Big-Step Semantics

Shahram Esmaeilsabzali, Nancy A. Day, Joanne M. Atlee
David R. Cheriton School of Computer Science
University of Waterloo
Waterloo, Ontario, Canada N2L 3G1
{sesmaeil,nday,jmatlee}@cs.uwaterloo.ca

Jianwei Niu
Department of Computer Science
University of Texas At San Antonio
San Antonio, Texas, U.S.A 78249
niu@cs.utsa.edu

February 28, 2009

Technical Report: CS-2009-05

Abstract

With the popularity of model-driven methodologies, and the abundance of modelling languages, a major question for a requirements engineer is: which language is suitable for modelling a system under study? We address this question from a semantic point-of-view for *big-step modelling languages (BSMLs)*. BSMLs are a popular class of behavioural modelling languages in which a model can respond to an environmental input by executing multiple, possibly concurrent, transitions. We deconstruct the semantics of a large class of BSMLs into high-level, orthogonal semantic aspects and discuss the relative advantages and disadvantages of the semantic options for each of these aspects to allow a requirements engineer to compare and choose the right BSML. We accompany our presentation with many modelling examples that illustrate the differences between a set of relevant semantic options.

1 Introduction

With the growing popularity of model-driven development (MDD), and domain-specific modelling notations [44], there is a need to understand how to create well-designed behavioural modelling languages. Many existing modelling languages are descendants of either Harel's Statecharts [22], or the languages that subscribe to the synchrony hypothesis [9]. We call this family of languages *big-step modelling languages* (BSMLs) because they assume that the system can respond to an environmental input by executing multiple transitions, without worrying about missing the next environmental input. In this report, we focus on the range of semantics of BSMLs.

A plethora of variants of BSMLs exist (e.g., Statecharts [22], and its numerous variants [64], Statemate [24], RSML [38], Argos [41], Reactive Modules [3] and Esterel [9]). The semantics have become more complicated with the variety of mechanisms for modelling composed behaviour of components. The literature includes numerous descriptions of the syntax and semantics of BSMLs. However, two tasks remain mainly unfulfilled:

1. When should one choose which semantic variant?

2. How can different semantic variants be compared, and on the basis of which criteria?

In this report, we seek to address these two questions, by: (i) deconstructing the possible semantic variations for BSMLs into a set of almost orthogonal *semantic aspects*: *concurrency*, *transition consistency*, *maximality*, *memory protocols*, *external variable communication*, *event lifelines*, *external event communication*, and *priority*, and (ii) analyzing the relative advantages and disadvantages of the possible *semantic options* of each semantic aspect. Related work partly address the above two questions, but only for a specific family of BSMLs (e.g., Statecharts variants [64, 32], Synchronous languages [20], Esterel variants [11, 62]). The work in [32] is close to our work in its approach, but considers languages that only have events. In addition to the relative comprehensiveness of our work, our work is distinct from the related work in that it focuses on semantics, and aims for providing an analysis of the semantics of BSMLs in an accessible way for modellers, in order to empower them to choose an appropriate semantics when a modelling problem is considered.

The appropriateness of a semantic choice for a BSML can depend on many factors including the behaviour one is trying to model, the constraints of the domain, and expertise of a modeller in a notation. Rather than trying to enumerate all factors, in this work, we focus on semantics, and analyze the advantages and disadvantages of each semantic option compared to another. Of course, one can write equivalent behaviours in different semantics by modifying the model (all BSMLs can be reduced to their meaning in primitive modelling languages such as Kripke structures, Buchi automata, labelled transition systems, etc.), however, it can be significantly more convenient (i.e., less syntax, more understandable) to model some behaviours in one semantics than another. We envision a world where these choices are made on a model-by-model basis, and BSML syntax is viewed as having configurable semantics.

The contributions of our work are the following:

1. A systematic, high-level, yet precise, deconstruction of the semantics of BSMLs into eight semantic aspects, and their related options.

2. A discussion of the relative advantages and disadvantages of each semantic option.

3. A set of carefully constructed examples that illustrate many of the differences between the choices succinctly.

Our goal is to provide guidance to software/requirements engineers about how to choose a BSML for modelling a system under study (SUS). We also aim to empower software/requirements engineers to be able to specify the semantic properties of a desired BSML, if existing BSMLs do not satisfy such properties. While it is impossible to claim that our options are complete, they cover a wide range of existing BSMLs, as well as, new semantics that arise through the enumeration of our proposed semantic options, which are likely to be useful for a future language designer. One major class of BSMLs, which we do not consider in this report and defer its investigation to our future work is the class of BSMLs that communicate events through buffers (e.g., SDL block diagrams [1] , Rhapsody [23], and UML Statemachines [50, 10]). However, except for "event lifelines" and "external event communication," other semantic aspects that we consider in this report are relevant for BSMLs with buffers too.

Some of the semantic variations that we discuss in this report are tightly related to certain syntactic well-formedness criteria; for example, a semantic option may be implemented only if a syntactic well-formedness is enforced. Whenever relevant, we discuss the syntactical well-formedness that are necessary for the implementation of a semantic option.

The remainder of this report is organized as follows. In Section 2, we describe a common syntax/semantics framework in which we describe the semantic variations of BSMLs; we also present a semantic taxonomy for BSMLs that allows us to specify the scope of our work. In Section 3, we present the semantic options for the semantics of BSMLs that are categorized under eight semantic aspects; we also include advantages and disadvantages of each semantic option along with examples. In Section 4, we conclude our report, and lay out our plans for future work.

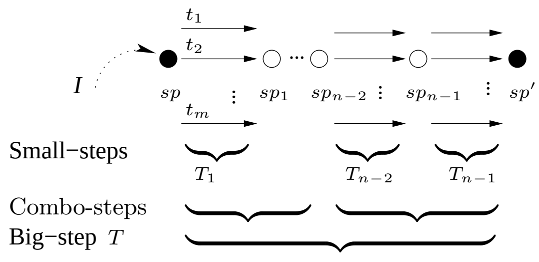


Figure 1: Big-step T. Snapshot *sp* is the "source snapshot," snapshots sp_i $(1 \leq i < n)$ are the "intermediate snapshots," and snapshot sp' is the "destination" snapshot of T. The dotted arrow represents the act of receiving the environmental input I whose effect is captured in source snapshot *sp*. A solid arrow represents a transition execution. Sets of transition executions T_i $(1 \leq i < n)$ are the small-steps of T. T has two combo-steps.

2 Preliminaries

In this section, we describe the common syntactic and semantic concepts that we use in the rest of the report. We start by describing what we mean by a *big-step modelling language* (BSML). In Section 2.1, we define a common syntax that is expressive enough to represent the syntax of many BSMLs. In Section 2.2, we define the common basic semantic model that we use, and refine, to describe the semantic variations of BSMLs. Finally, in Section 2.3, we present a taxonomy of BSMLs based on their semantics of interaction with the environment, which allows us to specify the scope of BSMLs that we investigate in this report. Table 1 and its continuation, Table 2, provide a concise description of the key terminology that we define in this section.

A BSML is a modelling language whose execution semantics is described as a sequence of *big-steps*, each of which specifies the reaction of a model to an *environmental input*. An environmental input is of a set of environmental input events and/or a set of environmental variable assignments. A big-step itself is a sequence of *small-steps*, each of which consists of a set of transition executions.[1] The execution of a small-step moves a model from one of its *snapshots* to another, where a snapshot is a collection of information about the values of variables, status of events, etc. The sequence of small-steps of a big-step starts from its *source snapshot*, and is followed by a maximal alternating sequence of small-steps and *intermediate snapshots*, until the execution concludes in the *destination snapshot* of the big-step. Figure 1 pictorially shows the structure of the execution of a big-step T. The source snapshot of T is *sp*. The solid arrows represent the transition executions of the big-step. The sets of transition executions T_i $(1 \leq i < n)$ are small-steps of the big-step. The snapshots sp_i $(1 \leq i < n)$ are the intermediate snapshots of the big-step, and snapshot sp' is the destination snapshot of the big-step. We consider a modelling language as a BSML, if it allows for executing more than one small-step before the model receives a new input from the environment.

Some BSMLs introduce an additional structure on top of the sequence of small-steps. A *combo-step* is a contiguous segment of the sequence of small-steps of a big-step where computation is carried out based on the values of the elements of the snapshot at the beginning of the combo-step. In BSMLs that support a notion of combo-step, the sequence of small-steps of a big-step can be viewed as a sequence of combo-steps, as shown in Figure 1. The usefulness of combo-steps is described in Sections 3.4 and 3.6.

[1]Big-steps are often called macro-steps in the semantic description of many BSMLs. We adopt a new term because: (i) our family of BSMLs is larger than those which have used the term macro-step, and (ii) we want to avoid association with the fixed semantics of the languages that use this term. Similarly, instead of "micro-step," which is used in some BSMLs, we use the term small-step. The big-step/small-step terminology has been used in the study of the operational semantics of programming languages, in a similar spirit as we use them here [53].

Term	Description
Ancestrally related	Two control states are ancestrally related if one is *child/grandchild* of another.
Atomic Execution	The execution of a set of transitions in a *small-step* is atomic if none of the transitions can see the effect of the execution of other transitions (except for rendezvous event communication as described in Section 3.1 and Section 3.6).
Basic-state	A control state that does not have any *children.*
Big-step	A maximal execution of a sequence of *small-steps* in response to an *environmental input.*
Child	A control state is a child of another if it appears as its immediate child node in the *composition tree.*
CHTS	A CHTS is a “composed hierarchical transition system,” which is a *composition tree* along with the transitions defined over its control states.
Combo-step	A consecutive segment of the sequence of *small-steps* of a *big-step* that for its execution uses the values of variable and/or the status of events from the beginning of the segment.
Composition tree	A tree whose nodes are control states and its leaves are *Basic-states.*
Concurrent-state	A control state that models concurrent execution, and has at least two *children.*
Configuration	A set of *Basic-states* of a model that specify in which control states the model resides. Each *Concurrent-state* is represented by a set of *Basic-states*, each of which represents one of its children. Each *Or-state* is represented by the *Basic-state(s)* that represents one of its children (a child of an *Or-state* might be a *Concurrent-state*).
Default state	A state whose *parent* is an *Or-state*, and is entered when a transition enters its *parent.*
Destination snapshot	The last *snapshot* in the execution of the sequence of *small-steps* of a *big-step.*
Enabled transition	A transition whose source is in the *configuration* of the current snapshot or is a *parent* of a member of the *configuration* of the current snapshot, and its *event trigger* and *variable condition* are satisfied.
Priority enabled transition	It is an *enabled transition* that has a higher priority than other enabled transitions whose sources are ancestrally related with its source.
Environmental input	A collection of set of inputs and a set of variable assignments that are received from the environment of a model.
Event action	A part of a transition, which is a set of generated events by a transition.
Event trigger	A part of a transition, which is a conjunction of the events and the negation of events of the model.
Grandchild	A control state is grandchild of another, if it is its *child* through transitivity, but not immediately.
Grandparent	A control state is grandparent of another, if it is its *parent* through transitivity, but not immediately.
HTS	An HTS is a “hierarchical transition system,” which is a maximal subtree of a *composition tree* that includes some leaves of a *composition tree*, and does not have any *Concurrent-states.*

Table 1: Summary of Terminology (continued in Table 2).

Intermediate snapshot	A *snapshot* of a *big-step* that is reached via the execution of a sequence of *small-steps* of the *big-step.*
Initial snapshot	A *snapshot* of the system where all *Basic-states* resides in their *default* states, and the value of variables are their default/initial values, and the status of all events is absent.
Or-state	A control state that has at least one *child*, and has a *default* state.
Orthogonal	Two control states are orthogonal if they are not *ancestrally related*, and their smallest, mutual parent is a *Concurrent-state.*
Parent	A control state s is a parent of control s', if s' is a *child* of s.
Small-step	An *atomic execution* of a set of *transition occurrences.*
Snapshot	A collection of information about the *configuration* that model resides in, the values of its variables, and the status of its events.
Source snapshot	The *snapshot* in the beginning of a big-step.
Transition occurrence	The effects of executing an *enabled transition.*
Variable action	A part of a transition, which is a set of assignments to some variables of a model.
Variable condition	A part of a transition, which is a logical expression over the variables of the model.

Table 2: (Continued from Table 1) Summary of Terminology.

2.1 Syntax

There is a plethora of BSMLs, including those with graphical syntax (e.g., Statecharts [22], and its numerous variants [64], Statemate [24], RSML [38], Argos [41]) and those with textual syntax (e.g., Reactive Modules [3], Esterel [9], SCR [27]). As is usual when studying a class of related notations and their semantics, we use a syntactic *normal form* [31] that is expressive enough to allow for convenient mapping of the syntax of other notations. Our normal form syntax is based on *composed hierarchical transition systems (CHTSs)* syntax, first presented in [48].[2] Our terminology in this section is adopted from [48] and [55].

A *model* is a CHTS, which consists of: (i) a *composition tree* whose nodes are *control states*, and (ii) a set of *transitions* between the control states. Figure 2 shows a model. A control state is shown as a rounded rectangle with a label that specifies the name of the control state, and a transition is shown as an arrow between two control states (i.e., between a *source* and a *destination* control state). The model in Figure 2 has five transitions: t_1, t_2, t_3, t_4, and t_5. The *root* of the composition tree of the model is control state S_0. A non-leaf node of a composition tree has *children* (e.g., control states S_1 and S_8 are the children of S_0). We call a control state that is associated with a non-leaf node of the composition tree a *hierarchical* control state. If a control state is a child of another control state through transitivity, we say that it is its *grandchild.* Similarly, the *parent* and *grandparent* relations are defined. Throughout the report, when clear from the context, we use the terms "state" and "control state" interchangeably. In the model in Figure 2, S_{11} is a control state, a child of S_1 and a grandchild of S_0.

A control state has a type: *Basic*, *Or*, or *Concurrent.* A Basic-state has no children. An Or-state has at least one child, and graphically, its children are contained in the rounded rectangular that represents it. For an Or-state S, one of its children is its *default* control state, which specifies the destination of a transition whose destination is S. A Concurrent-state has at least two children, and graphically, its children are separated via dashed lines. BSMLs have a different interpretation of the concurrency model of a Concurrent-state. In [48], we model the Concurrent-states of different BSMLs by using different composition operators. In this report, for the sake of brevity in our presentation, we use the same syntax for modelling

[2]In this report, we do not systematically consider the translation of different BSMLs into our syntax, but many of the translations are obvious. For more information on translation of a notation into our syntax, interested readers can refer to our previous works [48, 49, 47].

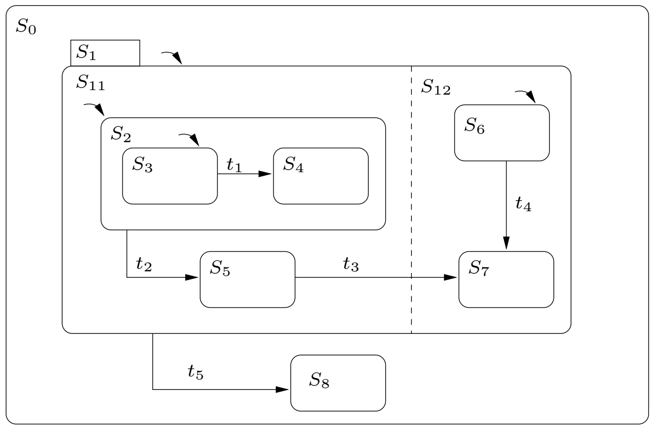


Figure 2: A model graphically representated as CHTS.

the Concurrent-states of different BSMLs, but discuss their semantic variants in Section 3.1. In the model in Figure 2, S_8 is a Basic-state, and S_1 is the only Concurrent-state of the model, whose two children, S_{11} and S_{12}, are Or-states. The default control state of S_{11} is S_2, which is graphically represented with an arrow with no source.

Two control states are *ancestrally related* if one is a (grand)child of another. The *least common ancestor* of two control states S_1 and S_2 is a control state S that is the (grand)parent of S_1 and S_2, and for any other control state that is the parent of S_1 and S_2, it is also a (grand)parent of S (i.e., S is the *smallest* (grand)parent of S_1 and S_2). Two control states are *orthogonal*, if neither is a (grand)parent of the other and their least common ancestor is a Concurrent-state. In the model in Figure 2, control states S_3 and S_7 are orthogonal (there are more pairs of orthogonal control states). For a CHTS, we call a maximal subtree in its composition tree that includes some leaves of the tree and does not include any Concurrent-states a *hierarchical transition system (HTS)* of the CHTS [48]. In the model in Figure 2, Or-states S_{11} and S_{12} are the two HTSs of the model.

When graphically representing an Or-state, we designate the default control state using an arrow with no source control state. The root of a composition tree must be an Or-state, but sometimes we present a model whose root is not an Or-state, which means that there is an implicit Or-state that is its parent, but we do not draw it pictorially, for the sake of brevity.

The computation in a model happens by executing its *transitions*, which change the values of its *variables* and the status of its *events* and move the model from one set of control states to another. Variables can be of any enumerable type; we assume that all expressions and assignments are well-typed. In this report, we only consider transitions that have one source and one destination. However, our discussions can be generalized to the case of multiple-source and/or multiple-destination transitions.[3] Figure 3 is a graphical representation of transition t whose source and destination control states are S_1 and S_2, respectively. A transition consists of four components: (i) an *event trigger*, which is a conjunction of events and the negation of events; (ii) a *variable condition*, which is a logical expression defined over the set of variables of the model; (iii) a *variable action*, which is a set of assignments, and (iv) an *event action*, which is a set of generated events. For transition t in Figure 3, its event trigger is $(e_1 \wedge \neg e_2)$, its variable condition is $x = 1$, which is graphically represented by enclosing it with a pair of square brackets; its variable action is $\{x := x + 1, y = 1\}$, which is graphically represented by prefixing it with a slash symbol; and its event action is $\{g_1, g_2\}$, which is graphically represented by prefixing it with a caret symbol.

We assume a global scope for events and variables. Modelling local variables or events can be achieved by

[3]To support transition with multiple-source and/or multiple-destination transitions, we only need to generalize our discussions in the sections on consistency semantics and hierarchical priority semantics, discussed in Section 3.2 and Section 3.8, respectively.

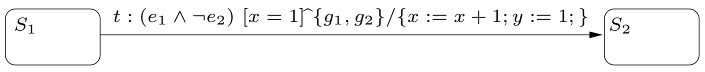

S_1
$t : (e_1 \wedge \neg e_2)\ [x = 1]\hat{}\{g_1, g_2\}/\{x := x + 1; y := 1;\}$
S_2

Figure 3: A Transition.

renaming them to their corresponding global variables or events. An event with parameters can be modelled by associating a variable to each parameter of an event that represents the value of the parameter when the event is generated. If an event is generated by more than one transition, then its parameter should have a *combining operator* [9] that merges the parameter values of different event generations; usually a combining operator is both commutative and associative (e.g., addition for integer values).

We do not consider event triggers with disjunctions, because an event trigger that has a disjunct normal form can be split into multiple transitions, each of which has exactly the same elements as the original except that it has only one of the disjuncts of the original event trigger as its event trigger; such a transformation yields a model that is semantically the same as the original model [55].

2.2 Common Basic Semantics

The semantics of a BSML describe how a model reacts via a big-step to an input that is received from its environment. The execution of a big-step is defined as a sequence of small-steps. A transition is *enabled* and can be executed if: (i) its event trigger and variable condition are satisfied, (ii) the model resides in a set of control states such that the source of the transition either belongs to the set, or is a (grand)parent of one of the states in the set. An enabled transition is *priority enabled* if it has the same or higher priority than other transitions that can be executed instead of it. We call the execution of a transition a *transition occurrence*. The semantics of when a transition is enabled/executed varies and will be discussed in the remainder of the report. We use the name of a transition to refer to its corresponding transition occurrence (e.g., for transition t, we use the same symbol t to refer to its occurrence). Usually, when clear from the context, we use the term "transition" instead of "transition occurrence." A small-step is a set of transition occurrences. When a small-step is executed, the model moves from one snapshot to another, where a snapshot is a tuple that consists of three elements:

- S: a *configuration* of the model that specifies the control states that the model resides in. A configuration of a model is a set of its control states, such that if the model resides in a child of a Concurrent-state, then it resides in all of its children, and if the model resides in an Or-state, it resides in exactly one of its children. We specify a configuration by only specifying the basic control states of the model, the rest of control states can be derived.
- V: a set of pairs, where each pair consists of a variable name and its value in the snapshot.
- E: a set of *present* events, which are either generated or are received from the environment.

We use the special value "*" instead of the value of an snapshot element to represents an unknown/irrelevant value.

A model starts its execution from the *initial snapshot* of the model, which is a snapshot such that: (i) all Or-states are in their default control states, (ii) all variables have their initial/default values, and (iii) the status of all internal events (excluding those that are received from the environment) is absent.

We write $sp = (S, V, E) \xrightarrow{T_1} sp' = (S', V', E')$ to denote that by executing small-step T_1, the model moves from snapshot sp to snapshot sp'. For transition t in Figure 3, if we consider snapshot $(\{S_1\}, \{x = 1, y = *\}, \{e_1\})$, then t can be executed as a small-step (sometimes we represent a singleton without curly brackets [e.g., set $\{t\}$ below can be written as t]).:

$$\begin{array}{ll}(\{S_1\}, \{x = 1, y = *\}, \{e_1\}) & \xrightarrow{\{t\}} \\ (\{S_2\}, \{x = 2, y = 1\}, *). & \end{array}$$

We chose to use "*" as the event element of the destination snapshot of the small-step to indicate that depending on the semantics of generated events in a BSML (discussed in Section 3.6), the generated events of a small-step may "persist" for some or all snapshots of a big-step.

The execution of a small-step is *atomic*, which means that: (i) for two transitions in the small-step, variable/event actions of a transition cannot be seen by the other (except for rendezvous event communication as described in Section 3.1 and Section 3.6, which allows a generated event of a transition in the small-step to be seen by other transitions of the small-step); (ii) all variable/event actions of a transition occurrence in the small-step take effect (except for race condition [described in Section 3.4.3], where multiple assignments to the same variables by different transitions are overwritten); and (iii) during the execution of a small-step of a model, it either resides in the source or the destination configuration, but never in between. Some BSMLs use a sequence of assignments as the variable action of a transition, following the semantics that the value of a variable at the right hand side (RHS) of an assignment is the last value that has been assigned to that variable in the sequence of assignments of that transition. However, since we assume atomic execution of transitions, a sequence of assignments can always be turned into a set of assignments by substituting the RHS of variables accumulatively into later assignments [39, 37]. Therefore, we always consider a set of assignments for transitions.

We use the symbol "|" as a delimiter to separate the combo-steps of a big-step. As an example, the representation of the execution of big-step in Figure 1 is the following sequence:

$$\begin{array}{ll} sp_1 & \xrightarrow{T_1} \\ \ldots & \\ sp_{n-2}| & \xrightarrow{T_{n-2}} \\ sp_{n-1} & \xrightarrow{T_{n-1}} \\ sp'. & \end{array}$$

In the semantics of BSMLs, the computation of the sequence of small-steps of a big-step is carried out incrementally: It starts from the source snapshot of the big-step, and the small-steps of the big-step are computed iteratively until there are no more small-steps to be taken according to the maximality semantics of the BSML. In a few exceptional semantics, in order to compute the next small-step of a big-step in an intermediate snapshot, the semantics needs to know about the variable/event actions of the future transitions in the sequence of small-steps; we call these semantics *forward-referencing semantics*, which can be difficult/impossible to describe/implement, because sometimes a forward reference to the future small-steps of a big-step is not feasible. Throughout the report, we mention the BSMLs that have forward-referencing semantics, and describe their advantages and disadvantages.

2.3 A Taxonomy for BSMLs

In order for the semantics of a BSML to be sensible, there needs to be a semantic justification for why a model is entitled to execute multiple small-steps in response to an environmental input, without worrying about missing the new inputs that might arrive during the execution of the big-step. Three justifications are possible:

– *Fast computation*: This justification states that if the BSML is considered for modelling a system that is assumed to be fast enough not to miss any environmental inputs when processing a previous input, then the model of the system is entitled to take multiple transitions in one big-step. This justification is in accordance with the design philosophy of BSMLs that support the *synchrony hypothesis* [9, 55, 32, 20]. (The synchrony hypothesis is sometimes referred to as the *zero-time assumption* [9].) The domain of systems that are modelled by using this paradigm is called *reactive systems* [25, 9, 20]. A reactive system is usually a mission-critical system that is meant to react to an environmental input in a timely manner (e.g., the controller system of a nuclear reactor). A reactive system might be either implemented as an embedded software of a piece of hardware, or directly as a piece of hardware [6, 8, 18]. As such, many of the BSMLs that support the synchrony hypothesis adopt their underlying principles from the principles of hardware. For example, a BSML might equate a big-step as a reaction of the model during a "tick" of the global clock of the system, where the notion of clock might be described explicitly (e.g., Lustre [21] and Signal [4]) or implicitly (e.g., Esterel [9] and Argos [41]). The implication of adhering to hardware concepts in a BSML is that, for example, the status of an event of a model during a big-step, similar to the status of a signal in a clock tick of a synchronous hardware, can be either present or absent, but not both.

– *Helpful environment*: This justification states that if the BSML is considered for modelling a system whose environment is *helpful* enough not to overwhelm the system with inputs when it is not ready, then the model of the system is entitled to take multiple transitions in one big-step.[4] Statecharts [26] and many of its variants [64] follow the "helpful environment" justification in their design. The domain of systems that are modelled by using this paradigam are called the *intereactive systems* [20]. An interactive system is different from a reactive system in that the rate of change in the environmental inputs of a model is dictated by the system, rather than by the environment. An example of an interactive system is an automated banking machine, which interacts with its environment (i.e., a customer) at its own rate when it is ready, rather than at the rate the customer would like to provide inputs for it.

– *Asynchronous communication*: This justification states that if the system can be equipped with a buffering mechanism to store the environmental inputs that might arrive during the execution of a big-step, then the model of the system can adopt an asynchronous communication mechanism with its environment, which ensures that an environmental input is never missed.

The "fast computation" and "helpful environment" justifications are mutually exclusive with the asynchronous communication justification. This is because if we assume either of the "fast computation" or "helpful environment" justifications, then since a model never misses any environmental input, it is not necessary to buffer environmental inputs. Conversely, if we assume the "asynchronous communication" justification, then since the environmental inputs are buffered, it means that neither the computation needs to be fast, nor the environment necessarily needs to be helpful.

In this report, we do not consider the BSMLs that follow the asynchronous communication justification, and focus on the semantics of the BSMLs that support the first two justifications. Throughout the report, we use the term BSML to mean a BSML that uses the "fast computation" or "helpful environment." The BSMLs that follow the "fast computation" and "helpful environment" justifications share many semantic aspects (e.g., both use a broadcast event communication mechanism, both support hierarchical state structures etc.) As such, sometimes it is difficult, and unnecessary, to conclusively label a BSML as a BSML that follows one or the other justification.

3 Semantic Aspects

We deconstruct the semantics of BSMLs into eight semantic aspects: *concurrency*, *transition consistency*, *maximality*, *memory protocols*, *external variable communication*, *event lifelines*, *external event communication*, and *priority*. Figure 4 illustrates the eight semantic aspects, their related semantic options, and the related section in the report that we describe the semantic aspect. A semantic aspect itself might be deconstructed into some underlying semantic sub-aspects, each of which would consist of its own semantic options. We study each semantic aspect in a separate subsection. We use the Small Cap font for semantic options. For each semantic option, we enumerate the BSMLs that support it (differentiating between different Statechart variants by prefixing the term "Statecharts" with the abbreviations of the name of the corresponding authors of the corresponding variant). Throughout the section, we use many examples to describe semantic options and their differences clearly. The summary of the discussions in each subsection is presented in a tabular format, which includes: (i) the semantic options of the semantic aspect, (ii) the relative advantages

[4]The term "helpful environment" is coined in [16, 17], but not in the context of BSMLs.

Option Description	Advantages	Disadvantages	Example
Number of Transitions in a Small-step			
SINGLE	Understandable	Excessive non-determinism	1
MANY	Reduced non-determinism	Concurrency complexity (e.g., race conditions)	1
SYNCHRONIZATION OPTIONAL/MANDATORY	Atomic event communication	Synchronization complexity	2
Order of Small-steps in a Big-step			
NONE	Ease of use	Non-determinism, difficult to analyze	3
IMPLICIT/EXPLICIT DATAFLOW	Ease of use, and terminating big-steps	Hard to analyze, less reactive, and cyclic orders	3
EXPLICIT	Great control for order specification	Burden of specifying orders, and less reactive	4

Table 3: Concurrency Semantic Aspect.

and disadvantages of each semantic option compared to each other, and (iii) the list of example models that are relevant for each semantic option.

3.1 Concurrency

BSMLs vary in the semantics of how components execute concurrently. There are two sub-aspects for concurrency: (i) can more than one transition occurrence be taken together in a small-step? and (ii) is there an order of execution between the small-steps of a big-step? Table 3 summarizes the concurrency semantic sub-aspects and the related semantic options. Compared to programming languages, concurrency in BSMLs is simpler because the transitions of a small-step are executed "atomically."

3.1.1 Number of Transitions

There is a dichotomy in hardware and software about how to model the execution of a system: *Single-transition* vs. *Many-transition* models [45, 60, 57, 57, 63]. Given a set of enabled transitions, this semantic aspect specifies how many transition occurrences can appear in a small-step.

SINGLE: This option allows at most one transition to execute in each small-step (e.g., Statecharts [22], P&S Statecharts [54], H&P&S&S Statecharts [26], Statemate [24], Reactive Modules [3], and RSML [38]). In some notations, for example process algebras such as CSP [30] and CCS [46], this option is called interleaving. [5] An **advantage** of this option is that models can be more easily reviewed and understood, because a modeller only needs to consider one transition occurrence at a time. A **disadvantage** of this option is that in the absence of an exhaustive priority scheme for transitions, it can lead to many non-deterministic executions that a modeller needs to consider.

MANY: In this option, a maximal set of enabled transitions can be taken together in a small-step; examples are: Argos [41] and Esterel [9]. Process algebraic notations with the MANY flavour have also been introduced (e.g., SCCS [46, 45]). An **advantage** of this semantic option is that it avoids the undesired non-determinism of the SINGLE option. A **disadvantage** is that this semantic option requires a modeller

[5] The notion of transition in a BSML, which includes variable condition, event trigger, variable/event action, is more expressive than the primitive notion of transition in process algebras, and thus we use the term SINGLE instead of "interleaving" to distinguish their concurrency semantics differences.

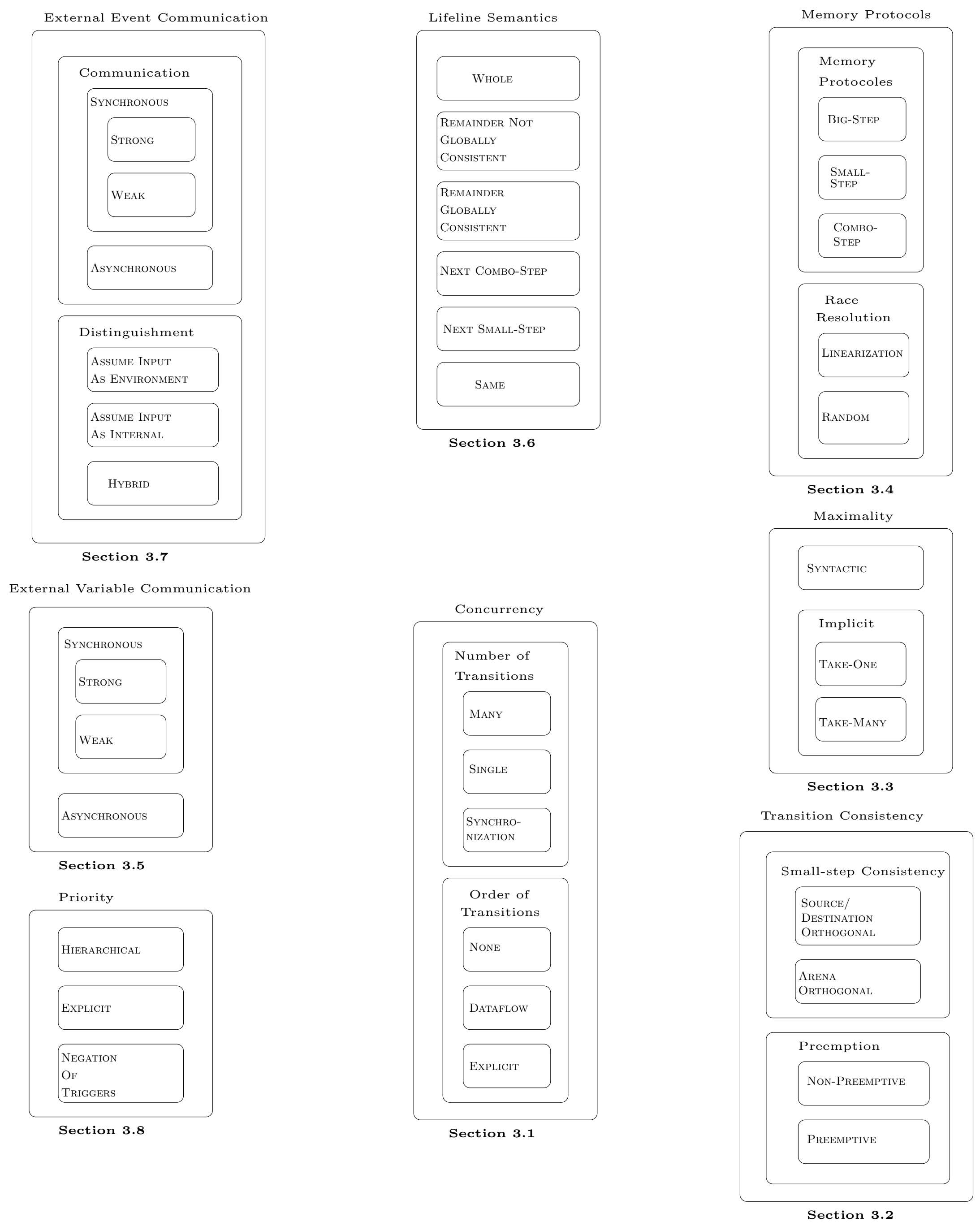


Figure 4: Semantic aspects and their semantic options.

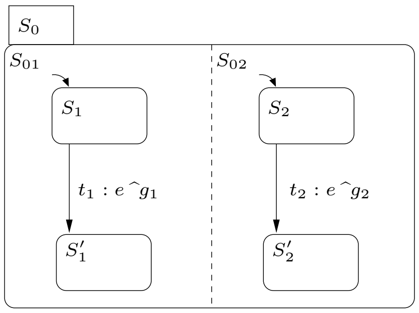


Figure 5: A model including an Concurrent-state.

to consider the more complicated model of execution where the simultaneous effects of more than one transition occurrences (e.g., the effect of race condition needs to be considered [we consider race conditions in Section 3.4.3]).

We only consider the MANY semantics that always take all of the enabled transitions in a snapshot, rather than non-deterministically choosing to take a subset of the set of enabled transitions. The latter semantics can be a source of undesired non-determinism, which is not suitable for the purpose of specification. However, such a semantic option can be useful when considering concurrency in the context of models of computations [35, 60].

Example 1 *Consider the model in Figure 5 and its snapshot* $\mathrm{sp} = (\{\mathrm{S}_1, \mathrm{S}_2\}, *, \{\mathrm{e}\})$*. If the semantic option* SINGLE *is chosen, then either* $\mathrm{sp} \xrightarrow{\{\mathrm{t}_1\}} (\{\mathrm{S}_1', \mathrm{S}_2\}, *, *)$*, or* $\mathrm{sp} \xrightarrow{\{\mathrm{t}_2\}} (\{\mathrm{S}_1, \mathrm{S}_2'\}, *, *)$*, but not* $\mathrm{sp} \xrightarrow{\{\mathrm{t}_1,\mathrm{t}_2\}} (\{\mathrm{S}_1', \mathrm{S}_2'\}, *, *)$*. If semantic option* MANY *is chosen, then only* $\mathrm{sp} \xrightarrow{\{\mathrm{t}_1,\mathrm{t}_2\}} (\{\mathrm{S}_1', \mathrm{S}_2'\}, *, *)$ *is possible.*

SYNCHRONIZATION: This option combines the SINGLE and MANY options. In this option, the SINGLE option is the default concurrency semantics, but when a *synchronization* happens, all of the transitions that are involved are executed together. A *synchronization* happens when some of the transition occurrences of a small-step provide the triggering events of some other transition occurrences of the same small-step. We consider the semantics of event communication of the synchronization mechanism in Section 3.6.1.[6] An **advantage** of this option is that through its concurrency semantics, when there is no event communication it provides the simple concurrency semantics of SINGLE, and when there is an event communication it facilitates for event communication within a small-step. A **disadvantage** of this option is that a modeller should always be aware of the fact that the execution of a transition might lead to an act of synchronization.

Synchronization can be optional or mandatory. In the OPTIONAL synchronization, which is similar to the "|" composition operator in CCS [46], it is not required that two transitions synchronize if they can, and the model can choose to take a small-step without synchronization. In the MANDATORY synchronization, which is similar to the "‖" composition operator in CSP [30], a transition that generates a *shared* event can only be taken if the transition(s) that are triggered by it in other parts of the model are taken too. In process algebras, the shared events of a model can be specified per composition operator (e.g., as in LOTOS [33]) or per model (e.g., as in CSP [30]). In CCS [46], a syntactic construct called *restriction* can be applied to an optional composition operator, which forces the two components of the composition to synchronize, and behave according to the MANDATORY synchronization.[7] In the examples where we use the MANDATORY synchronization, we assume that the shared events of a model are all of the events that are generated by an

[6]We only consider the semantics of synchronization so far as it is related to the semantics of concurrency in BSMLs, and do not consider process algebras [5], whose languages have some specialized syntax for synchronization, which allow for a more elaborate treatment of synchronization over the terms of a process algebra notation. In process algebras, such as CSP [30] and CCS [46], instead of events, there are similar notions such as *alphabets*[30], and CCS *labels* and *co-labels*[46].

[7]More precisely, the restriction operator in CCS [46] is an operator that is applied to a part of a model, along with a specified set of *restricted* events, and requires any transition in that part of the model that generates or is triggered with a restricted event to be only taken if it is getting synchronized with its complementing transition occurrence.

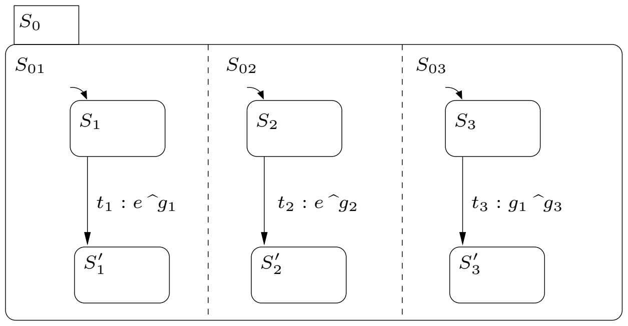


Figure 6: Synchronization Example.

HTS and used by the other, which means that, for example, an event might be a shared event of two HTSs H_1 and H_2, but not HTSs H_1 and H_3.

Example 2 *Consider the model in Figure 6. Transition* $\mathrm{t_1}$ *generates an event that is the triggering event of transition* t_3*; if we assume that there is a* MANDATORY *synchronization semantics (*g_1 *is the only shared action), then from snapshot* $\mathrm{sp} = (\{\mathrm{S_1}, \mathrm{S_2}, \mathrm{S_3}\}, *, \{\mathrm{e}\})$*, the following small-steps are possible:* $\mathrm{sp} \xrightarrow{\{\mathrm{t_1}, \mathrm{t3}\}} (\{\mathrm{S_1'}, \mathrm{S_2}, \mathrm{S_3'}\}, *, *)$*, or* $\mathrm{sp} \xrightarrow{\{\mathrm{t_2}\}} (\{\mathrm{S_1}, \mathrm{S_2'}, \mathrm{S_3}\}, *, *)$*. If we choose the* OPTIONAL *synchronization semantics, then additionally, small-step* $\mathrm{sp} \xrightarrow{\{\mathrm{t_1}\}} (\{\mathrm{S_1'}, \mathrm{S_2}, \mathrm{S_3}\}, *, *)$ *is possible.*

A possible semantic variation point for the synchronization semantics, regardless of the choice of the MANDATORY or OPTIONAL semantic option, is to consider a priority between taking a small-step that involves a synchronization over a small-step that does not, or vice versa (e.g., in the example above, giving priority to execute $\{t_1, t3\}$ over $\{t_2\}$, which does not involve synchronization). In process algebras, such a semantic variation is not considered for the general composition operators that permit both the synchronization small-steps and non-synchronization small-steps. However, a notion of *prioritized* actions is considered that allows for choosing the execution of small-steps with certain actions over other small-steps [13]. We study the priority semantics of BSMLs in Section 3.8.

Another possible semantic aspect could have been the choice between two-way vs. multi-way synchronization. But we do not consider this aspect as a primitive semantic aspect, because the two options have similar semantics. A language might have a binary and an n-ary synchronization operator, one for a two-way and one for a multi-way synchronization, respectively. In the remainder of this report, we assume an n-ary synchronization semantics, which is compatible with the notion of broadcast communication in BSMLs.

3.1.2 Order of Transitions

Within a big-step, the most common choice is to let transition occurrences happen when transitions are enabled (i.e., no defined order for transitions, semantic option NONE), but some BSMLs are more restrictive.[8]

NONE: Many BSMLs do not require an order for the execution of the transition occurrences of a big-step (e.g., Statecharts [22], P&S Statecharts [54], H&P&S&S Statecharts [26], Statemate [24], and RSML [38]). An **advantage** of this approach is that a modeller need not worry about the specification of the order of the execution of transition occurrences.[9] A **disadvantage** of this semantic option is that

[8]Orders, and particularly partial orders, between the actions of a model have been extensively used to describe the concurrent behaviours of models [56, 36, 35], but in this semantic aspect, we are interested in a notion of order that enforce a policy on how the sequence of small-steps should be organized, rather than using orders to describe the entire concurrent execution of a model, as in [56, 36, 35].

[9]Of course, the enabledness of variable conditions and/or event triggers of transitions can induce an order between the transitions of a big-step, but here, by order we mean an extra explicit level of control over how transitions can be sequenced.

a modeller needs to consider the effect of non-deterministic orders of the execution of enabled transitions, which can make the analysis of a model difficult.

DATAFLOW: Some BSMLs specify an order for transition occurrences in a big-step, by considering the *dataflow* order between the variables that are used in the transitions of a big-step (e.g., SCR [27], Lustre [21], Reactive Modules [3], and Signal [4]). A variable x appears before variable y in the dataflow order of a big-step, if the value assigned to x depends on the value assigned to y, either directly by using y in the RHS of the assignment to x, or indirectly through transitivity of dataflow order. Two variables in a big-step are *independent*, if they are not related through dataflow order. As such, a dataflow order can be a partial order, instead of a total order. If there is more than one assignment to the same variable in a big-step, then there would be an ambiguity about which assignment should be considered in the dataflow order of the big-step. Therefore, most of the BSMLs that use this semantic option only permit big-steps that assign a value to a variable at most once [3, 27, 21, 4].

For better understandability of models, it is desired that the dataflow order of variables of a model can be inferred by reviewing its syntax. However, depending on the characteristics of a notation, this can be difficult, because: (i) each big-step might introduce a different dataflow order between the variables, and (ii) identifying all of the big-steps of a model from its syntax is not feasible. As such, some BSMLs use a explicit syntax to specify the dataflow order between the variables of a model; we call such semantics EXPLICIT DATAFLOW semantics (e.g., SCR [27]). Alternatively, some other BSMLs do not have a syntax for the explicit specification of the dataflow order of a model, but permit models that have at most one assignment to a variable. The RHS of such an assignment syntactically specifies the variables that it expects to be assigned a value during a big-step, before the assignment can take place. This means that each variable assignment induces a dataflow order, and the dataflow orders of all assignments together induce the dataflow order of the model; we call the semantics of such BSMLs the IMPLICIT DATAFLOW semantics (e.g., Lustre [21], Reactive Modules [3], and Signal [4]). (In Section 3.4.2, we will consider the semantics of syntactic keywords that can prefix a variable to specify that the newly assigned value of the variable is needed in the RHS of an assignment.)[10]

An **advantage** of the DATAFLOW option is that a modeller can specify an order in the model where a variable is guaranteed to be updated before being used in the RHS of an assignment. Furthermore, as an **advantage**, by the definition of a dataflow order that requires a variable to be assigned value at most once during a big-step, big-steps are guaranteed to have finite number of small-steps. A **disadvantage** is that this option is less reactive in that the set of transitions that can be taken in an big-step is more restricted, and this must be considered in creating the model. Another **disadvantage** is that a modeller can mistakenly create a cyclic order between the transitions of a model when two transition occurrences wait for each other's assignments.

Example 3 *The model in Figure 7 consists of a Concurrent-state and two transitions* t_1 *and* t_2 *that assign values to variables* y *and* x*, respectively. If we assume that the model is specified in a semantics that supports* MANY *semantic option, and the model is in configuration* $\{\mathrm{S}_1, \mathrm{S}_2\}$*, then one would expect that* t_1 *and* t_2 *to execute together. However, if we assume that the semantics support the* EXPLICIT DATAFLOW *semantics and in this model* y *is less than* x *in the dataflow order, then* t_1 *can only be executed after* t_2*, allowing* t_1 *to read the updated value of* x.

EXPLICIT: There is a syntactical order that explicitly specifies the order of the execution of enabled transitions. In Stateflow [14], for example, the way the HTSs of a model are ordered graphically clockwise specifies the order in which each HTS can execute a transition during a big-step. This semantic option has the **advantage** of giving great control over how transitions should be executed, but it has the **disadvantage** that a modeller needs to consider constantly the specification of the order of execution, even when the order should not matter.

[10]A variation of the IMPLICIT DATAFLOW semantics would permit models with more than one transition that assign values to a same variable. But there should be a guarantee that regardless of which transition is executed in a big-step, the induced dataflow order remains the same (i.e., the assignments to a same variable in different transitions use the same set of variables in their RHSs). Furthermore, there should be a guarantee that only one of the transitions is executed in each big-step, which for example can be enforced by requiring the enabledness of the transitions to be pairwise mutually exclusive.

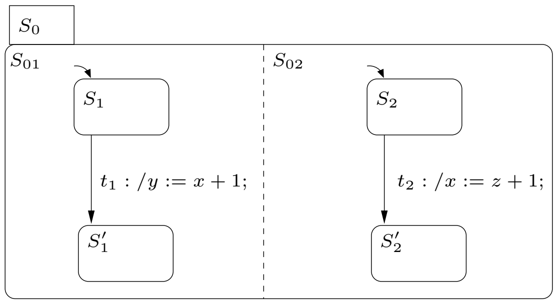


Figure 7: A model with dataflow order.

Example 4 *For example, if we consider a semantics that supports a clock-wise order of execution within a Concurrent-state, then if model in Figure 5 resides in snapshot* $\mathrm{sp} = (\{\mathrm{S}_1, \mathrm{S}_2\}, *, \{\mathrm{e}\})$*,* t_1 *should be executed before* t_2 *can be executed.*

If a semantics allows for both explicit and dataflow ordering of the transition occurrences of a big-step, then these orders may conflict, which is an undesirable and should be avoided, because it is a source of confusion for modellers. Such conflicts can be either avoided by analysis of a model, or by explicitly specifying that one order has higher precedence than another (e.g., the EXPLICIT order has a higher priority than the DATAFLOW order).

In this section, we did not consider the technical details of how different concurrency semantics are implemented. But as shown in [48], by using a variety of composition operators (e.g., parallel, interleaving, interrupt, rendezvous, etc.), these semantics can be implemented.

3.2 Transition Consistency

Many BSMLs have a notion of a *control state*, which is usually shown graphically as a rounded box or an ellipse (e.g., Argos [41], Statechart [22] and many of its variants [64], Statemate [24], and RSML [38]). As in an automaton, a control state is a named syntactic artifact that a modeller uses to represents a noteworthy moment in the execution of a model.[11] Such a moment is an abstraction that groups together the past behaviours (consisting of inputs received by the model and the model's past reactions to these inputs) that have a common set of future behaviours. By using a control state, a modeller can describe future behaviour in terms of the current control state and the input, which abstracts the fact that current behaviour depends on past behaviours. If a model's reaction to an environmental input is always independent of its past behaviours, then the notion of control state is not useful for the model (the notion of hierarchy of control states might still be useful for specifying priority between transitions; see Section 3.8 for priority semantics).

If a BSML lacks syntax for control states, the same information can be captured in the model using variables. For example, in Reactive Modules [3], variables can be used to store the information about the history of computation. The values of such variables are carried from one big-step to another, as opposed to *history-free* variables [3], whose values in the previous big-step are not used in the computation. The notion of control state can be realized textually in the form of a line of a program. BSMLs such as Esterel [9, 2] have a notion of interrupt transition and the scope of an interrupt transition, specified by `exit` and `trap` statements respectively [9, 2], which can be modelled via hierarchical control states. Conversely, control states can be translated into a textual syntax, but with possibly more complication than the converse translation (e.g., translation of deterministic Statecharts to Esterel [58]).

The syntax of some BSMLs allows for specifying computation within a control state; we call those control states *executable control states.* If the computation within an executable control states is specified as a set

[11]The syntax of BSMLs are similar to the syntax of Mealy Automata [42], which allow a transition to have both an event trigger and a generated event, as opposed to regular automata.

Option Description	Advantages	Disadvantages	Example
Small-Step Consistency (also possible at big-step level)			
Arena Orthogonal	Simplicity	Undesired non-determinism	5, 7
Source/Destination Orthogonal	Deterministic choice of small-steps	Complicated to determine destination	6, 7
Preemption Semantics			
Non-Preemptive	Supports "last wish"	jump-like flow of control	7
Preemptive	Easy flow of control	Does not support "last wish"	7

Table 4: Transition Consistency Semantics.

of operations (i.e., a set of assignments and event generations), then the semantics of the computation of an executable control state is similar to the semantics of a transition occurrence. If the computation of an executable control state is a sequence of operations, then if they are executed in one small-step, then they can be converted to a set of operations [39, 37], but if they are not, then various semantic options arise, which we defer their investigation to our future work.

In the remainder of this section, we consider the *transition consistency* semantic aspect that specifies the different ways that a set of transitions can be taken together in a small-step, based on their source and destination control states. There are two semantic sub-aspects that together specify whether a set of transitions can be taken together in a small-step or not. Table 3.2 lists the two semantic aspects and their options. The two semantic aspects are relevant when the Many concurrency semantics is chosen. The *small-step consistency* semantics is relevant for a pair of transitions when one does not *interrupt* the other, where as the *preemption* semantics is relevant for them when one does *interrupt* the other.

3.2.1 Small-Step Consistency

When control states are arranged in a hierarchy, it gives rise to options regarding the semantics of transitions that exit concurrent and hierarchical states. This semantic aspect specifies whether a pair of transitions whose source control states are orthogonal and whose destination control states are orthogonal can be taken together in a small-step. There are two semantic options.

Arena Orthogonal: In this option, used in the original Statecharts [22, 26] and many of its variants [64], two transition occurrences are included in the same small-step only if their *arenas* are orthogonal, where the *arena* of a transition is the smallest (lowest in the hierarchy of composition tree) Or-state that is the (grand)parent of the source and destination control states of the transition [55].

Example 5 *Consider the model in Figure 8. If the model resides in snapshot* $(\{\mathrm{S_2}, \mathrm{S_3}, \mathrm{S_4}\}, \{\mathrm{x} = 1\}, \{\mathrm{e}\})$*, and the* Arena Orthogonal *is considered, along with the* Many *concurrency model, then the possible small-steps are:*

$$(\{S_2, S_3, S_4\}, \{x = 1\}, \{e\}) \xrightarrow{\{t_6\}} (\{S_5\}, \{x = 2\}, *),$$

$$(\{S_2, S_3, S_4\}, \{x = 1\}, \{e\}) \xrightarrow{\{t_1,t_2\}} (\{S_2', S'3, S_4\}, \{x = 2\}, *),$$

and

$$(\{S_2, S_3, S_4\}, \{x = 1\}, \{e\}) \xrightarrow{\{t_3\}} (\{S_2, S_3, S_4'\}, \{x = 1\}, *).$$

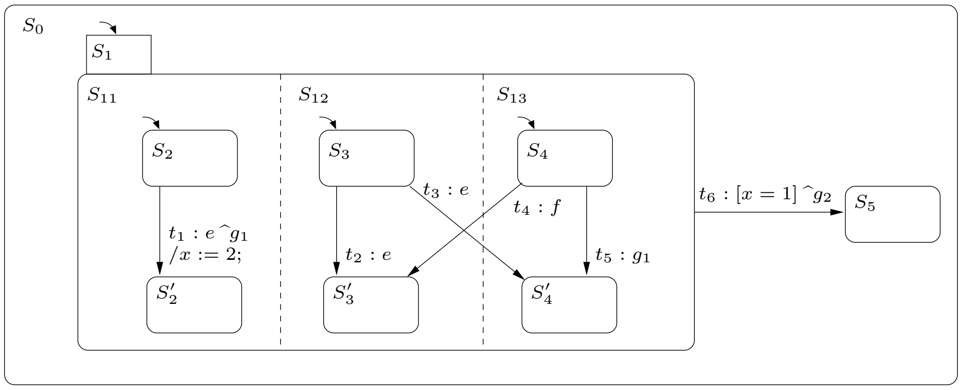


Figure 8: A model with concurrent and hierarchical transitions.

An advantage of the Arena Orthogonal option is its simplicity. The simplicity of the Arena Orthogonal can be a **disadvantage**, because it can introduce non-determinism as to which enabled transition should be taken, when there is more than one transition enabled within an Concurrent-state and each of them has a source and destination control state belonging to different children of the Concurrent-state (e.g., in example 5 above, t_1 and t_3 cannot be taken together because their arenas are not orthogonal, but they might be taken one after the other in subsequent small-steps, in a non-deterministic order).

Source/Destination Orthogonal: In this semantic option, a small-step is consistent if for any of its two distinct transition occurrences, their sources and destinations are pairwise orthogonal (e.g., Statemate [24] and UML Statemachines [50]).

Example 6 *Consider the model in Figure 8, but this time assume* Source/Destination Orthogonal *semantics, along with the* Many *concurrency model, if the model resides in snapshot* $(\{\mathrm{S}_2, \mathrm{S}_3, \mathrm{S}_4\}, \{\mathrm{x} = 1\}, \{\mathrm{e}\})$*, then the possible small-steps are:*

$$(\{S_2, S_3, S_4\}, \{x = 1\}, \{e\}) \xrightarrow{\{t_6\}} (\{S_5\}, \{x = 2\}, *),$$

$$(\{S_2, S_3, S_4\}, \{x = 1\}, \{e\}) \xrightarrow{\{t_1,t_2\}} (\{S'_2, S'3, S_4\}, \{x = 2\}, *),$$

and

$$(\{S_2, S_3, S_4\}, \{x = 1\}, \{e\}) \xrightarrow{\{t_1,t_3\}} (\{S'_2, S_3, S'_4\}, \{x = 1\}, *).$$

Which, as opposed to the Arena Orthogonal *option,* t_1 *and* t_3 *can be taken together.*

More formally, a small-step is consistent according to the Source/Destination Orthogonal option, if for any of its two distinct transition occurrences t and t': (i) their sources are orthogonal, (ii) their destinations are orthogonal, and (iii) the destination of t and the source of t' are orthogonal, and vice versa for the destination of t' and the source of t. The third condition disallows a transition occurrence to enter the arena of another transition occurrence; for example, in Figure 8, if t_3 and t_5 are both enabled, they cannot be taken together because their destinations are not orthogonal.

An **advantage** of the Source/Destination Orthogonal is that it avoids the non-determinism of the Arena Orthogonal option by permitting a small-step to include two transition occurrences whose sources and destinations belong to two different children of the Concurrent-state. A **disadvantage** of this option is that determining the destination of a small-step is not as simple as the previous option. To determine the destination configuration of a small-step, the effect of a transition occurrence in the small-step whose

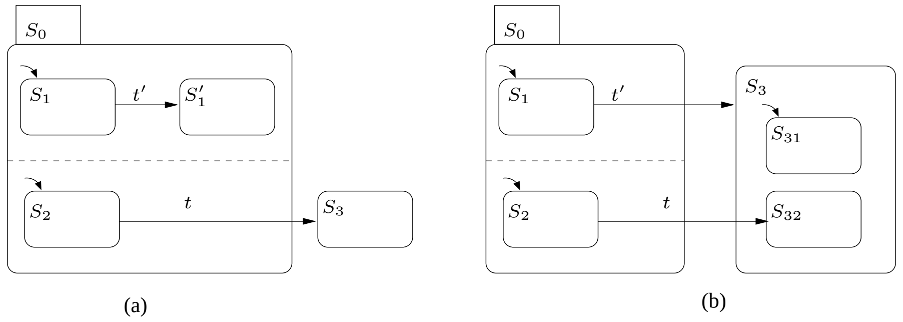


Figure 9: Preemption semantics is relevent when the models reside in their $\{S_1, S_2\}$ configurations.

source and destination belong to different HTSs needs to be considered, in which case the destination of the small-step includes the default state of the source HTSs too.

Some BSMLs define their semantics for transition consistency at the scope of a big-step, instead of a small-step. The same two semantic options as above are available for transition consistency at the big-step level. For example, in P&S Statechart [55], the semantic option ARENA ORTHOGONAL is chosen for big-steps, which conceptually implies that if we flatten the sequence of small-steps into a set of transition occurrences, then they can be all taken together, in the spirit of synchrony hypothesis.

3.2.2 Interrupt Transitions and Preemption

The notion of *preemption* is relevant when two transitions are enabled, their sources are orthogonal, and one of the following conditions holds:

– either the destination of one of the transitions is orthogonal with the source of the other, and the destination of the other transition is not orthogonal with the sources of neither transitions; we call the latter transition to be an *interrupt for* the former transition; or

– the destination of neither transitions is orthogonal with the sources of the two transitions, but their destinations are ancestrally related; we call the transition that has a destination that is (grand)child of the other an *interrupt for* the other.

Figure 9 (a) and (b) illustrate the above two possibilities for preemption semantics. Assume the models reside in their $\{S_1, S_2\}$ configurations. In Figure 9 (a), t is an interrupt for t' because the destination of t is not orthogonal with the source of t and t'. In Figure 9 (b), t is an interrupt for t' because the destination of t and t' are not orthogonal with their sources, and the destination of t is a child of the destination of t'.

PREEMPTIVE and NON-PREEMPTIVE: The preemption semantics of a BSML specifies whether two transitions can be taken together in the same small-step when one is an interrupt for the other. The preemption semantics of a BSML is PREEMPTIVE, if it does not permit a small-step to include such two transitions, and is NON-PREEMPTIVE otherwise. The "preemptive" terminology is used because in a PREEMPTIVE semantics by executing a transition t that is an interrupt for another enabled transition t' the execution of t "preempts" the execution of t'. Examples of the NON-PREEMPTIVE semantics are Argos [41][12] and Esterel [9]. In Esterel [9], an `exit` statement can be an interrupt for another transition. Esterel, through special syntax, allows for both PREEMPTIVE and NON-PREEMPTIVE semantics. Our notions of PREEMPTIVE and NON-PREEMPTIVE semantics are similar to the notion of *strong preemption* and *weak preemption* in [7].

[12]Argos [41] has a different notion of hierarchical states than other BSMLs with graphical syntax. In Argos, a transition with a source on an Or-state, say Or-state S, is an interrupt for a transition whose arena is S or a child of S. We can translate this notion of Or-state and "interrupt for" relation to our framework by turning an Or-state with an interrupt transition to an Concurrent-state with two children: one representing the original Or-state without the interrupt transition, and another having one transition that models the interrupt transition.

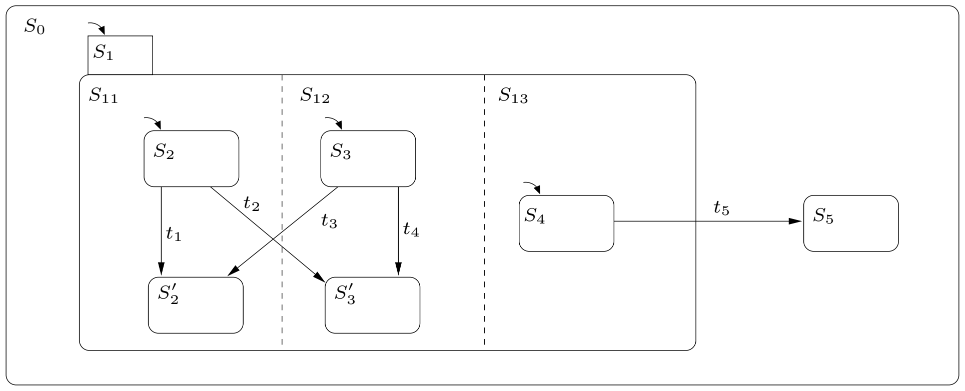


Figure 10: Non-preemptive semantics and small-step consistency semantics, considered together.

The preemption semantics of a BSML is independent of its small-step consistency semantics (each semantic aspect deals with a different pair of transitions). The set of transitions of a small-step can be taken together, only if for each pair of transitions in the small-step, they are either small-step consistent or they satisfy the preemption semantics of the BSML.

The destination configuration of a small-step for a BSML that supports theNON-PREEMPTIVE preemption semantics is not necessarily determined by graphically following the destination of its constituent transitions. For two transitions t and t' of a small-step, if t is an interrupt for t', then the destination control state of t' is not relevant, and the destination of t overrides it. For example, in the model in Figure 9 (a), if t and t' are executed together in a small-step, the destination of the small-step is S_3. Similarly, for the model in Figure 9 (b), if t and t' are executed together in a small-step, the destination of the small-step is S_{32}.

Example 7 *Consider the model in Figure 10. If the model resides in* $\{\mathrm{S}_2, \mathrm{S}_3, \mathrm{S}_4\}$ *configuration, then* t_5 *is an interrupt for* t_1, t_2, t_3, *and* t_4. *If we assume the* NON-PREEMPTIVE *preemption semantics, along with the* MANY *concurrency semantics, regardless of the small-step consistency semantics (as it happens in this example), the following three big-steps are possible:*

$$(\{S_2, S_3, S_4\}, *, *) \xrightarrow{\{t_1, t_4, t_5\}} (\{S_5\}, *, *),$$

$$(\{S_2, S_3, S_4\}, *, *) \xrightarrow{\{t_2, t_5\}} (\{S_5\}, *, *),$$

and

$$(\{S_2, S_3, S_4\}, *, *) \xrightarrow{\{t_3, t_5\}} (\{S_5\}, *, *).$$

An **advantage** of the NON-PREEMPTIVE option is that the "last wish" of a Concurrent-state can be satisfied when a transition whose source belongs to the state exits it. A **disadvantage** of the NON-PREEMPTIVE option is that specifying the destination configuration of a small-step is complicated, which can make the task of analysis/understanding of a model very difficult.

Option Description	Advantages	Disadvantages	Example
SYNTACTIC	Easy-to-use syntax for specifying and identify a big-step and its scope	Non-terminating big-steps	8
IMPLICIT			
TAKE ONE	Ease of use, in spirit of synchronous hardware, and understandability	Unclear scope for big-steps (especially when sequence of big-steps are considered)	9
TAKE MANY	Freedom of specifying a big-step through multiple transitions	Non-terminating big-steps, and unclear scope for big-steps (even when one big-step is considered)	9

Table 5: Maximality Semantics (for big-steps and combo-steps).

3.3 Maximality

BSMLs need to have a maximality condition that specifies when the sequence of small-steps of a big-step ends (i.e., when the model becomes *stable*, and is ready to sense the environment for new inputs). We taxonomize maximality semantic options based on: whether there is a *syntactic* mechanism that specifies the configuration in the execution where the model becomes stable, or there is an *implicit* mechanism that specifies when a big-step should end, without relying on any particular syntactic construct. Table 5 lists the semantic variations for maximality semantics.

SYNTACTIC: A big-step becomes stable when a configuration is reached during the execution of the big-step, where the model, or part of the model, syntactically ends its execution, and is ready to sense the environment. If a BSML uses Concurrent-states in its syntax, then a model becomes stable, when all of its concurrent components (all of its HTSs) become stable. In Esterel [9], for example, when a `pause` statement is reached in an HTS of a model, then the execution of the big-step ends in that HTS; the entire model becomes stable when all its HTSs are stable. In Rhapsody [23] and UML Statemachines [50, 10], when the execution of a *compound transition* of an *object* finishes, then it means that the current big-step for the object has ended; the entire system stabilizes when all objects in the model stabilize.

An **advantage** of the SYNTACTIC option is that a modeller can easily identify the syntactic scope of a big-step (which provides an easy-to-understand mechanism to keep track of the big-steps of a model, and analyze them). A **disadvantage** of this approach is that it may lead to non-terminating small-steps, if there exists an execution that never reaches a point in the model to receive environmental inputs.

In our normal form, and our examples for the SYNTACTIC option, a big-step becomes stable when a control state that is syntactically designated to end the big-step is entered. Similarly, a designated syntax can be used to annotate a transition as a transition whose occurrence ends a big-step (or conversely, starts a big-step). A model specified in a BSML that uses transitions for SYNTACTIC maximality can be translated into a model in a similar BSML that uses control states for SYNTACTIC maximality (the translation has the cost of introducing new control states). A **disadvantage** of using transitions for SYNTACTIC maximality is that it is possible to arrive at a control state where a big-step can optionally continue a big-step without sensing the environment, or end the big-step; this situation can be avoided by requiring some syntactical, conservative well-formedness conditions. If a well-formedness condition is not enforced, it is possible, although difficult and in presence of variables undecidable, to identify the models that have such problems.

Example 8 *The model in Figure 11 specifies a communication system.*[13] *The Concurrent-state* S_{01} *senses the environment, and if there is a* msg*, it asks* S_{02} *to* transmit *a message from the input channel to an output*

[13]This example is inspired by the running example of [16].

channel. In case of failure, when err *is received by* S_{02} *and a* nack *is sent to* S_{01}*,* S_{01} *will move to state* S_1''*, where it waits for a* reset *event from the environment. In case of success, when* succ *is received by* S_{02} *and an* ack *is sent to* S_{01}*, the system can start another send operation. We have used our own syntax for specifying the end of an environment: a control state that is marked with a "✓" (i.e.,* S_1*,* S_2*, and* S_3*) indicates that upon entering it, its corresponding HTS stabilizes. Events* msg*,* reset*, and* sent *are "input" events that are received from the environment, at the beginning of a big-step.*

In this example, we assume the Single *concurrency semantics, and an event semantics that assumes input events and internal events, once generated, persist during a big-step (described in sections 3.6 and 3.7).*

If we start from snapshot $(\{\mathrm{S}_1, \mathrm{S}_2, \mathrm{S}_3\}, \{\mathrm{full} = \mathrm{false}\}, \{\mathrm{msg}\})$*, then the following execution trace can be executed:*

$$\begin{array}{ll}
(\{S_1, S_2, S_3\}, \{full = false, cap = *\}, \{msg\}) & \xrightarrow{t_9} \\
(\{S_1, S_2, S_3'\}, \{full = false, cap = *\}, \{msg\}) & \xrightarrow{t_1} \\
(\{S_1', S_2, S_3'\}, \{full = false, cap = *\}, \{msg, send\}) & \xrightarrow{t_5} \\
(\{S_1', S_2', S_3'\}, \{full = false, cap = *\}, \{msg, send, transmit\}) & \xrightarrow{t_{10}} \\
(\{S_1', S_2', S_3\}, \{full = *, cap = *\}, \{msg, send, transmit, succ\}) & \xrightarrow{t_7} \\
(\{S_1', S_2, S_3\}, \{full = *, cap = *\}, \{msg, send, transmit, succ, ack\}) & \xrightarrow{t_2} \\
(\{S_1, S_2, S_3\}, \{full = *, cap = *\}, \{msg, send, transmit, succ, ack\}). &
\end{array}$$

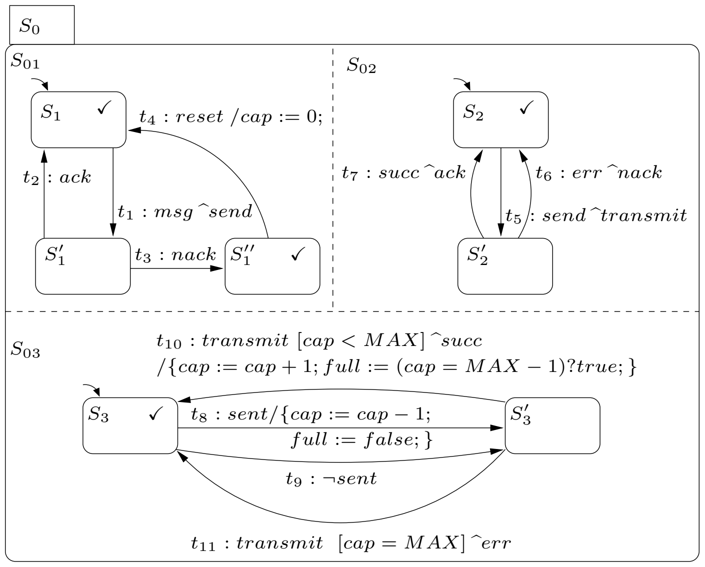
The above big-step is maximal, because all HTSs arrive at their stable control states. If the output channel is full, then the following execution would happen:

$$\begin{array}{ll}
(\{S_1, S_2, S_3\}, \{full = true, cap = MAX\}, \{msg\}) & \xrightarrow{t_9} \\
(\{S_1, S_2, S_3'\}, \{full = false, cap = *\}, \{msg\}) & \xrightarrow{t_1} \\
(\{S_1', S_2, S_3'\}, \{full = true, cap = MAX\}, \{msg, send\}) & \xrightarrow{t_5} \\
(\{S_1', S_2', S_3'\}, \{full = true, cap = MAX\}, \{msg, send, transmit\}) & \xrightarrow{t_{11}} \\
(\{S_1', S_2', S_3\}, \{full = true, cap = MAX\}, \{msg, send, transmit, err\}) & \xrightarrow{t_6} \\
(\{S_1', S_2, S_3\}, \{full = *, cap = *\}, \{msg, send, transmit, err, nack\}) & \xrightarrow{t_3} \\
(\{S_1'', S_2, S_3\}, \{full = *, cap = *\}, \{msg, send, transmit, err, nack\}). &
\end{array}$$

The above big-step is maximal, because all HTSs become stable, this time S_{01} *goes to its* S_1'' *control state, waiting for a* reset *event from the environment.*
By replacing the order of the execution of t_9 *and* t_1*, for each of the scenario above, another big-step, with the same final snapshot as above, can be derived.*

Implicit: There are two implicit ways to define the maximality semantics of a BSML, without using any specific syntactic constructs to specify the scope of a big-step.

Take One: This semantic option considers a big-step maximal if either there are no more enabled transitions to be taken, or there does not exist any enabled transition whose source belongs to the configuration of the system at the beginning of the big-step and it has not been exited during the current big-step. This semantic option roughly equates to a semantics that allows each HTS of a model to take at most one transition. But if a transition is considered whose source is a (grand)parent of multiple HTSs, there are two semantic variations to be considered: (i) the execution of the transition is counted toward the quota of the HTSs that are (grand)children of the source of the transitions, or (ii) the execution of the transition is not counted towards the quota of such HTSs. In this report, we follow the (i) semantic variation. Examples of this semantic option are: Statecharts [22, 26] (and many of its variants [64], including P&S Statecharts [55]), Reactive Modules [3], and Argos [41]. Some of the BSMLs that support the Take One option are influenced with the principles of synchronous hardware, which assumes that during a big-step, a non-concurrent part of a model can only take one transition; or alternatively, in synchronous hardware terminology: each hardware component reacts once during a clock tick.



S_0
S_{01}
S_{02}
S_{03}
S_1 ✓
S_2 ✓
S_1'
S_1'' ✓
S_2'
S_3 ✓
S_3'
$t_4 : reset\ /cap := 0;$
$t_2 : ack$
$t_1 : msg\ \hat{}\ send$
$t_3 : nack$
$t_7 : succ\ \hat{}\ ack$
$t_6 : err\ \hat{}\ nack$
$t_5 : send\ \hat{}\ transmit$
$t_{10} : transmit\ [cap < MAX]\ \hat{}\ succ$
$/\{cap := cap + 1; full := (cap = MAX - 1)?true;\}$
$t_8 : sent/\{cap := cap - 1;$
$full := false;\}$
$t_9 : \neg sent$
$t_{11} : transmit\ \ [cap = MAX]\ \hat{}\ err$

Figure 11: Syntax "✓" specifies that the model must sense the environment.

An **advantage** of the TAKE ONE approach is that if the source configuration of a big-step is identified, then a modeller can easily follow the execution of a big-step to the next configuration, because each HTS of the model can contribute at most one transition occurrence to a big-step. A **disadvantage** of this approach is that, considering a sequence of environmental inputs and their corresponding big-steps, a modeller cannot easily keep track of the scope of big-steps, as opposed to the SYNTACTIC option, which specifies the scope of a big-step clearly.

TAKE MANY: This semantic option continues the sequence of small-steps, until there is no more small-steps to be taken. Examples of this semantic option are Statemate [24] and RSML [38]; Statemate [24] provides the user of its tool-set both the semantic option of TAKE ONE, which is called a *step*, and the semantic option of TAKE MANY, which is called *super-step*.

An **advantage** of this semantic option is that a user does not need to squeeze the behaviour of a HTS of a model to an environmental input into one transition, as is necessary in TAKE ONE. A **disadvantage** of this semantic option is that it may lead to a non-terminating sequence of small-steps. Furthermore, similar to the TAKE ONE option, when reviewing a model, it is far from clear what the syntactic scope of a big-step is; this effect is even worse than the TAKE ONE option, because in this case, an arbitrary number of transition occurrences are possible for each big-step.

Example 9 *The model in Figure 12 shows a two-bit counter.*[14] *Control states* S_{01} *and* S_{02} *model the least and most significant bits of the counter, respectively. Each time the input event* tk_0*, which is the tick of a clock, is sensed as present, the Or-state* S_{01} *does a transition. After even number of ticks,* S_{01}*, by sending event* tk_1*, instructs* S_{02} *to toggle its status. When the counter finishes counting four clocks, it generates a* done *event. We assume the* SINGLE *concurrency semantics, and the event semantics that assumes the input events and the internal events, once generated, persist during a big-step. If we assume the* TAKE ONE *semantic option, then starting from snapshot* $(\{\mathrm{S}_1, \mathrm{S}_2\}, *, \{\mathrm{tk}_0\})$*, the first big-step would be*

$$\begin{array}{l}(\{S_1, S_2\}, *, \{tk_0\}) \quad \xrightarrow{t_1} \\ (\{S_1', S_2\}, *, \{tk_0\}),\end{array}$$

which includes only one small-step.

[14]This example is adopted from [41], where a more elaborate version of it is used as the running example of the paper.

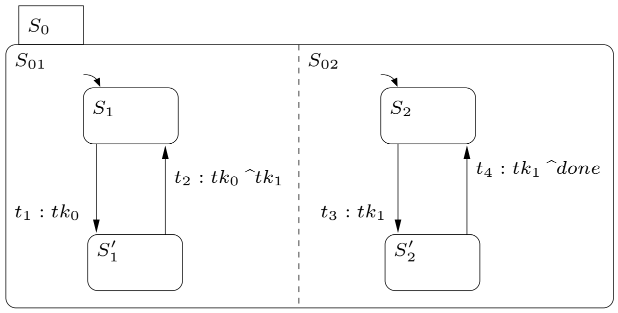


Figure 12: A model for a two-bit counter.

If we start the second big-step by sensing the environment and receiving the "new" tk_0 *(i.e., starting from* $(\{\mathrm{S}_1', \mathrm{S}_2\}, *, \{\mathrm{tk}_0\})$*), then the following big-step is produced:*

$$\begin{array}{ll} (\{S_1', S_2\}, *, \{tk_0\}) & \xrightarrow{t_2} \\ (\{S_1, S_2\}, *, \{tk_0, tk_1\}) & \xrightarrow{t_3} \\ (\{S_1, S_2'\}, *, \{tk_0, tk_1\}). & \end{array}$$

The third big-step includes one transition occurrence, and the fourth one including two transition occurrences, and generates the end *event.*

If we start from $(\{\mathrm{S}_1, \mathrm{S}_2\}, *, \{\mathrm{tk}_0\})$*, but this time assume the* Take Many *semantic option, then it is not possible to generate a terminating big-step. There are different ways to create a non-terminating big-step, one of them the following trace, which never gives* S_{02} *a chance to execute a transition:*

$$\begin{array}{ll} (\{S_1, S_2\}, *, \{tk_0\}) & \xrightarrow{t_1} \\ (\{S_1', S_2\}, *, \{tk_0\}) & \xrightarrow{t_2} \\ (\{S_1, S_2\}, *, \{tk_0, tk_1\}) & \xrightarrow{t_1} \\ \cdots . & \end{array}$$

3.3.1 Maximality of Combo-Steps

Similar to the notion of maximality of a big-step, the notion of maximality is needed for combo-steps, whenever there is a notion of combo-step in the semantics of a BSML. The maximality semantics of a combo-step specifies the extent of a contiguous segment of a big-step where computation is carried out based on reading fixed values of variables and/or events, without considering the changes to the values of variables and/or events that have occurred during the execution of the big-step. The same semantic options as the semantic options of big-steps are possible for combo-steps too. In practice, however, we are only aware of BSMLs that use the Take One option for combo-steps. RSML [38] and Statemate [24] use the Take One option for their combo-step maximality semantics and the Take Many option for their big-step maximality semantics. There is an obvious semantic constraint of disallowing a semantics that uses the Take One option for its big-step maximality semantics, and the Take Many option for its combo-step maximality semantics.

3.3.2 Non-reactiveness and Environmental Input Assumptions

A model is *non-reactive* if:

- the model is specified in a BSML that subscribes to the SYNTACTIC maximality semantics, it resides in a non-stable configuration of a big-step, and there is no small-step to be executed to reach a stable snapshot (this situation is different from a non-terminating big-step where there are small-steps that can be executed, but the model never reaches a stable configuration); or,
- the model resides in a configuration of the model that regardless of the input that the environment provides, it cannot produce a big-step.

Some BSMLs have syntactic well-formedness criteria that make it impossible for a model to be non-reactive. For example, for a BSML that supports the SYNTACTIC maximality semantics, if there is an *else* transition in all of its possible intermediate configurations that is always enabled, then it would never halt in an intermediate snapshot (this is the approach that UML Statemachines [50] and Rhapsody [23] have adopted for their compound transitions).[15]

In order to avoid the second type of non-reactiveness, which happens in a source snapshot of a big-step, the model needs to react to all possible inputs from the environment. Some BSMLs allow a modeller to specify the *input assumptions* of a model, which means that the model is only exposed to inputs that satisfy its input assumptions (e.g., Esterel [9] and SCR [27]). By limiting the number of scenarios that need to be considered for non-reactiveness analysis, the input assumptions of a model can significantly simplify the analysis of non-reactiveness for the model. A common input assumption that makes the analysis of a model and its non-reactiveness easy is the *single-input assumption.* The single-input assumption means that at each source snapshot, only one input can be received from the environment (e.g., one event can be received from the environment). The single-input assumption has the **advantage** of allowing a modeller to analyze a source snapshot of a model with respect to one input at a time instead of considering all combinations of all inputs, which is combinatorially larger number of cases to analyze. The **disadvantage** of the single-input assumption is that it might not be compatible with the reality of the domain of the system that is being modelled.

3.4 Memory Protocols

In a BSML, variables are mediums for carrying out computation, and also provide the means for a persistent communication mechanism between the different parts of a model. Variables are persistent artifacts: The value of a variable in a current big-step is carried from one big-step to the next, even if it is not assigned a value during the current big-step. The major semantic aspect of variables is how to obtain the value of a variable when it is accessed in a variable condition or in the RHS of an assignment; we call this semantic aspect of a BSML its *memory protocol.* We consider three memory protocols, which differ in when a *write* to a variable in a big-step can be sensed by a *read.* A "write" to a variable is an assignment to the variable, and a "read" from a variable is an access to it in a variable condition of a transition, or in the RHS of an assignment. We only consider global variables. A local variable can be treated as a global variable whose name is prefixed with an identifying scope.

With global variables and concurrency, race conditions arise. In Section 3.4.3, we consider race resolution mechanisms that can be used when multiple transitions within a small-step write to the same variable.

An *external variable* in a model is a variable that is assigned values by the environment of the model. In this section, we present the semantics of non-external variables, and in Section 3.5, we consider the semantics of external variables. Throughout the report, whenever we use the phrase "variable" without a prefix, we mean non-external variable. Table 6 lists the semantic choices for variables, along with their corresponding advantages and disadvantages.

[15] An "else" transition can be specified in different ways. For example, an "else" transition can be a transition without any event trigger/variable condition, which has the lowest priority compared to all transitions of a model (see Section 3.8 for priority semantics). Another way is to have an "else" transition that is only enabled when all other transitions are disabled.

Option Description	Advantages	Disadvantages	Example
Memory Protocol (same three options for variable conditions and RHS of assignments)			
Big-Step	Transitions do not interfere/disable each other and compatible with the synchrony hypothesis, and modular with respect to variables	Sequential operations hard to specify	10
Small-Step	Allows sequential computation	Transitions can affect/disable each other	10
Combo-Step	Transitions do not interfere/disable each other within a combo-step	Difficult to keep track of the values of variables	11

Table 6: Memory Protocol Semantics.

3.4.1 Memory Protocols

Next, we consider the three possible memory protocols for BSMLs. The semantics of a BSML can use different memory protocols for the variable conditions and the RHS of assignments.

Big-Step: In this memory protocol, a read from a variable during a big-step always returns the value of the variable at the beginning of the big-step, even if it has been assigned a value in the big-step (e.g., H&P&S&S Statecharts [22] and Reactive Modules [3]). When this option is used for variable conditions, an **advantage** of this semantic option is that the enabledness of the variable condition of a transition does not change throughout the big-step. When it is used for assignments, the assignments of the transitions of a big-step need not be considered according to the sequence of small-steps of the big-step; they can all be considered together in an order-independent way. This behaviour has the **advantage** of being compatible with the synchrony hypothesis, which considers a big-step as a set of transitions, rather than a sequence. The Big-step memory protocol is *modular* [32] with respect to variables, because a variable assignment within a big-step can be conceptually considered as a value assignment by the environment, which happens at the beginning of a big-step. Modularity is defined for events in [32], but, in the same spirit, we extend it to other parts of syntax too. A modular semantics has the **advantage** of allowing a model to be extended by adding a new part (e.g., an HTS), without worrying that the new extension interferes with the previous behaviour of the model (the extension can affect the behaviour of the model as much as environmental inputs can). In a non-modular memory protocol, a part of a model cannot play the role of the environment for another part, which means that a model cannot be constructed incrementally. Extensions of the model may change the behaviour in different ways than the environment does. Therefore, all parts of a model should be created together. A **disadvantage** of this memory protocol is that it does not allow a modeller to specify a computation sequentially by using a sequence of transitions (e.g., specifying an arithmetic computation in separate sequential steps, which is common practice in modelling).

Small-Step: This memory protocol uses the value of a variable as computed by the transitions in the previous small-step (e.g., Esterel [9], Lustre [21], and SCR [27]). When used for assignments, an **advantage** of this option is that a modeller can describe a sequence of computations in a sequence of transitions. When considered for variable conditions, a **disadvantage** of this memory protocol is that a transition can disable another transition, which means that a modeller, or a reviewer of a model, must check the enabledness of all transitions after the execution of each small-step.

Example 10 *The model in Figure 13, adopted from an example in [31], is meant to specify a computation that maintains the invariant that the value of* $\mathrm{a} - \mathrm{b}$ *is the same before and after the execution of a big-step. We assume the* Take One *semantic option for maximality, and the* Single *semantic option for concurrency. Consider snapshot* $(\{\mathrm{S}_1, \mathrm{S}_2\}, \{\mathrm{a} = 7, \mathrm{b} = 2\}, *)$*, if we assume the* Big-Step *memory protocol,*

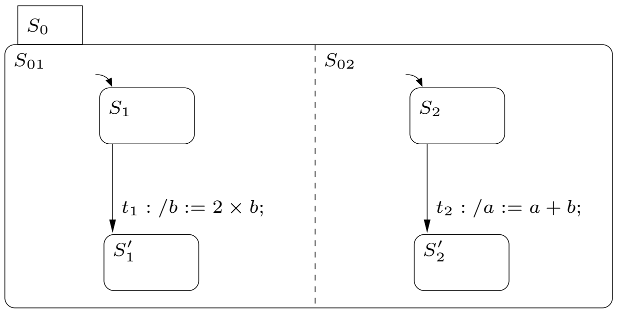


Figure 13: A model to maintain an invariant between a and b.

then the following big-step can be taken:

$$\begin{array}{ll}(\{S_1, S_2\}, \{a = 7, b = 2\}, *) & \xrightarrow{t_1} \\ (\{S'_1, S_2\}, \{a = 7, b = 4\}, *) & \xrightarrow{t_2} \\ (\{S'_1, S'_2\}, \{a = 9, b = 4\}, *). & \end{array}$$

The invariant of $\mathrm{a - b}$ *is maintained, because* $\mathrm{a - b}$ *is 5 before and after the big-step. Alternatively,* $\mathrm{t_2}$ *could have been taken before* $\mathrm{t_1}$*, which would again maintain the invariant.*

If we assume the Small-step *semantic option, then the following two big-steps are possible:*

$$\begin{array}{ll}(\{S_1, S_2\}, \{a = 7, b = 2\}, *) & \xrightarrow{t_1} \\ (\{S'_1, S_2\}, \{a = 7, b = 4\}, *) & \xrightarrow{t_2} \\ (\{S'_1, S'_2\}, \{a = 11, b = 4\}, *), & \end{array}$$

and

$$\begin{array}{ll}(\{S_1, S_2\}, \{a = 7, b = 2\}, *) & \xrightarrow{t_2} \\ (\{S'_1, S_2\}, \{a = 9, b = 4\}, *) & \xrightarrow{t_1} \\ (\{S'_1, S'_2\}, \{a = 9, b = 8\}, *). & \end{array}$$

None of them maintain the invariant.

Combo-Step: In this memory protocol, a read from a variable returns the value of the variable at the beginning of the current combo-step (e.g., Statemate [24]). When used for variable conditions, an **advantage** of this choice is that the variable condition of a transition does not change within a combo-step. A **disadvantage** of this memory protocol, which is inherent to the notion of combo-step, is that by mere review of a model it is difficult to determine the combo-steps of a model, and hence to determine the variables's values that are used by a small-step of a combo-step.

Example 11 *Similar to the previous example, the model in Figure 14 shows a model that is meant to maintain the invariant of* $\mathrm{a - b}$ *remaining the same before and after a big-step. Compared to the model in Figure 13, this model has two further transitions. Again, we assume the* Single *semantic option for concurrency, but this time, we consider the* Combo-Step *memory protocol, the* Take Many *maximality semantics for big-steps, and the* Take One *maximality semantics for combo-steps (i.e., each combo-step uses the same values for variables as in the beginning of the combo-step, it can include at most one transition belonging to* $\mathrm{S_{01}}$ *and* $\mathrm{S_{02}}$*, and a big-step can continue until there is no more transitions to be taken). Consider snapshot* $(\{\mathrm{S_1, S_2}\}, \{\mathrm{a = 7, b = 2}\}, *)$*, the following big-step can be taken (in the trace of a big-step, we use "|" as a delimiter to separate different combo-steps of the big-step):*

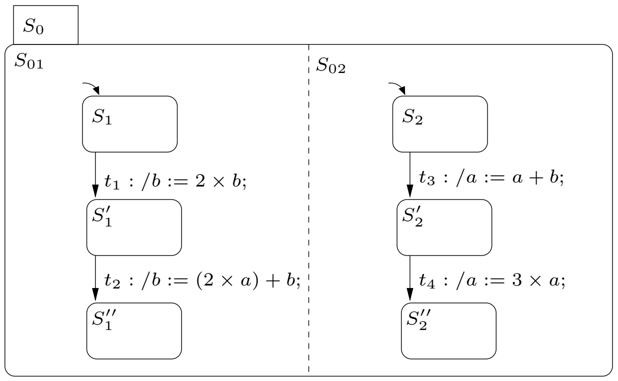

S_0
S_{01}
S_{02}
S_1
S_1'
S_1''
S_2
S_2'
S_2''
$t_1 : /b := 2 \times b;$
$t_2 : /b := (2 \times a) + b;$
$t_3 : /a := a + b;$
$t_4 : /a := 3 \times a;$

Figure 14: Another model to maintain an invariant between a and b.

$$\begin{array}{ll}
(\{S_1, S_2\}, \{a = 7, b = 2\}, *) & \xrightarrow{t_1} \\
(\{S_1', S_2\}, \{a = 7, b = 4\}, *) & \xrightarrow{t_3} \\
(\{S_1', S_2'\}, \{a = 9, b = 4\}, *)| & \xrightarrow{t_2} \\
(\{S_1'', S_2'\}, \{a = 9, b = 22\}, *) & \xrightarrow{t_4} \\
(\{S_1'', S_2''\}, \{a = 27, b = 22\}, *). &
\end{array}$$

The value of $\mathrm{a} - \mathrm{b}$ *remains the same after the execution of the big-step, as well as at the end of the first combo-step. Three other traces are possible by exchanging the order of the execution of* t_1 *and* t_3*, and the order of execution of* t_2 *and* t_4*, all of them maintaining the invariant.*

If we had chosen the maximality semantics of TAKE MANY *for combo-steps, then the invariant would be maintained by the big-steps of the model, but this time the execution always arrives at snapshot* $(\{\mathrm{S}_1'', \mathrm{S}_2''\}, \{\mathrm{a} = 21, \mathrm{b} = 16\}, *)$*; execution of* t_2 *and* t_4 *always overwrites the execution of* t_1 *and* t_3*, respectively, which means that the execution of* t_1 *and* t_2 *are irrelevant to the outcome of the big-step.*

A semantics can use different memory protocols for evaluating the values of the variables in the variable conditions and RHS of transitions. For example, in SCR [27], conditions are evaluated according to the BIG-STEP memory protocol, but the RHS of assignments are evaluated according to the SMALL-STEP memory protocol.

Memory protocols of BSMLs avoid many complications of dealing with global variables in programming languages because in BSMLs transition occurrences are *atomic*, and the reads of one transition cannot be influenced by the writes of another transition within the same small-step.[16]

3.4.2 Syntactic Keywords

A BSML may provide syntax for operators that obtain a value of the variable that is different from its value according to the memory protocol of the semantics of the BSML. Table 7 summarizes the operators that we consider in this section, along with their properties. (Operator `change` is different in that it is a boolean operator to represent the change of a variable in a big-step). A variable operator that always returns a value, regardless of the snapshot of a big-step in which its corresponding transition is executed and regardless of other transitions in its small-step, is called a *total* operator. As mentioned in Section 3.1, some of these operators can be used to define the "dataflow" order of a big-step/model.

[16] A similar notion to memory protocol is the notion of *memory consistency models* in programming languages [61]. A memory consistency model is different from a memory protocol in that it deals with individual read/write accesses to memory locations, as opposed to a memory protocol, which deals with the effect of atomic transitions, consisting of multiple reads and writes.

Operator	Reads From Snapshot	Dataflow	Total
`pre`	Big-step source	✗	✓
`cur`	Small-step source	✗	✓
`change`	N/A	✓	✓
`new`	Small-step source	✓	✗
"mandatory" `new_small`	Small-step destination	✗	✗
"non-mandatory" `new_small`	Small-step destination	✗	✓
"mandatory" `new_big`	Big-step destination	✓	✗
"non-mandatory" `new_big`	Big-step destination	✗	✓

Table 7: Properties of variable operators.

`pre`: This operator returns the value of a variable at the beginning of a big-step (e.g., RSML [38]). This operator is not relevant for the BIG-STEP memory protocol, because a read access to a variable, by definition, returns the value of the variable at the beginning of a big-step.

`cur`: This operator returns the value of a variable at the beginning of the current small-step, which can be an assigned value by a previous small-step of the big-step or the value from the previous big-step, depending on when the last assignment to the variable has happened (e.g., H&P&S&S Statecharts [26]). This operator is not relevant for the SMALL-STEP memory protocol, because a read access to a variable, by definition, returns the current value of the variable at the beginning of a small-step.

`change`: This operator is a boolean condition, whose status is true if the variable has been assigned a value in the current big-step so far, or false otherwise (e.g., Reactive Modules [3] and SCR [27]). Example 13 shows a model that uses the `change` operator.

`new`: This operator is similar to `cur`, but it differs from `cur` in that it returns a value for a variable only if the variable has been assigned a value during the big-step so far (e.g., Reactive Modules [3]). If variable x has not been assigned a value during a big-step, then the return value of statement $\texttt{new}(x)$ is not defined in that big-step. If statement $\texttt{new}(x)$ is used in the RHS of an assignment of a transition, then there is an *implicit condition* in the variable condition of the transition that is satisfied only if x is assigned a value during the current big-step. Operator `new` can be defined using operators `change` and `cur`:

$$\texttt{new}(x) \equiv \begin{cases} \texttt{cur}(x), \quad \mathit{if}\ \texttt{change}(x) = \mathit{true}, \\ \\ \mathit{Not\ Defined}, \quad \mathit{if}\ \texttt{change}(x) = \mathit{false} \end{cases}$$

Where $\texttt{change}(x)$ is the "implicit condition" of any transition that uses $\texttt{new}(x)$.

To avoid having undefined behaviour in a model, the semantics does not allow a transition with a $\texttt{new}(x)$ statement to execute until there is at least an assignment to x (i.e., until the implicit $\texttt{change}(x)$ condition in its variable condition is satisfied). Such a transition is said to be *blocked.* The use of the `new` operator creates an *implied order* among the transitions of a big-step, which satisfies the implicit conditions of the transitions that are executed during the big-step. In general, the implied order between the transitions of a big-step is not a fixed order for the entire model; it is dependent on the big-step. Example 12 shows how different big-steps of a model create different implied orders. (The notion of implied order is related to the notion of "dataflow" order, described in Section 3.1.2, but as opposed to a dataflow orders, it is defined among transitions instead of variables.)

A **disadvantage** of `new` is that two transitions might be blocking because their assignments *cyclically depend* on each other via their `new` statements. In a BSML that has a SYNTACTIC maximally semantics,

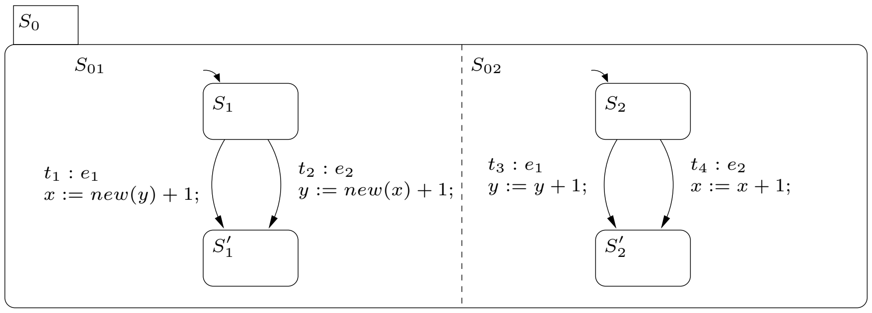

S_0
S_{01}
S_{02}
S_1
S_1'
S_2
S_2'
$t_1 : e_1$ $x := new(y) + 1;$
$t_2 : e_2$ $y := new(x) + 1;$
$t_3 : e_1$ $y := y + 1;$
$t_4 : e_2$ $x := x + 1;$

Figure 15: Using operator `new` in a model.

cyclically dependent transitions can cause non-reactiveness when the model cannot reach a snapshot where it becomes stable, and can receive new environmental inputs. A possible solution to the problem of a set of cyclically dependent transitions is to execute them *together*, which means that their implicit conditions are satisfied during the small-step that they are executed together. But this solution poses new problems: (i) it defeats the purpose of using the `new` operator to order the execution of transitions in a big-step, and (ii) it evaluates the `new` statements non-deterministically, which itself can be considered as a **disadvantage**. As an example, consider a model that resides in its snapshot $(*, \{x = 1, y = 1\}, *)$. Also, consider two transitions t_1 and t_2 and their corresponding sets of assignments: $\{x := \texttt{new(}y\texttt{)}+1;\}$ and $\{y := \texttt{new(}x\texttt{)}+1;\}$. Transitions t_1 and t_2 are are cyclically dependent. If we consider a semantics that resolves the cyclic dependence of the transitions by executing them "together," then the new values for x and y can be inferred non-deterministically as: $(*, \{x = 2, y = 1\}, *)$, $(*, \{x = 3, y = 2\}, *)$, etc.

Example 12 *The model in Figure 15 carries out a trivial arithmetic operation. If we assume that the model is specified in a BSML that supports the* Many *concurrency semantics, the* Big-step *memory protocol, and an event model where an input event persists during the big-step, then one would expect that two transitions, one from* S_{01} *and one from* S_{02}*, to be able to execute together in the same small-step. However, because of the implied order of a big-step, two transitions can never execute in a same small-step. Consider snapshot* $(\{\mathrm{S}_1, \mathrm{S}_2\}, \{\mathrm{x} = 1, \mathrm{y} = 1\}, \{\mathrm{e}_1\})$*, the following big-step is the only possible big-step:*

$$\begin{array}{ll}(\{S_1, S_2\}, \{x = 1, y = 1\}, \{e_1\}) & \xrightarrow{t_3} \\ (\{S_1, S_2'\}, \{x = 1, y = 2\}, \{e_1\}) & \xrightarrow{t_1} \\ (\{S_1', S_2'\}, \{x = 3, y = 2\}, \{e_1\}). & \end{array}$$

The implied order of the above big-step is $\mathrm{t}_3 < \mathrm{t}_1$ *and* $\mathrm{y} < \mathrm{x}$*, where "<" means earlier in the order.*

If we consider snapshot $(\{\mathrm{S}_1, \mathrm{S}_2\}, \{\mathrm{x} = 1, \mathrm{y} = 1\}, \{\mathrm{e}_2\})$*, then the following big-step is the only possible big-step:*

$$\begin{array}{ll}(\{S_1, S_2\}, \{x = 1, y = 1\}, \{e_2\}) & \xrightarrow{t_4} \\ (\{S_1, S_2'\}, \{x = 2, y = 1\}, \{e_2\}) & \xrightarrow{t_2} \\ (\{S_1', S_2'\}, \{x = 2, y = 3\}, \{e_2\}), & \end{array}$$

with the implied order $\mathrm{x} < \mathrm{y}$ *and* $\mathrm{t}_4 < \mathrm{t}_2$*, which is different from the implied order of the previous big-step.*

In the example above, there does not exist a fixed implied order between x and y for the entire model. Therefore, there does not exist a fixed implied order between transitions.

Having a fixed order between the variables/transitions of a model is conceptually elegant. It avoids the problem of cyclically dependent transitions because the fixed order can be easily checked for being non-cyclic. In Reactive Modules [3], for example, which uses the `new` operator, the implied order of a big-step coincides with its notion of static "dataflow" order (dataflow order is described in Section 3.1.2). A transition in Reactive Modules [3] is equal to an *atom*, which is a syntax for grouping a set of assignments together. Each variable belongs to exactly one atom, and the atoms of a model are (partially) ordered based on the reads of an atom from the new values of the variables of other atoms. The order of the execution of the atoms are syntactically specified, and needs to be acyclic. Therefore, in Reactive Module [3]: (i) there is a static order between the variables/small-steps of the model, and (ii) two transitions never cyclically depend on each other.

`new_small`**:** This operator returns the value of a variable at the end of the current small-step (e.g., P&S Statecharts [54]). `new_small` is different from `new` in that for a variable x, `new_small(`x`)` returns the value of x at the end of the current small-step, only if it has been assigned a value in the current small-step; as opposed to `new(`x`)`, which returns the most recent assigned value of x, if it has been assigned a value so far during the current big-step. A possible semantic variation for `new_small` is a semantics that does not require a variable x to be assigned a value during the current small-step, in order for `new_small(`x`)` to return a value; if x is not assigned a value in the current small-step, then `new_small(`x`)` returns the value of x at the beginning of the current small-step. We call our initial semantics of `new_small` the *mandatory* semantics, and its semantic variation the *non-mandatory* semantics. In this report, unless otherwise stated, we follow the mandatory semantics. A simple way to guarantee that for a transition t, `new_small(`x`)` always has a value, is to require that there is an assignment for x in the set of assignments of t; this approach is used in P&S Statecharts [54]. Example 14 shows a model that uses the `new_small` operator.

An **advantage** of `new_small` is that one can ignore the order of the execution of the transitions of a small-step. Similar to `new`, as a **disadvantage**, there is a possibility for two transitions of a small-step to block, and cyclically depend on each other's assignments. In [54], two cyclically dependent assignments in a small-step are evaluated "together" (as described for the `new` operator), which means a `new_small` statement can return non-deterministic values.

`new_big`**:** This operator returns the value of a variable at the end of a big-step. For a variable x, if there are multiple assignments to x during a big-step, then `new_big(`x`)` returns the value of the last assignment to x, as opposed to `new(`x`)`, which returns different values, depending on where in the sequence of the small-steps of the big-step `new(`x`)` has been used. Operator `new_big` has the same **advantage** and **disadvantage** as `new_small`, but its semantics can be significantly more complicated than `new_small`. For example, if a BSML uses the TAKE MANY semantic option for its maximality and the SMALL-STEP semantic option as its memory protocol, then in order to evaluate statement `new_big(`x`)`, an unknown number of future transitions might need to be considered (possibly involving reasoning about cyclic dependencies between different `new_big` statements). Similar to `new` and `new_small` operators, it is possible to resolve the circular dependencies of a set of transitions, due to `new_big` statements, by taking them "together." Also, similar to the `new_small` operator, it is possible to consider a "non-mandatory" semantics for `new_big` operator; in which case, if variable x is not assigned a value during a big-step, the `new_big(`x`)` statement in the big-step returns the value of x according to reading x at the beginning of the current big-step.

Using some of the above operators with some memory protocols is not natural. For example, using `new` in a BSML that uses the BIG-STEP memory protocol violates the paradigm of the BIG-STEP memory protocol of only looking at the values of variables at the beginning of a big-step. Similarly, using `pre` with the COMBO-STEP memory protocol makes the semantics of a BSML complicated; a modeller needs to keep track of three different sets of values of the variables of a model: (i) the values of the variables at the beginning of the big-step (to evaluate `pre` statements), (ii) the values of the variables at the beginning of the current combo-step (to carry out the actions of the transitions in a combo-step), and (iii) the current values of the variables (to compute the values of the variables for the next combo-step).

Example 13 *The model in Figure 16 is similar to the model in Figure 14, except that transitions* t_2 *and* t_4 *use conditions "change(b)" and "change(a)" in their variable conditions, respectively. We assume the* TAKE MANY *semantic option for maximality, the* SINGLE *semantic option for concurrency, and the* SMALL-STEP *memory protocol. Again, we examine whether the value of* $\mathrm{a} - \mathrm{b}$ *remains the same after the execution of a big-step. Assuming snapshot* $(\{\mathrm{S}_1, \mathrm{S}_2\}, \{\mathrm{a} = 7, \mathrm{b} = 2\}, *)$*, then the following big-step can be taken:*

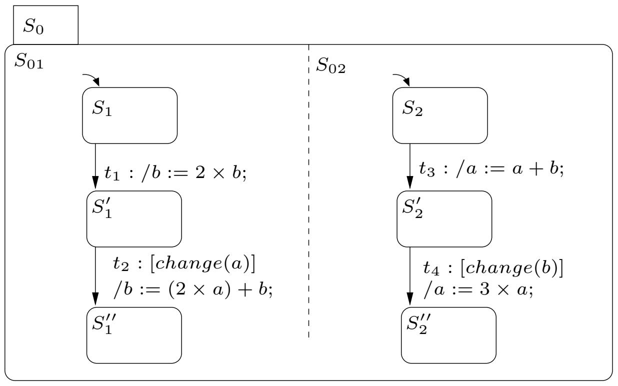


Figure 16: Using operator `change` to orchestrate the execution of a model.

$$\begin{array}{ll}
(\{S_1, S_2\}, \{a = 7, b = 2\}, *) & \xrightarrow{t_1} \\
(\{S_1', S_2\}, \{a = 7, b = 4\}, *) & \xrightarrow{t_3} \\
(\{S_1', S_2'\}, \{a = 9, b = 4\}, *) & \xrightarrow{t_2} \\
(\{S_1'', S_2'\}, \{a = 9, b = 22\}, *) & \xrightarrow{t_4} \\
(\{S_1'', S_2''\}, \{a = 27, b = 22\}, *). &
\end{array}$$

Similar to Example 11, three other big-steps are possible, all of them yielding the same outcome. (Here, the "change" conditions play the role that a combo-step played in Example 11.)

If we had chosen not to include the "change" conditions for t_2 *and* t_4*, then the invariant, for some big-steps, would not have held. For example, the following big-step would have been possible:*

$$\begin{array}{ll}
(\{S_1, S_2\}, \{a = 7, b = 2\}, *) & \xrightarrow{t_1} \\
(\{S_1', S_2\}, \{a = 7, b = 4\}, *) & \xrightarrow{t_2} \\
(\{S_1'', S_2\}, \{a = 7, b = 18\}, *) & \xrightarrow{t_3} \\
(\{S_1'', S_2'\}, \{a = 25, b = 18\}, *) & \xrightarrow{t_4} \\
(\{S_1'', S_2''\}, \{a = 75, b = 18\}, *), &
\end{array}$$

which does not satisfy the invariant, because $\mathrm{a} - \mathrm{b}$ *is 57.*

Example 14 *The model in Figure 17 is similar to the model in the Example 14, but does the extra functionality of reporting the sum and the difference of* a *and* b*, via variables* sum *and* diff*, respectively. In this example, we assume the* TAKE MANY *option for maximality, the* MANY *option for concurrency, and the* SMALL-STEP *memory protocol. If we start from snapshot* $(\{\mathrm{S}_1, \mathrm{S}_2\}, \{\mathrm{a} = 7, \mathrm{b} = 2, \mathrm{sum} = *, \mathrm{diff} = *\}, *)$*, the following big-step is the only possible big-step of the model:*

$$\begin{array}{ll}
(\{S_1, S_2\}, \{a = 7, b = 2\, sum = *, diff = *\}, *) & \xrightarrow{t_1, t_3} \\
(\{S_1', S_2'\}, \{a = 9, b = 4\}, sum = *, diff = *\}, *) & \xrightarrow{t_2, t_4} \\
(\{S_1'', S_2''\}, \{a = 27, b = 22\}, sum = 49, diff = 5\}, *). &
\end{array}$$

If we do not to use the `new_small` *operator, then* $\mathrm{sum} = 23$ *and* $\mathrm{diff} = 5$*, where the value of* sum *is wrong, and is computed after the first small-step, instead of at the end of the big-step.*

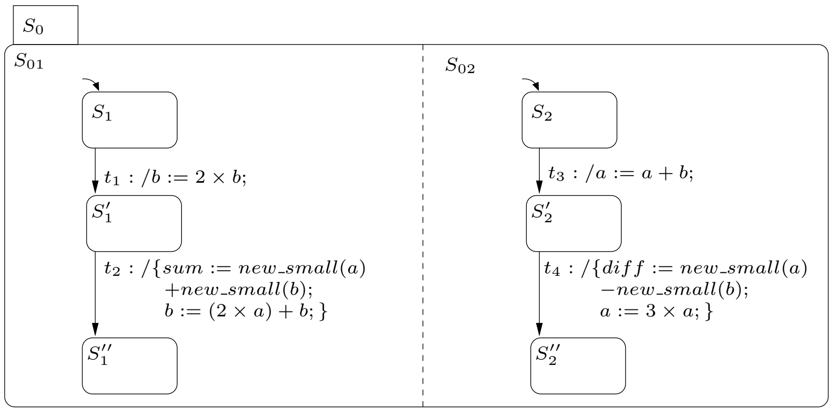

S_0
S_{01}
S_1
$t_1 : /b := 2 \times b;$
S'_1
$t_2 : /\{sum := new\_small(a) + new\_small(b); b := (2 \times a) + b; \}$
S''_1
S_{02}
S_2
$t_3 : /a := a + b;$
S'_2
$t_4 : /\{diff := new\_small(a) - new\_small(b); a := 3 \times a; \}$
S''_2

Figure 17: Using operator `new_small` to obtain the desired values of variables a and b.

Option Description	Advantages	Disadvantages	Example
LINEARIZATION	Atomicity of transitions, and supports the intuition of sequentiality	Assignments are not equalities, non-associative race resolution operator	15
RANDOM	Associative race resolution	Assignments are not equalities, non-intuitive from sequential point of view, and weakened notion of atomicity	15

Table 8: Race Resolution Semantic Options.

3.4.3 Race Resolution

In BSMLs, a race condition is a situation where more than one transition in a small-step assigns values to the same variable (only possible when MANY or SYNCHRONIZATION concurrency semantics is chosen).[17] We call such a small-step a *racy small-step*, and the variables that are written to by more than one transition of the small-step, its *racy variables*. Unless there are syntactic constraints that ensure that race conditions do not exist (e.g., each HTS only writes to one variable), a BSML should have a policy for how to resolve race conditions. The main **disadvantage** of a BSML that allows race conditions is that we cannot reason about the set of assignments of a transition independently of other transitions in a small-step. When a transition is considered in isolation, then the assignment signs of its set of assignments can be replaced with equality signs, which provides a convenient paradigm for reasoning about models. Similar to race conditions in programming languages, in BSMLs, race conditions lead to non-deterministic behaviour. In BSMLs, because transitions execute atomically, a transition cannot see the intermediate assignments of other transitions, which makes the race conditions in BSMLs less complicated than in programming languages.

Next, we consider two race resolution mechanisms for BSMLs, as shown in Table 8. The RANDOM option has been studied in our previous work [48], but the LINEARIZATION option is being introduced in this report.

LINEARIZATION: In this semantic option, the values of the variables at the end of a racy small-step are equal to the values of the variables according to an arbitrary sequential execution of the transitions of the small-step when a transition does not read the effects of the assignments of its preceding transitions in the sequence. We call such a sequence of transition execution of a racy small-step its *linearization*. In

[17]In some BSMLs, such as [24], a transition could have a *race condition* [24] within itself, by having multiple assignments to same variable when the assignments of a transition are considered as a set; but those "races" can be syntactically detected and resolved, and therefore we do not consider them in this section.

other words, an outcome of a racy small-step according to the LINEARIZATION option is equivalent to the execution of its transitions in an arbitrary order. This semantic option is similar to the notion of serializability in databases [51] and linearizablity in concurrent programs [29, 28], but it is different from them in that the updated values of variables by the earlier transitions in a linearization are not read by the later transitions. Obviously, as the number of transitions and racy variables of a racy small-step increases, the number of possible *outcomes* can increase in a factorial order, where an outcome of a racy small-step is a snapshot of the model after the execution of the racy small-step.

For a racy small-step $T = \{t_1, t_2, \cdots, t_n\}$, there are $n!$ possible linearizations, some of the outcomes being the same. For example, for small-step $T = \{t_1, t_2, t_3\}$ and the corresponding assignments of its transitions: $a_1 = \{x = 1\}$, $a_2 = \{y = 2, z = 2\}$, and $a_3 = \{x = 3, y = 3, z = 3\}$, there are 4 ($4 \neq 3!$) distinct outcomes: $\{x = 1, y = 2, z = 2\}$, $\{x = 1, y = 3, z = 3\}$, $\{x = 3, y = 2, z = 2\}$, and $\{x = 3, y = 3, z = 3\}$. There are two reasons why there could be fewer distinct outcomes than the factorial of the number of the transitions of a small-step:

1. If a set of variables are assigned values by a transition in a linearization, then the assignments of the earlier transitions in the linearization that only write to the same variables, or only write to a subset of the variables, can be ignored. For example, in the example above, if t_3 appears as the last transition of a linearization, then earlier transitions do not matter, the outcome is always $\{x = 3, y = 3, z = 3\}$.

2. In a linearization, if two consecutive transitions assign values to disjoint sets of variables, then their order in the linearization can be swapped, without affecting the outcome. For example, in the example above, both linearizations $\langle t_3, t_1, t_2\rangle$ and $\langle t_3, t_2, t_1\rangle$ yield the same outcome, namely $\{x = 1, y = 2, z = 2\}$.

An **advantage** of this semantic option is that it partially follows the intuition of sequential programming. Depending on the number of transitions and racy variables in a small-step, it could be easy for a modeller to analyze the result of the race resolution of a racy small-step. A **disadvantage** of this semantic option is that it does not work incrementally, in an associative way; instead, all transitions need to be considered together, which can be confusing for a modeller. More formally, if $\oplus$ is a binary operator on transitions that determines the possible outcomes of a racy small-step according to the LINEARIZATION semantics, then for three transitions t_1,t_2, and t_3 in a racy small-step, it might be the case that the set of possible outcomes of $(t_1 \oplus t_2) \oplus t_3$ is not equal to the set of possible outcomes $t_1 \oplus (t_2 \oplus t_3)$.

RANDOM: This semantic option differs from the LINEARIZATION option in that it sequentializes all of the assignments belonging to all of the transitions of a racy small-step, instead of sequentializing their transitions. An outcome of a racy small-step when using the LINEARIZATION option can always be reproduced by the RANDOM option. The RANDOM option is comparable to *asymmetric lock atomicity* [43] in transactional memories.

The number of distinct outcomes of a racy small-step is the product of the number of the assignments to each variable in the actions of the transitions of the small-step. Consider the example in the LINEARIZATION section, the number of assignments to x, y, and z are all 2, and thus the number of possible outcomes using the RANDOM option is $2 \times 2 \times 2 = 8$. One of the outcomes that is possible here, but not possible when using the LINEARIZATION option, is $\{x = 1, y = 2, z = 3\}$; there are three more such outcomes.

An **advantage** of this semantic option is that a modeller simply needs to consider all combinations of the assignments of all transitions in an incremental, associative way. If $\oplus$ is a binary operator on transitions that determines the possible outcomes of a racy small-step according to the RANDOM semantics, then for three transitions t_1,t_2, and t_3 in a racy small-step, the set of possible outcomes of $(t_1 \oplus t_2) \oplus t_3$ is equal to the set of possible outcomes $t_1 \oplus (t_2 \oplus t_3)$. From a modeller's point of view, a **disadvantage** of this semantic option is that it is a non-intuitive and a non-predictable semantics, because in a specification a modeller rarely is interested in specifying a behaviour in which variable assignments of one transition are mixed with the variable assignments of others.

Example 15 *The model in Figure 18 is similar to the model in Figure 14, but this time is enhanced to report which HTS takes the last transition of a big-step, by setting the appropriate values to variables* me *and* other. *In this example, we assume the* TAKE MANY *semantic option for maximality, the*

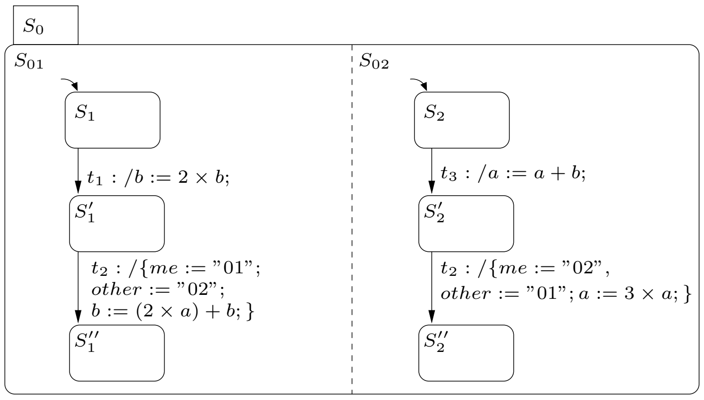


Figure 18: Race condition for variables *me* and *other*.

MANY *semantic option for concurrency, and the* SMALL-STEP *memory protocol. If we start from snapshot* $(\{\mathrm{S_1}, \mathrm{S_2}\}, \{\mathrm{a} = 7, \mathrm{b} = 2, \mathrm{me} = *, \mathrm{other} = *\}, *)$*, and assume the* LINEARIZATION *semantic option for race resolution, then the following big-step is possible:*

$$\begin{array}{ll} (\{S_1, S_2\}, \{a = 7, b = 2\, me = *, other = *\}, *) & \xrightarrow{t_1, t_3} \\ (\{S_1', S_2'\}, \{a = 9, b = 4\}, me = *, other = *\}, *) & \xrightarrow{t_2, t_4} \\ (\{S_1'', S_2''\}, \{a = 27, b = 22\}, me = \text{"01"}, other = \text{"02"}\}, *). & \end{array}$$

Another possible outcome is created by the same big-step as above, but the final snapshot being:

$$(\{S_1'', S_2''\}, \{a = 27, b = 22\}, me = \text{"02"}, other = \text{"01"}\}, *).$$

If we choose the RANDOM *option, then the same two outcomes as above are possible, but additionally:*

$$(\{S_1'', S_2''\}, \{a = 27, b = 22\}, me = \text{"01"}, other = \text{"01"}\}, *),$$

and

$$(\{S_1'', S_2''\}, \{a = 27, b = 22\}, me = \text{"02"}, other = \text{"02"}\}*)$$

are possible, both being nonsensical.

3.5 External Variable Communication

In order to avoid modelling flaws, many have advocated that the interface of a model with its environment should be clearly and explicitly specified [52, 65, 34]. A straightforward and celebrated way to achieve this interface is to distinguish between the variables that the environment can assign values to (*environmental input variables*), and the variables that the model assigns values to (*controlled variables*). Controlled variables can be partitioned further into the variables that can be observed and read by the environment (*environmental output variables*), and the variables that cannot be observed by the environment (*private variables*). We call the union of the set of environmental input variables and the set of environmental output variables of a model its set of *external variables*. Figure 19 summarizes the taxonomy of variables for BSMLs that support external variables; a solid box represents a set of variable (e.g., the set of private variables), and a dashed box represents a set including other sets of variables (e.g., the set of controlled variables is the union of the set of environmental output variables and the set of private variables). Many modelling languages, including many BSMLs, provide syntax to distinguish between different types of variables [52, 27, 3]. In Section 3.4, we considered the semantics of non-external variables, in this section, we present the semantics of other types of variables.

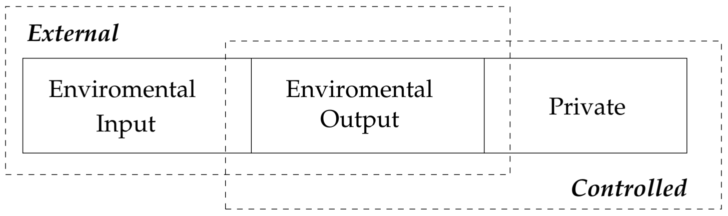


Figure 19: A taxonomy for the types of variables in a BSML that distinguishes between a model and its environment.

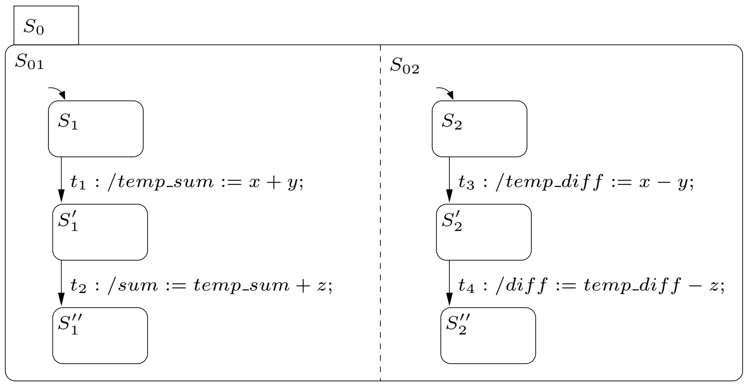

S_0
S_{01}
S_{02}
S_1
$t_1 : /temp\_sum := x + y;$
S_1'
$t_2 : /sum := temp\_sum + z;$
S_1''
S_2
$t_3 : /temp\_diff := x - y;$
S_2'
$t_4 : /diff := temp\_diff - z;$
S_2''

Figure 20: Environment vs. model, and different types of variables.

Example 16 *Figure 20 shows a simple model that does summation and subtraction on its "environmental input" variables* x, y, *and* z. *The result of the summation and the subtraction are sent back to the environment through* environmental output *variables* sum *and* diff. *We assume that the summation and the subtraction are binary operators, and therefore we use two* private *variables* temp_sum *and* temp_diff, *to carry out the computation. The* external *variables of the model are* x, y, z, sum, *and* diff; *and the* controlled *variables of the model are* sum, diff, temp_sum, *and* temp_diff.

The memory protocol of environmental input variables is usually the BIG-STEP memory protocol. The memory protocol of environmental output variables can be any of the memory protocols.

3.5.1 Inter-Component Communication

Some BSMLs structure a model as a composition of a set of *components*, each of which can play the role of the environment, or part of the environment, for others. Components of a model are meant to represent physically distinct parts of a model that communicate with each other through an *inter-component communication* mechanism. The semantics of inter-component communication for variables, similar to memory protocols for private variables, should specify when a write by a component can be read in another component. Figure 21 illustrates the taxonomy of variables for the BSMLs that support an inter-component communication mechanism. We call the set of variables of a model that are involved in the inter-component communication its set of *interface variables.* We require a well-formedness constraint on how interface variables are used in a model: an interface variable can be assigned a value by exactly one component (the *sending component*), and some *receiving components* that read the value of the interface variable. In the presence of inter-component communication, the set of "external" and "private" variables of a model are defined the same as they were defined earlier in this section, but its set of "controlled" variables, additionally, includes the set of "interface" variables. An implication of the well-formedness constraint for interface variables is that the set of interface variables of a model is partitioned into sets of variables, each of which

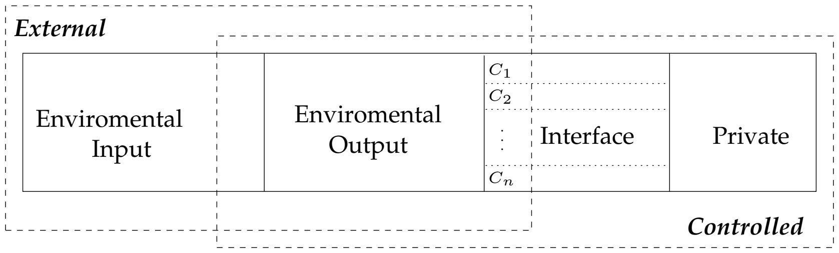


Figure 21: The taxonomy of types of variables for inter-component communication.

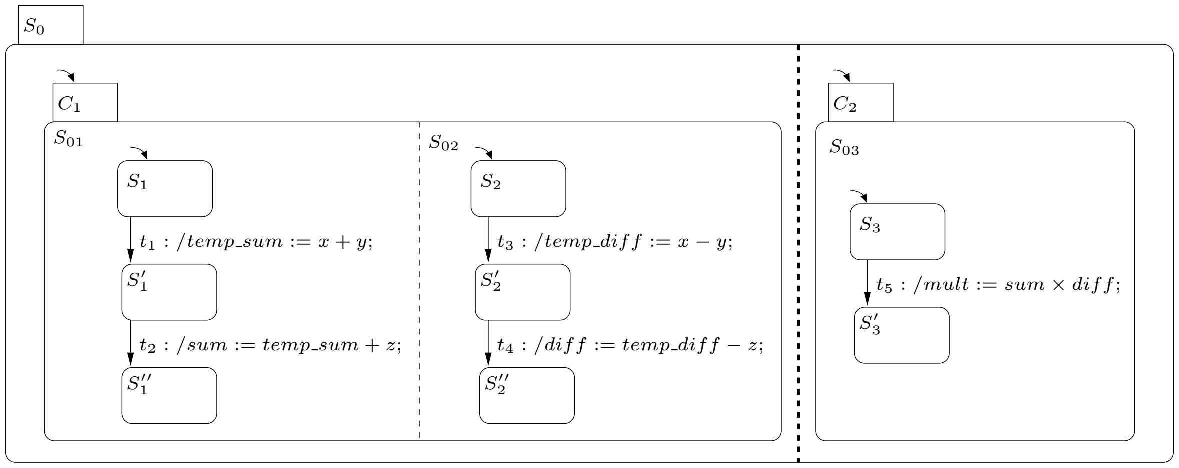

S_0
C_1
S_{01}
S_1
$t_1 : /temp\_sum := x + y;$
S_1'
$t_2 : /sum := temp\_sum + z;$
S_1''
S_{02}
S_2
$t_3 : /temp\_diff := x - y;$
S_2'
$t_4 : /diff := temp\_diff - z;$
S_2''
C_2
S_{03}
S_3
$t_5 : /mult := sum \times diff;$
S_3'

Figure 22: Inter-component communication via interface variables.

includes only the variables that are being written to the same component of the model (this partitioning is illustrated by dotted lines in Figure 21).

Example 17 *The model in Figure 22 is similar to the model in Figure 20, but is different from it in that it does the extra functionality of computing the multiplication of the sum and the diff of the three input numbers. Here we assume that the model is composed of two* components C_1 *and* C_2*, which are pictorially separated by a thick dashed line. Variables* sum *and* diff *are "interface" variables;* C_1 *is the "sending" component for both interface variables, and* C_2 *is the receiving component of both interface variables . The "environmental input variables" of the model are* x, y, *and* z*; the "environmental output variables" of the model is* mult*; the "private" variables of the model are* temp_sum *and* temp_diff*; the "external" variables of the model are* x, y, z, *and* mult*; and the "controlled" variables of the model are* sum, diff, temp_sum, temp_diff, *and* mult*.*

When an interface variable is assigned a value by a transition of a sending component, the semantics of a BSML should specify when the new value of the variable becomes available for the receiving component(s) (i.e., it should specify how long it will take for the inter-component communication mechanism to carry the new value of the variable from one component to another). Interface variables may use a different memory protocol than the one used for private variables. We consider two types of inter-component communication: one in the spirit of the synchrony hypothesis (the SYNCHRONOUS option) and one in the spirit of delayed communication (the ASYNCHRONOUS option). Table 9 summarizes the semantic options that we consider in this section, along with their advantages and disadvantages.

SYNCHRONOUS: In this inter-component communication mechanism, once a sending component assigns a value to an interface variable during a big-step, the assigned value becomes available for the receiving components within the big-step. Two semantic sub-options for this semantics are:

Option Description	Advantages	Disadvantages	Example
SYNCHRONOUS			
STRONG SYNCHRONOUS	Compatible with the synchrony hypothesis and modular with respect to interface variables	Blocking read	18
WEAK SYNCHRONOUS	Compatible with the synchrony hypothesis, and non-blocking read	Not differentiating between a stale and a new value of a variable	18
ASYNCHRONOUS	Non-blocking read and modular with respect to interface variables	Not compatible with the synchrony hypothesis, and not differentiating between a stale and a new value of a variable	18

Table 9: Inter-component Communication: summary of semantic variations, and their advantages and disadvantages.

– STRONG SYNCHRONOUS: In this option, if the sending component writes to an interface variable during a big-step, a *stale* value of the variable (from the previous big-step) cannot be read by a receiving component. However, if the sending component does not write to an interface variable during a big-step, then a stale value can be read by a receiving component. Speaking in terms of memory protocols, the semantics of STRONG SYNCHRONOUS is the BIG-STEP memory protocol, along with applying the "non-mandatory" semantic variation of `new_big` operator to all reads from the interface variables. (According to the non-mandatory semantics of `new_big`, in order to evaluate `new_big(`x`)`, x does not need to be assigned a value during the big-step.)

– WEAK SYNCHRONOUS: This option relaxes the STRONG SYNCHRONOUS option by allowing a read access to an interface variable to return a stale value, even if the variable will be assigned a value during the big-step. A read access to an interface variable is either a stale value, if the variable has not been written to by a preceding small-step in the current big-step, or otherwise it is the newly assigned value. This semantic option is the SMALL-STEP memory protocol for the interface variables.

Both STRONG SYNCHRONOUS and WEAK SYNCHRONOUS options have the **advantage** of following the zero-time computation principle of the synchrony hypothesis: The value of an interface variable is exchanged between two components in "zero-time." The STRONG SYNCHRONOUS option has the extra **advantage** of treating the environmental input variables and the interface variables similarly, in a *modular* way [32]. A semantics is modular, if it treats a component of the model exactly the same as the external environment of the model, which is outside the context of the model. In the STRONG SYNCHRONOUS option, interface variables are treated the same as the environmental input variables, because a read from an interface variable in a big-step must happen either after a write to it or else there should not be any write to it during the big-step. This behaviour is the same as assuming that either a write to an interface variable happens at the beginning of the big-step or there is no write to it during the big-step, which is the same as the semantics of an environmental input variable. Modularity is valuable because when a new component is added to an existing model, it can do inter-component communication with the existing components without requiring them to change their behaviours; the existing components consider the new component just as a part of the environment. A **disadvantage** of the STRONG SYNCHRONOUS option is that reading from an interface variable in a transition can be blocking. As usual, in the presence of blocking, two transitions of two components may cyclically depend on each other's assignments. A **disadvantage** of the WEAK

SYNCHRONOUS option is that during a big-step it is not clear whether a read value of an interface variable is stale or new.

ASYNCHRONOUS: This semantic option means that the effect of an assignment to an interface variable is available in the next big-step. Speaking in terms of memory protocols, this semantic option is the BIG-STEP memory protocol for its interface variables. An **advantage** of this semantic option is that a transition never blocks on reading the value of an interface variable. Also, as an **advantage**, this option treats interface events in a modular way, by making them available in the beginning of a big-step. A **disadvantage** of this semantic option is that it is not compatible with the synchrony hypothesis, because communication between components takes one big-step to complete. Furthermore, from an understandability point of a view, a modeller needs to keep track of the assignments to the interface variables from the previous big-steps.

Example 18 *The model in Figure 23 shows a door controller system, which is responsible for unlocking the door to an industrial area, only if the temperature inside the area is not above 40°C. The system has two physical "components,"* $\mathrm{C_1}$ *and* $\mathrm{C_2}$ *specified by the separating thick dashed lines. Variable "danger" is an interface variable. We assume the* TAKE MANY *semantic option for maximality, the* SINGLE *semantic option for concurrency, and an event semantics that assumes that input events and generated events, when generated, persist during a big-step (event semantics are described in sections 3.6 and 3.7). Assuming snapshot* $(\{\mathrm{S_1}, \mathrm{S_2}\}, \{\mathrm{danger} = \mathrm{false}, \mathrm{door} = \mathrm{closed}, \mathrm{temp} = 99\}, \{\mathrm{open}\})$*, if we assume the inter-component communication model of* STRONG SYNCHRONOUS*, then only the following big-step can be taken:*

$$\begin{array}{ll}
(\{S_1, S_2\}, \{danger = false, door = closed, temp = 99\}, \{open\}) & \xrightarrow{t_1} \\
(\{S_1', S_2\}, \{danger = false, door = closed, temp = 99\}, \{open, temp\_ok\}) & \xrightarrow{t_6} \\
(\{S_1', S_2'\}, \{danger = true, door = closed, temp = 99\}, \{open, temp\_ok\}) & \xrightarrow{t_3} \\
(\{S_1'', S_2'\}, \{danger = true, door = closed, temp = 99\}, \{open, temp\_ok\}). &
\end{array}$$

If we assume the inter-component communication model of WEAK SYNCHRONOUS*, then additionally the following big-step can be taken:*

$$\begin{array}{ll}
(\{S_1, S_2\}, \{danger = false, door = closed, temp = 99\}, \{open\}) & \xrightarrow{t_1} \\
(\{S_1', S_2\}, \{danger = false, door = closed, temp = 99\}, \{open, temp\_ok\}) & \xrightarrow{t_2} \\
(\{S_1, S_2\}, \{danger = false, door = closed, temp = 99\}, \{open, temp\_ok, unlock\}) & \xrightarrow{t_6} \\
(\{S_1, S_2'\}, \{danger = true, door = closed, temp = 99\}, \{open, temp\_ok, unlock\}), &
\end{array}$$

which unlocks the door, despite the high temperature.

If we assume the inter-component communication model of ASYNCHRONOUS*, then the only possible big-step is:*

$$\begin{array}{ll}
(\{S_1, S_2\}, \{danger = false, door = closed, temp = 99\}, \{open\}) & \xrightarrow{t_1} \\
(\{S_1', S_2\}, \{danger = false, door = closed, temp = 99\}, \{open, temp\_ok\}) & \xrightarrow{t_2} \\
(\{S_1, S_2\}, \{danger = false, door = closed, temp = 99\}, \{open, temp\_ok, unlock\}) & \xrightarrow{t_6} \\
(\{S_1, S_2'\}, \{danger = true, door = closed, temp = 99\}, \{open, temp\_ok, unlock\}), &
\end{array}$$

which again unlocks the door, despite the high temperature. Also, although $\mathrm{t_6}$ *assigns the value* true *to* danger*, the value of* danger *cannot be read by another transition in the current big-step.*

Similar to private variables, race conditions arise for interface variables, which would happen when the concurrent parts of the sending component assign multiple values to the same interface variable during a small-step. Exactly the same race resolution mechanisms as the ones for private variables can be used for interface variables. Additionally, a new notion of race conditions arises when an interface variable is assigned a value in more than one small-step of a big-step, in which case the semantics of the STRONG SYNCHRONOUS and the the WEAK SYNCHRONOUS options need to be clarified to accommodate such scenarios.

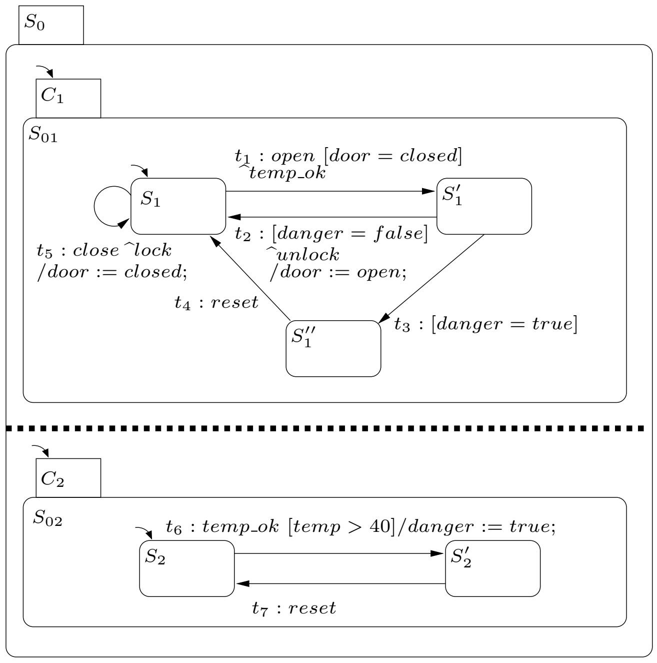

S_0
C_1
S_{01}
S_1
S_1'
S_1''
$t_1 : open\ [door = closed]\ \hat{}temp\_ok$
$t_2 : [danger = false]\ \hat{}unlock\ /door := open;$
$t_3 : [danger = true]$
$t_4 : reset$
$t_5 : close\ \hat{}lock\ /door := closed;$
C_2
S_{02}
S_2
S_2'
$t_6 : temp\_ok\ [temp > 40]/danger := true;$
$t_7 : reset$

Figure 23: Inter-component communication on variable *danger*.

3.6 Internal Events

In BSMLs, an *internal event* of a model is a named artifact that is used to orchestrate (e.g., sequence) the execution of the transitions of the model. An internal event has a status, which is either *present* or *absent*, and can be sensed by the event triggers of the transitions of the model. As opposed to a variable, an internal event is a transient medium for communication, which means that when generated, its present status *persists* only during some intermediate snapshots of the big-step in which it is generated. We refer to the big-step in which an internal event is generated as *its big-step*. Internal events are usually communicated via a *broadcast communication* mechanism that delivers the status of an internal event to all parts of a model at the same time, providing a uniform view of the status of the internal events for all parts of the model. In this section, we first describe the semantic options for an internal event's persistence, which we call its *lifeline* semantics. We then consider options for the *negation* of an event, which can be used in the event trigger of a transition to check for the absence of an internal event.

Table 10 illustrates the semantic options that we consider in this section, along with their advantages and disadvantages. In Section 3.7, we consider *external events*, which are events that communicate with the environment. Throughout the report, whenever clear from the context, we use the terms "internal event" and "event" interchangeably. In some BSMLs, such as Esterel [9] and Argos [41], an internal event has a scope, which makes it local to a part of the model. We can treat such BSMLs and their semantics similar to BSMLs with global scope internal events, by assuming a proper renaming that turns scoped events into global events.

3.6.1 LifeLine

A major semantic aspect for internal events is when a generated event in a model can be sensed by the event triggers of the transitions in its big-step. We describe this semantic aspect by introducing the notion of the *lifeline* of a generated event. The lifeline semantics of a BSML specifies the intermediate snapshots of a big-step in which a generated event is present (i.e., it can enable the transitions of its big-step). In some semantics, an event can have multiple *instances* during a big-step, each of which is associated with one or

Option Description	Advantages	Disadvantages	Example
Lifeline and global consistency semantic options			
WHOLE	Compatible with the synchrony hypothesis, and modular with respect to internal events	Non-causal big-steps, and forward-referencing semantics	19
REMAINDER NOT GLOBALLY CONSISTENT	Causal and non-forward-referencing semantics	Does not support the synchrony hypothesis, the order that events are generated does not affect the order they enable other transitions	19 and 22
REMAINDER GLOBALLY CONSISTENT	Compatible with the synchrony hypothesis, forward-referencing semantics, and causal	The order that events are generated does not affect the order they enable other transitions	19 and 22
NEXT COMBO-STEP	Causal, non-forward-referencing semantics, and provides more rigorous ordering of transitions via the notion of combo-step	Does not support the synchrony hypothesis, and complications of combo-step semantics	20
NEXT SMALL-STEP	Causal, non-forward-referencing semantics, the equality of the order of small-steps and the causal order of event generations and event triggers	Does not support the synchrony hypothesis	20 and 21
SAME	Atomic communication, non-forward-referencing semantics, and algebraic flavour of the semantics	Non-causal big-steps	21

Table 10: Internal Events: Semantic options and their advantages and disadvantages.

more transitions of the big-step that generate it. Each instance of an event has its own lifeline, which is a contiguous sequence of the intermediate snapshots of its big-step where it is present. Two instances of the same event have disjoint lifelines. The lifeline semantics of an instance of an event differ in: (i) where in the sequence of the intermediate snapshots of a big-step a generated event becomes present first, and (ii) how far its present status persists in the sequence of the intermediate snapshots. In this section, we present five lifeline semantics for BSMLs.

WHOLE: In this lifeline semantics, a generated event is: (i) considered present from the beginning of the current big-step, regardless of the small-step in which it has been generated; and (ii) persists until the end of the big-step. Examples of the BSMLs that use this semantic option are Argos [41] and Esterel [9], both of which subscribe to the "perfect" [9, 41] synchrony hypothesis. In this option, there could exist at most one instance of an event during a big-step.

An **advantage** of this semantic option is that it is *modular* [32] with respect to events. For a semantics to be modular [32], an event generated by the model should be treated the same as if it was received from the environment. In the WHOLE option, a generated event is available from the beginning of its big-step, which is the same as the events received from the environment. The order of the generation of events in a big-step does not matter. This order independence is in accordance with the vision of the "perfect" [9, 41] synchrony hypothesis, where a big-step is assumed to take "zero-time," and thus a generated event cannot persist for only a part of its big-step. As a result, and as an **advantage**, the constituent small-steps of a big-step can be considered as a set, instead of a sequence, as far as events are concerned.

A **disadvantage** of this semantic option is that it permits *non-causal* big-steps [9, 11, 32]. A big-step is causal if it is possible to sequence its constituent transitions such that an event can enable a transition only if it is generated by a previous transition in the sequence of small-steps. Non-causality may lead to nonintuitive behaviours where transitions seem to execute "out of the blue." Furthermore, depending on other semantic aspects of a BSML (such as its maximality semantics and the possibility of the negation of events), it can be impossible to define a semantics for a BSML without considering the effects of the future transitions in a big-step. We call such a semantics a "forward-referencing semantics," as described in Section 2.

To avoid non-causal big-steps, some semantics introduce a notion of a "correct" model, which is a model that regardless of the environmental inputs that it receives, it never generates a big-step with non-causal transitions [9, 62, 11]. Developers of such BSMLs constantly try to improve their semantics, and sometimes syntax, to identify the "incorrect" models more effectively [11]. There are a variety of analysis tools, some of which conservatively detect incorrect models, and reject them at compile time [20, 11]. As one might expect, in the presence of variables, the detection of incorrect models is undecidable [20]. Another **disadvantage** of this option is that if an event is generated by multiple transitions within the same big-step, it is only considered as one instance.[18] Furthermore, if events have value parameters, as in Esterel [9], then there should exist a function that *combines* the values of the parameters of multiple instances of a generated event.

REMAINDER: In this option, a generated event: (i) is considered present in the intermediate snapshot after the small-step that generates it, and (ii) persists during the remainder of the big-step. Examples of this semantic option are the original Statecharts [22], H&P&S&S Statecharts [26], and P&S Statechart [55]. In this option, similar to the previous option, there could exist at most one instance of an event during a big-step.

An **advantage** of this semantic option is that there is a clear causality relationship between transitions: a transition cannot be taken until its trigger events have been generated by the earlier small-steps of the big-step. The description of this semantics is intuitive for modellers because the status of an event can be calculated in a non-forward-referencing way. A **disadvantage** of this semantic option is that the effects of the generated events of a big-step are not necessarily sequenced (i.e., in a sequence of small-steps, if event e_1 is generated earlier than event e_2, then transitions that are triggered with e_1 do not necessarily execute earlier than the ones triggered with e_2). Another **disadvantage** of this semantic option is that it does not support the synchrony hypothesis, because the state of an event in a big-step can be both absent and present,

[18]To alleviate this problem, it is possible to create a semantics in which the events are considered as a bag, and have generated events match event triggers in the number of copies of the same event.

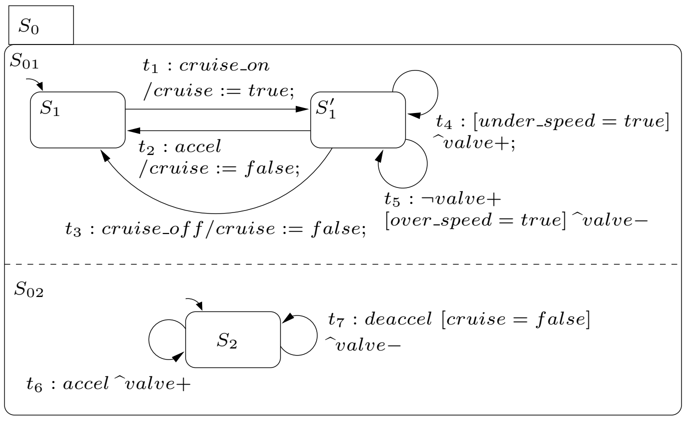

S_0
S_{01}
S_1
S'_1
$t_1 : cruise\_on$
$/cruise := true;$
$t_2 : accel$
$/cruise := false;$
$t_3 : cruise\_off/cruise := false;$
$t_4 : [under\_speed = true]$
$\frown valve+;$
$t_5 : \neg valve+$
$[over\_speed = true] \frown valve-$
S_{02}
S_2
$t_6 : accel \frown valve+$
$t_7 : deaccel\ [cruise = false]$
$\frown valve-$

Figure 24: Speed control system for a car.

which contradicts the notion of "zero-time" computation.

Example 19 *The model in Figure 24 is a simple, and naive, speed control system of a car that regulates the amount of power transmitted to the wheels by adjusting the amount of openness of the valves in its engine. The Concurrent-state* S_{01} *communicates with the accelerator sensor of the car; events* accel *and* deaccel *specify whether the accelerator is being pressed or depressed, respectively. Events* valve+ *and* valve− *can instruct the engine to slightly increase or decrease the amount of fuel that can be passed via valves into the engine, making the car move faster or slower, respectively. Events* cruise_on *and* cruise_off *turn the cruise control system on or off; if cruise control system is on (i.e.,* cruise = true*, then the system automatically adjusts the valve to maintain the desired speed). We assume that boolean variables* over_speed *and* under_speed*, which specify whether the vehicle is moving faster or slower than the target speed of the cruise control system, are set properly by some other parts of the system, not shown here. If the cruise control system is on, de-accelerating does not have any effect on how the valve is controlled. But if the cruise control system is on, and event* accel *is received, then the cruise control system is turned off, and event* accel *is processed as usual. In this example, we assume the* SINGLE *concurrency and the* TAKE ONE *maximality semantics. If we start from snapshot* $(\{\mathrm{S}'_1, \mathrm{S}_2\}, \{\mathrm{over\_speed} = \mathrm{true}, \mathrm{under\_speed} = \mathrm{false}, \mathrm{cruise} = \mathrm{true}\}, \{\mathrm{accel}\})$*, and assume the* WHOLE *lifeline semantics, the only possible big-step is:*

$$\begin{array}{ll}
(\{S'_1, S_2\}, \{over\_speed = true, under\_speed = false, cruise = true\}, \{accel, valve{+}\}) & \xrightarrow{t_6} \\
(\{S'_1, S_2\}, \{over\_speed = true, under\_speed = false, cruise = true\}, \{accel, valve{+}\}) & \xrightarrow{t_2} \\
(\{S_1, S_2\}, \{over\_speed = true, under\_speed = false, cruise = false\}, \{accel, valve{+}\}), &
\end{array}$$

where valve+ *is generated by* t_5*, but it appears right from the first snapshot, because of the* WHOLE *semantic option.*

If we assume the REMAINDER *lifeline semantics, additionally the following big-step is possible:*

$$\begin{array}{ll}
(\{S'_1, S_2\}, \{over\_speed = true, under\_speed = false, cruise = true\}, \{accel\}) & \xrightarrow{t_5} \\
(\{S'_1, S_2\}, \{over\_speed = true, under\_speed = false, cruise = true\}, \{accel, valve{-}\}) & \xrightarrow{t_6} \\
(\{S'_1, S_2\}, \{over\_speed = true, under\_speed = false, cruise = true\}, \{accel, valve{-}, valve{+}\}), &
\end{array}$$

which is not an intended behaviour of the system, because it both decreases and increases the openness of the valve.

NEXT COMBO-STEP: The third semantic option for an event's lifeline is that a generated event is considered present only during the combo-step after the combo-step in which it was generated. Examples of this semantics are Statemate [24] and RSML [38]. In this option, there could exist more than one instance of an event during a big-step, the lifeline of each instance being its corresponding combo-step.

In this semantic option, a generated event can only influence the enabledness of the transitions in the immediate next combo-step. Therefore, as an **advantage**, compared to the two previous options, this option provides a "more rigorous causal ordering" [38] between the generated events and the triggering events of the transitions of a big-step. A clear causal ordering between the transitions of a big-step is helpful for modellers, because it allows them to understand and analyze a model more effectively, by focusing on the limited parts of the model that can effect the behaviour of the model in the next small-steps. In this option, it is sufficient for a modeller to focus on the effect of the previous combo-step of the model to determine the next combo-step of a model; as opposed to the REMAINDER option, where a modeller needs to consider the entire big-step so far, or the WHOLE option, where a modeller needs to consider the entire big-step, to determine the next small-step of the big-step. A **disadvantage** of this option is that an event can have multiple instances, which is incompatible with the synchrony hypothesis, where the status of an event during a big-step should be either absent or present. Furthermore, when analyzing a model it is not easy to determine the different instances of an event in a big-step.

NEXT SMALL-STEP: In this semantic option a generated event is present only for the next small-step (i.e., it is only present in the intermediate snapshot after it has been generated). An example of this semantics is a variation of Statecharts in [15].

An **advantage** of this semantic option is that it provides a clear causal order between the generated and triggering events of the transitions of a big-step. This causal order is even more "rigorous" than the NEXT COMBO-STEP option; in fact, it is the same as the order of the small-steps of a big-step. As a **disadvantage**, similar to the NEXT COMBO-STEP option, this option allows multiple instances of the same event to exist in the same big-step, which can be a source of confusion for modellers.

Example 20 *The model in Figure 25 shows a fire alarm system for a building that performs two activities: (i) in the case of a smoke detection (i.e., the presence of event* smoke*), it turns on the siren and flashes the emergency lights, using* siren_on *and* flash_on *events, respectively; and (ii) in the case of detecting an excess heat (i.e., the presence of event* heat*), in addition to the actions for smoke detection, it turns on the extinguisher via* extin_on *event, which in turn, opens the valves of the extinguishing fountains in the building, by generating* valve_open *event. Concurrent-states* S_{01} *and* S_{04} *are responsible for dealing with the excess heat scenario;* S_{01} *directs* S_{04}*, using* emerg_on, emerg_off*,* extin_on*, and* extin_off *events. Similarly, Concurrent-states* S_{02} *and* S_{03} *are responsible for dealing with smoke;* S_{02} *directs* S_{03}*. Once the alarm or fire extinguishing system is activated, it can only be deactivated via a* reset *event. In this example, we assume the* SINGLE *concurrency and the* TAKE MANY *maximality semantics for big-steps and the* TAKE MANY *maximality semantics for combo-steps. If we start from snapshot* $(\{\mathrm{S}_1, \mathrm{S}_2, \mathrm{S}_3, \mathrm{S}_4\}, *, \{\mathrm{heat}, \mathrm{smoke}\})$*, and assume the* NEXT COMBO-STEP *lifeline for the generated events, and assume that environmental inputs (*heat *and* smoke *in our example) persist, then the following big-step can be taken:*

$$\begin{array}{ll}
(\{S_1, S_2, S_3, S_4\}, *, \{heat, smoke\}) & \xrightarrow{t_1} \\
(\{S_1', S_2, S_3, S_4\}, *, \{heat, smoke\}) & \xrightarrow{t_5} \\
(\{S_1', S_2, S_3', S_4\}, *, \{heat, smoke, emerg\_on, extin\_on\})| & \xrightarrow{t_3} \\
(\{S_1', S_2', S_3', S_4\}, *, \{heat, smoke, emerg\_on, extin\_on\}) & \xrightarrow{t_7} \\
(\{S_1', S_2', S_3', S_4'\}, *, \{heat, smoke, siren\_on, flash\_on, valve\_open\}). &
\end{array}$$

In the above big-step, as usual, we use "|" to specify the end of a combo-step. All generated events during a combo-step are only available at the end of the combo-step, making them usable for the next combo-step. Similar big-steps can be derived by replacing the order of the execution of t_1 *and* t_5 *and/or* t_3 *and* t_7*, all of them arriving at the same destination snapshot.*

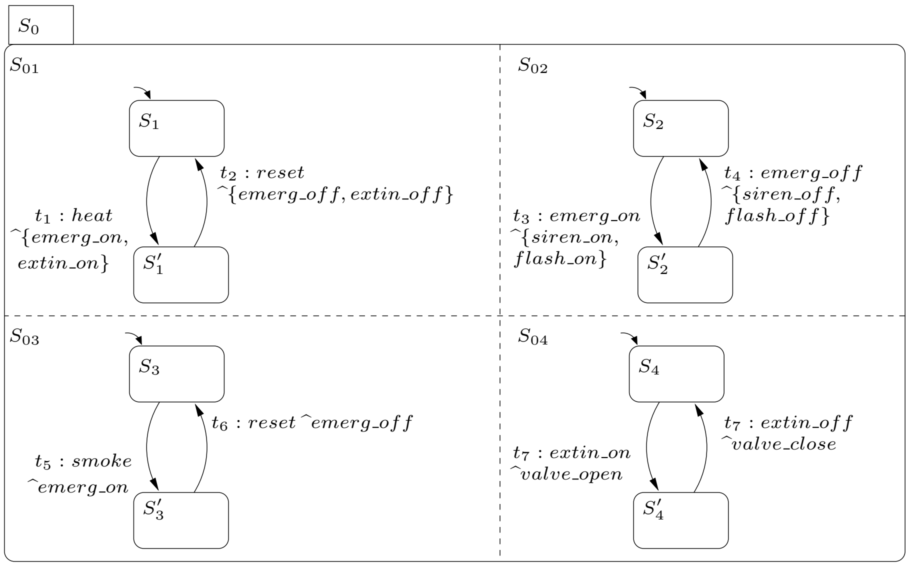

S_0
S_{01}
S_1
$t_2 : reset$ $\hat{}\{emerg\_off, extin\_off\}$
$t_1 : heat$ $\hat{}\{emerg\_on, extin\_on\}$
S'_1
S_{02}
S_2
$t_4 : emerg\_off$ $\hat{}\{siren\_off, flash\_off\}$
$t_3 : emerg\_on$ $\hat{}\{siren\_on, flash\_on\}$
S'_2
S_{03}
S_3
$t_6 : reset \hat{} emerg\_off$
$t_5 : smoke$ $\hat{} emerg\_on$
S'_3
S_{04}
S_4
$t_7 : extin\_off$ $\hat{} valve\_close$
$t_7 : extin\_on$ $\hat{} valve\_open$
S'_4

Figure 25: A fire alarm system.

If we assume the Next Small-Step *lifeline semantics, then the following big-step is possible:*

$$\begin{array}{ll}(\{S_1, S_2, S_3, S_4\}, *, \{heat, smoke\}) & \xrightarrow{t_5} \\ (\{S_1, S_2, S'_3, S_4\}, *, \{heat, smoke, emerg\_on\}) & \xrightarrow{t_3} \\ (\{S_1, S'_2, S'_3, S_4\}, *, \{heat, smoke, siren\_on, flash\_on\}) & \xrightarrow{t_1} \\ (\{S'_1, S'_2, S'_3, S_4\}, *, \{heat, smoke, emerg\_on\, extin\_on\}) & \xrightarrow{t_7} \\ (\{S'_1, S'_2, S'_3, S'_4\}, *, \{heat, smoke, valve\_open\}). & \end{array}$$

While the above big-step generates the same set of events as the previous one, none of other possible big-steps in this semantic option has the expected behaviour. For example, big-step

$$\begin{array}{ll}(\{S_1, S_2, S_3, S_4\}, *, \{heat, smoke\}) & \xrightarrow{t_1} \\ (\{S'_1, S_2, S_3, S_4\}, *, \{heat, smoke, emerg\_on, extin\_on\}) & \xrightarrow{t_3} \\ (\{S'_1, S_2, S'_3, S_4\}, *, \{heat, smoke, siren\_on, flash\_on\}) & \xrightarrow{t_5} \\ (\{S'_1, S'_2, S'_3, S_4\}, *, \{heat, smoke, emerg\_on\}) & \end{array}$$

does not open the valves.

Same: In the last option for lifeline semantics, a generated event is present during the current small-step (i.e., it is present in the source snapshot of the current small-step). The Same semantic option is different from other semantic options in that the generated events of a small-step cannot affect the enabledness of another small-step, making the two small-steps of a big-step independent of each other. This semantic option can be chosen only along with the Mandatory Synchronization or the Optional Synchronization concurrency semantics. An example of this semantic option is CCS [46], if we interpret labels and co-labels of CCS as events (CCS supports point-to-point communication, but can simulate broadcast [46, 19]).

An **advantage** of this option is that it has an algebraic flavour, and as such elegant algebraic properties can be derived. Furthermore, this option supports an atomic act of communication, in which an event communication happens exactly in one small-step. Similar to the Whole option, as a **disadvantage**, non-causal big-steps are possible. However, since most of the languages that subscribe to the Same option do not permit the syntax that can create non-causal big-steps (they usually permit a transition of a model either to have an event trigger or generate an event, but not both), the notions of non-causality is not as common as it is for the WHOLE option. Similar to NEXT COMBO-STEP and NEXT SMALL-STEP options, this option allows for multiple instances of the same event to exist in the same big-step, and thus, has the related **disadvantages**.

Example 21 *The model in Figure 26 is a simple model of an automatic flight control system that determines how two aircrafts, modelled in Concurrent-states* S_{01} *and* S_{02}*, can access a single runway. We only model how an aircraft obtains access to the runway when it wants to take off. We describe the take off procedure of the first aircraft, the same procedure applies to the second aircraft. When the first aircraft is ready to fly, a* fly_1 *event is initiated by its pilot, which: (i) results in a request to the automated control tower system for access to the runway, via event* $\mathrm{req}_1\_\mathrm{run}$*, and (ii) simultaneously turns on the take off lights of the aircraft. If the runway is empty then the synchronization between* t_1 *and* t_5 *happens, and the aircraft can use the runway. If the runway is not empty, the model blocks until* t_1 *and* t_5 *can synchronize. Once the aircraft takes off and it is sufficiently far from the runway, a* glide_1 *event is received from the aircraft, indicating that the aircraft is sufficiently far from the airport, and thus: (i) the runway can be released, and (ii) the take off lights can be turned off; this process includes the synchronization of transitions* t_2 *and* t_4*. In this example, we assume the* MANDATORY SYNCHRONIZATION *concurrency semantics and the* TAKE ONE *maximality semantics. If we assume the* SAME *lifeline semantics for events (*$\mathrm{req}_1\_\mathrm{run}$ *and* $\mathrm{req}_1\_\mathrm{run}$ *are the only "shared" events), and assume that input events* fly_1*,* fly_2*,* glide_1*, and* glide_2 *persist throughout a big-step, starting from snapshot* $(\{\mathrm{S}_1, \mathrm{S}_2, \mathrm{S}_3\}, *, \{\mathrm{fly}_1, \mathrm{fly}_2\})$*, the following two big-steps are possible:*

$$\begin{array}{ll}(\{S_1, S_2, S_3\}, *, \{fly_1, fly_2\}) & \xrightarrow{t_1, t_5} \\ (\{S_1', S_2, S_3'\}, *, \{fly_1, fly_2\}), & \end{array}$$

and

$$\begin{array}{ll}(\{S_1, S_2, S_3\}, *, \{fly_1, fly_2\}) & \xrightarrow{t_3, t_6} \\ (\{S_1, S_2', S_3'\}, *, \{fly_1, fly_2\}). & \end{array}$$

If we choose the OPTIONAL SYNCHRONIZATION *concurrency semantics, then the model does not behave correctly. For example, the following big-step would have been possible:*

$$\begin{array}{ll}(\{S_1, S_2, S_3\}, *, \{fly_1, fly_2\}) & \xrightarrow{t_1} \\ (\{S_1', S_2, S_3\}, *, \{fly_1, fly_2\}) & \xrightarrow{t_3, t_6} \\ (\{S_1', S_2', S_3'\}, *, \{fly_1, fly_2\}), & \end{array}$$

which allows both aircrafts to have access to the runway.

If we choose the NEXT SMALL-STEP *lifeline semantics along with the* SINGLE *concurrency semantics, then the system does not behave correctly; it allows both aircrafts to access the runway simultaneously. One of such undesired big-steps is the following big-step:*

$$\begin{array}{ll}(\{S_1, S_2, S_3\}, *, \{fly_1, fly_2\}) & \xrightarrow{t_1} \\ (\{S_1', S_2, S_3\}, *, \{fly_1, fly_2, req_1\_run, fly_1\_light\_on\}) & \xrightarrow{t_3} \\ (\{S_1', S_2', S_3\}, *, \{fly_1, fly_2, req_2\_run, fly_2\_light\_on\}) & \xrightarrow{t_6} \\ (\{S_1', S_2', S_3'\}, *, \{fly_1, fly_2\}). & \end{array}$$

3.6.2 Negation of Events and Global Inconsistency

To use the absence of an internal event to trigger a transition, a *negation* operator can be applied to the event. For an event e, its negation, denoted as $\neg e$, means the transition is enabled, only if e is absent. For a BSML, the semantics of the negation of events is defined based on its lifeline semantics for internal events. A transition that has $\neg e$ in its event trigger can be executed in the intermediate snapshots of the big-step that do not belong to the lifeline of e. As an example, consider Figure 27, which illustrates a big-step, its

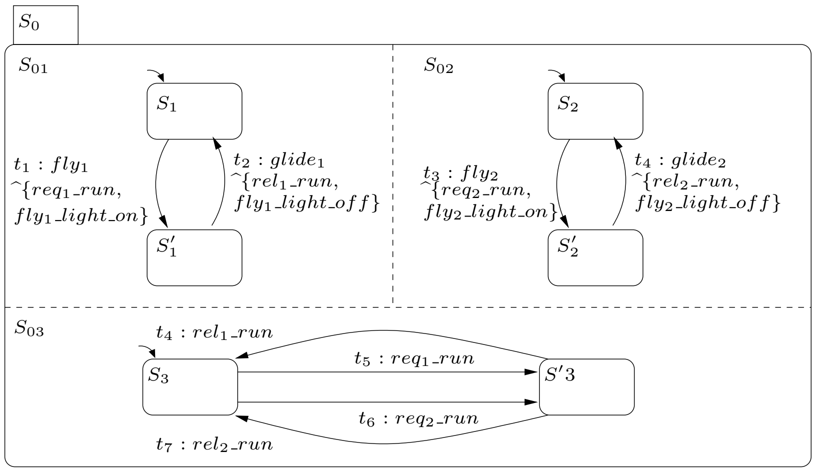

S_0
S_{01}
S_{02}
S_{03}
S_1
S_1'
S_2
S_2'
S_3
$S'3$
$t_1 : fly_1$ ^{req_1_run, $fly_1\_light\_on$}
$t_2 : glide_1$ ^{rel_1_run, $fly_1\_light\_off$}
$t_3 : fly_2$ ^{req_2_run, $fly_2\_light\_on$}
$t_4 : glide_2$ ^{rel_2_run, $fly_2\_light\_off$}
$t_4 : rel_1\_run$
$t_5 : req_1\_run$
$t_6 : req_2\_run$
$t_7 : rel_2\_run$

Figure 26: An airport runway control system.

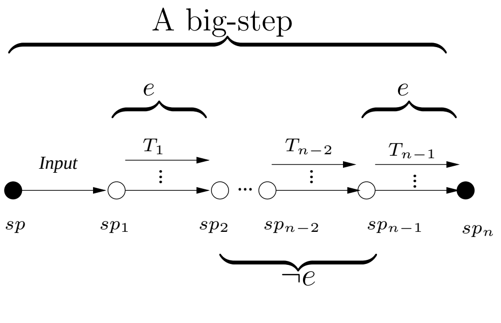


Figure 27: A big-step and the transitions that can be enabled by e and $\neg e$.

small-steps, and its intermediate snapshots. There are two instances of event e in the big-step. Event e is present in intermediate snapshots sp_1 and sp_{n-1}, which means a transition of small-step T_1 or T_{n-1} can have e as a literal in its event trigger, but cannot have $\neg e$ in its trigger. Event e is absent in the intermediate snapshots sp_i, where $(2 \leq i \leq n-2)$, which means a transition of a small-step T_i can have $\neg e$ in its trigger, but cannot have e in its trigger.

A *globally inconsistent* big-step is one that has a transition with $\neg e$ in its event trigger, as well as, a transition that generates e [54, 55].[19] A globally inconsistent big-step is conceptually undesirable because an event is considered both present and absent within the same big-step. Except for the WHOLE semantics, where an event is either present throughout a big-step or not present at all, the other four lifeline semantics can produce globally inconsistent big-steps.

In particular, global inconsistency is considered problematic for the REMAINDER semantics [54, 55]. The REMAINDER semantic option is used in the BSMLs that are meant to comply with the synchrony hypothesis, and thus one would expect an event to have a uniform status throughout a big-step. However, a globally inconsistent REMAINDER semantics permits the status of an event to be evaluated as absent in the early intermediate snapshots of a big-step before the event is generated, and to be evaluated as present after it has been generated. Furthermore, the REMAINDER semantics allow maximum one instance of an event in a

[19] In [54, 55], global inconsistency is studied only in the context of REMAINDER semantics.

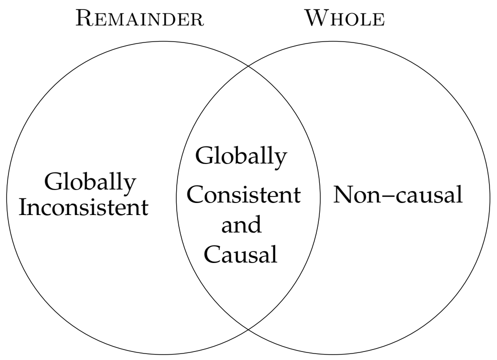


Figure 28: The relationshipts between the big-steps of the Whole and Remainder semantic options.

big-step (as opposed to the other three lifeline semantics), and thus if an event is generated in a big-step, it is natural for *the* instance of the event to be considered present throughout the big-step.

Global Consistency: If a semantics that subscribes to the Remainder semantic option somehow avoids the problem of "global inconsistency," then it is a *globally consistent* semantics [54, 55]. Figure 28 shows the relationships between the big-steps of the Remainder semantics and the Whole semantics. Figure 28 shows the relationship between the big-steps of the Remainder semantics and the Whole semantics. A big-step T that is produced by a globally consistent Remainder semantics can be also produced by a Whole semantics because T's generated events, by the definition of global consistency, can be assumed to be present from the beginning of the big-step. Conversely, a big-step T' that is produced by a causal Whole semantics can be also produced by a Remainder semantics because, by the definition of causality, an event is sensed as present by a transition of T' only if it is already generated in the big-step. Therefore, if global consistency is guaranteed syntactically (e.g., there are no negated event triggers) then the Remainder semantics generates a subset of the big-steps of the Whole semantics.

Unfortunately, the non-forward-referencing flavour of a Remainder semantics is lost when it is turned into a globally consistent semantics [54, 55]. However, the semantic description of such a semantics could be defined *declaratively* [54, 55]. To overcome the problem of a forward-referencing semantics, in [40], a globally consistent semantics is described that would lead to counter-intuitive big-steps that may avoid taking some enabled transitions. The difference between the semantics of [54, 55] and [40] is that the former semantics follows the Take One maximality semantics, where as the latter semantics has its own ad-hoc maximality criteria, in which a big-step might be ended prematurely to avoid the global inconsistency (i.e., an enabled transition that would have been taken according to the the Take One maximality semantics is not taken).

Example 22 *We consider the model in Example 19, illustrated again in Figure 29. Similar to Example 19, we assume the* Single *concurrency and the* Take One *maximality semantics. If we assume the* Remainder *lifeline semantics, starting from snapshot* $(\{\mathrm{S}_1', \mathrm{S}_2\}, \{\mathrm{over\_speed} = \mathrm{true}, \mathrm{under\_speed} = \mathrm{false}, \mathrm{cruise} = \mathrm{true}\}, \{\mathrm{accel}\})$*, similar to Example 19, the following big-step is possible:*

$$\begin{array}{lr}
(\{S_1', S_2\}, \{over\_speed = true, under\_speed = false, cruise = true\}, \{accel\}) & \xrightarrow{t_4} \\
(\{S_1', S_2\}, \{over\_speed = true, under\_speed = false, cruise = true\}, \{accel, valve-\}) & \xrightarrow{t_5} \\
(\{S_1', S_2\}, \{over\_speed = true, under\_speed = false, cruise = true\}, \{accel, valve-, valve+\}), &
\end{array}$$

which is not a globally consistent big-step, because "valve+" is both considered absent and is generated during the big-step.

If we choose the global consistency semantics in the style of [54, 55], then the above big-step is not possible,

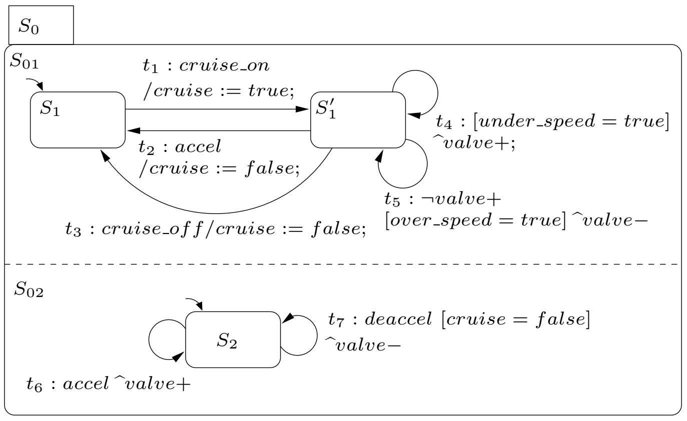

S_0
S_{01}
S_1
S_1'
t_1 : $cruise\_on$
$/cruise := true$;
t_2 : $accel$
$/cruise := false$;
t_3 : $cruise\_off/cruise := false$;
t_4 : $[under\_speed = true]$
$\hat{}valve+$;
t_5 : $\neg valve+$
$[over\_speed = true]\hat{}valve-$
S_{02}
S_2
t_6 : $accel\hat{}valve+$
t_7 : $deaccel$ $[cruise = false]$
$\hat{}valve-$

Figure 29: Speed control system for a car, copied from Figure 24.

and the only possible big-step is:

$$\begin{array}{ll}(\{S_1', S_2\}, \{over\_speed = true, under\_speed = false, cruise = true\}, \{accel\}) & \xrightarrow{t_5} \\ (\{S_1', S_2\}, \{over\_speed = true, under\_speed = false, cruise = true\}, \{accel, valve+\}) & \xrightarrow{t_2} \\ (\{S_1, S_2\}, \{over\_speed = true, under\_speed = false, cruise = false\}, \{accel, valve+\}), & \end{array}$$

which is similar to the first big-step in Example 24, when the Whole *option was chosen.*

If we choose a non-forward-referencing, globally consistent semantics for lifeline semantics in the style of [40], then in addition to the previous big-step, the following globally consistent big-step is also possible:

$$\begin{array}{ll}(\{S_1', S_2\}, \{over\_speed = true, under\_speed = false, cruise = true\}, \{accel\}) & \xrightarrow{t_4} \\ (\{S_1', S_2\}, \{over\_speed = true, under\_speed = false, cruise = true\}, \{accel, valve-\}), & \end{array}$$

which only includes one small-step. Transition t_5 *can be executed at the end of the big-step above, but because it generates* valve+, *and thus violates the global consistency of the big-step, it is not executed.*

Global consistency can be considered for BSMLs that subscribe to the Next Combo-Step, the Next Small-Step, and the Same semantics too. Three alternative notions of global consistency for the Next Combo-Step semantics are:

– *Per Big-Step Consistency*: A generated event in a combo-step of a big-step should not be evaluated as absent in the triggering condition of a transition of the combo-step or any previous combo-steps.

– *Per Combo-Step Consistency:* If an event is evaluated as absent in the event trigger of a transition of a combo-step, the event should not be generated in the same combo-step.

– *Future Consistency:* a generated event in a combo-step, should not be considered absent in the current combo-step and all future combo-steps. (This requirement is different from stating that a generated event in a combo-step is present in all future combo-steps, instead it says it cannot be considered absent in any future combo-step).

Similar notions can be considered for the Next Small-Step and the Same semantics.

Table 11 summarizes our discussions about the semantic properties of events and negation of events.

Semantic options / *Semantic Property*	Whole	Remainder	Remainder Globally Consistent	Combo-Step	Next Small-Step	Same
Synchrony Hypothesis	Yes	No	Yes	No	No	No
Global Consistency	Yes	No	Yes	Yes	Yes	Yes
Causality	No	Yes	Yes	Yes	Yes	NO
Modularity	Yes	No	No	No	No	No
Multiple-Instance	No	No	No	Yes	Yes	Yes
Non-forward-referencing Semantics	No	Yes	No	Yes	Yes	Yes
Causal Order = Small-Step Order	No	No	No	No	Yes	No

Table 11: Event semantic options and their properties.

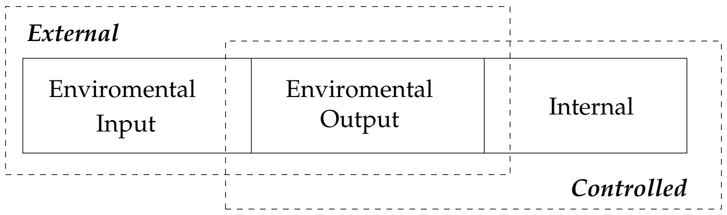


Figure 30: The taxonomy of types of events when distinguishing between a model and its environment.

3.6.3 Implicit Events:

A BMSL may include syntactic constructs that recognize implicit events. Some examples of such events are: `entered(`s`)` [64], which is generated when control state s is entered in a small-step; `@T(`c`)` [27], which is generated when the status of a variable condition c changes from false to true; and `change_eve(`v`)` [3], which is similar to the operator `change` described in Section 3.4.2, but is an event, rather than a condition, and is generated when a new assignment is made to v. By default, these implicit events would have the same lifeline as internal events of the model, however, separate lifeline semantics could be chosen for each implicit event.

3.7 External Event Communication

To avoid modelling flaws, as with variables, a BSML may syntactically distinguish between those events that are used by a model for communication with its environment (*external events*) and those events that are used by a model for communication between its different parts ("internal events"). The set of external events of a model can be further partitioned into the set of *environmental input events* and the set of *environmental output events*. Figure 30 illustrates this taxonomy of events, which is similar to the one for variables in Figure 19 (here internal events play the same role as private variables in the taxonomy for variables). As before, a dashed box in the figure represents a set that includes two or more sets of events (e.g., *controlled events* = *internal events* ∪ *environmental output events*). Often, the lifeline semantics of the environmental input events is the Whole semantics, and the lifeline semantics of the environmental output events is the Remainder semantics. This means that an environmental input event, once received from the environment in the beginning of a big-step, persists throughout the big-step, and an environmental output event, once generated during the big-step, persists throughout the remainder of big-step.

In the remainder of this section, we start by discussing the "inter-component communication" mechanisms for events (summarized in Table 12), which is similar to that of variables. We then consider the complexities

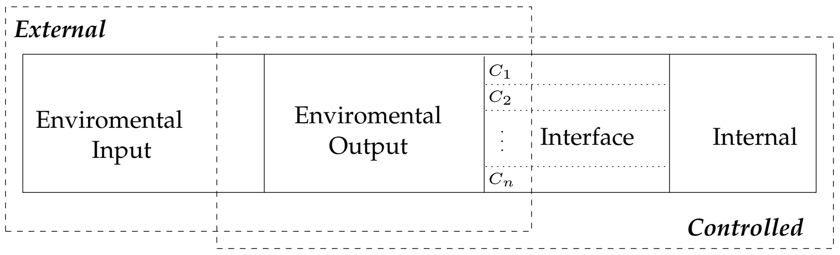


Figure 31: The taxonomy of types of events for inter-component communication.

Option Description	Advantages	Disadvantages	Example
SYNCHRONOUS EVENT			
STRONG SYNCHRONOUS EVENT	Compatible with the synchrony hypothesis and modular with respect to interface events	Non-causal, and forward-referencing semantics	23
WEAK SYNCHRONOUS EVENT	Causal, and non-forward-referencing semantics	Not compatible with the synchrony hypothesis	23
ASYNCHRONOUS EVENT	Non-forward-referencing semantics and modular with respect to interface events	Not compatible with the synchrony hypothesis	23

Table 12: Inter-component Event Communication: summary of semantic variations, and their advantages and disadvantages.

of the semantics of the BSMLs that do not syntactically distinguish between the internal and external events of a model (summarized in Table 13).[20]

3.7.1 Inter-component Event Communication

Events can be used to carry out "inter-component communication," similar to variables as described in Section 3.5.1. A model may consist of multiple *components*, each of which represents a physically distinct part of a system. An *inter-component event communication* mechanism uses the *interface events* of a model to provide the means for communication between the components. Figure 31 illustrates the taxonomy of types of events, when the inter-component event communication mechanism is considered (it is similar to the taxonomy for variables in Figure 21). Interface events of a model are syntactically distinguished from other types of events in the model. Similar to interface variables, for an interface event, there is a notion of its *sending* component (*the* component that generates the event) and its *receiving* components (the components that use the event in the event trigger of their transitions). Also, similar to interface variables, the set of interface variables of a model are partitioned into sets, each of which includes all of the interface events that a component generates (this partitioning is shown by dotted lines in Figure 31).

The lifeline semantics of the interface events of a BSML is not necessarily the same as the lifeline semantics of other types of events in the BSML. Similar to interface variables, we consider two types of inter-component communication: one in the spirit of the synchrony hypothesis (the SYNCHRONOUS EVENT option) and one

[20]When discussing variables in Section 3.5, we did not consider the semantics of the BSMLs that do not distinguish between external and private variables, because, in practice, we are not aware of such BSMLs.

in the spirit of delayed communication (the ASYNCHRONOUS EVENT option).

SYNCHRONOUS EVENT: In this inter-component communication mechanism, once a sending component generates an interface event during a big-step, the generated event becomes available for the receiving components within the same big-step. By analogy, a semantics that subscribes to this option provides a rendezvous-like communication for the interface events at a big-step level. Two semantic sub-options for this semantics are:

- STRONG SYNCHRONOUS EVENT: In this option, an interface event in a receiving component is either present throughout a big-step from the beginning, or is absent throughout the big-step. This semantics is the WHOLE lifeline semantics for interface events. An example of this semantic option is "Hybrid Semantics" in [32], which distinguishes between the "local" and the "global" events of a model, and treats the "global" events according to the STRONG SYNCHRONOUS EVENT semantics. The STRONG SYNCHRONOUS EVENT option has the advantages and the disadvantages of the WHOLE semantic option. As its **advantages**: it is compatible with the synchrony hypothesis, and is modular with respect to interface events; and as its **disadvantages**: it is non-causal, and has a forward-referencing semantics.

- WEAK SYNCHRONOUS EVENT: In this semantic option, an interface event need not be present from the beginning of a big-step. However, if an interface event is generated by a sending component, the receiving components would receive the event during the current big-step. This semantics is the REMAINDER lifeline semantics for interface events. A STRONG SYNCHRONOUS EVENT semantics, similar to the REMAINDER lifeline semantics, may be "globally consistent" or not. The WEAK SYNCHRONOUS EVENT has similar advantages and disadvantages as the REMAINDER semantic option. Considering a globally inconsistent variation of the WEAK SYNCHRONOUS EVENT option, it has the **advantages** of being causal, and having a non-forward-referencing semantic description, and has the **disadvantage** of not supporting the synchrony hypothesis.

ASYNCHRONOUS EVENT: In this semantic option, a generated interface event will be sensed in the big-step after the big-step in which it is generated. For example, in RSML [38], an "output" event is generated by a `SEND` command, which can be received by a destination component via a `RECEIVE` event in the next big-step. In Esterel [2], a `registered` event has the ASYNCHRONOUS EVENT semantics, but it is not used for inter-component event communication. In *globally asynchronous locally synchronous (GALS)* languages [12, 59], the communication of events within the "local" components of a system follow the semantics of the WHOLE option, and the "global" communication of events between the components follow the semantics of the ASYNCHRONOUS EVENT option. An **advantage** of this option is that the semantics of the interface events can be described in a non-forward-referencing way. Furthermore, this semantic option is modular with respect to interface events (similar to the ASYNCHRONOUS option for interface variables) A **disadvantage** of this option is that it does not comply with the synchrony hypothesis. In this option, a generated interface event persists during the next big-step. However, different semantic variations could be considered for the persistence of a generated interface event in the next big-step.

Example 23 *The model in Figure 32 is similar to the model in Example 18, but has been modified to use events, instead of variables. The model illustrates a system that controls the unlocking of the entrance to an industrial area. The door is only unlocked if the temperature is below 40°C. We use the thick dashed line to specify the two components of the system. Events* danger *and* is_temp_ok *are interface events. Events* open *and* close *are the environmental input events of the model, and* lock *and* unlock *are the environmental output events of the model. Events* check_temp, heat, *and* ok *are internal events. We assume the* TAKE MANY *semantic option for maximality, the* SINGLE *semantic option for concurrency, and the* NEXT SMALL-STEP *lifeline semantics for internal events. Consider snapshot* $(\{S_1, S_2, S_3\}, \{\text{door} = \text{closed}, \text{temp} = 99\}, \{\text{open}\})$. *If we assume the inter-component event communication model of* STRONG SYNCHRONOUS EVENT, *then the following big-step can only be taken:*

$$\begin{array}{ll}
(\{S_1, S_2, S_3\}, \{\mathit{door} = \mathit{closed}, \mathit{temp} = 99\}, \{\mathit{open}, \mathit{is\_temp\_ok}, \mathit{danger}\}) & \xrightarrow{t_1} \\
(\{S_1', S_2, S_3\}, \{\mathit{door} = \mathit{closed}, \mathit{temp} = 99\}, \{\mathit{open}, \mathit{is\_temp\_ok}, \mathit{danger}, \mathit{check\_temp}\}) & \xrightarrow{t_6} \\
(\{S_1', S_2', S_3\}, \{\mathit{door} = \mathit{closed}, \mathit{temp} = 99\}, \{\mathit{open}, \mathit{is\_temp\_ok}, \mathit{danger}\}) & \xrightarrow{t_9} \\
(\{S_1', S_2', S_3'\}, \{\mathit{door} = \mathit{closed}, \mathit{temp} = 99\}, \{\mathit{open}, \mathit{is\_temp\_ok}, \mathit{danger}\}) & \xrightarrow{t_8} \\
(\{S_1', S_2, S_3'\}, \{\mathit{door} = \mathit{closed}, \mathit{temp} = 99\}, \{\mathit{open}, \mathit{is\_temp\_ok}, \mathit{danger}, \mathit{heat}\}) & \xrightarrow{t_3} \\
(\{S_1'', S_2, S_3'\}, \{\mathit{door} = \mathit{closed}, \mathit{temp} = 99\}, \{\mathit{open}, \mathit{is\_temp\_ok}, \mathit{danger}\}). &
\end{array}$$

If we assume the WEAK SYNCHRONOUS EVENT *option, then the above big-step is possible, but additionally the following big-step, which unlocks the door despite the high temperature, is possible too.*

$$\begin{array}{ll}
(\{S_1, S_2, S_3\}, \{\mathit{door} = \mathit{closed}, \mathit{temp} = 99\}, \{\mathit{open}\}) & \xrightarrow{t_1} \\
(\{S_1', S_2, S_3\}, \{\mathit{door} = \mathit{closed}, \mathit{temp} = 99\}, \{\mathit{open}, \mathit{check\_temp}\}) & \xrightarrow{t_6} \\
(\{S_1', S_2', S_3\}, \{\mathit{door} = \mathit{closed}, \mathit{temp} = 99\}, \{\mathit{open}, \mathit{is\_temp\_ok}\}) & \xrightarrow{t_7} \\
(\{S_1', S_2, S_3\}, \{\mathit{door} = \mathit{closed}, \mathit{temp} = 99\}, \{\mathit{open}, \mathit{is\_temp\_ok}, \mathit{ok}\}) & \xrightarrow{t_2} \\
(\{S_1, S_2, S_3\}, \{\mathit{door} = \mathit{closed}, \mathit{temp} = 99\}, \{\mathit{open}, \mathit{is\_temp\_ok}, \mathit{unlock}\}) & \xrightarrow{t_9} \\
(\{S_1, S_2, S_3'\}, \{\mathit{door} = \mathit{closed}, \mathit{temp} = 99\}, \{\mathit{open}, \mathit{is\_temp\_ok}, \mathit{danger}\}). &
\end{array}$$

Lastly, if we assume the ASYNCHRONOUS EVENT *option, then the system never behaves properly, because the external events are not exchanged in a timely manner. For example, the following undesired big-step:*

$$\begin{array}{ll}
(\{S_1, S_2, S_3\}, \{\mathit{door} = \mathit{closed}, \mathit{temp} = 99\}, \{\mathit{open}\}) & \xrightarrow{t_1} \\
(\{S_1', S_2, S_3\}, \{\mathit{door} = \mathit{closed}, \mathit{temp} = 99\}, \{\mathit{open}, \mathit{check\_temp}\}) & \xrightarrow{t_6} \\
(\{S_1', S_2', S_3\}, \{\mathit{door} = \mathit{closed}, \mathit{temp} = 99\}, \{\mathit{open}\}) & \xrightarrow{t_7} \\
(\{S_1', S_2, S_3\}, \{\mathit{door} = \mathit{closed}, \mathit{temp} = 99\}, \{\mathit{open}, \mathit{ok}\}) & \xrightarrow{t_2} \\
(\{S_1, S_2, S_3\}, \{\mathit{door} = \mathit{closed}, \mathit{temp} = 99\}, \{\mathit{open}\}). &
\end{array}$$

In the above big-step, the asynchronous communication of external events is_temp_ok *does not affect any snapshot of the big-step, because it is only accessible starting in the next big-step, although generated in the current big-step.*

3.7.2 Non-distinguishing BSMLs

Some BSMLs, such as H&P&S&S Statecharts [26], P&S Statecharts [54, 55], and Statemate [24], do not syntactically distinguish between the external events and the internal events of a model; we call such BSMLs *non-distinguishing* BSMLs. In a non-distinguishing BSML, an event can play the roles of both an environmental input event and an internal event. It is even possible that during the same big-step an event is both received from the environment (playing the role of an environmental input), and is generated by a transition (playing the role of a generated event). In a non-distinguishing BSML, we call an event that has been received from the environment at the beginning of a big-step a *virtual input* of the big-step. We call a virtual input that is not generated by any transition in the model a *genuine input* of the model. An important semantic aspect for non-distinguishing BSMLs is the possible lifeline semantics of virtual inputs and genuine inputs. There are three possibilities (as shown in Table 13):

ASSUME INPUT AS ENVIRONMENTAL: In this semantic option, a virtual input (i.e., an event received at the beginning of a big-step) is treated as an environmental input, and as such it will persist throughout a big-step (e.g., P&S Statecharts [54] and H&P&S&S Statecharts [26]). The lifeline semantics of a virtual input of a big-step is the WHOLE semantics. An **advantage** of this option is that, as expected in BSMLs, an environmental input event remains present throughout a big-step. A **disadvantage** of this option is that it is difficult for a modeller to discern between the two different roles of an event within a big-step. For example, an event e that is absent at the beginning of a big-step (i.e., not provided by the environment), can be generated by a transition and become present. However, a modeller, being focused

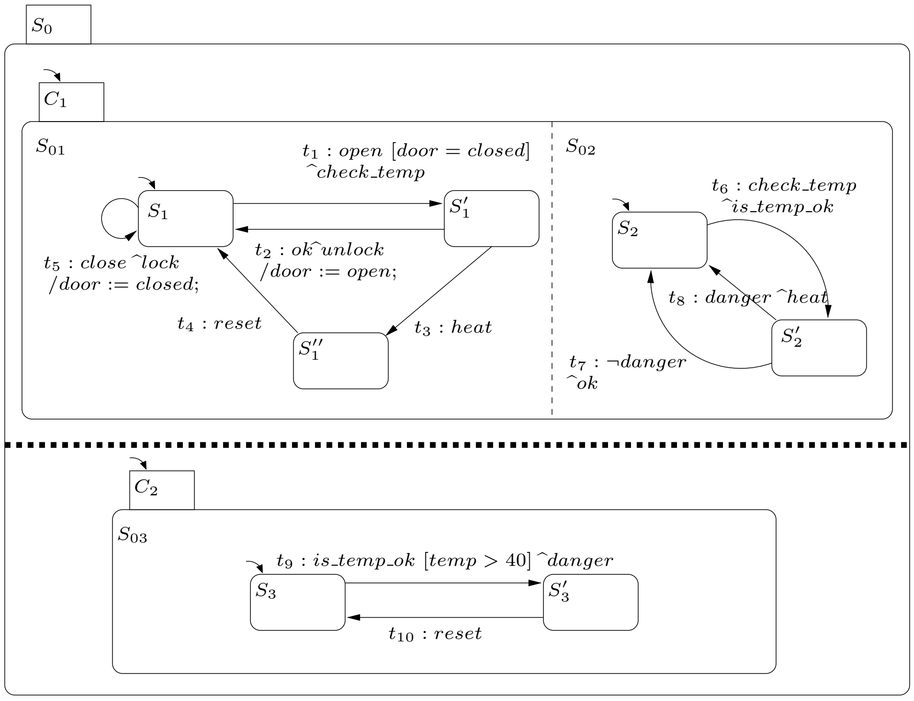


Figure 32: Using interface events *danger* and *is_temp_ok*.

Option Description	Advantages	Disadvantages	Example
ASSUME INPUT AS ENVIRONMENTAL	Compatible with the intuition of the environmental inputs	Complexity of discerning between the internal/external role of an event	24
ASSUME INPUT AS INTERNAL	Treating all of the events of a model uniformly	Multiple-instance/multiple-role events	24
HYBRID	Partly compatible with the intuition of the environmental inputs	Multiple-instance/multiple-role events	24

Table 13: Semantic options for communicating with environment when external and internal events are not syntactically distinguished.

on the role of e as an environmental input event, might mistakenly continue to assume that e is absent; moreover, based on the lifeline semantics of internal events, e might become absent again, which makes the evaluation of the status of e even more complicated. Conversely, if event e is present at the beginning of a big-step (i.e., provided by the environment), then it is possible for a transition to generate e during the big-step, which causes confusion because the generation of e does not affect the behaviour of the big-step (event e is already provided by the environment).

ASSUME INPUT AS INTERNAL: In this semantic option, a virtual input is considered the same as other events (i.e., the same as non-virtual input events). As an example, in RSML [38], a virtual input is treated the same as the lifeline semantics of other events, according to the NEXT COMBO-STEP lifeline semantics, which means that a virtual input is only available in the first combo-step of a big-step. An **advantage** of this semantics is that it has a simple semantic description, which treats all events uniformly. A **disadvantage** of this semantic option is that it does not comply with the vision of BSML notations, where an input event is expected to persist throughout a big-step. Another **disadvantage** of this option is that it is difficult for a modeller to discern between the two different roles of an event within a big-step. For example, a virtual input event of a big-step can have multiple instances, one initial instance due to its being a virtual input, and the others due to its being generated during the big-step.

HYBRID: This semantic option is similar to the ASSUME INPUT AS INTERNAL option, but treats a "genuine input event" of a model (i.e., a virtual input event that is not generated by any transition of the model) as an environmental input. The lifeline semantics of genuine input events is the WHOLE semantics. The **advantage** of this option is that once a modeller identifies a genuine input of a model (e.g., by a simple analysis of the model), it can always be treated as an environmental input event in all big-steps. However, for a virtual input event that is not a genuine input event, similar to the ASSUME INPUT AS INTERNAL option, there is the **disadvantage** of the difficulty for a modeller to discern between the two different roles of the event.

Example 24 *We consider the model in Example 9, which is illustrated again in Figure 33 for convenience. We assume a non-distinguishing BSML. We assume the* SINGLE *option for concurrency, the* NEXT SMALL-STEP *lifeline semantics for internal events, and the* TAKE MANY *for maximality. If we consider snapshot* $(\{\mathrm{S}_1, \mathrm{S}_2\}, *, \{\mathrm{tk}_0\})$ *as the first snapshot of the big-step, then* tk_0 *is a virtual input, as well as, a genuine input. If we assume the* ASSUME INPUT AS ENVIRONMENTAL *semantics, then similar to Example 9, only the following big-step is possible:*

$$\begin{array}{ll}(\{S_1, S_2\}, *, \{tk_0\}) & \xrightarrow{t_1} \\ (\{S_1', S_2\}, *, \{tk_0\}) & \xrightarrow{t_2} \\ (\{S_1, S_2\}, *, \{tk_0, tk_1\}) & \xrightarrow{t_1} \\ (\{S_1', S_2\}, *, \{tk_0\}) & \xrightarrow{t_2} \\ \cdots . & \end{array}$$

If we assume the HYBRID *option, then the above big-step is the only possible big-step.*

If we assume the ASSUME INPUT AS INTERNAL *option, then the following big-step is the only valid big-step:*

$$\begin{array}{ll}(\{S_1, S_2\}, *, \{tk_0\}) & \xrightarrow{t_1} \\ (\{S_1', S_2\}, *, \{tk_0\}), & \end{array}$$

which can be continued by a second big-step, when receiving the second input tk_0*:*

$$\begin{array}{ll}(\{S_1', S_2\}, *, \{tk_0\}) & \xrightarrow{t_2} \\ (\{S_1, S_2\}, *, \{tk_1\}) & \xrightarrow{t_3} \\ (\{S_1, S_2'\}, *, \{\}). & \end{array}$$

For non-distinguishing BSMLs, we did not consider the notion of inter-component event communication mechanism because distinguishing between the external and the internal events of a model is a more primitive

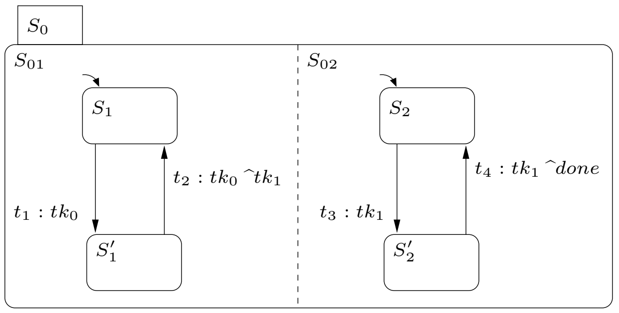


Figure 33: Revisiting the two-bit counter.

distinguishment than distinguishing between the component-controlled events of a model and its internal events. Furthermore, we are not aware of any non-distinguishing BSMLs that support an inter-component event communication mechanism.

3.8 Priority

There could exist multiple sets of transitions that can be chosen to be executed as a small-step of a model. We call the set of such sets of transitions that are enabled in a snapshot its set of *potential small-steps*. Commonly, a semantics *non-deterministically* chooses a set of transitions from the set of "potential small-steps" of a snapshot, as the next small-step of a big-step (e.g., Statecharts [22] and many of its variants [64]). Non-determinism is a rich means for modelling a system because at a high-level of abstraction, a modeller might be interested in specifying the alternative behaviours of the system, without specifying when each behaviours should be chosen. A non-deterministic model is usually refined into a deterministic model at a later stage of the life cycle of the system. Some BSMLs strive to avoid non-determinism behaviour via syntactic constraints, input assumptions (see Section 3.3.2), model analysis methods etc., because it is important in such BSMLs to create models that are deterministic (e.g., Esterel [9] and Argos [41]).

The *priority semantics* of a BSML specifies which set(s) of transitions should be chosen from the set of potential small-steps of a snapshot (there could exist more than one set of transitions with the highest priority). Table 14 shows three common ways of assigning priority to transitions that we describe in the remainder of this section. If two transitions are enabled in a certain snapshot, and by executing one of them the other one cannot be executed in the same small-step, then the priority semantics determines which transition to execute, based on the assigned priority of the transitions. A set of transitions T_1 has a higher priority than a set of transitions T_2, if for all transitions $t_1 \in T_1$ and all transitions $t_2 \in T_2$, either t_1 and t_2 are not comparable in terms of priority, or t_1 has a higher priority than t_2. A priority semantics might choose more than one set of transitions from a set of potential small-steps, in which case, each set of transitions has a higher priority than the sets of transitions that are not chosen and none of the chosen sets of transitions has a higher priority than any other chosen sets of transitions.

HIERARCHY: Some BSMLs use the hierarchical structure of the control states of a model as a way to assign priorities to transitions that are ancestrally related. According to the *basis* and the *scheme* of a HIERARCHY priority semantics, six priority semantics can be defined. The "basis" of a HIERARCHY semantics specifies the element of a transition that is considered in the semantics, and can be: SOURCE, DESTINATION, or ARENA options.[21] The "scheme" of a HIERARCHY semantics specifies whether being higher (i.e., being a parent) in a hierarchy of states gives higher or lower priority; possible options for the scheme of a semantics are: PARENT and CHILD. For example, the ARENA-CHILD is a HIERARCHY priority semantics that gives a higher priority to a transition whose arena is the lowest (i.e., its arena is the children of the arenas of other transitions) in the hierarchy of states. Some examples of HIERARCHY priority semantics are: the

[21]As described in Section 3.2, the arena of a transition is the smallest Or-state that includes both the source and destination states of a transition.

Option Description	Advantages	Disadvantages	Example
HIERARCHICAL	Graphically specified/understood, and precedence between transitions syntactically understood	Non-exhaustive	25
EXPLICIT	Great control over priority specification, and precedence between transitions syntactically understood	Tedious to use	26
NEGATION OF TRIGGERS	Seamless semantics with the BSML's semantics	Tedious to use, and precedence between transitions can be understood with respect to a snapshot, but not syntactically	27

Table 14: Advantages and disadvantages of different priority semantic options.

SOURCE-CHILD semantics in Rhapsody [23], the ARENA-PARENT semantics in Statemate [24], and the DESTINATION-CHILD Esterel [9] (see Section 3.2 for the role of hierarchical states in translating the syntax of Esterel to CHTS).

An **advantage** of using a HIERARCHY priority semantics is that it can be easily/graphically understood, and reviewed by a modeller. A **disadvantage** of using a HIERARCHY priority semantics is that in some models it may not be possible to provide an exhaustive prioritization amongst the sets of transitions in a set of potential small-steps. For example, consider the SOURCE-CHILD semantics and two sets of transitions that are enabled in a snapshot, each of which with a single transition whose source is at the same level of hierarchy as the other. None of the sets of transitions has a higher priority than the other, and it is not possible to give one set of transitions a precedence over another without modifying the model. It is possible to use two HIERARCHY semantics in the same BSML: one of them to act as the primary priority semantics, and the other to act as the secondary priority semantics that is considered only if the primary semantics cannot choose the set of transitions with the highest priority (e.g., using the SOURCE-CHILD as the primary semantics and the DESTINATION-CHILD semantics as the second semantics). (This idea can be extended to allow multiple HIERARCHY semantics in a single BSML.)

Example 25 *The model in Figure 34 is the same model as the model in Example 5. If we assume the* MANY *concurrency semantics, the* TAKE-ONE *maximality semantics, the* ARENA ORTHOGONAL *small-step consistency semantics, and consider the* SOURCE-PARENT *priority semantics, then starting from snapshot* $(\{\mathrm{S}_2, \mathrm{S}_3, \mathrm{S}_4\}, \{\mathrm{x} = 1\}, \{\mathrm{e}\})$*, the following big-step is the only big-step that can be taken:*

$$\begin{array}{l}(\{S_2, S_3, S_4\}, \{x = 1\}, \{e\}) \xrightarrow{t_6} \\ (\{S_5\}, \{x = 1\}, *).\end{array}$$

If we choose the ARENA-PARENT *priority semantics, again only the above big-step is possible.*

If we choose the SOURCE-CHILD *or* ARENA-CHILD *priority semantics, then the following two big-steps are possible:*

$$\begin{array}{l}(\{S_2, S_3, S_4\}, \{x = 1\}, \{e\}) \xrightarrow{\{t_1, t_2\}} \\ (\{S_2', S'3, S_4\}, \{x = 2\}, *),\end{array}$$

and

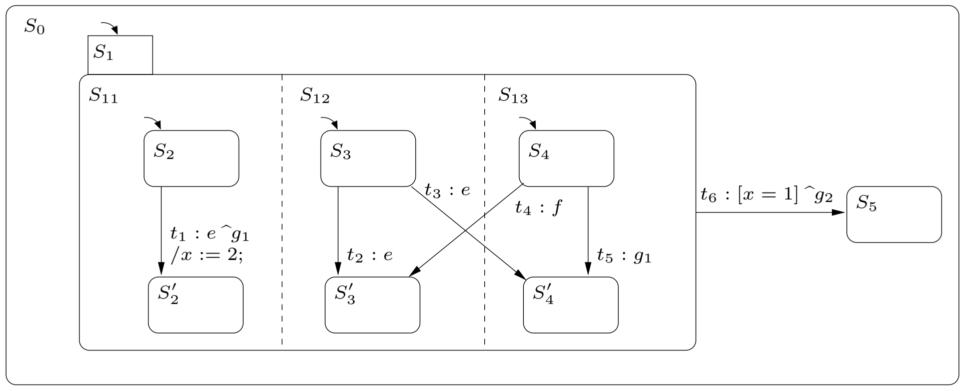


Figure 34: A model with hierarchical states.

$$\begin{array}{l}(\{S_2, S_3, S_4\}, \{x = 1\}, \{e\}) \quad \xrightarrow{\{t_3\}} \\ (\{S_2, S_3, S'_4\}, \{x = 1\}, *).\end{array}$$

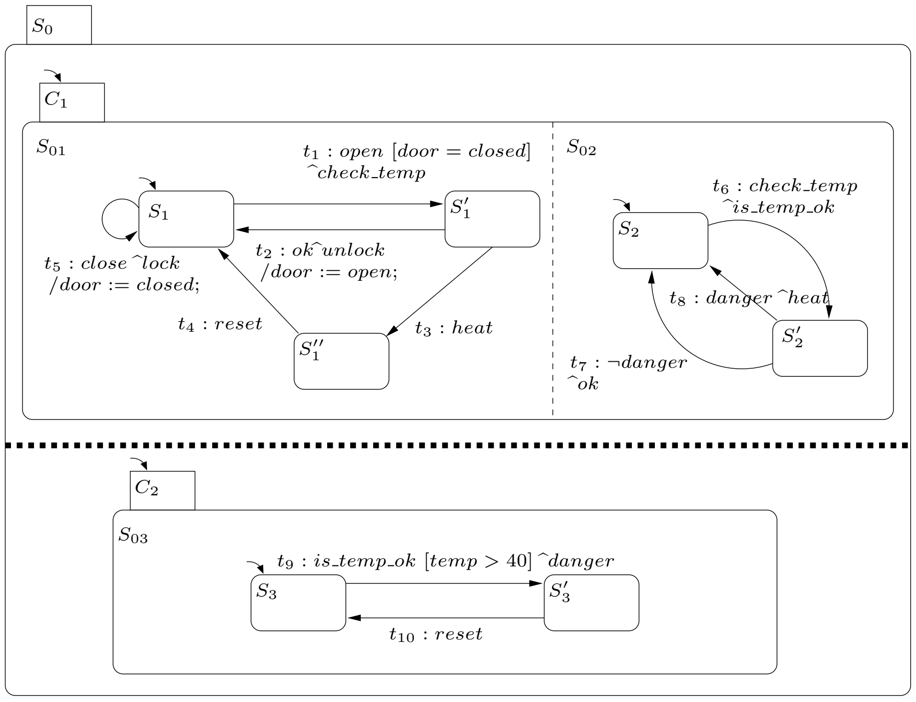
If we choose the DESTINATION-CHILD *or* DESTINATION-PARENT *priority semantics, then none of the enabled transitions* t_6*,* t_1*,* t_2*, and* t_3 *has priority over another, and therefore all above three big-steps can be taken.*

EXPLICIT: This option uses an explicit way to assign priority to the transitions of a model (e.g., assigning numbers to transitions where a bigger number means a higher priority). An **advantage** of this approach is that a modeller has a great control over specifying the priority of transitions. For example, for two orthogonal HTSs, a transition of one HTS can be assigned a priority such that its execution has precedence over the transitions of the other HTS, which is not possible to enforce in the HIERARCHY semantics. A **disadvantage** of this approach is that it can be tedious to assign explicit priorities to all transitions of a model: A modeller needs to know about all reachable snapshots of the model, otherwise a transition might be assigned a priority that is appropriate in one snapshot, but is weak/strong in another.

Example 26 *We consider the model in Example 23 again, which is copied in Figure 35 for convenience. In Example 23, starting from snapshot* $(\{\mathrm{S}_1, \mathrm{S}_2, \mathrm{S}_3\}, \{\mathrm{door} = \mathrm{closed}, \mathrm{temp} = 99\}, \{\mathrm{open}\})$*, when assuming the* TAKE MANY *semantic option for maximality, the* SINGLE *semantic option for concurrency, the* NEXT SMALL-STEP *lifeline semantics for internal events, and* WEAK SYNCHRONOUS *external event communication semantics, the following big-step that opens the door in spite of the high temperature is possible:*

$$\begin{array}{ll}(\{S_1, S_2, S_3\}, \{door = closed, temp = 99\}, \{open\}) & \xrightarrow{t_1} \\ (\{S'_1, S_2, S_3\}, \{door = closed, temp = 99\}, \{open, check\_temp\}) & \xrightarrow{t_6} \\ (\{S'_1, S'_2, S_3\}, \{door = closed, temp = 99\}, \{open, is\_temp\_ok\}) & \xrightarrow{t_7} \\ (\{S'_1, S_2, S_3\}, \{door = closed, temp = 99\}, \{open, is\_temp\_ok, ok\}) & \xrightarrow{t_2} \\ (\{S_1, S_2, S_3\}, \{door = closed, temp = 99\}, \{open, is\_temp\_ok, unlock\}) & \xrightarrow{t_9} \\ (\{S_1, S_2, S'_3\}, \{door = closed, temp = 99\}, \{open, is\_temp\_ok, danger\}). & \end{array}$$

If we assign an explicit priority to the transitions of the model such that t_9 *has a higher priority than* t_7*, then the above undesired big-step is not possible (because at snapshot* $(\{\mathrm{S}'_1, \mathrm{S}'_2, \mathrm{S}_3\}, \{\mathrm{door} = \mathrm{closed}, \mathrm{temp} = 99\}, \{\mathrm{open}, \mathrm{is\_temp\_ok}\})$*, where both* t_9 *and* t_7 *can be taken,* t_9 *takes precedence in execution, disabling* t_7 *for the rest of the big-step). This behaviour is the same as assuming the* STRONG SYNCHRONOUS *external event communication semantics, which only allows the following big-step to be taken:*



S_0
C_1
S_{01}
S_{02}
t_1 : open [door = closed] ⌒check_temp
t_2 : ok⌒unlock /door := open;
t_3 : heat
t_4 : reset
t_5 : close⌒lock /door := closed;
t_6 : check_temp ⌒is_temp_ok
t_7 : ¬danger ⌒ok
t_8 : danger⌒heat
S_1
S_1'
S_1''
S_2
S_2'
C_2
S_{03}
t_9 : is_temp_ok [temp > 40] ⌒danger
t_{10} : reset
S_3
S_3'

Figure 35: Door controller system revisited.

$$\begin{array}{ll}
(\{S_1, S_2, S_3\}, \{door = closed, temp = 99\}, \{open, is\_temp\_ok, danger\}) & \xrightarrow{t_1} \\
(\{S_1', S_2, S_3\}, \{door = closed, temp = 99\}, \{open, is\_temp\_ok, danger, check\_temp\}) & \xrightarrow{t_6} \\
(\{S_1', S_2', S_3\}, \{door = closed, temp = 99\}, \{open, is\_temp\_ok, danger\}) & \xrightarrow{t_9} \\
(\{S_1', S_2', S_3'\}, \{door = closed, temp = 99\}, \{open, is\_temp\_ok, danger\}) & \xrightarrow{t_8} \\
(\{S_1', S_2, S_3'\}, \{door = closed, temp = 99\}, \{open, is\_temp\_ok, danger, heat\}) & \xrightarrow{t_3} \\
(\{S_1'', S_2, S_3'\}, \{door = closed, temp = 99\}, \{open, is\_temp\_ok, danger\}). &
\end{array}$$

The Explicit and a Hierarchy priority semantics can be used together in a BSML in the same way that multiple Hierarchy semantics can be used together. An **advantage** of the Explicit and Hierarchy priority semantics is that for two transitions, their relative precedence can be determined syntactically, which is convenient for modellers. (Of course, the relative precedence of two transitions is relevant if, and when, they are enabled together in a snapshot.)

Negation of Triggers: A common way to assign priority between transitions is to use the negation of events/conditions in the event trigger/variable condition of a transitions. For example, if two transitions t_1 and t_2 with event triggers $trig(t_1)$ and $trig(t_2)$, respectively, are enabled, then by replacing $trig(t_2)$ with $trig(t_2) \wedge \neg(trig(t_1))$, in effect, t_1 would have a higher priority than t_2. An **advantage** of this approach is that no extra syntax and semantics need to be considered to assign priority to the transitions of a model. A **disadvantage** of the Negation of Triggers option is that in order for a modeller to assign a priority to a transition, the modeller should know about all of the transitions that should have a lower priority than the transition, and conjunct their triggers with the negation of the trigger of the transition. It is possible to reduce the number of such conjunctions by considering the transitive property of the priority between transitions, but that might lead to different precedences between transitions in different snapshots.

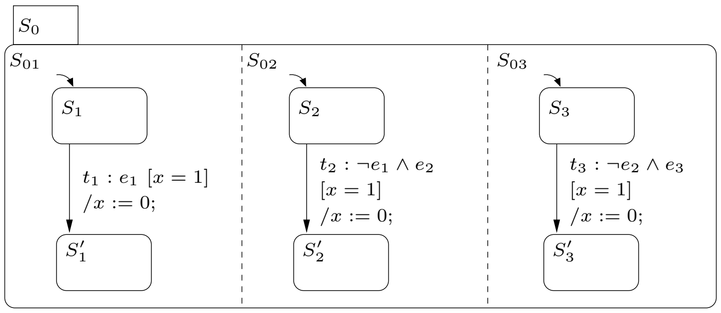


Figure 36: Priority and the negation of events.

Example 27 *Consider the model in Figure 36. If we consider* $(\{\mathrm{S_1, S_2, S_3}\}, \{\mathrm{x = 1}\}, \{\mathrm{e_1, e_2, e_3}\})$*, assume the* Take One *concurrency semantics, and the* Small-step *memory protocol, then* $\mathrm{t_1}$ *has the highest priority, and the only possible big-step is the following big-step:*

$$\begin{array}{l}(\{S_1, S_2, S_3\}, \{x = 1\}, \{e_1, e_2, e_3\}) \xrightarrow{t_1} \\ (\{S'_1, S_2, S_3\}, \{x = 0\}, *).\end{array}$$

Transition $\mathrm{t_3}$ *has a lower priority than* $\mathrm{t_1}$ *because it has a lower priority than* $\mathrm{t_2}$*, and* $\mathrm{t_2}$ *has a lower priority than* $\mathrm{t_1}$*.*

Another **disadvantage** of this option is that the relative precedence of two transitions cannot be determined syntactically, by only inspecting the syntax of the transition. Instead, the precedence between two transitions can be determined with respect to a specific snapshot, which can be a difficult task for a modeller/reviewer who would like to be able to compare the precedence of two transitions in isolation, regardless of which snapshot the model resides in. For example, in Example 27, the relative precedence of transitions t_1 and t_3 cannot be determined by inspecting their syntax, but if, snapshot $(\{S_1, S_2, S_3\}, *, \{e_1, e_2, e_3\})$ is considered, then t_1 has a higher priority than t_3. If snapshot $(\{S_1, S_2, S_3\}, *, \{e_1, e_3\})$ is considered, then neither of t_1 or t_3 has a higher precedence than the other. If t_3 had $\neg e_1$ in the conjunction of its event trigger, then t_1 would always have a higher precedence than t_3, regardless of which snapshot their priorities are compared.

The Negation of Triggers method can be used along with the other priority specification methods. Since the Negation of Triggers option does not require any special semantics, it always overrides the other priority semantics by disabling a transition that has a lower priority.

4 Conclusion and Future Work

We deconstructed the semantics of big-step modelling languages into mainly orthogonal semantic aspects, along with their corresponding semantic options. We also analyzed the advantages and disadvantages of each semantic option, when compared to the other options. Our semantic framework can serve as a part of a methodology for requirements engineers to choose a BSML for modelling a SUS. Of course there are other parameters, such as syntax, socioeconomic factors etc., to be considered too, but we believe that the semantics is one of the most complex criteria in the choice of a BSML. We strived to design our framework to be intuitive and accessible for non-semanticist audience: (i) we chose our semantic aspects at a high-level of abstraction (already familiar for modellers), and (ii) we managed to describe complicated semantic concepts without using formalism. Once the semantic options of a desired BSML is chosen by a requirements engineer, then it is possible to either find an existing BSML that matches the choices, or create a new BSML (e.g., by using our semantic framework in [48]).

In our future work, we plan to extend our framework to BSMLs that support the asynchronous communication with environment. We are also interested in including the semantics of "executable control states" (described in Section 3.2) in our framework.

References

[1] Specification and description language (SDL). ITU-T Recommendation Z.100, International Telecommunications Union, November 1999.

[2] The Esterel v7 reference manual version v7.30, initial IEEE standardization proposal. 2005.

[3] Rajeev Alur and Thomas A. Henzinger. Reactive modules. *Formal Methods in System Design*, 15(1):7–48, 1999.

[4] Albert Benveniste, Paul Le Guernic, and Christian Jacquemot. Synchronous programming with events and relations: The SIGNAL language and its semantics. *Science of Computer and Programming*, 16:103–149, 1991.

[5] J. Bergstra, A. Ponse, and S. M. Smolka, editors. *The Handbook of Process Algebra.* Elsevier, 2001.

[6] Gérard Berry. A hardware implementation of pure ESTEREL. *Sadhana, Academy Proceedings in Engineering Sciences, Indian Academy of Sciences*, 17(1):95–130, 1992.

[7] Gérard Berry. Preemption in concurrent systems. In Rudrapatna K. Shyamasundar, editor, *Proceedings of Foundations of Software Technology and Theoretical Computer Science*, volume 761 of *LNCS*, pages 72–93, Berlin, Germany, December 1993. Springer.

[8] Gérard Berry. Hardware and software synthesis, optimization, and verification from esterel programs. In Ed Brinksma, editor, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1217 of *Lecture Notes in Computer Science*, pages 1–3. Springer-Verlag, 1997.

[9] Gérard Berry and Georges Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[10] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide.* Addison Wesley, Reading, Massachusetts, 2nd edition, May 2005.

[11] Frederic Boussinot. Sugarcubes implementation of causality. Technical Report RR-3487, Inria, Institut National de Recherche en Informatique et en Automatique, 1998.

[12] Daniel M. Chapiro. *Globally-Asynchronous Locally-Synchronous Systems.* PhD thesis, Stanford University, October 1984.

[13] Rance Cleaveland and Matthew Hennessy. Priorities in process algebras. *Information and Computation*, 87(1/2):58–77, 1990.

[14] James Dabney and Thomas L. Harman. *Mastering Simulink.* Pearson Prentice Hall, 2004.

[15] Nancy Day. A model checker for Statecharts: Linking CASE tools with formal methods. Master's thesis, University of British Columbia, 1993.

[16] Luca de Alfaro and Thomas A. Henzinger. Interface Automata. In Volker Gruhn, editor, *Proceedings of the Joint 8th European Software Engeneering Conference and 9th ACM SIGSOFT Symposium on the Foundation of Software Engeneering (ESEC/FSE-01)*, volume 26, 5 of *Software Engineering Notes*, pages 109–120. ACM Press, 2001.

[17] Luca de Alfaro and Thomas A. Henzinger. Interface Theories for Component-Based Design. In *Proceedings of the First International Workshop on Embedded Software*, volume 2211, pages 148–165. Lecture Notes in Computer Science 2211, Springer-Verlag, 2001.

[18] Stephen A. Edwards. Compiling esterel into sequential code. In *Proceedings of the 37th Conference on Design Automation (DAC-00)*, pages 322–327, NY, 2000. ACM/IEEE.

[19] Colin Fidge. A comparative introduction to CSP, CCS and LOTOS. Technical report, The university of Queensland, Department of Computer Science, 1994.

[20] Nicolas Halbwachs. *Synchronous Programming of Reactive Systems.* 1993.

[21] Nicolas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The synchronous data flow programming language Lustre. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[22] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[23] David Harel and Hillel Kugler. The RHAPSODY Semantics of Statecharts (or, On the Executable Core of the UML). In *Integration of Software Specification Techniques for Application in Engineering*, volume 3147 of *Lecture Notes in Computer Science*, pages 325–354. Springer-Verlag, 2004.

[24] David Harel and Amnon Naamad. The STATEMATE semantics of Statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, 1996.

[25] David Harel and Amir Pnueli. On the development of reactive systems. In K. R. Apt, editor, *Logics and Models of Concurrent Systems.* Springer Verlag, 1985.

[26] David Harel, Amir Pnueli, J. P. Schmidt, and R. Sherman. On the formal semantics of statecharts. In *Proceedings of the Second IEEE Symposium on Logic in Computation*, pages 54–64. IEEE, 1987.

[27] K. L. Heninger, J. Kallander, David Lorge Parnas, and J. E. Shore. Software requirements for the A-7E aircraft. NRL Memorandum Report 3876, United States Naval Research Laboratory, Washington DC, November 1978.

[28] M. Herlihy and J. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3), July 1990.

[29] Maurice P. Herlihy and Jeannette M. Wing. Axioms for concurrent objects. In *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages*, pages 13–26, Munich, West Germany, January 21–23, 1987. ACM SIGACT-SIGPLAN, ACM Press.

[30] Tony Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.

[31] Tony Hoare and He Jifeng. *Unifying Theories of Programming.* International Series in Computer Science. Prentice Hall, 1998.

[32] Cornelis Huizing and Rob Gerth. Semantics of reactive systems in abstract time. In *Proceedings of the Real-Time: Theory in Practice, REX Workshop*, pages 291–314, London, UK, 1992. Springer-Verlag.

[33] ISO. LOTOS: a formal description technique based on the temporal ordering of observational behaviour. Technical Report 8807, International Standards Organisation, 1989.

[34] Michael Jackson and Pamela Zave. Deriving specifications from requirements: an example. In *Proceedings of the Seventeenth International Conference on Software Engineering*, pages 15–24. ACM, 1995.

[35] Ryszard Janicki and Maciej Koutny. Structure of concurrency. *Theoretical Computer Science*, 112(1):5–52, 26 April 1993.

[36] Leslie Lamport. On interprocess communication, Parts I and II. *Distributed Computing*, 1(2):77–101, 1986.

[37] Leslie Lamport and Fred B. Schneider. Pretending atomicity. Technical Report 44, Digital Equipment Corporation, Systems Research Center, May 1989.

[38] Nancy G. Leveson, Mats Per Erik Heimdahl, Holly Hildreth, and Jon Damon Reese. Requirements specification for process-control systems. *IEEE Transaction on Software Engineering*, 20(9):684–707, September 1994.

[39] Robert J. Lipton. Reduction: A method of proving properties of parallel programs. *Communications of the ACM*, 18:717–721, 1975.

[40] Andrea Maggiolo-Schettini, Adriano Peron, and Simone Tini. Equivalences of statecharts. In *International Conference on Concurrency Theory*, pages 687–702, 1996.

[41] Florence Maraninchi and Yann Rémond. Argos: an automaton-based synchronous language. *Computer Languages*, 27(1/3):61–92, 2001.

[42] George H. Mealy. A method for synthesizing sequential circuits. *Bell System Technical Journal*, 34(5):1045–1079, 1955.

[43] Vijay Menon, Steven Balensiefer, Tatiana Shpeisman, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Bratin Saha, and Adam Welc. Practical weak-atomicity semantics for java stm. In Friedhelm Meyer auf der Heide and Nir Shavit, editors, *SPAA 2008: Proceedings of the 20th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 314–325. ACM, 2008.

[44] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and how to develop domain-specific languages. *ACM Computing Surveys*, 37(4):316–344, December 2005.

[45] Robin Milner. Calculi for synchrony and asynchrony. *Theoretical Computer Science*, 25(3):267–310, July 1983.

[46] Robin Milner. *Communication and Concurrency.* International Series in Computer Science. Prentice Hall, 1989.

[47] Jianwei Niu. *Template Semantics: A Parameterized Approach to Semantics-Based Model Compilation.* PhD thesis, University of Waterloo, 2005.

[48] Jianwei Niu, Joanne M. Atlee, and Nancy A. Day. Template semantics for model-based notations. *IEEE Transaction on Software Engineering*, 29(10):866–882, 2003.

[49] Jianwei Niu, Joanne M. Atlee, and Nancy A. Day. Understanding and comparing model-based specification notations. In *International Requirements Engineering Conference*, pages 188–199. IEEE Computer Society, 2003.

[50] OMG. OMG Unified Modeling Language (OMG UML), Superstructure, v2.1.2. 2007. Formal/2007-11-01.

[51] Christos H. Papadimitriou. *The Theory of Database Concurrency Control.* Computer Science Press, 1986.

[52] David L. Parnas and Jan Madey. Functional documents for computer systems. *Science of Computer Programming*, 25(1):19–23, 1995.

[53] Benjamin C. Pierce. *Types and Programming Languages.* MIT Press, 2002.

[54] Amir Pnueli and M. Shalev. What is in a step? In *J.W. De Bakker, Liber Amicorum*, pages 373–400. CWI, 1989.

[55] Amir Pnueli and M. Shalev. What is in a step: On the semantics of statecharts. In *Theoretical Aspects of Computer Software*, pages 244–264, 1991.

[56] Vaughan R. Pratt. Modelling concurrency with partial orders. *International Journal of Parallel Programming*, 15(1):33–71, 1985.

[57] Vladimiro Sassone, Mogens Nielsen, and Glynn Winskel. Models for concurrency: Towards a classification. *Theoretical Computer Science*, 170(1–2):297–348, 15 December 1996.

[58] S. A. Seshia, R. K. Shyamasundar, A. K. Bhattacharjee, and S. D. Dhodapkar. A translation of Statecharts to Esterel. In Jeanette M. Wing, Jim Woodcock, and Jim Davies, editors, *FM'99—Formal Methods, Volume II*, volume 1709 of *Lecture Notes in Computer Science*, pages 983–1007. Springer, 1999.

[59] Sandeep K. Shukla and Michael Theobald. Special issue on formal methods for globally asynchronous and locally synchronous (GALS) systems. *Formal Methods in System Design*, 28(2):91–92, 2006.

[60] Signe J. Silver and Janusz A. Brzozowski. True concurrency in models of asynchronous circuit behavior. *Formal Methods in System Design*, 22(3):183–203, 2003.

[61] Robert C. Steinke and Gary J. Nutt. A unified theory of shared memory consistency. *Journal of ACM*, 51(5):800–849, 2004.

[62] Olivier Tardieu. A deterministic logical semantics for pure Esterel. *ACM Transactions on Programming Languages and Systems*, 29(2):8:1–8:26, April 2007.

[63] Robert van Glabbeek. The linear time – branching time spectrum. In J. C. M. Baeten and J. W. Klop, editors, *Proceedings of CONCUR'90*, LNCS 458, pages 278–297. Springer-Verlag, 1990.

[64] Michael von der Beeck. A comparison of statecharts variants. In Hans Langmaack, Willem P. de Roever, and Jan Vytopil, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems, Third International Symposium Organized Jointly with the Working Group Provably Correct Systems - ProCoS, Lübeck, Germany, September 19-23, Proceedings*, volume 863 of *LNCS*, pages 128–148. Springer, 1994.

[65] Pamela Zave and Michael Jackson. Four dark corners of requirements engineering. *ACM Transactions on Software Engineering and Methodology*, 6(1):1–30, January 1997.