might call the same function in the state machine. Because we only allow state-machine functions to be called in a state machine, and a state-machine function is thread-safe, multiple state-machine instances can call the same function without interfering with each other.

- Multiple state-machine instances might call the same function through event bindings to out-events. Because event-binding calls are executed at the end of a big-step, an out-event is allowed to be bound to any environmental function, which does not conflict with the semantics of a big-step. In this way, thread-safety for bound functions cannot be guaranteed on the language side. Thus, we ask the modeller to keep bound functions thread-safe if they are possible to be called by multiple state-machine instances, and then non-interference among state-machine instances is achieved.

Last, we have added language features to improve integration of state-machine models and their C-code environment, whereas we make sure that the added features do not violate the semantics of a big-step. For example, a *big-step start* (*end*) *block*, inspired by the $constructor()$ ($finalizer()$) of a Java class, is a code block associated with a state machine, which is executed immediately before the beginning (after the end) of a big-step. Since the code block is out of the regulation of the big-step semantics, it belongs to the environment that can contain any environmental code. In addition, we allow it to instantaneously manipulate state-machine variables (i.e., instantaneously manipulate the state-machine snapshot at the beginning (end) of a big-step). *Big-step start* (*end*) *block* gives BSML-mbeddr the semantic power to execute some actions in between big-steps, and it is guaranteed to be executed exactly once for a big-step.

Chapter 8

Related Work

Our work is based on high-level, systematic deconstruction of BSML semantics, which covers a more comprehensive range of semantics than previous studies that compares different subsets of BSMLs [9] (e.g., Statecharts variants [36][18], Synchronous languages [16], Esterel variants [4][31], UML StateMachines [30]). BSML-mbeddr has a powerful execution semantics which is regulated by big-steps and small-steps – in each small-step all enabled transitions are identified, and the decision on which transitions shall be executed is made based on the semantic configuration. In contrast, many other state-machine modelling languages [6][22][13] have simpler execution semantics – all transitions starting from the activated state are checked by some order, and the first transition found enabled is executed.

By implementing within mbeddr, BSML-mbeddr gains the power to combine low-level executable code and high-level state-machine models in the same development artifact, whereas some previous works [26][11][9] create their language solely for modelling. For code generation, BSML-mbeddr uses mbeddr generator language supported by the rich Java library, which is more advanced than previous techniques such as code generation based on Template Semantics [26].

8.1 Semantically Configurable Code Generator

Prout, Atlee, Day and Shaker [26] have provided a prototype of a semantically configurable Code-Generator Generator (CGG) for a family of state-machine modelling languages. It uses template semantics based on preprocessor directives and conditional compilation as

the technique for code generation. CGG supports 26 semantic parameters, 89 parameter values and 8 composition operators, but not all combinations of parameter values result in a consistent semantics definition – configuring CGG's semantic requires expertise understanding of the consequences of the decisions and their interdependencies [26]. In contrast, BSML raises the abstraction level of semantic deconstruction by decomposing the semantics into only 12 mostly orthogonal semantic aspects and around 30 semantic options, which makes the semantics easier and more intuitive to be configured. Concurrency in CGG is achieved by composing multiple HTSs with various composition operators, whereas in BSML-mbeddr, concurrency is achieved by regions and the Concurrency aspect. In BSML-mbeddr, composition operators of CGG can be modelled via Concurrency, Consistency and Event Lifeline semantic aspects [9].

Exelmans [11] has worked on semantically configurable step-execution algorithm that accommodates various extended semantics of StateCharts and Class Diagram (SCCD). The step-execution algorithm utilizes the semantic deconstruction of BSML, and it has implemented semantic options within aspects Big-step Maximality, Internal Event Lifeline, Input Event Lifeline, Concurrency and Priority. In contrast, we claim to implement BSML the language itself and we have implemented more semantic aspects and options than in this work.

Faghih and Day [12] has performed semantically configurable analysis on BSML, whereas Lu, Atlee, Day and Niu [19] has done semantically configurable analysis on Template Semantics. By providing a source model and a set of parameter values that encodes the model's semantics, the source model is translated into a SMV model suitable for model checking. Esmaeilsabzali, Fischer and Atlee [10] has provided a framework that injects aspect code into the generated code of BSML, to enhance it with the capability to monitor a property at runtime. In addition, research by Cohen and Maoz [5] and research by Maoz, Ringert and Rumpe [20] provide analysis tools for Live Sequence Charts and Class/Object Diagrams, respectively. They have defined and formalized the variability in the semantics of the source modelling languages using feature models with multiple features. They have developed semantically configurable analysis solutions based on parametrized transformation, whose result complies with the selected feature configuration.

8.2 Code-model Co-development

mbeddr [22][27][35] provides a predefined DSL (*mbeddr.statemachine* [22]) that supports for basic state-machine models, which have simple execution semantics without concurrent regions. Similar to BSML-mbeddr, *mbeddr.statemachine* allows a mixture of code and

Language Features	BSML-mbeddr	mbeddr.statemachine
Configurable Semantics	✓	
Concurrent Region	✓	
Event Binding/Event Argument	✓	✓
Input with Multiple Events	✓	
Transition with Multiple Triggers	✓	
Negation of Triggers	✓	
Cross-hierarchy Transition	✓	
Entry Block	✓	✓
Variable	✓	✓
Function Call	✓	✓
Name Scoping	✓	✓
Priority	✓	
Multiple Instances of state machine	✓	✓
Asynchronous Execution	✓	

Table 8.1: Comparison between BSML-mbeddr and mbeddr.statemachine

state-machine models, and provides event binding and triggering event as methods for communication between C code and the model. We have compared *mbeddr.statemachine* with BSML-mbeddr comprehensively in Table 8.1. Rosenberger [28] has investigated the transformation of mbeddr state-machine model into NuSMV code for model checking with restrictions, such as no composite states, single assignment actions, no access to global state, etc.

Umple [1][15][13] is a programming/modelling language with a development environment in heterogeneous languages, where highly abstracted modelling notations (e.g., class diagrams, state machines) are integrated in the same development artifact of a programming language (e.g., Java, PHP, C++). Umple supports most of UML StateMachines [24] semantics, including events, signals, guards, transition actions, entry or exit actions, composite states and concurrent states. However, the execution of Umple state-machine is not regulated by big-step and no configurable semantics is provided. Badreddin et al. [2] has investigated code generation process of state-machine model in Umple, providing a concise and scalable code generation approach for state-machine models.

Chapter 9

Conclusion

This thesis have provided BSML-mbeddr, a state-machine modelling language with hierarchical states, concurrent regions and configurable semantics, which has implemented a large subset of BSML within the mbeddr C programming language environment. By implementing on mbeddr, BSML-mbeddr is integrated into a C programming environment that supports programs made with heterogeneous languages, including a combination of programming language and modelling language.

We introduced background knowledge about projectional editor, MPS, and mbeddr. mbeddr is a DSL workbench which provides a tool suite that supports the incremental construction of modular DSLs on top of C, together with a set of predefined DSLs. mbeddr allows low-level code and high-level state-machine models being developed in the same artifact in heterogeneous languages; they together are transformed into textual C code for execution.

We described the basic syntax and semantics, as well as configurable semantics of BSML-mbeddr. BSML-mbeddr supports sophisticated state-machine constructs that are available in real-world notations used by professionals. The modellers are allowed to choose the proper option for each semantic aspect and fulfil their per-domain or per-model semantic requirements. We summarized the differences between BSML-mbeddr and the original BSML, and explained the rationale behind the changes we made. We briefly showed the overall syntactic structure of the design, and the layout of the generated code for our implementation.

We validated the correctness and expressiveness of BSML-mbeddr by testing and case studies. We systematically designed test cases that covers all semantic options and language features in BSML-mbeddr. We have conducted case studies to assess the applica-

bility and integrability of BSML-mbeddr into mbeddr's C programming environment, and to check that one can use BSML-mbeddr to build real-world state-machine models with various semantic requirements. We discussed the challenges we encountered during implementing BSML-meddr, that might be of interest of people who work on similar language implementations.

Finally, we compared our language implementation with previous works, including the current support of state-machine modelling language for mbeddr, CGG with template semantics, and Umple which is a code-model co-development platform, etc. We showed the advantages of our language against previous works.

APPENDICES

Appendix A

Implementation: Editor, Constraint, Type System and Behaviour

Here we discuss the roles of the language aspects of *editor*, *constraint*, *type system* and *behaviour* in BSML-mbeddr in detail. Readers who are interested in the complete source code of BSML-mbeddr may access our github page `https://github.com/z9luo/BSML-mbeddr` to install BSML-mbeddr and view the full version of our source code. Installation instructions are also on the web page.

A.1 Editor

The *editor* aspect is to define how abstract syntax of the model is projected to concrete representations viewed by the modeller. Additionally, it acts as a key role to hide or recover syntax when certain syntax need to be enabled or disabled along with semantic configuration change. The editor aspect of a language concept is based on *cells*. A cell contains either a constant or a property from one of its child nodes or reference nodes. Shown in Figure A.1, the editor of **StateLocalDeclaration** is projected to a constant **stable**, followed by its type which is **state**, followed by its name and lastly its content which is a list of statements surrounded by open brackets. Then, the editor of **Statement** (or the sub-concepts of **Statement**) defines how each statement is projected to its concrete representation. The syntax **?(stable)** means the projection of constant **stable** is optional, depending on a condition function defined in the *inspector* window shown in Figure A.1b: **stable** is projected only if BigStepMaximality is SYNTACTIC.

```
show if          (editorContext, node)->boolean {
                   node<StateMachineConfigItem> sem_config = BCHelper.findBC(node.model).
                       findItemOfType(concept/StateMachineConfigItem/) : StateMachineConfigItem;
                   if (sem_config.bigStepMaximality.is(< SYNTACTIC >)) {
                     return node.stable;
                   }
                   return false;
                 }
```

(a) Editor, **StateLocalDeclaration**.

```
show if          (editorContext, node)->boolean {
                   node<StateMachineSemanticsConfigItem> sem_config =
                       BCHelper.findBC(node.model).findItemOfType(concept/StateMachineSemanticsConfigItem/) :
                       StateMachineSemanticsConfigItem;
                   if (sem_config.isNotNull && sem_config.bigStepMaximality.is(< SYNTACTIC >)) {
                     return node.stable;
                   }
                   return false;
                 }
```

(b) Inspector of cell "?(stable)" in A.1a, Editor, **StateLocalDeclaration**.

Figure A.1: Editor Example.

A.2 Constraint

We use the *constraint* aspect for two kinds of usage: 1) specify whether a given node can be parent/child/ancestor of the current node, and 2) specify the search scope of its reference node, if any. Shown in Figure A.2, the constraint aspect of **RegionLocalDeclaration** specifies that a region can only contain definitions of states, variables, events, transitions, blocks, and empty statements. In addition, it defines the search scope of the initial state to be its contained states.

A trickier situation is where the type of the current node is considered in combination with constraints. For example, a *sm_trigger* statement is to trigger environment inputs comprising multiple generated in-events (a list of **SMGenEvent** nodes), to a given state-machine instance. According to BSML-mbeddr semantics, only events defined within the given state machine and determined to be in-events can be generated. As shown in Figure A.2b, to define the search scope of in-events that can be generated in a *sm_trigger* statement, we need first resolve the type of an **Expression** which represents the referred state-machine instance, to a pointer type that points to a **SMType**. By the resolved type, we are able to refer to the state-machine declaration that associated with the type, as well as all the in-event declared within it. Resolving the type of an expression is achieved by importing the language module *mps.lang.typesystem* to the language that we used to define *constraint*, and use the type resolving operation provided by *mps.lang.typesystem*.

```
concepts constraints RegionLocalDeclaration {
  can be child <none>

  can be parent
    (childConcept, node, childNode, operationContext, link)->boolean {
      if (childNode.isInstanceOf(StatementList)) {
        return childNode : StatementList.statements.all({~it =>
              it.isInstanceOf(IState) || it.isInstanceOf(AbstractBlock) || it.isInstanceOf(LocalVariableDeclaration) ||
              it.isInstanceOf(IEvent) || it.isInstanceOf(ITransition) || (it.concept == concept/Statement/); });
      } else {
        return true;
      }
    }

  can be ancestor <none>


  <<property constraints>>

  link {initState}
    referent set handler:<none>
    scope:
      (exists, referenceNode, contextNode, containingLink, linkTarget, operationContext, enclosingNode, model, position,
        referenceNode.ancestor<concept = IRegion, +>.getContainedElements().ofConcept<IState>;
      }
```

(a) Constraint, **RegionLocalDeclaration**.

```
concepts constraints SMGenEvent {
  can be child
    (childConcept, node, link, parentNode, operationContext)->boolean {
      parentNode.ancestor<concept = SMTrigger, +>.isNotNull;
    }

  can be parent <none>

  can be ancestor <none>


  <<property constraints>>

  link {event_ref}
    referent set handler:<none>
    scope:
      (exists, referenceNode, contextNode, containingLink, linkTarget, operationContext, enclosingNode, model, position,
        node<StateMachineConfigItem> config = BCHelper.findBC(model).findItemOfType(concept/StateMachineConfigItem/) :
            StateMachineConfigItem;
        node<> t = enclosingNode.ancestor<concept = SMTrigger, +>.sm_handle.type;
        if (t.isInstanceOf(PointerType) && t : PointerType.baseType.isInstanceOf(SMType)) {
          return t : PointerType.baseType : SMType.sm_ref.descendants<concept = IEvent>.
              where({~it => config.externalInEvent.is(< SYNTACTIC >) ? it.isInEvent() : true; });
        }
        return new sequence<node<IEvent>>(empty);
      }
```

(b) Constraint, **SMGenEvent**.

Figure A.2: Constraint Example.

```
rule typeof_SMStart {
  applicable for concept = SMStart as smStart
  overrides false

  do {
    node<SMType> smt = new node<SMType>();
    smt.sm_ref = smStart.sm_ref;
    node<PointerType> ptr = new node<PointerType>();
    ptr.baseType = smt;
    typeof(smStart) :==: ptr;
  }
}
```

(a) typeof_**SMStart**

```
checking rule check_Event {
  applicable for concept = Event as eventDeclaration
  overrides false

  do {
    if (eventDeclaration.binding().isNotNull) {
      if (eventDeclaration.args().size != eventDeclaration.binding().binding.arguments.size) {
        error "wrong number of arguments: " + eventDeclaration.binding().binding.qualifiedName() -> eventDeclaration;
      }
      for (int i = 0; i < eventDeclaration.args().size; i++) {
        if (!(eventDeclaration.args.get(i).type.isSubtypeOf(eventDeclaration.binding.binding.arguments.get(i).type))) {
          error "wrong type of argument: " + eventDeclaration.args.get(i).type + " is not subtype of " +
              eventDeclaration.binding.binding.arguments.get(i).type -> eventDeclaration;
        }
      }
    }
  }
}
```

(b) check_Event

```
checking rule check_ContainerOfUniqueNames {
  applicable for concept = IContainerOfUniqueNames as coun
  overrides false

  do {
    set<string> names = new hashset<string>;
    foreach e in coun.getUniquelyNamedElements() {
      string n = e.name;
      if (names.contains(n)) {
        error "duplicate name " + n -> e;
      }
      names.add(n);
    }
  }
}
```

(c) check_ContainerOfUniqueNames

Figure A.3: TypeSystem Example.

A.3 Type System

The *type system* aspect is used to derive types or check types of concepts. Variables in the environment with type **SMType** are correctly resolved by defining proper type derivation and checking rules, so that they can be assigned by *sm_start*, and used in *sm_trigger*, *sm_terminate*, as well as passing around through variable assignments and function arguments as first-class citizens (Figure A.3a.

We use type-checking rules as an extension of *constraint* aspect, to impose type conformity (e.g., check whether the types of actual arguments match their declarations, or check conflict of unique names). For example. the type system of **Event** (Figure A.3b) checks

type matching for its event binding. Specifically, it makes sure that 1) the number of arguments in an event is the same as the number of arguments in the bound function; and 2) the type of each argument in the event is a sub-type of that of the corresponding argument in bound function. Another example is the type-system of **IContainerOfUniqueNames** (Figure A.3c), which checks conflict of unique names. We also use type-checking rules to make sure the execution of a big-step is not observable by the environment until the end of the big-step, by banning operations that may modify the status of the environmental in a state machine, as well as in functions that are called in a state machine.

A.4 Behaviour

The *behaviour* aspect of an interface concept is used to define abstract or concrete methods, whereas that of a concrete concept is used either to define functions that ease the task of retrieving information from the model by language creators, or to implement abstract functions required by the interface. For example, **EventCall** and **SMGenEvent** are used to generate in-event and internal-event of a state machine, respectively. They both implement the interface **ICallLike** so that type checking of arguments of function calls is performed automatically. Shown in Figure A.4, the behaviour aspect of **ICallLike** contains four abstract functions and two concrete functions. It also defines the *type system* aspect that performs type checking on arguments, based on the information from the six functions defined in its behaviour. Any concrete concept that implements **ICallLike** and the four abstract functions gains the benefit of automatic type checking of arguments.

```
concept behavior ICallLike {

  constructor {
    <no statements>
  }

  public virtual boolean hasEllipsis() {
    false;
  }

  public virtual abstract nlist<Expression> getActuals();

  public static virtual abstract node<LinkDeclaration> getActualsLink();

  public virtual abstract nlist<IArgumentLike> getFormals();

  public virtual abstract node<> getNodeForTypeCalc();

  public virtual string getFormalsAsString() {
    int formalCount = this.getFormals().size;

    string formalArgsString = "";
    foreach f in this.getFormals() {
      formalArgsString += f.type.getPresentation() + " " + f.name;
      if (f.index < formalCount - 1) {
        formalArgsString += ", ";
      }
    }
    if (this.hasEllipsis()) {
      formalArgsString += ", ...";
    }
    return formalArgsString;
  }

}
```

Figure A.4: Behaviour Example, mbeddr.**ICallLike**. By implementing this interface, benefits such as argument type checking are gained for **EventCall** and **SMGenEvent**.

Appendix B

Implementation: Template-based Generator

Here we discuss in detail about the usage of template-based generator in BSML-mbeddr. Readers who are interested in the complete course code of BSML-mbeddr generator may access our github page https://github.com/z9luo/BSML-mbeddr to install BSML-mbeddr and view the full version of our source code. Installation instructions are also on the web page.

B.1 Mapping configuration

The *generator* aspect of BSML-mbeddr contains a *mapping configuration* and a list of *reduction templates*. The mapping configuration is a container of generator rules, mapping label declarations and references to pre/post-processing scripts. It is the overall controller for the generation process.

The mapping configuration specifies a list of reduction rules, that apply corresponding reduction templates, such as reduce_StateMachine, reduce_Region, reduce_EventCall to given nodes. Some of the templates are in-line which are equivalent to normal templates. There are weaving rules that generates code from nothing. We have two weave templates weave_Common and weave_SM which is applies for each mbeddr program and each state machine, respectively, in order to generate auxiliary data structures and functions. Reduction and weaving templates are discussed in the following sections.

```
mapping configuration StateMachine2C
top-priority group    false

mapping labels:
  label terminate_event : SMGlobalDeclaration -> EnumLiteral

parameters:
  << ... >>

is applicable:
  <always>

conditional root rules:
  << ... >>

root mapping rules:
  << ... >>

weaving rules:
  [concept     ImplementationModule                                          ] -->
  [inheritors false
  [condition  (genContext, node, operationContext)->boolean {
                 node.children.ofConcept<SMGlobalDeclaration>.isNotEmpty;
              }
     [Weave_Common
     [context : (genContext, operationContext, node)->node<> {
                   genContext.get copied output for (node);
                }

  [concept     SMGlobalDeclaration] -->
  [inheritors false
  [condition   <always>
     [Weave_StateMachine(  (genContext, node, operationContext)->node<StateMachineConfigItem> {          )
                             BCHelper.findBCConfigItem(genContext.inputModel, genContext, "BSML/main.StateMahcine2C",
                                  concept/StateMachineConfigItem/, "") : StateMachineConfigItem;
                           }
     [context : (genContext, operationContext, node)->node<> {
                   genContext.get copied output for (node.ancestor<concept = ImplementationModule>);
                }

reduction rules:
  [concept     SMGlobalDeclaration] --> reduce_StateMachine
  [inheritors false                        (  (genContext, node, operationContext)->node<StateMachineConfigItem> {     )
  [condition   <always>                         BCHelper.expectBCConfigItem(genContext.inputModel, genContext,
                                                     "BSML-mbeddr/main@generator", concept/StateMachineConfigItem/) :
                                                     StateMachineConfigItem;
                                                }

  [concept     SMTrigger] --> reduce_SMTrigger
  [inheritors false
  [condition   <always>
```

Figure B.1: Mapping Configuration

BSML-mbeddr use only one label: *terminate_event*. It tracks a special "terminate" Event to its EnumLiteral value in the generated code. With this label, template reduce_SMTerminate is able to get the enum value of the "'terminate" event, and generate code that send it to a state-machine instance and expect the state machine to terminate after processing the "terminate" event.

We use a preprocess_script to make sure the semantic configuration exists in the mbeddr program before code generation.

```
[concept     SMStart ] --> reduce_SMStart
 inheritors false
 condition   <always>]

[concept     SMTerminate] --> reduce_SMTerminate
 inheritors false
 condition   <always>   ]

[concept     SMType  ] --> content node:
 inheritors false          dummy                                constraints
 condition   <always>]     model    BSML.generator.template.main   imports     Generator_Header

                           <TF [ ->$[SMHandle] ] TF>* sm_handle;

[concept     EventArgRef] --> reduce_EventArgRef
 inheritors false
 condition   <always>   ]

[concept     LocalVarRef
 inheritors false
 condition   (genContext, node, operationContext)->boolean {
               node.var.ancestor<concept = IStateMachine>.isNotNull && (
                   node.ancestor<concept = AbstractBlock>.isNotNull || node.ancestor<concept = ITransition>.isNotNull
                   || node.ancestor<concept = StartBigStepBlock>.isNotNull ||
                   node.ancestor<concept = EndBigStepBlock>.isNotNull) && (
                   node.var.ancestor<concept = AbstractBlock>.isNull && node.var.ancestor<concept = ITransition>.isNull
                   && node.var.ancestor<concept = StartBigStepBlock>.isNull &&
                   node.var.ancestor<concept = EndBigStepBlock>.isNull);
             }                                                                                               ]
    --> [ case: (genContext, node, operationContext)->boolean {
                node.ancestor<concept = AssignmentExpr>.isNotNull && node.ancestor<concept = AssignmentExpr>.
                    getLValue().descendants<concept = LocalVarRef, +>.contains(node);
            }
          content node:
          dummy                                     constraints
          model    BSML.generator.template.main     imports     Generator_Header

          void f(SMStruct* snapshot_cur) {
            <TF [(snapshot_cur->->$[__cur_state]) ] TF>;
          } f (function)

        default:
          content node:
          dummy                                     constraints
          model    BSML.generator.template.main     imports     Generator_Header

          void f(SMStruct* snapshot_read) {
            <TF [(->$[snapshot_read]->->$[__cur_state]) ] TF>;
          } f (function)                                                                          ]

[concept     EventCall] --> reduce_EventCall
 inheritors false
 condition   <always> ]
```

Figure B.2: Mapping Configuration (continue)

B.2 weave_Common

Template weave_Common is applied once for each mbeddr program, which generates:

- A message list containing a list of messages for debugging purpose, such as *big_step_start*, *small_step_start*, *transition_executed*, *transitions_enabled* (with the number of enabled transitions as arguments).
- A list of auxiliary data structures functions that are shared among different state machines. This includes data structures for Event, EnvInput, SMHandle, Binding Call as well as functions for creating event instances and for reset/free a pointer array.

B.3 weave_StateMachine

Template weave_StateMachine is applied once for each state machine, which generates data structures that cannot be shared among different state machines, including:

- Enum types for states, regions, events, and transitions. A special "terminate" event enum is also generated and labelled here.
- Struct type SMStructStatic that holds static information for a state machine that can be retrieved at run-time in constant time complexity, such as the information whether two given regions are orthogonal, whether a given event is used as triggering event or generated in an action/block.
- Struct SMStruct that stores all the run-time information of a state-machine instance. Each state-machine instance has its own SMStruct instance so that their concurrent executions do not interfere with each other.
- Struct Transition that stores the static and run-time information of a transition instance.
- Some auxiliary functions that manipulate on the above data structures.

```
<TF messagelist sm_msg {                                                                    TF>
      message SM_initialized(string name) INFO: initialize state machine  (inactive)
      message transition_executed(string trans_id, string from, string to) INFO:    (inactive)
      message trans_enabled(uint32 num) INFO: number of trans enabled in the small step  (inactive)
      message start_big_step( ) INFO:    (inactive)
      message end_big_step( ) INFO:    (inactive)
      message start_small_step( ) INFO:    (inactive)
      message end_small_step( ) INFO:    (inactive)
      message other(string info) INFO: other info  (inactive)
    }

<TF void reset_pointer_array(void** data, uint32 length) {                      TF>
      //[ scan the pointer array and set non-null pointers to null ]
      for (uint32 index = 0; index < length; index++) {
        if (data[index] != null) {
          data[index] = null;
        } if
      } for
    } reset_pointer_array (function)
<TF void free_pointer_array(void** data, uint32 length) {                   TF>
      //[ scan the pointer array and free() non-null pointers ]
      for (uint32 index = 0; index < length; index++) {
        if (data[index] != null) {
          free(data[index]);
        } if
      } for
    } free_pointer_array (function)
<TF struct _Event {                     TF>
      uint32 type;
      void** args;
      boolean has_in_syntax;
      boolean has_out_syntax;
      boolean has_renzd_syntax;
      gint small_step_count;
    };
<TF typedef _Event as Event; TF>
<TF Event* create_event(uint32 type, void** args, boolean has_in_syntax, boolean has_out_syntax,  TF>
        boolean has_renzd_syntax, gint small_step_count) {
      Event* ret = ((Event*) malloc(sizeof[Event]));
      ret->type = type;
      ret->args = args;
      ret->has_in_syntax = has_in_syntax;
      ret->has_out_syntax = has_out_syntax;
      ret->has_renzd_syntax = has_renzd_syntax;
      ret->small_step_count = small_step_count;
      return ret;
    } create_event (function)
<TF typedef GPtrArray as EnvInput; TF>

<TF exported struct SMHandle {  TF>
      GThread* instance;
      GAsyncQueue* queue;
    };
```

Figure B.3: weave_Common

```
<TF [typedef (gpointer)=(gpointer) as SMStartRef; ] TF>
<TF [SMHandle* create_sm_instance(SMStartRef sm_start) {            ] TF>
       SMHandle* ret = ((SMHandle*) malloc(sizeof[SMHandle]));
       ret->queue = g_async_queue_new( );
       ret->instance = g_thread_new("", sm_start, ret->queue);
       return ret;

     } create_sm_instance (function)
<TF [exported typedef (void**)=(void) as BindingRef; ] TF>
<TF [exported struct _BindingCall { ] TF>
       BindingRef func;
       void** args;
       gint small_step_count;
       boolean event_as_trigger;
     };
<TF [exported typedef _BindingCall as BindingCall; ] TF>

void func(int8 x) {

} func (function)
<TF [ $LOOP$[void $[proxy_func](void** args) {                       ]] TF>
              ->$[func]($LOOP$[*(($COPY_SRC$[int8]*) *(args + $[1]))]);
              $LOOP$[free((($COPY_SRC$[int8]*) *(args + $[1])));]
              free(args);
            } proxy_func (function)
<TF [enum Dummy_Enum { ] TF>
       a_dummy_enum;
     }
```

Figure B.4: weave_Common (continue)

B.4 reduce_StateMachine

Template reduce_StateMachinei is the most important template for BSML-mbeddr, it generates:

- Initialization function and entry function for each state and region.
- Action function for each transition with an action block.
- *sm_start*() function that is the entry function of a state-machine instance thread. It keeps listening to the input queue, retrieving environmental input, and calling *execute_big_step*() for each environmental input.
- *execute_big_step*() function that process a big-step. It contains a while-loop handling small-steps. Within a small-step, template reduce_Region is called for the *main* region that generates code for identifying all the enabled transitions. Then it generates function *is_consistent*() that is called for calculating the result set. All transitions in the result set are executed by calling function *handle_transition*(), which is also generated by template reduce_StateMachine. Function *handle_event_lifeline*() is called

```
<TF enum $[EventsEnum] {                                  TF>
      $LOOP$[$[event]; ]
      $MAP_SRC$ terminate_event[ $[sm_terminate]; ]
   }
<TF enum $[StatesEnum] {  TF>
      $LOOP$[$[state]; ]
      $IF$[no_states; ]
   }
<TF enum $[RegionsEnum] {  TF>
      $LOOP$[$[region]; ]
      $IF$[no_regions; ]
   }
<TF enum $[TransEnum] {         TF>
      $LOOP$[$[a_trans_enum]; ]
      $IF$[no_transitions; ]
   }

<TF struct $[SMStructStatic] {                          TF>
      boolean[$[20]][$[20]] are_regions_orthogonal;
      gint[$[20]][$[20]] compare_elements_hier_parent;
      gint[$[20]][$[20]] compare_elements_hier_child;
      boolean[$[20]][$[20]] trans_interrupt;
      boolean[20] event_as_action;
      boolean[20] event_as_trigger;
   };

<TF struct $[SMStruct] {                         TF>
      $LOOP$[StatesEnum $[__cur_state]; ]
      $LOOP$[$LOOP$[$COPY_SRC$[int8 ] $[x]; ]]
      ->$[Event]*[$[20]] present_events;
      GPtrArray* bindings;
      boolean[$[20]] region_disabled;
      SMStructStatic* static_info;
      gint small_step_count;
   };

<TF [typedef (SMStruct*, SMStruct*, SMStruct*)=(void) as $[BlockRef]; ] TF>

<TF [struct $[_CurStateSet] {              TF>
      StatesEnum* __cur_state;
      StatesEnum new_cur_state_value;
   };
<TF [typedef _CurStateSet as $[CurStateSet]; ] TF>
<TF CurStateSet* $[create_cur_state_set](StatesEnum* __cur_state, StatesEnum new_value) {  TF>
      CurStateSet* cur_state_set = ((CurStateSet*) malloc(sizeof[CurStateSet]));
      cur_state_set->__cur_state = __cur_state;
      cur_state_set->new_cur_state_value = new_value;
      return cur_state_set;
   } create_cur_state_set (function)
```

Figure B.5: weave_StateMachine

```
<TF [struct $[_Transition] {                                   ] TF>
      TransEnum trans_enum;
      GPtrArray* cur_state_sets;
      string trans_name;
      string source_state;
      string target_state;
      BlockRef action_ref;
      GPtrArray* entry_refs;
      RegionsEnum source_region_enum;
      RegionsEnum target_region_enum;
      RegionsEnum arena_enum;
      boolean enter_stable_state;
      guint priority;
      boolean[$[10]] regions_need_disabled;
      guint textual_order;
      gint hier_compare_enum;
      boolean is_interrupted;
    };
<TF [typedef _Transition as $[Transition]; ] TF>
<TF [ Transition* $[create_trans](TransEnum trans_enum, string trans_name, string source_state, string target_state, ] TF>
        BlockRef action_ref) {
      Transition* trans = ((Transition*) malloc(sizeof[Transition]));
      memset(trans, 0, sizeof[Transition]);
      trans->cur_state_sets = g_ptr_array_new_with_free_func(:free);
      trans->entry_refs = g_ptr_array_new( );
      trans->trans_enum = trans_enum;
      trans->trans_name = trans_name;
      trans->source_state = source_state;
      trans->target_state = target_state;
      trans->action_ref = action_ref;
      trans->priority = 0;
      trans->hier_compare_enum = 0;
      trans->is_interrupted = false;
      return trans;
    } create_trans (function)
<TF [void $[free_trans](void* trans) {        ] TF>
      Transition* t = ((Transition*) trans);
      g_ptr_array_free(t->cur_state_sets, true);
      g_ptr_array_free(t->entry_refs, false);
      free(trans);
    } free_trans (function)
```

Figure B.6: weave_StateMachine (continue)

```
<TF gint $[compare_trans](gconstpointer trans1, gconstpointer trans2, gpointer snapshot_static_info) {   TF>
   ->$[Transition]* t1 = *((->$[Transition]**) trans1);
   ->$[Transition]* t2 = *((->$[Transition]**) trans2);
   ->$[SMStructStatic]* static_info = ((->$[SMStructStatic]*) snapshot_static_info);
   if (t1 == t2) {
     return 0;
   } if
   gint ret = 0;
   //[ EXPLICIT priority ]
   $IF$[{
       if (t1->priority == t2->priority) {
         ret = ((t1->textual_order > t2->textual_order))?(1):(-1);
       } else if (t1->priority == 0) {
         ret = 1;
       } else if (t2->priority == 0) {
         ret = -1;
       } else {
         ret = ((t1->priority > t2->priority))?(1):(-1);
       }
     }]
   //[ HEIRARCHICAL priority ]
   $IF$[ret = static_info->compare_elements_hier_parent[t1->hier_compare_enum][t2->hier_compare_enum]; ]
   $IF$[ret = static_info->compare_elements_hier_child[t1->hier_compare_enum][t2->hier_compare_enum]; ]
   return ret;
 } compare_trans (function)
<TF [ $INCLUDE$gen_consistency [] ] TF>
```

Figure B.7: weave_StateMachine (continue)

at the end of a small-step to deactivate event instances that should not exist in the following small-steps.

- Functions *is_consistent*(), *handle_transition*() and *handle_event_lifeline*() as mentioned above.

B.5 reduce_Region

Template recude_Region generates:

- A switch-case statement for each sub-region within its sub-states, where template reduce_Region is recursively called.
- A switch-case statement for collecting enabled transitions in the current region, where template reduce_Transtition is called for each transition whose source state is contained in the current region.

```
<TF [ $CALL$gen_init_and_entry generate entry and init function for state machine [] ] TF>
void handle_event_lifeline(SMStruct* snapshot_cur, gint small_count, boolean last_small) {

} handle_event_lifeline (function)
<TF [ $INCLUDE$gen_execute_big_step [] ] TF>
<TF [ gpointer $[sm_start](gpointer queue) {                                                   ] TF>
    inactive report(0) sm_msg.other("$[]") on/if;
    ->$[SMStruct] snapshot_big;
    ->$[SMStruct] snapshot_small;
    ->$[SMStruct] snapshot_cur;
    //[ three snapshots share the same "bindings" (GPtrArray) ]
    ->$[init_snapshot](&snapshot_big);
    g_async_queue_ref(((GAsyncQueue*) queue));
    while (true) {
      ->$[EnvInput]* input = ((->$[EnvInput]*) g_async_queue_pop(((GAsyncQueue*) queue)));
      //[ terminate state-machine ]
      boolean term = false;
      for (i ++ in [0..input->len[) {
        if (((->$[Event]*) g_ptr_array_index(input, i))->type == enum2int(->$[sm_terminate])) {
          term = true;
          break;
        } if
      } for
      if (term) {
        char* retval = "terminate event received. state machine terminated successfully.";
        g_async_queue_unref(((GAsyncQueue*) queue));
        g_ptr_array_free(snapshot_big.bindings, false);
        return retval;
      } if
      for (i ++ in [0..input->len[) {
        ->$[Event]* e = ((->$[Event]*) g_ptr_array_index(input, i));
        snapshot_big.present_events[e->type] = e;
      } for
      ->$[execute_big_step](&snapshot_big, &snapshot_small, &snapshot_cur);
    } while
  } sm_start (function)
```

Figure B.8: reduce_StateMachine

```
exported void SM_small_step(SMStruct* snapshot_big, SMStruct* snapshot_small, SMStruct* snapshot_cur) {
  GPtrArray* enabled_transitions;
  boolean b = false;
  gint rendezvous_count = 0;

  <TF [{
        $IF$[//[ collect enabled transitions in subregions ] ]
        $IF$[switch (snapshot_small->->$[__cur_state]) {
              $LOOP$[case ->$[a_state_enum]: {
                        $LOOP$[$CALL$reduce_Region [ int16 gen_switch_for_sub_regions; ]]
                        break;
                     } case
              ]
            } switch
        ]
        //[ collect enabled transitions in current region ]
        switch (snapshot_small->->$[__cur_state]) {
          $LOOP$[case ->$[a_state_enum]: {
                   //[ collect enabled transitions ]
                   $LOOP$[if ($MAP_SRC$[rendezvous_count <= 1]) {
                            boolean run_time_trigger_has_rende = false;
                            $CALL$reduce_Transition [ int8 x; ]
                          } if
                   ]
                   break;
                 } case
          ]
        } switch

      }] TF>
} SM_small_step (function)
```

Figure B.9: reduce_Region

B.6 reduce_Transition

Template reduce_Transition generates:

- If-else statements that checks whether the transition should be enabled. This includes checking whether all triggering events are present (or absence if negated) and the guard condition is true.
- statements that generate transition instance if it is enabled. This includes filling static and run-time information in the instance for usage by consistency checking and transition execution. The transition structure also contains information that specifies the status change of the state machine and entry blocks to be executed as effect of executing the transition.

B.7 Miscellaneous

reduce_EventCall It generates statements that create an event instance, with proper type and actual arguments. If the generated event is bound to a function, then a

```
<TF [if ($MAP_SRC$[snapshot_small->present_events[enum2int(->$[a_region_enum])] != null]) {                 TF>
    $IF$[if (snapshot_small->present_events[enum2int(->$[a_region_enum])]->has_renzd_syntax) { ]
          run_time_trigger_has_rende = true;
        } if
    $IF$[$CALL$reduce_Transition [if (true) { ]]
                                 } if
    $IF$[if ($COPY_SRC$[b]) {
          if (rendezvous_count <= 1 || run_time_trigger_has_rende) {
            ->$[Transition]* trans = ->$[create_trans](
                ->$[a_trans_enum], "$[trans_id]", "$[source]", "$[target]", :->$[action]);
            $LOOP$[{
                  //[ enter states/regions on the way from arena to target state; enter sibling regions
                      cascadely on the way. ]
                  $IF$[{
                        g_ptr_array_add(trans->cur_state_sets,
                            ->$[create_cur_state_set](&(snapshot_cur->->$[__cur_state]),
                            ->$[a_state_enum]));
                        g_ptr_array_add(trans->entry_refs, :->$[entry_state]);
                      }
                  $IF$[{
                        $LOOP$[g_ptr_array_add(trans->entry_refs, :->$[entry_state]); ]
                        g_ptr_array_add(trans->entry_refs, :->$[entry_state]);
                      }
                 }
            //[ enter the target state at last, cascadely ]
            g_ptr_array_add(trans->cur_state_sets,
                ->$[create_cur_state_set](&(snapshot_cur->->$[__cur_state]), ->$[a_state_enum]));
            g_ptr_array_add(trans->entry_refs, :->$[entry_state]);
            trans->priority = $[0];
            trans->source_region_enum = ->$[a_region_enum];
            trans->target_region_enum = ->$[a_region_enum];
            trans->arena_enum = ->$[a_region_enum];
            trans->enter_stable_state = $MAP_SRC$[false];
            //[ regions_need_skip stores the RegionEnum of regions need to be skipped for big-steo maximality if
                this transition is executed ]
            $LOOP$[trans->regions_need_disabled[->$[a_region_enum]] = true; ]
            trans->textual_order = $[1];
            g_ptr_array_add(enabled_transitions, trans);
            //[ hier_compare_enum stores the int value for state/target/scope enum (either StateEnum or
                RegionEnum.
                For hierarchical priority comparison ]
            $IF$[trans->hier_compare_enum = $[0] + enum2int(->$[a_state_enum]); ]
          } if
        } if
    } if
} SM small step (function)
```

Figure B.10: reduce_Transition

BindingCall instance is created as well, to delay the call of the bound function to the end of a big-step.

reduce_SMStart It generates statements that create an input queue and a thread that executes *sm_start*(). The result is stored in a SMHandle struct.

reduce_SMTrigger It generates statements that create an environmental input comprising the generated in-event instances. Then the environmental input is put into an input queue referred by the state-machine handle as specified in **SMTrigger**, as an expression. The generation of an in-event is similar to that of **EventCall**, but their search scopes are different.

reduce_SMTerminate It generates statements that send an "terminate" in-event to a state-machine instance, join the thread, and destroy the input queue.

reduce_LocalVar It recognizes and resolves reference to a state-machine variable (either for write or for read) onto its correct location in a SMStruct instance. This is implemented as an in-line template in the mapping configuration.

```
void binding(void** args) {

} binding (function)

exported void f(SMStruct* snapshot_big, SMStruct* snapshot_small, SMStruct* snapshot_cur) {
  <TF [{
        ->$[Event]* __event = null;
        {
          void** __args = null;
          $IF$[__args = ((void**) malloc($[2] * sizeof[void*])); ]
          $IF$[//[ initialized actual arguments ] ]
          $LOOP$[$COPY_SRC$[int8]* $[arg1] = (($COPY_SRC$[int8]*) malloc(sizeof[$COPY_SRC$[int8]])); ]
          $LOOP$[*->$[arg1] = $COPY_SRC$[0]; ]
          $LOOP$[__args[$[1]] = ->$[arg1]; ]
          __event = ->$[create_event](enum2int(->$[a_event_enum]), __args, $MAP_SRC$[false], $MAP_SRC$[false],
              $MAP_SRC$[false], snapshot_cur->small_step_count);

          $IF$[{
              //[ delay event binding call ]
              ->$[BindingCall]* call = ((->$[BindingCall]*) malloc(sizeof[->$[BindingCall]]));
              call->func = :->$[binding];
              call->args = __args;
              call->small_step_count = snapshot_cur->small_step_count;
              call->event_as_trigger = snapshot_cur->static_info->event_as_trigger[enum2int(->$[a_event_enum])];
              g_ptr_array_add(snapshot_cur->bindings, call);
            }]
        }
        if (snapshot_cur->present_events[__event->type] != null) {
          free(snapshot_cur->present_events[__event->type]);
        } if
        snapshot_cur->present_events[__event->type] = __event;
      }] TF>
} f (function)
```

Figure B.11: reduce_EventCall

```
gpointer sm_start(gpointer p) {
  SMHandle* s = <TF [ ->$[create_sm_instance](:->$[sm_start]) ] TF>;
  return null;
} sm_start (function)
```

Figure B.12: reduce_SMStart

```
void f() {
  SMHandle* handle;
  <TF [{
    $LOOP$[void** $[args] = null; ]
    $LOOP$[->$[args] = ((void**) malloc($[2] * sizeof[void*])); ]
    $LOOP$[{
             ->$[args] = ((void**) malloc($[2] * sizeof[void*]));
             void** tmp = ->$[args];
             $LOOP$[$COPY_SRC$[int8]* $[arg1] = (($COPY_SRC$[int8]*) malloc(sizeof[$COPY_SRC$[int8]])); ]
             $LOOP$[*->$[arg1] = $COPY_SRC$[0]; ]
             $LOOP$[tmp[$[1]] = ->$[arg1]; ]
           }]
    ->$[EnvInput]* input = g_ptr_array_new();
    $LOOP$[g_ptr_array_add(input, ->$[create_event](enum2int(->$[a_event_enum]), ->$[args], $MAP_SRC$[false],
               $MAP_SRC$[false], false, 0));]
    g_async_queue_push(($COPY_SRC$[handle])->queue, input);
  }]                                                                                                    TF>
} f (function)
```

Figure B.13: reduce_SMTrigger

```
void f() {
  SMHandle* smhandle;
  <TF [{
    ->$[SMHandle]* cur = $COPY_SRC$[smhandle];
    ->$[EnvInput]* input = g_ptr_array_new();
    g_ptr_array_add(input, ->$[create_event](enum2int(->$[a_event_enum]), null, true, false, false, 0));
    g_async_queue_push(cur->queue, input);
    gpointer retval = g_thread_join(cur->instance);
    g_async_queue_unref(cur->queue);
    if (retval != null) {
      inactive report(0) sm_msg.other(((char*) retval)) on/if;
    } if
    free(cur);
  }]                                                                                                    TF>
} f (function)
```

Figure B.14: reduce_SMTerminate

References

[1] O. Badreddin and T.C. Lethbridge. Model oriented programming: Bridging the code-model divide. In *Modeling in Software Engineering (MiSE), 2013 5th International Workshop on*, pages 69–75, May 2013.

[2] Omar Badreddin, Timothy C. Lethbridge, Andrew Forward, Maged Elaasar, Hamoud Aljamaan, and Miguel A. Garzon. Enhanced code generation from uml composite state machines. *Model-Driven Engineering and Software Development (MODELSWARD), 2014 2nd International Conference on*, pages 235–245, Jan 2014.

[3] Gérard Berry and Georges Gonthier. The esterel synchronous programming language: Design, semantics, implementation. *Sci. Comput. Program.*, 19(2):87–152, November 1992.

[4] Frederic Boussinot. Sugarcubes implementation of causality. Technical report, 1998.

[5] Barak Cohen and Shahar Maoz. Semantically configurable analysis of scenario-based specifications. *Fundamental Approaches to Software Engineering*, 8411:185–199, 2014.

[6] James B. Dabney and Thomas L. Harman. *Mastering SIMULINK*. Prentice Hall Professional Technical Reference, 2003.

[7] Shahram Esmaeilsabzali. *Prescriptive Semantics for Big-Step Modelling Languages*. PhD thesis, University of Waterloo, Waterloo, Ontario, Canada, 2011.

[8] Shahram Esmaeilsabzali and Nancy Day. Prescriptive semantics for big-step modelling languages. In *Proceedings of the 13th International Conference on Fundamental Approaches to Software Engineering*, FASE'10, pages 158–172, Berlin, Heidelberg, 2010. Springer-Verlag.

[9] Shahram Esmaeilsabzali, Nancy Day, Joanne M. Atlee, and Jianwei Niu. Deconstructing the semantics of big-step modelling languages. *Requirements Engineering*, 15(2):235–265, 2010.

[10] Shahram Esmaeilsabzali, Bernd Fischer, and Joanne M. Atlee. Monitoring aspects for the customization of automatically generated code for big-step models. In *Proceedings of the 10th ACM International Conference on Generative Programming and Component Engineering*, GPCE '11, pages 117–126, New York, NY, USA, 2011. ACM.

[11] Joeri Exelmans. Configurable semantics in the sccd statechart compiler. http://msdl.cs.mcgill.ca/people/joeri/files/semantic_options.pdf, 2014.

[12] Fathiyeh Faghih and Nancy Day. Mapping Big-Step Modeling Languages to SMV. *Grace Hopper Celebration of Women in Computing*, pages 1–57, 2011.

[13] Andrew Forward, Omar Badreddin, Timothy C. Lethbridge, and Julian Solano. Model-driven rapid prototyping with umple. *Softw. Pract. Exper.*, 42(7):781–797, July 2012.

[14] Martin Fowler. Language Workbenches: The Killer-App for Domain Specific Languages? 2004.

[15] M.A. Garzon, H. Aljamaan, and T.C. Lethbridge. Umple: A framework for model driven development of object-oriented systems. *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on*, pages 494–498, 2015.

[16] Nicolas Halbwachs. *Synchronous Programming of Reactive Systems*. Springer-Verlag, Berlin, Heidelberg, 2010.

[17] David Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3):231–274, June 1987.

[18] C. Huizing and R. Gerth. Semantics of reactive systems in abstract time. *Real-Time: Theory in Practice*, 600:291–314, 1992.

[19] Yun Lu, J.M. Atlee, N.A. Day, and Jianwei Niu. Mapping template semantics to smv. In *Automated Software Engineering, 2004. Proceedings. 19th International Conference on*, pages 320–325, Sept 2004.

[20] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Semantically configurable consistency analysis for class and object diagrams. In *Proceedings of the 14th International Conference on Model Driven Engineering Languages and Systems*, MODELS'11, pages 153–167, Berlin, Heidelberg, 2011. Springer-Verlag.

[21] Florence Maraninchi and Yann RéMond. Argos: An automaton-based synchronous language. *Comput. Lang.*, 27(1-3):61–92, April 2001.

[22] mbeddr Team. mbeddr C user guide. https://github.com/mbeddr/mbeddr.core/releases/download/0.8.1-EAP/mbeddr-userguide-0.8.1-EAP.pdf, 2014.

[23] Jianwei Niu, Joanne M. Atlee, and Nancy Day. Template semantics for model-based notations. *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, 29:149–158, 2003.

[24] OMG. Omg unified modeling language (omg uml). *Superstructure*, 2007.

[25] Vaclav Pech, Alex Shatalin, and Markus Voelter. Jetbrains mps as a tool for extending java. In *Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, PPPJ '13, pages 165–168, New York, NY, USA, 2013. ACM.

[26] Adam Prout, Joanne M. Atlee, Nancy Day, and Pourya Shaker. Code generation for a family of executable modelling notations. *Software and Systems Modeling*, 11(2):251–272, 2012.

[27] Daniel Ratiu, Markus Voelter, Zaur Molotnikov, and Bernhard Schaetz. Implementing modular domain specific languages and analyses. In *Proceedings of the Workshop on Model-Driven Engineering, Verification and Validation*, MoDeVVa '12, pages 35–40, New York, NY, USA, 2012. ACM.

[28] Christoph Rosenberger. Model checking for state machines with mbeddr and nusmv. http://mbeddr.com/files/modelcheckingforstate-machineswithmbeddrandnusmv.pdf, 2013.

[29] P. Shaker, J.M. Atlee, and Shige Wang. A feature-oriented requirements modelling language. In *Requirements Engineering Conference (RE), 2012 20th IEEE International*, pages 151–160, Sept 2012.

[30] Ali Taleghani and Joanne M. Atlee. Semantic variations among UML statemachines. In *Model Driven Engineering Languages and Systems*, volume 4199 of *Lecture Notes in Computer Science*, pages 245–259. Springer Berlin Heidelberg, 2006.

[31] Olivier Tardieu. A deterministic logical semantics for pure esterel. *ACM Trans. Program. Lang. Syst.*, 29(2), April 2007.

[32] F. Tomassetti, A. Vetro, M. Torchiano, M. Voelter, and B. Kolb. A model-based approach to language integration. In *Modeling in Software Engineering (MiSE), 2013 5th International Workshop on*, pages 76–81, May 2013.

[33] M. Voelter and E. Visser. Product line engineering using domain-specific languages. In *Software Product Line Conference (SPLC), 2011 15th International*, pages 70–79, Aug 2011.

[34] Markus Voelter. Language and ide modularization and composition with mps. In Ralf Lmmel, Joo Saraiva, and Joost Visser, editors, *Generative and Transformational Techniques in Software Engineering IV*, volume 7680 of *Lecture Notes in Computer Science*, pages 383–430. Springer Berlin Heidelberg, 2013.

[35] Markus Voelter, Daniel Ratiu, Bernhard Schaetz, and Bernd Kolb. Mbeddr: An extensible c-based programming language and ide for embedded systems. In *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity*, SPLASH '12, pages 121–140, New York, NY, USA, 2012. ACM.

[36] Michael von der Beeck. A comparison of statecharts variants. *Formal Techniques in Real-Time and Fault-Tolerant Systems*, 863:128–148, 1994.

[37] Tuba Yavuz-Kahveci and Tevfik Bultan. Specification, verification, and synthesis of concurrency control components. In *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '02, pages 169–179, New York, NY, USA, 2002. ACM.