Universiteit Antwerpen

A Study and Implementation of Semantic Variability in Statecharts

Joeri Exelmans

Principal Adviser: Hans Vangheluwe
Assistant Adviser: Simon Van Mierlo

Dissertation Submitted in August 2020 to the
Department of Mathematics and Computer Science
of the Faculty of Sciences, University of Antwerp,
in Partial Fulfillment of the Requirements
for the Degree of Master of Science.

Contents

List of Figures

List of Tables

Abstract

Over the years, the statecharts formalism has been proven to be useful for the development of reactive systems. However, the semantics of statecharts are not precise, with implementations making their own semantic decisions.

This thesis is a study of the possible semantics of statecharts, based on the work on Big-Step Modeling Languages, and an implementation of these semantics in a statechart interpreter. By supporting a range of semantics, the modeler can select the semantics most fit for the problem at hand, and we can claim (some level of) compatibility with existing implementations. As part of our statechart language, an action language was developed, in order to support certain variable assignment semantics, and to make models portable.

We evaluate our solution by comparing it to the open-source statechart IDE and compiler YAKINDU, studying its semantics and features, and perform a case study, implementing example models in both solutions.

Nederlandstalige Samenvatting

De toepasbaarheid van het statecharts-formalisme voor het ontwikkelen van reactieve systemen heeft zich over de jaren bewezen. De semantiek van statecharts is echter niet exact bepaald, wat implementaties heeft toegelaten hun eigen semantische keuzes te maken.

Deze thesis omvat een studie van de mogelijke semantieken van statecharts, gebaseerd op het onderzoek omtrent Big-Step Modeling Languages, en een implementatie van deze semantieken in de vorm van een statechart-interpreter. Door een spectrum aan semantieken te ondersteunen, kan de modeller de meest geschikte semantiek kiezen om een probleem op te lossen, en kunnen we beweren (in zekere mate) compatibel te zijn met bestaande implementaties. Als onderdeel van onze statechart-taal werd een actietaal ontwikkeld, wat noodzakelijk was om bepaalde variabele-toewijzingssemantieken te ondersteunen, alsook het ondersteunen van platformonafhankelijke modellen.

We evalueren onze oplossing aan de hand van een vergelijking met de open-source statechart IDE en compiler YAKINDU. We beschrijven YAKINDU's semantiek en kenmerken, en implementeren voorbeeldmodellen in beide oplossingen.

Acknowledgements

I would like to thank Simon Van Mierlo for his excellent guidance, review and feedback of this thesis. I would also like to thank my promotor, Hans Vangheluwe for giving me the chance to re-enroll at university after dropping out a couple of years ago. And naturally, I thank family and friends for their support.

CHAPTER 1

Introduction

Statecharts [10] are a domain-specific language (DSL) suitable for modeling reactive, discrete-event, autonomous, timed, complex and concurrent systems. Implementing such systems with the use of low-level primitives such as threads, locks and timers is a very difficult task. Statecharts hide these implementation details and allow the modeler to focus on the *what*, instead of the *how*.

However, the semantics of statecharts are not precisely defined, in many cases leaving room for choices to be made by implementors. Since their inception, several statechart implementations and standards have been developed with varying semantics.

The SCCD formalism [17] combines Class Diagrams with statecharts, supporting multi-statechart models with dynamic creation (and destruction) of instances. In the context of the SCCD project, a statechart compiler has to be developed. Given the semantic variability of existing statechart implementations, the most general solution is to support a range of semantics.

Big-Step Modeling Languages (BSML) [9] are a family of DSLs, including various statecharts implementations, that have been mapped onto a common syntax, with their semantics decomposed into a set of precisely described variation points. This makes BSML a powerful framework for studying semantic variability in statecharts.

In this work, we have selected a subset of BSML semantic options that we think are applicable to statecharts. We have implemented these options in a statechart interpreter. (The reasons to develop an interpreter instead of a compiler will be motivated.) In accordance with BSML, we allow the modeler to choose the semantics on a by-model basis.

1.1 Technical remarks

Since January 2020, Python 2 no longer receives (security) updates, so our solution solely targets Python 3, compatible with versions $\geq$ 3.6, including the just-in-time compiler PyPy [1]. Function and method signatures are extensively provided with type annotations, which can be statically checked for correctness, but primarily serve as a form of documentation. The implementation's dependencies on 3rd party libraries have been intentially kept small. The latest version of our implementation can be found in the 'joeri'-branch of the SCCD project: `https://msdl.uantwerpen.be/git/simon/SCCD/src/joeri`

1.2 Code fragments

In this thesis, code fragments are sometimes used to illustrate hypothetical or actual implementations. We do not use a separate "pseudocode" syntax. Instead, Python's syntax is used everywhere, as it is clear and compact. The only exception is when we show code fragments of our action language, which has its own textual syntax.

1.3 Outline

The rest of this thesis is organized as follows: The first half of Chapter 2 introduces the statecharts formalism and already illustrates the non-precise semantics of the statecharts formalism. The second half of 2 introduces the semantic options of BSML and discusses their applicability to statecharts. Chapter 3 describes the implementation of SCCD's statechart execution runtime. Chapter 4 evaluates our solution by comparison with YAKINDU and the implementation of example models.

[1] https://www.pypy.org/

CHAPTER 2

Background

In this chapter, we introduce the statecharts syntax and semantics, while showing that in many cases, the semantics of statecharts are not very precisely defined. Next, we introduce Big-Step Modeling Languages (BSML), which is a framework for precisely describing the semantics of a family of DSLs, including (variants of) statecharts. We then look at what aspects of the BSML framework are applicable to statecharts, on which we will base our selection of aspects to implement.

2.1 Statecharts

Statecharts are a formalism for modeling *reactive, discrete-event, autonomous, timed, complex* and *concurrent* systems:

Reactive A statechart may receive input at any point in time.

Discrete-event Input arrives in the form of events at discrete points in time, as opposed to continuously.

Autonomous + timed A statechart has an internal notion of time, and may spontaneously act as result of time having passed.

Complex Statecharts can model complex behavior in a compact, understandable way.

Concurrent Multiple types of behavior can simultaneously co-exist.

Implementing these types of systems in a procedural programming language using timers, threads, shared memory and locks is extremely difficult. As we will see, statecharts alleviate this difficulty by focusing on the *what* instead of the *how*.

2.1.1 Flat statecharts

Statecharts [10] are an extension of finite-state machines (FSM), primarily adding, *hierarchy*, *concurrency* and *history*.

For now, we explain the syntax and semantics of *flat* statecharts.

Syntax

A flat statechart consists of:

- A finite set of states S.
- An initial state $i \in S$, sometimes also called *default state*.
- A set of input events X, internal events Y and output events Z. An event $e \in X \cup Y \cup Z$ has:
  - a *unique name* within the statechart.
  - Optional: a set of *named parameters* that are assigned values upon event instantiation.
- A set of variables V, and for every variable $v \in V$, an initial value.
- A set of transitions T, where every transition has:
  - A source state $s \in S$.
  - A target state $t \in S$.
  - Optional: An event trigger $e \in X \cup Y$, or a time duration after which to automatically enable the transition.
  - Optional: A guard condition, a boolean expression, possibly reading variables and event parameters, that, if evaluating to false, disables the transition.
  - Optional: A set of actions, executed when the transition fires. Possible actions include
    - Raising an event $e \in Y \cup Z$.
    - Assigning a new value to a variable $v \in V$.

The set of transitions that has a state as its source is referred to as that state's set of *outgoing* transitions. Similarly, the set of transitions with the same state as a target is called that state's set of *incoming* transitions.

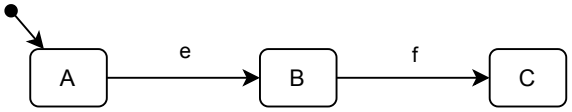


Figure 2.1: Simple flat statechart

Semantics

During execution, the "state" of a statechart is defined by its set of current states, called the statechart's *configuration*, and its variable values. For flat statecharts, the configuration always consists of a single state. The configuration changes when a transition fires: the transition's source state is removed from the configuration, and its target state is added. When a state is added to the configuration, we say that the state is *entered*, and when removed, the state is *exited*.

Similar to FSM's, statecharts only react when there is input (timed transitions are a special case of input, as we will see). In the general case for statecharts, input comes in the form of a set of simultaneous events. For a set of input events, a statechart may make any number of transitions, producing a set of output events and a new configuration, ready to react to a next set of input events. Often, we will only consider input consisting of a single event, like with FSM.

Figure 2.1 shows a very simple statechart with initial state A. If from the initial state, the event e is received, the statechart will *enable* the transition from A to B. The transition from B to C will not be enabled, because B is not in the configuration. For input e, the set of enabled transitions thus consists only of a single transition, which is fired, changing the configuration from $\{A\}$ to $\{B\}$.

Possible execution algorithms

In the previous example, what happens if from initial state A, the input events $\{e, f\}$ are simultaneously received? One might say that, "naturally", in response to this set of input events, both transitions are made, resulting in the final configuration $\{C\}$. An execution algorithm producing this behavior may look as follows:

```
def step(configuration, input_events):
  while True:
    enabled_transitions = determine_enabled_transitions(configuration,
       input_events)
    if len(enabled_transitions) == 0:
      break # done
    elif len(enabled_transitions) == 1:
      configuration = fire_transition(configuration, enabled_transitions
         [0])
    else:
      # ???
  return configuration
```

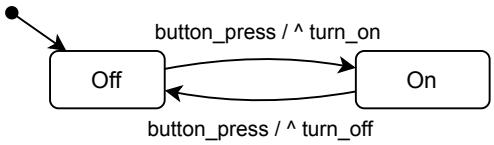


Figure 2.2: Simple light switch statechart

```
# example ''loop''
def statechart_loop():
  while True:
    input_events = wait_for_input()
    step(configuration, input_events)
```

This algorithm however, can lead to a never-ending "step", if the model contains a "loop" of transitions, re-entering a previous state. This is highly common in practice: Figure 2.2 shows a statechart implementing a light than can be switched on or off by pressing the same button. Our algorithm from above would endlessly oscillate between Off and On, never concluding the step. A proper semantics for this statechart would be for the button press event to be "consumed" when a transition is made. There are 2 ways to implement this in an algorithm:

1. We could *clear the set of input events* after the first transition made
2. We could allow our step-function to only *execute a single transition.*

Variant (1) would clear the `input_events` list at the end of each loop iteration. Variant (2) would only keep the loop body.

The semantics of most statechart implementations behave along the lines of these variants. For instance, STATEMATE [12] and YAKINDU [6] use (1). Simply allowing only a single transition to be made, can be quite restrictive though: Figure 2.3 shows a statechart with a sequence of algorithm steps. Some transitions have only a guard condition, which is allowed, and several have no event trigger or guard, which is also allowed. If we want to reach the Finish-state, any number of transitions should be allowed after receiving the "start"-event. An alternative would be to implement the sequence of steps as a single transition.

Internal events

There are more semantic choices to be made when we introduce internal events.

Figure 2.4 shows a flat statechart that responds to (input) event e by raising (internal) event f. If only one transition is allowed to be made for every "step" (set of inputs), the statechart's final configuration is $\{B\}$, no matter the event semantics. If multiple subsequent transitions are allowed, the configuration at the end of the step could be:

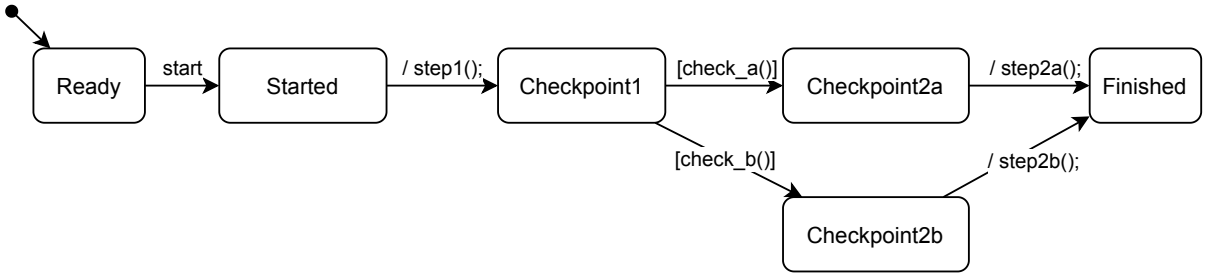


Figure 2.3: Sequence of algorithm steps modeled as a statechart

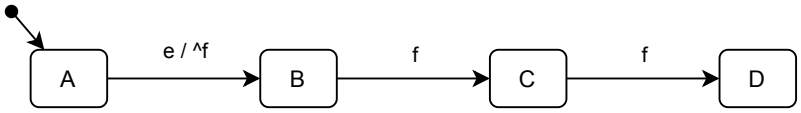


Figure 2.4: Flat statechart with internal events

- $\{D\}$, if we decide event f remains *active throughout the remainder of the step.*

  This behavior is trivially implemented by appending internally raised events to the set of enabled events.

- $\{C\}$, if we decide event f is *allowed to trigger at most 1 transition* (the transition "consuming" it). This would prevent "looping" behavior, while still allowing responses to internal events.

  This could be implemented by enabling f only in the loop iteration following the iteration in which it was raised.

- $\{B\}$ (!): If we decide to treat internally raised events as input events for a later step.

Again, there is no "right" answer. Statechart dialects exist for each of these choices.

Enter and exit actions

Enter and exit actions are additional syntax, part of most statechart dialects, and, until we introduce history-states, may be regarded as a form of *syntactic sugar*: they increase the clarity of the language, but can be expressed in other constructs.

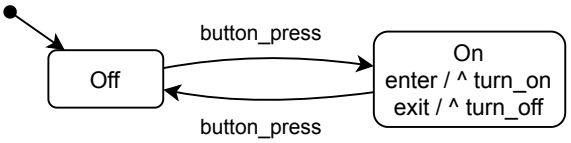
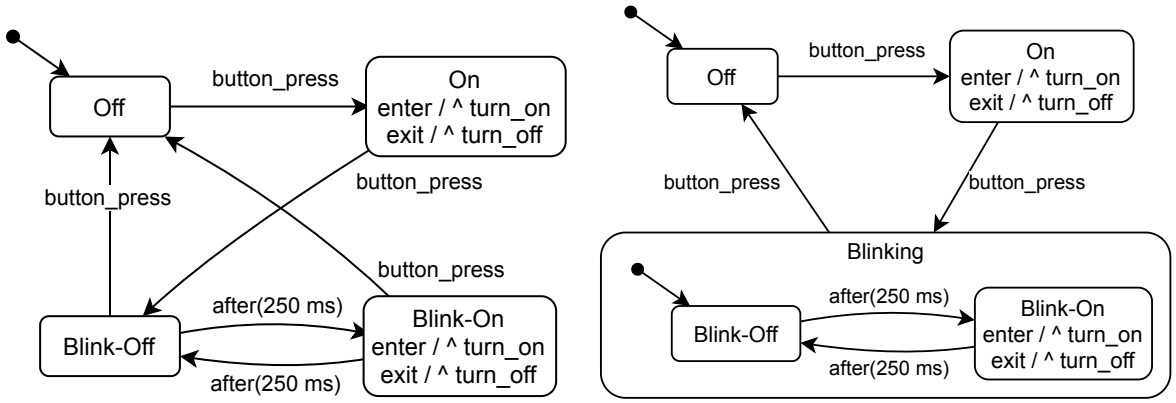


Figure 2.5: Light switch with enter and exit actions



(a) Flat statechart (b) Hierarchical statechart (next section)

Figure 2.6: Light controller with blinking-state

Enter- and exit actions are properties of a state. They are actions to be executed when the state is entered and exited, respectively. For any state, enter actions could also be appended to the actions of every incoming transition, and exit actions could be prepended to the actions of every outgoing transition.

Figure 2.5 shows the same behavior as the light switch in Figure 2.2, but with enter- and exit-actions. By putting the output events "turn_on" and "turn_off" as actions in the On-state, we cannot "forget" to turn the light off (e.g. when adding more outgoing transitions to the On-state), since it is automatically turned off when exiting that state, and the only way to turn it on, is to enter the On-state.

Timed transitions

A feature often included in statechart implementations, is that of *timed transitions*. It allows transitions to be triggered by having spent a certain amount of time in a state.

Suppose we want to add a "blinking" state to our light controller from Figure 2.5, e.g. to serve for a bicycle light. Figure 2.6a shows the solution. 2 new states are added, together representing the "blinking" state. The 2 new states are alternated between "spontaneously" (without interaction through any button presses). Pressing the button again from any of the 2 states takes us back to the Off-state.

Timed transitions seem to contradict our earlier statement that a statechart can only react to an input. This remains true, but the statechart will ask the caller (through a callback) of the step-function for a timer to be started, and to be sent a special *input event* when its timeout passes. Likewise, the statechart may ask the caller to cancel an earlier timer, if the source state requesting the timer is exited before the timout has passed.

The "caller" here, is the integration of the statechart's execution into some platform. It is responsible for scheduling and canceling timers, but also for collecting and delivering input and output events.

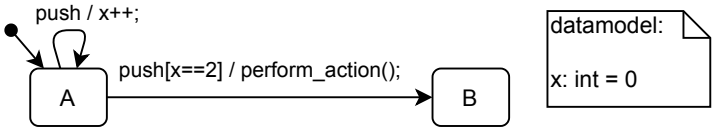


Figure 2.7: Example statechart with ambiguous transitions in state A if $x == 0$

Transition ambiguity

Up until now, the set of enabled transitions, also called the set of *transition candidates* consisted of at most 1 transition. When the set of candidates contains more than 1 transition, a single transition should be *deterministically* chosen, and preferrably in a fashion that is transparent to the modeler.

Figure 2.7 shows a statechart representing a button that causes an action to be performed when it is pushed 3 times. The statechart has an integer variable `x` that is initialized with value 0. After the button is pushed twice, the statechart having responded 2 times to the input event "push", a third push-event could be responded to with either the self-transition $A \rightarrow A$, or $A \rightarrow B$. In order to make the model behave correctly, the modeler could:

- Narrow the circumstances under which A's self-transition is enabled, by adding a guard `x < 2`.
- Assign a higher *priority* to the transition from A to B.

Even if the first strategy does not require an extension of the syntax or semantics, supporting explicit priorities for outgoing transitions of a state prevents non-determinism in the general case for flat statecharts. This would further alter our execution algorithm, allowing multiple candidates and always firing the candidate with the highest priority.

Conclusion

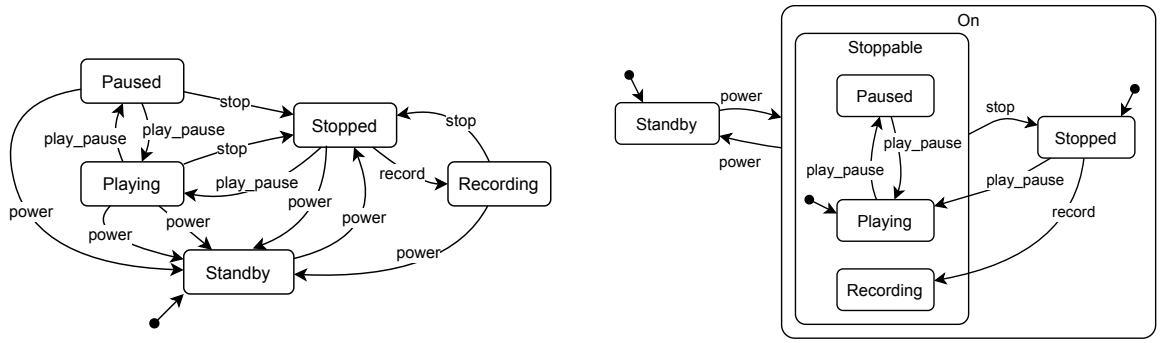
We have introduced the features of flat statecharts. We have also seen that even for flat statecharts, the semantics are not always clear, and choices can be made in implementations. Now, we will introduce hierarchy, followed by orthogonality.

2.1.2 Hierarchy

By adding *hierarchy* to FSM / flat statecharts, states can be *clustered* into arbitrary hierarchical levels. Clustering allows expressing the same behavior with fewer transitions.

Figure 2.6b shows the behavior of our light controller from an earlier example, with the 2 blinking-states explictly clustered into a Blinking-state. By considering this state as a state on its own, this results in one fewer "button_press"-transition.

Figure 2.8a shows the behavior of a video-cassette recorder (VCR) modeled as a flat statechart. The VCR has 5 states and can respond to 4 different commands (power, play_pause, stop, record), and has 13 transitions. A hierarchical statechart modeling the same behavior is shown in Figure 2.8b. Paused, Playing and Recording are clustered into



(a) Flat statechart (b) Statechart with hierarchy

Figure 2.8: The behavior of a VCR, modeled as a finite-state machine, and as a statechart

the Stoppable-state, and at a higher level, Stoppable and Stopped have been clustered into the On-state. All 4 power-event triggered transitions to the Standby-state have been replaced by a single transition from the On-state, just like the 3 stop-event triggered transitions.

The meaning of a cluster, or *composite state*, as we will call them, is an (exclusive-)OR-relation between its substates. For instance, the Stoppable-state represents Paused, Playing OR Recording. Other terms sometimes used for a composite state include *Or-state* and *region*.

State hierarchies are easier to comprehend, as higher-level states abstract the detail expressed by lower-level states. While modeling, *clustering* is a *bottom-up* operation, where detail is abstracted. *Refinement* is a *top-down* operation, where abstractions are given detail. An example of refinement would be adding substates for playback speed to the Playing-state in Figure 2.8b.

Syntax

A hierarchical statechart consists of the same elements as a flat statechart, but with the set of states forming a *state tree*. The root of the tree is called the *root state*, which usually implicit (not drawn) in the visual representation of a statechart. The leaf nodes of the state tree are sometimes referred to as *basic states*. The non-leaf nodes are composite states. The root state is always a composite state. Figure 2.9 shows the state tree of the hierarchical VCR statechart.

Instead of a single initial state at the level of the statechart, every composite state, including the root state, has an initial state, which must be a direct child. In Figure 2.6b, Blinking-Off is the default state of Blinking. In Figure 2.8b, Stopped is the default state of On.

Definitions

Set of ancestors For every state, the set of ancestors includes the state's parent state, and the ancestors of the parent. The set of ancestors always includes the root state, except for the root state itself, whose set of ancestors is empty.

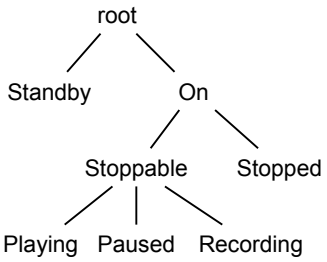


Figure 2.9: State tree of VCR statechart

Set of descendants For every state, the set of descendants consists of the direct children of the state, and the descendants of the children.

Transition arena The arena of a transition is the composite state that is the lowest common ancestor (LCA) of its source and target states.

Semantics

The semantics of statechart execution do not drastically change with the introduction of hierarchical states. Transitions can still have any state as source, and any state as target. The set of basic states still sums up the different configurations that the statechart can be in, but a statechart's configuration will consist of multiple states now: a basic state, and its ancestors.

Entering and exiting states

When a transition is fired, several states may be exited, and several states may be entered, if the transition spans several hierarchical levels. The following hierarchical "rules" apply:

- Before a state is entered, all of its ancestors must be present in the configuration.
- Before a state is exited, all of its descendants must be exited first.

This causes a child-to-parent ordering of exited states, and parent-to-child ordering of entered states. This ordering only affects the order in which enter- and exit-actions are performed.

In Figure 2.6b, with the transition On → Blinking, the On-state is exited, followed by entering the Blinking and Blink-On states, in that order. With the transition Blinking → Off, either the Blink-Off or Blink-On state is exited (depending on the configuration), followed by exiting the Blinking-state, and finally, the Off-state is entered.

Chapter 3 will show an implementation that calculates the sets of exited and entered states mostly statically.

Hierarchical priority

In the section on flat statecharts, we explained how an explicit priority-ordering between the outgoing transitions of a state can prevent non-determinism. With hierarchy, there could be a similar ambiguity between the outgoing transitions of a state and the outgoing transitions of its ancestors, as they can be simultaneously enabled.

By always giving higher priority to transitions whose source is deeper or shallower in the state tree, this form of non-determinism is avoided: The combination of a hierarchical priority and an explict "flat" priority yields a total ordering of all the transitions in a statechart.

Giving a higher priority to transitions with shallower source states has the benefit that *refinement* of a state cannot alter the behavior of a statechart at the existing level. Giving higher priority to deeper transitions allows refinement to override behavior.

STATEMATE [12] gives higher priority to shallower transitions. Rhapsody [11], UML state machines and ROOM [16] give higher priority to deeper transitions.

2.1.3 History

Another addition of statecharts to FSM is *history*. History enables a statechart to "remember" the configuration of an Or-state when it was exited, in order to be able to restore it at a later point in time.

Syntax

History adds 2 new types of *pseudo-states* to the state tree: *shallow history* and *deep history*. A history state always occurs as the child of an Or-state, that it records the history of, each time that state is exited.

A history state can be the target of a transition, but not the source. A transition targetting a history state indicates that it wants to restore the previously recorded configuration.

Semantics

Pseudo-states, like history states, exist in the syntax, but can never occur in the statechart's configuration. If the history state is the target of a transition, during execution of the transition, the previously recorded configuration of the history's parent state is entered.

When exiting an Or-state that has a history state as its direct child, the configuration of the Or-state must be recorded. A shallow history state records the configuration only at the level of the direct children of the Or-state it belongs to (= a single state). A deep history state records the configuration of all the descendants of the Or-state (= a set of states).

The statechart in Figure 2.10 has a history state. When the transition $C \to D$ is fired, the states C, Y and X are exited, in order. When X is exited, the shallow history state in X records the state Y as its "past configuration". When the transition $D \to H$ is fired, state X is entered, followed by state Y (the recorded "history value"), and B (the default

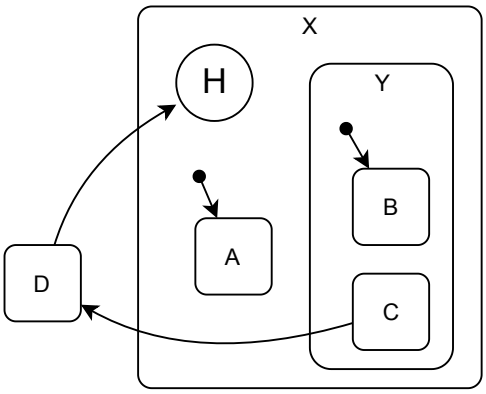


Figure 2.10: Statechart with a (shallow) history state

state of Y), resulting in the final configuration $\{X, Y, B\}$

If the shallow history state in Figure 2.10 were replaced by a deep history state, it would record $\{Y, C\}$ as a history-value (instead of just Y), resulting in the final configuration $\{X, Y, C\}$

When a history state is entered for which no history has yet been recorded, its parent is simply entered instead (and as a consequence, the parent's default states, recursively). As an alternative, some statechart implementations allow a pseudo-transition from a history state to a sibling of the history state, indicating that the sibling is the "default" state to enter, in case no history has been recorded yet.

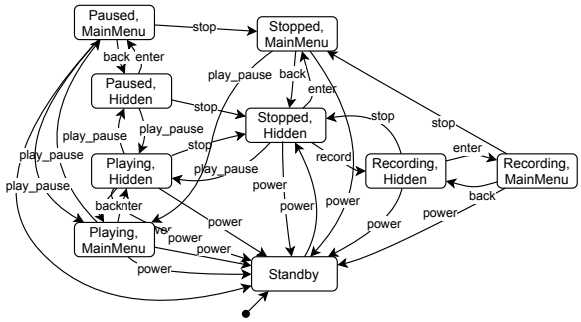
It is clear that implementations are required to keep in memory a recorded configuration for every history state in the statechart. This is part of the statechart's "state", along with the statechart's configuration, and scheduled timers (for timed transitions).

2.1.4 Orthogonality

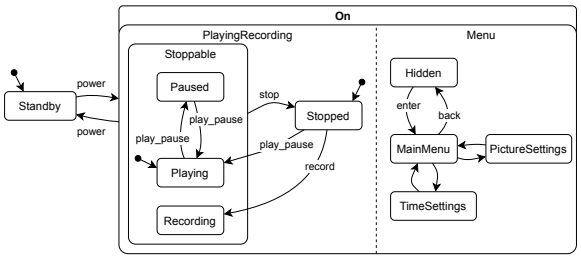
Another addition of statecharts to finite state machines is *orthogonality*. Up until now, the statechart's configuration always consisted of a single basic state, and the ancestors of that state. Orthogonality allows statecharts to be simultaneously in multiple (basic) states that are not ancestors of each other, allowing modeling of independent features.

Suppose we add an onscreen settings menu to our VCR example. The menu can be accessed at any time. Modeling this behavior with a flat statechart, or even making use of composite states would lead to a *blowup* of states, as every existing state must be duplicated for every possible state of the menu: menu shown/hidden, the item in the menu selected, etc. Figure 2.11a shows a FSM/flat statechart with only the possibility of the menu being shown/hidden modeled, and the number of states and transitions has almost doubled.

The same behavior, modeled as a statechart with orthogonality is shown in Figure 2.11b: The On-state split up in 2 *orthogonal regions*: PlayingRecording and Menu. The meaning of the On-state has become PlayingRecording AND Menu, as both sub-regions



(a) Finite-state machine



(b) Statechart with hierarchy and orthogonal components

Figure 2.11: The behavior of menu-extended VCR

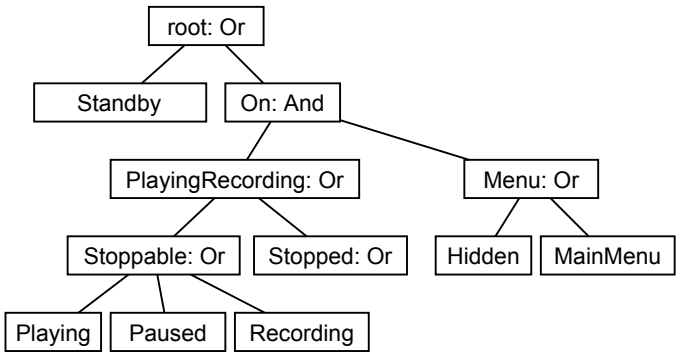


Figure 2.12: State tree of menu-extended VCR statechart

are active states at the same time. The On-state is called a *parallel* state, or *And-state*.

Syntax

Syntactically, orthogonality builds on hierarchy, only introducing a new type of state to the state tree: The *parallel state*, or *And-state*. Just like composite states, parallel states are always non-leaf nodes in the state tree. Figure 2.12 shows the state tree of the orthogonal VCR statechart.

We will more often talk about *orthogonal regions* than about And-states. Orthogonal regions are Or-states that have an And-state as their common parent. The And-state itself is often not explicitly mentioned, and sometimes the name of the And-state is not even drawn in the visual representation of the statechart. Orthogonal regions are usually drawn next to each other, separated by a dotted line.

Transitions are still allowed from any state, to any state, including between orthogonal regions (Figure 2.13), although this is rarely encountered.

Semantics

The "invariant" that must hold before and after firing a transition, is that, if an And-state is part of the statechart's configuration, all of its direct children must also be part

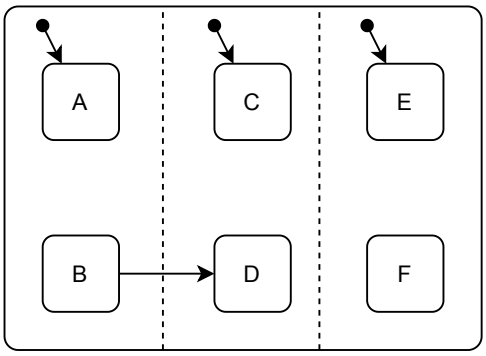


Figure 2.13: Cross-orthogonal region transition

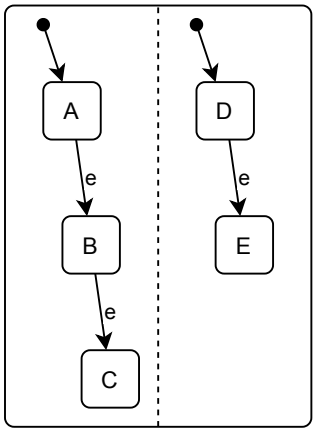


Figure 2.14: Statechart with 2 orthogonal regions

of the statechart's configuration. This means that upon entering an And-state, all of its children are entered (after the And-state is entered), and when exiting an And-state, all of its children are exited (before exiting the And-state).

This makes determining the set of entered and exited states of a transition somewhat complex, but still free from any ambiguities. In principle, when firing a transition, all of the states in the configuration that are descendants of the transition's arena (lowest common Or-state ancestor) are exited. Usually transitions do not cross orthogonal regions. Figure 2.13 shows an uncommon example of a cross-orthogonal region transition: The transition will cause the entire And-state to be exited and re-entered. If the configuration is $\{B, C, F\}$, the configuration becomes $\{A, D, E\}$.

Section 3.4.3 will explain our implementation with a complex transition as an example.

Concurrent transitions

Figure 2.14 shows a statechart with 2 orthogonal regions, both responding to an event e. There are several ways we can interpret the statechart's reaction to e, if we are in $\{A, D\}$:

- One possible type of semantics would allow both regions to fire as many transitions

as they can, e remaining enabled throughout. This would result in the configuration $\{C, E\}$. With this type of semantics, what would be the order of the transitions fired?

- For flat statecharts, we saw that it may be desirable to allow only 1 transition per input event, "consuming" the event, and avoiding endless steps. But would this result in the configuration $\{B, D\}$ or $\{A, E\}$?
- Another type of semantics would be to allow at most one transition to be executed *per orthogonal component*. This would result in the configuration $\{B, E\}$.

Orthogonal region priority

A partial answer to the above questions would be to explicitly assign priorities to orthogonal regions, similar to how we could assign explicit priorities to a state's outgoing transitions. The assigned priorities would then determine the order in which regions are allowed to make transitions.

Interaction between orthogonal regions

Unlike the orthogonality examples we have seen until now, there are typically some interactions between orthogonal regions. These can take the form of:

- A transition in one region raising an event, a transition in another region responding to this event.
- A transition's guard condition in one region checking whether a state of another region is part of the configuration.
- A transition's actions assigning a new value to a variable, and this value being read by a transition (guard or actions) in another region.

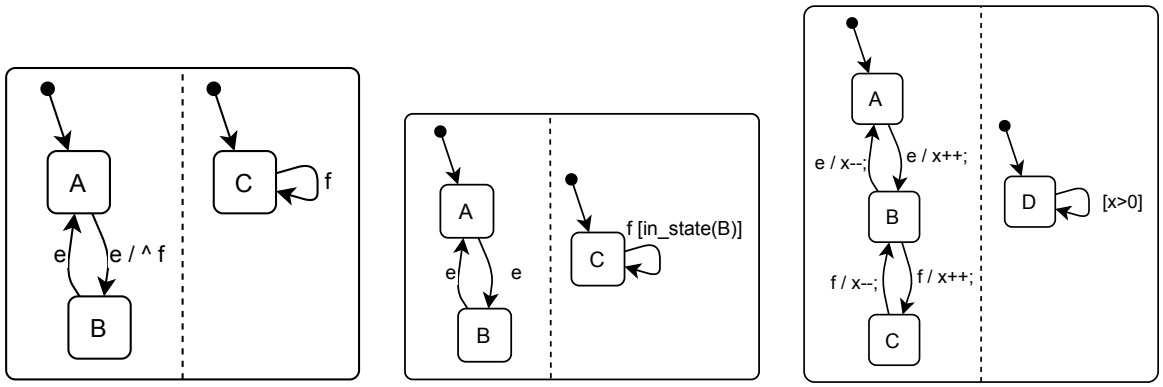
2.1.5 Statechart interface

During its execution, a statechart usually interacts with other parts of software. A statechart is controlled by the software that its delivers inputs, and a statechart can control software (or hardware) that responds to its outputs.

We have seen before that a statechart can only react if it receives an input event. However, during its reaction to an event, a statechart may receive inputs in other ways. We have also seen that a statechart may produce output events during its reactions, but this is also just one of the ways a statechart can produce output.

A statechart's interface may consist of sets of the following elements, usually explicitly declared:

- Input/output events
- Input/output variables
- Synchronous functions



(a) Internal event (b) "In state"-check (c) Variables

Figure 2.15: Different ways of communication between orthogonal regions

Input/output events

Input events have already been covered, they are what causes a reaction of the statechart. *Output events* are received by the environment the statechart interacts with, at the end of a statechart's reaction. They are useful for asynchronous outputs, as the receiver does not have to respond or acknowledge their reception.

Input/output variables

A statechart may also declare a number of *input variables*, that it can read *while reacting* to an input event. Input variables are useful when a changing value is of interest to the statechart, but the changing of the value itself does not require a reaction of the statechart (i.e. should not be an event): The statechart decides when to read the value. Likewise, *output variables* can be useful when a statechart regularly updates some value, but the environment decides when to read the value.

Synchronous functions

With synchronous functions, a statechart can call a piece of code that is implemented outside of the statechart. Functions are called during the execution of a transition, and are blocking: the transition cannot complete before the function returns control to the statechart, possibly with a *return value*. The statechart can use functions for querying information (input) or signaling (output), or both.

Because of their flexibility, synchronous functions are very useful for letting a statechart control another piece of software.

Synchronous functions can serve as a primitive for

- Output events: if the function called queues a generated event and immediately returns control.
- Input/output variables: if the function called is a getter/setter.

although "native" variables may be more efficient than getters and setters.

2.1.6 Execution traces

In order to get insight in a statechart, for debugging, logging, profiling, ..., we may be interested in the history of a statechart's execution.

The execution of a statechart is a sequence of reactions to sets of input events. Every reaction may also contain a set of produced output events. The sequence of inputs and outputs is the *I/O trace* of the statechart, and is the result of looking at a statechart as a *black box*. We may also look closer to see what is happening *inside the statechart*. The most obvious trace would be the sequence of transitions that were fired, from which we can restore a *state trace*.

2.2 Big-Step Modeling Languages

Big-Step Modeling Languages (BSML) [9] are a family of modeling languages that may perform several transitions in response to an input. These languages were mapped onto a common syntax, which is pretty much the syntax of statecharts. Within this common syntax, the semantics of languages in the BSML family are represented as a large feature diagram of semantic options, with at the highest level 8 orthogonal semantic aspects, i.e. independent categories of semantic choices to be made.

Because the family of BSML languages includes statecharts, (a subset of) the semantic aspects of BSML can serve as the basis for describing the possible semantics of statecharts accurately.

In this section, we will discuss the 8 semantic aspects of BSML, and for each of them, determine whether they are applicable to statecharts. Following the conventions from [9], we will use Small Caps Font for the names semantic options.

2.2.1 Syntax

The syntax of BSML, called the Composed Hierarchical Transition System (CHTS) syntax, is in many ways identical to the statechart-syntax discussed in the first half of this chapter. A model in BSML has a state tree of Or-, And- and basic-states, with transitions between any of the states. Transitions can have event triggers, guard conditions and actions. Actions are (1) variable assignments, and (2) the raising of internal and output events.

BSML syntax is more "primitive" than statecharts because it has **no explicit syntax for timed transitions, no enter- or exit-actions and no history**. It doesn't forbid these constructs, it is just that these constructs do not fundamentally touch on the semantic choices that can be made by a BSML language.

BSML also considers many syntactical features optional, such as variables, events, and explicit differentiation between input/internal/output events, but we assume a full-featured syntax.

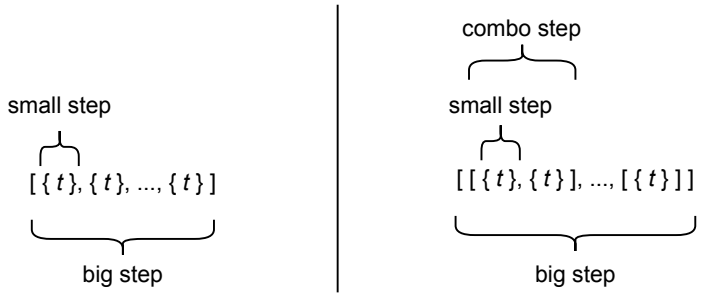


Figure 2.16: Left: Big-step as a sequence of small-steps. Right: Big-step as a sequence of combo-steps.

2.2.2 Fundamental semantics

Before we explore the semantic options of BSML, we must first explain the "fundamental" semantics of BSML, i.e. the semantics that are always the same.

The execution of a BSML is a sequence of *big-steps*. A big-step is a reaction to a set of input events. A big-step consists of a sequence of *small-steps*. A small-step is a set (unordered) of transitions that were *concurrently* fired. The meaning of "concurrently" here is highly specific, and unless Concurrency-semantics are used (discussed later), one may think of a small-step as a single transition. Depending on the semantic options chosen, sequences of small-steps may also be grouped into *combo-steps*. A big-step then consists of a sequence of combo steps. See Figure 3.22.

The semantic options of BSML only determine how a big-step is executed. BSML says nothing about the wider scope of an execution runtime, such as scheduling and queueing of input events, as is often done in statechart implementations.

2.2.3 Big-Step Maximality

The first semantic aspect is *Big-Step Maximality*, which limits the transitions that can be fired together within a big-step. Options are:

Take Many No limitation: The big-step may consist of as many transitions as possible (other semantic aspects may still limit the number of transitions that *can* be taken).

Combined with orthogonality, this semantic option usually lets orthogonal regions take turns in trying to fire a transition. This behavior is called *fairness*.

A hazard of this semantic option is never-ending big-steps. Because of this, Take Many is usually combined with other semantic options limiting the number of transitions that can be made (such as Event Lifeline, see later).

Take One No transitions with *overlapping arenas* are allowed to be taken together within a big-step. The arena of a transition is the lowest common ancestor of source and target that is an Or-state. Arenas are overlapping if they are the same state, or if one is an ancestor of the other.

For non-orthogonal statecharts (hierarchy allowed), this option means that only a single transition can be taken within a big-step. For orthogonal statecharts, this means that a transition can be taken within each orthogonal region (if both the source and destination of the transitions are in the same region).

Syntactic No transitions with *overlapping arenas and whose target states are marked as "stable"* can be taken together in a big-step. This option requires additional syntax for marking states as "stable".

This option is similar to Take One, but only transitions with a stable target state "count", and mark the transition's arena as "used".

This option may also lead to never-ending big-steps, if a state in a region can be re-visited during a big-step without passing through a stable state.

This option may be useful to model "checkpoints" in a sequence of algorithm steps as (non-stable) states (Figure 2.3).

Ordering these options by the number of transitions allowed in a big-step, in *any model*, would yield Take Many $\geq$ Syntactic $\geq$ Take One.

While executing a big-step, the "limiting" options (Take One and Syntactic) work by constraining the set of enabled transitions, based on transitions that were previously executed. A big-step ends when no more transitions can be executed.

Example

Figure 2.17 shows a statechart with 2 orthogonal regions. Suppose we are in configuration $\{A, D\}$ and some input is received (triggering a reaction in the statechart). The labels on the transitions are not event triggers, they are to refer to the transitions. None of the transitions have event triggers or guards, meaning that they are enabled whenever their source state is active. States C and E are marked stable, which only affects behavior when the Syntactic option is chosen. We also suppose the region on the left is always allowed to fire a transition first:

- If we choose Take Many with fairness, the big-step would be the infinite sequence $[\{t1\}, \{t3\}, \{t2\}, \{t4\}, \{t3\}, \{t4\}, ...]$ because of the "loop" of transitions in the region on the right.
- If we choose Take One, the big-step would be the sequence $[\{t1\}, \{t3\}]$, yielding the configuration $\{B, E\}$. Transitions $t2$ and $t4$ can no longer be fired, because their arenas overlap with those of $t1$ and $t3$, respectively.
- If we choose Syntactic, the big-step would be $[\{t1\}, \{t3\}, \{t2\}]$, yielding the configuration $\{C, E\}$. The transition $t4$ can no longer be fired, because by executing $t3$, a stable state was entered, preventing its arena (the region on the right) from executing any more transitions.

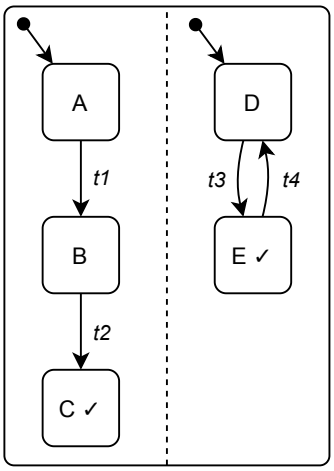


Figure 2.17: Example statechart for Big-Step Maximality

2.2.4 Combo-Step Maximality

As we have seen, combo-steps are an additional grouping of small-steps within a big-step. Combo-steps by themselves serve little purpose, but other semantic options will give meaning to them.

The semantic aspect *Combo-Step Maximality* is optional: if there is no need for combo-steps, no choice needs to be made here.

The options for this aspect limit the transitions that can be taken together in a combo-step, similar to the options for Big-Step Maximality:

Combo Take Many Exactly the same as Take Many.

Combo Take One Exactly the same as Take One.

Combo Syntactic Exactly the same as Syntactic, but instead a "stable" state here is marked as being "combo-stable", another additional syntactic construct.

Note that combining Take Many with Combo Take One is a way to introduce fairness to Take Many: A combo-step would end when every orthogonal region has had a chance to fire a transition, but at the level of the big-step, this would repeat itself until no more transitions are enabled. These two options are often combined in statechart implementations.

Note that combining Take One with any Combo-Step Maximality option would be quite useless, as the limitation of Take One would also apply within every combo-step, resulting in Combo Take One-semantics, and every big-step consisting of only a single combo-step.

Example

For the statechart in Figure 2.17:

- Choosing Take Many and Combo Take One would result in the endless big step $[[\{t1\}, \{t3\}], [\{t2\}, \{t4\}], [\{t3\}], [\{t4\}], ...]$.
- Choosing Syntactic and Combo Take One would result in the big step $[[\{t1\}, \{t3\}], [\{t2\}]]$.

In both these examples, the same transitions are taken as without combo-step semantics, and the same configuration results. However, other semantic options may refer to combo-steps, as we will now see.

2.2.5 Event Lifeline

Options for the *Event lifeline* semantic aspect determine *when* raised events in a big-step become present, and for *how long*. As long as a raised event is present, it may enable transitions. Event Lifeline is therefore a way to control when, and for how long a raised event can trigger transitions.

Option are:

Present in Whole A raised event is present throughout the entire big-step, even from before the point where it was raised. An event either is, or isn't present in a big-step. This option comes from synchronous programming languages such as Esterel.

Present in Remainder A raised event becomes present right after the small step in which it was raised, and remains present for the rest of the big-step.

Present in Next Combo-Step A raised event is present for the entirety of the next combo-step. This option (often combined with Combo Take One), and has the advantage of separating input events from internal events in separate "rounds".

Present in Next Small-Step A raised event is present only in the next small-step, allowing at most one transition to be made as a reaction to the event.

Present in Same A raised event is present in the same small step as where it was raised. This option is only useful in combination with Concurrency-semantics (allowing multiple transitions within a small step).

Figure 2.18 visually shows the times and durations of the various options.

Event Lifeline options can be separately applied to input events and internal events: For instance, we could choose Present in Whole for input events, and Present in Next Combo-Step for internal events, but often, the same option is chosen for both.

For input events, there is no difference between Present in Whole and Present in Remainder, and we often rename the options Present in Next Combo-Step and Present in Next Small-Step to Present in First Combo-Step and Present

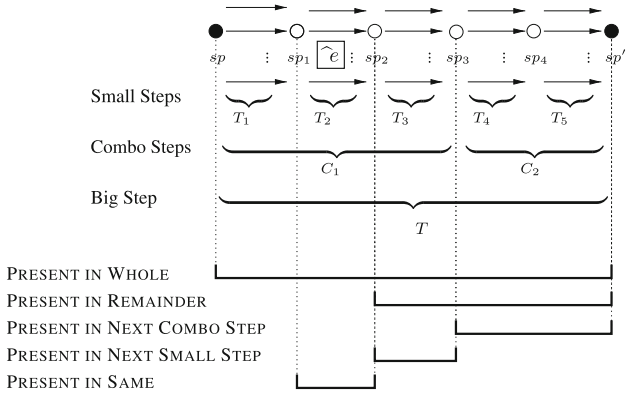


Figure 2.18: Event lifeline options
Figure taken from [9].

in First Small Step, respectively. Present in Same is not a valid option for input events.

The options Present in Whole and Present in Remainder are usually not combined with Take Many, because of the possibility of never-ending big-steps.

Present in Next Combo-Step Present in Next Small-Step allows at most one transition to be made as a reaction to a raised event.

Queueing of (internal) events

Many statechart implementations queue input events, and send them to the statechart one by one. This type of queueing is compatible with BSML, but not a part of it: The statechart simply performs a big-step as a reaction to each input event.

However, there are also statechart implementations that queue internal events. The benefit is that only a single internal event is handled at a time. A statechart implementation may put internal events in the same event queue used for input events (e.g. Rhapsody [11]), or maintain a separate, higher priority queue for them (e.g. YAKINDU [6] in event-driven mode). With the former, reactions to internal events may be interleaved with input events, while with the latter, the reaction to an input event is atomic, containing also the reactions to internal events.

BSML defines no semantic options for the queueing of internal events. We could define at least the following options to extend BSML:

Queue Internally raised events are added to the statechart's input event queue, and will therefore be reacted to in a later big-step.

Combo-Queue Internally raised events are pushed to a special FIFO queue, only meant for input events, and a new combo-step is started for each popped event from the queue, with only that event being present, until the queue is empty.

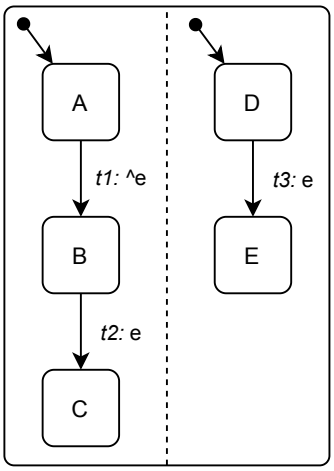


Figure 2.19: Example statechart for Event Lifeline

Example

In the statechart in Figure 2.19, suppose a big-step starts by firing the transition $t1$, raising the (internal) event e.

- If we choose Take Many (with fairness) and Present in Remainder, the big-step is $[\{t1\}, \{t3\}, \{t2\}]$.
- If we choose Take Many and Present in Next Combo-Step along with Combo Take One, the big-step is $[[\{t1\}], [\{t2\}, \{t3\}]]$.
- If we choose Take Many (with fairness) and Present in Next Small-Step, the event e can only trigger a single transition, and the big-step is $[\{t1\}, \{t3\}]$.

External Events

Event Lifeline also defines options (independent to the options listed above) determining how to distinguish input and output events from internal events. For statecharts, we assume this is done syntactically, so we aren't interested in any other options here.

Interface Events

Event Lifeline also defines options (independent to the options listed above) on interface events. Interface events are events used for communication between *components* of a model. The components here are actually models on their own. For statecharts, there are no interface events: a statechart may send an event to another statechart, but such events are not treated different from output/input events.

2.2.6 Memory Protocol

For memory protocol, 2 independent semantic aspects exist: *Enabledness Memory Protocol* and *Assignment Memory Protocol*. Both have the same options, having effect on the

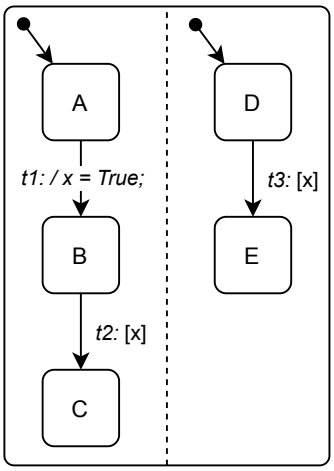


Figure 2.20: Example statechart for Memory Protocol

values of variables that are read during guard condition evalution (enabledness) or during execution of a transition's actions (when assigning new values to variables), respectively. Usually the same option is chosen for both aspects.

The options are:

Big Step Values of variables are read as they were at the beginning of the big-step.

Combo Step Values of variables are read as they were at the beginning of the combo-step.

Small Step Values of variables are read as they were at the beginning of the small-step.

By choosing Combo Step, transitions within the same combo-step cannot see each other's effects on the statechart's variables. If also Present in Next Combo Step is chosen for Internal Event Lifeline, then transitions within a combo-step truly have no influence on each other, apart of course from the changes they make to the statechart's configuration.

Similarly, with Big Step, all transitions in a big-step cannot see each other's effects on the variables. This option is likely to be more useful when big-steps are small, e.g. when Take One is chosen.

Choosing Small Step causes the most actual version of variables' values to be read, effectively "disabling" the memory protocol.

Example

For the statechart in Figure 2.20, suppose we choose Take Many and Combo Take One. Transition $t1$ is the first one to execute in a big-step.

- If we choose Big Step, then no other transition can fire. The big step is $[[\{t1\}]]$.

- If we choose COMBO STEP, then the big step is $[[\{t1\}], [\{t2\}, \{t3\}]]$.
- If we choose SMALL STEP, then the big step is $[[\{t1\}, \{t3\}], [\{t2\}]]$.

Interface Variables

Just like interface events, some BSML languages feature interface variables for coummnication between *components*. Statecharts do not have such "components", so we will not discuss semantic options for interface variables.

2.2.7 Order Of Small-Steps

The semantic aspect *Order of Small Steps* determines the order in which small-steps (transitions) are executed, if more than one of them can be executed, independently. For statecharts, this is when transitions are enabled in multiple orthogonal regions. Options are:

None Small-steps are unordered: an order is chosen non-deterministically.

Explicit Ordering Introduces additional syntax for explitly ordering small-steps. In a statechart, this would be done by assigning a priority relationship between orthogonal regions at the same level.

Dataflow Small-steps are ordered such that a transition assigning a value to a variable, happens before a transition reading that variable.

Since a variable is used to communicate information from one small-step to another, this option should only be combined with SMALL STEP for Memory Protocol semantics.

For statecharts, the non-determinism of NONE makes it an unsuitable option. Almost always, EXPLICIT ORDERING is used. DATAFLOW is an interesting option, but in many statecharts, may only yield a *partial ordering* between orthogonal components. Since transitions in a statechart can also make synchronous calls during their execution, to functions implemented outside of the statechart, and perhaps an ordering is desired on these function calls, the statechart itself would contain not enough information to execute the transitions in their right order.

Example

In all previous examples, we assumed EXPLICIT order of small-steps, giving higher priority to orthogonal regions from left to right.

2.2.8 Priority

Highly related to the previous semantic aspect, *Priority* deals with all other situations where multiple transitions could be selected for inclusion in a small-step. The options for this semantic aspect can be combined arbitrarily to make the model behave deterministically.

Hierarchical This is actually a *collection* of semantic options that assign priority according to some hierarchical property of a transition. Most commonly, the nested depth of the source state of the transition determines its priority: The sub-options Source-Parent and Source-Child create a priority relationship between transitions whose sources are ancestors of one another, preferring "parent" (ancestor) or "child" (descendant), respectively.

Other options mentioned in [9] are Arena-Parent and Arena-Child, creating a priority relationship between transitions whose arena's are ancestors of each other.

Explicit Priority Introduces additional syntax to explicitly order transitions. In statecharts, this option is often chosen to create an ordering between the outgoing transitions of a state.

Negation of Triggers If 2 transitions respond to different events, and both events are present, one transition can be given a higher priority adding the *negated event* of the other transition to its trigger.

Relation to Order of Small-Steps

All of the semantic options for Priority create a *partial ordering* between transitions. Order of Small-Steps also does this.

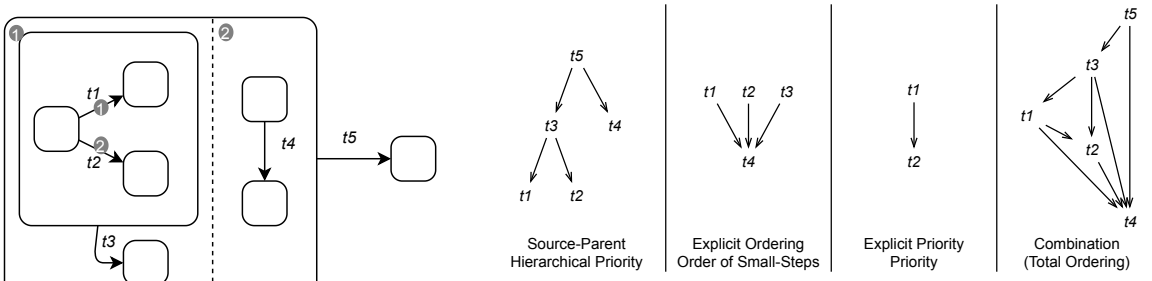
In a statechart, transitions can only be simultaneously enabled in 3 ways:

1. They have the same source state.
2. One's source is an ancestor of the other's source.
3. One is orthogonal to the other, i.e. the transitions have non-overlapping arenas.

Note how (1) was mentioned when discussing flat statecharts, (2) arises when we introduce hierarchy, and (3) arises when we introduce orthogonality.

Explicit Priority can create an ordering between transitions of type (1). The priority options Source-Parent and Source-Child create an ordering between transitions of type (2). Order of Small-Steps creates an ordering between transitions of type (3).

An apparent difference between Order of Small-Steps and Priority, is that Order of Small-Steps defines an order on small-steps (executing all small steps), while Priority chooses a single transition among a set of candidates that cannot be executed together, because they disable each other. At least for statecharts, this distinction is not quite correct, because simultaneously enabled orthogonal transitions may still disable each other in unpredictable ways (due to action language code). So while a priority relation is defined between orthogonal transitions, a statechart implementation cannot (fully) predict which subset of these transitions is going to fire. All it can do, is select 1 highest-priority enabled transition at a time, execute it, and repeat until no more transitions can be executed as part of the combo- or big-step, following the rules of Big-/Combo-Step Maximality



(a) Statechart

(b) Orderings
Meaning of an arrow: "has higher priority than"

Figure 2.21: A statechart and the partial orderings between its transitions according to various semantic options. Together they produce a total ordering.

(+fairness), Event Lifeline and Memory Protocol semantics. An "order" of small-steps arises in retrospect. A careful conclusion is that Order Of Small-Steps is really just a collection of Priority-options.

The partial orderings of Priority and Order of Small-Steps are statically known, and if the combination of them yields a total ordering on all transitions, then the statechart behaves deterministically. This total ordering may be constructed explitly, as a list of transitions from high to low priority. The first enabled transition in the list will always be the next transition to fire.

Example

Figure 2.21 shows how SOURCE-PARENT and explicit ordering of orthogonal regions along with explicit ordering of outgoing transitions of the same state yields a total ordering $[t5, t3, t1, t2, t4]$, meaning that if e.g. $t3$ can only fire if $t5$ is not enabled. $t4$, the lowest-priority transition, can only fire if all other transitions are disabled, which wil be the case if, e.g. TAKE ONE is chosen and a transition in the orthogonal region on the left has already fired during the big-step.

2.2.9 Concurrency

Concurrency allows small-steps to consist of multiple transitions. Transitions within a small step logically happen concurrently, but implementations will typically execute them in some (potentially non-deterministic) order.

Single Every small-step consists of 1 transition.

Many A small step may consist of more than 1 transition.

If MANY is chosen, additional options may be chosen for *Small-Step Consistency* and *Preemption*.

Small-step consistency options:

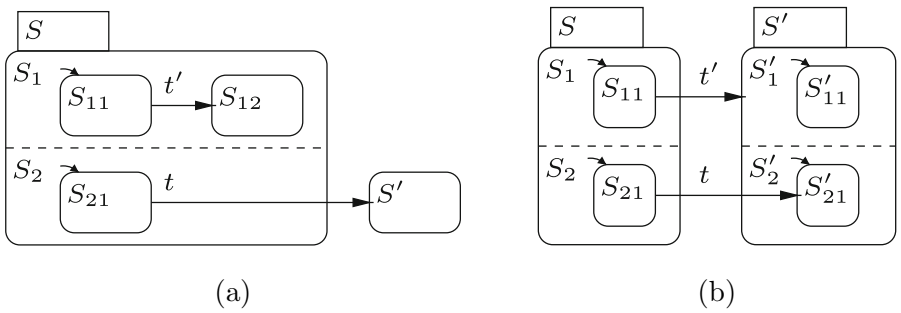


Figure 2.22: Interrupting transitions
Figure taken from [9]

Arena Orthogonal Transitions can be taken together if their arenas do not overlap (similar to Take One for big-steps)

Source/Destination Orthogonal Transitions can be taken together if their sources and destinations are respectively orthogonal, i.e. the lowest common ancestor of the sources and of the targets is an And-state.

There are no BSML languages making use of this option, but it is mentioned in [9] as "interesting".

Preemption options:

Non-Preemptive Transitions that are an *interrupt* for one another can be taken together. One transition t is an interrupt for another transition t' if both transitions' sources are orthogonal to each other, and (a) the target of t' is orthogonal to the source of t, but the target of t is not orthogonal with the sources of either transition; or (b) the target of neither transition is orthogonal with the sources of the two transitions, but the target of t is a descendant of the target of t'. Both cases are shown in Figure 2.22.

Preemptive Transitions that are an interrupt for one another cannot be taken together. No priority is assigned to the interrupting or the interruptee transition.

Note that, at least for statecharts, the semantics of Many in combination with Arena Orthogonal and Preemptive can also be achieved by choosing Combo Take One and the Memory Protocol option Combo-Step, if Present in Same is not used. It is only when the Source/Destination Orthogonal or Non-Preemptive semantic options are desired, that Many is required.

2.2.10 Applicability to statecharts

Of the semantic aspects seen, most are applicable to statecharts. We decided not to support the following options:

- Event Lifeline: PRESENT IN WHOLE, PRESENT IN SAME

  Both options can create *non-causal* big-steps, i.e. big-steps containing transitions that were triggered by transitions executed *after* them. This may lead to transitions being fired because they enable *each other*, instead of being fired as a direct or indirect result of the set of inputs. This can be counter-intuitive, and may also be complex to implement. [9] states that static detection of the possibility of non-causal big-steps is undecidable for languages supporting variables, such as statecharts.

- Order of Small-Steps: NONE, DATAFLOW

  NONE would result in non-determinism, which is non-desirable in statecharts. DATAFLOW would only yield a partial ordering between orthogonal transitions in many statecharts, while transitions in a statechart can affect each other in more ways than just through variable assignments.

- Concurrency: MANY

  Allowing concurrency adds little, because combo-steps can be used to hide the effects of "concurrent" transitions from one another. Concurrency would allow for the SOURCE/DESTINATION ORTHOGONAL option, but the authors of [9] were not aware of any languages using it. Concurrency would also allow for the NON-PREEMPTIVE option, but this may be difficult to implement, and confusingly go against other restrictions chosen on what transitions are allowed to fire together in a big/combo-step.

CHAPTER 3

Implementation: The SCCD Runtime

This chapter discusses the main artifact of our work: our implementation of a statechart execution runtime with configurable semantics.

We first explain how our implementation differs from the original SCCD project, from which it was forked. We then introduce the basic design of our implementation, which consists, among other things, of a simple action language, the statechart language with semantic variability, and, on top of it, the SCCD language and its event-queueing execution, implemented in what we call "the Controller". We then explain the detailed design and implementation of each of those languages, giving a deep insight into our solution. Finally, we go back to discuss and motivate changes made from the original SCCD project, at a more detailed level.

3.1 Current State of SCCD

The original implementation of SCCD [17] was a statechart & class diagram compiler + execution runtime library, which supported a (smaller) subset of BSML semantic configurations as well.

At the highest level, an SCCD model is a *class diagram* where the behavior of the classes is defined as a statechart. The class diagram has a single *default class*, which is instantiated when the model is instantiated. This instance may *dynamically create* new instances of other classes (statecharts). The relationships between instances (e.g. multiplicities) are modeled in the class diagram and enforced during execution.

It generated executable code in a number of target languages. The supported target languages were Python, JavaScript and C# [8].

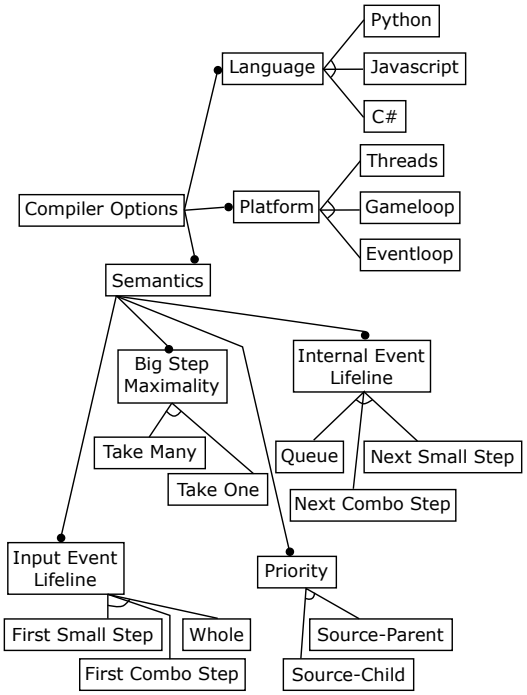


Figure 3.1: Feature diagram of original SCCD
Figure taken from [17]

Semantic configurations were limited to Big-Step Maximality (Take One, Take Many), Internal+Input Event Lifeline (Next Small Step, Next Combo Step, Present in Remainder, Queue) and Priority (Source-Child, Source-Parent).

3.1.1 SCCD in this thesis

The SCCD discussed in this thesis is a fork from the original SCCD. The most important "functional additions" to original SCCD are the following:

More semantic options The main goal of this thesis. On top of the semantic options supported in original SCCD, the following options were added: Big-Step Maximality: Syntactic, Combo-Step Maximality: Combo Take One, Combo Take Many, Combo Syntactic, Memory Protocol: Big Step, Combo Step, Small Step, Priority: Arena-Parent, Arena-Child.

Action language In the original SCCD, action code had to be written in the same language as the target language of compilation, making models non-portable. This was only a temporary solution, the plan was to add an action language eventually. Another reason for integrating an action language is to have precise control over Memory Protocol semantics. Therefore, SCCD now has a built-in (textual) statically type-checked action language.

Under the hood, SCCD has changed so much, that little resemblance with the original remains. Most importantly, SCCD is no longer a compiler (code generator), but an execution runtime, which loads models in an XML format and executes them. A detailed discussion and motivation for the changes made, follows at the end of this chapter, in Section 3.6.

3.2 Overview of Implementation

The SCCD runtime consists of 3 languages: the action language, the statechart language, and the SCCD language. The action language is part of the statechart language, and the statechart language is part of the SCCD language. Apart from these 3 languages, a few libraries are included. The dependency graph between SCCD's packages reflects this (Figure 3.2).

An overview of the directory structure (packages) at the highest level:

`action_lang` Implementation of the action language parser, syntax and semantics.

`statechart` Implementation of the statechart parser, syntax and variable semantics. Naturally depends on `action_lang` since the action language is part of the statechart language as well.

`cd` Placeholder implementation of the "class diagram" (SCCD language) parser and syntax. A model in SCCD is always a class diagram, with the behavior of each class defined as a statechart. Each class diagram also has a *default class* defined, which is initialized at startup, just like e.g. the "main class" in a Java program. In this thesis, all SCCD models consist of only a single default class, whose behavior is defined by a statechart.

`controller` Implementation of the Controller, the primitive for executing SCCD models.

`realtime` An optional library, wrapping around the Controller, for soft real-time (wall-clock sync'ed) execution of models. Most "real" applications would want to use this. Includes support for event loop integration.

`test` Parser and executor of SCCD tests. An SCCD test consists of an SCCD model, a set of timed input events, and a set of timed output events.

`util` Utility library, used by most of the project. Contains classes like `Bitmap`, `Duration`, etc.

Every one of the 3 languages has a parser, a set of language constructs (the abstract syntax), and an implementation of its execution. These elements are implemented in the

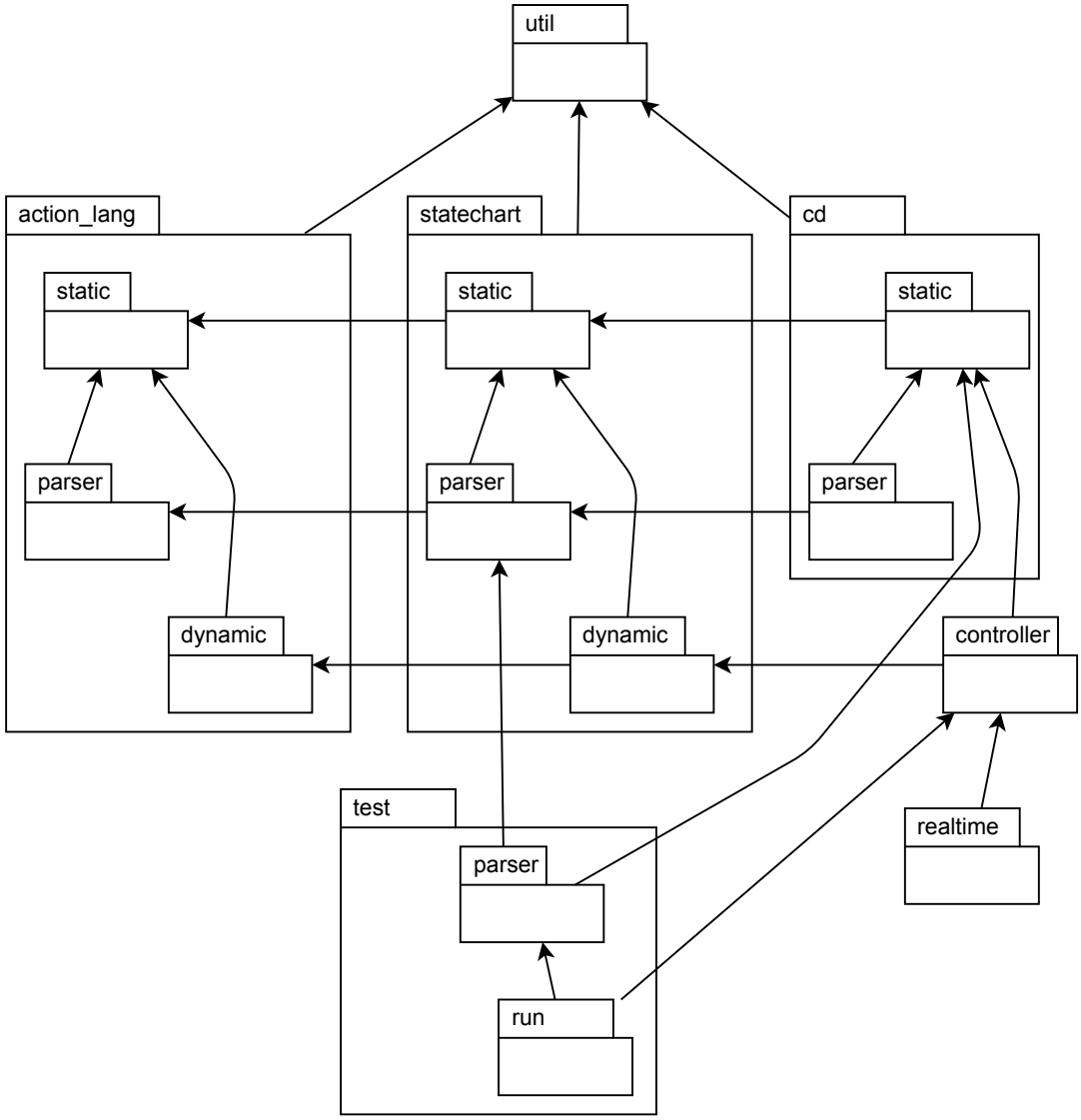


Figure 3.2: Dependencies between packages (directories) of SCCD.
The meaning of an arrow is: "depends on".

sub-packages `parser`, `static` and `dynamic` of each language. The parser and dynamic packages always depend on static: The parser package *produces* instances declared in static; the dynamic package *reads/interprets* instances from static (it does not modify them) in order to carry out execution.

Also note how the the sub-packages of a language on the right depend on their respective sub-packages of a language on the left: E.g. the statechart language's parser-package depends on the action language's parser-package, as the statechart language may include action language fragments, and therefore in order to parse statechart models, it also has to parse action language code, but it does not depend on the action language's dynamic-package, as in order to parse statechart models, it does not have to execute action language code.

Figure 3.2 shows the dependencies between the packages (directories) of SCCD. Note how the `action_lang`, `statechart` and `cd` packages (directories) have the same sub-package structure of `static`, `dynamic` and `parser`. The sub-packages on the right depend on their equivalent on the left. The recurring sub-packages have the following meaning:

`static` Roughly, the *syntactical constructs* of the language. A loaded model is a connected graph of instances of the classes (types) defined in this package. They are constructed by the parser, and some semantic processing steps / checks (e.g. static analysis) are possibly done. Those processing steps are also implemented in this package. Once the model is loaded, the model itself no longer changes.

`dynamic` The types defined in this package are responsible for creating and executing instances of a loaded model. This package only *reads* loaded models (i.e. instances from `static`), it does not modify them.

`parser` The parser logic: a composition of rule-based algorithms for parsing textual (in the case of the action language) or XML (in the case of the statechart and class diagram languages) data into loaded models, producing instances of `static`.

The dependency graph among packages in Figure 3.2 hints that the Controller is the "`dynamic`" part of the "`cd`" package. This is a correct observation, as the Controller *reads* class diagram models and creates and executes instances of it. Initially, the Controller was given its own package to make it prominent. At some point, it may be put in the `cd.dynamic` package.

In the following sections, we will introduce the detailed design of the action language (Sec 3.3), the statechart language (Sec 3.4), and finally, the Controller (or better, the class diagram language top-level execution semantics) (Sec 3.5).

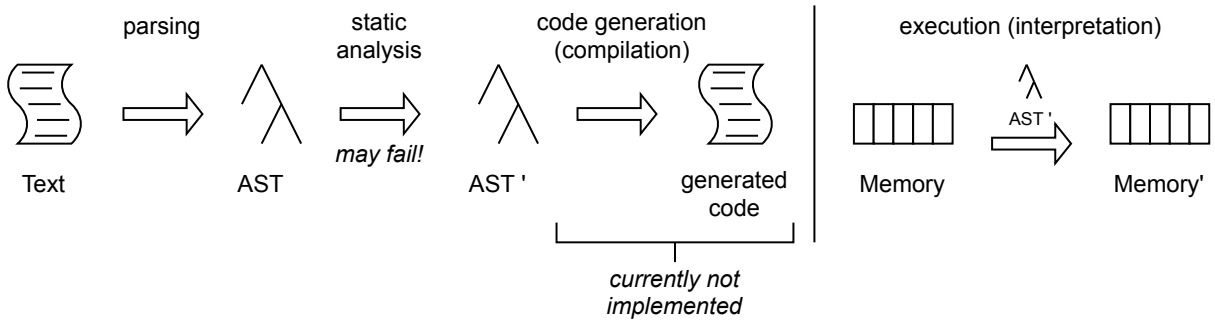


Figure 3.3: Usage of the action language

3.3 Action Language Implementation

The action language is an important new feature in SCCD. In statecharts, action language code is mostly used in a transition's guard condition (an expression) and a transition's action (a block of statements). The integration of an action language makes the statechart language much more powerful, while remaining portable. By using a customly developed language, we have complete control over its features.

The action language is a simple **procedural**, **statically typed** language with a syntax somewhat resembling Python (for expressions) and JavaScript (for function declarations and blocks). It is a language on its own, and can be used independently of other parts of the SCCD project. (An interactive prompt app is included in the project to demonstrate this.) Currently, the language is interpreted, but it would be trivial to add a code generator, since much of the work a compiler does has already been implemented in the static analysis step, such as assigning a memory layout to declared variables.

In this section, the action language itself is the focus, but we may mention statecharts now and then, as a motivation for why e.g. some design decision was made. Figure 3.3 shows how the action language is used. First, the parser constructs an AST of syntactical constructs. Next, a static analysis step "initializes" the AST tree (note the prime symbol). Then, using the initialized AST tree, execution happens as a function transforming memory. We will now briefly discuss the parser and syntactical constructs, followed by static analysis and execution.

3.3.1 Parser and syntactical constructs

The Python library Lark[1] is used to parse action language fragments. Lark constructs a parser from a grammar file. Lark can switch between 2 parsing algorithms: LALR(1) and Early. The former offers better performance, while the latter is easier to write grammars for. We use LALR(1) since it was found to work perfectly well.

The grammar file used by the action language is listed in Appendix A. Apart from the grammar file, the parsing step also uses a "transformer" class, written in Python,

which translates encountered textual constructs to our own syntactical constructs. The transformer class also *desugars* some syntax, e.g. `i += 1` becomes `i = i + 1`.

Figures 3.4 and 3.5 show a class diagram of the syntactical constructs that make up the language. Some constructs are composed of other constructs, and can form a tree structures, called ASTs (abstract syntax trees). The result of parsing a piece of action code is always an AST.

Example: Figure 3.7 shows the AST for the statement `y = 2 * x`. At the root of the AST is the assignment statement. The left-hand side of the assignment is the identifier `y`, which is treated as an LValue. At the right-hand side is the expression `2 * x`, where x is treated as an RValue.

Expressions, statements and LValues

Every construct either implements the `Expression` or `Statement` interface. `Expression` is not a subtype of `Statement`, or vice versa. Expressions can be *evaluated*, statements can be *executed*, and both can have side-effects, such a new value to a variable being written. Expression evaluation always yields a result, statement execution does not.

There is also the abstract class `LValue`, which is a bit special. `LValue` inherits `Expression` because all `LValue` instances can also be treated as expressions (i.e. RValues), but only if they occur in an expression context. Vice versa, an `LValue` instance occuring in an LValue context is never treated as an expression. The only LValue-context currently existing in the action language is the left-hand side of an `Assignment` statement. When an `LValue` type is the root of an AST, it is always treated as an `Expression`.

All constructs have at least 2 methods (See Table 3.1):

1. A *static analysis* method, implementing the static analysis step, which must be executed on every AST before it can be executed.

2. An *execution* (or evaluation) method, implementing the *actionable* part of the language.

We will first discuss static analysis, execution follows later.

	Static Analysis	**Execution**
Expression	init_expr(:Scope): SCCDType	eval(:MemoryInterface): Any
LValue	init_lvalue(:Scope, rhs_type: SCDDType)	eval_lvalue(): int
Statement	init_stmt(:Scope): ReturnBehavior	exec(:MemoryInterface): ReturnValue

Table 3.1: Methods for static analysis and execution. The meaning of these methods is discussed in the subsections on Static Analysis (3.3.2) and Execution (3.3.3).

3.3.2 Static analysis

Static analysis is an important feature of the action language. It is responsible for 2 related things:

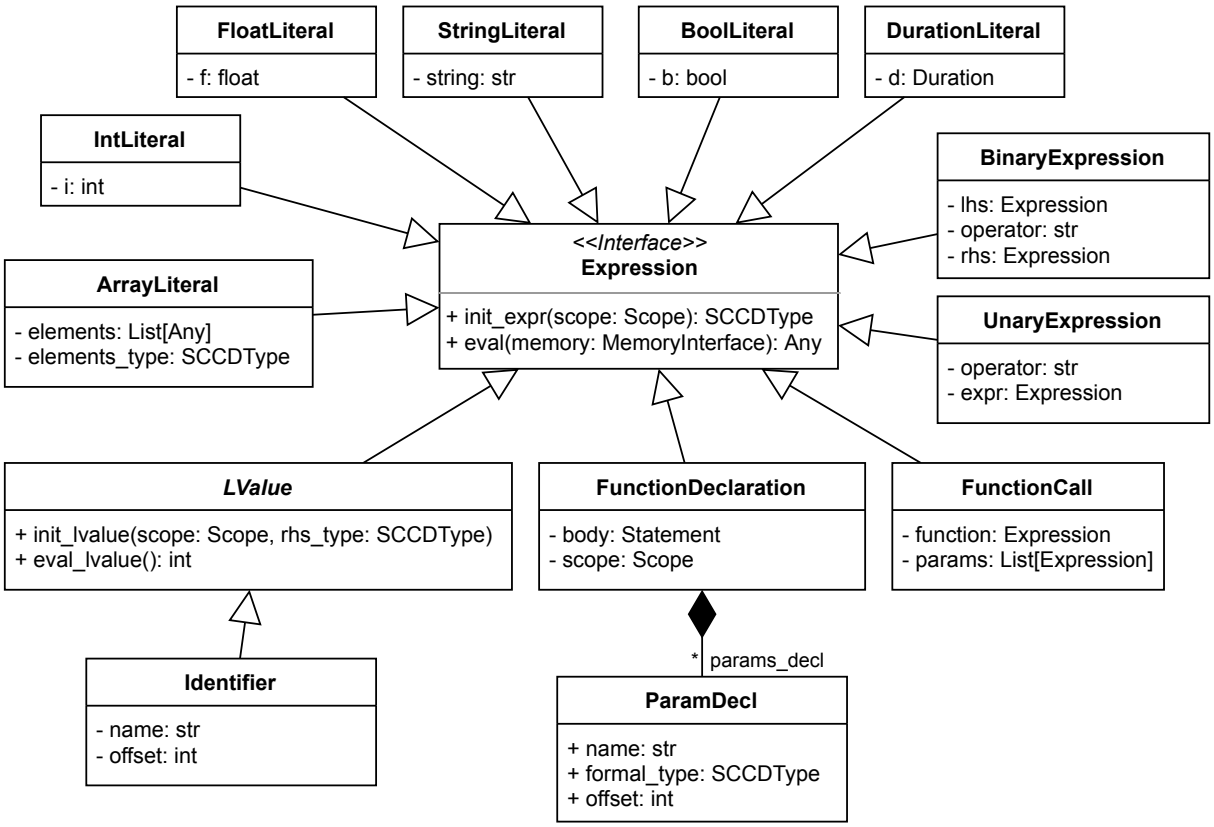


Figure 3.4: Syntactical constructs, Expression type.

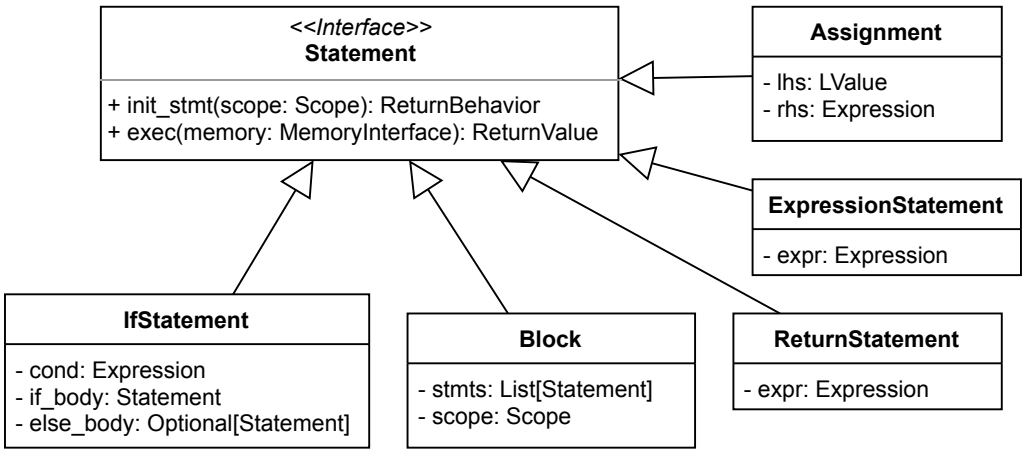


Figure 3.5: Syntactical constructs, Statement type.

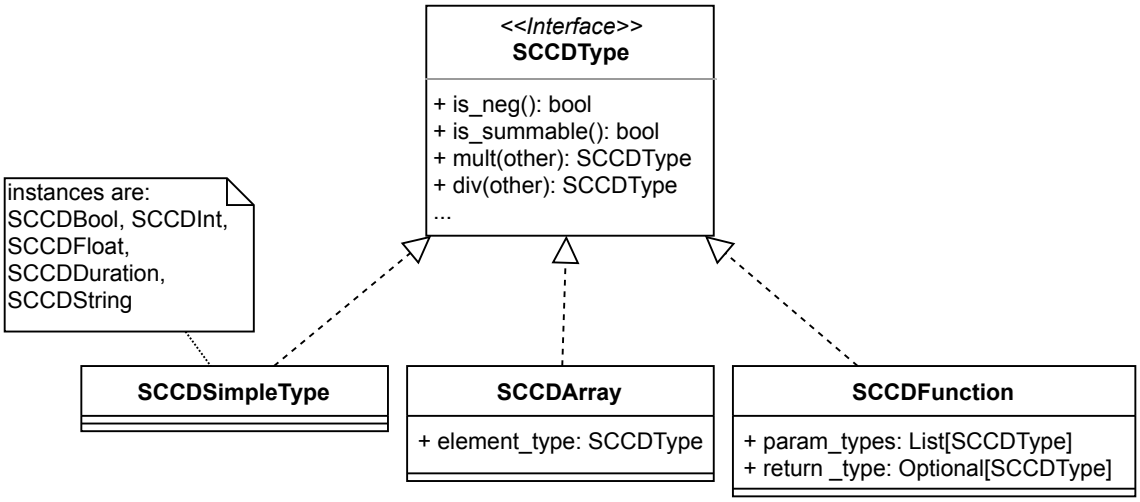


Figure 3.6: Type constructs

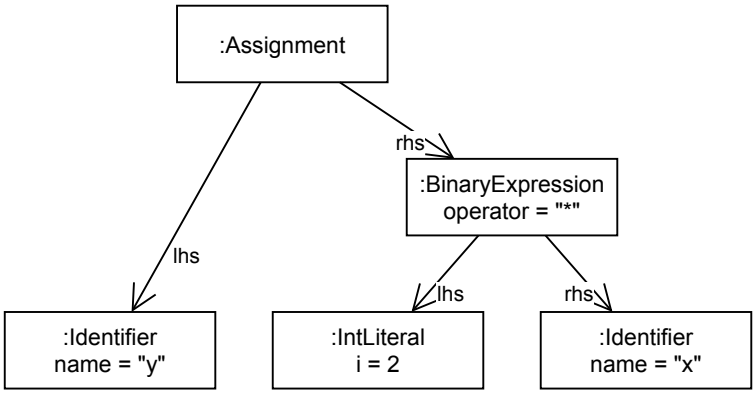


Figure 3.7: AST for statement: `y = 2 * x`

1. Static typing with type inference: Determining the types of all expressions, while checking whether types are compatible in all expressions and statements.

2. Assigning a memory layout to declared variables. Currently, only stack memory is supported, there is no heap. (The execution runtime allocates stack frames on Python's heap, though.)

Static analysis must be done once, after the structure of the AST has been created, and before the AST can be "executed" (interpreted). Static analysis may *fail* (yield an error) if the AST is found to be semantically invalid, e.g. when a type error is found. Static analysis is an (idempotent) operation on the AST, that only adds information to the tree, such as types, memory offsets, and scope objects (see later).

Every construct in the action language has a method implementing the analysis step. To run static analysis on an AST, one calls the static analysis method on the root node of the AST. Nodes that have children implement static analysis by also invoking it on their children. As such, the analysis step is performed on the entire AST.

Depending on whether a node is an `Expression`, `LValue` or `Statement`, the method for performing static analysis is as follows:

Expression: `init_expr(scope: Scope): SCCDType`

Initializes the expression as part of the given scope (more on scopes later), determines and returns the type of the expression.

Typically, expressions use the scope to lookup types and memory offsets of encountered variable names.

LValue: `init_lvalue(scope: Scope, rhs_type: SCCDType)`

Initializes the LValue as part of the given scope, with the given type `rhs_type`.

An LValue may introduce a new variable to the scope, or if the variable already exists, it merely asserts that the types match (assignment-wise), and looks up the memory offset of the existing variable.

Statement: `init_stmt(scope: Scope): ReturnBehavior`

Initializes the statement as part of the given scope, determines and returns the "return behavior" (see later section) of the statement.

Determining expression types

The static analyzer determines the type of every expression. Different strategies are used for different expression types:

Literals (e.g. `IntLiteral`) Trivial, the type is always the same.

For `ArrayLiteral`, the type is Array-of-element-type, and mixed element types are not allowed.

BinaryExpression (e.g. a sum) The type depends on the operator, the left-hand side and right-hand side expressions, e.g. the sum of two integers is an integer.

UnaryExpression (e.g. unary minus) The type depends on the operator and the expression.

Identifier If the identifier occurs in a RValue context, its type is discovered by looking up the variable name in the scope object received during static analysis.

FunctionDeclaration The type is "function type" (Figure 3.6). The formal parameters are type-annotated in the syntax. The return type is inferred (see later).

FunctionCall First, it is asserted that the expression being called is of function type. Then the type is the return type of the function being called.

Variable declaration type inference

A useful feature of the action language is its type inference for declared variables (this section), and return types of functions (next section). Type inference results in less verbose code, saving development and maintenance time.

In many statically typed languages such as C or Java, when declaring a variable, a type must be given. This is not the case for the action language, because of the following principles:

1. The only way to declare a new variable is by *assigning* a value to a name that does not yet exist in the current scope (or parent scopes).

   As a consequence, variables are always initialized with a value, which can prevent certain errors.

2. The type of a variable on the left-hand side of an assignment is inferred from right-hand side of the assignment. The right-hand side is just an expression, and the type of expressions is statically known.

A type annotation is also not optional (like in TypeScript or Haskell): it is simply not part of the language. The only place where type annotations occur, is in the formal parameters of a function declaration. Unlike e.g. Haskell, the action language is not powerful enough to infer the possible types of a function's formal parameters.

Function return type inference

As mentioned before, the return type of a declared function is inferred:

```
inc = func(i: int) {
  return i + 1;
};
```

In the above code fragment, the value assigned to `inc` will be of type `func(int) -> int`, meaning, a function taking an integer as parameter, and returning an integer.

This feature is more complex than one might think, as a piece of code may contain multiple branches. Consider the following fragment, which the static analyzer will reject:

```
inc = func(i: int) {
  if (i < 10)
    return i + 1;
  else
    return "too large" # error: not all branches return the same type
};
```

Different branches return different types, making the return type of the function vary depending on the input. This is not allowed in the action language. Detection of this is implemented by having static analysis come up with a static description of the *return behavior* of every statement (just like static analysis determines the type of every expression). The return behavior is recorded in an object of type `ReturnBehavior` (Figure 3.9), returned from the `Statement.init_stmt` method, which was already mentioned.

The return behavior can be one of:

NEVER The statement never returns, i.e. there are no return-statements in any of the branches.

SOME_BRANCHES The statement contains a conditional branch. One or more branches return a value of the same known type, other branches do not have return-statements.

ALWAYS All of the statements branches contain a return statement, and they all return the same known type.

For SOME_BRANCHES and ALWAYS, the "return type" (e.g. "int") is included in the `ReturnBehavior` object.

For simple statements, such as `Assignment` or `ExpressionStatement` (return behavior: NEVER) and `ReturnStatement` (return behavior: ALWAYS), the return behavior is always the same. For complex types of statements, the return behavior depends on their sub-statements:

`Block` A sequence of statements. The return behavior of a block is calculated with an algorithm that walks over the sequence statements: Initially, the return behavior is NEVER. For each next statement, the "so-far"-calculated return behavior is *sequenced* with the return behavior of that statement. Sequencing means: the return behavior can only go from NEVER to SOME_BRANCHES or ALWAYS, and only from SOME_BRANCHES to ALWAYS. While this happens, as soon as a return type has been established, later statements must return the same type, if they have non-NEVER return behavior, or we throw an error.

This algorithm uses the static method `ReturnBehavior.sequence` (Figure 3.9), which takes to "so-far"-calculated behavior, and the behavior of the "next"-statement to produce a new "so-far"-return behavior.

`IfStatement` A conditionally executed statement, with an optional else-branch. First of all, if there is no else-branch, the return behavior of the else-branch is considered NEVER. Next, the branches are *combined* according to an algorithm: If both branches have the exact same return behavior, that will be the return behavior of the `IfStatement` as well. In all other cases, the return behavior will be SOME_BRANCHES, as at least one of the branches does not return ALWAYS and at least one does not return NEVER. We also check if the returned types match. If not, we throw an error.

This algorithm is implemented in the static method `ReturnBehavior.combine_branches` (Figure 3.9).

It is clear that for a function body, only NEVER and ALWAYS are allowed, because whether a function returns something, and the type of what is returned, must always be the same.

The SOME_BRANCHES option thus is illegal for function bodies, but it is allowed in other parts of the AST. For instance, look at the code fragment and its AST in Figure 3.8. The `IfStatement` in the AST has return behavior SOME_BRANCHES, meaning it may or may not return (an integer). However, it occurs in a `Block` (which is a sequence of statements), where it is followed by a `ReturnStatement`, which has return behavior ALWAYS, meaning it always returns (an integer). As a result, the `Block` itself also has return behavior ALWAYS, and as such, is a valid function body.

Scope object

As we have seen before, the static analysis method of every construct expects a reference to a `Scope` object as parameter. A scope object primarily serves to lookup and declare variables, containing their types, const-ness and names. It is also a static description of the memory layout of a stack frame during execution, containing the memory offsets of variables in the frame. See Figure 3.9 for a UML diagram of the Scope class.

During static analysis, most types of constructs (expressions, statements) simply pass the `Scope` object on to their children. This means the scope did not change.

Example: Figure 3.10 shows static analysis performend on the AST of Figure 3.7. The identifier "`x`" is analyzed as an RValue (`init_expr`), and uses the `Scope` object to lookup the memory offset of its variable, and finds that it is 0. The identifier "`y`" is analyzed as an LValue (`init_lvalue`) and uses the `Scope` object to attempt to "put" a variable `y` of type `SCCDInt`. The operation succeeds, and `y` is given memory offset 1.

In some cases, the scope does change: A function declaration introduces a new scope for the function's body. This means that variables declared in functions are not visible outside of that function declaration. Function bodies can access (read/write) variables of surrounding scopes, however. (A function declaration is just an expression, so function

Code fragment:

```
inc = func(i: int) {
  if (i < 10)
    return i + 1;
  return 0;
};
```

AST:

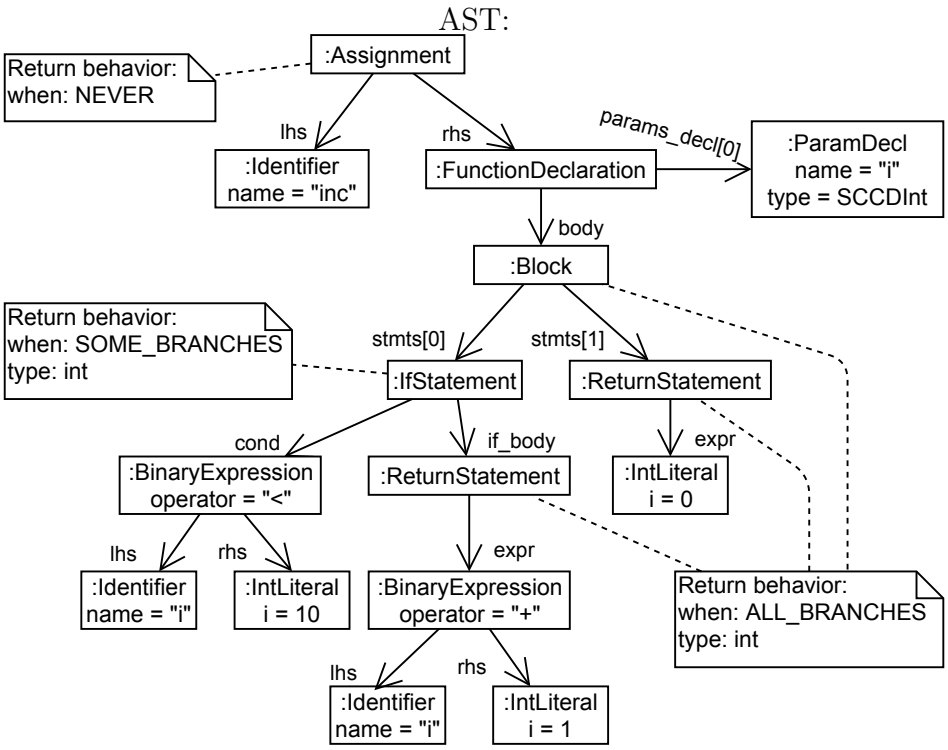


Figure 3.8: Code fragment and its AST, with return behavior annotated for all statements

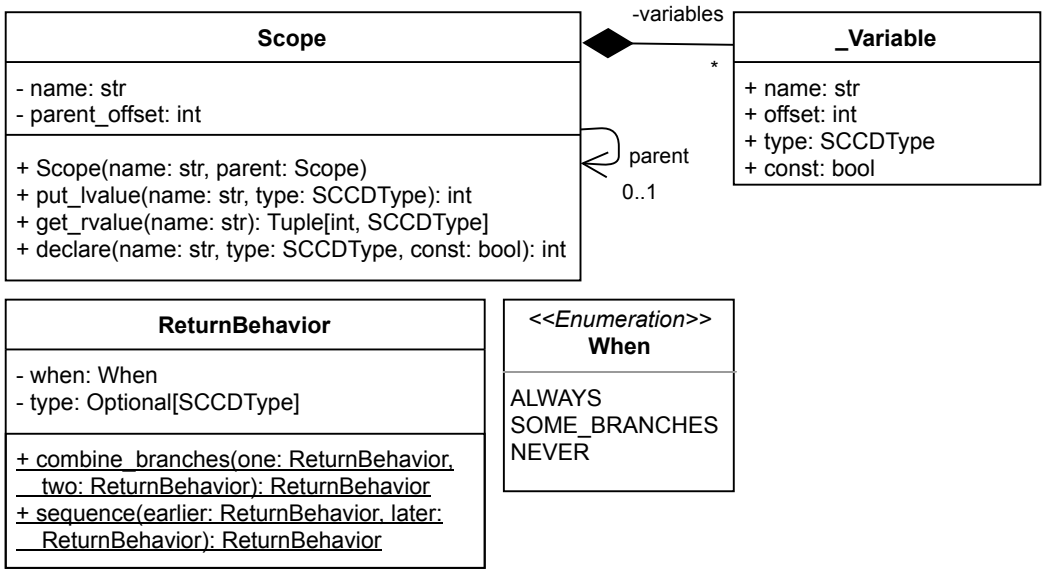


Figure 3.9: Classes involved in static analysis

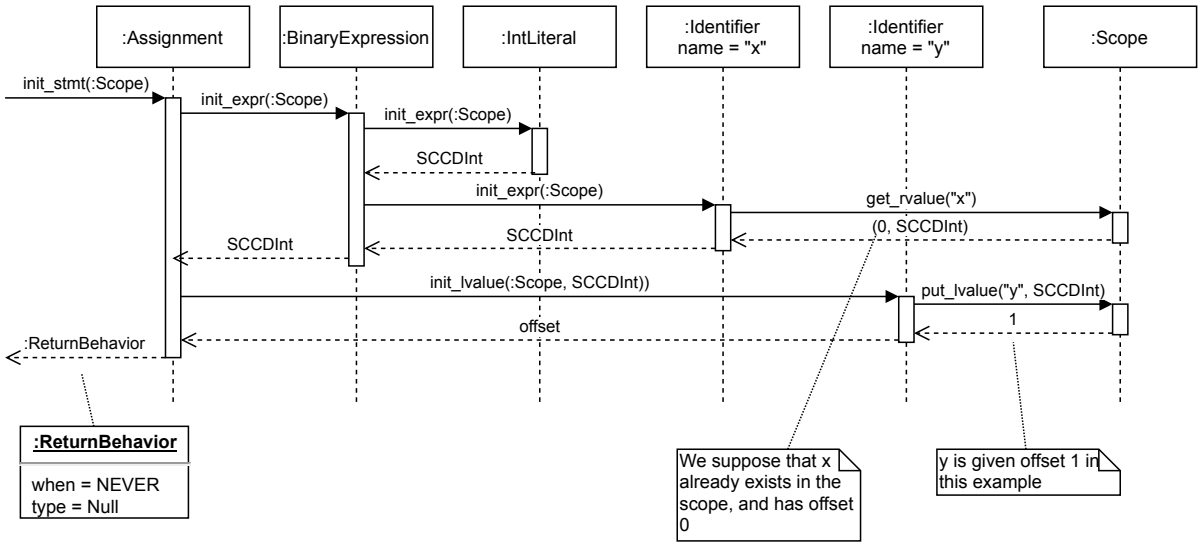


Figure 3.10: Sequence diagram for static analysis of statement `y = 2 * x`

declarations are allowed almost everywhere, and can be nested arbitrarily.) In order to access surrounding scopes, when a new scope is introduced, that scope always has a *parent*. The only scope without a parent is the *global scope*, which is typically created by the invoker of the static analysis step on the root node of the AST.

Action language constructs can create new scopes during static analysis. Currently, the only construct doing this, is the `FunctionDeclaration` expression. Created scope objects are stored in the AST, because they contain important information for execuction (such as the size by which to grow the stack upon calling a declared function).

Example: Figure 3.11 shows a fragment of code and its AST after static analysis. Upon static analysis, the `FunctionDeclaration` object creates a new, nested scope for its function body, setting the parent to the scope object it received from the `Block` object at the top. As a result, the variable `x` can be successfully looked up from within the function body, while the variable `y` remains private to the function body. The nested scope is stored in the `FunctionDeclaration` object itself, because it will be required in order to evaluate the declaration, as seen in the later section on Execution.

There is one more use case for nested scopes, but not in the action language itself: In the statechart language, we also use many nested scopes, as will be discussed in Section 3.4.4. For instance, every transition has its own scope, containing the names of the transition's event parameters as variables, so they can be accessed from the guard condition and transition's action code, but not from elsewhere.

3.3.3 Execution

After static analysis has been performed on an AST, and has succeeded, one can execute the AST. The execution of an AST does not modify the AST, it only modifies *memory*, as was shown in Figure 3.3.

Similar to the static analysis step, to execute an AST, one invokes the execution method of the root node, as it will invoke execution on its children, etc. Depending on the type of the node, the execution method is as follows:

Expression `eval(memory: MemoryInterface): Any`

> Evaluates the expression, reading/writing from/to the memory object given, and yielding a result. (An expression may write to memory, because a function call is an expression.)

LValue `eval_lvalue(): int`

> Returns the offset of the LValue (variable) relative to the start of the current stack frame. This offset has already been computed in the static analysis step, and has been stored in the LValue object. It can be a positive or negative value, depending on whether the variable exists in the current stack frame, or one of its ancestors.

Statement `exec(memory: MemoryInterface): Return`

Code fragment:

```
x = 1;
func() {
  y = 2;
  x = y;
};
```

AST:

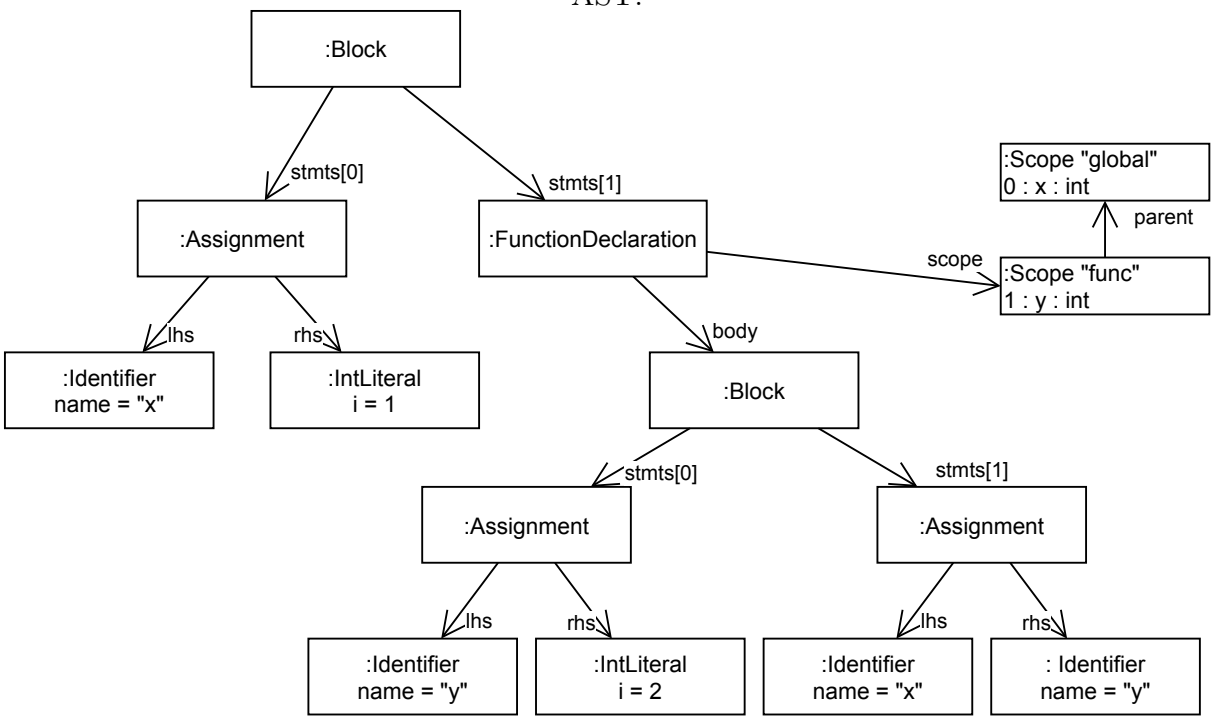


Figure 3.11: Code fragment with function declaration and its AST after static analysis, showing the hierarchy of `Scope` objects on the right

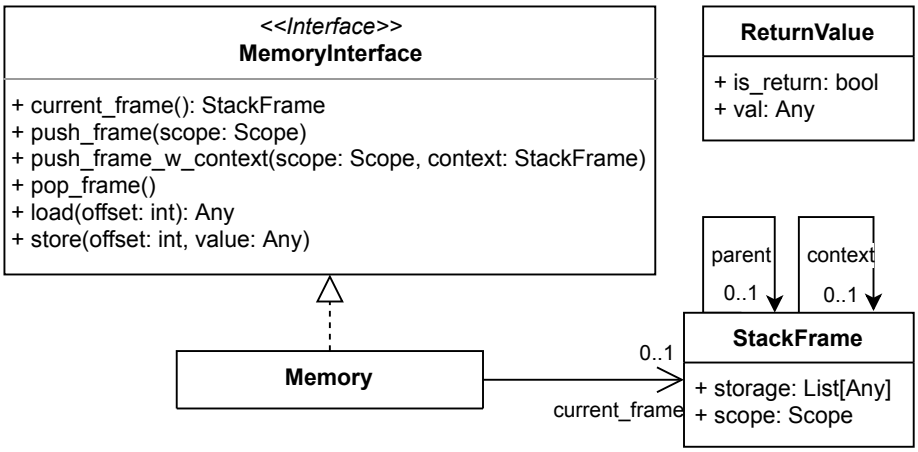


Figure 3.12: Classes involved in execution.

Executes the statement, reading/writing from/to the memory object given, and yields a `Return` object, indicating whether the statement execution caused a return-statement to be executed, and if so, what the returned value was.

The `MemoryInterface`-implementing object received by these methods as a parameter, is conceptually the *stack memory* of the instance being executed. It defines operations for *loading and storing* data from/to memory, as well as *pushing/popping stack frames* (UML diagram in Figure 3.12).

For most constructs, the implementation of the execution method is trivial. The not-so-trivial cases are function declarations and function calls. A function call should push a stack frame, and this is done by calling `MemoryInterface.push_frame`. This method expects a `Scope` object. The scope object's primary function here is to describe the size by which the stack should grow. For every function declaration, a scope object has been statically computed during static analysis, and stored in the `FunctionDeclaration`-expression. Every time the `FunctionDeclaration` is evaluated, a function-value is returned with the scope object embedded in it. A `FunctionCall` therefore has access to this `Scope` object in order to grow the stack.

MemoryInterface implementation

The default implementation of `MemoryInterface`, called `Memory`, supports the following interesting features:

Recursion makes it easier to code certain algorithms, but in the general case (ignoring tail call optimization), has an unpredictable time and memory footprint.

Closures are a feature in high-level, garbage-collected languages such as Python, Java and JavaScript. They allow functions to declare other functions, and return those functions as values. Those returned functions are allowed to refer to variables of the function in which they were declared, even if that function's stack frame has been

popped. Closures only work in a garbage-collecting environment as stack frames must be allocated on the heap, and not be freed until functions that still have access to them go out of scope.

To support these features, the current implementation allocates stack frames on the heap, and maintains 2 singly-linked lists between stack frames:

- List of parents: Every stack frame has a pointer to its parent, i.e. the stack frame of the function that invoked (called) the current function, and hence created the current stack frame. When the current stack frame is popped, its parent becomes the current stack frame (again). This alone suffices to supports recursion.
- List of contexts: Every stack frame has a pointer to its context, i.e. the stack frame that was at the top of the stack when the function that created the stack frame was declared. If memory access to a negative offset relative to the current stack frame is requested, the list of contexts is used. This is to support closures.

Neither recursion nor closures are necessary or perhaps even desired for usage in statecharts. These are high-level features not encountered in e.g. an embedded environment. An upside is that with these features, it is impossible to write code that contains certain types of memory errors, such as accessing an invalid memory address. Support for these features can be dropped without changing the syntax of the language, but then the static analyzer *should* detect and reject code making use of these features.

If support for recursion and closures is dropped, the maximum size of the call stack can be easily computed statically, allowing for a much simpler implementation of `MemoryInterface`, that could work with a pre-allocated, fixed amount of memory, suitable for use in an embedded environment.

3.4 Statechart Language Implementation

The statechart language implementation is central to this thesis. Part of the language is a parser/loader function, which parses models in an XML format to construct an AST, and performs an initialization step on this tree, calculating certain static properties of its elements required in our execution implementation. A loaded statechart can then be instantiated and executed.

The execution part of the statechart language includes no queueing of input or output events. It consists only of the instance initialization function (entering the default configuration, must be called once) and the big-step function (responding to a set of input events, may be called repeatedly). Both these functions are only executed when explicitly called. The statechart language itself is therefore *not autonomous*.

Autonomous behavior is implemented through an event loop in the Controller, which is discussed in Section 3.5. Although the statechart language does not depend on the

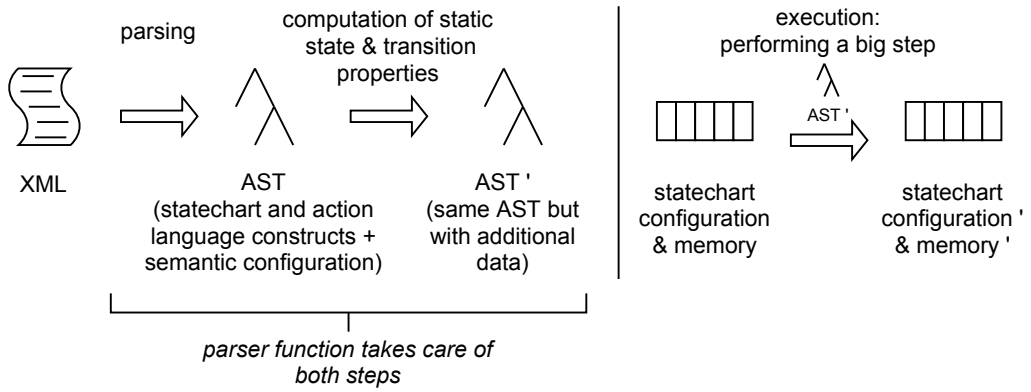


Figure 3.13: Steps of loading and executing models in the statechart language

Controller, statechart instances assume some implementation of an event loop, which they use for scheduling and canceling of (future) events, for timed transitions, as we will see.

3.4.1 Parser and Constructs

Although statecharts are a visual, topological language, our runtime parses statechart models in an XML format. The XML format was originally based on the SCXML standard [19], but "liberties" were taken to deviate as desired: no intention of syntactical compatibility was ever on the agenda.

In order to parse XML, we use the well-known Python library lxml [3], which uses the C-library libxml2 [2] under the hood.

As the statechart language can contain action language expressions and statements in various places, the statechart parsing logic will also regularly invoke the action language parser (and static analyzer). An error in action language code "bubbles up" as an error in the statechart model.

Parsing logic and schema

Apart from parsing our input files, we also want to validate them against a schema. However, no schema was explicitly written in an XML schema language such as XML Schema [18], because of the burden of keeping the schema in sync with the parser logic, which, in a system-under-development, is easily forgotten, and can feel like a useless endeavor, since parser logic and schema contain much of the same information. Instead, the parser code was converted into 2 parts:

1. A tree of nested declarations of expected XML elements, their order and multiplicities, and callbacks for handling those elements. Figure 3.14 shows a class diagram of the structure of the parser rules.

2. A generic parser function, built on top of lxml, using the tree of nested declarations and callbacks to carry out the parsing.

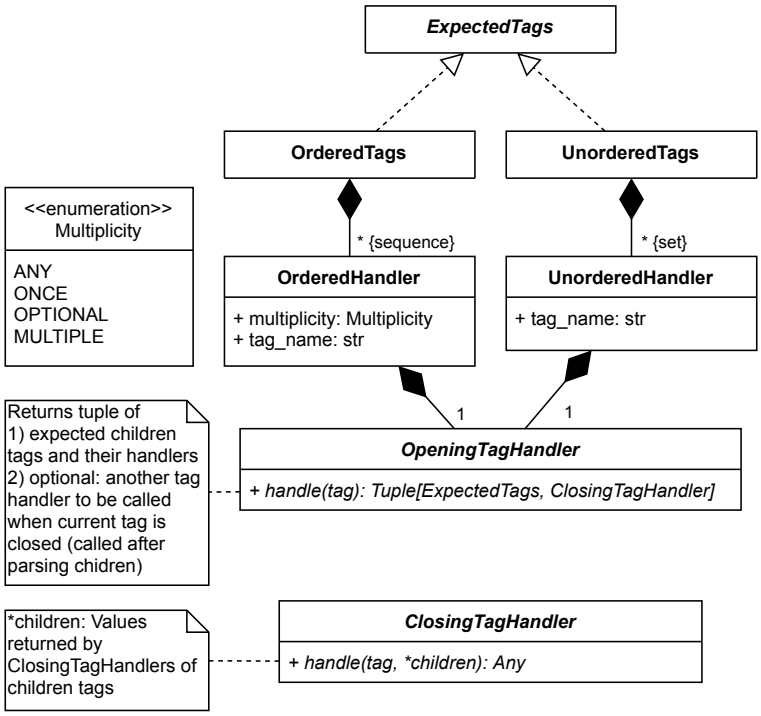


Figure 3.14: Class diagram (conceptual, not actually implemented as these classes) for structure of parser rules

This way, the parsing logic *is* the schema, with all the reusable functionality of checking multiplicities un-duplicated in a generic parser function. This function also shows very comprehensible error messages, printing out a fragment of the input file, with the error highlighted. The function is potentially reusable in other projects, and is also used in other parts of the SCCD project, such as the parsing of test files.

Statechart XML format

We'll introduce the XML format for statechart models. The root XML node is always `<statechart>`. It has the following children, in order:

`<semantics>` (0..1) The semantic options chosen for the model. For every semantic option that is not specified, SCCD's default value for that option is used.

`<datamodel>` (0..1) The initialization code (in action language) of the statechart's *data model*. The data model is the set of variables (and their types) that are readable and writable from everywhere in the statechart.

`<inport>` (*) An input port. A statechart model can only receive input in the form of *input events*. An input port defines a set of input events.

`<outport>` (*) An output port. A statechart model can only produce output in the form of *output events*. An output port defines a set of output events.

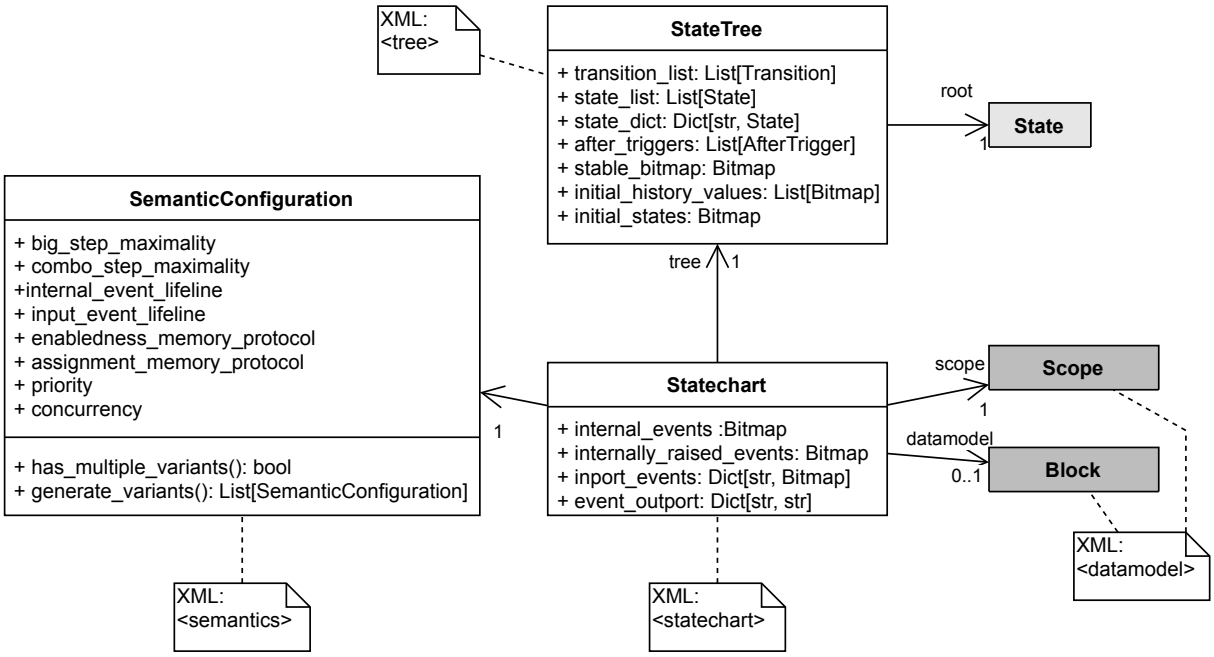


Figure 3.15: Syntactical constructs of the statechart language, part 1
Dark-gray: Action language constructs, see Figure 3.4 and Figure 3.5.
Light-gray: Statechart "tree" constructs, see Figure 3.16.

`<root>` (1) The root state of the *state tree* of the statechart model.

Syntactical constructs

A `<statechart>` XML node at the highest level is parsed into a `Statechart` object in the runtime. Figure 3.15 shows a UML diagram of this class, and the other classes it is composed of. It is trivial how the XML format maps to these constructs. The `Statechart` object represents a loaded model and can be executed by the execution runtime. The `Statechart` object consists of a `StateTree` object, which owns the root of the state tree structure, which is a `State`. The `State` class and the other constructs making up the tree structure are shown in Figure 3.16.

In the runtime, the `StateTree` object is created *after* the state tree structure has been created (by the parser). Besides containing the root state, the `StateTree` object contains a lot of additional fields of information derived from the state tree structure. This information is used for efficiently executing the statechart model.

We'll now introduce the statechart language constructs that make up the state tree structure (Figure 3.16). It is clear that the statechart language extends on the action language: For the action language constructs, we refer to Section 3.3.1.

`State` The state class represents a basic state (if it has no children) or Or-state (if it has 1 or more children).

It is also the base class for other types of states. It can have a parent state and any number of children states, creating the state tree structure. The root of a state tree is always of this type (an Or-state).

`ParallelState` represents an And state.

`HistoryState` represents a history (pseudo-)state. Since this class is abstract, its subtypes `ShallowHistoryState` and `DeepHistoryState` are to be used instead.

`Transition` represents a transition. A transition has a source and a destination state. Except for the root state, *any* state can be the source and/or destination of a transition. The "source" is a bi-directional association: every transition is also listed in its source state's list of outgoing transitions.

A transition can have a *guard condition* (`guard`), which is just an `Expression` of our action language.

A transition can have an *event trigger* (`trigger`), which is a set of events that have to be present in order for the transition to be enabled.

A transition can have *actions*, such as raising events or executing action code.

In the diagram, a transition is also shown to have a `Scope` object (see action language). This is the variable scope used for evaluating the guard condition, and executing actions. The scope contains the transition trigger's event parameters as variables, so they can be read.

`Trigger` is a transition's trigger, a set of events required to be present for the transition to be enabled.

It is also a base class for 2 other kinds of triggers: A `NegatedTrigger` also defines events *not* allowed to be present; an `AfterTrigger` listens for an *after-event*.

`Action` is an *action*. Actions may be executed when a transition fires. There are 2 types of places in the state tree where actions can be defined: (1) directly on a transition, or (2) as the enter- or exit-actions of a state, which are executed when any transition causes that state to be entered or exited, respectively.

There are 3 types of actions:

`RaiseInternalEvent` raises an internal event. If and when the internal event becomes visible to the statechart, depends on the semantic configuration chosen.

`RaiseOutputEvent` raises an output event.

The statechart language does not queue output events, instead they are "delivered" through a callback, synchronously called during transition execution. The callback's implementation may queue the output event, or immediately "handle" the event, performing some action (but caution is required since the statechart is in the middle executing of a transition).

`Code` executes a piece of action language code.

Example state tree

As an example, we will show the XML format, visual representation and state tree of a statechart. The statechart implements a very simple control panel of a 4-burner stove, with 2 buttons: A button for increasing the heat of a burner, and a button for selecting the next burner. Holding the button for increasing the heat for longer than 1 second will cause the heat to be increased every 200 ms. There is no way to decrease the heat of a burner.

The visual representation is shown in Figure 3.17. The state tree is shown in Figure 3.18. The XML format is as follows:

```
<statechart>
  <datamodel>
    burners = [0, 0, 0, 0];
    selected = 0;

    min = func(a: int, b: int) {
      if (a &lt; b) return a;
      return b;
    };

    increase = func {
      burners[selected] = min(burners[selected] + 1, 9);
    };
  </datamodel>

  <inport name="in">
    <event name="pressed_increase"/>
    <event name="released_increase"/>
    <event name="select_next"/>
  </inport>

  <root>
    <parallel id="p">
      <!-- upper orthogonal region -->
      <state id="heat" initial="Released">
        <state id="Released">
          <transition event="pressed_increase" target="../Pushed"/>
        </state>

        <state id="Pushed" initial="Waiting">
          <onentry>
            <code> increase(); </code>
          </onentry>
          <transition event="released_increase" target="../Released"/>

          <state id="Waiting">
            <transition after="1␣s" target="../Increasing"/>
```

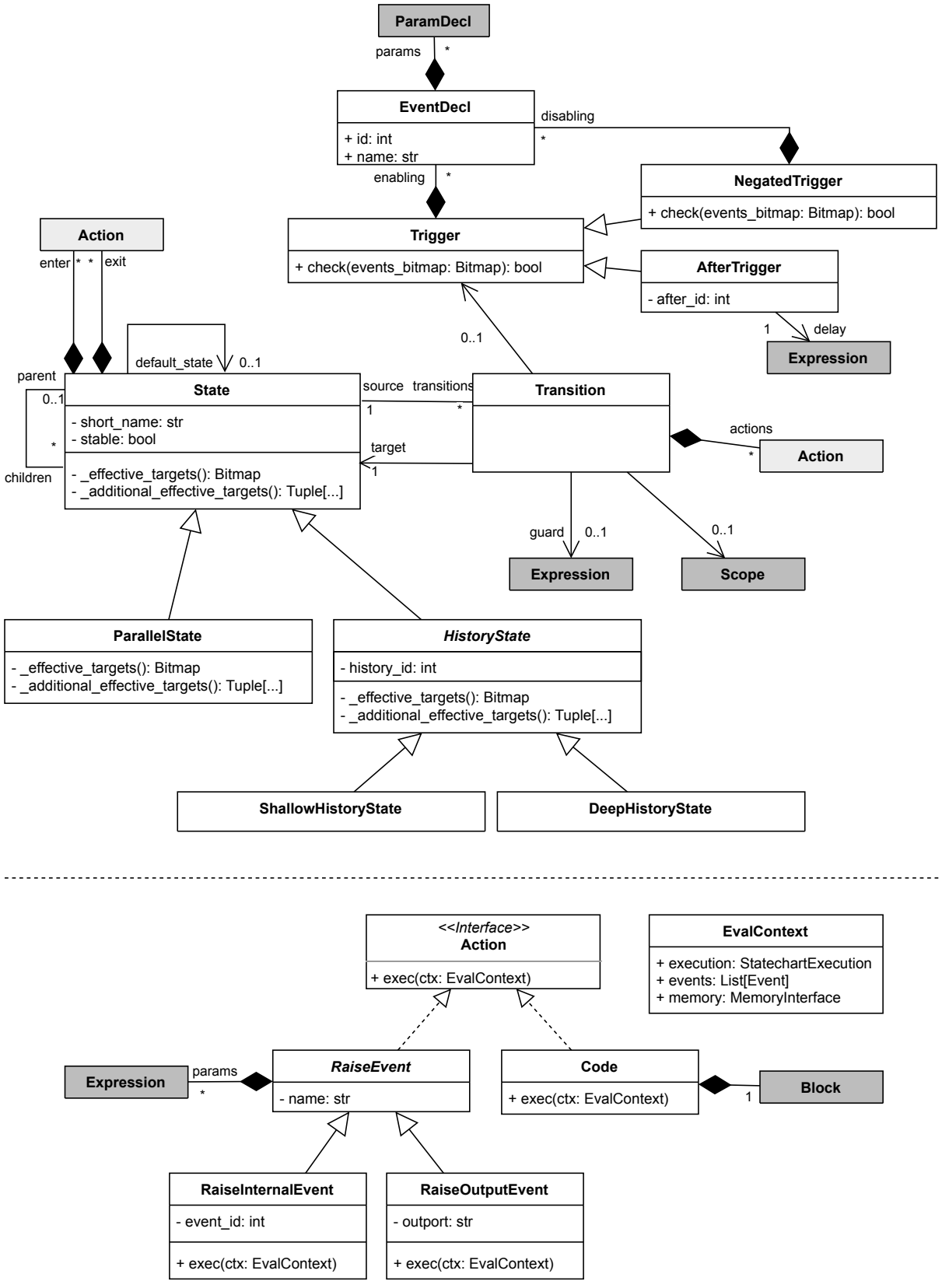


Figure 3.16: Syntactical constructs of the statechart language, part 2
Dark-gray: Action language constructs, see Figure 3.4 and Figure 3.5.
Light-gray: "Action" class, see bottom of diagram.

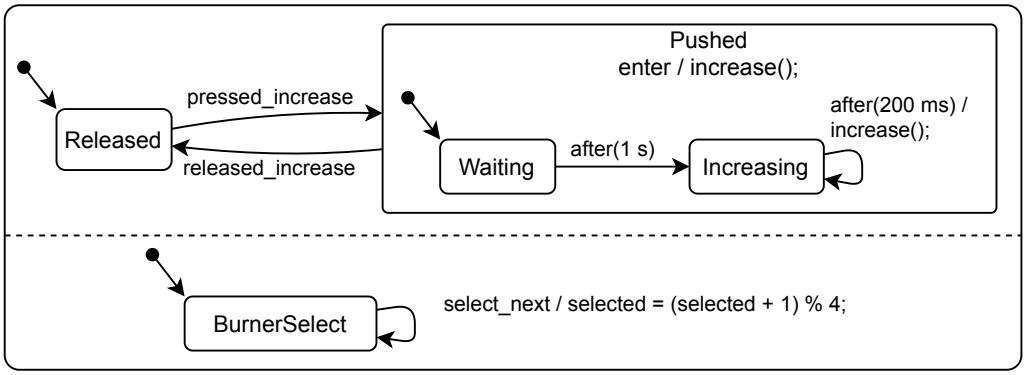


Figure 3.17: Statechart for 4-burner stove example

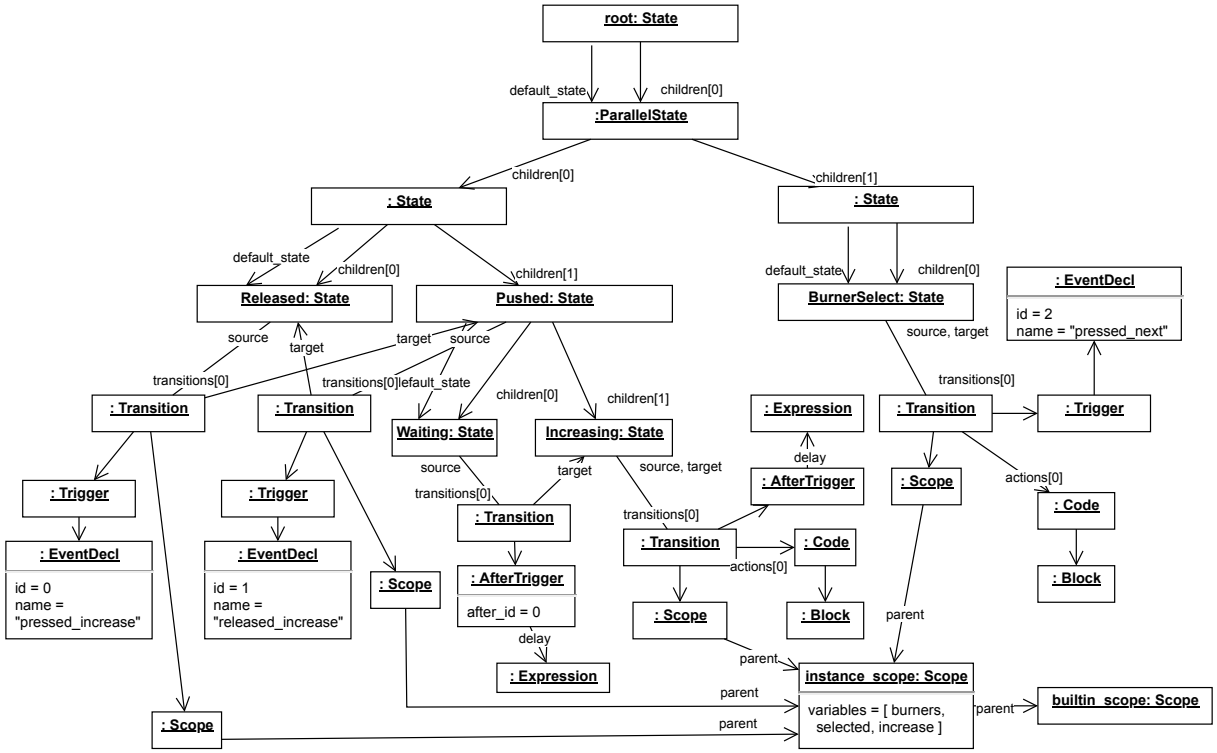


Figure 3.18: State tree for 4-burner stove example
Detailed action language constructs omitted.

```
          </state>
          <state id="Increasing">
            <transition after="200␣ms" target=".">
              <code> increase(); </code>
            </transition>
          </state>
        </state>
      </state>

      <!-- lower orthogonal region -->
      <state id="burner_select">
        <state id="BurnerSelect">
          <transition event="select_next" target=".">
            <code> selected = (selected + 1) % 4; </code>
          </transition>
        </state>
      </state>
    </parallel>
  </root>
</statechart>
```

The targets of transitions are XPath-like paths. During parsing, the targets of transitions are only filled in after building the state tree, as an additional step.

3.4.2 Execution: overview

The rest of this chapter explains how statechart models are executed. In doing so, we will also mention regularly that certain properties of statechart constructs are *statically computed*. This points to a static analysis step (or "tree initialization" in Figure 3.13) that is performed by the parser, right after the state tree has been constructed. This step is not separately discussed.

We will first explain our transition-firing implementation, as it can be understood separately from other parts of the execution runtime. Next, we explain how action language code can be used in statechart models, and how statechart execution causes this code to be executed. Then, we "zoom out" to a broad view of how a statechart's execution, i.e. how it is "stepped", and how semantic variation is possible in the implementation of this stepping. We then explain the stepping algorithm in detail, including our implementation of the semantic variation points described in BSMLs. Finally, we mention a number of implemented performance optimizations.

3.4.3 Firing a transition

In order to understand statechart execution in our runtime, a good place to start is the transition firing function, since it is independent of the various semantic options that can be chosen.

We assume the transition to fire has already been chosen, and the set of current events is known.

For now, the firing of a transition does 3 things, in the following order:

1. Exit a subset of the current states, removing them from the *statechart configuration* (= the set of current states). States are exited in child → parent order. For each exited state, execute its exit actions, if there are any.

2. Execute the transition's actions (in the order specified

3. Enter a new set of states, adding them to the statechart configuration. States are entered in parent → child order. For each entered state, execute its enter actions, if there are any.

The only non-triviality about firing a transition, is determining the sets of exited and entered states. Before we go into detail about how these sets are calculated, we must first explain how the statechart's configuration, and sets of states in general, are represented internally in the SCCD execution runtime.

Configuration as a bitmap

It is clear that, as a statechart executes, we have to keep track of its configuration (set of current states). There are many ways to represent the configuration. Conceptually, it is an unordered set. Originally, a Python list of `State` objects was used, ignoring its order. Simon Van Mierlo at some point introduced *bitmaps* for representing configurations. In order to represent a statechart configuration as a bitmap, every state in the state tree is given a unique integer ID. The root state is given ID 0, and the other states are assigned incrementing numbers in a depth-first fashion, so children are always assigned larger IDs than their parent. Every state ID would be a position in the bitmap, 1 or 0 (in the current configuration or not). Because the resulting bitmaps are relatively small and *dense*, we use Python integers to represent them.

This usage of bitmaps was further extended to all places in the code where *sets of states* had to be represented, because of significant advantages:

- Compact representation: For every state in the state tree, only a single bit is used.

- Efficient union and intersection operations: These operations are implemented using bitwise-OR and bitwise-AND operations. A single machine instruction can perform these operations for typical statecharts.

- When enumerating a bitmap's elements (by iteratively right-shifting the bitmap, and yielding when the least significant bit is 1), they are automatically sorted. This, combined with the depth-first fashion state IDs are assigned, means that when enumerating the "current states", they always appear in shallow → deep order.

- In Python, unlimited size: Our bitmap implementation uses an integer internally. There is no limitation on the maximum size of integers in Python 3, making our solution "work" with very large statecharts as well, although performance may take a hit.

- A bitmap can be used as key in a mapping. (Useful for caching of transition candidates, see later)

To find a `State` object based on its state ID, an array of all `State`s, indexed by their ID, is kept in the `StateTree` object (Figure 3.15).

Now that we have seen how bitmaps are used to represent the statechart configuration, and can be used to represent sets of states in general, we can explain how the sets of exited and entered states of a transition can be efficiently calculated.

Calculating set of exited states

The set of exited states is always equal to the intersection of the set of the transition arena's descendants (not including the arena state itself), and the current configuration. The arena of a transition is the lowest-common-ancestor of its source and target, that is an Or-state. For every transition, the arena's set of descendants can be statically computed. This statically computed set is stored as a bitmap in the `Transition` object. Now that both the arena's set of descendants and the statechart configuration are bitmaps, the intersection is therefore a simple bitwise-AND operation, yielding the set of exited states, as a bitmap. Enumerating the elements in this bitmap, in reverse, yields the IDs of the states-to-exit in the right order, i.e. from child to parent.

Example: Figure 3.19 shows a statechart and a transition t. First, we show that the set of exited states depends on the statechart configuration, and can therefore not be computed statically: As t is fired, it is clear that we exit the AND-state S, and therefore all descendants of S are exited as well. The transition t can be fired regardless of D or E being current states, so either D or E should be exited depending on the configuration. Second, we give an example of what the exited states set could look like. Suppose the statechart configuration consists of the states $\{root, S, U, C, V, D\}$ The arena of transition t is the root state, so the arena's descendants set consists of all other states $\{root, S, U, A, B, C, V, ...\}$. The intersection between these 2 sets is the exit-set $\{S, U, C, V, D\}$. After these states have been exited, only the root state is a current state.

Calculating set of entered states

The set of entered states is more complex, but can be computed entirely statically for any transition, if the transition's target state is not a history state. (Our implementation supporting history states is an extension to the current algorithm and will be explained later.) As a start, we take the intersection set between the target's state set of ancestors and the arena's descendants, and we call this set the *enter path*. When enumerating the states of the enter path, a sequence of states is obtained, such that the next state is always a child of the current, from the arena to the target state (not including the arena or target state). All states on the enter path are in the entered states set. Naturally, the target state itself is also entered, and if the target state has children, some of those children should also be entered, depending on the type of the target state:

- If the target state is an And-state, the And-state and all of its children should be considered targets, recursively.

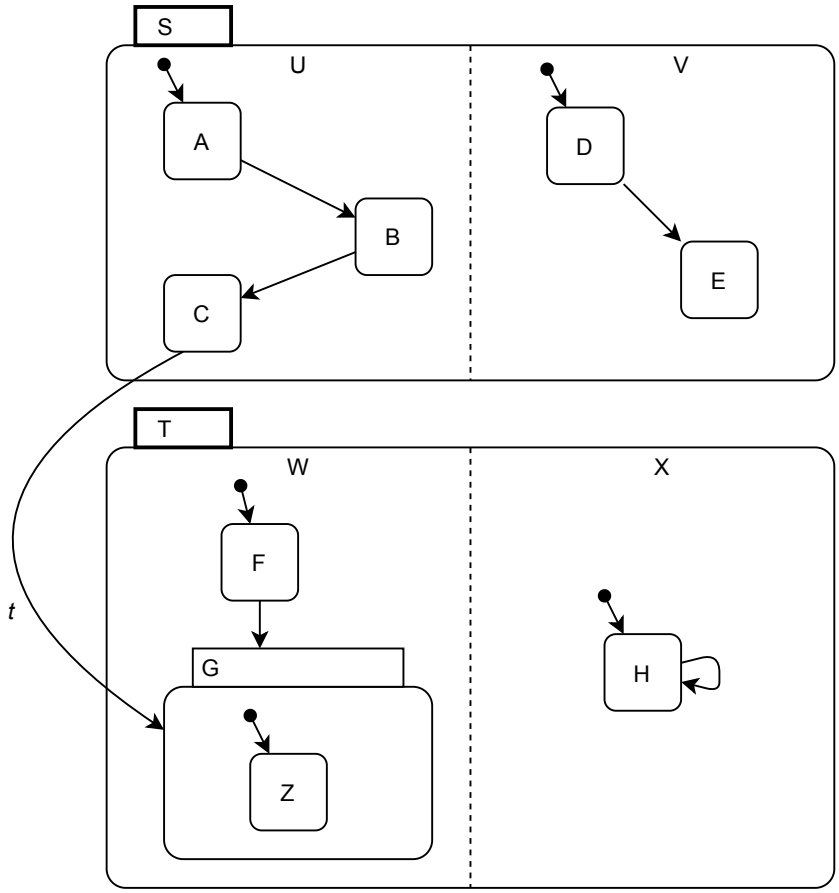


Figure 3.19: Statechart with a complex transition t, as an example for entered and exited sets of states.

- If the target state is an Or-state, the Or-state and its default state should be considered a target, recursively.

This behavior is implemented for all states in the polymorphic `_effective_targets` method. It returns a set of states to enter, if the state is the target of a transition. This includes the state itself, and a subset of the children, as described above. This method is called during calculation of the static set of a transition's entered states.

Example: In Figure 3.19, the transition t's enter path consists solely of the state T. The state G is also entered, as it is the target of the transition, and calling the `_effective_targets` method on G yields its default state Z, which is also added to the set of entered states.

But we are not done: in the previous example, the state H should also be added to the enter-set. This is because on the enter path, there was an And-state T. And-states occuring on the enter path add their children that are not on the enter path as target states. The polymorphic method `_additional_effective_targets` behaves this way for And-states, and is called for every state on the enter path.

We have now successfully calculated the set of entered states statically, and this set is stored as a bitmap in the `Transition` object, so that it can be quickly looked up during transition firing.

History implementation

As we have mentioned before, if the target of a transition is a history state, the entered states set of a transition cannot be computed entirely statically. However, our previous calculation is still correct, except that the target state (in this case, a history state) itself is not entered (history states are pseudo states, and cannot be entered!), but instead, the *history value* for that state should be looked up. A history value is simply the set of states to be entered if its history state is the target of a transition, and is simply added to (union-ed with) the statically calculated entered states set. The history value is another bitmap that we keep for every history state in the statechart, making up the "dynamic" part of the statechart, together with the statechart configuration.

How to generate the history values? If we exit a state A, we have to save some history value if A has a history state as a direct child. The value to save depends on the type of history:

- If a *deep history state*, the history value to be saved is the intersection of A's descendants with the *exited states set*, which was of course calculated as explained above, before any state was exited. The descendants bitmap of every state is pre-calculated, so this operation is another simple intersection of 2 bitmaps.

- If a *shallow history state*, state A must be an Or-state, since shallow history states make no sense for And-states. The history value to be saved is the result of calling the method `_effective_targets` on A's exited child: When the history state is a transition target, A's exited child should be treated as the target of the

transition, also recursively entering A's default state(s), if there are any. For a given state tree structure, the result of calling the recursive (and therefore slow) `_effective_targets` on any state never changes, so it is also computed statically.

So even for shallow history states, the saved history values are deep, i.e. *partial configurations* (instead of recording a single state). With this approach, when restoring a history value, it is not necessary to treat shallow and deep history states separately.

Example: In Figure 2.10, when the transition from C to D is made, the shallow history state records the history value $\{Y, B\}$. If the history state were deep, the recorded history value would be $\{Y, C\}$.

At the beginning of a statechart's execution, there would be no history values unless we explicitly initialize them. We do this, to have *sane* (non-crashing) behavior when a history state is entered before its history value has been saved. The default history values for all states are equal to the result of calling the method `_effective_targets` on the parent of the history state, i.e. we simply enter the "default" states.

In order to efficiently set and look up history values in our "dynamic" part of the statechart, every history state has a unique, statically-assigned `history_id` (incrementing from 0, depth-first), and we store all history values in an array, indexed by history id. We cannot write history values directly to the `HistoryState` object itself, since it belongs to the "static" part of the statechart, which should never change during execution.

Timed transitions

There is one more feature that is yet unsupported by the transition firing algorithm as explained so far: timed transitions, i.e. transitions with an after-trigger.

In visual syntaxes of the statecharts, as well as in our statechart XML format, a timed transition is given no event trigger, but a *time duration expression*, such as $100ms$, indicating that after the source of the transition having been part of the configuration for $100ms$, the transition becomes enabled.

In the implementation, an after-trigger behaves much like a regular trigger, in the sense that it also responds to an event. This event, called an *after-event*, is hidden to the modeler. It is *scheduled* with its future timestamp upon entering the source state of the transition, and *canceled* upon exiting it.

There are 2 types of events a statechart can respond to: *Input events* come from the environment, and *internal events* are raised by fired transitions in the statechart itself. At first intuition, it may seem like after-events are internal events, since they are scheduled by the statechart itself, but this is not true: internal events cannot have a time delay, they always occur (conceptually) at the same timestamp as they were raised, even if the transition that raised the internal event occured *logically* before the transition responding to it. Input events, by contrast, can be given any timestamp, and are put in a global *event queue*, where they are sorted by timestamp, and popped when they become due. It has been mentioned before that the implementation of this event queue is not part of the statechart language, yet a statechart instance expects *some implementation*, and must be given callbacks for scheduling and canceling events upon instantiation.

Just like the history values, we cannot store the scheduled IDs in the state tree (`State`, `Transition` or `AfterTrigger`, ... objects), as shouldn't be altered by execution. We therefore statically assign every `AfterTrigger` a unique after-ID (incrementing from 0, depth-first), which serves as index in an array of scheduled IDs, stored in the "dynamic" part of the statechart.

Final algorithm

To summarize, the transition firing algorithm looks as follows:

1. Calculate the sets of entered and exited states as described above.
2. Exit states in order child $\rightarrow$ parent. For every exited state A:
   - 2.1 If A has a history state as its direct child, save the history value for that history state, as described above.
   - 2.2 For all outgoing transitions of A that have an after-trigger, cancel the scheduled after-event.
   - 2.3 Perform the exit-actions of A.
   - 2.4 Remove A from the configuration.
3. Perform the transition's actions.
4. Enter states in order parent $\rightarrow$ child. For every entered state B:
   - 4.1 Add B to the configuration.
   - 4.2 Perform the enter-actions of B.
   - 4.3 For all outgoing transitions of B that have an after-trigger, schedule their after-event.

The dynamic part of a statechart consists of the following values:

- The statechart configuration (set of current states), represented as a bitmap.
- The history values of all history states in the statechart, represented as a history-ID-indexed array of bitmaps.
- The scheduled IDs for current states that have timed outgoing transitions, represented as an after-ID-indexed array of "None" (no scheduled event) or a "scheduled ID".

These 3 values are fields in an object called `StatechartExecution`. This is also the object that implements the method for firing a transition (`fire_transition`).

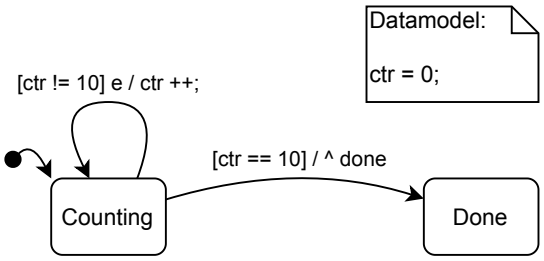


Figure 3.20: "Counter" statechart model, with a datamodel and action language expressions and statements

3.4.4 Action language in a statechart

Action language code can occur in various places in a statechart model:

- The statechart's *data model*, which is a `Block` (a sequence of `Statement`s), declaring and initializing the statechart's variables.
- The guard condition of transitions is an action language `Expression`.
- The *delay* of an after-trigger is also an `Expression`.
- The various types of actions (enter state, exit state, or fire transition):
  - Code-actions execute a `Block`.
  - Raise event-actions have `Expression`s for the raised event's parameters.

Example: Figure 3.20 shows a statechart model containing action code. When the statechart is initialized, the datamodel code is executed, initializing the `ctr` variable to 0. Both transitions have a guard condition reading the `ctr` variable. The transition from Counting to itself updates `ctr` by incrementing it by one.

The following desired features influenced the design of the action language, and its integration in the statechart language:

- Variables and functions declared in the statechart's data model should be readable and writeable in all parts of the statechart.
- Variables and functions can be declared outside of the data model section, but they are temporary and local to the fragment (scope) where they appear.
- If a transition's trigger defines parameters on its events, those parameters should be readable from the transition's guard condition, and its actions.
- As part of the statechart language, builtin functions (and possibly variables) can be defined, which are readable (not writable) in all parts of the model. An example of a useful builtin function, is the *in_state* function, checking if a given state (or lists of states) is part of the current configuration. Another useful builtin function *log*, logs text to the console.

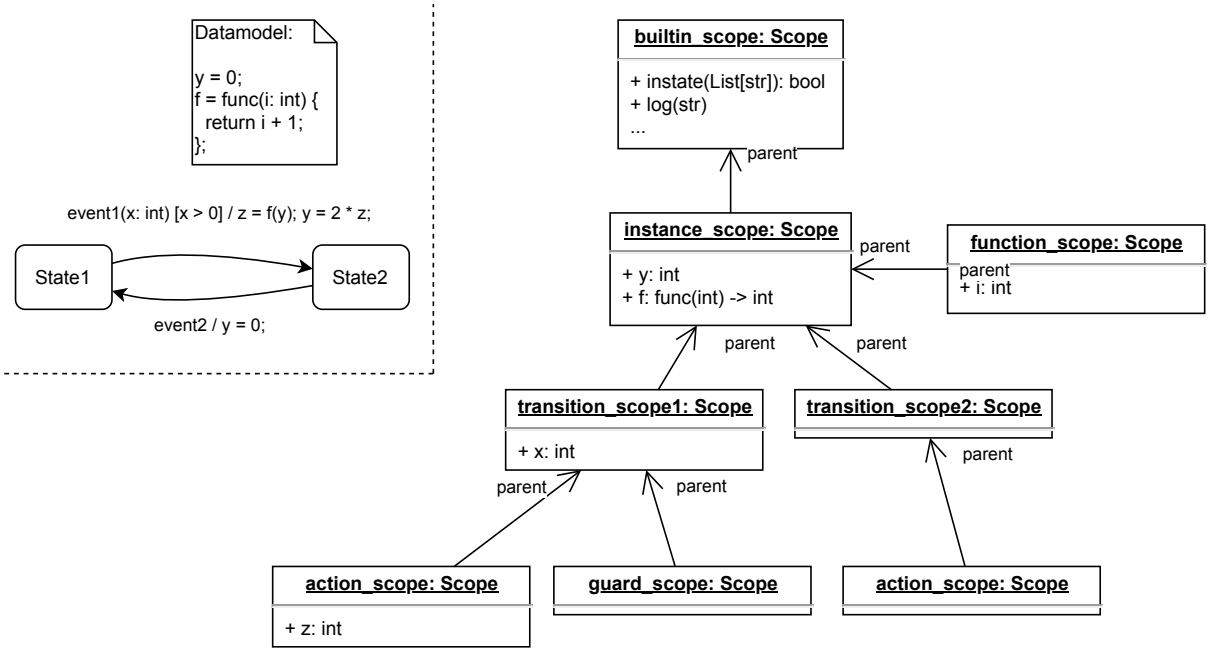


Figure 3.21: A statechart model (upper left) and its hierarchy of scopes

Action language static analysis (in a statechart)

As explained in Section 3.3, every parsed expression or statement in the action language has to be statically analyzed before it can be executed. For static analysis, a `Scope` object must always be given, in which variables are looked up, and to which declared variables are added.

The solution is to first define a special "builtin" scope, consisting of our builtin functions, which are declared as "constant" (read-only). Then an "instance" scope, with as parent the builtin scope, is created, and passed to the static analysis function of the statechart's datamodel, adding all of the variables declared in the datamodel to that scope. Finally, all other occurrences of action code are given their own scope, with the instance-scope as parent. To allow transition's guard conditions and action code to read event parameters, transitions themselves also get their own scope.

Example: Figure 3.21 shows a statechart model and its scope hierarchy. The variable `y`, declared in the datamodel, is part of the instance-scope. Both transitions have their own scope, containing their event parameters, if there are any. Action and guard scopes have their transition's scope as parent, so they can read event parameters. Also note that the function f declared in the datamodel also has its own scope, not explicitly created by the statechart parser, but by the action language itself.

Action language execution (in a statechart)

In the action language, the hierarchy of `Scope` objects is a static representation of stack memory during execution. If the action code contains no recursive function calls, an upper

bound can be put on the maximum amount of stack memory required for a statechart's execution. The absense of recursive calls would also guarantee halting for every piece of action code, as the action language has no loop-structures. However, neither the action language nor the statechart language checks for the presence of recursive calls to claim any of these model properties.

At the beginning of a statechart's execution, we start out with an empty (zero-length) stack memory object. Then, the stack frame for the builtin-scope is pushed, and the builtin values (function implementations) copied to that frame. Next, the stack frame for the instance-scope is pushed, and the action code of the statechart's datamodel is ran, initializing that stack frame. Now the statechart's memory has been initialized, and default states can be entered (possibly triggering more action code) to complete the statechart's initialization. For transition execution and guard evaluation, stack frames are pushed and popped, always symmetrically. In between the execution of transitions, the stack frames of the builtin-scope and instance-scope are never popped. When e.g. evaluating the guard condition of the upper transition in Figure 3.21, the stack frames of the transition's event parameters and of the guard condition (the latter is an empty frame) are pushed, and popped when evaluation is complete.

3.4.5 Broader picture: Stepping of a statechart

We've looked at the transition execution algorithm and the way action language fragments in a statechart are statically analyzed and executed. We "zoom out" now, to look at the statechart interface ...

We treat statecharts as a subclass of Big-Step Modeling Languages [9], where at the highest level, a model's execution is a sequence of *big-steps*. A big-step is a model's response to a set of input events, and consists of the execution of zero or more transitions, changing the model's state and producing output events.

As we have seen in Section 2.2, it is here that many semantic variations can occur, such as:

- If multiple transitions are enabled, which subset and in what order to fire them?
- If the firing of transitions raises internal events, when and for how long can those enable transitions?
- If transitions make changes to the statechart's memory, at what point do they become visible to other transitions?

We will now cover our implementation of the statechart's stepping implementation, which is where all of the semantic variation points are implemented.

3.4.6 Rounds

In BSMLs, a big-step consists of any sequence of *small-steps*. Small-steps are sets of concurrently executed transitions, but since concurrency is not yet implemented in the

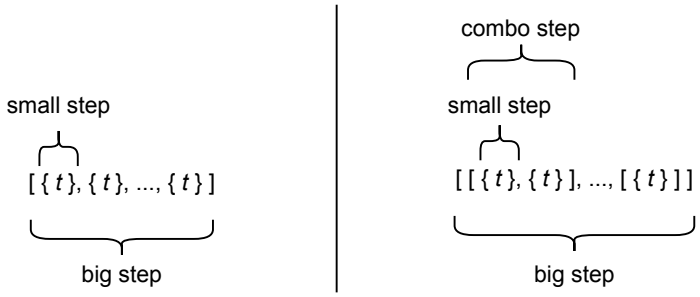


Figure 3.22: Left: A big-step without combo-step semantics. Right: A big-step with combo-step semantics.

runtime, a small-step always consists of a single transition. Depending on the semantic options chosen, sequences of small-steps may also be grouped into *combo-steps*, with a sequence of combo-steps making up a big-step. *Maximality*-semantic options exist for both big-steps and combo-steps, defining restrictions on the transitions that can be taken together in such a step.

In the implementation, this grouping (big-step > combo-step > small-step) was generalized into the `SuperRound` class (Figure 3.23). There is also a `SmallStep` class, which has the superclass `Round` in common with `SuperRound`. Big-steps and combo-steps are implemented as `SuperRound`s. A `SuperRound` has a fixed *subround* object, of type `Round`, and can therefore be another `SuperRound`, or a `SmallStep` round. This way, arbitrary numbers of levels of groupings can be constructed.

A `Round` can be executed, which may cause transitions to be fired. If the execution of a round caused no transitions to be fired, we say the round was *empty*. When a `SuperRound` is executed, it will repeatedly execute its subround, until its subround becomes empty.

The round execution method is called `run_and_cycle_events`, and returns a pair of bitmaps containing information about the transitions that were fired during the round:

arenas_changed A bitmap containing the arenas of the transitions fired during round execution.

arenas_stabilized A bitmap containing the arenas of the transitions fired during round execution that have a target state syntactically marked as "stable".

If a round was empty, both bitmaps will be zero.

Maximality implementation

Every SuperRound has a *maximality* setting. This setting enforces restrictions on the transitions that can be taken together during a round's execution. Possible maximality settings are:

TakeMany No restrictions

TakeOne No 2 transitions with overlapping arenas can be taken together

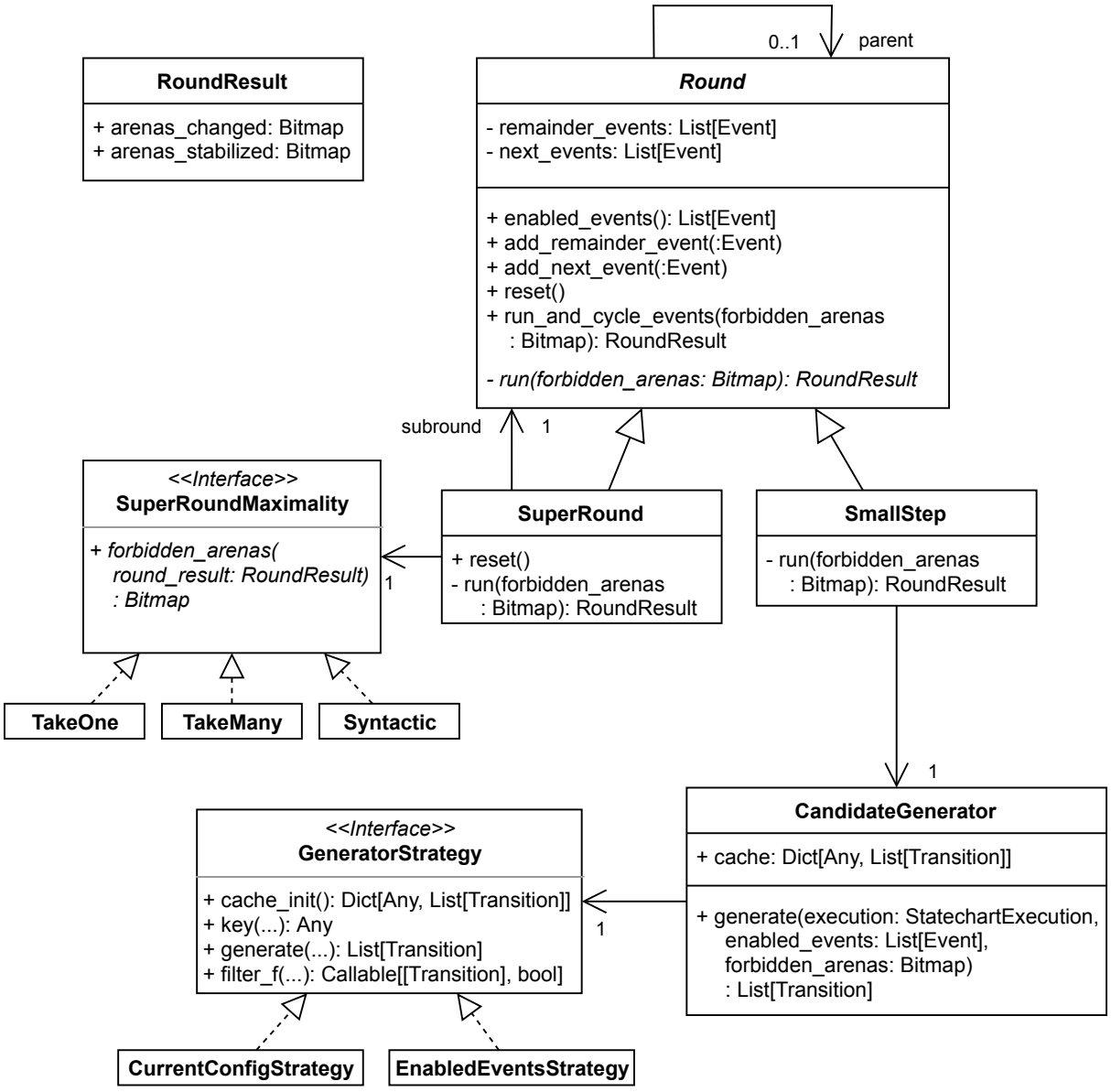


Figure 3.23: Class Diagram for the runtime implementation of rounds and transition candidate generation

Syntactic No 2 transitions with overlapping arenas entering "stable" states can be taken together

These restrictions are implemented by maintaining for every SuperRound an ever-growing set of *forbidden arenas*, i.e. arenas that new transition candidates are not allowed to overlap with. At the beginning of every round, the initial set of forbidden arenas is received as a parameter. For a big-step, this parameter will be 0. For other rounds, this parameter are the forbidden arenas mandated by its parent round. After every fired transition, the forbidden arenas-set grows according to the maximality option for the SuperRound.

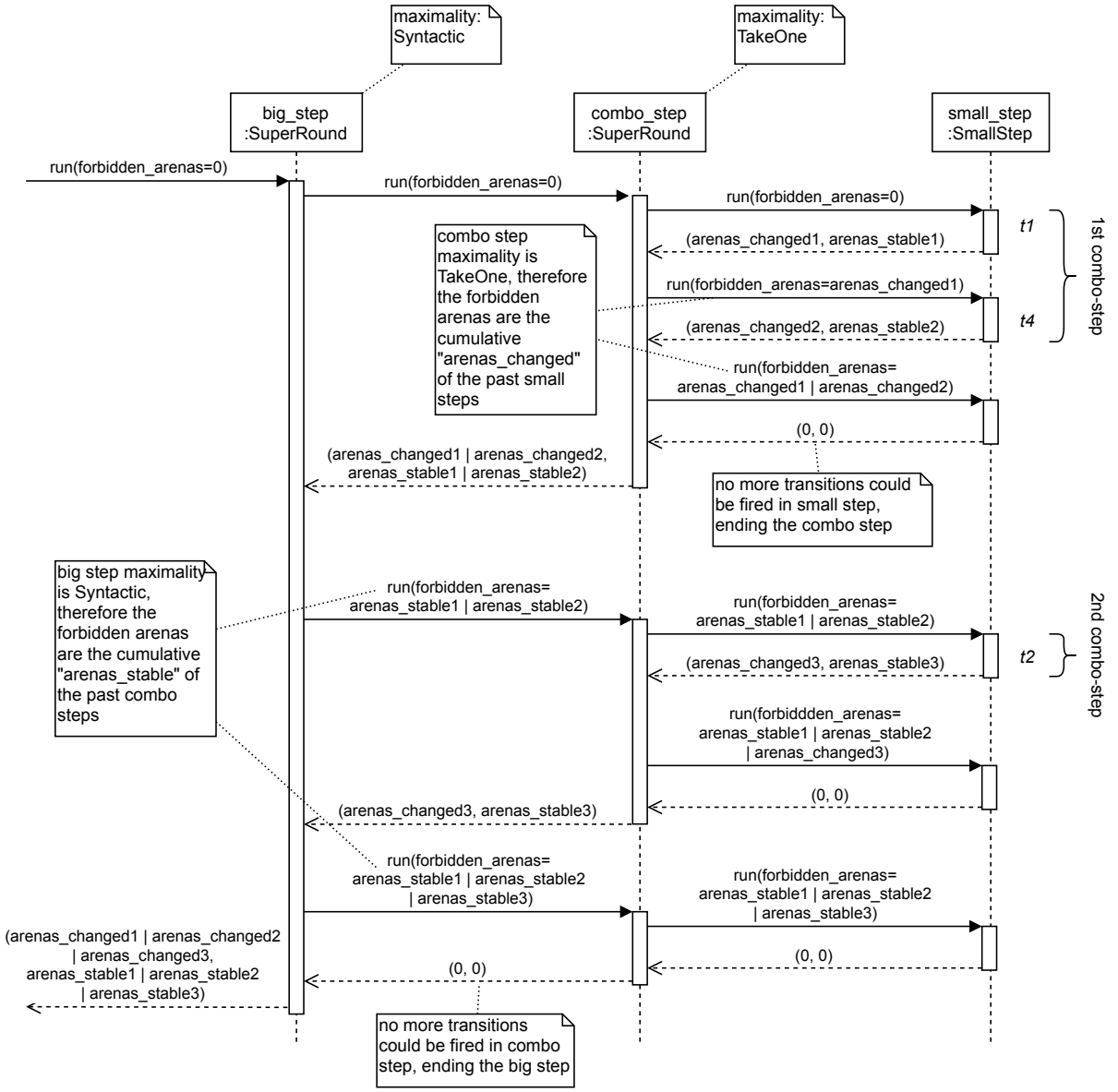
The set of forbidden arenas is implemented as a bitmap. When an arena is "in the set", not just the state ID-bit of the arena is "1", but also the state ID-bits of its *descendants*. This way, checking for overlapping arenas (binary-AND) when considering transition candidates can be efficiently done, as well as adding new arenas to the set (binary-OR).

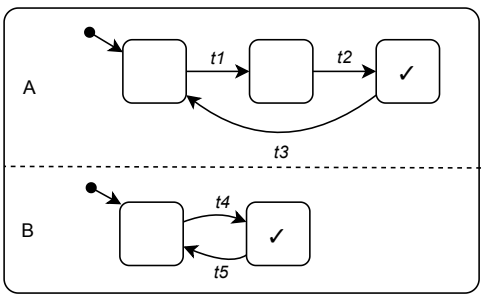
Example: Figure 3.24. shows a sequence diagram of the execution of a big-step. In the example, there are 3 levels of rounds: The superrounds "big_step" and "combo_step", and "small_step". At the highest level, big-step execution is initiated by another part of the runtime, with the forbidden arenas set to 0, because at the beginning of a big-step, no transitions have yet been fired, and therefore no restrictions exist. The big-step executes the first combo-step, which in turn executes the first small-step. The forbidden arenas for the first small-step are still 0, as no transitions have yet been executed. When the first small-step has executed, a pair (`arenas_changed1`, `arenas_stable1`) is returned to the combo-step. The combo-step records these values, but in its next execution call to the small-step, it only passes `arenas_changed1` as the set of forbidden arenas, since its maximality setting (TakeOne) does not care about stable states. The next small-step returns a pair like the first one did. The combo-step's 3rd attempt to execute a small-step results in an empty small-step, causing the combo-step to end. At the end of the combo-step, the union of the `arenas_changed`- and `arenas_stabilized`-values of both small-steps are returned, for the big-step to deal with. The big-step maximality setting (Syntactic) only looks at the `arenas_stabilized`-value, and passes it on as the forbidden arenas-set in the execution of the 2nd combo-step. The second combo-step consists of only a single small-step. An attempt at a 3rd combo-step yields an empty step, causing the big-step to end.

Using our constructs of rounds and superround-maximality, all combinations of Big-Step Maximality and Combo-Step Maximality can be implemented. Figure 3.25 shows the rounds-implementations for all combinations of maximality options currently supported.

Detection of never-ending rounds

Take Many and Syntactic allow for an infinite number of transitions to be taken. Similar to Rhapsody [11], this is dealt with by counting the number of sub-rounds in a `SuperRound`. When a certain limit is exceeded, a runtime error is thrown. This limit is currently hardcoded at 100, but could be a property of the model.





arenas_changed1 = A
arenas_stable1 = 0

arenas_changed2 = B
arenas_stable2 = B

arenas_changed3 = A
arenas_stable3 = A

Figure 3.24: Top: Sequence diagram for the execution of a big-step.
Bottom: An example statechart producing the big-step sequence above, from its initial state, upon any input event. All transitions are triggerless. States with checkmark-symbol are stable states.
Note: the method `run` is actually called `run_and_cycle_events` in the runtime.

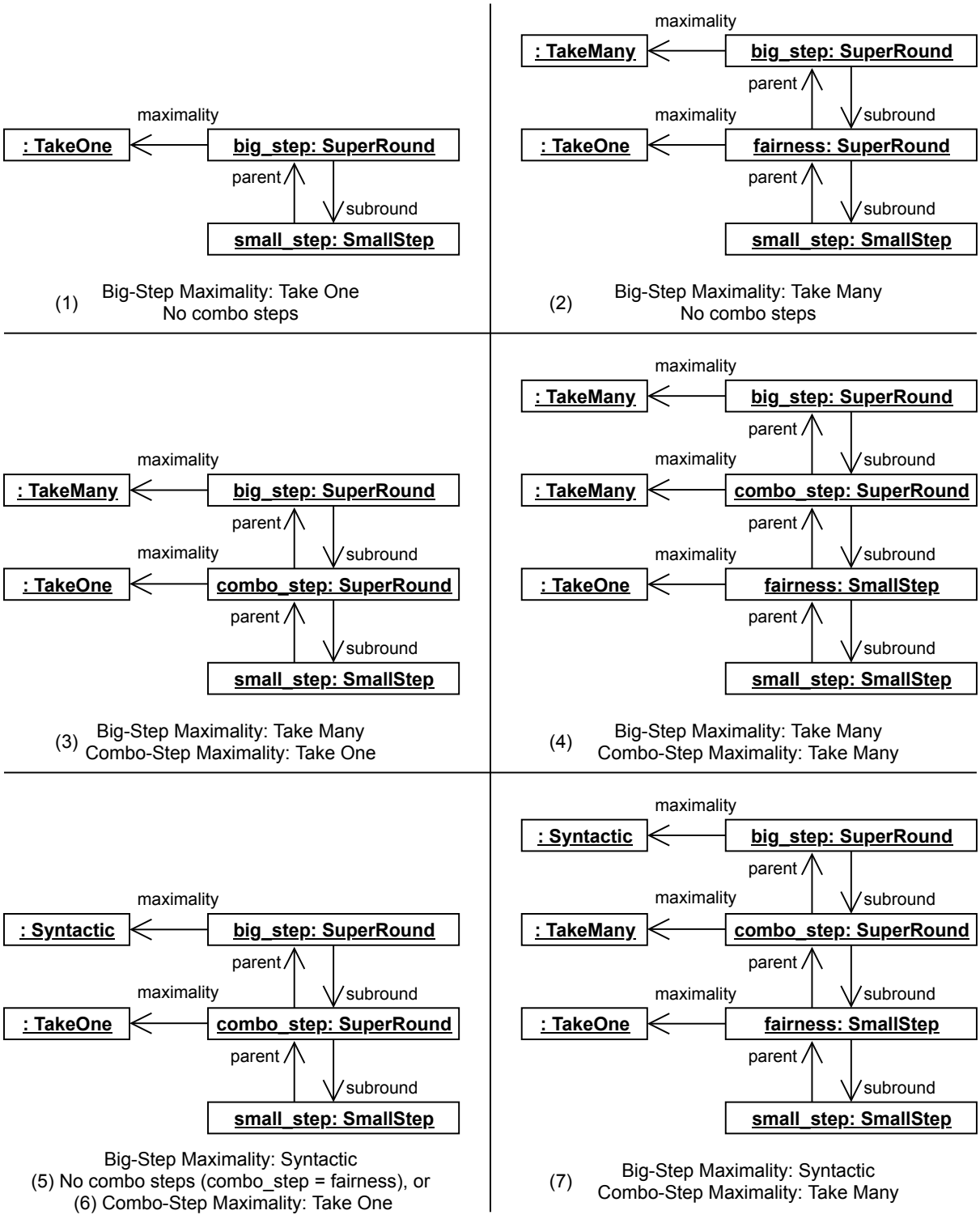


Figure 3.25: Implementations of currently supported maximality configurations. Note that the parent of a small-step is always a TakeOne-maximality superround. This assures fairness.

	Current round (remainder of)	Next round	Special
Big Step	input: WHOLE, internal: REMAINDER	-	internal: QUEUE
Combo Step	input: FIRST COMBO STEP	internal: NEXT COMBO STEP	
Small Step	input FIRST SMALL STEP	internal: NEXT SMALL STEP	internal: SAME

Table 3.2: Event Lifeline options in "rounds" implementation

Fairness under all circumstances

It is important to note that in Figure 3.25, the parent of a small-step is always a super-round with TakeOne-maximality. Even if no TakeOne-semantic option has been chosen in the statechart's semantic options, a TakeOne-superround is secretly inserted, to ensure *fairness*: Even if there never is any restriction on the transitions that can execute, we still demand them to be executed in a fair order, i.e. transitions with non-overlapping arenas are given a higher priority.

Event Lifeline implementation

The event lifeline semantic options of BSMLs specify when input- and internal events are present. They can all be considered relative to some round (big/combo/small-step), and put into two categories of their relationship to that round (with exceptions, discussed later): Either the event becomes present in the *current*, or in the *next* round. Table 3.2 shows these categories.

The implementation of the `Round` class offers methods to add an event to the current or next round. Depending on the semantic options chosen for the statechart, these methods bound to the "right" `Round`-objects, serve as "the" callbacks for raising an internal event and adding input events, during execution.

At the beginning of every big-step, the big-step's input events are added to the right round, always in the category "remainder".

"Remainder" and "next" events are added to the lists `Round.remainder_events` and `Round.next_events`, respectively (shown in Figure 3.23). At the end of every round, those lists are cycled:

```
remainder_events = next_events
next_events = []
```

At any point in time, the current set of enabled events is the union of all `remainder_events` of all `Round`s. Retrieving this union is recursively implemented in the `Round.enabled_events` method, merging the remainder-events with the enabled events of the parent round. It is always the `SmallStep` object initiating this recursive request, as it needs to know the current set of enabled events in order to generate transition candidates (see next section).

There are 2 exceptions to the categories "remainder" and "next". The first is the Internal Event Lifeline option QUEUE makes raised events present in a later big-step. This is not an option mentioned in the BSMLs paper, but it was the standard behavior of the original version of SCCD. It is included merely for completeness. A major drawback of this option is the unpredictability of when a raised event will be responded to: raised

events are added to the global event queue, where they may be interleaved with input events.

The second exception is the Internal Event Lifeline option SAME, which is only to be used with concurrency-semantics. Although SCCD currently does not support *true* concurrency, there is a concurrency option, allowing more than one transition to be included in a small-step. When this option is chosen, an event raised by one transition may be responded to by another transition in the same small-step. There is however still a causality relation between the transition raising the event, and the transition responding to it. It is therefore not a "true concurrency" and the same semantics can be achieved by replacing the small-steps with combo-steps.

3.4.7 Transition candidate generation

In order to execute a small-step, conceptually, a set of transition candidates is generated, and from this set, a candidate is chosen, depending on Priority semantics. This candidate is fired, concluding the small-step.

The set of transition candidates can be understood as a *series of filters* applied to the set of all transitions. Those filters are:

1. Is the transition's source-state part of the statechart's **current configuration**?
2. Is the transition's trigger is satisfied with respect to the set of **currently enabled events**? No distinction is made here between input events or internal events.
3. Does the transition's arena overlap with any of the **forbidden arenas**? This set is determined by the maximality-semantics chosen.
4. Is the transition's **guard condition** satisfied with respect to the current state of the statechart's memory?

We'll refer to these filters by their number in the enumeration above.

Efficient candidate generation

These filters could be applied in any order without changing the result, but not all orders have the same performance: Filter (4) is a potentially expensive operation (it may cause a billion decimals of π to be calculated), so it is best to apply this filter at the end, when as many transition candidates as possible have already been eliminated.

The other filters, (1), (2) and (3), have the interesting property, that for the same input (current configuration, currently enabled events, forbidden arenas), they will give the same output: A transition's source state, trigger, or arena does never change! This means that, theoretically, we can pre-calculate their results for all inputs. However, the number of possible inputs would be too large, to compute them all.

A better alternative is to *cache* computed semi-filtered sets of candidates at runtime[1]. This is possible since all filter inputs are representable as bitmaps (including the set of

[1]Credit for this feature originally goes to Simon Van Mierlo

enabled events, if we assign a unique integer ID to every event), so they (and tuples of them) can serve as keys in a hashtable or search tree. We simply use Python's dictionary type, which uses a hashtable.

The rationale behind caching transition candidates is that a statechart will re-visit the same state configuration / enabled events / forbidden arenas during its lifetime. Some consideration needs to be made when choosing our caching key. We could cache the candidate sets of applying filters (1), (2) and (3), but then our cache be highly specific, making it grow very large, and *cache hits* would be rare. The opposite choice of a non-specific filter, such as only (1), would make the cache smaller and cache hits more frequent, but would yield candidate sets that still have to be filtered further by the unapplied filters, possibly reducing performance.

Filter key to be cached can be swapped out in the SCCD runtime. They are called GeneratorStrategies (see Figure 3.23). 2 strategies are included:

CurrentConfigStrategy Computes and caches candidates based on the tuple (current configuration, forbidden arenas), then filters based on enabled events, and finally filters on guard condition evaluation.

EnabledEventsStrategy Computes and caches candidates based on the tuple (enabled events, forbidden arenas), then filters based on current configuration, and finally filters on guard condition evaluation.

A performance comparison of these strategies, plus additional ones could be considered future work.

Cache initialization: Partial pre-computation of candidates

We have seen that complete pre-computation of transition candidates is intractable. We can however initialize the candidate-cache with pre-computed values that are expected to be requested frequently. This would pass for partial pre-computation of candidates, and mixes nicely with caching.

Currently, pre-computation is only implemented for the EnabledEventsStrategy, computing candidate sets for the following values:

- (0, 0): No current events and no forbidden arenas. (Containing all triggerless transitions)
- ($\{e\}$, 0): Where $\{e\}$ is the singleton-set of event e, e being any event that serves as a trigger for a transition in the statechart. No forbidden arenas.

This pre-computation scheme significantly increases the number of cache hits at the beginning of a statechart's execution, since very frequently, the set of currently enabled events consists only of a single event, or of no events.

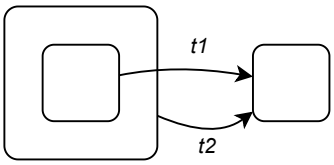


Figure 3.26: Transitions with different source states, but the same arena

Priority implementation

In the BSML paper, Priority and Order of Small Steps (OOSS) are considered separate semantic aspects, but in our implementation they are considered one and the same option. We described in Section 2.2.8 how they both produce partial orderings on the transitions in a statechart, and how their combination can produce a total ordering, in which case the model behaves deterministically, in a way that the modeler has full control over.

We have talked about *sets* of transition candidates, but we never deal with (unordered) set types in the implementation. Instead, we statically build a *list* of total ordering of all transitions in the statechart. To this list, a series of filters is applied as explained above, in an order-preserving way, producing a new list of candates, ordered by priority. If this candidate list is non-empty, its first element is always the selected candidate.

We always use document order of the XML format for Explicit Ordering of small-steps and for Explicit Priority of transitions with the same source state. Additionally, one of the following Hierarchical options can be chosen:

Source-Parent The higher a transition's source in the state hierarchy, the higher its priority

Source-Child The lower a transition's source in the state hierarchy, the higher its priority

Arena-Parent The higher a transition's arena in the state hierarchy, the higher its priority

Arena-Child The lower a transition's source in the state hierarchy, the higher its priority

To construct the priority-ordered list of transitions, we start with a list of the transitions in XML document order. We then sort this list with a sort key matching the Hierarchical option chosen, using a stable sort algorithm.

Note that the arena-options may not yield a total ordering of all transitions in the statechart, since in SCCD, there is no way to explicitly define a priority between transitions that have different source states, but share the same arena (e.g. Figure 3.26). The model will behave deterministically in the sense that an order (hence, priority) between the transitions exists in the XML document, but this order is typically not presented or controllable by the modeler in the visual view of the model. These options were implemented nevertheless, because it was only a small task to do so.

Lazy evaluation of transition candidates

We explained how the list of transition candidates is the result of applying a sequence of filters on a totally ordered list of transitions. Every consecutive filter operation could produce a new (smaller) list, but we only fully construct such a (intermediate) list if the result is being cached. Otherwise, we *lazily evaluate* the list, as we are only interested in its first element (= the next transition to fire), or whether the list is empty (= no more enabled transitions). For lazy evaluation, we use Python's generator expressions.

3.4.8 Memory snapshots

SCCD supports the Enabledness Memory Protocol and Assignment Memory Protocol semantic aspects. (Almost always, one would chose the same option for both Enabledness and Assignment Memory Protocol.) For each aspect, the options Big Step, Combo Step and Small Step exist. These options determine the "moment in time" that action language expressions "see" when evaluating variables. This moment in time is always the beginning of some step: E.g. if the Combo Step-option is chosen, all guard conditions evaluated and transitions executed as part of the combo-step will read the statechart's variables as they were at the beginning of the combo-step.

This behavior is trivially implemented by creating a snapshot of a statechart's memory at the beginning of the step chosen in the Memory Protocol option. The snapshots are used for reading, while the actual memory is used for writing.

For any given statechart, the size of the snapshot is always the same. Snapshots are taken in between transitions, when the only stack frames in the statechart's memory are the "builtin" and "instance" scopes. The "builtin"-scope is read-only, so does not have to be snapshotted. Therefore the size of every snapshot is equal to the size of the "instance"-scope, which is statically computed from the statechart's datamodel section (see Section 3.4.4).

The snapshotting is implemented in the `MemoryPartialSnapshot` class (Figure 3.27). It wraps around a normal `Memory` object (the default implementation of `MemoryInterface`, expected by action language constructs for execution), which it uses as a delegate for all write-operations, as well as read-operations outside of the snapshotted memory (i.e. builtin variables and temporary variables, local to the action code fragment being executed). At the end of every Memory Protocol-step, the snapshot is refreshed (`flush`). This is done by registering a callback on the correct `Round` object.

Race condition detection

If more than one transition writes a value to the same location in memory during the same Memory Protocol-step, a race condition exists. Race conditions are detected by maintaining a bitmap (`round_dirty` in `MemoryPartialSnapshot`) of memory locations written during the current Memory Protocol-step. A memory location written twice causes a *runtime error*. Race conditions cannot be detected statically. At the end of every Memory Protocol-step, when the snapshot is refreshed, the bitmap is reset to 0.

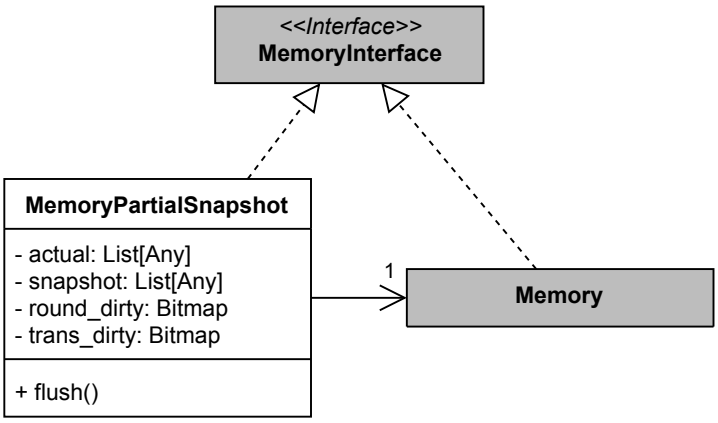


Figure 3.27: Class diagram of MemoryPartialSnapshot class.
Dark-gray classes defined in action language package.

Locally observable writes

It would be counter-intuitive if due to the Memory Protocol option chosen (e.g. COMBO STEP), a transition writing to a statechart's variable x would not be able to consequently read the variable x and retrieve the value it had just written. This is taken care of in `MemoryPartialSnapshot` by maintaining another bitmap, `trans_dirty` of memory locations written during the current transition. For these locations, consequent read operations are performed on the actual memory, not the snapshot. Consequent writes to these locations are also not erroneously detected as race conditions.

3.4.9 Various optimizations

In the previous sections, the following performance optimizations were described:

- Use of bitmaps to represent sets of states and sets of events.
- Static calculation of a transition's arena, and arena's descendants, allowing for efficient calculation of a transition's *exited states* when firing a transition.
- Static calculation of a transition's *entered states*.
- Swappable transition candidate generation strategies, with caching.
- Lazy evaluation of transition candidates.

In this section, we discuss a few optimizations that haven't been covered yet.

No semantic-option-dependent conditional branches in execution loop

The semantics of a statechart model never change during execution, therefore, no sort of execution loop should waste time checking them with each iteration.

Upon initialization of a statechart instance, based on the semantics defined in the statechart model, objects and relationships between them are constructed in a way that they will carry out statechart execution according to the semantics specified, but the objects themselves are unaware of the semantics they represent.

For instance, Figure 3.25 shows the construction of different hierarchical `Round`-structures, as constructed based on different maximality-semantics, but the `Round`-objects themselves do not have access to the semantic configuration of the model.

Converting all durations to the same unit

The action language has a builtin *time duration* type. Duration-literals make time durations unambiguous by including a duration-suffix (e.g. `ms` for milliseconds). In a statechart, the duration type is expected for the delay-value of timed transitions.

Internally, a duration is an integer value with a *unit*. Units reflect the duration-suffixes used, but upon construction, every duration is automatically converted to the largest possible unit, without losing information, called *normalization*. For arithmetical operations between durations, such as addition or subtraction, they have to be converted to their *greatest common divisor*-unit before the operation can be performed, again followed by normalization. Units have no absolute "size" (they are defined only relative to each other), offering great flexibility and an unlimited range of units to be defined, but also making these duration conversions expensive operations.

Because units can be arbitrarily mixed in a statechart, potentially requiring many unit conversions during execution, hurting perfomance, all durations are transparently converted to *integer* values representing multiples of the same time unit (or more precisely: time duration), called the *model delta*. The model delta is the smallest representable amount of time during model execution. Even if a model only contains durations in the order of seconds, it is wise to set the model delta much smaller, because the timestamps assigned to input events are also expressed in model delta units, and are rounded down if necessary. By default, a model delta of $100\mu s$ is used, allowing for maximum durations (and timestamps) of $5.8 * 10^7$ years if 64-bit unsigned integers are used for timestamps. The model delta can be overriden in the model.

Python-specific optimizations

Our implementation contains some optimizations specific to Python, the languge in which it is written:

Native integer type vs. SCCD's Bitmap class Bitmaps are used in many parts of the code. A `Bitmap` type was defined, inheriting Python's integer type, overriding its string-representation `__str__`-method to return a bitstring instead of a decimal number. Arithmetic operators between Bitmaps were also overridden to return Bitmaps, not integers. It was found however, that this had a significant negative impact on performance, simple arithmetic operations on bitmaps requiring function calls. The solution was to disable our custom bitmap implementation when the environment variable `SCCDDEBUG` was not set, falling back to native integers. The

Python language is very flexible in this regard, the solution having the following form:

```
if DEBUG:
  class Bitmap:
    ...
else:
  Bitmap = int
```

With this modification alone, when executing all models from the test framework, the average time of executing a transition went from 0.2 ms to 0.16 ms, a 20% speedup!

Slotted classes By default, Python allows arbitrary attributes to be added to objects. This is implemented by storing the attributes of objects in (modifiable) dictionaries. This introduces a significant memory overhead and slows down attribute access. Python offers an alternative in the form of *slotted classes*, that declare a sequence of fields to always be part of the object.

Slotted classes have drawbacks, such as not supporting multiple inheritance. They also require additional syntax, for the declaration of the "slots". We *make the common case fast* and use slotted classes in much of the `statechart.static` package defining syntactic constructs, whose attributes are constantly accessed during execution.

3.5 Controller Implementation

The Controller is the main primitive for executing SCCD models.

3.5.1 SCCD models

An SCCD model is at the highest level a *class diagram* (CD) of statechart instances that may be created and destroyed at runtime. In this thesis, the class diagram always consists of a single class, of which a single instance is created. To work with these models, we developed an ad-hoc XML format with an extremely simple structure:

```
<single_instance_cd>
  <model_delta>100 us</model_delta>
  <statechart> ... </statechart>
</single_instance_cd>
```

The `<model_delta>` node is optional and sets the "model delta", the smallest possible time duration that can be simulated in the model. The `<statechart>` node is the root node of a model in our statecharts XML format, described in Section 3.4.1.

This format is parsed into a `SingleInstanceCD` object (Figure 3.29), implementing the `AbstractCD` interface for SCCD models. The `SingleInstanceCD` object can be passed to the Controller's constructor.

3.5.2 Controller Interface

The Controller's interface is a primitive for executing SCCD models. The Controller maintains a single global *priority queue* of timestamped events. Events serve as input for one or more statechart instances. (An event with more than one target instance is a multicast or broadcast event.) Timestamps in the Controller are just integers, they have no physical meaning, only logical.

The Controller maintains an integer value of *simulated time*, which is 0 right after the Controller's creation, and always equal to the timestamp of the most recently popped event from the queue. The simulated time can only increase (or stay the same).

These are the Controller's most important methods, towards its environment:

`add_input(timestamp: int, ..., event_name: str, ...)` Adds an input event to the controller's queue.

`run_until(timestap: int)` Advances simulated time. In a loop, pops events from the queue and delivers them as input to their target instances. The target instances respond to the input by making a big-step. The call does not return until either (1) the queue is empty, or (2) the next event to be popped would advance simulated time further than the `timestamp` parameter of the call.

The following fragment is almost literally taken from the source code:

```
def run_until(self, now: int):
    for timestamp, entry in self.queue.due(now):
        self.simulated_time = timestamp
        for instance in entry.targets:
            instance.big_step([entry.event])
```

`next_wakeup(): int` Getter. Returns the timestamp of the earliest event in the controller's queue.

All methods are synchronous (taking control of the current thread until they return). Using these 3 methods, the 3 different runtime *platforms* foreseen in the original version of SCCD can be implemented. Section 3.5.4 contains a description of these platforms and a demonstration of their implementation using the Controller as a primitive.

Integer timestamps

Figure 3.28 shows how the Controller only "talks" integer timestamps with its environment, as well as with its model. If meaning must be given to the Controller's timestamps (e.g. for real-time simulation), it is up to the environment to query the model for its

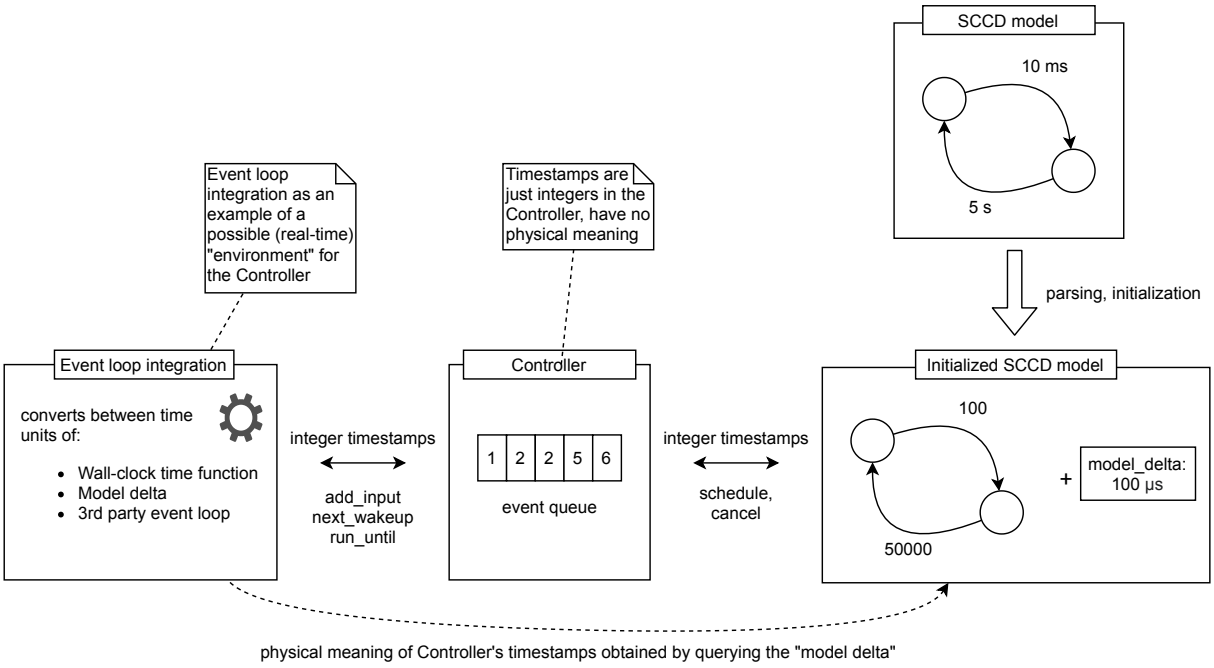


Figure 3.28: The Controller only "talks" integer timestamps

"model delta", which is a time duration of which the controller's timestamps are multiples (see also Section 3.4.9).

These are the motivations for making the Controller work with timestamps only as logical entities:

- The Controller's environment may not care about real-time simulation: E.g. when running a model test (a model + timed inputs + expected, timed outputs), we are only interested in the correctness wrt. its output, and run the model as fast as possible.
- It allows the Controller's timestamps to be interpreted as a wide range of time units. This could also be done by using our `Duration` type (which also offers a wide range of units) instead of raw integers, but its performance is worse, because of possible unit-conversions.
- Following the single-responsibility principle of OO-design, the Controller's purpose is to queue incoming events, and to handle them in the right order, when simulated time is requested to advance, and nothing more.

Output

It was mentioned before that statecharts can produce output in several ways:

1. As *output events*, produced as the result of a big-step.

2. By assigning values to *output variables*, that can be read at any time by the environment.

3. By calling *synchronous functions* from action code, i.e. during the firing of a transition.

Of these, SCCD supports (1) and (3). Output variables are currently not supported.

In BSML, output events are unordered sets, produced at the end of every big-step. The Controller deviates here by delivering every output event immediately, when it is raised, via a callback function, i.e. during transition firing and from the controller's thread. This has the following benefits:

- It saves the overhead of adding output events to an output queue.
- If desired, the environment can still queue output events.

The implementation of the callback function should take care not to block, as it will block the model's execution.

No attempt is made to add output events of a big-step together in a "bag of events", but a special event signals the end of a big-step, so these "bags" can still be constructed (e.g. the test framework does this)

3.5.3 Design

Figure 3.29 shows the class diagram of the `Controller` class and several other classes that play a role in SCCD model execution. The Controller has an `EventQueue` to which input events are added. The Controller also has an `ObjectManager`, which maintains a list of all created instances during execution (in our case, only a single instance), and checks adherence of these instances to the class diagram (not yet implemented in our branch). `Instance`s perform *big-steps* in response to a set of input events.

`AbstractCD` is an interface that loaded SCCD models implement. The `SingleInstanceCD` class is an ad-hoc implementation of this interface for models that at all times only consist of a single statechart instance. The Controller receives an `AbstractCD` model as its constructor parameter, and passes it on to the ObjectManager, which uses it to instantiate the *default class* (in our case, the single statechart the model consists of).

The classes below the dotted line, `EventLoop` and `EventLoopImplementation` are not part of the "core runtime", and optionally wrap around the controller. They are mentioned in the next section.

3.5.4 Runtime platforms

An important feature in original SCCD was the ability to select a type of runtime "platform" for generated code. The available platforms were (Figure 3.30):

Event Loop The event loop platform is intended for easy integration with existing 3rd party event loop implementations, as typically found in UI toolkits, like TkInter.

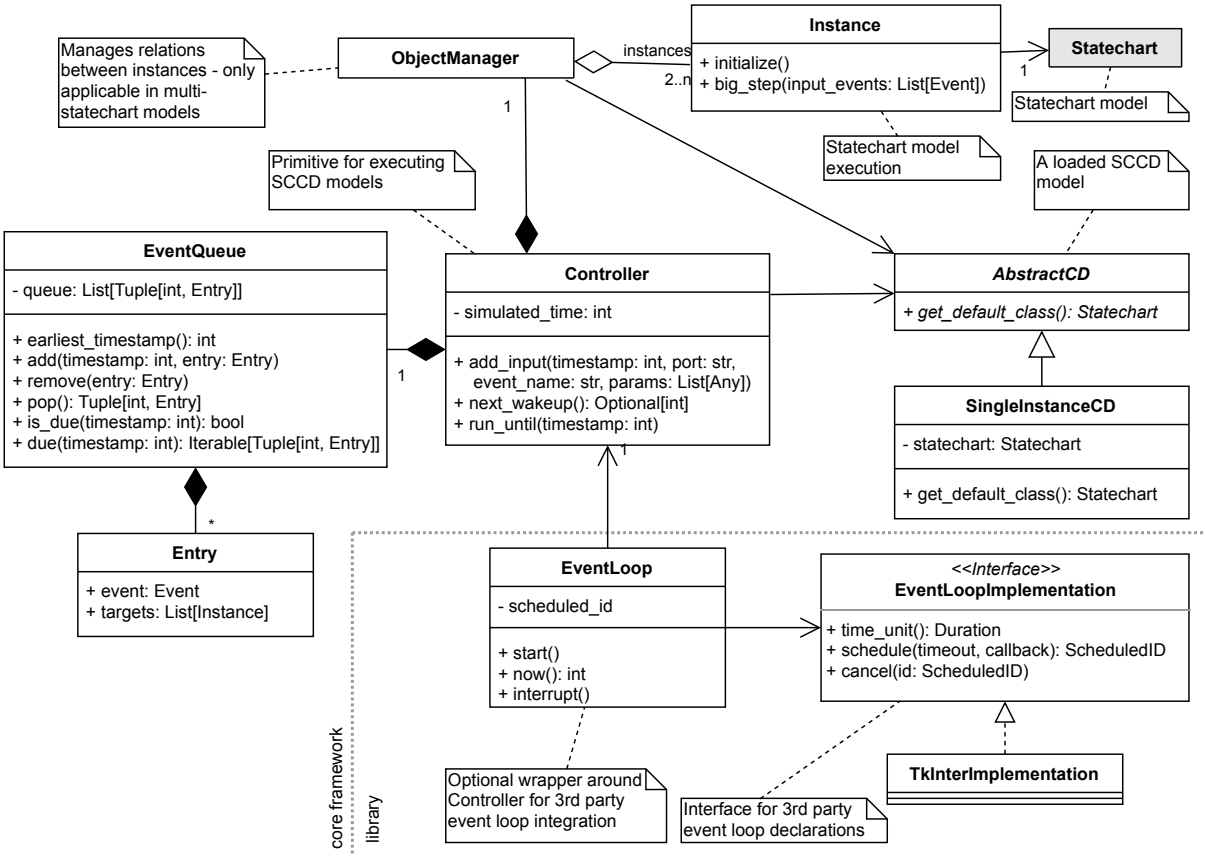


Figure 3.29: Class diagram of Controller and other classes involved in SCCD model execution

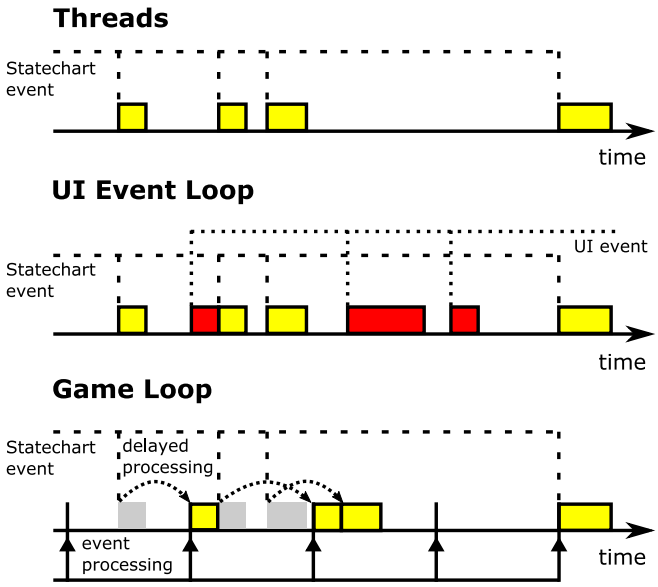


Figure 3.30: Platforms supported in original SCCD.
Figure taken from [17].

This platform can be integrated with any existing event loop implementation that offers functions for

1. scheduling a future callback
2. clearing a scheduled callback

Using these callbacks, the platform will attempt to let the statechart simulation run in sync with wall-clock time, scheduling periods of statechart execution (i.e. responding to a statechart input events) as tasks in the 3rd party event loop. The statechart execution would thus run from the same thread as the 3rd party event loop, interleaving it with e.g. UI events. The runtime library comes with implementations for TkInter and JavaScript (browser, NodeJS).

Threads The threads platform has its own native event loop implementation, using only the target language's standard library. This event loop implementation will also attempt to run the simulation in real-time. When it is started, the threads platform takes over the current thread, so the user is required to write any input or output logic in separate threads.

Game Loop The game loop platform is perhaps the simplest of all, as it only offers a function that advances the simulated time to "now" (wall-clock time). The function is blocking and runs as-fast-as-possible. Traditionally, with each iteration of a game loop, the function would be called to update "the world", followed by rendering the display.

Implementation with Controller primitive

In original SCCD [17], these platforms were explicitly implemented side-by-side, but shared a lot of common (duplicated) logic. Now, with the Controller's interface as a primitive, all 3 "platforms" can be built, as we demonstrate.

The most trivial is *game loop* integration, where with each iteration, the simulated time advances up to the current wall-clock time:

```
start_time = now()
while True:
  controller.add_input( ... ) # e.g. keyboard or mouse events
  controller.run_until(now() - start_time)
  render()
```

The *threads* platform is slightly more complicated, at least if we want to run the simulation in real time. We use a `threading.Condition` object instead of the sleep-function, in order to be woken up when there is an input:

```
input_queue = queue.Queue() # thread-safe queue
condition = threading.Condition()

def controller_thread():
  while True:
    while event := input_queue.pop():
      controller.add_input(..., event, ...)
    controller.run_until(now()) # this may take a while
    sleep_duration = controller.next_wakeup() - now():
    if sleep_duration > 0:
      with condition:
        condition.wait(sleep_duration)

def add_input(event):
  input_queue.put(event)
  with condition:
    condition.notify() # wake up controller thread

thread = threading.Thread(target=controller_thread)
thread.start()
```

The *event loop* platform is similar, but instead of sleeping, we schedule a future callback in the event loop we are integrating with. Also, we do not need a thread-safe input queue, as it is safe to directly call `controller.add_input` from everywhere. The event loop platform is as follows:

```
scheduled_id = None
def run():
```

```
  controller.run_until(now()) # this may take a while
  sleep_duration = controller.next_wakeup - now():
  if sleep_duration > 0:
    scheduled_id = schedule(sleep_duration, run)
  else:
    scheduled_id = schedule(0, run)
schedule(0, run) # start controller when event loop starts

def add_input(event):
  controller.add_input(..., event, ...)
  # "wake up":
  cancel(scheduled_id)
  run()
```

Finally, an example from our *test framework*, where for the execution of a test case, all input events are known from the beginning, and simulation runs as fast as possible. The controller runs in its own thread, so that test output can be verified in parallel:

```
for i in test_case.input:
  controller.add_input(..., i.event, ...)

pipe = queue.Queue() # thread-safe queue

def controller_thread():
  controller.run_until(None) # None here means: +infinity (return when
      event queue empty)
  pipe.put(None) # signal end of run

thread = threading.Thread(target=controller_thread)
thread.start()

while True:
  output = pipe.get()
  ... # verify test output
```

Event loop library

Part of the SCCD project is a library implementing the logic from the above *event loop* example as a class, called `EventLoop`. Figure 3.29 shows how this class wraps around the Controller. In reality, the class does more than shown in the example, because:

1. The schedule-callback of the chosen event loop implementation may expect a timeout in a different unit than the (Python) time function in use, which may in turn differ from the time unit expected by the model itself. See also Figure 3.28.

2. If the simulation continuously cannot keep up with the wall clock time (e.g. because the computer is too slow), with a naive implementation, invocations of `run_until` will take longer with each invocation, increasingly starving other tasks (such as adding input events).

To solve the first problem, event loop implementations are communicated with the `EventLoop` class in a declarative manner (see `EventLoopImplementation in Figure 3.29`). Similarly, the time function and its unit are also declared (not pictured). Because time unit conversions have to happen all the time, the conversion ratios are pre-calculated for efficiency.

To solve the second problem, it is checked whether the `sleep_duration` variable calculated is a negative number. This indicates wall-clock time running faster than our simulation. If this is the case, the negative number is added to the parameter of `run_until` in the next round (reducing the amount of time-to-simulate), purposefully making the model run at a reduced speed. This solution was evaluated to effectively keep the model responsive to input under heavy load. The simulation would also recover ("catch up") with wall-clock time when load was reduced.

3.6 Important changes from original SCCD

As mentioned in the beginning of this chapter, our fork of SCCD differs significantly from the original. The most drastic change is that SCCD is no longer a code generator. We'll first motivate our decision of abandoning the code generation approach. Next, we explain how the 3 different runtime platforms (each with their own interface) are no longer a fundamental part of the "core" runtime, and instead use a single primitive interface usable by all 3 platforms. Finally, we list a number of smaller improvements.

3.6.1 No more code generation

In order to support multiple target languages without having to replicate much of the compilation logic, the original SCCD compiler transformed input models to an intermediate procedural language. This language did not have a textual syntax, only abstract. It was developed to be the greatest common divisor of the target languages supported. All supported target languages (Python, JavaScript, C#) were similar enough (procedural, object-oriented and garbage collected) to make this approach work.

However, it would have been non-trivial, to add a language like C, because of its manual memory management. This limitation contributed to abandoning this compilation strategy.

Another, more important problem, was that the part of the compiler that generated the intermediate language code (or better: AST) was very hard to understand. The construction of the AST tree looked nothing like readable code. Some improvement was

made by using a stateful "writer" object to build the syntax tree, but then still identifiers in the target language were strings in the compiler code, making it impossible to statically check them for errors. Example:

```
self.writer.beginConstructor()
self.writer.addFormalParameter("controller")
self.writer.beginMethodBody()
self.writer.beginSuperClassConstructorCall("ObjectManagerBase")
self.writer.addActualParameter("controller")
self.writer.endSuperClassConstructorCall()
self.writer.endMethodBody()
self.writer.endConstructor()
```

Listing 3.1: A fragment of `generic_generator.py` of original SCCD

Finally, perhaps the most important reason for abandoning the code generator, was the fact that the execution runtime library had already become the place where most of the complexity was located: it was where most of the semantic variability was implemented, as well as a few runtime optimizations (such as transition candidate caching[2]).

In order to extend SCCD with additional semantic variability, in a clean way, the decision had to be made whether to move all complexity to the code generator, or to the execution runtime. Because of the reasons just mentioned, the latter was chosen.

Our decision to go with the runtime approach still does not rule out the future addition of a code generator to the SCCD project: The insights obtained from implementing statechart semantic variability as a runtime may be used as a foundation for a code generator.

3.6.2 No more explicit runtime platforms

In original SCCD, there were 3 types of controllers, each implementing one of the 3 runtime platforms described in Section 3.5.4.

Limitations

The original, side-by-side implementation of the 3 platforms had some shortcomings:

- Complex design: The available platforms were implemented in the SCCD runtime library, so, in principle, they should have been selectable independently of the compilation step. However, due to contrains in the original design, caused by heavy reliance on class inheritance (with many overrides), the platform had to be selected at compile time, and the generated code depended on a specific platform. This is shown in Figure 3.31.

- Use of floating point numbers for timestamps: Older Python versions use floats for timestamps, so this choice seemed natural. However, floats have non-uniform

[2]Credit for this feature goes to Simon Van Mierlo

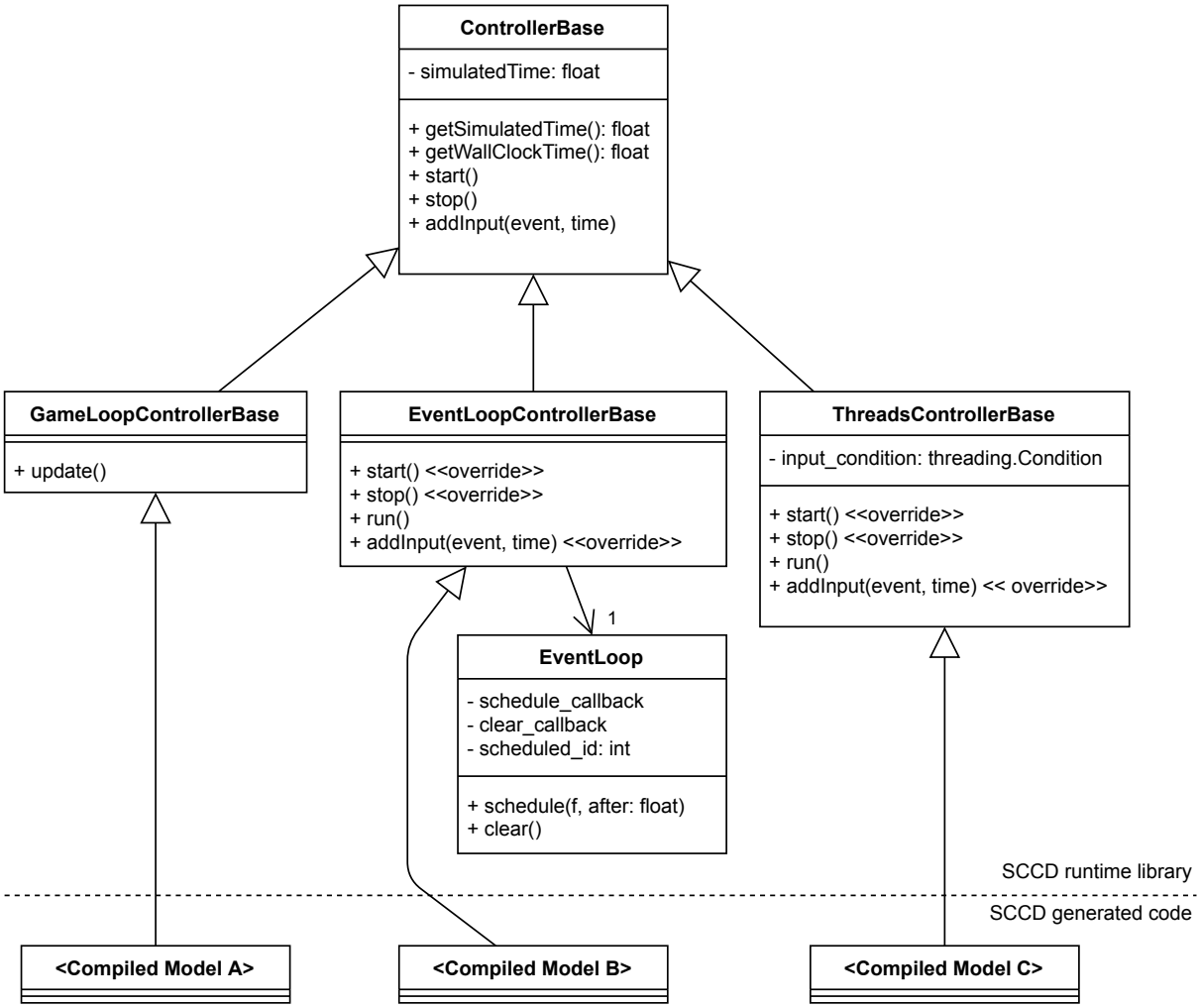


Figure 3.31: Class diagram of original SCCD's implementation of the 3 platforms

precision. At some point, this led to a problem, where a very small update in the simulated time would be rounded down, unexpectedly not incrementing the simulated time, and freezing the simulation. A workaround was developed, detecting this issue, and forcing the result to be rounded up. However, the use of integer timestamps is a much safer choice, for its uniform, predictable precision. Also, since Python 3.7, a more precise time function exists, returning time as an integer.

- Wall-clock time-aware: All `Controller`-classes would query the current wall-clock time, to attempt to let the simulation run in (soft) real time. In some cases however, it is desirable to run a simulation at lower or higher speed. For instance, when (non-interactively) testing a model, the simulation typically needs to run at the highest possible speed, to produce results (test passed/failed) as soon as possible. Simulations for scientific research would also be run at the highest possible speed, if one is only interested in the eventual outcome.

Solution

As explained in Section 3.5.4, there is now a single Controller primitive that can be used to create any of the 3 runtime platforms, and can be used to run the simulation at any speed relative to wall-clock time, including as-fast-as-possible execution. The event loop platform is included as a library that wraps around the controller.

3.6.3 Single event queue

In the original version of SCCD, every statechart instance had its own event queue, as well as a separate event queue in the Controller. The vague intent was to allow parellel execution of instances, but this goal was never accomplished. The existance of multiple event queues made it complex to find the timestamp of the next event to be processed (having to check all queues).

Now, there is only a single event queue, in the Controller, for containing the input events for all statechart instances in the runtime. Hypothetically, parallel execution is still possible by making the event queue thread-safe, and sharing it among "worker threads".

3.6.4 Duration units

Treating time durations as numbers is confusing, because the unit may differ depending on the context. All time durations in after-transitions in original SCCD were expressed in seconds, as floating point values. Now, every time duration has a mandatory suffix, and the action language's static type system considers durations their own type, and cannot simply be casted to or from a number. Suffixes are 'fs' (femtoseconds), 'ps' (picoseconds), 'ns' (nanoseconds), 'us' (microseconds), 'ms' (milliseconds), 's' (seconds), 'm' (minutes), 'h' (hours) and 'D' (days).

3.6.5 More powerful test framework

Since we are interested in comparing the behavior of different semantic configurations, it is useful to see how the same statechart model behaves if we change the semantics. In original SCCD, the semantic configuration was inseparable of the model, which makes sense, because a model is usually developed only with a specific semantic configuration in mind. So now in SCCD, this is currently still the case, but it is also possible to share statechart models between test cases, and override the semantics, achieving our goal.

Another useful feature is that when defining a semantic configuration for a test case, for each semantic aspect, one can use the wildcard symbol "$*$" or a comma-separated list to express that multiple options should cause the test to succeed. If multiple options are chosen for multiple semantic aspects, all configurations making up the cartesian product are tested.

Finally, one can denote tests that are expected to fail with the filename prefix `fail_` instead of the usual prefix `test_`. This feature is mostly used to check whether syntactically invalid models are detected as such.

CHAPTER 4

Evaluation

In this chapter, we first compare our solution to YAKINDU, studying its semantics and overall feature-set. Next, we demonstrate 2 example models, implemented in SCCD and YAKINDU, and discuss the chosen semantic options for those models. Finally, we discuss some generally applicable insights obtained through implementation of the example models.

4.1 Comparison study: YAKINDU

YAKINDU Statechart Tools [6] is a well-known open-source tool for statechart development. It features a graphical interface for editing and simulating statechart models, and can compile models, generate optimized code for Java, C, and C++. It also features an advanced model testing framework.

In this section, we first study YAKINDU's semantics and attempt to describe them as a BSML. We then compare YAKINDU's features to SCCD.

4.1.1 YAKINDU's semantics

YAKINDU features some semantic variation points, although fewer than SCCD. We discuss each of the variation points and map them onto BSML.

Cycle-based and event-driven execution

At the very highest level, the modeler can choose between 2 execution algorithms: *Cycle-based* and *event-driven* execution. From the viewpoint of BSML, each algorithm represents a set of choices for several semantic options.

	Cycle-based	**Event-driven**
Big-Step Maximality	Take One superstep: Take Many	Take Many
Combo-Step Maximality	-	Combo Take One superstep: Combo Take Many
Input Event Lifeline	Present in Whole	Present in First Combo Step
Internal Event Lifeline	Present in Remainder	Combo-Queued
Priority	Hierarchical (Source-Parent or Source-Child), Explicit	
Order of Small Steps	Explicit	
Memory Protocol	Small Step	
Concurrency	Single	

Table 4.1: Semantics of YAKINDU

With **cycle-based execution**, the statechart responds at regular intervals to a set of simultaneously enabled input events, by performing a *run-to-completion step* (RTC step). During an RTC step, every region (YAKINDU's term for an Or-state) may fire a single transition. Internal events raised during transition execution are simply added to the set of enabled events of the current RTC step, for the remainder of unfired regions to possibly respond to. When no more transitions can be made, the RTC step is complete.

These semantics clearly map onto BSML. An RTC step here is a big-step, if we choose the semantic options Big-Step Maximality: Take One, Input Event Lifeline: Present in Whole and Internal Event Lifeline: Present in Remainder.

With **event-driven execution**, the statechart performs an RTC step immediately for every raised input event. As a result, during every RTC step, exactly one event is enabled. The semantics of the RTC step itself are similar as with cycle-based execution, as every region is allowed to fire a single transition. Internally raised events, however, are not added to the set of enabled events of the current RTC step, but are put in a FIFO-queue of internally raised events. When the RTC step is over, another RTC step is immediately executed with as input the next event in the queue (giving higher priority to internal events than input events). This is repeated until the queue is empty.

These semantics do not map perfectly onto BSML. A "close" semantic configuration is when we treat RTC steps as combo-steps: Big-Step Maximalty: Take Many, Combo Step Maximality: Combo Take One, Input Event Lifeline: Present in First Combo Step and Internal Event Lifeline: Present in Next Combo Step. However, this configuration will accumulate all internally raised events of combo step n into enabled events of combo step $(n+1)$ (combo steps possibly having multiple simultaneously enabled events), as opposed to YAKINDU's semantics of starting a "combo" step per enabled event.

SCCD has an Internal Event Lifeline option called Queue which somewhat resembles YAKINDU's event-driven execution. It appends internally raised events to the global (external to the statechart) event queue, but this causes them to be potentially handled

after certain input events (if those input events have the same timestamp as the internal events, and pre-existed in the queue). YAKINDU, on the other hand, gives priority to internally raised events, by maintaining a separate queue for them.

Let's invent a new Internal Event Lifeline option for YAKINDU, called Combo-Queued, queueing internal events and handling them in FIFO order, one event per combo step. This option combined with the other options from our "close" configuration accurately describes YAKINDU's event-driven execution semantics.

Superstep semantics

Another semantic option, altering the behavior of RTC steps is the *superstep* option (to be included in their next release, version 4.0). Enabling this option allows regions to fire as many transitions per RTC step as they can. We did not test the precise semantics of this option, but from the execution pseudocode in the documentation[1], it seems that RTC steps use *fairness*, giving other regions a chance to fire a transition before the same region is given a chance again.

Mapping this option to BSML, for cycle-based execution, this changes Big-Step Maximality from Take One to Take Many (with fairness). Other semantic options remain the same.

For event-driven execution, we still handle one event per combo step, but a single event may now cause more than one transition to fire in the same region, so Combo Take One becomes Combo Take Many (with fairness). Other semantic options remain the same.

Child/parent first

For every statechart in YAKINDU, a choice must be made between the *parent-first* or *child-first* execution. There is no need to explain these options, as they simply map onto BSML's Hierarchical Priority options: Source-Parent and Source-Child.

Explicit outgoing transition priority

In order to avoid non-determinism, the outgoing transitions of a state always have an explicit priority relative to each other. If a state has multiple outgoing transitions, this explicit priority is graphically displayed as a number on every transition. The outgoing transitions' relative priorities are part of the syntax, and can be edited for every state. This behavior trivially maps onto Priority: Explicit.

Explicit region order

In YAKINDU, regardless of the rest of the execution semantics, regions get a chance on-by-one to fire a transition. Just like the order of outgoing transitions, the order in which regions are allowed to fire transitions, is an explicit part of the syntax, i.e. left-to-right, top-to-bottom in the graphical layout of the statechart. In BSML, this would be considered the option Order of Small Steps: Explicit.

[1] https://www.itemis.com/en/yakindu/state-machine/documentation/user-guide/sclang_definition_section#sclang_superstep

The combination of hierarchical priority, explicit outgoing transition priority, and explicit region order yields a total ordering on all transitions in the statechart, yielding fully deterministic behavior.

No memory snapshots

YAKINDU does not feature memory snapshot semantics, like SCCD: Transitions always have direct read/write access to a statechart's variables.

This behavior maps onto the Small Step options for (Enabledness / Assignment) Memory Protocol of BSML.

No concurrency

YAKINDU features no concurrency semantics.

Summary

YAKINDU offers only 3 semantic choices, with 2 options for each: cycle-based vs. event-driven execution, superstep semantics (off/on) and parent-first vs. child-first hierarchical priority.

YAKINDU's semantics map well onto BSML, if we invent a new Internal Event Lifeline option Combo-Queued. Table 4.1 shows the semantic options available in YAKINDU. By implementing the Combo-Queued option in SCCD, it would support all of YAKINDU's semantics.

4.1.2 Comparison to SCCD

Despite its narrower set of semantic options, YAKINDU offers many more features than SCCD currently does. We will give an extensive overview of YAKINDU features NOT (yet) implemented in SCCD. They may serve as inspiration for future development efforts.

Graphical statechart editor

Statecharts are a visual, topological language. The inclusion of a graphical editor is a requirement for any serious statechart development.

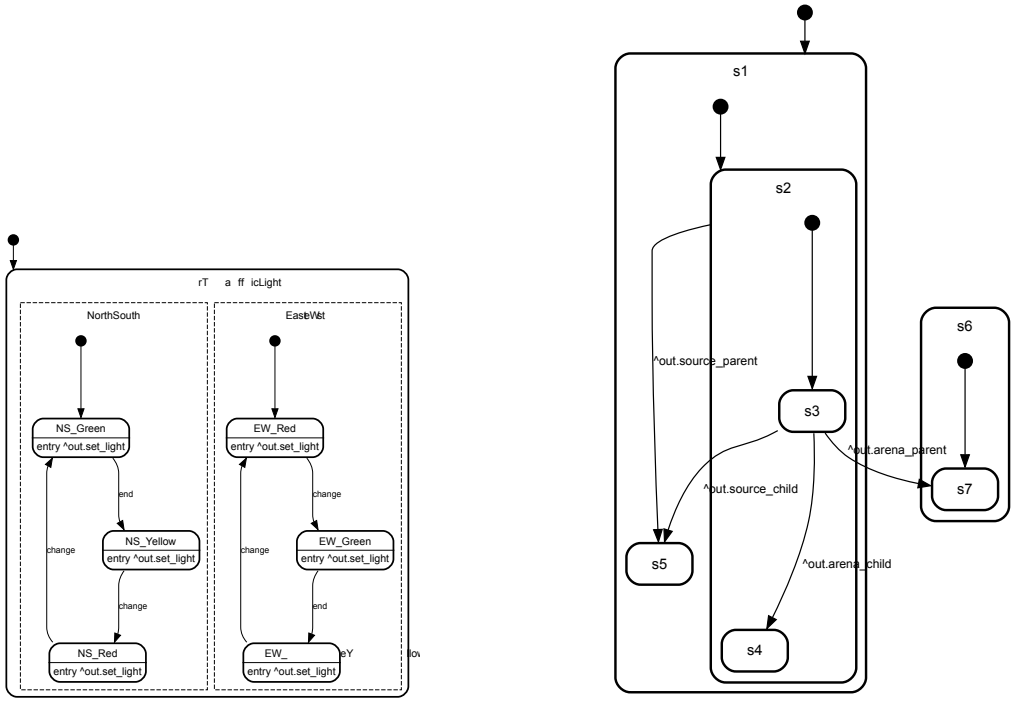
YAKINDU includes such a graphical editor. For SCCD, the intention is to eventually include a graphical editor as well, preferably modeled as a statechart itself.

Because of the burden of editing statecharts in XML format, SCCD does include a script (based on the NodeJs script state-machine-cat[2], which again is based on GraphViz[3]) that renders XML statechart files as SVG images. This has been a great help during development of SCCD, as many test files had to be maintained in XML format. Visualizing them quickly made their state hierarchy apparent.

The quality of the rendered images varies. GraphViz, the rendering backend used, can automatically generate appropriate layouts for graph data (the SCCD XML statechart format contains no layout information), but does not natively work with hierarchical graphs (or higraphs, as Harel [10] called them), so a workaround is used. This workaround

[2]https://github.com/sverweij/state-machine-cat

[3]https://graphviz.org/



(a) traffic light (b) test file for hierarchical priority semantics

Figure 4.1: Examples of SCCD statecharts automatically rendered from XML Quality of the result can be a hit-or-miss.

sometimes causes rendered statecharts to look weird. Figure 4.1 shows 2 examples of rendered statecharts.

It should also be noted that a graphical editor such as YAKINDU's puts somewhat of a burden on the modeler to choose a useful statechart layout. Especially for large statecharts with many transitions, this can be a nontrivial task, as transition labels tend to overlap with each other and with other parts of the statechart. Perhaps a hybrid approach, where the modeler is only concerned with the topology of the statechart, and possibly with layout decisions at the highest level, and the editor assisting by making (small) layout decisions, would work best.

Interactive debugging

YAKINDU can simulate and debug statechart models in its graphical editor, highlighting current states and allowing the modeler to enable input events interactively.

Syntactical constructs

YAKINDU features some syntactical constructs that SCCD does not have. Some of these can be considered syntactic sugar, while others add features to the language:

Synchronizations There are 2 types of synchronizations: The first type can be understood a transition with multiple source states, where the sources have to be

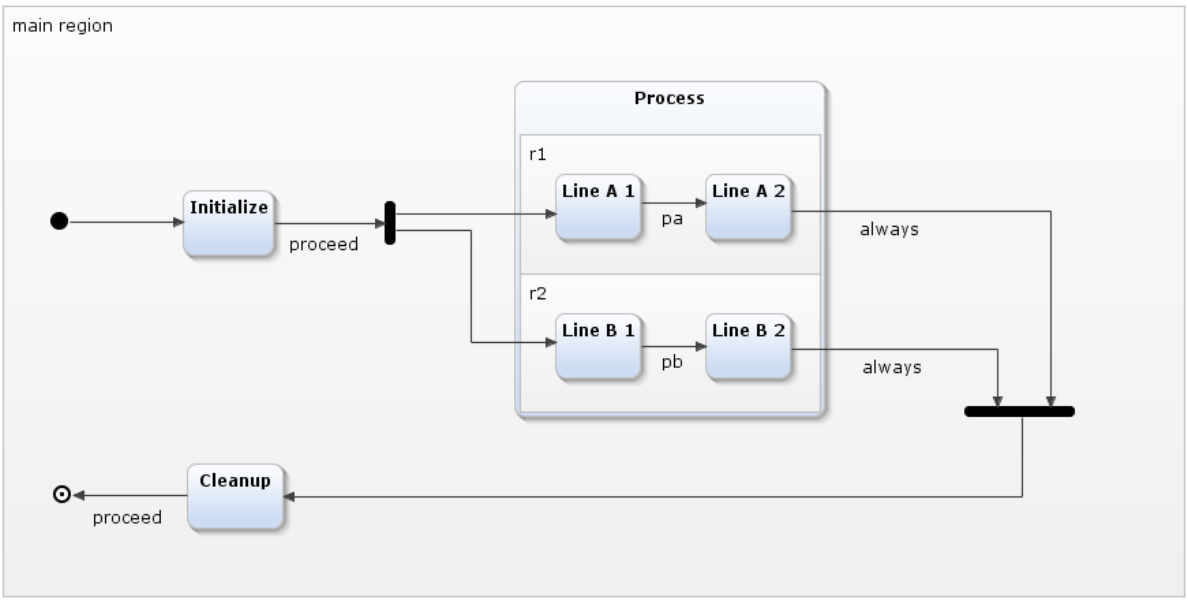


Figure 4.2: Synchronizations in YAKINDU.
Figure taken from https://www.itemis.com/en/yakindu/state-machine/documentation/user-guide/sclang_graphical_elements

orthogonal to each other. This type expresses a transition that can only be taken if several orthogonal parts of the statechart are in a specific configuration. The source states are exited, and a the target is entered in a regular fashion.

The second type can be understood as a transition with multiple target states, where the targets have to be orthogonal to each other. This type expresses a transition that sets up a specific configuration for several orthogonal parts of the statechart. With this type, the source state is exited in a regular fashion, and the target states are entered.

Figure 4.2 shows both types of synchronizations.

Synchronizations are not syntactic sugar and not supported in SCCD. SCCD does feature the `in_state` function (which is also supported in YAKINDU), where e.g. a transition's guard can check if another state (in any region) is part of the configuration. This can replace the first type of synchronization. There is no trivial replacement for the second type.

This feature also exists in Statemate [12] and Rhapsody [11], where it is called *join* (type 1) and *fork* (type 2).

Choice-pseudo states Syntactic sugar: a transition, possibly with a trigger, to a choice-pseudo state splits into several transitions, also with triggers, and actions. The same effect can be achieved by creating separate transitions with the combined triggers

and actions from the part before and after the choice pseudo-state. This construct is used a lot in practice, and helps decluttering the visual representation of the statechart.

Named entry- and exit-points In YAKINDU, the modeler can specify multiple named "initial" states, called named entry points. A transition to a composite state can specify the name of the entry to be taken. Likewise, a transition from a composite state can specify the name of the exit-point that should be taken. Entry- and exit-points are pseudo-states, as they are never part of the configuration. This feature gives composite states an *interface*, making them reusable into different statecharts or different parts of the same statechart. The same functionality can be achieved by setting the source or target of a transition directly to a composite states's inner state, but this breaks the composite state's encapsulation. Currently, YAKINDU does not support composable statecharts (this feature is planned for version 4.0). Reused states would probably be *inlined* in the compiled model. For this reason, this feature can be considered syntactic sugar.

Another statechart-like formalism that supports (or rather, forces) encapsulation of composite states through explicit entry- and exit-points is ROOMcharts, part of the ROOM modeling language [16].

External variables As an alternative to input/output events and synchronous function calls (called operations in YAKINDU), a statechart in YAKINDU can interact with its environment by reading or writing external variables, i.e. variables that are readable or writable externally. This feature can be mimicked by writing getter and/or setter operations, but direct access to variables may have a smaller overhead.

Every-trigger This is a timed trigger (like an after-trigger). An every-trigger becomes enabled at fixed intervals in time. This construct can be considered syntactic sugar, as it can be modeled with a self-transition with after-trigger as well.

4.1.3 Unit-testing framework

YAKINDU has a very powerful unit-testing framework. YAKINDU has its own textual language for specifying tests. It features white-box testing (checking if certain states are active), and allows the modeler to write multiple scenarios for the same model, each scenario raising events, checking states and waiting at will.

SCCD currently only has a black-box testing framework, where every test includes only a single scenario of timed input events and expected output events. This approach was sufficient for working with artificial statecharts, specifically modeled to show different behavior under different semantics. For serious statechart development however, a framework like YAKINDU's is highly desirable. SCCD tests are also written in an XML format, which is not very convenient (to read or to write).

Code generation

An important feature of YAKINDU that our version of SCCD currently does not have, is statechart compilation (code generation). YAKINDU has several code generators, currently supporting C, C++ and Java, and support for more languages is on its way.

According to the documentation, YAKINDU's generated code does not take care of scheduling, timers, input event queueing, thread-safety or memory management. The generated code primarily exposes a function to execute, in BSML terms, a "big-step". This is very similar to the implementation of the statechart language part of SCCD. Scheduling and input event queueing are implemented in the Controller; timers and thread-safety are implemented in different classes of the "realtime"-library. As opposed to YAKINDU's generated code, statechart execution does dynamically allocate memory, as the action language's stack frames are allocated on Python's heap. YAKINDU's action code simply uses the current thread's stack instead for temporary variables. We could do the same if we compiled our action code.

Generated code is well optimized. From studying the generated C code of a simple model, it was observed that YAKINDU stores its set of current states as an array of the configurations of orthogonal regions. The configuration of each region is stored as the state ID (as integer) of the deepest child that is a current state. This is a more compact representation than the bitmap representation used by the SCCD runtime. Transitions are always selected based on the current set of states in the configuration, by switch-case statements over the configuration of each region. The code generator can be configured to inline many types of functions.

YAKINDU's code generation approach may serve as inspiration for a future SCCD code generator. It would be an interesting endeavor to develop a code generator supporting the wide semantic variation of SCCD (or substantial subset thereof).

4.2 Example models

2 "example" statechart models are included with the SCCD project: **Digital watch** and **chat client**. They were adapted from solutions to homework assignments for the "Modeling of Software-Intensive Systems" course[4] at the University Of Antwerp. With each assignment, a set of requirements was given, and an interface to interact with a piece of software (also given) implementing/simulating the rest of the system.

Both examples are similar, in that they were both intentionally designed to encourage maximal usage of statechart language constructs, such as timed transitions and history states.

The examples were also implemented in YAKINDU, and the YAKINDU solutions would have the exact same statechart structure, apart from some usage of YAKINDU's

[4]http://msdl.cs.mcgill.ca/people/hv/teaching/MoSIS/

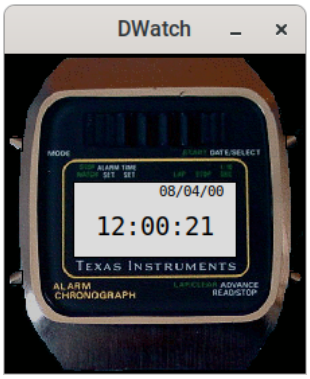


Figure 4.3: Screenshot of digital watch demo application

syntactic sugar, such as choice-states. In YAKINDU, the same semantics were always used: event-driven execution and parent-first priority. In SCCD, the default semantics (which are similar to YAKINDU's event-driven semantics) would yield a correct model, but for each example, the semantics were manually set to more "restrictive" options, which will be motivated.

4.2.1 Digital Watch

The digital watch example[5] is a controller for a (hardware) digital watch with 4 physical buttons, a time display, a chronometer display, an alarm display, an editing mode for setting the time and alarm, and backlighting.

Interface

The "hardware" and some other functionality is implemented as a GUI application in Python/TkInter (Figure 4.3). The statechart interacts with the Python code by calling synchronous operations on it, and the Python code may also send events to the statechart.

The Python code / "hardware" also has some **state**, in principle the following "variables":

- The value of the current time, as hours, minutes and seconds. The current value is this variable is drawn to the display only when requested by the statechart.
- The value of the current alarm, as hours and minutes. Similar to the current time.
- The value of the chrono, as minutes, seconds and 1/100 seconds. Similar to the current time.
- A flag, shown or not shown on the display, to indicate whether the alarm is active. The flag is only a display property and has no further effect.
- Whether the display background light is on or off.

[5] http://msdl.cs.mcgill.ca/people/hv/teaching/MoSIS/201314/assignments/Statecharts/

- Whether selection mode is active, on what "variable" it is active (time or alarm) and the group of digits (hours, minutes, seconds) currently selected. In selection mode, the selected group of digits blinks, and this is the only situation where the Python code *autonomously* updates the display.[6]

The Python code exposes a set of **operations** to the statechart. The statechart can call these synchronously from its action code, while making a transition:

refreshTimeDisplay() Redraw the time with the current internal time value. The display does not need to be cleaned before calling this function. For instance, if the alarm is currently displayed, it will be deleted before drawing the time.

refreshChronoDisplay() See refreshTimeDisplay()

refreshDateDisplay() See refreshTimeDisplay()

refreshAlarmDisplay() See refreshTimeDisplay()

increaseTimeByOne() Increase the time by one second. Note how minutes, hours, days, month and year will be modified appropriately, if needed (for example, when increaseTimeByOne() is called at time 11:59:59, the new time will be 12:00:00).

resetChrono() Resets the internal chrono to 00:00:00.

increaseChronoByOne() Increase the chrono by 1/100 second.

startSelection() Selects the leftmost digit group currently displayed on the screen.

increaseSelection() Increases the currently selected digit group's value by one.

selectNext() Select the next digit group, looping back to the leftmost digit group when the rightmost digit group is currently selected. If the time is currently displayed on the screen, select also the date digits. If the alarm is displayed on the screen, don't select the date digits. (to simplify the statechart).

stopSelection() Exits editing mode.

setIndiglo() Turn on the display background light.

unsetIndiglo() Turn off the display background light.

setAlarm() Toggles the alarm flag on the display on or off.

[6]Note that this could have also been implemented nicely in the statechart, by exposing operations to it for setting visibility of individual digit groups, making the display truly "passive", and moving all selection-state entirely to the statechart. A possible motivation for not doing so, could have been to make the assignment not too big.

checkTime() Checks if the alarm time set is exactly equal to the current clock time. If so, it will broadcast the "alarmStart" event to the statechart and return true. Otherwise, it returns false. Note that checkTime() does not care/check whether the alarm has been set "on".

Since the current time and the alarm time only change when explicitly requested (by the statechart) through the operations just explained, and statecharts have an *exact* notion of time through the use of simulated time (i.e. transitions do not take time to execute, and timed transition fire after precisely the amount of time specified), the statechart has exact control over the current time (and alarm time), as well as the timings of its calls to checkTime(), and will therefore, if properly designed, never "miss" an alarm.

The Python code may send the following **input events** to the statechart:

button press / release events Events are named 'topRightPressed', 'topRightReleased', 'topLeftPressed', 'topLeftReleased', 'bottomRightPressed', 'bottomRightReleased', 'bottomLeftPressed', 'bottomLeftReleased', and generated when their respective buttons are clicked by the mouse in the GUI.

alarmStart Generated when checkTime() operation is called (see later), and current time == alarm time.

Requirements

The requirements for the solution are as follows:

1. The time value should be updated every second, even when it is not displayed (as for example, when the chrono is running). However, time is not updated when it is being edited.

2. Pressing the top right button turns on the Indiglo light. The light stays on for as long as the button remains pressed. From the moment the button is released, the light stays on for 2 more seconds, after which it is turned off.

3. Pressing the top left button alternates between the chrono and the time display modes. The system starts in the time display mode. In this mode, the time (HH:MM:SS) and date (MM/DD/YY) are displayed.

4. When in chrono display mode, the elapsed time is displayed MM:SS:FF (with FF hundreths of a second). Initially, the chrono starts at 00:00:00. The bottom right button is used to start the chrono. The running chrono updates in 1/100 second increments. Subsequently pressing the bottom right button will pause/resume the chrono. Pressing the bottom left button resets the chrono to 00:00:00. The chrono will keep running (when in running mode) or keep its value (when in paused mode), even when the watch is in a different display mode (for example, when the time

is displayed). Note: interactive simulation (in AToM3) of a model containing time increments of 1/100 second is possible, but it is difficult to manually insert other events. Hence, while you are simulating your model, it is advisable to use larger increments (such as 1/4 second) for simulation purposes.

5. When in time display mode, the watch will go into time editing mode when the bottom right button is held pressed for at least 1.5 seconds.

6. When in time display mode, the alarm can be displayed and toggled between on or off by pressing the bottom left button. If the bottom left button is held for 1.5 seconds or more, the watch goes into alarm editing mode. This is not an example of good User Interface design, as going to editing mode will also toggle on/off and that may not be desired. It is however how the 1981 Texas Instruments LCD Alarm Chronograph works.

7. The alarm is activated when the alarm time is equal to the time in display mode. When it is activated, the screen will blink for 4 seconds, then the alarm turns off. Blinking means switching to/from highlighted background (Indiglo) twice per second. The alarm can be turned-off before the elapsed 4 seconds by a user interrupt (i.e.: if any button is pressed). After the alarm is turned off, activity continues exactly where it was left-off. Note that after the alarm finishes (flashing ends), the alarm is un-set (so it will not go off the next day at the same time).

8. When in (either time or alarm) editing mode, briefly pressing the bottom left button will increase the current selection. Note that it is only possible to increase the current selection, there is no way to decrease or reset the current selection. If the bottom left button is held down, the current selection is incremented automatically every 0.3 seconds. Editing mode should be exited if no editing event occurs for 5 seconds. Holding the bottom right button down for 2 seconds will also exit the editing mode. Pressing the bottom right button for less than 2 seconds will move to the next selection (for example, from editing hours to editing minutes).

Solution

The solution statechart consists of 5 orthogonal regions at the highest level, named Time, Alarm, Indiglo, Display and Chrono. These regions model (mostly) independent stateful parts of the behavior:

1. The “Time”-region that increases the time, every second, as long as we are not in “time editing mode”, implementing requirement (1). The entering and exiting of time editing mode is done in another region of the statechart, and signaled to this region with the use of “time_edit” and “edit_done” internal events. It is also this region that calls the “checkTime”-operation with each increase of the time, but only if we are in the alarm-on-state (in the “Alarm” region, see later). Checking if we are in a state from action code is done using the “in_state”-function.

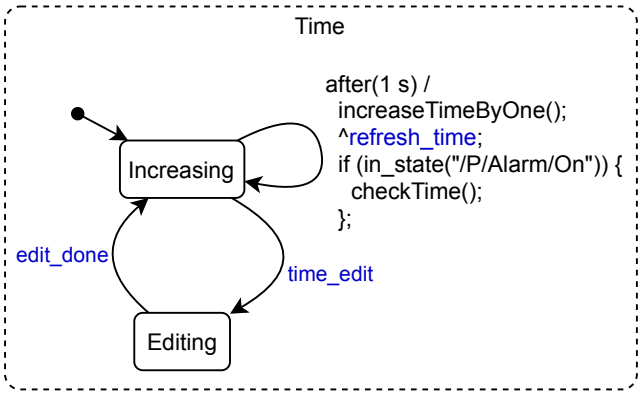


Figure 4.4: Time region of digital watch
Internal events are blue.

2. The "Alarm"-region keeps track of whether the alarm is on or off, and makes the alarm go off (blinking the light) if the "alarmStart" input event is received. We use enter- and exit-actions for the symmetric operations of (un)setting the watch's backlight. This way, we can be certain that e.g. the watch's backlight is not forgotten to be turned off, if the Blinking-state is exited. Similary, toggling the displays' alarm flag (operation "setAlarm()") is done by enter- and exit-actions of the On-state.

   We partially implement requirement (6), toggling the alarm with the bottom-left button while in time display mode. We also implement requirement (7), interrupting the alarm with any button when the alarm is blinking. This is done with 4 transitions (one for each button), but they are drawn as one transition for clarity.

3. The "Indiglo" region implements requirement (2).

4. The "Display" region is the largest region, implementing requirements (3), (5), (6) and (8). It controls what is visible on the display (the time, the chrono or the alarm time) and also implements the editing mode, which can be entered from the time or alarm view.

   Note that this region responds to the internal events "refresh_chrono" and "refresh_time" by refreshing the chrono and time view, resp. The region also raises the internal events "time_edit", "alarm_edit" and "edit_done" to signal the beginning and end of time / alarm edit mode.

5. The "Chrono" region implements requirement (4) and contains the logical states of the chrono: Paused and Running. These states are wrapped in a composite state in order to support resetting the chrono from either of these states.

   This region raises the internal event "refresh_chrono" when the chrono display should be updated.

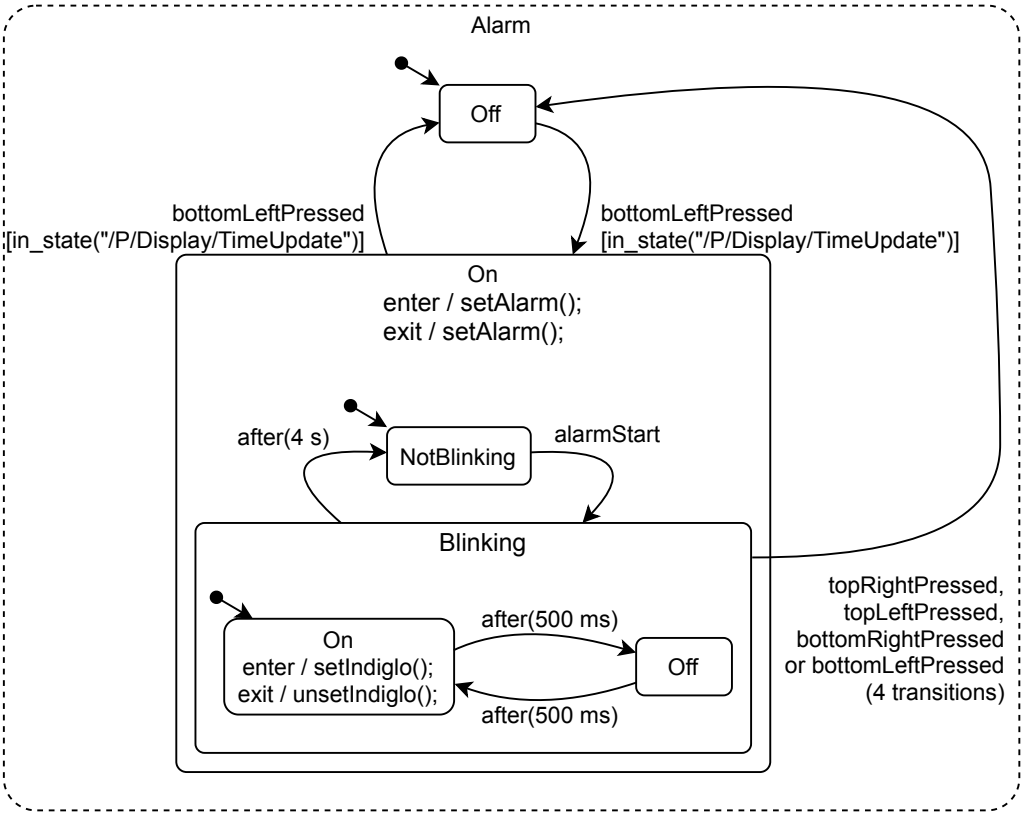


Figure 4.5: Alarm region of digital watch

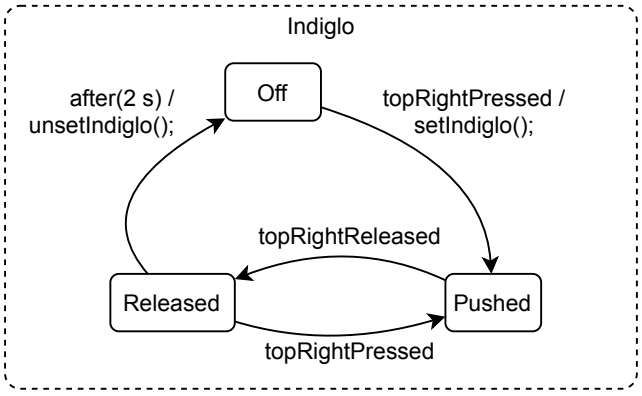


Figure 4.6: Indiglo region of digital watch

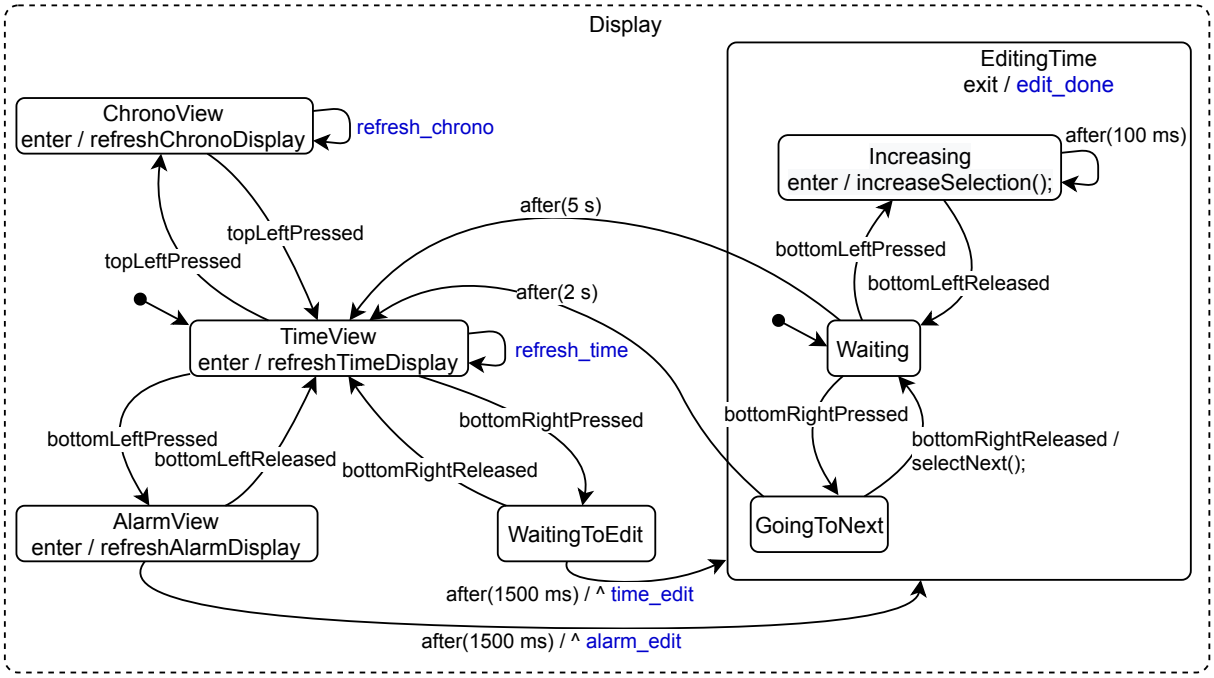


Figure 4.7: Display region of digital watch
Internal events are blue.

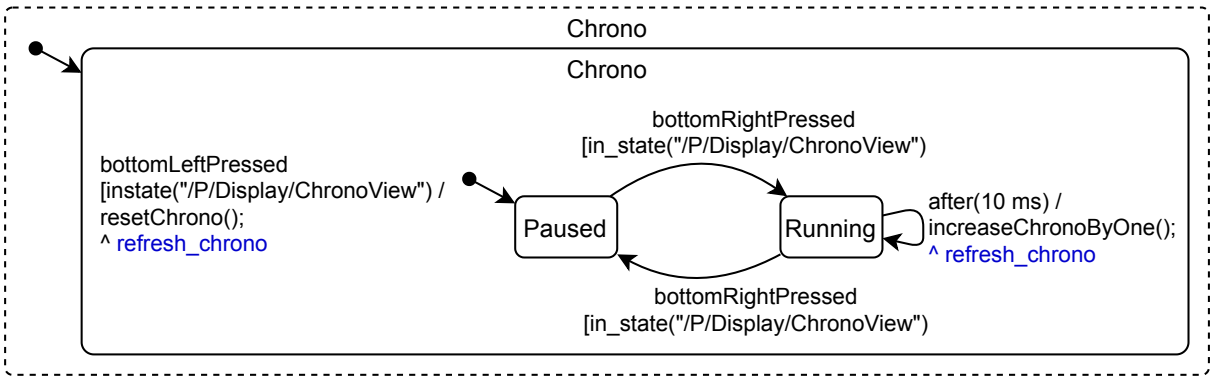


Figure 4.8: Chrono region of digital watch
Internal events are blue.

Valid semantic choices

The model has been observed to behave correctly under different semantics. Assuming we only handle 1 input event at a time, there are some observations about the intended behavior:

- A reaction to an input event raises at most 1 internal event.
- A reaction to an internal event raises no further internal events.
- A reaction to an input event fires at most 1 transition in the statechart (with one exception, to be explained).
- A reaction to an internal event fires at most 1 transition in the statechart, and in a different region than the reaction that raised the internal event. Every internal event "flows" from one region to another:
  - refresh_time: Time → Display
  - refresh_chrono: Chrono → Display
  - time_edit: Display → Time
  - edit_done: Display → Time

The last observation allows us to choose Take One for Big-Step Maximality. Take Many adds no benefit, and, under both YAKINDU (Section 4.1.1) and SCCD (Section 3.4.6, Figure 3.25) results in a more complex execution (featuring one more nested loop), making the behavior more difficult to predict for the modeler, and introducing the risk of never-ending big-steps.

If Take Many is chosen anyway, we must choose Present in First Small Step for Input Event Lifeline, to ensure that every input event only causes 1 transition to be made. Otherwise, never-ending big-steps would occur: Several regions contain transitions of the type shown in Figure 4.9. E.g. the Display-region would oscilate endlessly between ChronoView and TimeView when the topleft button is pressed. Similarly, we would also have to choose Present in Next Small Step for Internal Event Lifelife, to ensure every internal event is also only responded to once. Otherwise, e.g. the self-transition with trigger "refresh_time" of Display/TimeView would endlessly repeat.

Choosing Present in First Small Step slightly alters the behavior of the model: When the Alarm-region is in the Blinking-state, a button press may cause the alarm to be interrupted, or cause a transition elsewhere, in another region. (This is the only situation where an input event can be responded to in more than 1 region.) The transition first responding to the input event "consumes" it. In order to satisfy the requirements, we must thus give the Alarm-region a higher (Explicit) priority than the other regions. This can be done in both YAKINDU and SCCD. The button press, consumed by the Alarm-region, will no longer cause any other reaction. The requirements are ambiguous here, not mandating or forbidding this "alternative" behavior.

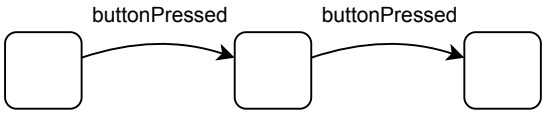


Figure 4.9: A statechart that would behave differently under TAKE ONE vs. TAKE MANY semantics

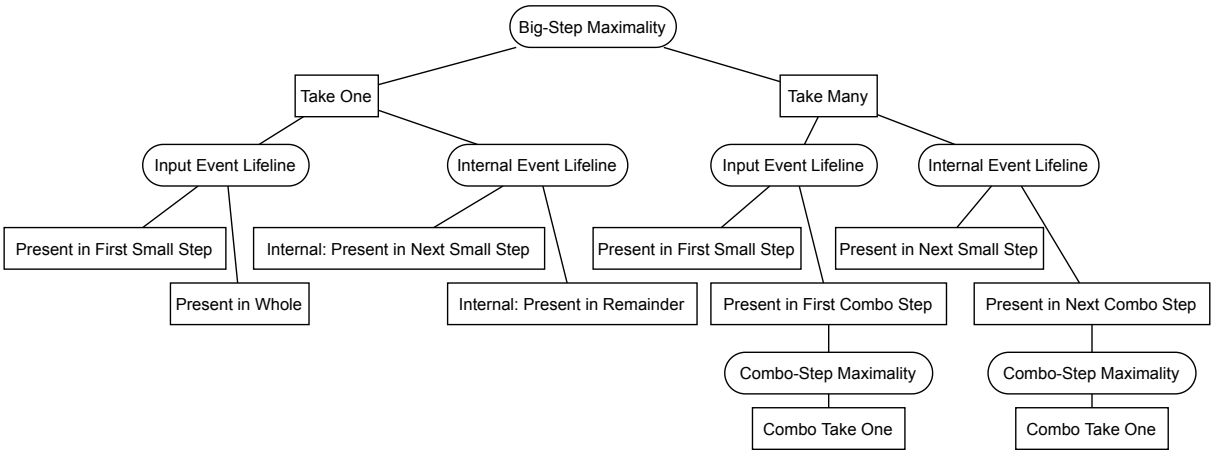


Figure 4.10: 8 correct semantic configurations for digital watch

Under TAKE ONE-semantics, all event input/internal lifeline options are valid (except PRESENT IN SAME). If Input Event Lifeline PRESENT IN WHOLE and Internal Event Lifeline PRESENT IN REMAINDER are chosen, input and internal events may be simultaneously enabled, but a region would never have to choose between an input and internal event, because in the model, internal events are always responded to in a (different) region that does not respond to the input event.

The model does not require combo steps. Combo steps are only useful if the reactions have to be *grouped*, and constraints need to be applied to each group. One case would be when input and internal event reactions are expected to have overlapping regions, but we still want to allow only 1 transition region per set of input/internal events (i.e. COMBO TAKE ONE, PRESENT IN FIRST/NEXT COMBO STEP). This is the default semantic configuration of SCCD, and also resembles the event-driven execution of YAKINDU, and is a valid semantic configuration for our model, but since it adds no value, while increasing execution complexity, there is no reason to use it.

Figure 4.10 shows a tree of valid combinations of the semantic options just discussed. TAKE ONE-semantics are definitely preferred for the reasons discussed above. Note that also YAKINDU's cycle-based and event-driven semantics are valid semantics, but that for cycle-based semantics, simultaneous input events could occur, which the model doesn't support (and also the requirements don't mention).

The explicit priority of regions has already been discussed, and only requires us to set a high priority to the Alarm-region if we choose PRESENT IN FIRST SMALL STEP for

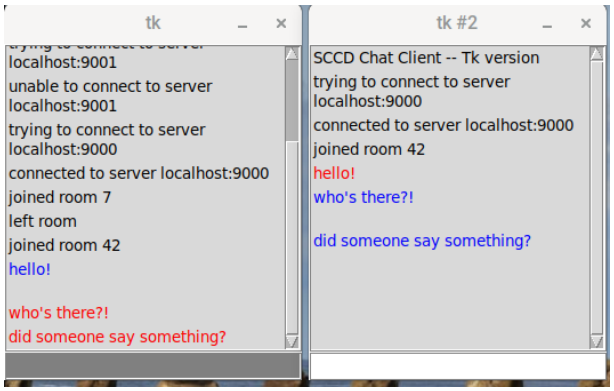


Figure 4.11: Screenshot of chat client demo application

input events.

Any explicit priority for every state's outgoing transitions is a valid one. All transitions have event triggers. For every state, there can be multiple outgoing transitions with different input events as triggers, but simultaneous input events are not expected to occur, and if they would occur (e.g. 2 buttons simultaneously pressed), the requirements do not specify what needs to happen, so any deterministic choice is a good one. Every state never has more than 1 internal event-triggered outgoing transition, which is given the lowest priority, since our chosen semantics always prioritizes input events.

For hierarchical priority, any choice is a correct one, since for every event, only a single transition per region can be selected.

The statechart has no variables, so Memory Protocol options do not apply.

Concrete implementation

The SCCD implementation differs from the YAKINDU implementation, because, at the time, SCCD did not support synchronous functions called directly from the statechart. A workaround was applied, using output events instead of synchronous functions. This workaround had no further impact on the statechart's structure or set of correct semantics. If output events are handled in the order that they were raised, there is no real difference between them or synchronous functions, as long as functions don't return data.

In SCCD, this example makes use of the event loop-integration library. All statechart simulation runs from TkInter's event loop. This way, the entire application runs from a single thread, making it easy to interact with TkInter (which is not thread-safe).

4.2.2 Chat Client

The chat client example[7] is a keyboard-commanded GUI chat application. The example consists of a client and a server program. The chat client is partially implemented as a statechart. The client-server protocol and the chat server are ad-hoc programmed in

[7] http://msdl.cs.mcgill.ca/people/hv/teaching/MoSIS/assignments/Statecharts

Python (Java for the YAKINDU version). The original version of SCCD includes a chat server implemented in statecharts as well, but it relies on multi-instance models, which our version does not yet support.

Interface

The statechart interfaces with 2 components: A network client, for interacting with a chat server, and a UI window, implementing the chat client's window. Both components are implemented in Python (Java for the YAKINDU version).

The network client exposes the following **operations**:

get_nr_of_servers(): int Returns the number of (hardcoded) servers in the client

get_server(index: int): str Returns the address of the server with given index

The network client can receive the following **output events** from the statechart:

connect(address: str) Request the client to connect to the server with given address.

disconnect Request the client to disconnect from the server.

join(room: int) Request the client to join the chatroom with given index.

leave Request the client to leave the current room.

ping Request the client to send a 'ping' request to the currently connected server.

send_message(msg: str) Request the client to send a message to the server. The server will broadcast the message to other clients in the same chatroom.

The network client may also send the following **input events** to the statechart:

connected Sent when the client is successfully connected to a server.

disconnected Sent when the client is successfully disconnected from a server.

joined Sent when the client successfully joins a room.

left Sent when the client successfully leaves a room.

receive_message: string Sent when a client receives a message from the server. (another participant in the room sent a message)

alive Sent when the client receives an "alive" message. (response to a "ping")

The UI window consists of a scrollable list of logs and received messages, and has an input box on the bottom. It exposes the following **operations**:

add_message(msg: str, type: str) Prints out a message to the scrollable list, with the specified type (as a string, either "info", "local", or "remote"))

append_to_buffer(char: str) Adds a string to the input box and visualizes the change

remove_last_in_buffer() Removes the final character from the input box and visualizes the change

clear_input() Clears the input box and visualizes the change

input_join() Colors the input box in the 'join' mode (green)

input_msg() Colors the input box in the 'message' mode (white)

input_command() Colors the input box in the 'command' mode (grey)

get_buffer(): str Returns the current content of the input box

The UI window can also send one type of **input event** to the statechart:

input(char: str) Sent when a keystroke occured. The parameter of the event contains the input character.

Requirements

1. The client is either disconnected or connected to a chatserver.
2. The client has a (hardcoded) list of servers that it can connect to. A server is identified by its IP address (or hostname) and a port.
3. If the client is disconnected, the user can not enter any input and must wait for a connection to be established.
4. The client will autonomously connect to a server by sending a connection request. If the server does not respond within a certain amount of time, another server is chosen until one of the connections succeeds.
5. As soon as the client is connected, the client will continuously ping the server with a certain interval. Should the server not respond after a certain timeout, the client will automatically go to disconnected mode and the user is again unable to enter any input.
6. After transferring to disconnected mode, the client will automatically try to establish a connection to another server. After a connection is successfully made, the user can again enter input.
7. All input that was made by the user before a disconnect should remain in memory and visible to the user.

8. When the user has connected to a server, the user starts without having joined any room. The user can give the input 'j' (Join) to join a specific room. After pressing 'j', the input box is colored green and only numerical input is accepted. Other input is not accepted.

9. After joining a room, the user starts to receive messages from the server that should be shown in the chat window. The user can either leave the room again by pressing 'l' (Leave), or can send a message by pressing 'm' (Message).

10. After pressing 'm', the entry box colors white and the user can type a message. The message will be sent after the return key is pressed.

11. Should a disconnect from the server occur while the user has already joined a room, the same room should automatically be joined after reconnection to the server.

12. Make sure that backspace also works in every input mode.

13. Input processing must happen in a way that resembles Finite State Automata.

14. You should test the above requirements by building a testing harnass that will determine whether your Statechart satisfies the requirements. Do this by using sctunit, the testing framework provided by Yakindu.

Solution

First of all, the solution statechart has 3 variables:

- `curr_server`: Integer, index of the server currently attempting to connect to, or connected to. Initially 0.
- `room_number`: Integer, number of the chatroom attempting to join, joined, or last joined. Initially 0.
- `reconnecting`: Boolean, whether a previous connection to a server was lost. Initially false.

The solution consists of 3 orthogonal regions at the highest level:

1. The "Receiving"-region has only a single sub-state with a self-transition, printing out any received message. No additional checks on server connectedness or chatroom joinedness are required, since we only expect to receive messages when connected and joined.

2. The "Pinging"-region starts pinging the server when the input event "connected" is received. When no response is received after 2 seconds, the region stops sending pings, and notifies other regions of this through the internal event "timeout".

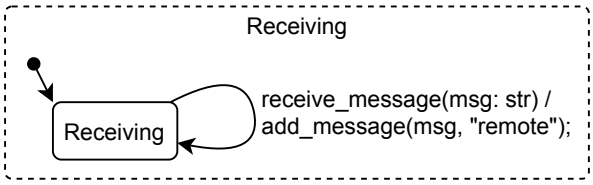


Figure 4.12: Receiving-region of chat client

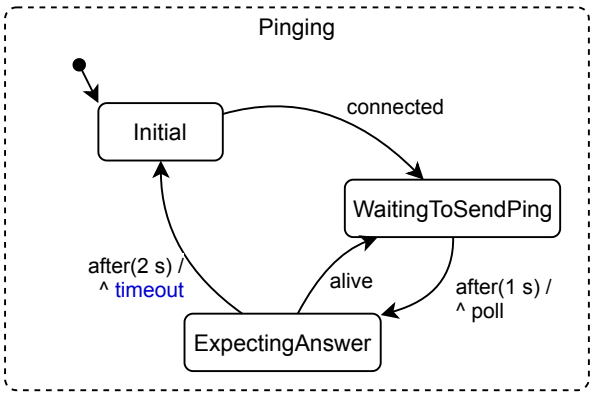


Figure 4.13: Pinging-region of chat client
Internal events are blue.

3. Finally, the "Main"-region models all user interactions and the main window's state. At the highest level, there are 2 sub-states:

   - Connecting: Initial state, attempts to connect in a round-robin fashion to one of the servers hardcoded in the network client. The first connection that succeeds takes us to the Connected-state.
   - Connected: Contains the client's behavior while connected to a server. When connection is lost (sensed through the "timeout"-internal event coming from the Pinging-region), we go back to Connecting, while setting the variable `reconnecting` to True, causing the Connecting-state to connect to the same server as before.

   The most complex feature, is automatically re-joining the chatroom after a restored connection. This is done by dividing the Connected-state into 2 (non-orthogonal) regions: LeavingOrLeft (should not re-join) and JoiningOrJoined (should re-join). At the same level, a history-state records which of those we were in.

   When going to the Joined-state, we must always send a join-request to the server first. Therefore we always pass through the Joining-state. When joined, another history-state is keeps track of whether we were entering a message, or not.

   When entering a regions that has history, we always enter the history state. If no history for the region exists yet, both YAKINDU and SCCD enter the region's

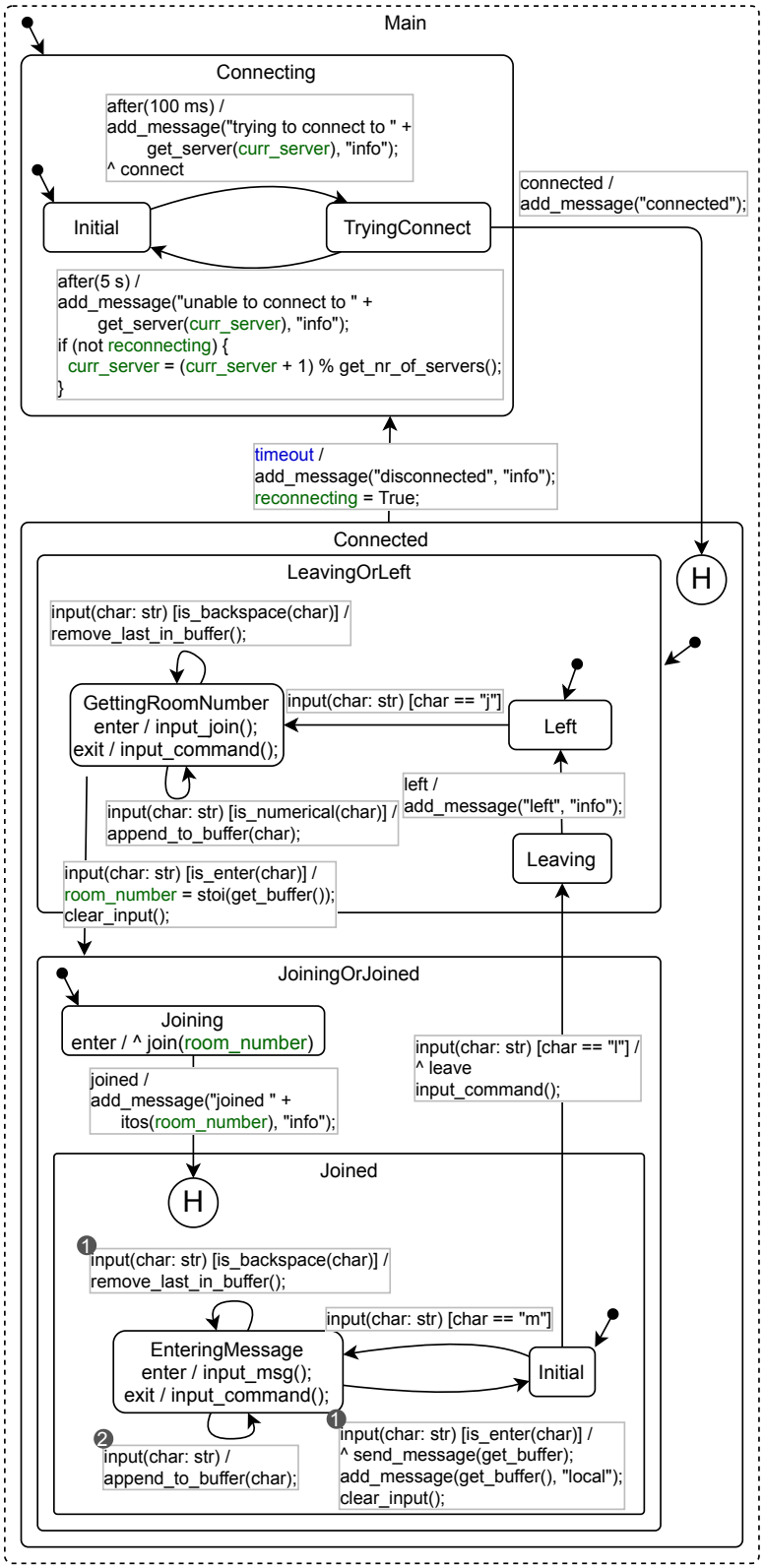


Figure 4.14: Main-region of chat client
Internal events are blue. Variables are green.

initial state, which is what we want.

Note that the outgoing transitions of "EnteringMessage" are given explicit priorities. The self-transition appending entered characters to the input field buffer is a lower priority than the rest, because it has no guard checking the key that was pressed, and may otherwise "steal" events from the other outgoing transitions.

Valid semantic choices

For the same reasons as explained in the digital watch example (Section 4.2.1), choosing Take One for Big-Step Maximality is appropriate. For Input Event Lifeline, we cannot choose Present in First Small Step, because the input event "connected" needs to be responded to in both the Pinging- and Main-region, so the only option remaining is Present in Whole. For Internal Event Lifeline, any option would be correct (except Present in Same).

Alternatively, SCCD's default semantics (Take Many, Combo Take One, Present in First/Next Combo Step) is also valid, but more complex to execute. No other working set of semantic choices can include Take Many.

The statechart includes variables, so a choice should be made for Memory Protocol semantics. Any choice is a valid one, because values written to a variable are not read until the next big step. An exception is the transition GettingRoomNumber → JoiningOrJoined, which writes a value to the variable `room_number`, and subsequently reads it (as the enter-action of Joining). However, SCCD considers this write-read sequence as part of a single transition, and therefore always reads the "fresh" value, ignoring memory protocol options. This feature was called "locally observable writes" in Section 3.4.8.

Concrete implementation

The YAKINDU solution makes extensive use of choice-pseudo states, which is syntactic sugar to group outgoing transitions with the same event trigger but with a different guard condition, such as for states "GettingRoomNumber" and "EnteringMessage". This makes the model more readable. SCCD currently does not have this feature, requiring the modeler to model them as separate transitions, which is also the way they were (manually) drawn in Figure 4.14.

In order to implement this model, SCCD had to be extended with the ability to call external functions directly from its action code, as certain interactions with e.g. the GUI (such as getting the text entered in the box at the bottom) had to be implemented as synchronous calls. Once this was done, the implementation was trivial, and identical to the YAKINDU-implementation.

In SCCD, like with the digital watch, the event loop integration library is used to run the statechart from the TkInter thread, but the network client library *has* to run separate threads for sending and receiving data, as these are blocking I/O operations on socket-objects in Python. In order to send or receive events from the statechart, thread-safe queues were used.

4.3 Meaningful semantics

Most semantic options of BSML can be arbitrarily combined, but not all combinations are useful. The paper on BSML [9] lists very few dependencies between options (such as COMBO TAKE MANY ⇒ SYNTACTIC ∨ TAKE MANY) but these dependencies are the bare minimum, technical restrictions for carrying out *some* form of execution. We will now see if we can further delimit combinations of semantic options for statecharts.

4.3.1 Maximality, Event Lifeline

The semantic aspects Big-/Combo-Step Maximality and Input/Internal Event Lifeline are considered "orthogonal" in BSML, which is correct from the viewpoint of "being able to execute them", but for realistic models, the options chosen for these aspects depend heavily on each other.

Together, these options control 2 things:

1. For how long transitions can be made as response to input or internal events.
2. At what point the effects of input event-triggered transitions begin to influence the remainder of the execution of the big-step, by means of internal events and variable assignments.

For the first property, an input/internal event lifeline spectrum exists from "short" to "long": For instance, PRESENT IN NEXT SMALL STEP is a "short" option, while PRESENT IN REMAINDER is a "long" option. For the second property, an internal event lifeline spectrum exists from "sooner" to "later": For instance, PRESENT IN NEXT SMALL STEP is a "soon" option, while QUEUE (only in SCCD) is a "late" option.

Choosing a "late" an event lifeline option helps separate reactions to input events from reactions to internal events. The "later" an event lifeline option, the more need for TAKE MANY. The "longer" the event lifeline option, the worse it behaves with TAKE MANY, because of never-ending big-steps: The options PRESENT IN REMAINDER and PRESENT IN WHOLE are unlikely to work well with TAKE MANY (e.g. the digital watch example, Section 4.2.1).

Introducing combo-steps (using COMBO TAKE ONE with a combo-step event lifeline option) helps to avoid never-ending big-steps with TAKE MANY. Combo-steps are very useful to build a semantic configuration that is "fool-proof", such as YAKINDU's event-driven execution, and SCCD's default configuration, which execute input events and internal events in separate rounds. However, many complex models (such as both examples discussed in the previous section) function perfectly fine under TAKE ONE semantics, even though they were developed with combo-steps in mind. The simplicity of TAKE ONE, both for the modeler and execution runtime (performance-wise) is a good argument for always chosing this option, if possible.

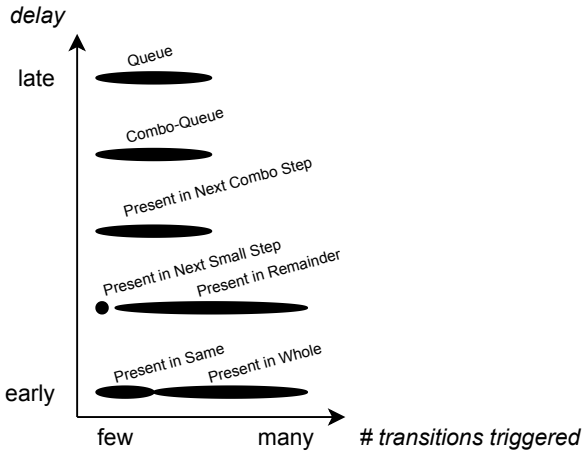


Figure 4.15: Properties of event lifeline options

4.3.2 Memory Protocol

Usually, the modeler would want to use the Memory Protocol option that matches the point in time where internally raised events become enabled. This way, the full effect of one transition, consisting of raised events and variable assignments, becomes visible at a single point in time.

The default semantics of SCCD follow this principle, with the Combo Step-option being the default, following the default event lifeline option Present in Next Combo Step. With these semantics, the statechart appears to behave as if the transitions within a combo-step fire concurrently.

When not using combo-step semantics or for any other reason, the modeler may also choose to disable Memory Protocol semantics by choosing Small Step (i.e. variable assignments become visible to the next transition). This is how YAKINDU behaves.

4.3.3 Priority

We consider priority truly orthogonal to other semantic options. In YAKINDU and SCCD, the priority of all transitions is a total ordering, explicitly created by the modeler. In models only responding to single-item sets of input events, usually there is no need for setting an explicit priority, because few transitions will be enabled at any point in time. Such models would still behave deterministically in the absense of priority rule(s). In the case of our example models (digital watch and chat client), we only sporadically had to set a priority. In our example models, there also were no hierarchical conflicts that needed resolution (e.g. “parent first”).

4.3.4 Conclusion

We agree with [9] that semantics should be chosen on a model-by-model basis. We advise to start out with a simple, restrictive set of semantics, e.g. Take One, as it is easy to

understand, efficient to execute, and has been shown to work even for large and complex models. The default semantics of SCCD and YAKINDU (event-driven) are less restrictive and fairly intuitive, but have a higher runtime overhead due to the use of combo-steps. Combo-steps can also be used to "simulate" concurrency, by enabling Memory Protocol semantics. Explicitly assigning priority to regions or transitions to resolve ambiguity, only has to be done sporadically even in large models.

CHAPTER 5

Related Work

Zhaoyi Luo has developed BSML-mbeddr [13], which is an implementation of much of the BSML language within mbeddr, a textual DSL workbench that integrates into the C language. The focus was not on statecharts, but on BSML as a language itself. Many of the same semantic aspects as in SCCD are supported, and additionally, concurrency (including Source/Destination Orthogonal and Non-Preemptive) is supported. Combo-step semantics are (currently) not implemented. BSML-mbeddr does not feature an action language, instead, a model's datamodel and action code are written in C. BSML-mbeddr does not feature history states or timed transitions, but these could be implemented "on top".

Adam Prout has developed a code-generator generator (CGG) [15] for a family of languages defined by *template semantics* [14], which preceded BSML. Template semantics is a parameterized operational semantics of hierarchical, non-orthogonal statecharts. Orthogonality is achieved with a set of compositional operators, working on components (individual non-orthogonal statecharts). The supported family of languages overlaps greatly with BSML, and is possibly larger, as even the semantics of Petri Nets can be described. BSML, however, works at a higher level of abstraction, offering a smaller set of semantic choices that are mostly orthogonal, making it easier to understand.

There are also efforts going on at precisely describing the semantics of statecharts. SCXML [19], on which SCCD's syntax is based, is a standardized (W3C, 2015) XML-based format for statechart descriptions, with precise semantics. Multiple implementations exist, such as SCION [7] (JavaScript) and Qt SCXML [5]. YAKINDU also provides an "SCXML Domain", allowing users to import/export models in the format, simulating and testing according to SCXML's semantics. Another standard is Precise Semantics of UML State Machines (PSSM) (OMG, 2019) [4], allowing for executable models.

CHAPTER 6

Conclusion and Future Work

We have studied the semantic variability of statecharts within the framework of Big-Step Modeling Languages, and have implemented a selection of options from said framework in the SCCD project. In the process, the SCCD project has been extended with an action language and a major design overhaul, making a clearer separation between statechart execution and the rest of the runtime. We have also studied the semantics of YAKINDU Statechart Tools, and compared our solution to it, in the domain of semantic variability, as well as overall feature-set, internal mechanics and usability. Example models were implemented in both solutions, to conclude that the default semantics of both SCCD and YAKINDU allow development of complex statechart models. Interestingly, the same models would also behave correctly under a range of semantic options. We suggest using restrictive options for maximality and event lifeline semantics, because those make it easy for the modeler to predict how the model behaves.

For future work, we foresee restoring Class Diagram functionality in our fork of the SCCD project. We also think a small extension to the semantics of SCCD could mimic YAKINDU's *event-driven* execution perfectly. We could also adopt other features of YAKINDU (and other statechart implementations), such as choice pseudo-states, entry- and exit-points and forks and joins.

Since the language of SCCD is platform-independent, the execution runtime should be ported to different platforms, as was the original intention. Lessons learned from the separation of statechart execution from the rest of the runtime (event queueing etc.), and insight into YAKINDU's optimized code generation, could fuel a new attempt at developing a multi-semantics, multi-target code generator.

Bibliography

[1] Lark parser library for Python. https://github.com/lark-parser/lark.

[2] libxml2 (C XML Library). http://xmlsoft.org/.

[3] lxml (XML Library for Python). https://lxml.de/.

[4] Precise Semantics of UML State Machines (PSSM). https://www.omg.org/spec/PSSM/1.0/PDF.

[5] Qt SCXML. https://doc.qt.io/qt-5/qtscxml-overview.html.

[6] YAKINDU Statechart Tools. https://www.itemis.com/en/yakindu/statechart-tools/.

[7] Jacob Beard. *Developing Rich, Web-Based User Interfaces with the Statecharts Interpretation and Optimization Engine*. PhD thesis, McGill University Libraries, 2013.

[8] Glenn De Jonghe and Hans Vangheluwe. Statecharts and Class Diagram XML-A general-purpose textual modelling formalism. Technical report, Technical report, University of Antwerp, 2014.

[9] 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.

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

[11] 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.

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

[13] Luo, Zhaoyi and Atlee, Joanne M. BSML-mbeddr: Integrating Semantically Congurable State-Machine Models in a C Programming Environment, 2016.

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

[15] Adam Prout, Joanne M Atlee, Nancy A Day, and Pourya Shaker. Code generation for a family of executable modelling notations. *Software & Systems Modeling*, 11(2):251–272, 2012.

[16] Bran Selic. Real-time object-oriented modeling. *IFAC Proceedings Volumes*, 29(5):1–6, 1996.

[17] Simon Van Mierlo, Yentl Van Tendeloo, Bart Meyers, Joeri Exelmans, and Hans Vangheluwe. SCCD: SCXML extended with class diagrams. In *Proceedings of the Workshop on Engineering Interactive Systems with SCXML*, volume 2, pages 1–2, 2016.

[18] W3C. W3C XML Schema Definition Language (XSD). `https://www.w3.org/TR/xmlschema11-1/`, 2012.

[19] W3C. State Chart XML (SCXML): State Machine Notation for Control Abstraction. `https://www.w3.org/TR/scxml/`, 2015.

Appendices

APPENDIX A

Action language grammar

The following is the grammar file used by the action language. It is meant for the Lark[1] parser library.

```
%import common.WS
%ignore WS
%import common.ESCAPED_STRING


// Expression parsing

// We use the same operators and operator precedence rules as Python

?expr: or_expr

?or_expr: and_expr
        | or_expr OR and_expr    -> binary_expr

?and_expr: not_expr
         | and_expr AND not_expr  -> binary_expr

?not_expr: comp_expr
         | NOT comp_expr           -> unary_expr

?comp_expr: add_expr
          | comp_expr compare_operator add_expr -> binary_expr

?add_expr: mult_expr
         | add_expr add_operator mult_expr -> binary_expr

?mult_expr: unary
```

```
          | mult_expr mult_operator unary -> binary_expr

?unary: exponent
      | MINUS exponent  -> unary_expr

?exponent: atom
         | atom EXP exponent -> binary_expr

?atom: IDENTIFIER               -> identifier
     | "(" expr ")"             -> group
     | literal
     | func_call
     | func_decl
     | array

IDENTIFIER: /[A-Za-z_][A-Za-z_0-9]*/

func_call: atom "(" param_list ")"
param_list: ( expr ("," expr)* )?  -> params

func_decl: "func" params_decl stmt
params_decl: ( "(" param_decl ("," param_decl)* ")" )?
?param_decl: IDENTIFIER ":" type_annot
type_annot: TYPE_INT | TYPE_STR | TYPE_DUR | TYPE_FLOAT
          | "func" param_types? return_type? -> func_type

param_types: "(" type_annot ( "," type_annot )* ")"
?return_type: "->" type_annot

TYPE_INT: "int"
TYPE_STR: "str"
TYPE_DUR: "dur"
TYPE_FLOAT: "float"

array: "[" (expr ("," expr)*)? "]"

?literal: ESCAPED_STRING -> string_literal
        | INT -> int_literal
        | FLOAT -> float_literal
        | bool_literal
        | duration_literal

?compare_operator: EQ | NEQ | GT | GEQ | LT | LEQ
?add_operator: PLUS | MINUS
?mult_operator: MULT | DIV | FLOORDIV | MOD

AND: "and"
OR: "or"
EQ: "=="
NEQ: "!="
```

```
GT: ">"
GEQ: ">="
LT: "<"
LEQ: "<="
PLUS: "+"
MINUS: "-"
MULT: "*"
DIV: "/"
FLOORDIV: "//"
MOD: "%"
EXP: "**"
NOT: "not"

bool_literal: TRUE | FALSE

TRUE: "True"
FALSE: "False"

INT: /[0-9]+/
FLOAT: /[0-9]+\.[0-9]*/

duration_literal: (INT duration_unit)+

?duration_unit: TIME_H | TIME_M | TIME_S | TIME_MS | TIME_US | TIME_NS |
    TIME_PS | TIME_FS | TIME_D

TIME_H: "h"
TIME_M: "m"
TIME_S: "s"
TIME_MS: "ms"
TIME_US: "us"
TIME_NS: "ns"
TIME_PS: "ps"
TIME_FS: "fs"

TIME_D: "d" // for zero-duration


// Statement parsing

?block: (stmt)*

?stmt: assignment ";"
     | expr ";" -> expression_stmt
     | "return" expr ";" -> return_stmt
     | "{" block "}" -> block
     | "if" "(" expr ")" stmt ("else" stmt)? -> if_stmt

assignment: lhs assign_operator expr
```

```
increment: lhs "+=" expr

?lhs: IDENTIFIER -> identifier

?assign_operator: ASSIGN | INCREMENT | DECREMENT | MULTIPLY | DIVIDE

ASSIGN: "="
INCREMENT: "+="
DECREMENT: "-="
MULTIPLY: "*="
DIVIDE: "/="
FLOORDIVIDE: "//="

COMMENT: "#" /(.)*/ "\n"
%ignore COMMENT
```