Configurable Semantics in the SCCD Statechart Compiler

Joeri Exelmans

October 2014

1 Introduction

The original semantics of SCCD have been extended with a number of options. The stepping algorithm of objects also has been rewritten.

We will first explain the original stepping algorithm and semantics in Section 2. We will then explain the new stepping algorithm in Section 3, and the available semantic options in Section 4.

2 Original SCCD Semantics

In this section, a rough overview will be given of the stepping algorithm in the original version of the SCCD compiler[5]. Next, a number of drawbacks from using this algorithm will be given.

2.1 Stepping Algorithm

In original SCCD, the `step` method of all objects is called in a round-robin fashion, advancing simulated time. The `step` method has a parameter `delta`, the amount of time to simulate. Each object has an event queue, its entries being tuples $(event, timeout)$. In a `step`, the timeouts of all entries are reduced by `delta`, and due events (an event whose $timeout <= 0$) are popped. For each due event, the `transition` function of the object's statechart is called, with the event as its parameter.

2.1.1 Transition function

The `transition` function is not part of the runtime; it is produced by the compiler for each statechart (class) in the diagram. Depending on the current state of the object, the `transition` function executes 0 or more transitions, changing the object's state. It returns `true` iff at least 1 transition was executed.

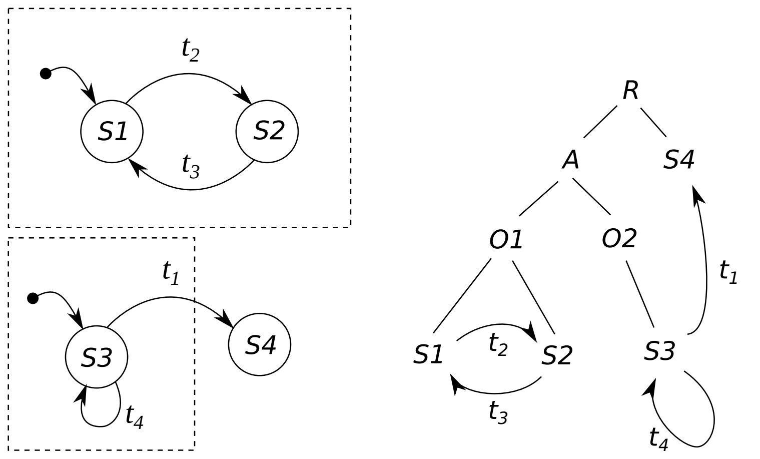

Figure 1: Left: A statechart with 2 orthogonal components. Right: Composition tree of the same statechart.

The `transition` function performs a depth-first search on the object's (hard-coded[1]) tree of current states. At every node, the event triggers and guard conditions of outgoing transitions are checked. Transitions can only be enabled if their event trigger is equal to the actual value of the `event` parameter of the `transition` function, or if they have an empty event trigger (called a *null transition* in Rhapsody). The first outgoing transition that is found to be enabled, is executed.

Whenever an outgoing transition of a node in the tree of current states has been executed, the children of that node are no longer visited in the current pass. This is because the current node is no longer guaranteed to be in the set of current states. This means that original SCCD gives priority to transitions whose source is higher up in the composition tree. As a result, if multiple transitions execute during a call to the `transition` function, those transitions must be orthogonal to each other. Since the `transition` function is called for each due event by the stepping method, each event can be sensed by multiple orthogonal transitions[2].

Example: In Figure 1, suppose the set of current states is $\{R, A, O1, O2, S1, S3\}$, and transitions t_2 and t_4 are enabled. A depth-first search on the current states visits R, A, $O1$, $S1$, $O2$ and finally $S3$. When $S1$ is visited, its outgoing transition t_2 is executed. When $S3$ is visited, t_4 is executed.

[1]For each state S in the composition tree, the compiler produces a function *transition_S*. This function calls the transition functions of the children of S, and returns *true* or *false*, depending on whether a transition with source S or any (in)direct child of S was made.

[2]2 transitions are orthogonal if their *arenas* are not ancestors of each other. The arena of a transition is the lowest common Or-state ancestor of its source and destination. E.g. in Figure 1, the pairs of orthogonal transitions are (t_2, t_4) and (t_3, t_4).

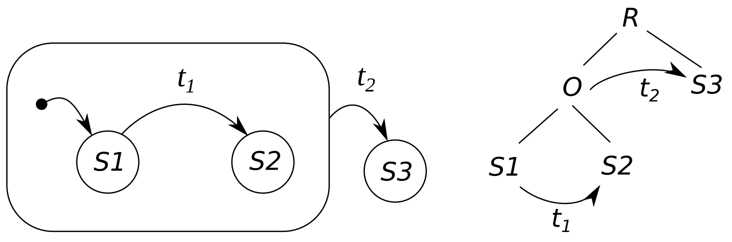

Figure 2: Left: Statechart. Right: Composition tree of the same statechart.

2.1.2 Priority Semantics

By default, original SCCD gives priority to transitions whose source state is closer to the root of the composition tree. It is also possible to configure it to give priority to states lower in the composition tree. To carry out this behavior, when visiting a node, before checking outgoing transitions of the node, its children are visited first. Later, the outgoing transitions of the node are only checked if no transitions were made in the node's children.

Example: Have a look at Figure 2. Suppose transitions t_1 and t_2 are both enabled, and the `transition` function is called. By default, original SCCD will execute t_2, because when scanning the composition tree in a depth-first manner, its source state (O) is encountered first. If we configure SCCD to give priority to inner transitions, the outgoing transitions of the children of O are checked first, before the outgoing transitions of O itself. This causes t_1 to be executed. Note that after t_1 is executed, t_2 will still be executed during a next call to the `transition` function.

2.1.3 End Condition

We have seen that the `step` function decreases time by a given `delta`, pops due events, and calls the `transition` function for each due event. When this is done, the `step` function keeps calling `transition` (with no argument) until it returns `false`, i.e. transitions keep executing until it is no longer possible to make a transition.

2.2 Drawbacks

We have just seen that in original SCCD, **transitions** are executed during the search for enabled transitions. There are a number of drawbacks that originate from this approach:

- In some cases, it is possible for conflicting transitions (i.e. transitions between whom there exists a *disables*-relationship, meaning one transition makes it no longer possible for the other transition to execute) to be executed in an illegal order. This is because if a transition affects the set of current states in such a way that the parent of the current node is no longer in the set of current states, other children of the current node's parent will still be checked as if they reside in the set of current states.

Example: In Figure 1, if $O2$ was visited before $O1$, t_2 or t_3 could still be executed after t_1 was executed, possibly bringing the model in an invalid state. There is no elegant way to implement this behavior correctly, and thus we cannot allow transitions of the type of t_1. In Section 3, we will show that this problem simply does not arise with the new stepping algorithm, so there will be no need to deal with it as a special case.

- It is harder/impossible to implement concurrency semantics. With concurrency, transitions that are orthogonal to each other are executed *concurrently*, meaning the transitions cannot see each other's side effects. The main advantage is that otherwise, orthogonal transitions are able to execute in any order, nondeterministically, but if we execute them concurrently, there no longer is an order. Of course, we have to check whether either transition can have an effect on the other transition through its actions/guard conditions (this is very complex), and even then there are additional semantic options for defining behavior. More on this in Section 3.

- The old stepping algorithm has no notion of the concepts (or equivalent concepts) *big step* / *combo step* / *small step* from Day & Atlee's 2010 paper [2]. More on this in Section 3.2.

- The main goal of configurable semantics was to embody the semantics of both Statemate[4] and Rhapsody[3]. Both have a stepping algorithm that first generates a list of candidate transitions before executing any transitions. The new stepping algorithm of SCCD is similar.

We have also seen that in original SCCD, the `step` function keeps executing transitions until no more transitions can be made. There are drawbacks from this as well:

- If a loop of transitions is always enabled for some reason, a `step` will never end.

- The statechart only returns to a stable state (by keeping executing transitions) after all due events have been processed. For the same series of tuples $(event, timeout)$ of input, known from the beginning, different outcomes are possible depending on the values for the `delta` argument with whom `step` invocations subsequently happened. It would make more sense for the statechart to return to a stable state after every event queue entry.

- It is not possible for multiple events to be present at the same time, and therefore transitions cannot have an event combination trigger. To support this, the form of event queue entries must be $(event\_list, timeout)$.

Finally, a drawback from the way internal events are handled in original SCCD:

- There is no differentiation between internally raised events and input events. This way, internally raised events do not become present until other (possibly external) timed out events have been processed. This makes it harder for the modeler to make assumptions on when the statechart will respond to an internal event.

We will now look at the new stepping algorithm in SCCD in Section 3. Later, in Section 4, the available semantic options will be discussed.

3 New Stepping Algorithm

3.1 Goal

The new stepping algorithm has been designed to be flexible enough to support multiple configurations of semantics, with the requirement of being able to express Rhapsody and Statemate semantics as a configuration. An additional goal was to be able to express the original SCCD semantics as a configuration, for compatibility reasons. This configuration will then serve as the default, i.e. if no options are specified by the modeler. Existing models are then guaranteed[3] to work, causing minimal frustration among developers working on existing models in original SCCD.

3.2 Big Step, Combo Step, Small Step

For the new stepping algorithm, inspiration came from Big Step Modeling Languages (BSML), as described in [2]. BSMLs serve as a greatest common denominator among statechart-like languages.

The execution of a BSML model is a sequence of *big steps*. A big step is a unit of interaction between a model and its environment. A big step takes input from the environment (at the beginning of the big step), and produces output to the environment (after the big step has taken place). Input cannot change during the big step.

A big step consists of 0 or more *small steps*. A small step is a set (unordered!) of 1 or more transitions, but in this paper, a small step will always be equal to 1 executed transition[4].

Small steps are grouped in so-called *combo steps*. A combo step is a maximal sequence of small steps, such that it only contains transitions that are orthogonal to each other[5].

[3]Well, almost...

[4]When using Concurrency semantics, a small step is a set (undordered!) of 1 or more transitions. At this point, Concurrency is not supported in SCCD, but it is possible that it will be, in the future. Section 4 provides a quick overview of Concurrency.

[5]Although we use combo steps, the semantic aspect Combo Step Maximality from [2] is fixed at "Combo Take One", because this is the only sane option.

3.3 Implementation

3.3.1 Event Queue Entries

In original SCCD, event queue entries where tuples (*event*, *timeout*). Because we also want to support multiple input events per big step, as well as to have a clear scope for big steps, event queue entries now are tuples (*event_list*, *timeout*). A big step is then executed for each such entry.

3.3.2 Big Step Implementation

The new `step` function still takes a parameter `delta`, advances time by decreasing timeouts, and pops due event queue entries. For each due entry (*event_list*, *timeout*), the new function `bigStep` is called with `event_list` as argument.

The `bigStep` method executes a single big step. It calls the `comboStep` method 1 or more times (depending on semantics, see Section 4). `comboStep` calls `smallStep` until it is no longer possible to include a small step in the current combo step.

The `smallStep` method calls `generateCandidates` to generate a list candidate transitions (in document order). Next, 1 or more candidates are selected and executed as a small step. If there are no candidates, `smallStep` returns `false`, indicating that no small step could be executed. This will cause the ending of the current combo step, and possibly the ending of the current big step as well.

Figure 3 shows the call stack associated with a `step`.

3.3.3 generateCandidates Function

The `generateCandidates` function is similar to the `transition` function in original SCCD. It is not part of the runtime; it is produced by the compiler. It produces a list containing *a subset of* all enabled transitions. Initially, this list is empty. It then performs a depth-first search on the object's tree of current states, checking the triggers and conditions of outgoing transitions. Transitions that are found to be enabled are *not* executed; they are added to the list of candidates. After the tree of current states has been scanned, the list of candidates is returned.

Just like with the `transition` function in original SCCD, by default, if at the current node an outgoing transition was added to the list of candidates, the children of that node are no longer checked. Again, it is possible to configure the semantics to do the opposite.

3.3.4 Transition Functions, Some Confusion

A transition candidate is implemented as a pair (*callback*, *parameters*), where `callback` is a member function that, when called with `parameters` as argument, executes the transition: Source state is exited, exit actions are executed,

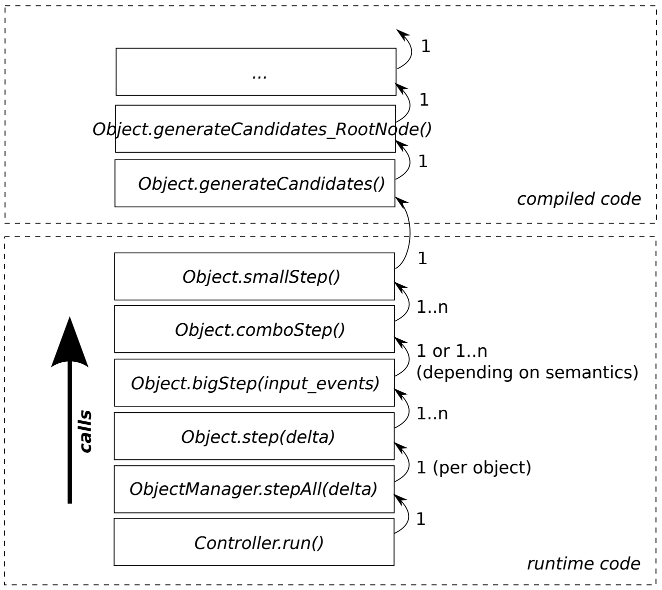

Figure 3: Call stack for the stepping of a single object.

transition actions are carried out, events are raised, enter actions are executed, destination state is entered. The callback member functions are produced by the compiler, and they are named `transition_<source state>_<index>`.

This may be confusing because in original SCCD, similarly named functions carried out the depth-first scan of current states. This is no longer the case. A member function with the above naming pattern now solely executes a single transition, without checking its enabledness.

4 Semantic Options

We will now discuss the new semantic options included in the new stepping algorithm of SCCD.

In the examples, a big step will be indicated as a sequence $[combo\_step, combo\_step, ...]$. A combo step will be indicated as a sequence $[small\_step, small\_step, ...]$. A small step will be indicated as a set $\{transition, transition, ...\}$.

4.1 Big Step Maximality

Big Step Maximality specifies when a big step ends. Currently, there are 2 options:

Take One A big step consists of at most 1 combo step.

Take Many A big step consists of as many combo steps as possible.

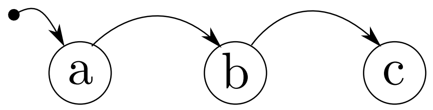

Figure 4: Statechart for demonstrating big step maximality semantics. All transitions have no trigger and no condition.

Example: In Figure 4, state a is entered during initialization of the statechart. In the next big step, the transition from a to b is enabled, and executes. That transition makes up the first small step, and that small step is part of the first combo step of the big step. The transition from b to c is not orthogonal with respect to the transition from a to b, so it cannot be part of the same combo step. If we use Take Many, a new combo step is executed, consisting of a single small step with the transition from b to c. It is then not possible to execute another small step, so the current combo step ends. Because at least 1 transition was executed in the last combo step, the big step attempts to execute another combo step, but due to the lack of enabled enabled transitions, that attempt fails and the big step ends. So the big step is $[[\{a\_to\_b\}], [\{b\_to\_c\}]]$. If we use Take One, the big step ends after the first combo step. The transition from b to c can still be executed in the next big step. The sequence of big steps then is $[[\{a\_to\_b\}]], [[\{b\_to\_c\}]]$.

Let us now look at an more complex example, involving orthogonal components.

Example: In Figure 5, states sa and sd are entered during initialization. In the first big step a combo step is executed, which executes a small step. The small step generates a list of candidate transitions, in document order. The candidates are sa_to_sb and sd_to_se. Supposing we don't use concurrency semantics, the first of these candidates is selected, so sa_to_sb makes up the first small step of the first combo step. The current combo step executes another small step. This time, the only candidate is sd_to_se, because the transition sb_to_sc is not orthogonal to sa_to_sb, which is already part of the current combo step. So sd_to_se is executed. It is not possible to include another small step in the current combo step, so the combo step ends. If we use Take One, the current big step also ends. The big step is then $[[\{sa\_to\_sb\}, \{sd\_to\_se\}]]$, followed by $[[\{sb\_to\_sc\}, \{se\_to\_sf\}]]$. If we use Take Many, another combo step begins, consisting of small steps $\{sb\_to\_sc\}$ and $\{se\_to\_sf\}$, in document order. The big step is then $[[\{sa\_to\_sb\}, \{sd\_to\_se\}], [\{sb\_to\_sc\}, \{se\_to\_sf\}]]$.

The major advantage of Take One is that a big step is guaranteed to end: The model will always be able to sense input again. Take Many has the potential risk of never-ending big steps. Take Many allows for sequences of non-orthogonal transitions to be taken in a single big step. There may be circumstances where this is useful.

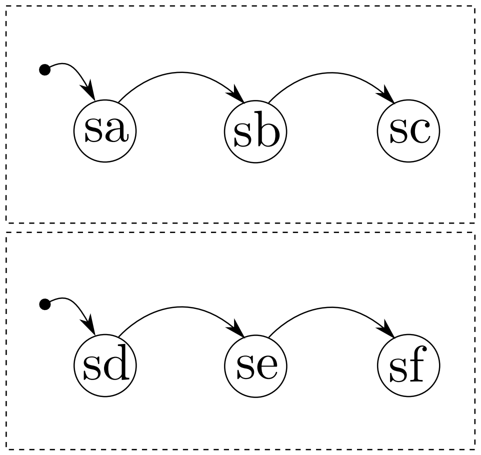

Figure 5: Statechart for demonstrating big step maximality semantics. All transitions have no trigger and no condition.

4.2 Internal Event Lifeline

Internal Event Lifeline specifies when an internally raised event becomes present. There are 3 options:

Next Small Step The event becomes present in the next small step.

Next Combo Step The event is present throughout the next combo step. This option only makes sense in combination with Take Many.

Queue Internally raised events are treated like external input events, and added to the object's event queue. It is possible that other external events are responded to before the raised event becomes present.

Example: Have a look at Figure 6. Suppose we use Take Many for big step maximality. The model enters a during initialization, and suppose event e is popped from the event queue. If we use Next Small Step, *a_to_b* makes up the first small step and first combo step. Event f is internally raised and present in the next small step. The next small step is part of the next combo step. The big step is $[[\{a\_to\_b\}], [\{b\_to\_c\}]]$. If we use Next Combo Step, behavior is exactly the same as with Next Small Step, because in this model, each combo step consists of only one small step. If we use Queue, again, *a_to_b* makes up the first small step and first combo step, but this time, event f is added to the event queue. As a result, f doesn't become present until some next big step. So the current big step $[[\{a\_to\_b\}]]$ ends and $[[\{b\_to\_c\}]]$ may happen later on.

Example: Have a look at Figure 7. We use Take Many, and event e is an input event. The model enters *sa* and *sc* during initialization. At first, only *sc_to_sd* can be executed, so that transition makes up the first small step.

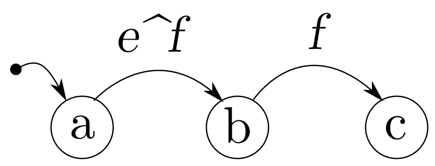

Figure 6: Statechart for demonstrating internal event lifeline semantics.

If we use Next Small Step, event f becomes present and causes the transition *sa_to_sb* to make up the next small step. This small step is still part of the same combo step, because it is orthogonal to the first transition. Event g is raised and becomes present in the next small step. Because it is not possible to include another transition in the current combo step, a new combo step begins. The candidates of the first small step of the new combo step are *sb_to_sa* and *sd_to_se*, in document order. The first is executed. It is not possible to execute another small step, because event g is no longer present. So *sd_to_se* never executes. We could say that event g "went lost". The big step is $[[\{sc\_to\_sd\}, \{sa\_to\_sb\}], [\{sb\_to\_sa\}]]$. If we use Next Combo Step, *sc_to_sd* makes up the first combo step. In the 2nd combo step consists of *sa_to_sb*. Throughout the 3rd combo step, event g is present. This allows both *sb_to_sa* and *sd_to_se* to be executed (in document order), because they are orthogonal to each other. If we use Queue, a series of big steps happens: $[[\{sc\_to\_sd\}]]$, $[[\{sa\_to\_sb\}], [\{sb\_to\_sa\}]]$ and finally $[[\{sd\_to\_se\}]]$.

Although Next Small Step may seem intuitive if we imagine an internally raised event to cause a transition in the next small step, sometimes a transition from another orthogonal component makes up the next small step, and the event "goes lost". It is also not possible for a raised event to be sensed in multiple orthogonal components. Queue semantics ensures that a big step is executed for every internally raised event. But this has the disadvantage that the modeler has no control over when a raised event will be responded to. It is possible for other external input events to be processed first. Next Combo Step allows for internally raised events to be responded to in the same big step, and allows for events to be present throughout a whole combo step, allowing every orthogonal component to sense every internal event. But this option can only be used in combination with Take Many for big step maximality.

Because in some cases, we want to allow the environment to "interrupt" the execution of a statechart, Queue can be useful. But if we don't want *every* internal event to be treated like an external event, it is possible to use Next Small Step or Next Combo Step for those events, and narrow cast "interruptable" events to the object itself.

4.3 Input Event Lifeline

Input Event Lifeline specifies when an input event (popped from the object's event queue) is present during a big step. There are 3 options:

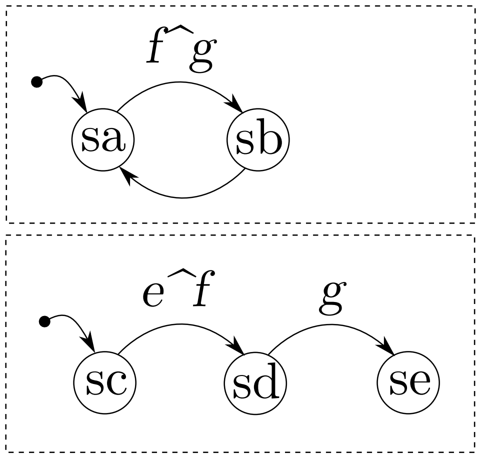

Figure 7: Statechart for demonstrating internal event lifeline semantics.

First Small Step The event is present during the first small step.

First Combo Step The event is present during the first combo step.

Whole The event is present entirely throughout the big step.

4.4 Priority

Priority specifies what kinds of transitions have priority to be added to the list of transition candidates during the depth-first search. This option used to be called "conflict" in original SCCD. There are 2 options:

Source-Parent Transitions whose source is an ancestor of the source of another transition have priority over that transition. This option used to be called "outer" in original SCCD.

Source-Child Transitions whose source is a (possibly indirect) child of the source of another transition have priority over that transition. This option used to be called "inner" in original SCCD.

Example: In Figure 2, if we use Source-Parent, the outer transition t_2 will execute as the only transition in the big step. If we use Source-Child, t_1 and t_2 will each make up one combo step.

4.5 Concurrency

Concurrency specifies whether multiple transitions can be included in a small step. This option specifies the selection of transition(s) among a list of candidates. Currently there is only 1 option, but this may change in the future.

Single Each small step consists of 1 executed transition.

If we use Single, orthonogal transitions are executed in document order and they make up the same combo step.

5 Conclusion

We have explained some shortcomings of the stepping algorithm in original SCCD, explained the concepts big step, combo step and small step, explained the implementation of the new stepping algorithm, and the semantic options currently supported by SCCD.

In the future, more options will be added. For instance, SCION[1] uses an event queue for internal events, such that one small step is executed for every raised event. It is currently not possible to achieve similar behavior with the available options.

6 Practical

Latest version can be found in the "semantics" branch of the SCCD project on SVN. Note that the Javascript runtime still has to be updated, so only Python will work for now.

A Overview of some differences between original SCCD and new SCCD

	Original SCCD	New SCCD
Compiled code		
Depth-first search performed by	`transition` function	`generateCandidates` function
Transitions executed by	-	`transition_<state>_<index>` callbacks
Runtime code		
Event Queue Entries	$(event, timeout)$	$(event\_list, timeout)$
Semantic options		
Priority Semantics	"conflict" (outer/inner)	"priority" (source_parent/source_child)

B Rhapsody, Statemate and Original SCCD Configurations for Semantic Options

Option	Rhapsody	Statemate	Original SCCD
Big Step Maximality	Take Many	Take Many	Take Many
Internal Event Lifeline	Queue	Next Combo Step	Queue
Input Event Lifeline	First Combo Step	First Combo Step	First Combo Step
Priority	Source-Child	Source-Parent	Source-Parent
Order of Small Steps	?	?	Document Order
Concurrency	Single	Single	Single

Options in last column (Original SCCD) are also the default options for the compiler.

C Pseudocode for Big Step, Combo Step and Small Step Algorithms

bigStep function:

```
stepped = False
reset small step and combo step state (clearing internal events)
while comboStep():
  stepped = True
  if semantics.big_step_maximality == TakeOne:
    break
return stepped
```

comboStep function:

```
stepped = False
while smallStep():
  stepped = True
return stepped
```

smallStep function:

```
c = generateCandidates()
if c:
  transition, parameters = c[0]
  transition(parameters)
  return True
return False
```

generateCandidates will only return candidates that can be executed with respect to the current combo step and priority semantics.

References

[1] Jacob Beard. Scion: Statecharts interpretation and optimization engine. `https://github.com/jbeard4/SCION`, October 2014.

[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 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.

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

[5] Glenn De Jonghe. Statecharts and class diagram xml: A general-purpose textual modelling formalism. `http://msdl.cs.mcgill.ca/people/glenn/report.pdf`, June 2014.