Title

Joeri Exelmans[a]

[a] *University of Antwerp, Middelheimlaan 1, 2020 Antwerpen, Belgium*

Abstract

Keywords: Statecharts, SCXML, Class Diagrams, UML, Model-Driven Engineering, Actor Model, Big-Step Modeling Languages

---

1. Introduction: UML and Object-Oriented Programming Languages

Object-oriented programming (OOP) has been widely adopted in industry because of its better abstraction and reuse capabilities over the previously used procedural paradigm.

The Unified Modeling Language (UML) is a very popular set of tools for system design. It incorporates a number of high-level to very high-level modeling languages. In the context of OOP, the following are of special interest:

Class Diagrams model the structure of objects (classes, prototypes) and/or relationships between them. For association relationships, multiplicities can be set on both ends. For each class of objects, a range on the global number of objects that can dynamically occur at any given time can also be specified.

Statecharts (also called State Diagrams in UML) model the behavior of objects.

There are 2 problems in the current situation of the way these modeling languages are typically being used.

The first problem is, that, although UML is widely used for system design, during implementation, the created UML models are often used for reference only. This is sad, because UML (especially the subset class diagrams + statecharts) can provide very complete descriptions of a working (or not working) system. Class diagrams map to class declarations in OO code intuitively, and statecharts can also be transformed to code. Compilers doing this already exist.

Second, most popular OOP languages fail to provide some of the features provided by the UML modeling languages. When converting UML models to code (either by hand or with a compiler), some of the information contained in the models is therefore neglected.

Example 1: In OO, an object can be seen as an instance of a statechart. Statecharts explicitely express the allowed operations on an object at a particular moment in execution depending on its state at that moment. Because

the state of an object in a language like C++ or Java is implicit (the cartesian product of the values of its fields) the programmer has to check a large amount (if not all) of values to adhere to certain conditions. Design by Contract is such an intensive means of forcing a higher-level state in OO programs[1]. Needless to say this is highly time consuming and therefore often only partially applied (e.g. in critical parts) or skipped altogether. Mistakes can also easily be made (e.g. when forgetting to check for certain conditions).

Example 2: A modeler uses class diagrams to specify a range on the number of objects of a class that can dynamically occur[2]. Again, this feature is not built into OOP languages used in industry, and therefore, responsibility is passed on to the implementation, where it is seen as mere non-critical information, and forgotten about.

2. Our Approach

We propose a new modeling language integrating both structure + relationships (as modeled by class diagrams) and behavior (statecharts). The intention is to exploit the full capabilities of these languages, possibly extending them further, and to implement entire systems (design and implementation) in it right away. We will develop a compiler converting models in the proposed language to a range of existing OO languages, with the compiled models still adhering to all of the constraints that were originally specified. If a target language does not have the capability to natively enforce some of those constraints (e.g. it is impossible to impose a lower and upper limit on the number of instances of a given class in Java), the compiled models are linked against an additional runtime kernel, also written in the target language, checking those (dynamic) constraints.

For a target language X, we thus provide:

1. A compiler transforming the models to native X code[3], while checking static constraints.
2. A runtime kernel, written in X and used by the compiled models, providing some of the source model language "magic"[4] not included in X, while checking dynamic constraints[5].

---

[1]Some languages (e.g. Eiffel) have design by contract built in, others require 3rd party extensions.

[2]On top of being an extra constraint, this information can also be used to optimize static allocation (e.g. in C++, not possible in Java), when the range is a fixed number.

[3]We first convert models to an intermediate - target-independent - representation. This will obviously be the largest portion of the effort.

[4]e.g. Broadcasting of messages, environment interaction, dynamic creation/deletion of objects

[5]e.g. Maximum number of objects of a certain class

3. Proposed Formalism

It is best to view models in our formalism as a collection of dynamically created objects, with asynchronous events as the only communication mechanism between objects. A list of properties:

- Each object has associated with it 1 class diagram and 1 statechart.
  - The class diagram defines the *private data fields* and *private methods* of the object.
    - Each method can only operate on data field values of the same object (by possibly calling other of its own methods).
    - An object's methods are *not* a public interface towards other objects. An object can not have a reference/pointer to another object as a field value, it cannot invoke an other object's methods.
  - An object's statechart specifies how the object responds to external input by receiving events.
    - Events are received on predefined input ports.
    - Upon receiving an event, depending on the current state and an additional boolean guard expression over its field values, the statechart can make a transition.
    - As a side effect of such a transition, an object can invoke some of its own methods (modifying data field values) and new events can be generated.
    - Generated events can be sent to certain output ports. Output ports are connected to input ports of other objects.
    - Timed events are also possible, causing a transition to become enabled after the model has been in a certain state for a given period of time.

A number of things are still unclear. We will investigate whether any of the following semantics are preferable, and in which cases we give control over them to the modeler.

- Are certain/all generated events locally broadcasted and received by other transitions in the same statechart? At what point, and until when, are such events visible by other parts of the statechart?

- How many transitions do we allow an object's statechart to take as a response to external input?

- Do we allow transitions to trigger solely because an expression over its data field values (guard condition) becomes true, without the need of an event?

- When we assign a new value to a variable, at what point can this cause a transition to become enabled/disabled?
- Is it possible to modify data field values directly from the statechart or only through "setter" methods?
- How do we define connections between output and input ports of (possibly dynamically created) objects?
- Because all communication between objects happens through events, is it necessary to introduce "parametric events", i.e. events that carry actual parameters (like with a function call)?

4. Big-Step Modeling Languages

To better understand the possible syntactic and semantic options for our proposed formalism, we will reason about it as a so-called Big-Step Modeling Language (BSML). As described in [2], a BSML is any language that responds to external input from the *environment*, by executing (possibly concurrently) multiple *transitions*. The authors compare a wide range of modeling languages as cases of BSMLs. To do so, they identified 8 orthogonal semantic aspects in which BSMLs can differ. Some of these aspects are optional, others are required.

First we will introduce the most important concepts of a BSML, while explaining how elements of our formalism map onto these. Next, we shall describe the 8 orthogonal aspects. Finally, we set out 4 different "schemes" of options for the 8 orthogonal aspects that we think are useful.

4.1. Concepts

4.1.1. Big step

The execution of a BSML model is a sequence of big steps. A big step is a possibility for a model to sense input information from the environment. Input sensed from the environment cannot change during a big step.

In our formalism, a big step begins when an external event is received by an object's statechart. Timed events ("after 5 seconds in state A, do this") and interface events are equally considered external.

4.1.2. Events and Variables

A model can have a set of *events* that are active or inactive at any point in execution. A model also can have a set of variables to which values can be assigned, and from which values can be read at certain points in execution.

It is clear that events and variables in our formalism correspond to the same concepts in a BSML.

4.1.3. Communication

Both events and variables can be used by BSML models to communicate with the environment and other compoments of the model (see 4.1.4). Input information from the environment comes in the form of activated events and values assigned to variables. Some BSMLs will distinguish explicitly between events that can only be activated by the environment (so-called *input events*), and likewise with variables (*input variables*). Some BSMLs will also explicitly specify certain events to be *output events*, i.e. events that are generated by the model and sensed by the environment (not by the model itself). Likewise with *output variables*.

In our formalism, we will not use input/output variables as a way of communication with the environment or between objects. All communication will be set out through events. Because of explicit input/output ports in the statecharts of objects, we syntactically differentiate internal events from input/output events.

4.1.4. Composition

Models of a BSML may be composed. Communication between components is set out through the same means of communication between a component and the environment, i.e. through events and variables. A model may explicitly differentiate between input coming from the environment and from other components. As such, separate properties for so-called *interface events* and *interface variables* can be defined.

Every object in our formalism corresponds to a component in a BSML. Inter-component communication also happens through predefined input/output ports. From the point of a component, inter-component communication is syntactically equal to communication with the environment.

4.1.5. Control States

A model (component) consists of a hierarchical tree of control states. A control state can be seen as a noteable point in execution of a model. During execution, a model has a set of *current states*, i.e. states it "is in". There are 3 kinds of control states: *basic states* (these occupy leaf nodes in the tree of control states), *and-states* (non-leaf nodes) and *or-states* (root node and other non-leaf nodes). If a model is in a certain state, it is also in all of the ancestors of that state. If a model is in an and-state, it is in all of the children of that state. If a model is in an or-state, it is in 1 of the children of that state, and among those children, 1 *default state* must always be syntactically defined, i.e. a state it is in initially when the or-state becomes a current state. 2 states *overlap* if one is an ancestor of the other.

In our formalism, control states correspond to the states in an object's statechart: The hierarchical structure of statecharts can be mapped on a hierarchical tree of control states in a BSML, see Table 1.

4.1.6. Transitions

A model also consists of *transitions*. At any point in execution, a transition can be enabled or disabled. The following properties of determine whether a

Statechart element	BSML control state
Normal state	Basic state
Superstate	And-state
Orthogonal region	Or-state

Table 1: Statechart elements and their corresponding BSML control states.

transition is enabled:

Source Control State (Required) If this state is not a current state, the transition is disabled.

Event trigger (Optional) A boolean expression over a number of events to be active or inactive for the transition not to be disabled.

Guard condition (Optional) A boolean expression over the values of a number of variables. If the expression evaluates to false, the transition is disabled.

If a transition is enabled, it can *trigger*. The following properties determine what happens as a result of being triggered:

Destination Control State (Required) State to become a current state. Source state is no longer a current state (unless destination is source).

Generated Events (Optional) Events to become active, possibly changing the enabledness of other transitions.

Variable Assignments (Optional) New values to be assigned to a number of variables, possibly changing the enabledness of other transitions.

The *arena* of a transition is the lowest or-state ancestor of its source and destination control states.

Intuitively, transitions in an object's statechart map onto transitions in a BSML. Note that variable assignments are set out through local method calls. States in our formalism can likely also be assigned 'enter-' and 'exit-actions'. Enter-actions of a state are added to the trigger actions (generated events, variable assignments) of transitions with the state as its source, exit-actions are added to the trigger actions of transitions with the state as its target.

4.1.7. Small step, Combo Step

A big step consists of zero or more *small steps*. A small step consists of 1 or more *transitions* (depending on *concurrency semantics*, see 4.2.2). In this way, some BSMLs will always respond to environmental input by making only a single transition, while others will make many transitions.

Some BSMLs use the notion of *combo steps* to introduce yet another level of 'responding to input'. When using combo step semantics, a big step consists of a number of combo steps, and a combo step consists of a number of small steps.

To reason about our formalism as a BSML, we will have to specify how many small steps it takes to conclude a big step, how many transitions there are in a small step, and whether we will group small steps into combo steps.

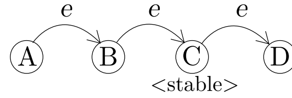

Figure 1: Big-Step Maximality: Suppose event e is present throughout the whole big step. State C is syntactically marked as *stable*. In the beginning of the big step, we are in state A. When using TAKE ONE semantics, we end in state B. When using TAKE MANY, we end in state D. When using SYNTACTIC, we end in state C.

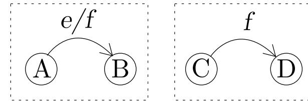

Figure 2: Concurrency: Suppose we are at the beginning of a big step, event e is present, and A and C are active states. The dashed rectangles indicate orthogonal regions, i.e. or-states (containing basic states A,B and C,D) that are children of a common and-state. With the MANY concurrency semantics, transitions caused by events e and f could be executed in the same small step (given that we use the correct event lifeline semantics, see 4.2.3), i.e. $\{\{e, f\}\}$. When using SINGLE, each transition has to take place in a separate small step, e.g. $\{\{e\}, \{f\}\}$ could be a valid big step.

4.2. 8 Aspects

We already mentioned some semantic options a BSML can specify. We will now explain them in further detail, fully covering the 8 orthogonal semantic aspects mentioned in [2].

4.2.1. Big-Step Maximality

The only required aspect. It specifies when a big step ends. Options are SYNTACTIC (certain states are determined 'stable' by the modeler, such states can entered only once per non-overlapping arena), TAKE ONE (in a big step, only 1 transition can be made per non-overlapping arena) and TAKE MANY (big step ends when there are no more enabled transitions). See Figure 1.

The SYNTACTIC option does not appear to be very attractive, because it requires extra work on part of the modeler, and it does not make models much easier to comprehend. TAKE MANY has the drawback of possibly never-ending big steps. TAKE ONE is the simplest option, and it is also used in Harel statecharts. We think both TAKE ONE and TAKE MANY should be modeler configurable options.

4.2.2. Concurrency

Specifies whether more than one transition can occur in a small step. Options are SINGLE (1 transition per small step) and MANY (1 or more transitions per small step). When using MANY, additional sub-options for *consistency* and *preemption have to be specified*. See Figure 2.

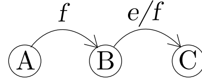

Figure 3: Suppose event e is present and we are in state A. The big step $\{\{f\},\{e\}\}$ could only occur when using PRESENT IN WHOLE semantics. Although PRESENT IN SAME also lacks causality, transitions e and f are never allowed to take place in the same small step, no matter the concurrency semantics.

4.2.3. Event Lifeline

Specifies at what point a generated event is sensed as active by the model, and until when.

The PRESENT IN WHOLE option makes a generated event active throughout the entire big step in which it was generated. It is intended for synchronous hardware specification languages. It lacks causality (an event generated at a later point in time can cause transition at an earlier point in time, see Figure 3) and is harder to implement.

The PRESENT IN REMAINDER option makes an event active after it is generated, until the end of the big step. Its behavior is causal and easy to implement.

2 other options, PRESENT IN NEXT SMALL STEP and PRESENT IN NEXT COMBO STEP are also causal, and force orderedness.

PRESENT IN SAME makes events active only in the same small step as the one in which they were generated. This only makes sense in the context of the MANY concurrency option.

It is also possible to specify these options separately for environmental or interface events. To do so, it is necessary to specify additional options telling which events are considered environmental, and which are considered interface events.

4.2.4. Combo-Step Maximality

Combo steps are typically used in BSMLs such that the lifeline of events and the memory protocols (see 4.2.5 and 4.2.6 can be set to NEXT COMBO STEP. This forces (partial) ordering on transitions (e.g. the events generated in combo step A can't be sensed until the next combo step B). It is clearly only useful in situations where BSMLs repond to environmental input by taking many transitions (i.e. big step maximality: TAKE MANY).

The authors in [2] are only aware of languages that use the COMBO TAKE ONE option.

4.2.5. Enabledness Memory Protocol

This aspect specifies what values are read from variables when evaluating the guard condition of a transition. Different options can be set for internal variables and interface variables. We ignore interface variables because our formalism will not use them.

For internal variables, options are GC BIG STEP, GC COMBO STEP and GC SMALL STEP, meaning the guard conditions are evaluated at the beginning of a big step, a combo step or a small step, respectively.

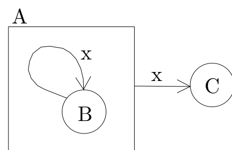

Figure 4: Hierarchical priority. If we used source-parent semantics, the transition from A to C would have higher priority. If we used source-child semantics, the transition from B to B would be taken.

4.2.6. Assignment Memory Protocol

This aspect specifies what values are read from variables when evaluating the right-hand side of an assignment expression in a transition. The same options are available as with *Enabledness Memory Protocol*, and languages typically use the same options for both aspects.

4.2.7. Order of Small Steps

It is possible to impose ordering on the sequence of small steps. Options are None (no ordering), Explicit (syntactic ordering) and Dataflow (try to make assignment happen before read).

4.2.8. Priority

If one of multiple different (sets of) transitions can be taken in a small step, it is possible to prefer one set over another. A very interesting option is Hierarchical (see Figure 4). Different statechart variants use different variations of hierarchical priority[6]

Another option is Explicit ordering.

4.3. Option schemes for our formalism

We will now reason about what aspects make sense in the context of our formalism: Some aspects will be abscent (i.e. we don't need them), some will be present and fixed (i.e. only one choice makes sense), while others will be present and configurable by the modeler (i.e. a multitude of choices is be useful or it is still unclear at this moment which option to choose).

Because the choices made for certain aspects often depend on the choices for other aspects, we present a number of different schemes of options that we see as useful.

Note that there is tremendous overlap in the resulting semantics. This is because the system of 8 orthogonal aspects is very open.

[6] In Statemate[5], if the arena of transition a is the parent of the arena of a transition b, a has higher priority. In SCXML[8] and Rhapsody[4], if the source of transition a is a child of the source of a transition b, a has higher priority (this way, lower-level transitions can override higher-level behavior).

	"Combo"	"Simple_One"	"Simple_Many"	"Concurrent"
Big Step Maximality	Take Many	Take One	Take Many	Take One
Combo Step Maximality	Take One	-	-	-
Concurrency	Single	Single	Single	Many, Arena Orthogonal
Event Lifeline	Present in Next Combo Step	Present in Remainder	Present in Remainder	Present in Same
External Events	Syntactic	Syntactic	Syntactic	Syntactic
External Event Lifeline	First Combo Step	Present in Whole	Present in Whole	First Small Step
Enabledness Memory Protocol	GC Combo Step	GC Small Step	GC Small Step	GC Small Step
Assignment Memory Protocol	RHS Combo Step	RHS Small Step	RHS Small Step	RHS Small Step
Order of Small Steps	-	-	-	-
Priority	Hierarchical	Hierarchical	Hierarchical	Hierarchical

Table 2: 3 different schemes for our formalism

4.3.1. Common patterns

As you can see in Table 2, in all schemes, we have chosen the same options for some aspects.

For *priority* (4.2.8), we chose HIERARCHICAL because many variants of statecharts also use it. Whether priority is given to parent or child states will be an option for the modeler (see Figure 4). We do not fancy EXPLICIT because it can make models too complex and it is very intensive for the modeler to specify a priority for each transition.

For *external events* (4.2.3), we chose SYNTACTIC, because for each object, external events pass through predefined input/output ports. For *external event lifeline*, we use the same semantics as for *event lifeline*, but in the table we renamed the options to something that is easier to understand (e.g. FIRST SMALL STEP instead of NEXT SMALL STEP, PRESENT IN WHOLE instead of PRESENT IN REMAINDER).

To keep things simple, we always chose the simlar options for *event lifeline*, *enabledness memory protocol* (4.2.5) as for *assignment memory protocol* (4.2.6).

We don't include the aspect *order of small steps* (4.2.7) because we don't see any useful case for it in our formalism.

4.3.2. "Combo"

This scheme closely resembles the semantics of Statemate as described in [5]. A big step lasts until no remaining transitions can be triggered. Small steps always consist of exactly 1 transition, and a combo step is a grouping of transitions of nonoverlapping arenas (in statecharts called *orthogonal regions*). The presentness of events and values of variables are sensed at the beginning of every combo step, i.e. per "sense" of events and variable values, at most 1 transition can be made per orthogonal region in the model, so if multiple transitions occur in a combo step, those transitions have no influence on each other. Because the results of one combo step become visible only in the next combo step, successive transitions have a causal, ordered relationship.

4.3.3. "Simple_One"

This scheme resembles the semantics of the original Harel Statecharts [3]. In 1 big step, it allows 1 transition to be taken per orthogonal region. This has the benefit of guaranteed ending big steps: An infinite (blocking) loop is not possible. Generated events become present immediately, until the end of the big step, so communication between orthogonal regions is causal, but not necessarily ordered. The most actual variable values are read when taking any transition. This scheme is probably the easiest to implement, while also being not too complex for the modeler.

4.3.4. "Simple_Many"

This scheme is like "Simple_One", but it allows an unlimited amount of transitions to take place, every big step. A big step ends when there are no more enabled transitions. This has the benefit of sequential transitions in the

same or-state, but it comes at the cost of greater complexity, and we no longer have guaranteed ending big steps. An error in the model can cause it to hang.

4.3.5. "Concurrent"

This scheme is special in that it is the only one providing concurrency: A small step can consist of multiple transitions, as long as those transitions take place in non-overlapping arenas. Generated events are active throughout the whole small step (i.e. the big step, because a big step in this scheme can at most consist of 1 small step), making communication between orthogonal regions synchronous, non-causal and non-ordered. Even though it gets rid of the problem of *multiple-instance events* (because an event is either present or not present throughout the whole big step), synchronous events can be harder for the modeler to work with, and they are certainly more difficult to implement.

We don't have to provide *preemption* semantics because preemptive transitions are impossible in statecharts.

4.3.6. Conclusion

Because both "Combo" and "Simple_One" are easy to understand for the modeler, and easy to implement (for us), we will likely support similar options in our compiler.

5. The Actor Model

The actor model [7] [1] is a minimalistic view on what is needed to achieve concurrent, fault-tolerant and highly scalable computation[6].

The basic building blocks are *actors* (entities of computation, "everything is an actor"). Each actor has an *address* ("capability"), on which it can receive *messages* ("photons of communication"). Messages travel over a (possibly imaginary) best-effort network, i.e. a sent message arrives once, at any time after it has been sent, or it may not arrive at all. Queues are not assumed to be present, but may be used by implementations (without being visible to the modeler). Upon receiving 2 messages at the exact same moment, one of them is undeterministically received first.[7] Upon receiving a message, an actor may:

1. Create new actors
2. Send *asynchronous* messages to actor addresses it knows. An actor can know another actor's address because:
   (a) It knew the address *from the beginning*
   (b) It created the other actor
   (c) It received the other actor's address in a message

[7] This can be achieved in hardware by a so-called 'arbiter' circuit. An arbiter has 2 binary inputs and 2 corresponding binary outputs. When one of its inputs becomes true, its corresponding output becomes true. When both inputs become true at the exact same moment, only one of its outputs *eventually* becomes true. The amount of time it takes to 'decide' decreases exponentially, so usually a decision is taken very rapidly.

3. Decide how to respond to a next message (by performing computation, storing addresses, etc.)

The inherently asynchronous nature and absense of a shared state lends itself for highly parallel or distributed implementation. With the "best-effort network" resembling our modern day internet, an implementation could easily scale computation accross the globe. This caused a recent spike of interest in the actor model.

Erlang[8] has actors built in, and is probably the most popular incarnation of these ideas.

The actor model is quite elegant, in that "normal", synchronous computation is a special case of the actor model. Functional programming nicely maps onto the actor model (because all operations are local), but Carl Hewitt claims the actor model is capable at computing nondeterministic functions on integers that are not possible in functional programming[6].

5.1. Our formalism as a case of the actor model

We can view our formalism as a case of the actor model: Objects resemble actors, because they can receive events (messages) on their input ports (addresses). Upon receiving an event, an object will be able to dynamically create new objects (we don't know exactly how this will happen at this point), generate events at certain output ports, connected to input ports of other objects (addresses of other actors), and of course decide on how to respond to a next event by making transition(s) (changing state).

To match the actor model more closely, we would also have to support sending input port "addresses" to other input ports through some sort of parameterized events (just like addresses can be sent around in the actor model).

6. SCXML

Because a large portion of the syntax and semantics of our formalism will concern statecharts, it makes sense to use a standardized document format for compatibility with existing and future tools. We choose the State Chart eXtensible Markup Language (SCXML)[8], an XML-based metamodel of statechart-like behavior. The document tree represents a hierarchical tree of states and transitions. References between transitions and states in the document are set out through XPath.

To integrate SCXML-formatted statechart behavior into our framework, we will also use XML to describe the rest (class diagrams, input/output ports) of our models. There will also be some extensions to the SCXML schema.

TODO: complete description, XML schema?

[8]Developed by Ericsson for use in telecommunication industry. It differs from other actor model implementations such as Scala because it sticks closely to the basics (e.g. synchronous communication is *not* built in).

References

[1] Gul Abdulnabi Agha. Actors: a model of concurrent computation in distributed systems. 1985.

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

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

[4] 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 Applications in Engineering*, pages 325–354. Springer, 2004.

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

[6] Carl Hewitt. What is computation? actor model vs. turing's model. `http://what-is-computation.carlhewitt.info/`, November 2013.

[7] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, IJCAI'73, pages 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc.

[8] W3C. Scxml: State machine notation for control abstraction (working draft). `http://www.w3.org/TR/scxml/`, August 2013.