

Modeling and Monitoring of Hierarchical State Machines in Scala

Klaus Havelund and Rajeev Joshi^(✉)

Jet Propulsion Laboratory, California Institute of Technology, Pasadena, USA
{klaus.havelund, rajeev.joshi}@jpl.nasa.gov

Abstract. Hierarchical State Machines (HSMs) are widely used in the design and implementation of spacecraft flight software. However, the traditional approach to using HSMs involves graphical languages (such as UML statecharts) from which implementation code is generated (e.g. in C or C⁺⁺). This is driven by the fact that state transitions in an HSM can result in execution of action code, with associated side-effects, which is implemented by code in the target implementation language. Due to this indirection, early analysis of designs becomes difficult. We propose an internal Scala DSL for writing HSMs, which makes them short, readable and easy to work with during the design phase. Writing the HSM models in Scala also allows us to use an expressive monitoring framework (also in Scala) for checking temporal properties over the HSM behaviors. We show how our approach admits writing *reactive monitors* that send messages to the HSM when certain sequences of events have been observed, e.g., to inject faults under certain conditions, in order to check that the system continues to operate correctly. This work is part of a larger project exploring the use of a modern high-level programming language (Scala) for modeling and verification.

1 Introduction

Hierarchical State Machines (HSMs) [13] are used extensively in the flight software that runs on spacecraft developed by NASA's Jet Propulsion Laboratory (JPL). The current practice is (depending on programmer taste) either to work textually and directly write low-level implementation code in C (which is hard to write and read), or work graphically and automatically synthesize C code from graphical HSM diagrams. In both cases it becomes difficult to prototype and execute (test) design choices during early development because the use of C forces introduction of low-level implementation details, hampering comprehension and analysis. Graphical formalisms are specifically not well suited for mixing control states and non-trivial code to be executed as part of transition actions for example. In this paper, we propose a method that allows HSMs to be

The research performed was carried out at Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration. Copyright 2017 California Institute of Technology. Government sponsorship acknowledged. All rights reserved.

implemented as an internal Domain-Specific Language (iDSL) in the high-level Scala programming language. Using an internal DSL allows us to express the HSM control state model and the code used in transitions all within the same language, which makes it easier to explore different design choices. We use Scala because it offers features that allow iDSLs to be implemented elegantly, including implicit functions, partial functions, call-by-name arguments, and dot-free method calls. We show how our iDSL for HSMs can be used with Daut (Data automata) [14], a library for writing monitors, thus allowing us to check temporal properties over the executions of an HSM. Daut offers a combination of flat state machines and a form of temporal logic, and furthermore allows monitoring of data parameterized events. An interesting feature of our approach is the ability to write *reactive monitors*, which allow the injection of specific events to the HSM when certain temporal properties are satisfied. This work is part of a long-term project exploring the use of a modern high-level programming language for writing models and properties, as well as programs.

The focus of this paper is the modeling and analysis of a single HSM, a situation typically facing a programmer responsible for a single module in a larger system. The modeling and analysis of multiple HSMs executing in parallel and communicating asynchronously via message passing over prioritized queues is described in [16]. Other topics not touched upon (and not used at JPL) are orthogonal regions, history states, and do-activities. The contributions of the paper are as follows. (1) We provide an elegant implementation of an internal Scala DSL for HSMs. (2) We show how single HSMs can be tested with the internal Daut monitoring DSL, which supports data parameterized monitors. This illustrates two forms of state machine DSLs, useful for different purposes. Each DSL is implemented in less than 200 lines of code. (3) We show how Daut monitors can be used to write reactive monitors that allow test scenarios to be described more conveniently. In [16], we extend the framework developed here to model multi-threaded systems with multiple HSMs, define a DSL for writing constraint-based test scenarios, and apply our approach to a real-life case study.

The paper is organized as follows. Section 2 describes related work. Section 3 introduces an example at an informal high level, and presents part of the implementation as an HSM in the Scala HSM iDSL. Section 4 outlines how the HSM is tested through monitoring with the Daut iDSL. Section 5 outlines how the HSM iDSL is implemented. Section 6 outlines how the Daut iDSL is implemented. Finally, Sect. 7 concludes the paper.

2 Related Work

The state pattern [11] is commonly used for modeling state machines in object-oriented programming languages. A state machine is implemented by defining each individual state as a derived class of the state pattern interface, and implementing state transitions as methods. The state pattern does not support hierarchical state machines. A variant of the state pattern to cover HSMs for C and C++ is described in [21]. This is a very comprehensive implementation compared to our less than 200 lines of code. The Akka framework provides features

for concurrent programming and fault protection for the JVM, and in particular it includes a library for writing non-hierarchical finite state machines (FSM) in Scala [1]. The Daut iDSL for monitoring event sequences is related to numerous runtime verification frameworks, including [3, 4, 6, 7, 12, 15, 19]. An approach to use state charts for monitoring is described in [10]. The Umple framework [2] advocates, as we do, an approach to unifying modeling and programming, although it differs by having the modeling language being distinct from the programming language. The system is interesting because it allows updates to the model and the program in the same environment, while supporting visualization of the textual models. In contrast our DSLs are internal, completely eliminating the distinction between modeling language and programming language. Other internal Scala monitoring DSLs have been developed [5, 15]. Daut itself is a simplification of the earlier TraceContract monitoring framework in Scala [5].

A standard way of formally verifying state machines is to encode them in the input language for, say, a model checker. However, this creates a gap between the modeling language and the implementation language. Model checkers have been developed for programming languages, for example Java PathFinder [17]. P# [8] is an extension of C# with concurrently executing non-hierarchical state machines, communicating asynchronously using message passing. It is inspired by the P external DSL [9] for modeling and programming in the same language, translated to C. P# supports specification of environment and monitors as state machines. However, such monitors do not support the temporal logic notation or data parameterized event monitoring that Daut does.

3 Hierarchical State Machines in Scala

Example. In this section, we illustrate our ideas with an example. The example is based on a simple HSM for taking images with a camera. In our example, the HSM can receive a `TAKE.IMAGE(d)` request, where `d` denotes the exposure duration. It responds to this request by sending a command to power on the camera, and waiting until the camera is ready. It then opens the shutter for the specified exposure duration (using a timer service which generates a timeout event after a specified period). Following this, it optionally takes a dark exposure¹ with the shutter closed (but only if the ambient temperature is above a specified threshold). Finally, it saves the image data, and powers off the camera. Although this is a toy example, it serves to illustrate the key ideas in our approach. In a related paper [16], we describe the application of our approach to a real-life case study (a module that manages communication between the Curiosity rover and Earth).

HSM as a Diagram. Figure 1 shows a graphical view of the HSM that implements our simple imaging example. Following standard HSM notation, the filled out black circles indicate the initial substate that is entered whenever a parent

¹ A dark exposure allows determination of the noise from camera electronics, so that this can be subtracted from the acquired image.

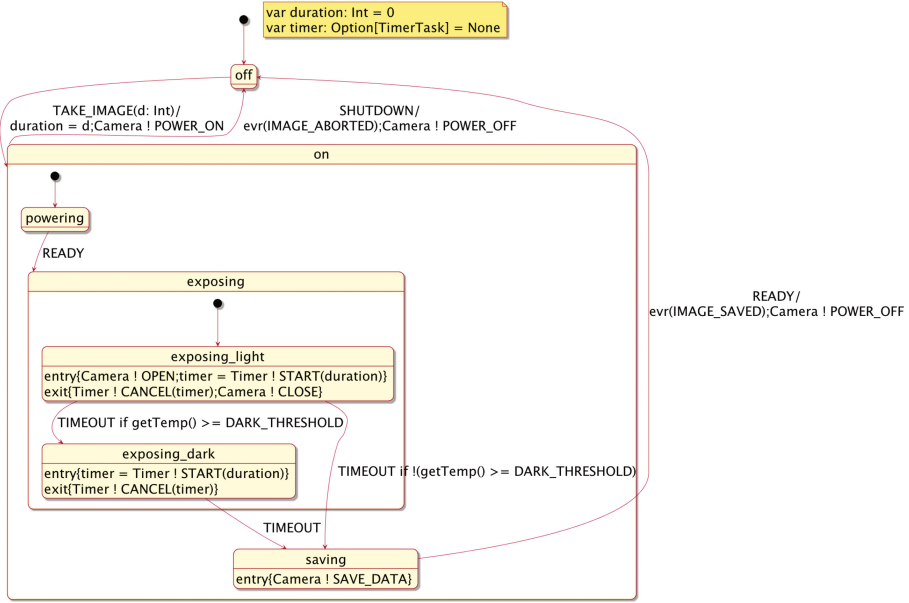


Fig. 1. HSM for imaging example

state is entered. (Thus, for instance, a transition to the **on** state ends with the HSM in the **powering** state.) Associated with each state are also two optional code fragments, called the *entry* and *exit* actions. The *entry* action is executed whenever the HSM transitions into a state, whereas the *exit* action is executed whenever the HSM transitions out of a state. Finally, the labeled arrows between states show the transitions that are caused in response to events received by the HSM. A label has the form `EVENT/code`, which denotes that the transition is triggered when the HSM receives the specified `EVENT`. In response, the HSM transitions to the target state, and executes the specified code fragment (which is optional). As an example, suppose the HSM is in state **exposing_dark**, and it receives the event **SHUTDOWN** (for which a transition is defined in the parent **on** state). This would cause the HSM to perform the following actions (in order): (1) the *exit* action for the state **exposing_light**, (2) the *exit* action for the state **on**, (3) the actions associated with the event handler `evr(IMAG_ABORTED)`; `Camera ! POWER_OFF`, and the (4) the *entry* action for the state **off**.

For our imaging example, the HSM works as follows. As shown in the figure, the HSM is associated with two variables: an integer-valued variable `duration`, which denotes the duration of the exposure when an imaging request is made, and the variable `timer`, of type `Option[TimerTask]`, which denotes whether there is an outstanding timer for which the HSM is waiting. The system starts off in the **off** state (marked initial). In the **off** state, the system responds only to a `TAKE_IMAGE(d)` event (where `d` is an integer). On receipt of this event, the system saves the requested exposure duration `d` in the state variable

```

trait Event
case class TAKE.IMAGE(duration: Int) extends Event
case object SAVE.DATA extends Event
...
class ImagingHsm extends SimpleMslHsm {
  var duration: Int = 0
  var timer: Option[TimerTask] = None

  initial(off)

  object off extends state() {
    when {
      case TAKE.IMAGE(d: Int)  $\Rightarrow$  on exec {
        duration = d ; Camera ! POWER.ON
      }
    }
  }

  object on extends state() {
    when {
      case SHUTDOWN  $\Rightarrow$  off exec {
        evr(IMAGE.ABORTED)
        Camera ! POWER.OFF
      }
    }
  }

  object powering extends state(on, true) {
    when { case READY  $\Rightarrow$  exposing }
  }

  object exposing extends state(on) {}

  object exposing.light extends state(exposing, true) {
    entry { Camera ! OPEN
      timer = Timer ! START(duration) }
    exit { Timer ! STOP(timer) ; Camera ! CLOSE }
    when {
      case TIMEOUT  $\Rightarrow$  {
        if (getTemp() >= DARK.THRESHOLD)
          exposing_dark
        else saving
      }
    }
  }

  object exposing.dark extends state(exposing) {
    entry { timer = Timer ! START(duration) }
    exit { Timer ! STOP(timer) }
    when {
      case TIMEOUT  $\Rightarrow$  saving
    }
  }

  object saving extends state(on) {
    entry { Camera ! SAVE.DATA }
    when {
      case READY  $\Rightarrow$  off exec {
        evr(IMAGE.SAVED) ; Camera ! POWER.OFF
      }
    }
  }
}

```

Fig. 2. The HSM for the imaging example in our internal DSL

duration, sends a request to the camera to power on (indicated by the action `Camera!POWER.ON`), and then transitions to the `on` state, which is a super-state, so the HSM ends up in the `powering` state. Here it waits until it gets a `READY` event from the camera, upon which it enters the `exposing` superstate, which in turn causes it to enter the initial `exposing.light` substate. The entry action for this state sends a request to the camera to `OPEN` the shutter, and then starts a timer for the specified `duration` (which was saved on receipt of the command). When the timer expires, the HSM receives a `TIMEOUT` event, which causes it to either transition to the `exposing.dark` state (if the ambient temperature is at least the `DARK.THRESHOLD`), or the `saving` state (if temperature is below the `DARK.THRESHOLD`). Whenever it is in a substate of the `on` state, the HSM can respond to a `SHUTDOWN` request, which causes it to power off the camera and transition back to the `off` state.

HSM in Scala. Figure 2 shows the formalization, in our internal Scala DSL, of the HSM in Fig. 1. The model first defines the `Event` type, denoting the set of events that trigger transitions in the HSM. The state machine itself is defined as the class `ImagingHsm` extending `SimpleMslHsm` which in turn extends the HSM trait defined by our iDSL, which is parameterized by the type of events sent to the HSM. The local state variable `duration` records the duration when an imaging

request is received; this duration is used for both the light exposure (with shutter open) and the (optional) dark exposure (with shutter closed). The variable `timer` records the value of any timer for which the HSM is waiting (the value is `None` if there is no timer currently in progress). In the `off` state, the event handler for the `TAKE.IMAGE(d)` event causes the HSM to execute the action code which records the value `d` in variable `duration` and sends a `POWER_ON` request to the camera. The HSM then transitions to the `on` superstate, and in turn to its initial substate, the `powering` state. (In our iDSL, initial substates are denoted by the keyword `true` as the second argument to the `extends state(..)` declaration.) In the `powering` state, receipt of the `READY` event causes a transition to the `exposing` state, and in turn to the `exposing_light` substate, where the entry actions result in a request to `OPEN` the shutter and start a timer for the specified `duration`. The rest of the example is similar. Since graphical formalisms are useful for human comprehension, we have developed a tool based on Scalameta (see <http://scalameta.org>) that generates a visual representation from the HSM description in our Scala iDSL. This tool generated Fig. 1 directly from the code in Fig. 2.

During execution, the HSM generates a *log* that contains *event reports*, generated for monitoring purposes. Our HSM framework contains special calls of the form `evr(E)` which generates the event `E`, which can be used in monitors. For instance, as shown in Fig. 2, when a `SHUTDOWN` request is received in state `on`, the HSM generates the `IMAGE_ABORTED` log event. Similarly, the `Timer` service generates events when a timer is started, is fired or is stopped. (These timer events are used in the monitors described in the following section.) Fig. 3 shows a sample log of the HSM corresponding to a test scenario in which a `TAKE.IMAGE(7)` is requested at time 101, and completes normally, followed by a `TAKE.IMAGE(10)` requested at time 200, which is terminated by sending a `SHUTDOWN` request at time 205. This log can be checked by the Daut monitoring engine to verify that the HSM execution obeys given temporal properties.

101:RECEIVED_REQUEST(TAKE.IMAGE(7))	200:RECEIVED_REQUEST(TAKE.IMAGE(10))
104:POWERING_ON	203:POWERING_ON
104:SHUTTER.IS_OPENED	203:SHUTTER.IS_OPENED
104:TIMER.STARTED	203:TIMER.STARTED
111:TIMER.FIRED	205:SHUTDOWN.REQUESTED
111:TIMER.CANCELED	205:TIMER.CANCELED
111:SHUTTER.IS_CLOSED	205:SHUTTER.IS_CLOSED
111:TIMER.STARTED	205:IMAGE_ABORTED
118:TIMER.FIRED	205:POWERING_OFF
118:TIMER.CANCELED	
120:SAVING.STARTED	
120:IMAGE.SAVED	
120:POWERING_OFF	

Fig. 3. Sample event log for imaging example

HSM Execution. As mentioned in the introduction, the focus of this presentation is the modeling and analysis of single HSMs. The modeling and analysis of multiple HSMs executing in parallel is described in [16], where we model the

full complexity of the implementation, such as message priorities, queue enabling and disabling, test scenario specifications, and analysis of timing properties. The HSM is composed with an *environment*, which *submits* events to, and receives requests from, the HSM as explained in the following. The environment contains a mutable set of events, which are waiting to be submitted to the state machine. This set can be augmented with new events from a test script and the HSM. In each iteration, the environment picks a random event from the set and submits it to the state machine. The state machine executes as far as it can, possibly sending new requests back to the environment, simulating communication with other state machines. The environment in addition keeps a mapping from requests it can receive to operations on the event set. For example, if the environment receives a `timer!START(d)` request, it adds a `TIMEOUT` event to the event set. This `TIMEOUT` event will then eventually be submitted back to the state machine after `d` seconds have elapsed. The notation `recv ! E` in the HSM denotes the sending of a request `E` to the receiver `recv` (via the environment). In our example, the receiver `Camera` denotes the camera hardware, whereas the receiver `Timer` denotes the timer service.

4 Monitoring with Daut

Daut (Data Automata) [14] is a simple internal Scala DSL for writing monitors on event streams. Daut, like many runtime verification systems, offers two major capabilities that HSMs do not: (i) the ability to track the state behavior for multiple instances of some data (spawning automata), and (ii) a convenient temporal logic formalism on top of a state machine formalism. In this section, we show how to use the Daut monitoring library to specify and monitor that certain temporal properties are satisfied by the executing HSM. We also show how one can use the monitoring framework to build *reactive monitors*, which allow us to inject events into the HSM when certain temporal patterns are met.

Figure 4 shows four temporal property monitors, representing requirements that the imaging HSM must satisfy. Each property is modeled as a class extending the `MSLMonitor` class, which itself is defined as an extension of the `Daut Monitor` class, which is parameterized with the type `EventReport` of event reports being monitored. The `Monitor` class defines the features of Daut. The `MSLMonitor` class defines additional functions that simplify writing monitors for our example. The monitors receive event reports as they are generated by the HSM and update their internal state accordingly, reporting any observed violations.

The first property, `TimerUse`, checks that once a timer is started, it should either fire or be canceled before another timer is started. The body of the class is an `always`-formula. The function `always` takes as argument a partial function from events to monitor states. In this case, whenever a `TIMER_STARTED` is observed, the monitor moves to a `watch` state, in which it is waiting for either a `TIMER_FIRED` or `TIMER_CANCELED` event – another `TIMER_STARTED` event is an error if observed before then.

The second property, `TimerState`, checks that if a timer is currently running (has been started but has not yet fired or been canceled), then the HSM must be

```

class MSLMonitor extends Monitor[EventReport] {
  def inState(name: String) = during(EnterState(name))(ExitState(name))
  ...
}

class TimerUse extends MSLMonitor {
  always {
    case TIMER.STARTED => watch {
      case TIMER.FIRED | TIMER.CANCELED => ok
      case TIMER.STARTED => error("Timer started before previous cancelation")
    }
  }
}

class TimerState extends MSLMonitor {
  val timerOn = during(TIMER.STARTED)(TIMER.FIRED, TIMER.CANCELED)
  val inExposing = inState("exposing")

  invariant("TimerState") {
    timerOn ==> inExposing
  }
}

class ImageRequest extends MSLMonitor {
  always {
    case RECEIVED.REQUEST(TAKE.IMAGE(d)) => hot {
      case IMAGE.SAVED | IMAGE.ABORTED => ok
      case RECEIVED.REQUEST(TAKE.IMAGE(.)) => error("Image was not saved or aborted")
    }
  }
}

class ImgReactiveMonitor extends MSLMonitor {
  always {
    case POWERING.ON => watch {
      case SHUTTER.IS_OPENED => perhaps { Env.delayEvent(2, SHUTDOWN) }
    }
  }
}

```

Fig. 4. Monitors for the imaging example

in the **exposing** state, meaning in any of its substates. The Boolean expression occurring as argument to the **invariant** function gets evaluated in each new state the HSM enters. The notation $p \implies q$ denotes implication and is interpreted as $!p \parallel q$. The property uses the **during** construct to define the period during which the timer is active, namely in between an observed **TIMER.STARTED**, and either a **TIMER.FIRED** or **TIMER.CANCELED** event report is observed. Also the **inState** function defined in class **MSLMonitor** is defined using the **during** function, here tracking the event reports indicating respectively entering and subsequently exiting a state.

The third property, **ImageRequest**, is similar to the **TimerUse** property in form, and checks that once an image request has been received, then eventually the image must be saved or the imaging must be aborted. It is an error if another image request is received before then. The **hot** operator causes Daut to check

that the image saving or image abortion is seen before the end of the execution (Daut reports an error if there are any **hot** states active at the end of the trace).

We have just discussed how we can use Daut to specify and monitor temporal properties. Since Daut is a Scala library, we can write Daut monitors to also take actions during a run, such as causing new events to be sent to the HSM, thus affecting the resulting behavior. We refer to such Daut monitors as *reactive monitors*. The last monitor, `ImgReactiveMonitor`, in Fig. 4 is an example of a reactive monitor, in this case randomly sending a **SHUTDOWN** event to the HSM whenever the monitor sees a **POWERING_ON** event followed by a **SHUTTER.IS_OPENED** event report. The `perhaps` function takes a code fragment (call-by-name) and randomly decides whether or not to execute it. In our example, this monitor results in a **SHUTDOWN** event being sent to the HSM 2s after the **SHUTTER.IS_OPENED** event is seen. In the example execution trace shown in Fig. 3, there are two occurrences where the monitor sees a **POWERING_ON** followed by an **SHUTTER.IS_OPENED** event report. The `perhaps` function decided to execute the action after the second instance of the **SHUTTER.IS_OPENED** event report (which occurs at time 203), issuing a **SHUTDOWN** at time 205.

5 HSM Implementation

The concept of a hierarchical state machine is implemented as the Scala trait `HSM` (a trait is similar to an interface in Java), which a user-defined HSM must extend, and which is parameterized with the type `Event` of events that can be submitted to it:

```
trait HSM[Event] {...}
```

The `HSM` trait defines the following types and values used throughout:

```
type Code = Unit ⇒ Unit
type Target = (state, Code)
type Transitions = PartialFunction[Event, Target]
val noTransitions: Transitions = {case _ if false ⇒ null}
val skip: Code = (x: Unit) ⇒ {}
```

`Code` represents code fragments (with no arguments and returning no result), that are to be executed on event transitions, and in entry and exit blocks. A `Target` represents the target state and the code to be executed when a transition is taken. `Transitions` represents the transitions leading out of a state, encoded as partial functions from events to targets. Applied to an event a transition function will either be undefined on that event (corresponding to the transition not being enabled), or it will return a target. The default transition function from a state is represented by `noTransitions` which is undefined for any event. Finally, `skip` represents the code with no effect.

We can now present the `state` class encoding the states in a state machine. The contents of this class can be divided into the DSL “*syntax*”, permitting a user to create a state, and the DSL *implementation*. The DSL syntax, including its update on internal variables, can be presented as follows:

```

case class state(parent: state = null, init: Boolean = false) {
  var entryCode: Code = skip
  var exitCode: Code = skip
  var transitions: Transitions = noTransitions
  ...
  def entry(code: ⇒ Unit): Unit = {entryCode = (x: Unit) ⇒ code}
  def exit(code: ⇒ Unit): Unit = {exitCode = (x: Unit) ⇒ code}
  def when(ts: Transitions): Unit = {transitions = ts}

  implicit def state2Target(s: state): Target = (s, skip)
  implicit def state2Exec(s: state) = new {
    def exec(code: ⇒ Unit) = (s, (x: Unit) ⇒ code) }
}

```

The class is parameterized with the parent state (if it is a sub-state), and whether it is an initial state of the parent state (false by default). The class declares three variables, holding respectively the entry code (to be executed when entering the state), the exit code (to be executed when leaving the state), and the transition function, all initialized to default values. Three methods for updating these are furthermore defined. The code parameters to the first two functions `entry` and `exit` are declared as “call by name”, meaning that at call time a code argument will not be evaluated, and will instead just be stored as functions in the appropriate variables. Since a method application $f(e)$ in Scala can be written using curly brackets: $f\{e\}$, we achieve the convenient code-block syntax for writing calls of these methods, making these methods appear as added syntax to Scala.

Finally two implicit functions are defined. An implicit function f will be applied to any expression e by the compiler if e occurs in a context $C[e]$ which does not type check, but $C[f(e)]$ does type check. Implicit functions are useful for defining elegant DSLs. In this case, the implicit function `state2Target` lifts a state to a target, allowing us to just write states as targets on transitions (and no code), and the function `state2Exec` lifts a state to an anonymous object, defining an `exec` method, allowing transition right-hand sides like: `top exec {table.insert(w)}`. The above definitions show the HSM syntax and how it is used to define states and transitions. In addition, the function `initial` is used for identifying the initial state in the HSM:

```

def initial(s: state): Unit = {current = s.getInnerMostState}

```

The function `getInnerMostState` is defined in the class `state` as follows, along with a method for finding the super states of a state (needed for executing HSMs):

```

var initialState: state = null
if (parent != null && init) {parent.initialState = this}

def getInnerMostState: state =
  if (initialState == null) this else initialState.getInnerMostState

```

```
def getSuperStates: List[state] =
  (if (parent == null) Nil else parent.getSuperStates) ++ List(this)
```

When a state is created, if it is an initial state, the `initialState` variable of the parent is initialized with the just created state (`this`). When a state is the target of execution, the innermost initial state of that state is the one becoming active.

An HSM is at any point in time in a current state, and will potentially change state when an event is *submitted* to it from the environment. Current state and the event submission method are defined as follows.

```
var current: state = null

def submit(event: Event): Unit = {
  findTriggerHappyState(current, event) match {
    case None =>
    case Some(triggerState) =>
      val (transitionState, transitionCode) = triggerState.transitions(event)
      val targetState = transitionState.getInnerMostState
      val (exitStates, enterStates) = getExitEnter(current, targetState)
      for (s <- exitStates) s.exitCode()
      transitionCode()
      for (s <- enterStates) s.entryCode()
      current = targetState
  }}
}
```

When executed from the `current` state, and given the submitted `event`, the function call `findTriggerHappyState(current, event)` finds the innermost state containing (or being equal to) `current`, which is ready to transition on the event. The result is `Option[state]`, where `None` represents that no such state exists. In case such a state exists, its transition function is applied to the event, obtaining a target (target state, and code to execute), then the innermost initial state of the target state is computed, and based on current and target state, we compute the list of states to exit and the list of states to enter via the call `getExitEnter(current, targetState)`, whose implementation is straightforward and not shown. It computes the super states (listed top down) for respectively the from-state and the to-state, and then strips off the common prefix of the two lists. The remaining lists are the lists of states to exit and enter respectively. Now we can execute exit codes, the transition code itself, and entry codes. Note that requests sent by the state machine in these code fragments will go back to the *environment*, which then in later iterations will submit corresponding events back to the state machine, as explained earlier. For performance reasons, we want to avoid repeated computation of innermost state for a state, and the list of exit and entry states. Thus our implementation caches these so they are only computed once (this is done with an additional 20 lines of code, not shown here due to space limitations). We can now define the function for finding the innermost enclosing state of the current state, containing a transition function enabled for an event:

```

def findTriggerHappyState(s: state, event: Event): Option[state] =
  if (s.transitions.isDefinedAt(event)) Some(s) else
  if (s.parent == null) None else findTriggerHappyState(s.parent, event)

```

The function calls itself recursively up the parent chain until it finds a state whose transition function is defined on the event. For verification purposes, a function is defined for determining which state (by name, including super states) an HSM is in, matching against a regular expression:

```

def inState(regexp: String): Boolean = {
  current.getSuperStates.exists(_.name.matches(regexp))
}

```

The presented code is the implementation in its entirety, except for the following concepts (30 lines of code): (i) computing exit/enter state chains; (ii) caching of computations of innermost states and exit/enter state chains; (iii) the ability for the user to announce call-back functions to be called whenever a state is entered, exited, or the monitor reaches a quiescent state.

6 Daut Implementation

The general idea behind the implementation of Daut is described in [14] (although the version used in this work differs in minor ways), summarized here with the addition of temporal invariants (*during* and *invariant*). The class `Monitor` contains a variable holding at any point during monitoring the set (logic conjunction) of active monitor states²:

```

class Monitor[E <: AnyRef] {
  type Transitions = PartialFunction[E, Set[state]]
  var states: Set[state] = Set()
  ...
}

```

A state is an instance of the following class, which contains a variable holding the transitions out of the state, as well as a variable indicating whether it is an acceptance state (`acc = true`) or not (by default a state is an acceptance state).

```

trait state {
  var transitions: Transitions = noTransitions
  var acc: Boolean = true
  if (first) {states += this; first = false}
}

```

```

def apply(event:E): Option[Set[state]] =
  if (transitions.isDefinedAt(event)) Some(transitions(event)) else None

```

² Note that there is some terminology overlap between the HSM DSL and the Daut DSL, e.g. the concepts of states and transitions, with similar meanings although not necessarily in the details. This works out due to a clear separation of name spaces in that HSMs and Daut monitors extend different classes (HSM and Monitor).

```

def watch(ts:Transitions) {transitions = ts}
def always(ts:Transitions) {transitions = ts andThen (_ + this)}
def hot(ts:Transitions) {transitions = ts; acc = false}
def next(ts:Transitions) {transitions = ts orElse {case _  $\Rightarrow$  error}; acc=false}
}

```

The first state created in a monitor becomes the initial state, e.g. the **always**-state in our monitors. A state is applied (the **apply** method) to an event to return an optional set of results, and **None** if the state does not contain transitions defined for the event. In addition a collection of temporal methods are defined: **watch**, **always**, **hot**, and **next**. Other methods are defined in the actual system, including **weaknext**, **until**, and **weakuntil**, known from temporal logic. These methods take as argument a transition function and store it or a modification of it in the **transitions** variable of the state, and also set the **acc** variable for non-acceptance states. The **Monitor** class in addition defines a method for each of the temporal methods defined inside the **state** class for creating states of the corresponding temporality. We show one of these, the rest follow the same pattern:

```

def always(ts: Transitions) = new state { always(ts) }

```

The **during**³ class is defined as a particular form of state. It contains a Boolean variable **on**, true when one of the **e1** events has been observed but an **e2** event has not yet been observed.

```

case class during(e1: E*)(e2: E*) extends state {
  states += this
  val begin = e1.toSet
  val end = e2.toSet
  var on: Boolean = false

  def startsTrue: during = {on = true; this} // allows interval initially true

  always {
    case e  $\Rightarrow$  if (begin.contains(e)) {on = true} else
      if (end.contains(e)) {on = false}
  }
}

```

We have seen how an object of class **during** can be used as a Boolean (e.g. **timerOn** in Fig. 4). This is made possible with the following implicit function:

```

implicit def liftInterval(iv: during): Boolean = iv.on

```

We finally illustrate how invariants are realized. A variable contains all declared invariants (as pairs of an error message, and the predicate itself). An invariant is declared with **invariant**(txt)(p) (where **p** is a call-by-name argument not evaluated before **invariant** is called), checked initially and after each event processing.

³ The $during(P)(Q)$ operator is inspired by the $[P, Q]$ operator in MaC [18].

```
var invariants: List[(String, Unit  $\Rightarrow$  Boolean)] = Nil
```

```
def invariant(e: String)(inv:  $\Rightarrow$  Boolean): Unit = {
  invariants \newcolon= (e, ((x: Unit)  $\Rightarrow$  inv))
  check(inv, e)
}
```

```
def check(b: Boolean, e: String) : Unit = {if (!b) printErrorMessage(e)}
```

We finally show how event reports are issued with the `verify` method, and how monitoring is ended (given a finite trace) with the `end` method. Note how invariants are evaluated after each processed event.

```
def verify(event: E) {
  for (sourceState <- states) {
    sourceState(event) match {
      case None  $\Rightarrow$ 
      case Some(targetStates)  $\Rightarrow$ 
        statesToRemove += sourceState
        for (targetState <- targetStates) {
          targetState match {
            case 'error'  $\Rightarrow$  printErrorMessage()
            case 'ok'  $\Rightarrow$ 
            case _  $\Rightarrow$  statesToAdd += targetState
          }
        }
      }
    }
    states -= statesToRemove; states += statesToAdd
    statesToRemove = Set(); statesToAdd = Set()
    invariants foreach { case (e, inv)  $\Rightarrow$  check(inv(), e) }
  }

def end() {
  val hotStates = states filter (!_.acc)
  if (!hotStates.isEmpty) {printErrorMessage();...}
}
```

7 Conclusion and Future Work

We have shown how HSMs can be elegantly modeled in an internal DSL in the Scala programming language. The iDSL has been illustrated with a simple example of an HSM used for taking images with a camera. We have additionally illustrated how an existing internal Scala DSL for monitoring was extended and applied to testing the HSM. In particular, our approach allows the definition of reactive monitors, which can send events to the HSM when certain temporal properties are satisfied, which makes it easier to write complex test cases. The code for each of these iDSLs is less than 200 lines, which makes it easier to validate their semantics. A more comprehensive validation would be to model

an existing HSM (written in C) in our iDSL (as done in [16]), and compare execution logs on the same inputs. We have also developed a capability for generating graphical representations (used to generate Fig. 1) directly from the HSM description in our iDSL. The work illustrates how a high-level programming language can be used for modeling as well as programming, as part of a model-based engineering approach. We plan to support refinement of high-level models into low-level programs which can directly be translated into C code. We are working on extending our approach to support automated test-case generation (using an SMT solver) and formal verification of Scala programs using the Viper framework [20].

References

1. Akka FSMs. <http://doc.akka.io/docs/akka/current/scala/fsm.html>
2. Umple - Model-Oriented Programming. <http://cruise.site.uottawa.ca/umple>. Accessed 26 May 2017
3. Barringer, H., Falcone, Y., Havelund, K., Reger, G., Rydeheard, D.: Quantified event automata: towards expressive and efficient runtime monitors. In: Gianakopoulou, D., Méry, D. (eds.) FM 2012. LNCS, vol. 7436, pp. 68–84. Springer, Heidelberg (2012). doi:[10.1007/978-3-642-32759-9_9](https://doi.org/10.1007/978-3-642-32759-9_9)
4. Barringer, H., Goldberg, A., Havelund, K., Sen, K.: Rule-based runtime verification. In: Steffen, B., Levi, G. (eds.) VMCAI 2004. LNCS, vol. 2937, pp. 44–57. Springer, Heidelberg (2004). doi:[10.1007/978-3-540-24622-0_5](https://doi.org/10.1007/978-3-540-24622-0_5)
5. Barringer, H., Havelund, K.: TRACECONTRACT: a scala DSL for trace analysis. In: Butler, M., Schulte, W. (eds.) FM 2011. LNCS, vol. 6664, pp. 57–72. Springer, Heidelberg (2011). doi:[10.1007/978-3-642-21437-0_7](https://doi.org/10.1007/978-3-642-21437-0_7)
6. Barringer, H., Rydeheard, D., Havelund, K.: Rule systems for run-time monitoring: from EAGLE to RuleR. J. Logic Comput. **20**(3), 675–706 (2010)
7. Basin, D., Klaedtke, F., Marinovic, S., Zălinescu, E.: Monitoring of temporal first-order properties with aggregations. Formal Methods Syst. Des. **46**(3), 262–285 (2015)
8. Deligiannis, P., Donaldson, A.F., Ketema, J., Lal, A., Thomson, P.: Asynchronous programming, analysis and testing with state machines. In: Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2015, pp. 154–164. ACM, New York (2015)
9. A. Desai, V. Gupta, E. Jackson, S. Qadeer, S. Rajamani, and D. Zufferey. P: Safe asynchronous event-driven programming. In Proceedings of PLDI '13, pages 321–332, 2013
10. Drusinsky, D.: Modeling and Verification using UML Statecharts, p. 400. Elsevier, Amsterdam (2006). ISBN-13: 978-0-7506-7949-7
11. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Boston (1995)
12. Hallé, S., Villemare, R.: Runtime enforcement of web service message contracts with data. IEEE Trans. Serv. Comput. **5**(2), 192–206 (2012)
13. Harel, D.: Statecharts: A visual formalism for complex systems. Sci. Comput. Program. **8**(3), 231–274 (1987)
14. Havelund, K.: Data automata in Scala. In: Proceeding of the 8th International Symposium on Theoretical Aspects of Software Engineering (TASE 2014) (2014)

15. Havelund, K.: Rule-based runtime verification revisited. *Int. J. Softw. Tools Technol. Transfer* **17**(2), 143–170 (2015)
16. Havelund, K., Joshi, R.: Modeling rover communication using hierarchical state machines with Scala. In: *TIPS 2017*, May 2017. Accepted for publication
17. Havelund, K., Visser, W.: Program model checking as a new trend. *STTT* **4**(1), 8–20 (2002)
18. Kim, M., Viswanathan, M., Kannan, S., Lee, I., Sokolsky, O.: Java-MaC: a runtime assurance approach for Java programs. *Formal Methods Syst. Des.* **24**(2), 129–155 (2004)
19. Meredith, P., Jin, D., Griffith, D., Chen, F., Roşu, G.: An overview of the MOP runtime verification framework. *STTT*, pp. 1–41 (2011)
20. Müller, P., Schwerhoff, M., Summers, A.J.: Viper: a verification infrastructure for permission-based reasoning. In: Jobstmann, B., Leino, K.R.M. (eds.) *VMCAI 2016*. LNCS, vol. 9583, pp. 41–62. Springer, Heidelberg (2016). doi:[10.1007/978-3-662-49122-5_2](https://doi.org/10.1007/978-3-662-49122-5_2)
21. Samek, M.: *Practical UML statecharts in C/C++*. In: *Event-Driven Programming for Embedded Systems*, 2nd edn. Newnes, MA, USA (2009)

Software Engineering for Resilient Systems
9th International Workshop, SERENE 2017, Geneva,
Switzerland, September 4-5, 2017, Proceedings
Romanovsky, A.; Troubitsyna, E.A. (Eds.)
2017, XIV, 201 p. 56 illus., Softcover
ISBN: 978-3-319-65947-3