BSML-mbeddr: Integrating Semantically Configurable State-Machine Models in a C Programming Environment

Zhaoyi Luo
University of Waterloo, Canada
zhaoyi.luo@uwaterloo.ca

Joanne M. Atlee
University of Waterloo, Canada
jmatlee@uwaterloo.ca

Abstract

In model-driven engineering, developers express their solutions in domain-specific modelling languages (DSLs) that support domain-specific abstractions. Big-Step Modelling Languages (BSML) is a family of extended state-machine DSLs for creating executable models that have a complex execution semantics. In this paper, we present BSML-mbeddr, which imbeds a large subset of BSML within the mbeddr C programming environment, thereby extending mbeddr with language constructs for extended, semantically configurable state-machines. We also report on three case studies that exercise the expressiveness of BSML-mbeddr, assess the integrability of BSML-mbeddr into mbeddr, and demonstrate the need to provide support for state-machine models with different execution semantics.

Categories and Subject Descriptors D.2.6 [*Software Engineering*]: Programming Environments - Integrated Environments; D.3.2 [*Programming Languages*]: Language Classifications - Specialized Application Languages

Keywords Domain-specific language, State-machine model, MPS, mbeddr, Language product line

1. Introduction

A *domain-specific language (DSL)* is a language that supports domain-specific abstractions, which allow domain experts to express their problems efficiently and effectively. *Extended state-machines* are cross-domain DSLs that are widely applied to interactive and reactive systems in multiple domains, such as network protocols and control systems of vehicles, elevators, and medical devices. However, modellers cannot agree on a single semantics for state-machine modelling languages. In fact, there is ample evidence that modellers want to use a wide variety of notations and semantics [18]. Moreover, depending on the domain, it can be significantly more concise and understandable to model behaviours in one state-machine semantics versus another semantics [7]. Modellers need to be able to choose a state-machine notation, especially semantic features, on the basis of the domain or even the problem being modelled.

Big-Step Modelling Languages (BSML) [6] is a family of state-machine modelling languages (e.g., UML StateMachines [16], Argos [14], Statecharts [11], Stateflow [4], etc.). In BSML, a model reacts to an environment 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 previous work by Esmaeilsabzali and Day [7], the semantic variation points of BSML have been systematically decomposed into several high-level, mostly orthogonal aspects, such as whether sets of transitions can execute concurrently, which variable values are used to evaluate expressions, how long events persist in a big-step, what priorities exist among transitions, and so on. By configuring the semantic aspects with predefined semantic options, one can create a wide variety of domain-specific notations.

In this paper, we present *BSML-mbeddr:* a state-machine modelling language with hierarchical states, concurrent regions, and configurable semantics, for which support has been implemented within mbeddr. mbeddr [21] is a DSL workbench that supports the incremental construction of modular DSLs on top of the C programming language. By extending mbeddr with BSML, we have created an environment in which the programmer can write normal C code mixed with interoperable state-machine models. The resulting $mbeddr\ program$ (i.e., mbeddr C plus state machines) is transformed into C code for execution.

The main contributions of our work are:

- We have built BSML-mbeddr: a non-trivial extension of the mbeddr ecosystem that allows programmers to create sophisticated state-machine models within mbeddr.
- We show the feasibility of integrating multiple kinds of state-machine modelling languages into a programming language environment, creating an environment where a

SLE'16, October 31 – November 1, 2016, Amsterdam, Netherlands
© 2016 ACM. 978-1-4503-4447-0/16/10...$15.00
http://dx.doi.org/10.1145/2997364.2997372

developer can create a program that intermixes C code with their choice of state-machine model.

- We conducted case studies that evaluate BSML-mbeddr with respect to its support for big-step semantics, hierarchical states and cross-hierarchy transitions, concurrent regions and inter-region communication, configurable semantics, and code-model interaction and integration.

The remainder of the paper is organized as follows. Section 2 provides an overview of mbeddr. Section 3 introduces the syntax of BSML-mbeddr. Section 4 describes the execution semantics and configurable semantics of BSML-mbeddr. Section 5 discusses the challenges we have encountered. Section 6 describes how BSML-mbeddr has been tested and evaluated through case studies. Section 7 summarizes related work and Section 8 concludes the paper.

2. Background of mbeddr

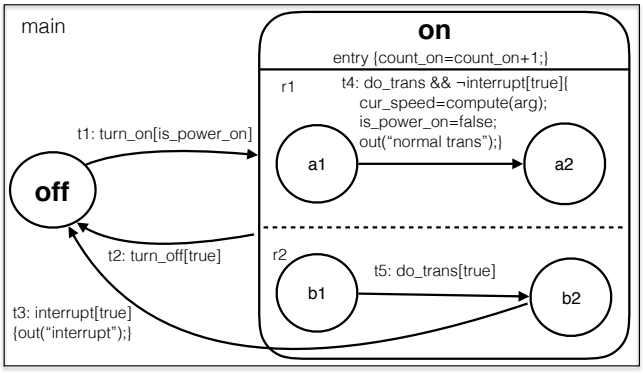
BSML-mbeddr is built on top of MPS and mbeddr[1]. The JetBrains Meta Programming System (MPS) is a projectional language workbench which provides a suite of language tools that support efficient definition, extension and use of DSLs [17]. In MPS, a language's $structure$ defines its abstract syntax. Any inputs from the programmer via the language's $editor$ are dynamically interpreted and built into an Abstract Syntax Tree (AST) that obeys the syntax rules as defined in the language's $structure$. The language's $generator$ resembles the code-generation phase of a traditional compiler – it transforms the abstract syntax into low-level textual code (i.e., the $base\ language$) for execution.

mbeddr [15][21] builds on MPS, by providing support for 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. By importing the desired DSLs into an mbeddr program, the developer can write an mbeddr C program augmented with additional language concepts and constructs that are native to the problem domain. mbeddr includes a predefined DSL for basic state machines that have no concurrency, no inter-machine communication, and a single (small-step) execution semantics (Section 7.2, Table 2).

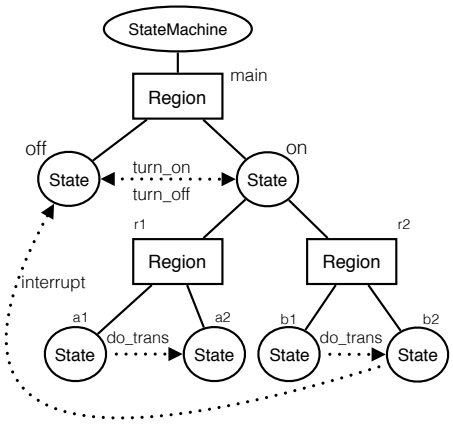
3. Syntax

We first give a light introduction to extended state-machine models, and then present details about BSML-mbeddr syntax in Section 3.1. A state-machine model contains states, events, regions and transitions. A $state$ can be $simple$ if it has no internal structure, or $composite$ if it contains one or more sub-$region$s each containing a sub state machine. A $transition$ represents an execution step in the model from a source state to a target state. A transition is labelled with an id, a triggering event, an optional guard condition, and actions that are performed if the transition is executed; these are described in detail below.

[1] We built BSML-mbeddr on mbeddr (nightly-94) and MPS (3.2).



(a) Model



(b) State Hierarchy

Figure 1: Illustration of an example model and its corresponding state hierarchy.

Figure 1a[2] depicts an example state machine that contains a $main$ region, within which there are two states – state *off* and state on. There are transitions between states *off* and on that are triggered by events $turn\_on$ and $turn\_off$. 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 do_trans.

Figure 1b represents the hierarchy of states and regions in the example machine. A transition may cross the boundaries of states or regions (called a $cross\text{-}hierarchy\ transition$), such as the transition *t3* from state $b2$ to state *off*.

3.1 State-Machine Elements in BSML-mbeddr

Figure 2 illustrates the BSML-mbeddr code for the example model in Figure 1. A $state\ machine$ (Line 1) is the root node of a state-machine's state hierarchy; 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 the sub-machines that reside

[2] For space reasons, we have omitted from the model the declarations of events and variables that are referred in the model.

in sibling regions. Each region contains one or more states, transitions, local events, local variables and other utility elements. Each region must designate an *initial state* (Line 2) that specifies the default current state whenever the machine's execution enters the region. A *state* that contains zero regions is a *simple* state (Line 17). A state that contains one or more regions is a *composite* state (Line 18). Each region contains a sub machine whose elements (states, transitions, variables) are declared within the region. A machine's state hierarchy forms a tree with interleaved layers of states and regions (Figure 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 an ancestor of the other (states on and $b1$ overlap). The *lowest common ancestor* of two states (regions) in the state hierarchy is the lowest node that is an ancestor of both states (regions) (state on is the lowest common ancestor of regions $r1$ and $r2$). Two states (regions) are *orthogonal* if they do not overlap and their least common ancestor is a state, not a region (states $a1$ and $b1$ are orthogonal). The *scope* of a transition is the lowest common ancestor – either a state or a region – of the source and target state (region $main$ is the scope of $t1$). The *arena* of a transition is the lowest region in the hierarchy that is the ancestor of both the source and target states (region $main$ is the arena of $t1$).

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 – it has a name and zero or more typed arguments. Additionally, an event can have an optional binding to a function that is defined in the environment[3]. There are three possible types of events: *in-event*, *out-event*, and *internal-event*. An in-event (Line 3-6) is expected to be generated by the environment of the state machine; an out-event (Line 7) delivers state-machine output to its environment via a call to a bound function in the environment; an internal-event is used for private communication among sub machines inside the model.

A *transition* (Line 13-15) specifies a trigger event or conjunction of trigger events, a guard condition, a source state, a target state, and an optional code block called *action*. A *trigger event* refers to an event declaration. A *guard condition* is a boolean expression that can refer to local variables and arguments of the triggering event, and can query the state of the environment. A transition is enabled if (1) all of its trigger events are present and all negated events are absent; and (2) the guard condition evaluates to true; and (3) source state is among the machine's current execution states. An *action* is a list of statements that is executed when the transition is executed. A state or region may optionally contain an *entry block* of statements that is executed every time its associated state or region is entered. An entry block (Line 19) or action (Line 24-26) may manipulate local variables, read arguments of triggering events, call functions, query the state of the environment, or generate events.

[3] *Environment* in this paper consistently refers to the mbeddr C environment that defines normal C functions, global variables, etc., with which the state machines interact.

```
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 do_trans(double arg);
7     event out(string msg) => handle_out;
8     boolean is_power_on = true;
9     double cur_speed = 0.0;
10    static int count_on = 0;
11    Status status = ON;
12    SM instance;
13    transition t1: turn_on[is_power_on] off -> on;
14    transition t2: turn_off[true] on -> off;
15    transition t3: 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 t4: do_trans && ¬interrupt[true] a1
                -> a2 {
24         cur_speed = compute(arg);
25         is_power_on = false;
26         out("normal trans"); }}
27     region r2 initial = b1 { ... }}}
28 }
29 int main() {
30   SM* m1 = sm_start(SM);
31   trigger_events(m1);
32   sm_trigger(m1, turn_on());
33   SM* m2 = sm_start(SM);
34   SM var = *m2;
35   trigger_events(m2);
36   sm_trigger(&var, do_trans(2.0), interrupt());
37   sm_terminate(&var);
38   sm_terminate(m1);
39   return 0;
40 }
41 void trigger_events(SM* arg) {
42   sm_trigger(arg, turn_on());
43   sm_trigger(arg, do_trans(2.0));
44   sm_trigger(arg, turn_off());
45 }
46 void handle_out(string arg) { printf("%s", arg); }
47 [isQuery]
48 double compute(double speed) {
49   //This function is called in an action to conduct
         computation or read status of environment.
50 }
```

Figure 2: Code for Example Model and Environment

3.2 Language Features

In this section we highlight language features that are introduced by BSML-mbeddr. Usages of language features in Figure 2 are referred to by line number.

- **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 during program execution, 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 out-event can be bound to a function that is called when the event is generated (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 do_trans is present and negated event $interrupt$ is absent.
- **Transition with Multiple Triggers** A transition may be triggered by conjunction of in-events, so that the transition is enabled only if all of its non-negated triggering events are present, and all of its negated triggering events are absent (Line 23).
- **Expression Language** BSML does not specify a concrete expression language. To fill the gap, BSML-mbeddr uses mbeddr C expressions and statements to express actions in transitions (Line 24-26) and entry blocks (Line 19). BSML-mbeddr actions can assign values to local variables, call functions, query the state of the environment, use control structures (if/while statements), use nested code blocks, etc.[4] In addition, language constructs from other DSLs that inherit from mbeddr C statements can also be used in BSML-mbeddr actions.
- **Cross-hierarchy Transition** BSML-mbeddr allows $cross\text{-}hierarchy$ transitions that cross the boundaries of states, including the proper exit and entry of concurrent regions, and the execution of a transition's action and the entry blocks of states or regions that the transition enters.
- **Big-step Start (End) Block** A big-step start (end) block belongs to the environment and it is executed immediately before (after) a big-step begins. It allows to access the status of state-machine models by reading state-machine variables, and perform any operations that are allowed in the environment code. It helps improve the integration of state machines and their environment.
- **State-Machine Variable** Variables defined in the environment code are not accessible inside the state machines, and vice versa. A state-machine variable can be $static$, meaning that it is initialized when the state-machine instance is created, and its value persists between entries to its defining state or region. 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), thereby counting the number of times state on is entered.
- **Function Call** We introduce $query$ functions as a language construct to facilitate a state-machine to query environment variables in the middle of a big-step, Any function that does not change the status of the environment can be tagged as $isQuery$ and thus can be called inside a state machine (e.g., inside a region, action, entry block, or guard condition). Such functions can be used to retrieve the current values of environment variables, or they can be used as helper functions within more complex computations (Line 24).
- **Name Scoping** A fully qualified name is assigned to each state-machine element or variable, and the search scope for variable references is defined modularly. For example, the code on Line 19 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.
- **Multiple Instances of a State-Machine Model** The modeller is able to create multiple concurrent instances of the same state-machine model (Line 30 and 33), and may send environment inputs to each of them without the machines interfering with each other, as long as the user keeps bound functions thread-safe (Line 32 and 36).
- **Input with Multiple Events** An environment input may contain instances of multiple in-events (Line 36) that simulate a "combo" action (e.g., a passenger of an elevator pushes multiple call buttons at the same time). However, an environment input is not allowed to generate multiple instances of the same in-event.

3.3 Interaction with Environment

BSML-mbeddr state machines are implemented as first-class citizens, so that they are defined on par with global variables, functions, structs, and enums, which form the environment of state machines[5].

Consider the environment code listed in Figure 2, Line 29-50. A state-machine model SM is instantiated and initialized with an $sm\_start(sm\_ref)$: launching a thread for the state-machine instance, creating an input queue for the machine, and returning a pointer to the machine (i.e., a sm_handle). Multiple instances of the same machine can be declared (Line 30 and 33). An $sm\_trigger(sm\_handle, event, ...)$ statement (Line 42-44) takes as arguments a pointer to a state-machine sm_handle, and a set of in-events; it generates an environment input comprising the set of in-event instances, and puts the environment input into the machine's input queue. A $sm\_terminate(sm\_handle)$ statement (Line 37-38) safely terminates a state-machine instance after all pending environment inputs are processed.

[4] We employ static analyses, described in Section 4.3, to prohibit statements from changing the values of environment variables.

[5] This paper describes the result of integrating BSML into mbeddr. Anyone who is interested in the details of design, implementation, test cases, or case studies may look at Zhaoyi's thesis [13] and BSML code base `https://github.com/z9luo/BSML-mbeddr`.

Because variables of a state-machine type are implemented as first-class citizens, they can be assigned to other variables (Line 34), returned from functions, or passed as arguments (Line 35). BSML-mbeddr imposes strict constraints and type-checking rules to ensure that (1) variables whose types are different kinds of state-machines cannot be assigned to each other, nor to variables that are not of state-machine types; (2) $sm\_start(sm\_ref)$ can be assigned only to a variable of type 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 model, and all $event$ arguments must be events declared in the state-machine model that sm_handle points to; (4) arguments to in-events, if any, must be provided and their types must match the declared types; and so on.

Delivering output from a state machine to its environment is achieved through event binding. In our example, event out is bound to function $handle\_out()$ (Line 7), which means that $handle\_out()$ is called whenever event out is generated and determined to be an out-event[6].

4. Semantics

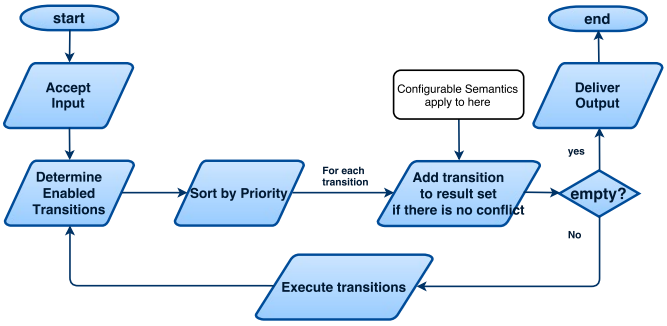


Figure 3: BSML-mbeddr Execution Semantics.

4.1 Execution Semantics

In BSML-mbeddr, a model's execution step starts with input from the model's environment. The model responds with a $big\text{-}step$, comprising a sequence of $small\text{-}steps$, each of which executes a set of transitions. Specifically, a big-step in BSML-mbeddr (shown in Figure 3) starts by accepting an environment input. Then a sequence of small-steps executes. Within each small-step, (1) the set of enabled transitions is identified, (2) the enabled transitions are sorted according to their priority, and (3) a maximal, consistent subset of highest-priority enabled transitions (called the $result\ set$) is deduced through a greedy approach – each enabled transition, in decreasing order of priority, is considered for inclusion in the result set and is added if and only if the transition does not conflict with any transition that is already in the result set, regarding to the semantic configuration. A small-step ends by executing all transitions in the result set and by calculating the new status of the state machine. At the end of the $big\text{-}step$, the model's outputs are delivered to the environment. If an environment input enables no transitions, then the result set is empty and the big-step ends without performing any small-steps. The difference of semantics between BSML [5] and BSML-mbeddr is that in BSML semantics, *all* possible maximal, consistent, highest-priority result sets are computed, from which one is arbitrarily selected. In contrast, BSML-mbeddr computes a single result set. In [13], we prove that the result set constructed by the greedy process of BSML-mbeddr is one of the result sets constructed by the BSML execution semantics.

[6] Whether an event is determined to be an out-event depends on the semantic aspect External Output Event (Section 4.2.6).

4.2 Configurable Semantics

In this section we introduce the configurable execution semantics of BSML-mbeddr in terms of semantic $aspect$s (i.e., variation points) and their $options$. To ease the presentation of descriptions, we present language syntax in **bold**, semantic aspects in font Sans Serif, and semantic options in font SMALL CAP. Unless otherwise noted, the semantic aspects and options originate from Esmaeilsabzali and Day's work on BSML semantic deconstruction [5][7].

We implemented configurable semantics using mbeddr's native support for program configuration. Each mbeddr program has a configuration file, within which the developer chooses an option for each semantic aspect. BSML-mbeddr's $generator$ specifies how executable code is generated according to the semantic configuration.

4.2.1 Big-step Maximality

The semantic aspect Big-step Maximality determines when a big-step ends. In option TAKE MANY, a big-step ends when no more transitions can be executed. In TAKE ONE, an executing transition inhibits other transitions in overlapping arenas from executing in the same big-step. In option SYNTACTIC, there is a syntactic language feature to tag a state as being **stable**: when an executing transition enters a stable state, no other transitions in overlapping arenas can execute in the same big-step.

4.2.2 Concurrency

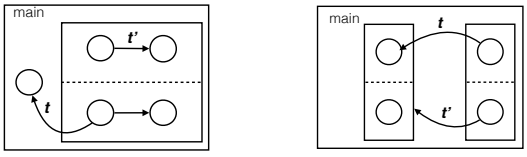
The semantic aspect Concurrency determines whether multiple transitions can execute in the same small-step. In option SINGLE, only one transition can execute in a small-step, whereas in option MANY, multiple transitions can execute in a small-step.

4.2.3 Consistency

The semantic aspect Consistency applies only if the Concurrency option is MANY – it determines which transitions can execute together in the same small-step. In option ARENA ORTHOGONAL, the transitions' arenas must be orthogonal in order for them to execute in the same small-step; whereas in option SOURCE-TARGET ORTHOGONAL, the transitions' source states and target states must be pairwise orthogonal.

Table 1: BSML-mbeddr Semantic Aspects and Options.

Aspect	Semantic Option	Definition
Big-Step Maximality	TAKE MANY	A big-step ends only when no more transitions can be executed.
	TAKE ONE	An executing transition inhibits other transitions in overlapping arenas from executing in the same big-step.
	SYNTACTIC	When an executing transition enters a stable state, no other transitions in overlapping arenas can execute in the same big-step.
Concurrency	SINGLE	Only one transition can execute in a small-step.
	MANY	Multiple transitions can execute in a small-step.
Small-Step Consistency	ARENA ORTHOGONAL	Two transitions whose arenas are orthogonal can execute in the same small-step.
	SOURCE-TARGET ORTHOGONAL	Two transitions whose source states and target states are pairwise orthogonal can execute in the same small-step.
Preemption	NON-PREEMPTIVE	If an interrupting and its interrupted transitions are executed in the same small-step, both their actions are executed. The state machine enters the target state of the interrupting transition.
	PREEMPTIVE	The interrupting and interrupted transitions cannot execute in the same small-step.
Event Lifeline	PRESENT IN NEXT SMALL	A generated event is present only in the next small-step.
	PRESENT IN REMAINDER	A generated event is present for the rest of the big-step.
External Input/Output Event	SYNTACTIC	In-events (out-events) are determined syntactically.
	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.
	HYBRID	An event received at the start of a big-step and never generated in the model is determined to be an in-event. An event generated in the last small-step and not a triggering event for any transition in the model is determined to be an out-event.
Memory Protocol	BIG STEP	Values of variables are read from the start of the big-step.
	SMALL STEP	Values of variables are read from the start of the current small-step.
Priority	EXPLICIT	Explicit priority is assigned to each transition.
	HIERARCHICAL	The priority of transition is implicitly inferred from the state hierarchy.



(a) **(b)**

Figure 4: Two ways in which transition t interrupts t'.

4.2.4 Preemption

The semantic aspect Preemption applies only if the Concurrency option is MANY; it determines whether an interrupting transition can preempt the interrupted transition. A transition t $interrupts$ transition t' if their source states are orthogonal, both t and t' are enabled, but t's target state determines the machine's next state if selected for execution – either because t exits t''s arena (Figure 4a) or because t's target state is a descendant of t''s target state (Figure 4b)).

In option NON-PREEMPTIVE, the actions of both the interrupting and interrupted transitions are executed, but the state machine enters the target state of the interrupting transition. In option PREEMPTIVE, only one of the transitions is selected to be executed[7]. Intuitively, the NON-PREEMPTIVE option allows the preempted transition to finish its "last wish" actions, whereas the PREEMPTIVE option does not.

4.2.5 Event Lifeline

The semantic aspect Event Lifeline determines how long a generated event remains $present$ in a big-step and able to continue triggering transitions. In option PRESENT IN REMAINDER, the generated event is present in all future small-steps of the current big-step. In option PRESENT IN

[7] Note that the semantics of Preemption itself does not impose any bias as to which transition is selected to be executed. To impose bias towards the interrupting rather than its interrupted transition, the modeller may use negated triggering events or explicit priority.

NEXT SMALL, the generated event is present only in the next small-step. The modeller may specify a distinct Event Lifeline option for in-events, out-events, and internal-events.

In addition, BSML-mbeddr supports rendezvous communication for events that are syntactically tagged $rendezvous$. When a rendezvous event is generated, it is present and able to trigger other transitions only in the same small-step in which the event is generated[8].

4.2.6 External Event

The determination of whether an event is an in-event or out-event depends on semantic aspects. An event that is neither an in-event nor an out-event is treated as an internal-event.

For aspect External Input Event, the option SYNTACTIC applies when the modeller is responsible for tagging in-events. Option RECEIVED IN FIRST SMALL applies when in-events are inferred to be those events received at the start of a big-step (i.e., generated by the environment, or received from the input queue). In the HYBRID option, an event that is received at the start of a big-step and is never generated in any transition or entry block is determined to be an in-event.

For aspect External Output Event, the option SYNTACTIC treats any event that is syntactically bound to a function as an out-event. In option GENERATED IN LAST SMALL, events that are generated in the last small-step are inferred to be out-events. In the HYBRID option, an event is inferred to be an out-event only if it is generated in the last small-step and is not a triggering event for any transition in the model.

4.2.7 Memory Protocol

The semantic aspect Memory Protocol determines which variables values are used to evaluate guard conditions and assignment expressions. In option BIG STEP, expressions are evaluated using variable values from the start of the big-step; in option SMALL STEP, expressions are evaluated using values from the start of the current small-step.

4.2.8 Priority

The Priority semantic aspect determines the order in which enabled transitions are considered for inclusion in the $result\ set$ (Section 4.1). The option EXPLICIT applies when the modeller indicate priority by syntactically assigning a positive integer to a transition. An unassigned priority is effectively an infinite value, 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 first has higher priority.

In option HIERARCHICAL, transition priority is determined implicitly by the state hierarchy: (1) on the Basis of the transitions' SOURCE, TARGET, or SCOPE; (2) according to a Scheme that gives priority to PARENT or CHILD states or scopes. For example, our default option is SCOPE-PARENT, which gives higher priority to a transition whose scope is the parent (ancestor) of the scope of other transitions[9].

4.2.9 Other Semantic Options

There are three categories of BSML semantic options that we did not implement in BSML-mbeddr: (1) A few options require dataflow analysis over a full big-step to determine if a transition is enabled in a small-step. We deemed these options to be too computationally expensive to implement. (2) $Combo\text{-}steps$ add further structure to a big-step, such that a big-step comprises a sequence of combo-steps, each of which comprises a sequence of small-steps. We omitted combo-step options from BSML-mbeddr because combo-steps are rarely used and their semantic options are similar to already-supported options. (3) $Components$ decompose a state-machine into encapsulated sub machines that communicate with each other through interface events and variables. Because the semantic options for inter-component communication are similar to the semantic options for communication among regions, we chose not to implement them.

4.3 Static Analysis

Making use of MPS $constraint$ and $type\text{-}system$ language aspects, we have embedded within BSML-mbeddr a number of static analyses that aim to ensure that BSML-mbeddr code is correct by construction. Specifically, static analyses check:

1. Consistency rules on the state-machine hierarchy. For example, states, entry blocks, state-machine variables, and transitions can be defined only within their parent region; and actions can be defined only within transition or entry-block definitions.
2. Type-checking on state-machines, to ensure that state-machine instances are used correctly as first-class citizens in the environment code (i.e., are assigned to state-machine variables of the same type, passed as arguments to parameters of the same type, addresses of and dereferences are assigned to variables of the correct type, etc.).
3. Type-checking rules on the arguments of sm_start, $sm\_trigger$, and $sm\_terminate$, to ensure that state machines interact with the environment code correctly.
4. Other type-checking rules to ensure that the guard conditions of transitions are of boolean type; the number and types of arguments of bound functions match those of the corresponding event declarations; etc.
5. Scope rules on references to declared language constructs, to ensure that only those that are visible in the

[8] In BSML, rendezvous communication is supported with the option PRESENT IN SAME; whereas in BSML-mbeddr, rendezvous is always enabled and only for syntactically tagged $rendezvous$ events.

[9] If the Basis of one transition is neither a descendant nor an ancestor of the Basis of the other transition, we resolve the nondeterminism as follows: the lowest common ancestor of the transitions' Basis is identified; and the transitions are prioritized according to the textual order in which the ancestors of their Basis states or scopes are declared within the lowest common ancestor.

current scope can be referred to. For example, $sm\_trigger$ can only trigger events that are actually defined in the target state machine and are determined as in-events.

6. Constraints on the expression language, to enforce big-step semantics. For example, expressions cannot change environment variables within a big-step. Thus expressions cannot modify global variables, assign values to dereferenced pointers, or perform self-mutation operations; and function calls in expressions must be queries.

5. Challenges

In this section we discuss issues we have encountered from our experiences building BSML-mbeddr. Much of the actual engineering was done in previous work, in defining the BSML family and expressing the family's semantics in terms of semantic variation points and options. But in order to realize BSML as an executable language, we had to make engineering decisions and fill in gaps, including (1) handling multiple, simultaneous instances of events; (2) communication of model outputs to the environment; (3) thread safety; (4) consistency between BSML syntax and semantic options (especially when configurations evolve); and (5) constructing a concrete expression language and action language.

5.1 Consistency between Language Features and Configurable Semantics

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 conflicts with the existing syntax and semantics of BSML. With configurable semantics, we also need to decide whether its semantics shall be fixed or configurable.

For example, we introduce the language feature $event\ binding$ to deliver outputs from a state-machine model to its environment. Before adding this feature, we asked ourselves: Shall $event\ binding$ apply to all types of events or just out-events? A reasonable potential usage for event binding with in-events would be to trigger environment input – an in-event is triggered whenever its corresponding bound function is called in the environment. However, we are not able to instrument functions imported from libraries in order to implement this. A second question is: If $event\ binding$ applies to out-events only, how do we recognize which events are out-events when aspect External Output Event is not SYNTACTIC – that is, when out-events are determined at runtime? Moreover, if aspect External Output Event is SYNTACTIC, then there are two means to designate an event as an out-event – with a syntactic tag or with an event binding. This might confuse the modellers, given that only the event bindings associated with out-events can be executed.

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, then all event-binding calls are executed because an event with a binding is always deemed an out-event no matter when the event is generated. If External Output Events is not SYNTACTIC, then the only event-binding calls that are executed are whose events are determined at runtime to be out-events. This semantics for event bindings are both consistent with BSML's External Output Events semantic aspect, and make sense from the modeller's perspective.

5.2 Evolving Semantic Configuration

An obstacle to implementing configurable semantics is to consider semantic-dependent syntax, given that the configuration of semantic options might change during development. For example, when semantic aspect Big-step Maximality is SYNTACTIC, states can be tagged as **stable**. If the modeller changes the option from SYNTACTIC to TAKE MANY, then the **stable** tags make no sense and 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. This helps to reduce loss of information about semantic-dependent syntax when the semantic configuration evolves.

5.3 Event with Multiple Instances

BSML-mbeddr allows multiple instances of an event to exist at the same time. Distinct instances of the same in-event can be generated within multiple environment inputs, each of which is put into an input queue and processed by a state machine with a big-step. For out-events, all event-binding calls associated with generated out-events are executed at the end of a big-step. For example, Esterel [2], a member of BSML family, allows multiple instances of the same out-event to be generated simultaneously. In Esterel, the modeller can associates a $combination\ function$ with each out-event. Simultaneous multiple instances of the same out-event result in the combined execution of the event's combination function, instantiated with the arguments of each out-event instance. 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 $combination\ function$.

However, multiple instances of the same internal-event are hard to resolve. The semantic aspect Internal Event Lifeline regulates how long a generated internal-event is present in a big-step: it may disappear after a small-step without being processed, or it may remain present in the big-step even after being processed by a transition. If the latter, and multiple instances of the same internal-event continue to be generated in the same big-step, which event instances (and arguments) are processed by triggered transitions? In BSML-mbeddr, we resolve this by retaining at most one instance of

an internal-event. If an internal-event is generated multiple times in a big-step, then each instance overwrites earlier instances, so that any transition enabled by the internal-event sees only the latest instance and its arguments.

5.4 Big-step Semantics

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

First, BSML requires that the result of a big-step not be observable by the environment until the end of a big-step. We use the following 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 ban any operation that might change the status of the environment, including global-variable references on the left-hand-side (LHS) of an assignment, pointer dereferences on the 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 $isQuery$. A query cannot change the status of the environment, nor call any function that is not a query.
- A state machine cannot directly call any function that is imported from an external C library. Because the source code of a library function is not accessible, we cannot analyze whether the function may change the status of environment. Thus, to preserve the semantics of a big-step, we banned such calls.

Second, it is desirable to let multiple instances of the same state machine run concurrently without the machines interfering with each other, so BSML-mbeddr maintains separate copies of the runtime data for each machine instance. Multiple state-machine instances can call the same query without interfering with each other, because queries affect only internal-variables and are thus thread-safe. The problem raises when multiple state-machine instances may call the same non-query function through event bindings – function calls are executed at the end of a big-step and thus consistent with big-step semantics, but may not be thread safe. The modeller is responsible for keeping bound functions thread-safe if they are called by multiple state-machine instances.

6. Evaluation

6.1 Correctness

We have tested all implemented semantic options (Section 4.2) and all language features (Section 3.2) of BSML-mbeddr. Each of our test suite has (1) a semantic configuration, (2) one or more state-machine models, (3) environment code that drives the machines, and (4) statements that check state machine output against expected value.

A language feature or semantic option may be tested multiple times in different contexts. For example, cross-hierarchy transitions included cases where the target state is the ancestor or descendant of the source state, as well as cases where the target state and source state are in orthogonal regions; variables were tested with respect to their use in guard conditions, entry blocks, actions, and assignment expressions; tests on the semantic option ARENA ORTHOGONAL included cases where two transitions were: (1) arena orthogonal, (2) not arena orthogonal but source-target pairwise orthogonal, and (3) neither source-target pairwise orthogonal nor arena orthogonal; and so on.

6.2 Case Studies

We have also conducted three case studies to exercise the expressiveness of BSML-mbeddr. Our motivation has been to assess the applicability and integrability of BSML-mbeddr into the mbeddr C programming environment, and to verify that the modeller can use BSML-mbeddr to build state-machine models with various semantic requirements. The case studies are: (1) a Ground Traffic Control (GTC) system [18] which exercises concurrent regions and big-step semantics; (2) a Dialler System case study, adopted from example models in BSML [7], 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.

6.2.1 Ground Traffic Control

The Ground Traffic Control (GTC) case study is adopted from the work by Prout [18], originally developed by Bultan and Yavuc-Kahveci [23]. 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 communications between parallel machines through shared variables and events, which helps to exercise the expressiveness of BSML-mbeddr.

GTC schedules the usage of two runways, RW1 and RW2, and three taxiways TW1, TW2 and TW3 that each extend from runway RW1 to RW2 to the 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 RW1 must taxi on a taxiway and cross RW2 to reach the hanger. 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. Landing have higher priority than take-off requests.
5. An airplane on a taxiway can cross runway RW2 only if no airplane is using it.

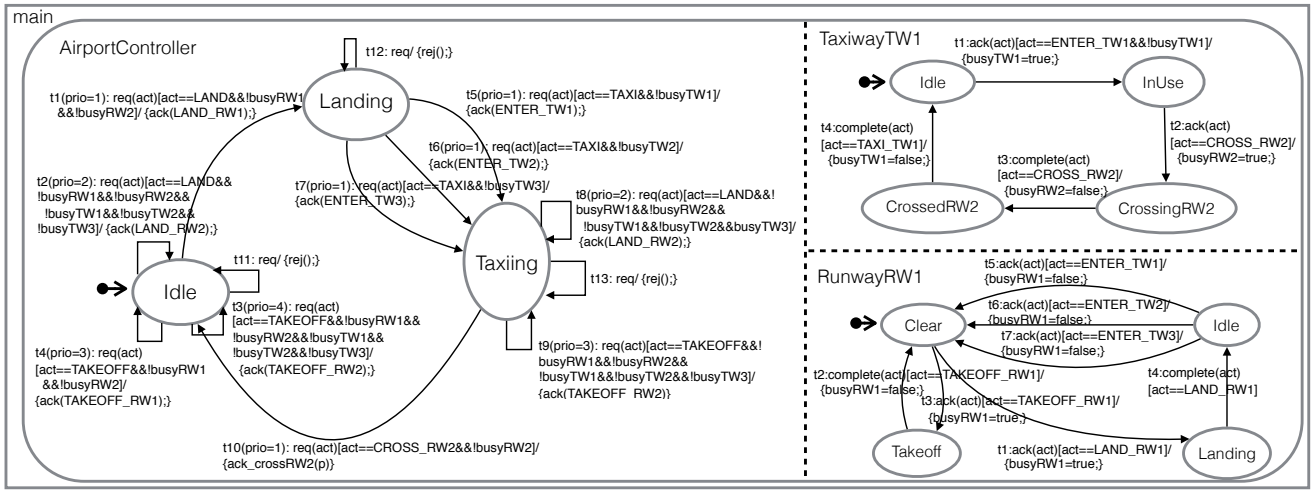


Figure 5: GTC Model

6. An airplane can land or take off on RW2 only if no airplane is on a taxiway.

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 (Figure 5). The case includes similar models for taxiways TW2, TW3 and runway RW2 (not shown in the figure). The Airport Controller receives from an airplane a $req(act)$ event that requests an action (e.g., request to take off, land, enter a taxiway) and generates an $ack(act)$ event that grants the action, if the action is safe. The generated ack event triggers transitions in the indicated taxiway or runway machines over the course of several small-steps; at the end of the big-step, the airplane is notified to take the requested action.

For semantic options, we chose SYNTACTIC for External Input/Output Event, and PRESENT IN REMAINDER for their Event Lifeline. We chose TAKE ONE for Big-step Maximality and SINGLE for Concurrency, which is simple and understandable, yet expressive enough for this model.

The environment code instantiates multiple airplanes and simulates their concurrent interaction with GTC. For verification, we express the properties described previously as assertions that must hold. For example, to verify property 3, in the entry blocks of states Landing and Takeoff within region RunwayRW1, we check that runway RW2 is in use. 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 state. The original model uses $rendezvous$ events to keep the Controller and Runways and Taxiways constantly in sync. In contrast, BSML-mbeddr uses normal events for inter-region communication, which are sensed in the next small-step. This semantics ensures that the machines are in sync and that all properties hold at the end of a big-step; they may be inconsistent in between small-steps, but this is not observable by environment. In a second version of GTC, we 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.

6.2.2 Dialler System

Adopted from Esmaeilsabzali's thesis [5], the Dialler System case study exercises big-step semantics, hierarchical states, and inter-region communication. In the Dialler System, a user can dial the digits of a phone number, or simply redial the previously dialled number. In addition, if the maximum concurrent calls are reached, the dialling process is interrupted. The state-machine model (shown in Figure 6) contains a state with two regions Dialler and Redialler, and a state Max that is entered when the limit on concurrent calls is reached. Region $Dialler$ accepts dialled digits one at a time (in-event $dial(d)$) and outputs only the first ten digits $out(d)$ to establish a phone connection. When the user hangs up the phone, in-event $reset()$ 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 previously dialled number in a single big-step. Specifically, region Redialler reacts to in-event $redial()$ by executing t_5 and t_6 that generate $dial(d)$ for each digit d in $last\_lp$. Each generated rendezvous event $dial(d)$ triggers transition t_2 or t_3 in the same small-step.

The selection of semantic options (shown in Figure 7) affects the correctness of the model:

- Big-step Maximality is SYNTACTIC, with three states marked as **stable** states: $WaitForDial$, $WaitForRedial$, and Max.
- Event $dial$ is a $rendezvous$ event, so it can trigger transitions in the same small-step in which it is generated. For example, when $dial$ is generated by transition t_5 (t_6), it triggers transition t_2 (t_3) in the same small-step.

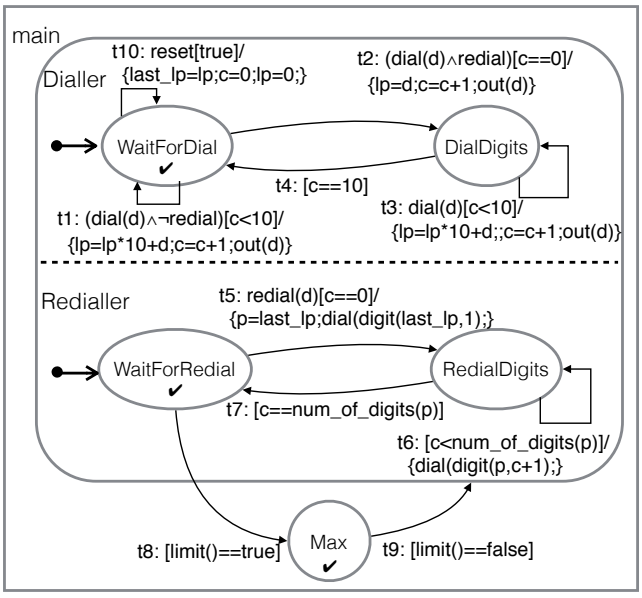


Figure 6: Dialler System Model

Big-Step Maximality SYNTACTIC
Concurrency MANY
Preemption PREEMPTIVE
Small-Step Consistency ARENA ORTHOGONAL
External Input Event RECEIVED IN FIRST SMALL
Input Event Lifeline IN REMAINDER
Internal Event Lifeline IN NEXT SMALL
External Output Event SYNTACTIC
Output Event Lifeline IN NEXT SMALL
GC Memory Protocol SMALL STEP
RHS Memory Protocol SMALL STEP
Priority HIERARCHICAL : SCOPE-PARENT

Figure 7: Dialler System Semantic Configuration

- 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. We do not choose SYNTACTIC because event *dial* is both an in-event and an internal event, making it impossible to tag it as an in-event. Note that even though *dial* is declared a rendezvous event, the in-event *dial* does not have a PRESENT IN SAME event lifeline because only internal-events can initiate rendezvous communications.
- Priority is HIERARCHICAL, with sub-options SCOPE-PARENT. This ensures that t_8 has higher priority and can interrupt any transition inside state Dialler[10].

[10] An alternative is to choose EXPLICIT priority and assign higher priority to t_8 than to other transitions.

```
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_inst(&ret_single),
         get_nonsingleton_inst(&ret_multi));
7    g_mutex_lock(&mutex);
8    return 0;}
9  statemachine SMFactory {
10  region main initial = main_state {
11    event set_inst(void* dest, void* const src,
          size_t bytes) => memcpy;
12    event unlock_mutex(GMutex* mutex) =>
          g_mutex_unlock;
13    state main_state {
14     region genSingleton initial = off {
15       Singleton* instance = null;
16       event get_singleton_inst(Singleton* ret);
17       state off {  };
18       state on {  };
19       t1: on get_singleton_inst[true] off -> on {
20         instance = sm_start(Singleton);
21         set_inst(ret,instance,sizeof[Singleton]);};
22       t2: on get_singleton_inst[true] on -> on {
23         set_inst(ret, instance, sizeof[Singleton]);}
               ;};
24     region genNonsingleton initial = any {
25       event get_nonsingleton_inst(NonSingleton* ret)
             ;
26       state any {  };
27       t1: on get_nonsingleton_inst[true] any -> any
             {
28         set_inst(ret, sm_start(NonSingleton), sizeof
              [NonSingleton]);
29    };};};
30  big-step end { g_mutex_unlock(&mutex); };
31 };};
32 statemachine NonSingleton {.....};
33 statemachine Singleton {.....};
```

Figure 8: State-Machine Factory Code

- Preemption is PREEMPTIVE, so that when t_8 is enabled, any other enabled transition is interrupted and the dialling/redialling process is aborted.

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

6.2.3 State-Machine Factory

The State-Machine Factory case study exercises 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.

In our model (Figure 8), a state machine SMFactory[11] is responsible for creating instances of machines Singleton and NonSingleton. The environment code creates an instance of SMFactory (Line 2), and generates in-events *get_singleton_inst*() and *get_nonsingleton_inst*() to get instances of Singleton and NonSingleton, respectively (Line 6). SMFactory keeps an instance of Singleton and returns a reference (Line 21, 23), whereas it creates a new instance of NonSingleton (Line 28). Note that an environment input

[11] The scoping rules of mbeddr C allow use before declaration. Thus, the use of SMFactory in Line 2 before its declaration in Line 9 is allowed.

may contain multiple in-events. To ensure that the SMFactory acts atomically (i.e., the SMFactory finishes processing a big-step before the environment code proceeds), we use a mutex to synchronize interactions between the state machine and the environment. A mutex is locked before a $sm\_trigger()$ statement (Line 5), as well as before the first time the returned state-machine instance is accessed (Line 7). The SMFactory releases the mutex to unblock the environment code after completing a big-step (Line 30).

7. Related Work

Our work is based on the high-level, systematic deconstruction of BSML semantics, which covers a more comprehensive range of semantics than previous studies on BSMLs [7] (e.g., Statecharts variants [22][12], Synchronous languages [10], Esterel variants [3][20], UML StateMachines [19]). BSML-mbeddr has a powerful execution semantics which is regulated by big-steps and small-steps – in each small-step, the set of enabled transitions and the set of transitions for execution are determined on the basis of semantic configuration. In contrast, many other state-machine modelling languages [4][8][15] have simpler execution semantics – the set of enabled transitions is considered by some order, and a single transition is executed.

7.1 Semantically Configurable Code Generator

Prout et al. [18] have implemented a prototype of a semantically configurable Code-Generator Generator (CGG) for a family of state-machine modelling languages. It supports semantic variability based on template semantics and uses preprocessor directives and conditional compilation to achieve configurable 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 semantics requires considerable expertise to understand the consequences of the decisions and their interdependencies [18]. In contrast, BSML raises the abstraction level of semantic deconstruction by decomposing the semantics of the same family of languages into only 12 mostly orthogonal semantic aspects and around 30 semantic options, which makes the semantics easier and more intuitive to be configured.

7.2 Code-model Co-development

mbeddr provides support for basic state-machine models though a predefined DSL $mbeddr.statemachine$ [15]. Similar to BSML-mbeddr, $mbeddr.statemachine$ allows a mixture of code and state-machine models, and provides event binding and triggering event as methods for communication between C code and the model. However, they have a simple execution semantics and no support for concurrent regions. In Table 2, we provide a comparison of the language and semantic features supported in both languages.

Umple [1][8][9] is a programming/modelling language and development environment that supports programming in code or models, and Umple keeps the two views in sync. Umple supports multiple modelling notations such as class diagrams and state machines), and multiple programming languages such as Java, PHP, and C++. Umple supports most of UML StateMachines [16] 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 the execution semantics are not configurable.

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 State-Machine Instances	✓	✓
Asynchronous Execution	✓	

Table 2: Comparison between BSML-mbeddr and mbeddr.statemachine

8. Conclusion

In this paper, we have introduced BSML-mbeddr: a state-machine modelling language with hierarchical states, concurrent regions and configurable semantics for which support has been integrated within the mbeddr C programming language environment. BSML-mbeddr is a non-trivial extension of the mbeddr eco-system that allows one to create sophisticated state-machine models within mbeddr. We have shown the feasibility of seamlessly integrating 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. We have evaluated the correctness and expressiveness of BSML-mbeddr through tests and case studies.

Acknowledgments

We would like to thank Markus Voelter for assistance in understanding mbeddr, and thank Shahram Esmaeilsabzali and Nancy Day for the help in understanding BSML.

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] Gérard Berry and Georges Gonthier. The esterel synchronous programming language: Design, semantics, implementation. *Sci. Comput. Program.*, pages 87–152, 1992.

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

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

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

[6] 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.

[7] 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.

[8] Andrew Forward, Omar Badreddin, Timothy C. Lethbridge, and Julian Solano. Model-driven rapid prototyping with umple. *Softw. Pract. Exper.*, pages 781–797, 2012.

[9] 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.

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

[11] David Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, pages 231–274, 1987.

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

[13] Zhaoyi Luo. Integrating semantically configurable statemachine models in a C programming environment. Master's thesis, University of Waterloo, Waterloo, Canada, 10 2015.

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

[15] 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.

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

[17] 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.

[18] 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.

[19] 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.

[20] Olivier Tardieu. A deterministic logical semantics for pure esterel. *ACM Trans. Program. Lang. Syst.*, 2007.

[21] 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.

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

[23] 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.