# Title

Joeri Exelmans[a]

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

**Abstract**

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

## 1. Introduction

(INTRO: TODO)

In Section 2, we will explain what *statecharts* are and how we can use them to model behavior. Because statecharts by themselves cannot serve as a language for building real-world, dynamic systems, we will use the *actor model* to reason about what is needed to do so, in Section 3. We find that we can use statecharts to describe the behavior of individual objects, and those objects will abide the rules of the actor model. The actor model assumes all relationships between objects (actors) are completely dynamic. We think that it is useful to specify some bounds on the relationships between objects. In Section 4, we briefly introduce *class diagrams*, a well-known formalism that we will use to specify the structure of our objects, and the relationships between them.

## 2. Statecharts

Statecharts are a visual formalism for expressing behavior. They were first introduced by David Harel in 1987 [3]. Statecharts are interesting because they allow complex behavior to be expressed with a relatively small number of language artifacts.

Before going into detail on the syntax and semantics, we will first compare statecharts with a formalism the reader is likely familiar with: State machines.

### *2.1. State Machines*

Statecharts are somewhat similar to (deterministic) state machines from language & automata theory. A state machine contains a finite number of nonhierarchical states. When running, we always start in the same *initial state* (or *default state*). Upon receiving input characters, the state machine makes transitions labeled by those characters, changing the current state. Eventually, when all characters are consumed, we check if the current state is in the set of

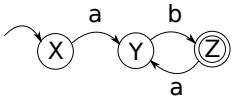


Figure 1: An example state machine. $X$ is the default state, $\{Z\}$ is the set of accepting states. The language represented is $\{ab, abab, ababab, ...\}$, or, in regex form: $(ab)+$

*accepting states*. If and only if this is the case, the string of consumed characters is part of the language represented by the state machine.

Statecharts differ from state machines in the sense that there are no accepting states. A statechart is more like a computer program, running as long as necessary, and producing side effects while doing so. Statecharts respond to incoming events by making transitions, just like state machines do, but in contrast to state machines, statechart events aren't necessarily known from the start: A statechart simply waits for incoming events, in realtime. In statecharts, a transition can also cause side effects, like generating new events (that can be sensed by e.g. another statechart).

Statecharts can be hierarchical, i.e. states can have children states, and statecharts can have orthogonal components, i.e. multiple current states at the same level of hierarchy. We will not go into detail on these features yet.

To make statecharts a complete alternative to a programming language, statecharts can also perform computation: a statechart can keep "local" variables, and events can carry parameter values (this way, a received event is like an asynchronous procedure call). As a side-effect of making a transition, simple subroutines can be executed, operating on variables.

In the next section, we will introduce the *composed hierarchical transition system* (CHTS) normal form syntax [5] [2]. It differs from Harel's original paper only in terminology. A motivation for doing so will follow later, in Section **??**.

### *2.2. Syntax and Semantics*

We will now explain the CHTS normal form syntax. We will refer to CHTS models simply as statecharts from now on.

Formally, a statechart consists of

1. A composition tree of *control states*
2. A set of *transitions* between the control states
3. (implicitly) A set of *events*.

Until we introduce hierarchy, the composition tree will always be a single root node with all other control states as its direct children.

#### *2.2.1. Events*

An event is a named artifact of communication; the name of an event serves merely as an identifier. During execution, a statechart communicates with the outside world (e.g. the execution *environment* and other statecharts) only through events. Events can also be used for communication between different parts of the same statechart (see Orthogonality, Section 2.2.7).

*2.2.2. Control States*

There are 3 different types of control states. For now, we will only be concerned with 1 type, called *basic states*.

During execution, a statechart instance will have a set of *current states*. Until we introduce hierarchy and orthogonality, the number of current states will always be equal to 1.

At the top level of hierarchy (and for now the only level of hierarchy), every statechart has a single *default state*. The default state is a state that becomes the current state when an instance of the statechart is first initialized. In this paper, the default state will be indicated syntactically with an incoming arrow, coming from nowhere[1], similar to state machine notation in language & automata theory.

*2.2.3. Transitions*

A *transition* is a tuple consisting of

1. A source control state
2. A destination control state
3. (optionally) An event trigger
4. (optionally) A guard condition
5. (optionally) A set of generated events
6. (optionally) A set of actions

We will treat guard conditions and actions later.

During execution, a transition can be *enabled* or *disabled*. A transition is enabled if all of the following are true:

1. Its source control state is in the statechart's set of current states
2. If an event trigger is specified, the event has to be *present* at the moment
3. If a guard condition is specified, it has to evaluate to true.

If a transition is enabled, it can trigger (or "fire"). The triggering of a transition causes the following things to happen:

1. The source state unbecomes a current state, the destination state becomes a current state
2. If a set of generated events is specified, they become present
3. If a set of actions is specified, they are executed.

The execution history of a statechart can be captured as an ordered sequence of triggered transitions.

Note that, although both the event trigger and guard condition are optional, when none are specified, a transition can always fire whenever its source state

[1] Tools often use a different notation, for instance displaying the default state in a separate color

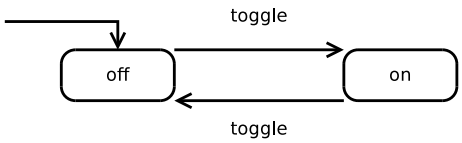


Figure 2: A simple statechart.

is a current state. This is not very interesting behavior, so a transition will typically have at least one of them specified.

We will discuss guard conditions and actions later. We will first present 2 examples making use of event triggers, and later, generated events.

**Example 2.1.** *Let us look at Figure 2. It models a simple on/off switch that can only be toggled. There are 2 basic states: "*off*" and "*on*". The arrow from "*on*" to "*off*" with label "*toggle*" is a transition. Semantically, it means that whenever the state "*off*" is the current state and we receive the event "*toggle*", "*on*" becomes the new current state. The transition in the reverse direction means that if we receive the event "*toggle*" one more time, "*off*" becomes the current state once again. Finally, state "*off*" has an incoming arrow coming from nowhere. This means that "*off*" is the default state in this diagram.*

*2.2.4. A word on the environment...*

In Example 2.1, the statechart responds to an event called "*toggle*". This event is coming from the *environment*. The environment is an abstract entity surrounding the statechart. Many things can act as an environment: a user could manually trigger the "*toggle*" event in an interactive simulator, or events could be coming from another statechart (see later). Consider events coming from the environment as being *input* for the model.

*2.2.5. Hierarchy*

Let us now introduce the concept of *hierarchy*. Recall that there are 3 types of control states. We have already seen basic states. Basic states cannot have children. The 2 other types of control states, *and-states* and *or-states*, always have 1 or more children. Therefore, they never occur as leaf nodes in the composition tree. And-states will be covered later; we now focus on or-states.

An or-state is defined such that if the or-state is in the set of current states, exactly 1 of its children is also in the set of current states. If the or-state is not in the set of current states, none of its children is a current state. Exactly one of the children of an or-state must be indicated to be the default state. When the or-state becomes a current state, then the default state becomes the current state as well. The root of a statechart composition tree is always (implicitly) an or-state. Remember we mentioned a statechart to always have a default state

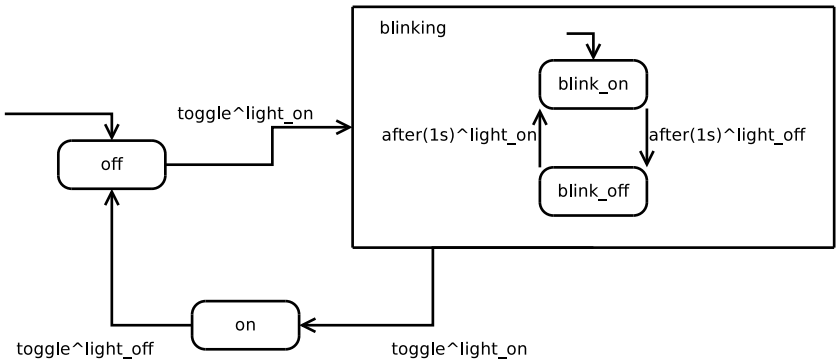


Figure 3: A slightly more complex statechart.

at the top level in the hierarchy: This default state is actually a property of the implicit or-state root node.

An or-state can have any kind of control state as a child, so there can be any (finite) number of hierarchy levels.

*2.2.6. Transitions, revisited*

The full syntax for the label of a transition is as follows:

$$trigger[condition]\hat{}generated\_events/actions$$

Unspecified parts are simply left out. So far, we have only seen transitions that specify an event trigger. In the next example, we will introduce generated events, denoted by the ˆ symbol.

When a triggered transition generates events, they can be sensed by the environment (in this case, we talk about *environmental output events*), or by another (part of the) statechart.

**Example 2.2.** *In Figure 3, we see a statechart similar to the one in Example 2.1, but this time, we added an additional "*blinking*" state in the cycle between "off" and "on", so it resembles a simple controller for e.g. a detachable bicycle LED light. Initially, the light is off. When the switch is toggled, it starts blinking. When the switch is toggled again, it keeps burning. When toggled one more time, the light is off again. The states "*off*" and "*on*" are basic states just like in Example 2.1. The large rectangle with the word "*blinking*" in it is an or-state. It has 2 children, "*blink_on*" and "*blink_off*", of which "*blink_on*" is the default state.*

*In this example, we see another new feature:* timed transitions*. A timed transition is a transition that doesn't require an event to be present, but can be fired after the source state is a current state for a certain amount of time. In this case, it causes "*blink_on*" and "*blink_off*" states to alternate being the current state every 1 second whenever the state "*blinking*" is a current state.*

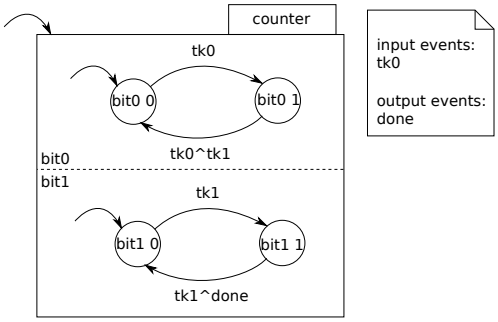


Figure 4: A 2-bit counter statechart with 2 orthogonal components. Based on [2], p.14

*The statechart generates 2 events: "*light_on*" and "*light_off*". We suppose these events are picked up by the environment, which in turn feeds it to e.g. a physical light switch.*

Timed transitions are not (explicitly) part of the CHTS normal form syntax/semantics. There are a number of reasons why we include them anyway:

1. They are a part of Harel statecharts
2. They are very useful in many situations, e.g. in the example just discussed
3. They don't interfere with the rest of the syntax/semantics. We can implement them by generating a *timed event request* whenever we enter the source state of a timed transition, for the environment, which in its turn generates a special event after the specified amount of time has passed, i.e. timed events can be regarded as regular event triggers.

Hierarchical statecharts can be flattened. This is done by changing the destination control state of incoming transitions of all or-states to the default states of those or-states. For each transition whose destination control state is some state $X$ and whose source control state is an or-state, we create new transitions whose source control states are the direct children of the or-state, and whose destinations are $X$.

*2.2.7. Orthogonality*

Orthogonality allows for very compact descriptions of relatively complex behavior.

Orthogonality relates to the 3rd and final type of control states: *and-states*. Just like or-states, and-states must have children and can therefore not occur as leaf nodes in the control state composition tree. Just like basic states and or-states, and-states can serve as the source and destination of transitions. An and-state is characterized such that if the and-state is in the statechart's set of current states, *all* of it's children are current states as well. Children of an and-state are also called *orthogonal components*. Orthogonal components are typically or-states.

We will denote orthogonal components syntactically as areas separated by a dashed line. And-states can then be recognized because of the orthogonal components in them.

**Example 2.3.** *Figure 4 shows a statechart with an and-state as the only child of the root node. There are 2 orthogonal components (*"bit0" *and* "bit1"*) and each of them is an or-state with 2 basic states (*"bit0 0" *and* "bit0 1", "bit1 0" *and* "bit1 1" *as their children. Each component represents a 1-bit counter. Together, they make a 2-bit counter by having the less-significant bit0 generate carry-out events upon overflow.*

*After initialization, both orthogonal components enter their default state. Upon receiving the input event* "tk0"*, bit0 counts to one. There are no transitions departing from* "bit1 0" *with* "tk0" *as event trigger, so bit1 remains in its default state. When receving* "tk0" *again, bit0 overflows back to zero and generates the* "tk1" *event, which is sensed by bit1, making it count to 1. When* "tk0" *is received for the third time, bit0 counts to 1 and bit1 stays in 1. Finally, when* "tk0" *is received for the fourth time, both bits go back to 0 and bit1 generates the* "done" *event.*

We mentioned a few times that events can be used for communication between different parts of the same statechart. In the previous example, *"tk1"* is an event solely used for that purpose.

Transitions in orthogonal components can sometimes be made simultaneously, but this still isn't a good opportunity for parallelism, since components usually depend on each other. Transitions in one componenent often rely on an event generated by a transition in another component, or they require another component to be in a certain state.

Orthogonal components can also be flattened. To do so, we generate *product* states for each combination of states, with one state per component in each combination. Transitions are duplicated between pairs of combination-states. The precise way in which this happens is left as an exercise for the interested reader.

*2.2.8. Variables*

Statecharts can have any finite number of variables, of any type supported by the target language. Upon compilation, variables in a statechart are translated into variables in the target language.

We need variables, because otherwise statecharts would not be able to perform computation. Although we can represent variable values as states, as in the past example (2.3), let it be clear that this a *rather inefficient* technique.

**Example 2.4.** *In Figure 5, we have a statechart with 3 orthogonal components. There are 2 integer variables:* "seconds" *and* "minutes"*. The uppermost component emits a* "tick_s" *event every 1 second. One component below, every* "tick_s" *causes a* "seconds" *counter to increment. Every 60 seconds, the counter is reset*

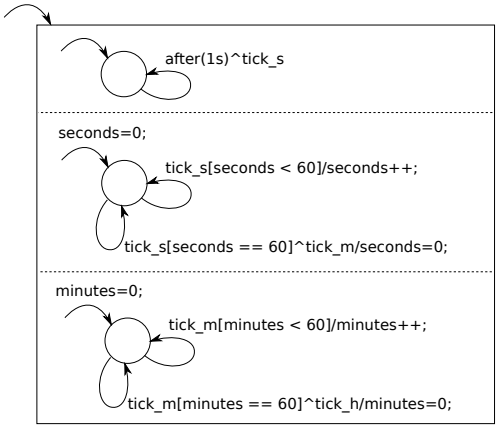


Figure 5: A clock.

*and a* "tick_m" *event is generated, which in turn causes the lowermost component to increment its* "minutes" *counter. Every 60 minutes, a* "tick_h" *event is generated.*

*The lower 2 orthogonal components use guard conditions testing the values of* "seconds" *and* "minutes" *in order to make a selection between 2 possible transitions on the reception of a* "tick_s" *and* "tick_m" *event, respectively. Both integer variables are incremented and reset by actions carried out by their respective transitions.*

*2.2.9. Enter and Exit Actions*

Enter and exit actions are a feature found in many statechart-like languages. They are properties of control states. Enter actions are executed whenever a control state becomes a current state, exit actions are executed whenever a control state unbecomes a current state.

Enter and exit actions are not part of the CHTS normal form syntax because it is possible to transform a system with this feature into a system without it.

Enter and exit actions are very useful when a control state has a lot of incoming/outgoing transitions, and all of them carry out the same (sub)set of actions. Because of this, we will include enter and exit actions in our formalism.

*2.3. Conclusion*

We have now seen the basic syntax and semantics of statecharts. Even though statecharts are very flexible and expressive, we still cannot build an entire system only as a single statechart. A real-life system typically consists of dynamic entities, created and destroyed during runtime. There is also no concept of locality/encapsulation: A statechart can have variables, but these variables are visible to all transitions of that statechart.

The solution we will present for this problem will use statecharts for describing the behavior of individual *objects*. Instances of those objects can then be

created and destroyed during execution, and they can send events to each other, pretty much like objects invoke each other's methods in object-oriented systems. The precise way of how we specify objects and the relationships between them is explained in Section 4.

In the next section, we focus on the issue of parallelization. It has already been mentioned at the end of Section 2.2.7 that (orthogonal) components of a single statechart typically do not allow for parallel execution. But if we use statecharts to express the behavior of individual objects, we can still run those objects in parallel. The *Actor Model*, an abstract formalism to achieve parallelism, closely resembles this paradigm.

## 3. The Actor Model

CPU speeds have not been growing exponentially for a while now, and parallelism is expected to be the only way to achieve higher returns on higher investments, now and in the future. First proposed as a way to create artificial intelligence in 1973 [4], and later generalized as a multi-purpose programming paradigm in 1985 [1], the *Actor Model* is an abstract model of computation (like the Turing machine), with the goal of making parallelism easier. A system modeled in actor model artifacts can be arbitrarily ran as a single threaded process on a single machine, as a multithreaded process on a single machine, or as a distributed system spanning the globe. In fact, the actor model is so generic that it allows implementations to dynamically scale the number computational threads of a running system.

Before we go into the details of the actor model, let us first examine the weaknesses of the way parallelism is usually implemented: raw threads.

### *3.1. Parallelism Using Raw Threads*

Massive parallelism is trivial to implement (and already used today) for uniform tasks that operate on large amounts of data, such as computer graphics (transforming lots of vectors by the same matrix), and the mining of large databases for the same types of patterns.

It is much harder to apply parallelism to more conventional problems. Existing programming languages offer API's for working with threads. Threads have a number of disadvantages:

1. The developer has to choose which parts of the program to run in threads.
2. The number of threads is therefore fixed, and computation throughput does not scale as more resources are added.
3. Communication between threads usually happens through shared memory, requiring synchronization, with a performance penalty and deadlock as common side-effects. Anti-deadlock measures dramatically increase system complexity and further negatively impact performance.

### *3.2. Parallelism in The Actor Model*

In the actor model, the basic building blocks are called *actors*. An actor is a unit of computation. Actors communicate by sending *messages* to each other's addresses. An actor may have any finite number of addresses.

The actor model does not imply any kind of parallelism, but it does make a number of basic assumptions to make it trivial to run systems designed in it in parallel:

1. There is no global state
2. All communication (between actors) is asynchronous.

Actors only "act" when they receive a message. Based on its current state, and the address on which a message was received, an actor may do any of the following things:

1. Create new actors
2. Send *asynchronous* messages to addresses of other actors
3. Decide how to respond to a next message[2]. (i.e. change its state)

An actor can only process one message at a time. However, the actor model assumes that on the reception of a message, computation is infinitely fast: All side-effects are carried out immediately. This is mostly a matter of making the actor model less complex. Queues are therefore not assumed to be present, but in reality (where computation *does* take time) they will typically be used behind the scenes by implementations. When multiple messages are received at exactly the same time (or so close to each other that the actor cannot distinguish which one was there first), the order in which they are responded to is picked randomly.

An actor can only send messages to addresses it somehow knows. Messages may carry addresses as parameters, and actors may store addresses received this way for later usage.

### *3.3. Cases of The Actor Model*

The actor model is not only Turing-complete, the Turing machine is a special case of the actor model. We can thus also view existing programming paradigms as special cases of the actor model. Synchronous function/method calls can be expressed using asynchronous messages by letting the caller wait for a return message. The asynchronous message sent by the caller will then carry one of its own addresses upon which it will receive the *return* message.

In the context of functional programming, a function is an actor that never changes. Because actors cannot natively wait for some value to be evaluated before returning a result, the actor will create a new actor that waits for the result of a sub-expression. Let's look at an example.

[2]This includes the actor destroying itself (by choosing not to respond to next messages)

```
actor factorial():
  when receive message 'compute' with parameters (return_addr, x):
    if x <= 1:
      send result(1) to return_addr;
    else:
      c = new actor wait_for_result(return_addr, x);
      send compute(c, x-1) to self;

actor wait_for_result(return_addr, x):
  when receive message 'result' with parameters (y):
    send result(x*y) to return_addr;
```

Figure 6: Factorial function in the actor model.

```
fact(0) ->
  1.
fact(N) ->
  N  * fact(N-1)
```

Figure 7: *Factorial* in Erlang, the most widespread actor-based language.

**Example 3.1.** *Suppose we have an actor representing the* factorial *function. The actor has a single address upon which it receives a request for calculating the factorial of* x *and sending the result to* return_addr*. If* $x <= 1$*, 1 is returned immediately. When* $x > 1$*, it requests itself to calculate the factorial of* $x - 1$*, and to send the result to a new actor* wait_for_result *specifically created. The* wait_for_result *actor gets 2 parameters upon creation: the* return_addr *originally received by* factorial*, and the value* x *with which to multiply the factorial of* $x-1$ *when it is received. Pseudocode in Figure 6*

*This example demonstrates a lot of common constructs in the actor model.* factorial *receives an address in a message,* wait_for_result *stores this address in its local storage. Both* factorial *and* wait_for_result *perform computation.* factorial *creates a new actor.*

Because it takes a lot of work programming actors this way, actor-based programming languages (e.g. Erlang) often make recursion possible by similar means as in functional programming languages. See Figure 7 as an example.

In [1], another example is presented that shows how much the actor model resembles the functional paradigm. In functional languages, data structures such as linked lists are represented by the recursive operations that construct the data structure, instead of linked pointers to blocks of memory. What follows is an example showing a stack structure of linked nodes.

**Example 3.2.** *In Figure 8 we see pseudocode for a* stack_node *actor. It can*

```
actor stack_node(content, previous_node):
  when receive message 'push' with parameters (new_content):
    next_node = new actor stack_node(content, link);
    become new actor stack_node(new_content, next_node);
  when receive message 'pop' with parameters (return_addr):
    if content != NIL:
      become previous_node;
      send result(content) to return_addr;
```

Figure 8: A recursive stack data structure.

*respond to 2 types of messages:* pop *and* push*. Every* stack_node *has a link to the previous node and content stored in it.* push *causes the* stack_node *actor to create a new* stack_node *with itself as the previous node. Then something interesting happens: the current* stack_node *(who received the* push *message), becomes the actor it just created. New messages sent to the same actor address will thus be received by the newly created actor, which is what we want: Another* push *or* pop *should always be received by the top node. The old actor becomes reachable only through the new actor, which has the old actor's address stored in the* previous_node *field. When a* pop *is received, the top node actor returns his content and becomes the actor pointed to by the* previous_node *address.*

### *3.4. Deadlock*

It is often claimed that deadlock cannot occur in the actor model. This is not the case: the "*when receive*" statement is blocking in nature. It is possible to create 2 actors that infinitely wait for each other to receive a message.

**Example 3.3.** *Figure 9 shows a deadlock situation in the actor model.*

Again, since it is possible to see other paradigms as special cases of the actor model, it is possible to create an actor that resembles a shared resource, and other actors that resemble threads, such that the shared resource actor only allows one of the thread actors to obtain access to it at the same time. Let this serve as a more realistic example of how the actor model can be perverted with all the trouble known to arise from the use of raw threads + shared state.

However, the actor model includes the feature of *futures*, that makes the use of locks unnecessary in many cases.

### *3.5. Futures*

A future is a special kind of actor that resembles the outcome/result of an asynchronous call known to evaluate to some result at a later point in time. Futures can be passed around just like regular values. Operations can be performed on a future, by pushing them into the future pipeline. A future keeps a

```
actor plus1(plus2):
  when receive message 'hello' with parameters (x):
    send hello(x+1, this) to plus2;
    become NIL;

actor plus2():
  when receive message 'hello' with parameters (x, from):
    send hello(x+2) to from;
    become NIL;

actor system():
  init:
    new actor plus1(new actor plus2());
```

Figure 9: 2 actors waiting infinitely for each other to say "hello".

pipeline of operations to be performed whenever the last operation has evaluated to some result. This way, an actor (thread) will never have to block until the result of some call is evaluated.

*3.6. Our Approach*

Because an actor is any kind of unit that responds to messages received from the outside, we can use statecharts to express the behavior of actors. To make our statecharts actor-compatible, when responding to a received message/event by making a transition, a statechart will be able to do any of the following:

1. Create new statechart instances (or: objects whose behavior is modeled by statecharts)
2. Generate events and send them to *specific* statecharts (actors/objects) asynchronously
3. Change its state (trivial)

For the first point, we will have to come up with a way for statecharts to create new statecharts as the side-effect of making a transition. From the statecharts point-of-view, this can only be done by generating some request event for the environment to create a new statechart instance, and letting the environment return an event parameterized with the "address" of the new statechart. As will be discussed in Section 4, the part of the environment that handles the creation (and deletion) of statechart instances will be referred to as the *object manager*. From the actor model point-of-view, the creation of new actors is native and thus the object manager is implicit from this perspective.

Second point: Once a statechart can create an instance, it will probably want to communicate with it. Typically, in OO systems, there is some specification with regard to the relationships between objects. For instance, in Java and

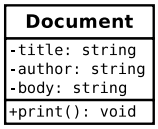


Figure 10: A class called *Document* with 3 private member fields of type string, and 1 public method *print()* of type void.

C++, a class definition contains a declaration list with references/pointers to other classes/objects. The relationships between objects are therefore known in advance, and often bounded. This is a useful property: the more we know about the topology of a system, the easier it is to debug, and compilers can use this information to make optimizations[3]. (e.g. using static allocation if possible) We will use *Class Diagrams* to specify the relationships between statechart instances. Even though the actor model assumes the topology of a system is completely dynamic, our approach is a special case of this, so it can still be regarded as a proper case of the actor model.

### *3.7. Conclusion*

We had a quick look at the actor model, an abstract model for parallel computation. In the next section, we introduce *Class Diagrams*, and we explain how they can be used to model both the structure of objects and the relationships between them.

## 4. Class Diagrams

Class Diagrams model the structure of classes/objects and the relationships between them. We assume the reader is somewhat familiar with them, so won't go into detail very much. While explaining the class diagrams syntax & semantics, we show how we can use class diagrams to define the structure of a system of objects, whose behavior in turn is specified by statecharts.

### *4.1. Syntax*

The basic entity of a class diagram is a *class*. A class is represented syntactically by a rectangle with the name of the class in it. Beneith the name, we encounter a list of (usually private) variables, followed by another list of (usually public) methods. Figure 10 shows an example.

#### *4.1.1. Variables*

Assume we now have an object whose behavior is specified by a statechart. In Section 2.2.8, we explained how statecharts can use variables to perform computation. The class (diagram) associated with a statechart will list the

[3]Most likely only in composition-type relations.

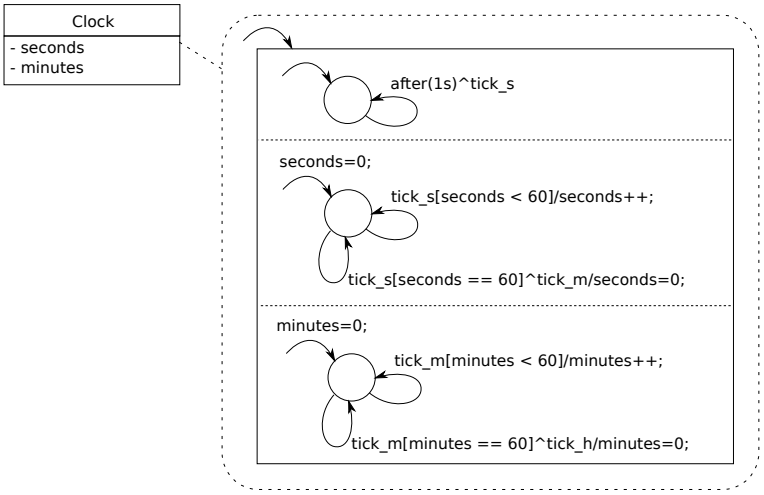


Figure 11: A class with a statechart attached to it.

statechart's variables as private attributes. This way, the class diagram specifies the structure of the object, while the statechart specifies its behavior. Figure 11 shows the statechart from Example 2.4 with a class diagram.

*4.1.2. Methods*

In typical OO design, the methods of an object are its public interface towards other objects. Objects communicate by invoking each other's methods.

Since we will already use statecharts to specify the behavior of objects, and statecharts solely use events for communication, in our approach, objects will not exhibit a list of public methods to be invoked by other objects. The actor model also demands communication between objects/actors to be inherently asynchronous. Event-based communication resembles this paradigm closer.

Even though methods will not be used to carry out communication, we will still use methods to group recurring operations/expressions on local data. The only 2 ways a method can be called are

1. As a side-effect of the statechart making a transition
2. The method was called by another method (of the same object).

In OO terms: methods are *private*.

Methods can be used to name operations/expressions that take a number of steps. This way, more complex behavior can be accessed from an object's statechart, without the need to state that behavior explicitly in the list of conditions/actions of a transition. We also save space if certain behavior is recurring among multiple transitions. Remember the syntax of a transition label

$$trigger[condition]\hat{}generated\_events/actions$$

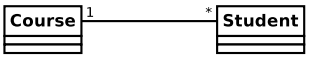


Figure 12: A bidirectional association relationshipship between two classes, with multiplicities on both ends.

Methods can be called from the list of actions, and because a method call is synchronous in nature, we can use its return value to evaluate the condition of a transition.

Methods are not allowed to generate events: No transitions can be triggered directly as the result of a method being executed.

*4.1.3. Relationships*

Up until now, we have seen how we can use class diagrams to specify the structure of individual objects. As mentioned a couple of times, we will also use class diagrams to describe the structure of a system of objects, i.e. to describe the possible communications between objects. In class diagrams, this is done with the use of certain types of *relationships*. Relationships are displayed as lines/arrows between classes/objects.

**Association** We will mostly be concerned with the *association* type of relationship. An association from one class to another means the former can send communications to the latter. An association is syntactically represented by an arrow from one class to another. Sometimes, associations exist in 2 directions: both classes can communicate with each other. In that case, the association is syntactically shown as a line between both classes.

Associations can have *multiplicities*. Multiplicities are numerical values (or ranges of values) indicating the number of associated instances there can be from one class to another. If no multiplicities are specified, the default value is 1. In bidirecional associations, multiplicities can be set on both ends.

Figure 12 shows an association between 2 classes, *Course* and *Student*. The multiplicity at the side of *Course* is 1, which means that every student has 1 *Course* instance it can refer to. The multiplicity on the side of *Student* is * (asterisk), which means that a course can have any number of students, including 0.

**Aggregation** A special case of the association relationship, *aggregation*, is used for "has a" types of relationships. Aggregation is typically used when one class is a container of other classes, but there is no life cycle dependency between one another: When the aggregate class is destroyed, its "parts" are not. Aggregation is denoted with a hollow diamond on the side of the aggregate class. Aggregation relationships are unidirectional, but multiplicities can be set on both ends. If a multiplicity is set on the opposite end of the aggregation, it means there is a normal association in that

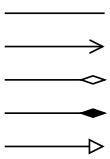

Figure 13: 5 types of relationships. From top to bottom: bidirectional association, unidirectional association, aggregation, composition, inheritance.

direction. We will not be concerned with aggregation very much: because of the absense of life-cycle dependency, aggregation does not make a significant semantical difference over association.

**Composition** *Composition*, in its turn, can be seen as a special case of aggregation: It represents a "part of" type of relationship. There is a life-cycle dependency: When the composite class is destroyed, so are its parts. Composition relationships are always unidirectional: Two instances can never be part of each other. Composition is denoted with a solid diamond on the side of the composite class. Composition makes a significant semantical difference over aggregation: We can use the life-cycle dependency information to optimize the allocation of objects.

An overview of the types of relationships and their syntax can be seen in Figure 13.

Now that we know how to describe the possible communications between classes, let us see how the statecharts of those classes can use this information to send events to each other.

### *4.2. Class Diagram from the Statechart Point-of-View*

All information contained in the class diagram the statechart is attached to, is visible to the statechart. The statechart can see its class' variables, methods and outgoing relationships. Variables can be used in guard condition expressions and actions. Methods can be called from the list of actions and their return value can be used in the evalation of a guard condition.

Relationships are used to create/delete instances and to send events to specific destinations. This is not very trivial: Recall from Section 3.6 that in the actor model, the creation of new actors is carried out by actors themselves. Statecharts, however, cannot natively create new instances, so the creation of new instances has to be dealt with by some external entity. The *object manager* will take care of this.

### *4.3. Object Manager*

The object manager is an entity providing a statechart-like interface to all statecharts. Its main task is handling the creation and deletion of statechart instances at runtime.

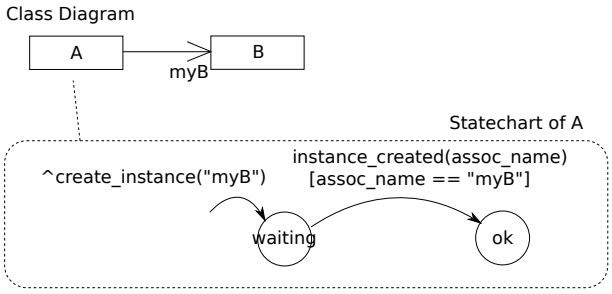


Figure 14: The class diagram at the top shows an association from class $A$ to $B$, called *myB*. When an instance of $A$ is created, the statechart of $A$ requests the creation of an instance of $B$ and waits (in state *waiting*) for the response event *instance_created*.

*4.3.1. Creation And Deletion of Instances*

The procedure of creating/deleting instances is as follows: Whenever a statechart wishes a new instance to be created (or deleted), it sends a request to the object manager. Since statecharts' only manner of communication with the outside world is through asynchronous events, this request will also happen through an event. After making such a request, a response (also in the form of an event) will be sent back to the statechart, denoting the status as *succes* or *failure*.

Figure 14 shows a statechart $A$ that, when created, immediately creates another statechart $B$.

The object manager will have to be completely aware of the topology contained in all classes: When a statechart requests the creation of a new instance, the object manager has to check if a relation between the creator statechart and created statechart exists, and whether the relation is still *available* (not occupied by some other instance). Also, when some instance gets deleted, the object manager has to check whether the instance is no longer part of any mandatory relation with another object. If not, the deletion cannot take place. We will go into further detail on this in Section **??**.

## 5. TODO

- Actor Model originates from A.I. research → this can serve as an additional motivation of why we use it (Glenn's work is on A.I. as well)
- Give examples of how the object manager will check relationships upon creation/deletion.
- Explain how relationships are visible to the statechart, i.e. how the statechart can use relationships from the class diagram to send events to other statecharts
- BSMLs: 8 aspects (this can mostly be copied from old report, but with additional examples)
- Formal Specification: Read Glenn's paper and build on top of that?

## References


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

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

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

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

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