Monitoring Aspects for the Customization of Automatically Generated Code for Big-Step Models

Shahram Esmaeilsabzali
Electronics and Computer Science
University of Southampton
se3@ecs.soton.ac.uk

Bernd Fischer
Electronics and Computer Science
University of Southampton
b.fischer@ecs.soton.ac.uk

Joanne M. Atlee
School of Computer Science
University of Waterloo
jmatlee@cs.uwaterloo.ca

Abstract

The output of a code generator is assumed to be correct and not usually intended to be read or modified; yet programmers are often interested in this, e.g., to monitor a system property. Here, we consider code customization for a family of code generators associated with big-step executable modelling languages (e.g., statecharts). We introduce a customization language that allows us to express customization scenarios for the generated code independently of a specific big-step execution semantics. These customization scenarios are all different forms of runtime monitors, which lend themselves to a principled, uniform implementation for observation and code extension. A monitor is given in terms of the enabledness and execution of the transitions of a model and a reachability relation between two states of the execution of the model during a big step. For each monitor, we generate the aspect code that is incorporated into the output of a code generator to implement the monitor at the generated-code level. Thus, we provide means for code analysis through using the vocabulary of a model, rather than the detail of the generated code. Our technique not only requires the code generators to reveal only limited information about their code generation mechanisms, but also keeps the structure of the generated code intact. We demonstrate how various useful properties of a model, or a language, can be checked using our monitors.

1. Introduction

Automatic code generation from high-level models is a key technology to raise the abstraction level in software development, and so to increase productivity, and improve code reliability. While the output of a code generator is usually not meant to be read or modified by a programmer, it is often necessary to have a means to understand the behaviour of the code, for example, in order to integrate it into an existing code base or to inspect its correctness.

In this paper, we introduce a framework for automatic customization of generated code to enhance it with the capability to monitor a property at runtime. Our system receives a property-of–interest for the generated Java code of a model, and produces AspectJ [1] code that is woven to the generated code to monitor the property. Our system works with the output of a family of code generators [2] that each generates code for a behavioural model specified in a *big-step modelling language* (BSML) [3].

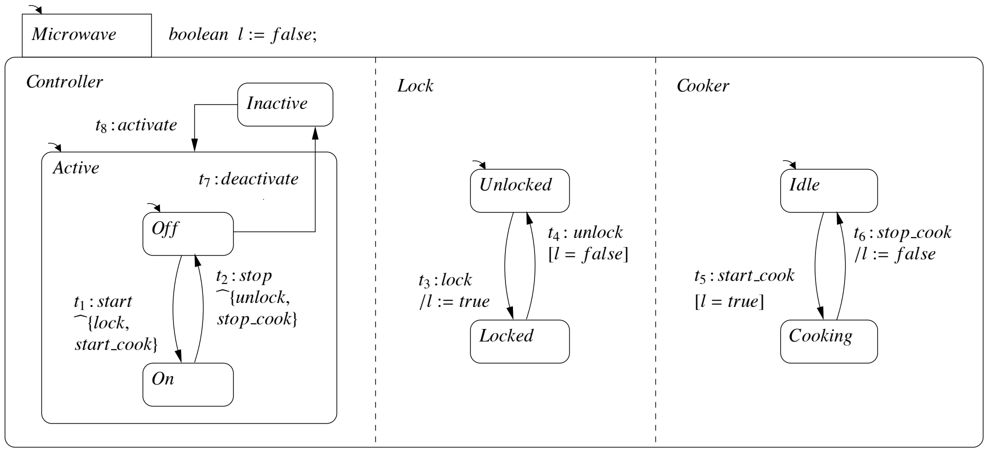
BSMLs are a widely used class of modeling languages, which includes the original statecharts [4] and its variants [5], among other formalisms. A BSML can be used to specify the behaviour of systems that interact with their environments continuously, e.g., an automated banking machine or a microwave. In a BSML model, the reaction of a model to an input is described by a big step, which consists of a sequence of small steps, each of which is the execution of a set of transitions. There is a plethora of BSMLs, which can essentially be distinguished by their semantic variations [3, 6]. These variations specify the detail of how transitions become enabled and how they are executed.

In our framework, a monitoring property is specified in the *big-step monitoring language* (BML), which we introduce in this paper. A BML monitor uses the vocabulary of a BSML model, and *not* the often unreadable vocabulary of the generated code, to specify a property for the generated code of the model. As such, our framework raises the level of abstraction that a developer works at: a developer neither needs to know about the code generation mechanism, nor about the mechanism by which a model-level property is monitored at the generated-code level. A BML property can be considered as a kind of predicate-logic formula over the transitions of a model, together with a *reachability* (or an *unreachability*) operator that specifies whether a certain state of a big step can reach (must not reach) another state of the big step. Quantification can be used to specify a general property about a model; e.g., the *global consistency* [7] property asserts that a transition that is triggered with the absence of an event and a transition that generates the event cannot be executed in the same big step. A BML property is either an *invariant*, required to hold in all big steps, or is a *witness*, required to hold in at least one big step. A novelty in the design of our BML is that it can be uniformly used by different BSMLs.

We define the semantics of BML by adapting the temporal operators of LTL [8] to work in the scope of a big step. A key idea in our semantics is that the *enabledness* and *execution* information about the transitions are treated as *uninterpreted functions* [9]: the semantics of BML is independent of how transitions become enabled and how they are executed. As such, BML abstracts away from the particularities of the semantics of BSMLs, and thus, is uniformly adoptable by the family of BSMLs. This allows for adopting BML for the output of different code generators as well.

We have implemented BML for a family of code generators [2] that generates code for a subset of BSMLs. We have developed a code generator that for each BML monitor generates the multi-threaded AspectJ code that collects the necessary information to interpret the monitor at runtime. Because of the design of our BML and the structure of the BSML-generated code, our generated aspects need to intercept the execution of only a few of methods

GPCE'11, October 22–23, 2011, Portland, Oregon, USA.
Copyright © 2011 ACM 978-1-4503-0689-8/11/10…$10.00



Microwave
boolean $l := false$;
Controller
Inactive
t_8 : activate
Active
t_7 : deactivate
Off
t_1 : start ⌢{lock, start_cook}
t_2 : stop ⌢{unlock, stop_cook}
On
Lock
Unlocked
t_3 : lock /l := true
t_4 : unlock [l = false]
Locked
Cooker
Idle
t_5 : start_cook [l = true]
t_6 : stop_cook /l := false
Cooking

Figure 1. A simple BSML model.

of the BSML-generated code. To implement BML for a new code generator, we require the BSML-generated code to expose only limited programmatic means to inspect the execution of big steps.

Our contribution in this paper is threefold. First, we introduce a language that allows to specify monitors to analyze the behaviour of generated code by using the vocabulary of a model, rather than the detail of its generated code. Thus, we combine model analysis with code analysis, as advocated by others [10, 11], to decrease the gap between the model and the code. Second, we present a uniform, automatic approach to implement BML monitors for a family of code generators that support different BSML semantics. Finally, we present a non-intrusive, aspect-based technique to customize the generated code to analyze its behaviour. Our technique is comparable with other approaches that aim at improving the extensibility of object-oriented software via aspects [12, 13].

The remainder of the paper is organized as follows. Section 2 presents an overview of the syntax and semantics of BSMLs. Section 3 presents our BML and its example applications. Section 4 presents the semantics of BML. Section 5 presents our implementation of BML for a family of code generators. Section 6 discusses related work. Section 7 concludes the paper.

2. Background: Big-Step Modelling Languages

In this section, we present an overview of the family of big-step modelling languages (BSMLs), whose semantics we deconstructed and compared in our previous work [3, 6]. We begin with describing the common, normal form syntax that we have adopted for BSMLs. We then briefly describe the common semantics of BSMLs, together with some of their semantic variations. There are further semantic variations that are not considered in this paper; details can be found in our previous work [3, 6]. We use the BSML model in Figure 1 as our running example throughout the paper.

2.1 Normal Form Syntax

To provide a unifying semantic framework for BSMLs, we introduced a normal form syntax, which is similar to the syntax of original statecharts [4, 7]. A BSML model is a hierarchical, extended finite state machine that consists of: (i) a hierarchy tree of control states, and (ii) a set of transitions between these control states.

Control states. A *control state*, graphically represented by a rounded box, represents a noteworthy moment in the execution of a model. Each control state has a *type*, which is either *And*, *Or*, or *Basic*. The control states of a model form a *hierarchy tree*, where the leaves (and only they) have type *Basic*. A *child* of an *And*-state or *Or*-state is surrounded by the box representing its *parent*; the children of an *And*-state are separated from one another by dashed lines. The *ancestor* and *descendant* relations are defined with their usual meanings. As an example, the model in Figure 1 is a simplified BSML model for the software system that controls the operation of a microwave oven. Control state *Microwave* is an *And*-state that has three children, namely, *Controller*, *Lock*, and *Cooker*, which are all *Or*-states; note that the surrounding lines around these *Or*-states have been removed to simplify the graphical representation. Control state *Unlocked* is a *Basic*-state and a child of *Lock*. One of the children of an *Or*-state is its *default* control state, which is signified by an incoming arrow without a source. The *root* of the hierarchy tree must be an *Or*-state, which is not explicitly shown if it has only one child; e.g., as in the model in Figure 1. Each control state has a unique *name* that appears at its top, left corner. The *least common ancestor* of two control states is the lowest (i.e., closest to the leaves) control state that is an ancestor of both; e.g., the least common ancestor of *Controller* and *Lock* is *Microwave*. Two control states are *orthogonal* if none of them is an ancestor of the other and their least common ancestor is an *And*-state; e.g., *Controller* and *Lock*.

Transitions. A *transition*, graphically represented by an arrow, specifies behaviour in a BSML model. Each transition, t, has a *source control state*, $src(t)$, and a *destination control state*, $dest(t)$, together with the following four optional elements: (i) a *guard condition*, $gc(t)$, which is a boolean expression over a set of variables, enclosed by a "[ ]"; (ii) a *triggering condition*, $trig(t)$, which is the conjunction of a set of events and negation of events; (iii) a set of *variable assignments*, $asn(t)$, which is prefixed by a "/", with at most one assignment to each variable; and (iv) a set of *generated events*, $gen(t)$, which is prefixed by a "⌢". Each transition name is followed by a ":". As an example, in the model in Figure 1, t_1 is a transition, with $src(t_1) = Off$, $dest(t_1) = On$, $gc(t_1) = true$, $asn(t_1) = \emptyset$, $trig(t_4) = start$, and $gen(t_4) = \{lock, start\_cook\}$. Transitions t_3, t_4, t_5, and t_6 use variables; e.g., $asn(t_3) = \{l := true\}$ and $gc(t_5) = [l = true]$. Variable l is used to disallow the situation where t_5 is executed before t_3 and t_4 is executed before t_6, to avoid microwave radiation while the door is unlocked. The *arena* of a transition t, denoted by $arena(t)$, is the lowest *Or*-state in the hierarchy tree such that the source and destination control states of the transition are its descendants. For example, in the model in Figure 1 $arena(t_7) = Controller$ and $arena(t_1) = Active$. For a model M, we denote the set of all its transitions as $Trans(M)$.

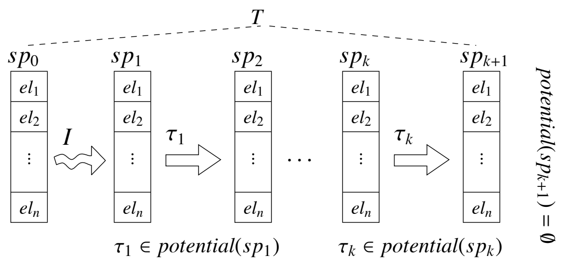

T
sp_0
sp_1
sp_2
sp_k
sp_{k+1}
el_1
el_2
el_n
I
τ_1
τ_k
$\tau_1 \in potential(sp_1)$
$\tau_k \in potential(sp_k)$
$potential(sp_{k+1}) = \emptyset$

Figure 2. Structure of a big step.

2.2 Common Semantics and Semantic Variations

A BSML model specifies the behaviour of a system that interacts with its environment. An *environmental input* is a set of input events together with a set of assignments to input variables. The reaction of a BSML model to an environmental input is a *big step* that consists of a sequence of *small steps*, each of which can be the execution of a set of transitions.

2.2.1 Model Initialization

Initially, in a BSML model, all variables are assigned their initial values; all events are *absent* (i.e., their statuses are *false*); and the model resides in the default control state of its root. Furthermore, the following invariants always hold for a BSML model: (i) if the model resides in one of its *And*-states, it resides in all of its children; and (ii) if it resides in one of its *Or*-states, it resides in exactly one of its children. As an example, initially, the BSML model in Figure 1 resides in control states *Microwave*, *Controller*, *Lock*, *Cooker*, *Active*, *Off*, *Unlocked*, and *Idle*. Variable l is initialized with a *false* value. Events *start*, *stop*, *activate*, *deactivate* are environmental input events of the model (they are not generated by any transition). The model has no environmental input variables.

2.2.2 Structure of a Big Step

Figure 2, adopted from our previous work [6], depicts the structure of a big step, T. The execution of a big step is an alternating sequence of snapshots and small steps, in response to an environmental input I. A *snapshot* of a BSML model is a valuation of its *snapshot elements*, each dealing with an aspect of the semantics of a BSML. For example, there is a snapshot element that maintains the set of control states that a model resides in: upon the execution of a transition, its source control state is removed from the snapshot element and its destination control state is added to the snapshot element. Similarly, there are snapshot elements that maintain the values of variables, the statuses of events, etc. The number of snapshot elements of a BSML model depends on the semantics of the BSML. In the big step in Figure 2, there are $k+2$ snapshots, namely, $sp_0, sp_1, \cdots, sp_{k+1}$, and n snapshot elements, namely, $el_1, \cdots, el_n$. We call snapshot sp_0 and snapshot sp_{k+1} the *source snapshot* and the *destination snapshot*, respectively, of big step T. Snapshot sp_1, called the *beginning snapshot*, includes the effect of receiving input I at snapshot sp_0. Each tuple, (sp_i, τ_i, sp_{i+1}), $(1 \le i \le k)$ is a small step of T. For each small step, (sp_i, τ_i, sp_{i+1}), sp_i and sp_{i+1} are its *source snapshot* and *destination snapshot*, respectively. The effect of the execution of a small step is stored in its destination snapshot. We often refer to a big step through its sequence of small steps; e.g., we refer to big step $T = \langle sp_0, I, sp_1, \tau_1, sp_1, \cdots, \tau_k, sp_{k+1}\rangle$ as $\langle \tau_0, \cdots, \tau_k\rangle$. At each snapshot, there might be more than one set of transitions that can be executed as the next small step; we call each of these sets of transitions a *potential small step* of that snapshot. We denote the set of potential small steps of a model at a snapshot, sp, as *potential*(sp). For a BSML model M, we denote the set of all its possible big steps as *bigsteps*(M).

2.2.3 Common Semantics

The flowchart in Figure 3, adapted from our previous work [3], depicts the conceptual stages in executing a single big step. At the beginning of a big step, an environmental input is received from the environment. The next six stages of the flowchart specify the necessary stages in forming and executing a small step. The flowchart iterates until its big step becomes *maximal*, meaning that there is no more small steps to be executed, at which point the big step concludes and the flowchart reaches its end. In each iteration of the flowchart, if there are more than one potential small steps, stage 5 chooses one non-deterministically. As an example, when the BSML model in Figure 1 resides in its default control states, if environmental input event *start* is received at the beginning of a big step, transition t_1 can be taken as a small step. The execution of t_1 generates events *lock* and *start_cook*, which trigger transition t_3 to be taken as the second small step. However, if $gc(t_5)$ would have been *true* and the BSML semantics would have only allowed one transition per small step, then $\{t_3\}$ and $\{t_5\}$ each would have been a potential small step after the execution of t_1.

2.2.4 BSML Semantic Variations

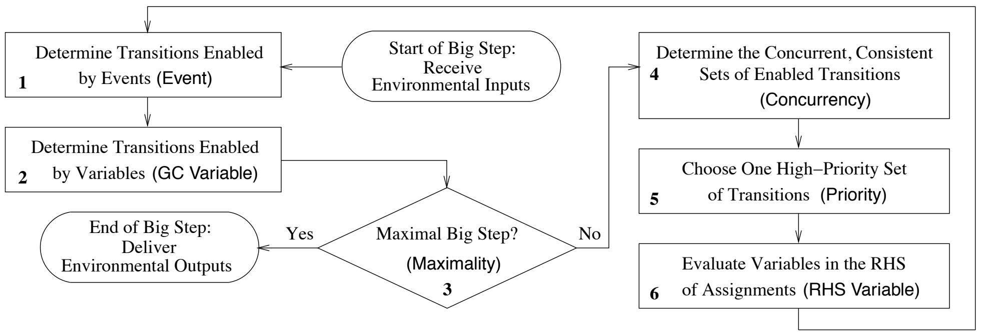
Each of the six numbered stages of the flowchart in Figure 3 could be carried out differently in different BSMLs, and thus, is a semantic variation point for BSMLs. We call these semantic variation points the *semantic aspects* of BSMLs [3]. Each semantic aspect can be instantiated with a *semantic option* that specifies how the corresponding stage of the semantic aspect must be carried out [3]. We use the sans serif and SMALL CAPS fonts to refer to the name of a semantic aspect and a semantic option, respectively.

The feature diagram [14] in Figure 4 shows six semantics aspects together with a common set of semantic options for each of the semantic aspects. Since variables and events are optional in the syntax of BSMLs, their corresponding semantic aspects are optional features of the feature diagram. In this paper, we consider only a commonly used subset of the semantic aspects and semantic options, but there are more semantic aspects and options [3]. The semantics that we consider in this paper are supported by the family of code generators that we consider in our implementation.

The Event semantic aspect specifies the snapshots in which a generated event is present and can trigger a transition. Three common semantic options for the Event semantic aspect are that a generated event is: (i) present only in the destination snapshot of the small step that generates it (the NEXT SMALL STEP semantic option); (ii) present in the destination of the small step that generates it, and in all subsequent snapshots in the big step (the REMAINDER semantic option); or (iii) present throughout the next big step after the big step in which it is generated (the NEXT BIG STEP semantic option).

The GC Variable semantic aspect concerns the variable values used to evaluate guard conditions. Two common semantic options are: (i) to use variable values from the beginning of the current big step, according to the assignments in the previous big step (the GC BIG STEP semantic option); or (ii) to use variable values in the current snapshot, thus taking into account the assignments made in the current big step (the GC SMALL STEP semantic option). The semantic options for the RHS Variable semantic aspect are similar. As an example, in the model in Figure 1, when the model resides in its default control states and input event *start* is received, only employing the REMAINDER and GC SMALL STEP semantic options results in the expected behaviour: $\langle\{t_1\}, \{t_3\}, \{t_5\}\rangle$; e.g., if the GC BIG STEP semantic option is employed, t_5 is not executed.

The Concurrency and Priority semantic aspects deal with forming the set of potential small steps of a BSML model at each snapshot, using the set of enabled transitions determined by stages 1 and 2 of the flowchart in Figure 3. The Concurrency semantic aspect specifies whether exactly one (the SINGLE semantic option) or all of



Start of Big Step: Receive Environmental Inputs
1 Determine Transitions Enabled by Events (Event)
2 Determine Transitions Enabled by Variables (GC Variable)
Maximal Big Step? (Maximality) 3
Yes
No
End of Big Step: Deliver Environmental Outputs
4 Determine the Concurrent, Consistent Sets of Enabled Transitions (Concurrency)
5 Choose One High–Priority Set of Transitions (Priority)
6 Evaluate Variables in the RHS of Assignments (RHS Variable)

Figure 3. Operation of a big step and its semantic variation points.

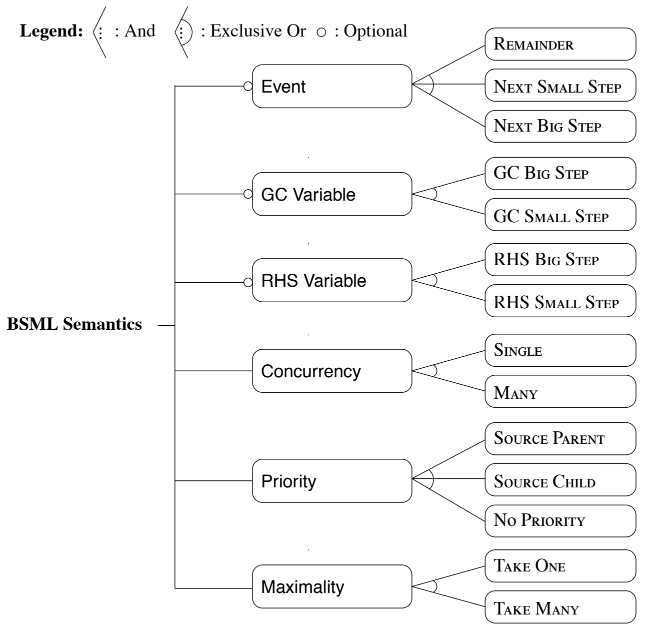


Figure 4. A set of common semantic aspects (in the sans serif font) and semantic options (in the Small Cap font) of BSMLs.

the transitions with orthogonal arenas (the Many semantic option) must be included in a small step. The Priority semantic aspect specifies whether a transition, t, has a higher priority than another transition, t', in which case if t can be included in a potential small step, then t' must not belong to any potential small step. Two common semantic options are that a transition with a higher source control state has a higher priority than a transition with a lower one (the Source Parent option) and vice versa (the Source Child option).

The Maximality semantic aspect specifies when a big step becomes maximal. Two common semantic options are that: (i) once a transition, t, is taken by a small step, no other transition whose arena is an ancestor or descendant of $arena(t)$ can be taken in that big step (the Take One semantic option); and (ii) a big step can continue until there is no more transitions whose trigger and guard condition are satisfied (the Take Many semantic option).

3. A Monitoring Language for BSMLs

This section introduces our *big-step monitoring language* (BML). A BML *monitor* for a BSML model specifies a property of the modelled system. The evaluation of such a property amounts to a run-time monitor that observes the execution of the model at the big-step granularity for adherence to the specified property. Monitoring at the big-step granularity is useful since it is compatible with the design philosophy of BSMLs that considers a big step (and not its constituent small steps) as a unit of execution. Section 4 describes the semantics of BML, which is oblivious to whether big steps of a BSML model are executed by the model itself or, for example, by the generated code for the model: it requires only information about the enabledness and execution of the transitions of the model. Similarly, this semantics is oblivious to the particularities of the plethora of BSML semantics: it treats the information about the enabledness and the execution of transition as uninterpreted boolean functions [9]. As such, BML provides a high-level means to specify properties about a BSML model that can be uniformly monitored both at the model and code level. In this paper, we are interested in monitoring only the behaviour of generated code. Section 5 presents our implementation of BML at the generated-code level.

The BNF in Figure 5 presents the abstract syntax of BML. As shown in the first line of the BNF, a monitor is either an *invariant*, which specifies a property that should hold for all big steps of the model, or a *witness*, which specifies a pattern that could happen in a big step of the model. An invariant monitor intercepts the execution of the model in order to find a *counterexample* for the property that it represents. A witness monitor intercepts the execution of the model in order to find a *witness example* of the property that it represents. A monitor is meant to execute as long as the model executes. An invariant monitor returns all counterexamples that it encounters during the execution of the model, and similarly, a witness monitor returns all witness examples.

Predicates. To specify monitors for a model, BML provides two basic unary predicates: en and ex, both of which operate over the transitions of the model. Predicate $en(t)$ evaluates to *true* in a snapshot iff transition t is *enabled*, i.e., iff t belongs to a potential small step in that snapshot. Predicate $ex(t)$ evaluates to *true* in a snapshot iff it is about to be *executed* by the next small step. By definition, if $ex(t)$ is *true* at a snapshot, so is $en(t)$. As an example, in the model in Figure 1, invariant monitor $I\colon \neg en(t_3) \vee \neg en(t_5)$ asserts that transitions t_3 and t_5 are never enabled together (i.e., there is no race between locking the microwave door and starting the radiation); and witness monitor $W\colon ex(t_7)$ asserts that transition t_7 executes at least in one big step (i.e., microwave can be deactivated). Predicates en and ex do not have a snapshot parameter because such a parameter is implicit in the semantics of BML.

Reachability expressions. Besides a predicate-logic-like syntax, BML uses two operators that each can be used to specify a kind of reachability or unreachability relation between the snapshots of a big step. Binary operators, $\hookrightarrow$ and $\not\hookrightarrow$ are *reachability* and *unreachability* operators, respectively. We call an expression that uses a reachability operator or an unreachability operator a *reachability expression* or an *unreachability expression*, respectively. In a reachability (or an unreachability) expression, the left operand is called

Monitor	::=	**I:** Invar | **W:** Witness
Invar	::=	ISpExp_1 | $\text{ISpExp}_1 \hookrightarrow \text{ISpExp}_2$ | $\text{ISpExp}_1 \not\hookrightarrow \text{ISpExp}_2$
ISpExp_1	::=	BoolExp | Quant_A BoolExp
ISpExp_2	::=	BoolExp | Quant BoolExp
Witness	::=	WSpExp_1 | $\text{WSpExp}_1 \hookrightarrow \text{WSpExp}_2$ | $\text{WSpExp}_1 \not\hookrightarrow \text{WSpExp}_2$
WSpExp_1	::=	BoolExp | Quant_E BoolExp
WSpExp_2	::=	BoolExp | Quant BoolExp
BoolExp	::=	**en**(t) | **ex**(t) | ¬BoolExp | BoolExp ∧ BoolExp | BoolExp ∨ BoolExp
Quant	::=	Quant_A | Quant_E
Quant_A	::=	$\forall\, t_v \in \text{Tr} \cdot$
Quant_E	::=	$\exists\, t_v \in \text{Tr} \cdot$
t	::=	t_v | t_c
t_v	::=	*A logical variable over transitions*
t_c	::=	*A transition of the model:* $t_c \in Trans(M)$
Tr	::=	*A subset of* $Trans(M)$; $\text{Tr} \in 2^{Trans(M)}$ | *A function's value,* f: $t \to 2^{Trans(M)}$

Figure 5. A BNF for the abstract syntax of BML.

the *source expression*, and the right operand is called the *destination expression*. The source expression specifies the snapshot(s) in a big step over which a reachability (or an unreachability) expression should be evaluated. The destination expression specifies the snapshot(s) that must be reached for the reachability to hold (or that must not be reached for the unreachability expression to hold). For example, $e_1 \hookrightarrow e_2$ specifies that from a snapshot of a big step that satisfies e_1, a future snapshot of the big step, including the current snapshot, can be reached in which e_2 is satisfied. An invariant monitor checks for a counterexample in which a reachability expression does not hold, while a witness monitor checks for an example in which a reachability expression holds. As an example, for the model in Figure 1, invariant I: $ex(t_3) \hookrightarrow ex(t_5)$ specifies that it is always the case that if transition t_3 is executed, then transition t_5 will also be executed in the same big step (i.e., if the microwave door is locked, then radiation eventually starts). The invariant I: $ex(t_5) \not\hookrightarrow ex(t_3)$ asserts that transition t_5 is not executed before or at the same time as t_3. Invariants I: $ex(t_5) \not\hookrightarrow ex(t_3)$ and I: $ex(t_5) \hookrightarrow \neg ex(t_3)$ are not the same: if $ex(t_5)$ is *true*, the latter invariant would hold when t_3 is not executed at least in one snapshot.

Quantification. To specify a monitor that applies to a range of transitions, the syntax of BML allows to quantify over a set of transitions of a model. For a monitor of a model that uses a (un)reachability[1] expression, each of its source and destination expressions can use a quantification. We assume the following two syntactic well-formedness conditions: (i) a monitor does not have any free variables; and (ii) its quantifiers use distinct logical variables. Invariants and witnesses use different kinds of *outer quantifications*, where an outer quantifier of a monitor is the quantifier that appears immediately after an "I :" or "W :". An invariant (a witness) could only use a universal (an existential) quantification as its outer quantifier. For example, invariant monitor I: $\forall t \in Trans(M) \cdot ex(t) \hookrightarrow \neg en(t)$ ensures that a transition of M cannot execute in all small steps of a big step, because it becomes disabled after it is executed. Witness monitor W: $\exists t \in Trans(M) \cdot ex(t)$ specifies that there exists a transition of M that is executed. The set of witness examples for this witness monitor at runtime could be used to animate the execution of M.

[1] "a (un)reachability" should be read "a reachability or an unreachability".

Range of quantification. The logical variable of a quantifier of an invariant ranges over a set of transitions. For a model, M, this range is either an explicit subset of the set of transitions of the model (i.e., a subset of $Tran(M)$) or a set that is determined by a syntactic function, f, where $f : t \to 2^{Trans(M)}$. For example, the invariant monitor, I: $\forall t \in Trans(M) \cdot ex(t) \not\hookrightarrow \exists t' \in samearena(t) \cdot ex(t')$, in effect, checks that the execution of M adheres to the TAKE ONE maximality semantics; function *samearena*(t) returns the set of transitions, excluding t, whose arenas are the same as t's.

Implicit quantifications. The meaning of a monitor involves some implicit quantifications as well. An invariant monitor has two implicit universal quantifications that assert that the invariant holds for all big steps and for all of their snapshots. A witness monitor has two implicit existential quantifications that specify that there exists a big step and a snapshot of that big step such that the witness property holds for it. Also, in the meaning of a reachability expression, in an invariant or a witness, there is an implicit existential quantification that asserts that there exists a destination snapshot in which the reachability expression holds.

Checking semantic properties. A *semantic property* of a modelling language is a semantic attribute of the language that is common to all models specified in that language [15]. Using BML, we can specify an invariant monitor to check a semantic property of a BSML model. Such a monitor cannot be used to prove the presence of a semantic property at the language level, but it can be used, for example, to confirm one's understanding about a BSML semantics, or to gain confidence about the correctness of an implementation of a BSML semantics. Next, we present two such monitors.

In a *globally consistent* BSML semantics [7], if the negation of an event is used to trigger a transition in a big step, that event is not generated during the same big step. The following invariant asserts the global consistency property, for BSML model M,

$$I\colon \forall t \in Trans(M) \cdot\ ex(t) \not\hookrightarrow \exists t' \in neggen(t) \cdot ex(t'), \qquad (1)$$

where *neggen*(t) is the set of transitions in the model that each generates at least one of the negated literals in the trigger of t.

In a *quasi non-cancelling* BSML semantics, which is similar to a *non-cancelling* [15] BSML semantics, if a transition, t, becomes enabled in a big step, either t or one of its *neighbouring transitions* will be executed in that big step; two transitions are neighbours if they have the same source or destination control states. The following invariant asserts this property,

$$I\colon \forall t \in Trans(M) \cdot en(t) \hookrightarrow \exists t' \in neigh(t) \cdot ex(t') \vee ex(t), \qquad (2)$$

where function *neigh*(t) returns the neighbouring transitions of t.

As an example, in Figure 1, when the model resides in its default control states and environmental input event *start* is received, assuming that $gc(t_5) = [true]$, if the model employs the SINGLE and NEXT SMALL STEP semantic options, then there are two big steps possible: $\langle\{t_1\}, \{t_3\}\rangle$ (not executing t_5) and $\langle\{t_1\}, \{t_5\}\rangle$ (not executing t_3), which violate invariant (2) above. Employing the MANY concurrency semantics results in a quasi non-cancelling semantics.

A limitation. The non-cancelling semantic property [15][2] is a stronger property than the quasi non-cancelling property: in a non-cancelling semantics, an enabled transition, t, in the above monitor, cannot *become disabled* until the destination expression of the reachability expression becomes *true*. We cannot express the non-cancelling property in BML, because BML does not have a syntax to capture the notion of "becoming disabled"; we plan to include such a syntax in BML in the future; cf., Section 7.

[2] In previous work [15], we used the term "executable" with the same meaning as the term "enabled" here; we changed our terminology here to provide a clear distinction between en and ex predicates.

4. Semantics of BML

The meaning of a monitor is defined with respect to a BSML model and its big steps. Our semantics for BML assigns a boolean value to each monitor of a model. If an invariant monitor is assigned a *false* value, then there exists at least one counterexample big step that makes it *false*. If a witness monitor is assigned a *true* value, then there exists at least one witness-example big step that makes it *true*. The semantics of finding counterexamples and witness examples is implicit in the semantics of assigning truth value to a monitor.

We present the semantics of invariant and witness monitors separately. Except for the semantics of reachability and unreachability expressions, the semantics of BML is simply based on the semantics of predicate logic. We first consider the semantics of *ground* reachability and unreachability expressions, whose source and destination expression do not use any quantification. We then extend these semantics to the cases with quantification.

Notation. We use notation $[\![y]\!]$ to denote the meaning of a BML term y. We use a subscript to denote the big step under which y is evaluated; e.g., $[\![y]\!]_T$ is the meaning of y under big step T. To specify the semantics of reachability and unreachability, we adapt the "globally" ($\Box$) and "finally" ($\Diamond$) temporal operators of linear temporal logic (LTL) [8], to express temporal properties of individual big steps of a model. To evaluate an LTL formula l against a big step T, we write $[l]_T$. Within a big step, the $\Box$ operator requires its operand to hold in all snapshots of the big step; the $\Diamond$ operator requires its operand to hold at least in one of the snapshots of the big step, including the current snapshot. These temporal operators can be nested and combined with logical operators. For example, predicate $[\Diamond(\Box\neg en(t_1))]_T$ asserts that transition t_1 is finally disabled in big step T and henceforth remains disabled in T.

4.1 Semantics of Monitors without Quantification

Semantics of invariant monitors. Given a big step, T, the meaning of a ground invariant monitor that neither uses a reachability nor an unreachability expression is easy; e should always be *true*:

$$[\![I\colon e]\!]_T \equiv [\Box e]_T$$

The meaning of a ground invariant monitor that uses a reachability expression is described by the following formula, which uses a request-response temporal pattern:

$$[\![I\colon e_1 \hookrightarrow e_2]\!]_T \equiv [\Box(e_1 \Rightarrow (\Diamond e_2))]_T.$$

Similarly, the meaning of a ground invariant monitor with an unreachability expression is described by the following formula:

$$[\![I\colon e_1 \not\hookrightarrow e_2]\!]_T \equiv [\Box(e_1 \Rightarrow (\neg\Diamond e_2))]_T.$$

As such, it can be observed that $[\![I\colon e_1 \not\hookrightarrow e_2]\!]_T \neq \neg([\![I\colon e_1 \hookrightarrow e_2]\!]_T)$, because, for example, if e_1 is always *false* in T, then both $[\![I\colon e_1 \not\hookrightarrow e_2]\!]_T$ and $[\![I\colon e_1 \hookrightarrow e_2]\!]_T$ are *true*.

Semantics of witness monitors. Given a big step T, the meaning of a ground witness monitor that neither uses a reachability nor an unreachability expression is easy; finally, e should become *true*:

$$[\![W\colon e]\!]_T \equiv [\Diamond e]_T.$$

The meaning of a ground witness monitor that uses a reachability expression is described by the following formula:

$$[\![W\colon e_1 \hookrightarrow e_2]\!]_T \equiv [\Diamond(e_1 \wedge \Diamond e_2)]_T,$$

which asserts that e_1 becomes *true* in T, followed by e_2.

Similarly, the meaning of a ground witness monitor that uses an unreachability expression is described by the formula,

$$[\![W\colon e_1 \not\hookrightarrow e_2]\!]_T \equiv [\Diamond(e_1 \wedge (\neg\Diamond e_2))]_T.$$

4.2 Semantics of Quantification

To specify the semantics of a monitor that uses quantification, we expand the monitor to a set of ground monitors. A quantified boolean expression has exactly one quantifier, while a quantified reachability or an unreachability expression can have up to two quantifiers (one in the source and one in the destination expression). Thus, we use two expansion functions: one for each quantifier.

The first expansion function, *exp_out*, eliminates the outer quantifier of a monitor. For an invariant monitor, $i = I\colon \forall t \in T \cdot e$, $exp\_out(i) = \bigwedge_{t_c \in T}(I\colon e[t_c/t])$, where $e[t_c/t]$ means rewriting e by replacing the quantification variable t with transition t_c. Similarly, For a witness monitor, $w = W\colon \exists t \in T \cdot e$, $exp\_out(w) = \bigvee_{t_c \in T}(W\colon e[t_c/t])$. Each of the monitor terms $I\colon e[t_c/t]$ or $W\colon e[t_c/t]$ might use another quantification, if e is a (un)reachability expression with two quantifications. The second expansion function, *exp_in*, eliminates these inner quantifications in a standard way. Function *exp_in*, as opposed to *exp_out*, does not introduce any new ground monitors; it just expands a destination expression. The two functions are identity functions if their inputs do not have an expected quantifier.

We then define the semantics of invariant $I\colon e$ of model M as:

$$[\![I\colon e]\!] \equiv \forall T \in bigsteps(M) \cdot [\![exp\_in(exp\_out(I\colon e))]\!]_T,$$

where each of $[\![exp\_in(exp\_out(I\colon e))]\!]_T$ is the evaluation of the conjunction of a set of ground invariants.

Similarly, we define the semantics of witness $W\colon e$ as:

$$[\![W\colon e]\!] \equiv \exists T \in bigsteps(M) \cdot [\![exp\_in(exp\_out(W\colon e))]\!]_T,$$

where each of $[\![exp\_in(exp\_out(I\colon e))]\!]_T$ is the evaluation of the disjunction of a set of ground witnesses.

An invariant must hold in all big steps, and thus a universal quantification is used in its semantics; a witness needs to hold in at least one big step, and thus an existential quantification is used.

5. An Implementation of BML

We adopt the output of the family of code generators introduced by Prout, Atlee, Day, and Shaker [2] for our implementation of BML. These code generators themselves are generated, on the fly, by a parametric code generator generator (CGG) that, based upon the semantic parameter values that it receives, uses conditional compilation to act as a particular code generator. The normal form syntax, the semantic aspects, and the semantic options of BSMLs, as described in Section 2, can be modelled by the syntax and the semantic parameters of the CGG, as outlined below.

First, the CGG uses a syntax that is comparable to our normal form syntax in Section 2.1: while we use a notion of a hierarchy of control states, CGG uses a notion of *composition tree*. A composition tree consists of a hierarchy tree of *composition operators*, each of which specifies a policy in the execution of the transitions of its operands. The two needed composition operators to model our normal form syntax are the `Micro-Interleaving` and the `Micro-Parallel` composition operators, which provide the means to model the Single and the Many Concurrency semantics, respectively. The leaves of a composition tree, and only they, are *hierarchical transition systems* (HTSs), each of which can be considered as an *Or*-state without any *And*-state descendant. We assume that a model employs either only `Micro-Interleaving` composition or only `Micro-Parallel` composition. We call a model that satisfies this criterion a CGG-BSML model.

Second, the CGG provides a set of parameterized snapshot elements and predicates that each could be customized to model a semantic option. We call a CGG semantics that corresponds to a combination of our semantic options a CGG-BSML semantics.

Organization of the section. Section 5.1 describes the code generation mechanism of CGG. Section 5.2 presents our own code generation mechanism that customizes a piece of CGG-generated code with aspects to evaluate a BML monitor. Section 5.3 reports about our experiments and discusses issues related to our implementation.

5.1 Structure of Generated Code and its Execution Pattern

Our implementation of BML is based on knowledge about the high-level structure of the code generated by CGG. Similar knowledge is needed when implementing BML for a different code generator. Given a CGG-BSML semantics and a CGG-BSML model, CGG generates sequential Java code that implements the behaviour of the model; concurrency is simulated via sequential execution.

Structure of generated Code. The structure of the generated code is based on: (i) the structure of the composition tree of the model; and (ii) the snapshot elements specified through the input semantic parameter values. There is a Java class for each composition operator and a Java class for each HTS. Each model has m HTSs. We refer to the names of the classes that represent them as HTS_1, $\cdots$, and HTS_m. There is a class called EnvSensor that provides an interface to implement an environment for the generated code. There is a root class called GeneratedSystem that instantiates and manages all classes. In our implementation of BML, we need to deal only with the classes that represent the HTSs of a model (to obtain information about the enabledness and the execution of transitions) and the EnvSensor class (to obtain information about the scope of a big step). Table 1 enumerates the classes that we use in our implementation of BML, together with the list of a few of their methods, fields, and variables that we need. For convenience, we have specified a symbol to refer to the name of each method.

Execution pattern. The generated code for a CGG-BSML model, specified in a CGG-BSML semantics, follows the semantic structure of a big step in Figure 2. The start of a big step is signified by the execution of senseEnv. The set of potential small steps at a snapshot are identified by the execution of a sequence of enabled_trans methods of the HTS_is ($1 \leq i \leq m$). The set of transitions of a small step are executed by the execute methods of the HTS_is ($1 \leq i \leq m$). Thus, the execution of a small step by a piece of CGG generated code can be encoded by regular expression $\mathtt{n}^{+}\mathtt{x}^{+}$, where n and x are symbols representing the invocation of an enabled_trans and an executed method, respectively. The execution of a big step can be encoded as $\mathtt{v}(\mathtt{n}^{+}\mathtt{x}^{+})^{*}\mathtt{n}^{+}$, where v represents the invocation of the senseEnv method; the last $\mathtt{n}^{+}$ denotes that the big step is maximal. The ongoing execution of a BSML model can be modelled as a sequence of big steps: $(\mathtt{v}(\mathtt{n}^{+}\mathtt{x}^{+})^{*}\mathtt{n}^{+})^{+}$.

5.2 BML Code Generator (BML-CG)

We have implemented a prototype system, *BML code generator* (BML-CG), which given a CGG-BSML model and a monitor without quantification generates the multi-threaded AspectJ code that performs the runtime evaluation of the monitor against the execution of the CGG generated code of the model. Intuitively, the generated aspects follow the semantics of BML in Section 4. For example, for an invariant monitor with an unreachability expression, whenever the source of the expression becomes *true* in a big step, the system checks whether its destination expression could become *true* in that big step, in which case it produces a counterexample.

Two key insights about the generated code by BML-CG are that: (i) by using AspectJ, a monitor expression is evaluated without modifying the structure and the behaviour of the CGG generated code; and (ii) by using threads, the evaluation of a monitor that uses a (un)reachability expression is orthogonalized into units that each corresponds to a snapshot where the source expression of the (un)reachability expression becomes *true*. The latter property makes BML-CG readily amenable to support quantification, by evaluating the constituent ground monitors of a monitor orthogonally. Next, we describe the design of BML-CG, focusing mainly on our use of aspects and threads in the BML-CG generated code.

5.2.1 Aspect Code Generated by BML-CG

AspectJ provides a rich language to specify *point cuts*, *join points* and *advices* for a Java program [1]. In a BML-CG generated code, however, we only need to use *before* and *after* advices for join points in the execution of the methods listed in Table 1.

Figure 6 presents the three pointcuts together with five advices that we use in our generated code to implement a BML monitor. In our implementation of BML-CG, we distinguish between four types of monitors: (i) invariants with reachability operators; (ii) invariants with unreachability operators; (iii) witnesses with reachability operators; and (iv) witnesses with unreachability operators. (The monitors without a (un)reachability operator are special cases of one of the above four types.) The point cuts and advices in Figure 6 are the same for these four types of monitors, except for the advice invoked before an enabled_trans method, i.e., before() : HTSenJoin(Object p). This advice is specialized for each type of monitor. Next, we describe each advice in detail.

Before the execution of an enabled_trans method, its corresponding advice checks whether the source expression of the (un)reachability expression of the monitor, i.e., src_exp(en, ex), is *true*. Expression src_exp(en, ex) is evaluated with respect to the information about the enabledness and the execution of transitions stored in arrays en and ex, respectively; these information are updated as the execution of the CGG generated code continues. When src_exp(en, ex) is *true*, a new thread is forked that will check if the destination expression of the (un)reachability expression of the monitor could become *true*. The effect of the execution of a small step is evaluated at its destination snapshot, before checking for the enabledness of the transitions for the next small step. As such, it suffices to do the above evaluation only if the big step is not at the beginning snapshot and before the first n in the $\mathtt{n}^{+}\mathtt{x}^{+}$ sequence of the next small step; i.e., when !begSnapshot && firstEn is *true*. Once all threads evaluate the effect of the last small step, i.e., after waitForAllThreadsToReact() terminates, the information in arrays en and ex are reset for evaluating the new small step.

After the execution of an enabled_trans method, its corresponding advice collects the set of enabled transitions determined by the method in array en; similarly, after the execution of an execute method, its corresponding advice collects the enabled transition that is executed by the method in array ex. We use Java reflection mechanisms to collect these information from the enabled_transitions and trans fields, described in Table 1. (To use Java reflection, we had to change trans variables to become fields of their corresponding classes. This is the only change that we made to the CGG-generated code.) Both advices update the variables firstEn and firstEx, which determine whether the first n and the first x in the sequence of method executions of a small step $\mathtt{n}^{+}\mathtt{x}^{+}$ are encountered, respectively. The latter advice is also responsible: (i) to increment the index representing the current snapshot of a big step, to add an element to the vector that stores the counterexamples/witness examples; and (ii) to update the variable that determines the beginning snapshot of a big step.

Before the execution of the senseEnvJoin method, i.e., when a current big step ends and a new big step is about to start, its corresponding advice sets the endBigStep to *true*. Setting this variable to *true* signals all forked threads during the big step to terminate; function waitForAllThreadsToEnd() ensures that these threads terminate. After the execution of this method, the set of all counterexamples or witness examples could be inspected in variable result. After the execution of the senseEnvJoin method, at the beginning of a big step, its corresponding advice resets the variables of the system, preparing for a new big step.

Class	Method	Symbol	Role
`EnvSensor`	`senseEnv(..)`	`v`	This method simulates the behaviour of the environment by setting the environmental input events and variables.
`HTS_1`, $\cdots$, `HTS_m`	`enabled_trans(..)`	`n`	For each `HTS_i`, $(1 \leq i \leq m)$, its method `enabled_trans` identifies the set of high-priority transitions whose arenas are in `HTS_i` and can be taken in the next small step; this set is stored in the field `enabled_transitions` of `HTS_i`.
`HTS_1`, $\cdots$, `HTS_m`	`execute(..)`	`x`	For each `HTS_i`, $(1 \leq i \leq m)$, its method `execute` executes one of the transitions stored in the field `enabled_transitions` of `HTS_i` non-deterministically; variable `trans` in `execute` method stores the identifier of the executed transition.

Table 1. List of the methods, fields, and variables in the CGG generated code that are of interest for the implementation of BML.

```
/*Enabledness and execution information.*/
boolean[] en = new boolean[#TRANS];
boolean[] ex = new boolean[#TRANS];
/*Counterexamples or witnesses of a big step.*/
Vector<HashSet<Integer>> result = ⟨⟩;
boolean begSnapshot = true; /*sp_1 or not.*/
boolean endBigStep = false; /*sp_{k+1} or not.*/
/*First n and x in small step n⁺x⁺ or not.*/
boolean firstEn = true;
boolean firstEx = true;
int curSnapshot =0; /*Current snapshot.*/
pointcut HTSenJoin(Object p):
  execution(HTS*.enabled_trans(..));
pointcut HTSexJoin(Object p):
  execution(HTS*.execute(..));
pointcut senseEnvJoin(): execution(*.senseEnv(..));
before(): HTSenJoin(Object p) {
 if (!begSnapshot && firstEn) {
  /*Based on the type of a monitor, one is executed.*/
  if (src_exp(en,ex)) {
   new [I:s ↪ d]findCounter(curSp).start(); /*or*/
   new [I:s ↪̸ d]findCounter(curSp).start(); /*or*/
   new [W:s ↪ d]findWitness(curSp).start(); /*or*/
   new [W:s ↪̸ d]findWitness(curSp).start();
   waitForAllThreadsToReact();
   for(i=0 to #TRANS-1) {en[i] = false; ex[i] = false;}
  }
 }
}
after(): HTSenJoin(Object p) {
 collectEnableds(p,en);
 firstEx = true;
 firstEn = false;
}
after(): HTSexJoin(Object p) {
 collectExecuted(p,ex);
 if (firstEx) {
  curSnapshot++; firstEx = false; firstEn = true;
  result.add(new HashSet());
 }
 if (begSnapshot) begSnapshot = false;
}
before(): senseEnvJoin() {
 endBigStep = true;
 waitForAllThreadsToEnd();
 /*Examine counterexamples and witness examples.*/
}
after(): senseEnvJoin() {
  for(i=0 to #TRANS-1) {en[i]= false; ex[i]=false;}
  result.clear();
  begSnapshot = firstEn = firstEx = true;
  endBigStep = false; curSnapshot = 0;
}
```

Figure 6. Point cuts and advices used in BML-CG generated code.

5.2.2 Multi-Threaded Code Generated by BML-CG

As mentioned earlier, the `before()` : `HTSenJoin(Object p)` advice, in the aspect in Figure 6, forks a thread when a snapshot of a big step is arrived at which the source expression of the (un)reachability expression of a monitor is *true*. Figure 7 shows these threads. Based on the type of a monitor, a thread is invoked to evaluate the monitor, through inspecting the value of the destination expression of the (un)reachability expression of the monitor, i.e., `des_exp(en, ex)`. A thread terminates when the big step ends, i.e., when `endBigStep` becomes *true*. Functions `addCounterExample` and `addWitnessExample` store a counterexample and a witness example, respectively, in the last index of `result`.

We note that, for a witness monitor, once a thread is forked during a big step, no more subsequent threads needs to be forked because one witness example suffices; similarly, for an invariant monitor that uses an unreachability expression, one thread per big step is enough. However, in our implementation, we continue to fork new threads, in order to, (i) find all counterexamples and all witness examples; and (ii) to develop a multi-threaded implementation, with the necessary synchronization mechanisms, to provide the foundation to support: (a) monitors with quantification, each of which comprises of multiple ground BML terms; and (b) concurrent evaluation of multiple monitors. As a result of this design decision, the first and the fourth threads, as well as, the second and the third threads, in Figure 7, are symmetric.

5.3 Discussion

Experiments. Using BML-CG, we have experimented with the generated code of a few example BSML models. We ran various BML monitors against the CGG generated code for the example model in Figure 1 (and its variations). Using different BSML semantics, we checked that a BML monitor behaves as expected, and thereby, tested the correctness of CGG, BML-CG, and the model as a whole. For example, in the model in Figure 1, if input events *stop* and *deactivate* are received together at the beginning of a big step, when the model resides in its default control states, the expected behaviour would be non-deterministic: either big step $\langle\{t_1\},\{t_3\},\{t_5\}\rangle$ or big step $\langle\{t_7\}\rangle$ would execute. To confirm this behaviour, the following two properties should hold: W: $en(t_1) \wedge en(t_7)$ and I: $(en(t_1) \wedge en(t_7)) \hookrightarrow (ex(t_7) \vee ex(t_5))$. However, if the Next Small Step event semantics is employed, the latter invariant would not hold because of counterexample big step $\langle\{t_1\},\{t_3\}\rangle$. As another example, to eliminate the above non-determinism, we changed the source of t_7 to *Active*, and employed the Source Parent priority semantics, which assigns t_7 a higher priority than t_1; the model then satisfied invariant $I: \neg en(t_1) \vee \neg en(t_7)$. In our experiments, we ensured that we cover the range of possible monitors and the range of BSML semantic options. We did not find any unexpected behaviour. We also experimented with other example models, including the CGG generated code for a model of an elevator system of a three-story building. This system was specified in a notation

```
/*Used for monitors with reachability expressions*/
void [I:s ↪ d]findCounter(int sp) {
 int myLastSnapshot = sp-1;
 while(!endBigStep) {
  if (myLastSnapshot < curSnapshot) {
   if (des_exp(en,ex)) return;
   myLastSnapshot++;
  }
 }
 addCounterExample(result); }
/*Used for monitors with unreachability expressions*/
void [I:s ↪̸ d]findCounter(int sp) {
 int myLastSnapshot = sp-1;
 while(!endBigStep) {
  if (myLastSnapshot < curSnapshot) {
   if (des_exp(en,ex)) addCounterExample(result);
   myLastSnapshot++;
  }
 } }
/*Used for witnesses with reachability expressions*/
void [W:s ↪ d]findWitness(int sp) {
 int myLastSnapshot = sp-1;
 while(!endBigStep) {
  if (myLastSnapshot < curSnapshot) {
   if (des_exp(en,ex)) addWitnessExample(result);
   myLastSnapshot++;
  }
 } }
/*Used for witnesses with unreachability expressions*/
void [W:s ↪̸ d]findWitness(int sp) {
 int myLastSnapshot = sp-1;
 while(!endBigStep) {
  if (myLastSnapshot < curSnapshot) {
   if (des_exp(en,ex)) return;
   myLastSnapshot++;
  }
 }
 addWitnessExample(result); }
```

Figure 7. Different kinds of threads for evaluating monitors.

that uses asynchronous events, which is out of the current scope of BSMLs [3]. Our BML-CG, however, could deal with such a generated code. For example, we monitored that the three transitions that open the three doors of the elevator are never enabled together.

Cost of monitoring. The BML-CG generated code incurs a runtime cost to the execution of the CGG generated code. In terms of space, this cost is modest: we introduce only a few global variables and two boolean arrays, `en` and `ex`, whose sizes are the number of the transitions of the model. In terms of time, however, the cost is proportional to the number of running threads, which in the worst case – where at each snapshot of a big step, one thread is forked – is proportional to the length of a big step. This cost includes the computation time of the threads, the cost of their synchronization, and the overhead of aspects and Java reflection. The cost of the evaluation of a BML monitor is not related to the size of the monitor: the evaluation of the source and destination expression of a (un)reachability expression are constant-time boolean evaluations. The size of a model, however, could indirectly affect the cost of evaluation: a big model can produce a long big step. In our experiments, we did not notice a tangible slowdown in the execution time of the GCC-generated code. However, we observed the importance of building the right environment for checking a BML property, so to avoid executing irrelevant transitions in checking the property.

BML for other code generators. Our implementation of BML relies on knowledge about how to obtain the set of enabled and executing transitions of each small step, and how to determine the start and the end of a big step at runtime. Any code generator that somehow exposes these information could be enhanced with a BML monitoring capability. The more explicit these information are exposed, the more efficient an implementation could be. For example, if the CGG generated code would expose each of the set of enabled and executed transitions of a small step in a single field of a single class, then it would be possible to use only two join points to collect these sets, instead of twice as many as the number of HTSs. Such a saving could result in a significant performance improvement. Even better, if these fields would have been accessible via existing methods of the generated code, no Java reflection would have been needed. We chose to use CGG as is to demonstrate the relative independence of BML from a code generator.

6. Related Work

Our work is related to *runtime monitoring frameworks* (RMFs), such as Temporal Rover [16] and PathExplorer [17], which provide tool support for monitoring an input temporal property against the execution of a program. In an RMF, an input temporal property is usually an LTL formula that is encoded in an *input format* (IF). Our BML and the IF of a typical RMF are comparable: they are both used to specify monitoring properties. Our BML, however, is distinct in two main respects. First, BML uses the vocabulary of models, such as the names of transitions and their enabledness and execution information, to specify a monitoring property for generated code. The IF of an RMF, however, uses the vocabulary of programs, such as the names of variables, methods, segments of the code, etc. Second, BML, by virtue of being specialized for the family of BSMLs, is preequipped with abstraction constructs that facilitate the specification of properties. As such, using the IF of an RMF to specify a property that is equivalent to a BML property could be challenging. For example, specifying the equivalent property to BML property (2) on page 5 could be a hard task; even articulating such a property in natural language through the vocabulary of the code can be very complicated. Of course, to evaluate a BML monitor at the code level, similar to an RMF property, the vocabulary of the code needs to be used. However, the abstraction constructs of BML provide guidelines not only about how to check these properties against the code, but also about how to generate/derive the code, in the first place, to facilitate such checks.

A class of RMFs, which we call *aspect-based* RMFs (AB-RMF), use aspects to specify and implement runtime monitors for programs [18–23]. The IFs of these RMFs and their implementation strategies are comparable to our BML and our BML-CG, respectively. While in our implementation of BML we use aspects, the syntax and the semantics of BML are independent of aspect technology. In an AB-RMF, however, its IF, its syntax, semantics, and implementation are all based on aspects, and thus based on the terminology of programs. While compared to a regular RMF, an AB-RMF provides a higher level of abstraction for property specification, it still uses a generic, program-level IF, as opposed to our BML, which is a specialized, model-level IF. Our use of aspects in the BML-CG generated aspects is comparable with the implementation of an AB-RMF: they both use the notion of execution join points to incrementally evaluate a property of the code at runtime. The difference is that we chose AspectJ simply because the output code of CGG naturally lends itself to be instrumented with aspects. Our regular-expression–like notation, in Section 5.1, is comparable to the IF of AB-RMF *Tracematches* [18].

Our work is comparable to frameworks that combine the aspect-oriented and generative programming paradigms [24–27]. Our work is distinct in that it focuses on a specific usage of generating aspects, as opposed to ”general-purpose aspect languages”, which are criticized for “losing their purposefulness” [28].

Our work is related to works that promote using aspects at the model level [12, 13], either to capture aspects during the modelling process [12], or to facilitate the extension of object-oriented code [13]. The point cuts in our implementation are model-based point cuts in that they originate from the vocabulary of a BSML model.

Hand-written aspects have been used to extend the functionality of a piece of generated code [29]. Our work is different in that BML works with model-level vocabulary of BSML models and our implementation automatically generates aspect code.

Lastly, our work follows the goals of software development methodologies that advocate model-driven code analysis [10, 11].

7. Conclusion and Future Work

In this paper, we introduced a language for specifying runtime monitors that analyze the behaviour of a piece of generated code that is derived from a model specified in a big-step modelling language (BSML). Also, we introduced a customization mechanism that modifies the generated code to enhance it with a runtime monitoring capability. Our big-step monitoring language (BML) has a high-level syntax that uses the vocabulary of a model, rather than the detail of the generated code, to specify a runtime monitor. As such, our BML raises the level of abstraction that a developer works at when analyzing the generated code. A novelty in the design of our BML is that it abstracts away from the particularities of the syntax and semantics of the plethora of BSMLs, and thereby, lends itself to be adopted by a wide range of modelling languages and by the output of a wide range of code generators. We have implemented the core, quantified-free fragment of BML for a family of code generators. We have developed a non-intrusive code generation technique that customizes a piece of generated code with the AspectJ, multi-threaded code that monitors a property.

We plan to extend our implementation to support quantified BML monitors. As discussed in Section 3, we plan to extend BML with predicates that capture the notions of "becoming disabled", and "becoming enabled". Also, to specify a wider range of runtime monitors, we plan to extend BML to support *backward* reachability and unreachability operators, so that a monitor could refer to the past snapshots of a big step. Lastly, we are interested in introducing an *action* syntax to BML so that a BML term could not only monitor the behaviour of generated code, but also could modify it. As an example, using action $disable()$, which removes a transition from the set of enabled transitions, it is possible to enforce a globally-consistent behaviour, as specified in property (1) on page 5, by disabling all transitions $t' \in neggen(t)$ in property (1).

Acknowledgements. We thank the reviewers for their helpful comments. The first and second authors were supported by EPSRC grant EP/F052669/1.

References

[1] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, "An overview of AspectJ," in *ECOOP'01*, no. 2072 in LNCS, pp. 327–353, Springer, 2001.

[2] A. Prout, J. M. Atlee, N. A. Day, and P. Shaker, "Semantically configurable code generation," in *MoDELS'08*, vol. 5301 of *LNCS*, pp. 705–720, 2008.

[3] S. Esmaeilsabzali, N. A. Day, J. M. Atlee, and J. Niu, "Deconstructing the semantics of big-step modelling languages," *Requirements Engineering*, vol. 15, no. 2, pp. 235–265, 2010.

[4] D. Harel, "Statecharts: A visual formalism for complex systems," *Science of Computer Programming*, vol. 8, no. 3, pp. 231–274, 1987.

[5] M. von der Beeck, "A comparison of Statecharts variants," in *FTRTFT'94*, vol. 863 of *LNCS*, pp. 128–148, Springer, 1994.

[6] S. Esmaeilsabzali and N. A. Day, "Prescriptive semantics for big-step modelling languages," in *FASE'10*, vol. 6013 of *LNCS*, pp. 158–172, Springer, 2010.

[7] A. Pnueli and M. Shalev, "What is in a step: On the semantics of statecharts," in *TACS*, vol. 526 of *LNCS*, pp. 244–264, 1991.

[8] A. Pnueli, "The temporal logic of programs," in *Proceedings of the 18th IEEE Symposium on the Foundations of Computer Science (FOCS-77)*, pp. 46–57, IEEE Computer Society Press, 1977.

[9] J. R. Burch and D. L. Dill, "Automatic verification of pipelined microprocessor control," in *CAV'94*, vol. 818 of *LNCS*, pp. 68–80, Springer, 1994.

[10] G. J. Holzmann, R. Joshi, and A. Groce, "Model driven code checking," *Automated Software Engineering*, vol. 15, no. 3-4, pp. 283–297, 2008.

[11] G. Holzmann, "Reliable software development: Analysis-aware design," in *TACAS'11*, vol. 6605 of *LNCS*, pp. 1–2, Springer, 2011.

[12] J. Gray, T. Bapty, S. Neema, D. C. Schmidt, A. S. Gokhale, and B. Natarajan, "An approach for supporting aspect-oriented domain modeling," in *GPCE'03*, vol. 2830 of *LNCS*, pp. 151–168, Springer, 2003.

[13] A. Kellens, K. Mens, J. Brichau, and K. Gybels, "Managing the evolution of aspect-oriented software with model-based pointcuts," in *ECOOP'06*, vol. 4067 of *LNCS*, pp. 501–525, Springer, 2006.

[14] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson, "Feature-oriented domain analysis (FODA) feasibility study," Tech. Rep. CMU/SEI-90-TR-21, SEI, Carnegie Mellon University, 1990.

[15] S. Esmaeilsabzali and N. A. Day, "Semantic quality attributes for big-step modelling languages," in *FASE'11*, vol. 6603 of *LNCS*, pp. 65–80, Springer, 2011.

[16] D. Drusinsky, "The temporal rover and the atg rover," in *SPIN Model Checking and Software Verification*, vol. 1885 of *LNCS*, pp. 323–330, Springer, 2000.

[17] K. Havelund and G. Roşu, "Monitoring Java programs with Java PathExplorer," in *RV'01*, vol. 55 of *Electronic Notes in Theoretical Computer Science*, pp. 1–18, Elsevier, 2001.

[18] C. Allan, P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble, "Adding trace matching with free variables to AspectJ," in *OOPSLA'05*, pp. 345–364, ACM Press, 2005.

[19] F. Chen and G. Rosu, "Mop: an efficient and generic runtime verification framework," in *OOPSLA'07*, pp. 569–588, ACM Press, 2007.

[20] V. Stolz and E. Bodden, "Temporal assertions using AspectJ," *Electronic Notes in Theoretical Computer Science*, vol. 144, no. 4, pp. 109 – 124, 2006. RV'05.

[21] E. Bodden, P. Lam, and L. Hendren, "Clara: A framework for partially evaluating finite-state runtime monitors ahead of time," in *RV'10*, vol. 6418 of *LNCS*, pp. 183–197, Springer, 2010.

[22] K. Havelund, "Runtime verification of C programs," in *TestCom/FATES'08*, vol. 5047 of *LNCS*, pp. 7–22, Springer, 2008.

[23] J. Seyster, K. Dixit, X. Huang, R. Grosu, K. Havelund, S. A. Smolka, S. D. Stoller, and E. Zadok, "Aspect-oriented instrumentation with GCC," in *RV'10*, vol. 6418 of *LNCS*, Springer, 2010.

[24] D. Zook, S. S. Huang, and Y. Smaragdakis, "Generating AspectJ programs with Meta-AspectJ," in *GPCE'04*, vol. 3286 of *LNCS*, pp. 1–18, Springer, 2004.

[25] D. Lohmann, G. Blaschke, and O. Spinczyk, "Generic advice: On the combination of AOP with generative programming in aspectC++," in *GPCE'04*, vol. 3286 of *LNCS*, pp. 55–74, Springer, 2004.

[26] D. R. Smith, "A generative approach to aspect-oriented programming," in *GPCE'04*, vol. 3286 of *LNCS*, pp. 39–54, Springer, 2004.

[27] U. Kulesza, A. F. Garcia, and C. J. P. de Lucena, "An aspect-oriented generative approach," in *OOPSLA'04 Companion*, pp. 166–167, ACM, 2004.

[28] T. Cleenewerck and J. Noyé, "Editorial domain specific aspect languages," *IET Software*, vol. 3, no. 3, pp. 165 –166, 2009.

[29] C. Henthorne and E. Tilevich, "Code generation on steroids: Enhancing cots code generators via generative aspects," in *Second International Workshop on Incorporating COTS Software into Software Systems: Tools and Techniques*, IEEE Computer Society, 2007.