Modeling Rover Communication using Hierarchical State Machines with Scala

Klaus Havelund and Rajeev Joshi

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

We demonstrate the application of a new domain-specific language (DSL) for modeling Hierarchical State Machines (HSMs) to the software that manages communications for the Curiosity Mars rover. The spacecraft software is multi-threaded, where some threads implement an HSM that interacts with hardware devices, operating system services, and with other threads via asynchronous and prioritized message passing. Our DSL is implemented as a shallow embedding within the programming language Scala, which makes our models of HSMs textual, short, readable, and perhaps most importantly: easy to write, modify and test at design time. We also present a notation for writing high-level test scenarios that drive the system, and show how we use class inheritance to compactly express derived tests that are variations of a baseline test scenario. We furthermore apply a monitoring Scala DSL for checking temporal logic properties over the running log of events being generated by the HSMs. We show how our framework can be used to define *reactive monitors* that can be used to modify baseline test scenarios by injecting events when certain temporal constraints are met. We describe how we have used reactive monitors to identify certain timing assumptions made in the design. The work described here is part of a broader effort that is exploring the use of a modern high-level programming language for systems modeling, as an alternative to using a formal specification/modeling language.

1 Introduction

Embedded systems such as spacecraft flight software are typically written in low-level implementation languages like C and C^{++}, which provide the level of control and low overhead that is needed for such systems. However, modern spacecraft software is quite complex, and there has been an increasing trend towards developing intermediate, higher-level formalisms that make it easier for developers to design and write flight software. One formalism that is used at NASA's Jet Propulsion Laboratory (JPL) is Hierarchical State Machines (HSMs) [19]. Conventional finite state machines have a finite number of control states and transitions are labeled with atomic letters over a finite alphabet. HSMs allow the declaration of mutable state variables, which can be used in transition guards, and updated in transition actions. HSMs used in flight software are often difficult to write and understand due to the mixture of control states (the states in the state machine), code to be executed when taking transitions, and the pres-

ence of timers, device interactions, thread priorities, and asynchronous message communication.

The current approach used at JPL is to design the control structure of these HSMs using a graphical tool or a limited domain-specific language (DSL), and to separately write the code that is executed on transitions directly in the implementation language (C/C^{++}). While this approach helps somewhat by saving the developers from having to manually write code for managing the control states of the HSM, the translation from an intermediate DSL directly to low-level implementation code makes it onerous to experiment with design choices during early development. In this paper, we present the application of an internal Scala DSL (called iHSM) for writing HSMs, initially presented in [14], to the modeling of the software that manages communications for the Curiosity Mars rover. We extend the approach presented there (which models a single HSM) to modeling complex software systems in which multiple HSMs are executing concurrently and interacting with each other using asynchronous messaging. We illustrate how iHSM simplifies prototyping and modeling by integrating the notation for describing the HSM control structure within the same language (Scala) that is used to write the actions executed on transitions. We also introduce a notation for describing high-level test scenarios, that can be used to compactly and quickly specify test cases on which the HSMs can be exercised. We furthermore describe the application of the Daut monitoring framework [12] (also a Scala DSL), which can be used to express properties in temporal logic, allowing developers to quickly write and check temporal properties over HSM runs driven by scenario test cases, as well as writing *reactive monitors* that inject events into a running system when some specified temporal conditions are met. We show how iHSM fills a much-needed modeling formalism that allows developers to quickly prototype and test HSM designs.

The contributions of the paper are as follows. (1) We extend the iHSM notation developed in our previous work [14] to model multi-threaded systems, in which multiple HSMs are running in separate threads and interacting with each other via asynchronous messaging. (2) We apply our approach to a real-life case study[1]: the Coordinated Communications Behavior Module (CBM) used in the Curiosity rover for managing communications with Earth. (3) We develop a simple framework for expressing test constraints and a test engine that can automatically run tests satisfying these constraints. (4) We apply the Daut monitoring framework we have developed that allows writing properties in temporal logic and checking these properties during test runs. Our approach is part of a broader effort exploring the use of a modern high-level programming language for systems modeling, as an alternative to using a formal specification language such as VDM or a semi-formal modeling language such as SysML, as discussed in [6, 11]. We have chosen Scala for our work as it is a statically typed object-oriented functional programming language which provides many conve-

[1] Due to JPL restrictions on sharing of flight artifacts, neither the full case study in C, nor its complete formalization in Scala, can be made publicly available.

nient features (such as implicit functions, partial functions, call-by-name) that make it easy to develop internal DSLs.

The paper is organized as follows. Section 2 describes related work. Section 3 provides a high-level overview of the architecture of the flight software running on the Curiosity rover, and of the CBM module that is the target of our case study. Section 4 describes how the HSM for the CBM module is modeled in the iHSM notation, how test scenarios are expressed in iHSM, how monitors are expressed in Daut, and how they can be used to check timing assumptions. Finally, Section 5 concludes the paper.

2 Related Work

The state pattern [10] is commonly used for modeling state machines in object-oriented programming languages. A state machine is implemented by defining each state as a derived class of the state pattern interface, and by implementing state transitions as methods. The original state pattern does not, however, support hierarchical state machines. A variant of the state pattern to cover HSMs for C and C^{++} is described in [19]. This is a very comprehensive implementation compared to our less than 200 lines of code. However, using C and C^{++} is cumbersome for early modeling and analysis of a design. 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 DSL for monitoring event sequences is related to numerous runtime verification frameworks, including [17, 3, 13, 5]. An approach to use state charts for monitoring is described in [9]. Other internal Scala monitoring DSLs have been developed [4, 13, 16]. Daut itself is a simplification of the earlier TraceContract monitoring framework in Scala [4].

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 (JPF) [15] (JPF was originally developed to also support Java as a modeling language). P$^{\#}$ [7] 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 [8] for modeling and programming in the same language, translated to C. P$^{\#}$ supports specification of an environment also as a state machine. Monitors are written as state machines as well, distinguishing between cold and hot (eventually) states, as in Daut. However, these monitors do not support the temporal logic like notation or data parameterized monitors that are expressible in Daut. P$^{\#}$ programs can be analyzed statically for data races, and explored dynamically using randomized testing, exploiting the static analysis results.

Our HSMs differ from UML statecharts (SCs) [2] in a number of ways. First, in UML SCs any state can consist of orthogonal regions executing in parallel. In our approach orthogonal regions are only allowed at the outermost level, where

multiple HSMs run concurrently. Thus, concepts such as fork and join found in UML SCs are not available in HSMs. Second, while communication between UML SCs is synchronous (hand-shake communication), the communication between HSMs is asynchronous: a message sent from a machine A to a machine B ends up in B's input queue and is only consumed by B when its associated thread runs. Third, UML SCs support a built-in limited notion of timers, whereas HSMs support explicit programming of timers, which is needed to model JPL flight software faithfully. HSMs do not currently support history states, but we plan to add this in the future.

3 Overview of the Curiosity Flight Software Architecture

In this section, we give a brief overview of the flight software (FSW) architecture for the Curiosity rover. The main computer on Curiosity is a radiation-hardened PowerPC processor (the BAE RAD750) running the WindRiver VxWorks Operating System, with a priority-preemptive scheduler[2]. The Curiosity FSW consists of around 150 threads that communicate with each other via asynchronous messaging. In the following subsections, we describe the Curiosity software architecture in more detail.

3.1 Threads and Message Handling

Each thread has an associated incoming queue for storing messages sent to that thread. A thread T_1 may send a message M to any thread T_2 in the system (including itself). The message M is appended to the incoming queue for T_2. A key property of the Curiosity FSW is that sending a message is a nonblocking operation[3]. However, receiving a message from a queue is a blocking operation, which causes the thread to be suspended until a message becomes available. Figure 1 shows an outline of the main loop that is run by each thread.

```
1 while (true) {
2   m = msg_receive()      // blocks until a message is available
3   message_handler(m)     // nonblocking code, may only send messages
4 }
```

Fig. 1. Outline of main thread loop

[2] A priority-preemptive scheduler schedules for execution the highest priority task that is ready to run.

[3] If a message queue is full, an attempt to send a message to that queue results in either the message being dropped (for noncritical messages), or causes a system exception (for critical messages).

As shown in the figure, each thread executes an infinite loop that waits for a message to become available in its incoming queue; when a message m becomes available, the thread is unblocked and then executes the message_handler function, which processes m. As noted above, as per the architectural rules, the message_handler function is required to be nonblocking, so it may only send messages; it is not allowed to receive any messages.

One quirk of the Curiosity FSW design is that messages delivered to a thread are not consumed in a strict FIFO order. Instead, a thread's incoming queue consists of an ordered sequence $Q_1, Q_2, ..Q_n$ of *subqueues*. The ordering of subqueues denotes message priority, so messages in subqueue Q_k have higher priority than messages in subqueue Q_{k+1}. When a message is delivered to a thread, it is appended to one of the subqueues (depending on the priority associated with the message). Additionally, each subqueue Q_i is associated with a boolean flag B_i, which indicates whether the subqueue is *enabled* for receiving. We say a message M is *pending* for a thread if M is at the head of an enabled subqueue Q_i (that is, such that B_i is true). The msg_receive operation then retrieves the highest priority message that is pending for a thread. If there are no pending messages (that is, all enabled subqueues are empty), then the msg_receive blocks the calling thread. This unusual design of prioritized subqueues was introduced to support the following use case: a thread T_1 receives a message M_1 whose processing requires it to send a request message to another thread T_2 and wait for a reply. As per the architectural pattern, T_1 cannot make a blocking call to T_2, so it must go back to the head of its main thread loop (line 2 in Figure 1) and wait for the reply from T_2. However, while it is waiting for this reply, we would like to avoid processing new requests sent to T_1 (as this would make the implementation of T_1 more complicated). To achieve this, T_1 uses two subqueues (one for requests and one for replies), and it disables the request subqueue when it sends a request to T_2. Now, any new requests will be ignored until T_1 receives the reply from T_2 (which is delivered to the reply subqueue), at which point T_1 re-enables the request subqueue and processes the next waiting request.

3.2 Hierarchical State Machines (HSMs)

As mentioned earlier, spacecraft software (and embedded software in general) is often designed and implemented using hierarchical state machines (HSMs). HSMs can be characterized as state machines with an imposed hierarchy, allowing states to contain (sub) state machines, to an arbitrary depth. In addition, every state has optional associated *entry* and *exit* actions. When an HSM receives an event E in a state S, it finds the closest ancestor state A of S which has a transition α defined for event E to a target state T. It then computes the least common ancestor state P between A and T. It then executes the exit actions of all states (in order) along the path from S to P, then executes the action associated with the transition α, and finally executes the entry actions (in order) along the path from P to T. In the Curiosity FSW, each HSM is implemented by a thread, which receives events as *messages* sent to the thread's incoming

queue. (That is, each received message corresponds to a single event, and the message handler corresponds to the action associated with the transition.)

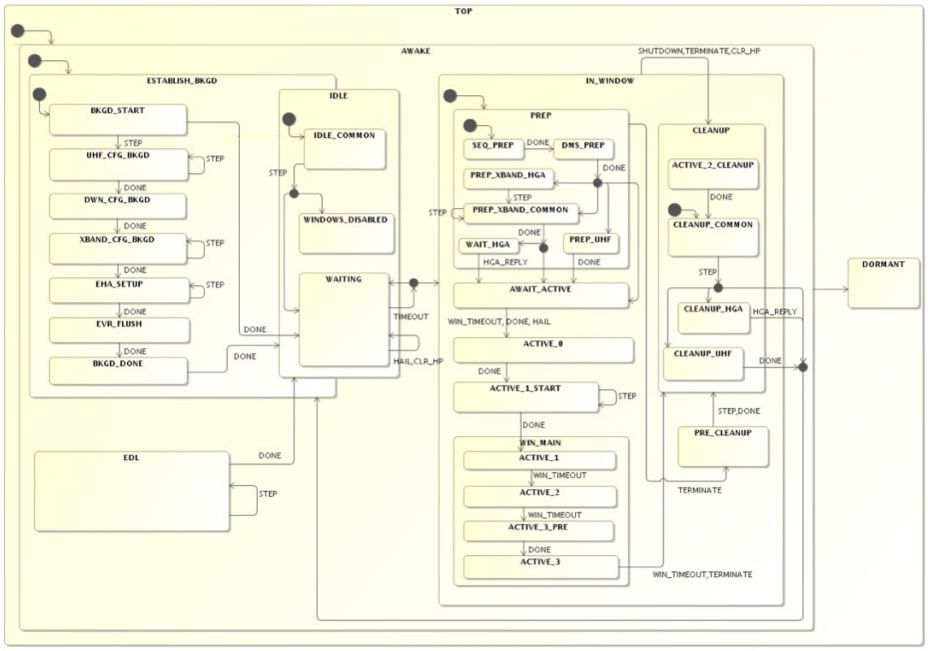

Fig. 2. Full HSM for the Communication Behavior module

Figure 2 shows a graphical view of the complete HSM for the Communications Behaviors (CBM) module that we are modeling for our case study. This informal diagram was created by the module developer during design and was not intended to depict all the details; it is only included here to show the overall complexity. For instance, none of the transitions are annotated with conditions or action code. The black circles with an incoming edge and multiple outgoing edges represent conditional transitions (where the branch conditions are not shown). The CBM module coordinates the activities needed to prepare the spacecraft for communication sessions (called *windows*) with Earth. Each window has an associated start and end time, and the set of future windows is stored in a table. Windows are added and deleted by ground operators. CBM selects the earliest window in the table and ensures that the telecommunications hardware is configured in time for the window (for instance, by ensuring that the antenna is pointed to and tracking Earth). While we have modeled the entire module HSM as part of our case study, in the interest of readability, we only discuss a small (slightly simplified) fragment of this HSM in this paper. This fragment is shown in Figure 3. As shown in the figure, there is a state `top` that contains all other states. Following usual HSM notation, the filled out black circles indicate the initial substate that is entered whenever the parent state is entered. (Thus,

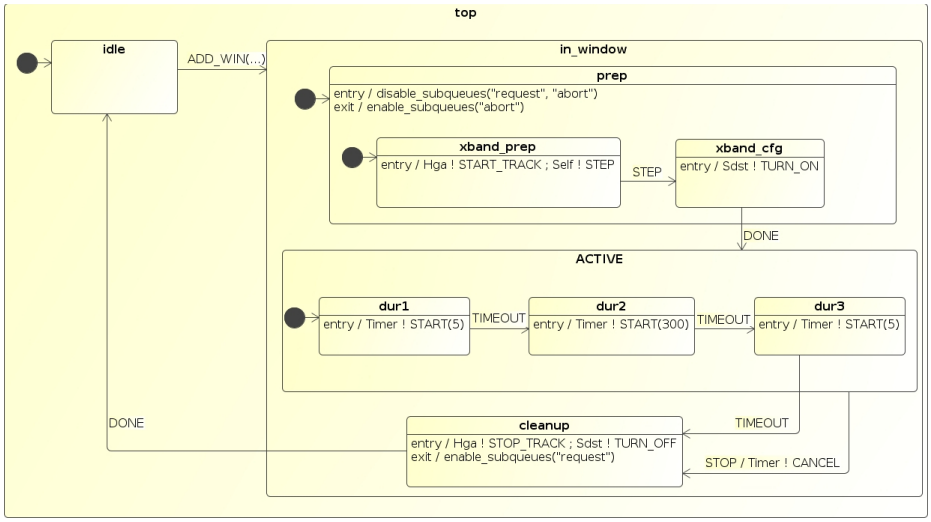

Fig. 3. Fragment of HSM for the CBM module

for instance, a transition to the in_window state causes the HSM to transition to the xband_prep substate.) Associated with each state are also two optional code fragments, the *entry* and *exit* actions. The *entry* action associated with a state is executed whenever the HSM enters that state, whereas the *exit* action is executed whenever the HSM leaves that state. Finally, the labeled arrows between states show the transitions that are caused in response to messages received by the HSM. A label has the form MESSAGE / code, which denotes that the transition is triggered when the HSM receives the specified MESSAGE. In response, the HSM transitions to the target state, and executes the (optional) code fragment. For instance, suppose the HSM is in state dur2, and it receives the STOP message. This causes the HSM to perform the following actions (in order): (1) the *exit* action for the state dur2, (2) the *exit* action for the state active, (3) the action Timer ! CANCEL, which cancels an existing timer, and (4) the *entry* action for the state cleanup.

3.3 Interactions between HSMs, Devices and Timers

The execution of an action may result in the HSM sending messages to other entities in the system, such as other threads (also possibly implemented as HSMs), to devices (for instance, powering on a radio), or to system services (for instance, the timer service, discussed below). Message transmissions are denoted as recv ! M where recv is the receiver to which the message M is sent. In our example, the receivers Hga and Sdst denote devices (the high-gain antenna and an X-Band radio, respectively), whereas the receiver Timer denotes the timer service. The receiver Self denotes the HSM itself; for instance, in state xband_prep, the entry action results in the HSM sending the START_TRACK message to

the Hga device, and the STEP message to itself; the latter in turn causes the HSM to transition from the xband_prep to the xband_cfg state, while sending the TURN_ON message to the Sdst device. (Messages to Self allow a thread to break up its processing into smaller units of computation, thereby allowing the processing to be interrupted by other, higher-priority messages.)

As explained earlier, threads can enable/disable some of their subqueues, to avoid receiving messages on those subqueues while in certain states. For the example shown in Figure 3, we assume that there are three subqueues, labeled "transition", "abort" and "request" (in decreasing priority order). The STEP, DONE and TIMEOUT messages are delivered to the "transition" subqueue, STOP messages are delivered to the "abort" subqueue, and the ADD_WIN message is delivered to the "request" subqueue. In the figure, entering the prep state causes the HSM to disable its two subqueues named "request" and"abort"; as a result, in any substate of prep, the HSM can only receive STEP, DONE or TIMEOUT messages. When it transitions to the active state, the exit action for prep re-enables the "abort" subqueue, which allows the HSM to respond also to STOP messages while in any of the dur1, dur2, dur3 substates of the active state.

The figure also illustrates the use of timers. For instance, when the HSM enters the dur1 substate, it starts a 5-second timer by sending a START(5) request to the Timer service. When the timer expires, the Timer service sends the TIMEOUT message back to the HSM, which causes it to transition to the dur2 state. Upon entering dur2, the HSM starts a 300-second timer. If the HSM receives a STOP message while in a substate of the active state, it cancels the outstanding timer by sending a CANCEL message to the Timer service, and transitions to the cleanup state[4].

3.4 Event Logs

During execution, threads generate a stream of telemetry that is periodically sent to Earth so that ground operators can assess the success of requested activities and the health of various spacecraft subsystems. A key element of the telemetry stream is an *event log* that contains a log of timestamped events that happened on the spacecraft. Event logs are used by engineers to assess if the system is behaving as designed, and are often checked (either manually or using ad-hoc scripts) to verify that the HSM behavior satisfies expected properties. As we discuss in a later subsection, in our approach, we provide a formal, declarative notation (based on temporal logic) in which properties may be expressed, and then the Daut runtime monitoring engine is used to check that the HSM behaviors satisfy these properties. Figure 4 shows fragments of event logs from two runs of the HSM. (In section 4.2, we describe how these runs are generated from test scenarios.) The number before each event denotes the time of the

[4] In the interests of readability, the simplified HSM shown here does not handle the case where a timer expires right when a CANCEL message is sent; the full HSM handles this condition gracefully.

event.[5] The log on the left shows a nominal run, where the HSM enters state dur2 at time 1090 and starts a 300-second timer, which expires at time 1390, after which the window completes nominally. The log on the right shows a run in which a STOP request is sent to the HSM at time 1380, which causes it to abort the window by canceling the timer and transitioning to the cleanup state. The figure shows how the logs differ after time 1380, showing the different system behaviors.

```
1085 : HSM_EVR_ENTER_STATE(active)
1085 : IPC_EVR_QUEUE_ENABLE(cbm,abort)
1085 : HSM_EVR_ENTER_STATE(dur1)
1085 : TIM_EVR_STARTED(1090)
1090 : TIM_EVR_FIRED(1090)
1090 : IPC_EVR_RECV(cbm,transition,TIMEOUT)
1090 : HSM_EVR_EXIT_STATE(dur1)
1090 : HSM_EVR_ENTER_STATE(dur2)
1090 : TIM_EVR_STARTED(1390)
1390 : TIM_EVR_FIRED(1390)
1390 : IPC_EVR_RECV(cbm,transition,TIMEOUT)
1390 : HSM_EVR_EXIT_STATE(dur2)
1390 : HSM_EVR_ENTER_STATE(dur3)
```

```
1085 : HSM_EVR_ENTER_STATE(active)
1085 : IPC_EVR_QUEUE_ENABLE(cbm,abort)
1085 : HSM_EVR_ENTER_STATE(dur1)
1085 : TIM_EVR_STARTED(1090)
1090 : TIM_EVR_FIRED(1090)
1090 : IPC_EVR_RECV(cbm,transition,TIMEOUT)
1090 : HSM_EVR_EXIT_STATE(dur1)
1090 : HSM_EVR_ENTER_STATE(dur2)
1090 : TIM_EVR_STARTED(1390)
1380 : IPC_EVR_RECV(cbm,abort,STOP)
1380 : HSM_EVR_EXIT_STATE(dur2)
1380 : HSM_EVR_EXIT_STATE(active)
1380 : TIM_EVR_CANCELED(1390)
1380 : HSM_EVR_ENTER_STATE(cleanup)
```

Fig. 4. Sample event logs for two runs of the HSM in Fig 3

4 Modeling and Testing the Communications HSM

4.1 The Communications HSM in iHSM

Figure 5 shows how the Communications Behavior Manager (CBM) HSM shown in Figure 3 is formalized in our iHSM notation. (In the interests of space, we show only a few states; the others follow a similar pattern.) Lines 1–5 show the definition of message types that are to be handled by the HSM. Lines 7-35 define the state machine as the class CbmHsm extending the MslHsm class, which itself extends the generic HSM class implementing our DSL. Line 8 defines the three subqueue priorities, along with the capacity of each subqueue. Line 9 defines the window table. Line 10 defines the outermost top state of the HSM. The handler for the ADD_WIN message (line 13) results in a transition to the in_window state. Lines 17–20 define the prep state, which has entry and exit actions. The entry action (line 18) results in the "request" and "abort" queues being disabled when the state is entered, whereas the exit action (line 19) results in the "abort" queue being re-enabled. (Note that the "request" queue stays disabled.) The entry action for the xband_prep state causes two messages to be sent: the message START_TRACK is sent to the Hga thread (corresponding to a request for the high-gain antenna to start tracking Earth), and then the HSM

[5] In our somewhat simplified execution model, we currently assume that entering and exiting states does not take any time; thus several such related events have the same timestamp.

sends a STEP message to itself. The thread will then receive this STEP message and execute the transition on line 23, which causes the HSM to transition to the xband_cfg state. The entry action for the xband_cfg state results in the HSM sending (line 26) a TURN_ON message to the Sdst thread. When it receives (line 28) the DONE reply from the Sdst, the HSM transitions to the active state. In the interests of space and readability, we have shown only a simplified fragment of the communications behavior HSM used on Curiosity. We have encoded the full CBM HSM in our iHSM notation. (The full HSM consists of 45 states and substates and over 130 transitions among these states.)

```
1   case object STEP extends CbmMessage("transition")
2   case object DONE extends CbmMessage("transition")
3   case object TIMEOUT extends CbmMessage("transition")
4   case object STOP extends CbmMessage("abort")
5   case class ADD_WIN(start: Int, ...) extends CbmMessage("request")
6
7   class CbmHsm extends MslHsm {
8     queues(("transition", 3), ("abort", 1), ("request", 17))
9     var table = new WindowTable()
10    object top extends state() {}
11    object idle extends state(top, true) {
12      when {
13        case ADD_WIN(..) ⇒ in_window exec { table.add(..) }
14      }
15    }
16    object in_window extends state(top) { }
17    object prep extends state(in_window, true) {
18      entry { disable_subqueues("request", "abort") }
19      exit { enable_subqueues("abort") }
20    }
21    object xband_prep extends state(prep, true) {
22      entry { Hga ! START_TRACK ; Self ! STEP }
23      when { case STEP ⇒ xband_cfg }
24    }
25    object xband_cfg extends state(prep) {
26      entry { Sdst ! TURN_ON }
27      when {
28        case DONE ⇒ active
29      }
30    }
31    object active extends state(in_window) {
32      when { case STOP ⇒ cleanup exec { Timer ! CANCEL } }
33    }
34    ...
35  }
```

Fig. 5. The CBM HSM expressed in iHSM

4.2 Extensible Test Scenarios

In order to test HSMs, we developed a notation for writing test cases. Our notation allows users to specify *test scenarios* that result in certain messages being sent to HSMs in the system at specified times. Figure 6 illustrates two such test scenarios. The notation at(100) exec Cbm ! M indicates that the message M is to be sent to the Cbm HSM at time 100. Note that, because our scenarios are Scala code, we can easily define a local variable S and specify that 3 messages be sent at specified times relative to S. The figure also illustrates how the use of Scala allows us to conveniently define new scenarios as extensions of existing scenarios using class inheritance. As shown, we define the stopTest scenario as an extension of the hgaTest scenario by specifying that an additional STOP message be sent to the HSM at time (S+510). Execution of these two scenarios results in the logs shown in Figure 4.

```
class hgaTest extends TestScenario {
  val S = 1000
  at(S)    exec Cbm ! SET_BKID("TEST")
  at(S+1) exec Cbm ! SET_MODE(NORMAL)
  at(S+2) exec Cbm ! ADD_WIN(311,...)
}
```

```
class stopTest extends hgaTest {
  at(S+510) exec Cbm ! STOP()
}
```

Fig. 6. Two sample test scenarios

4.3 Monitoring Temporal Properties

Next, we describe how to monitor HSMs using a monitoring framework that can check temporal properties. Figure 7 shows two example properties for the CBM HSM. The first property (lines 1–5) checks the state invariant that whenever the HSM is in the prep state, the "abort" subqueue is disabled. The predicate Cbm.inState(re) returns true if the HSM Cbm is in a state whose name matches the regular expression re. The second property (lines 7–13) checks that if a timer has been started, then the HSM either waits for the timer to fire, or it cancels the timer, before starting a new timer. The body of the class is an always-formula (line 9). The function always takes as argument a partial function from events to monitor states. In this case, whenever an TIM_STARTED event is observed, the monitor moves to a watch state, in which it waits for either a TIM_FIRED or a TIM_CANCELED event, but declares an error if another TIM_STARTED event is seen before.

```
1  // In state prep, the "abort" subqueue is disabled
2  class QueueCheck extends MSLMonitor {
3    invariant ("abortDisabled") {
4      Cbm.inState("prep") ==> !Cbm.isEnabled("abort")
5  }}
6
7  // Timer is not started unless previous timer has expired or been canceled
8  class TimerCheck extends MSLMonitor {
9    always {
10     case TIM_STARTED(_) ⇒ watch {
11       case TIM_FIRED(_) | TIM_CANCELED(_) ⇒ ok
12       case TIM_STARTED(_) ⇒ error("Timer restarted")
13 }}}
```

Fig. 7. Two temporal properties monitored

4.4 Derived Test Scenarios with Reactive Monitors

A key feature of our approach using the Daut framework is that monitors are written in Scala, and thus monitor evaluation can execute any Scala code, including sending of messages to the HSM being monitored. We refer to such monitors as *reactive monitors*. Figure 8 shows an example of how a reactive monitor allows us to express complex test scenarios in a compact and readable way. The InjectStop monitor looks for an event indicating that an HSM has entered the "active_1" state; and when this event is seen, it executes the code shown on line 4, which waits for 2 seconds and then sends a STOP message to the HSM. Note that unlike the stopTest scenario test in Figure 6 above, which required that a message be sent at a specified time, the use of a reactive monitor allows a message to be sent when a monitored property becomes true, which makes it easier to write test scenarios.

```
1  // 2 seconds after the HSM enters the "active_2" state, send a STOP
2  class InjectStop extends MSLMonitor {
3    always {
4      case HSM_ENTER_STATE("active_1") ⇒ after(2) { Cbm ! STOP() }
5  }}
```

Fig. 8. A reactive monitor

4.5 Checking Timing Properties

In this section, we illustrate how reactive monitors allow us to easily analyze the HSM design in order to check timing assumptions. Figure 9 shows an untimed monitor NoHgaReply that checks if a HGA_START_TRACK is followed by

a NO_HGA_REPLY. (This happens when the CBM HSM does not receive a reply from the high-gain antenna in response to a start tracking request). The figure also shows the reactive monitor InjectHgaDelay that waits for the HSM to enter the prep_xband_hga state, and then injects a delay D (which is a parameter to the monitor) before the hga_track_reply message is delivered to the HSM.

```
1  // Check if HGA tracking was started but no reply was received
2  class NoHgaReply extends MSLMonitor {
3    always {
4      case HGA_START_TRACK() ⇒ watch {
5        case NO_HGA_REPLY() ⇒ error
6    }}
7  }
8
9  // Delay reply from HGA by D seconds
10 class InjectHgaDelay(D: Int) extends MSLMonitor {
11   always {
12     case HSM_ENTER_STATE("prep_xband_hga") ⇒
13       Ipc.delay(D, "cbm", "hga_track_reply")
14 }}
```

Fig. 9. Timing monitors for the CBM case study

Next, in Figure 10, we show how we can use these monitors to check timing assumptions. The figure shows a Scala method search which takes a time range (lo, hi), and looks for the smallest value m in that range for which the provided function f throws a RuntimeException. This is achieved by iterating over the value of m (line 5) and calling f(m) in each iteration. If all iterations complete without an exception, the method returns None indicating the search was unsuccessful. However, as soon as an exception is encountered, the catch block (lines 7–9) returns with the value of m that caused the exception.

The next method, findMinHgaTimeout uses this search method to find the smallest delay d that can be injected into the system (using the InjectHgaDelay reactive monitor) that results in a violation of the property NoHgaReply. As shown in the figure, the method searches for a delay in the range 400..500 (line 13), and passes in a partial function that, given a delay value d, runs the hgaTest scenario with property monitor NoHgaReply and an instance of reactive monitor InjectHgaDelay(d). It then checks the value returned by the search method (lines 15–17). If the value returned is Some(m), it reports the value that caused the failure; else if the value is None, it reports that no failure was found. Running this search for the CBM HSM revealed that a delay of 409 seconds or longer results in a violation of the NoHgaReply property.

```
1  // Find the least value of m for which f(m) throws an exception
2  def search(lo: Int, hi: Int)(f: PartialFunction[Int,Unit]): Option[Int] = {
3    var m = lo
4    try {
5      while (m < hi) { f(m) ; m += 1 }
6      None
7    } catch {
8      case e: RuntimeException ⇒ Some(m)
9  }}
10
11 // Find the least delay that results in violation of NoHgaReply
12 def findMinHgaTimeout {
13   search(400, 500) {
14     case d ⇒ run(new hgaTest, new NoHgaReply, new InjectHgaDelay(d))
15   } match {
16     case Some(m) ⇒ println("Detected failure with value " + m)
17     case None    ⇒ println ("No failures found")
18   }
19 }
```

Fig. 10. Functions for finding smallest delay that causes NoHgaReply to fail

5 Conclusion

In this paper, we have built upon our previous work using an internal DSL in Scala for writing HSMs. We have described how this DSL is used to model systems with many HSMs, each implemented by a thread, which interact with each other and with devices and system services using asynchronous messaging. We have applied our ideas to a case study modeling a critical HSM that manages communications of the Curiosity rover with Earth. We have shown how to check that these HSMs satisfy properties written in a temporal logic, by integrating a monitoring framework (also written in Scala) that processes event logs generated by the HSMs. Finally, we have described a notation for writing high-level test specifications, which makes it convenient to write complex test cases by specifying a set of stimuli that are to be provided to the system when various constraints are satisfied. Our test specifications are expressed as Scala classes, which allows tests to be extended using inheritance, making it easy to develop multiple test variants from a baseline scenario. Our work is based on using a modern high-level programming language for modeling, testing and monitoring spacecraft software. We are working on making our test specification language more expressive by allowing more complex constraints (and then using an SMT solver to generate test instances). Work on visualizing HSMs from the Scala source code is in progress. We are also investigating more powerful verification techniques, such as model checking and theorem-proving (using the Viper framework [18]) that can be used to formally verify HSM properties.

Acknowledgments. The research performed was carried out at Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

References

1. Akka FSMs. `http://doc.akka.io/docs/akka/current/scala/fsm.html`
2. Unified Modeling Language. http://www.uml.org, accessed: 2017-06-08
3. Barringer, H., Falcone, Y., Havelund, K., Reger, G., Rydeheard, D.E.: Quantified event automata: Towards expressive and efficient runtime monitors. In: FM. pp. 68–84 (2012)
4. Barringer, H., Havelund, K.: TraceContract: A Scala DSL for trace analysis. In: Proc. of the 17th International Symposium on Formal Methods (FM'11). LNCS, vol. 6664, pp. 57–72 (2011)
5. Basin, D., Klaedtke, F., Marinovic, S., Zălinescu, E.: Monitoring of temporal first-order properties with aggregations. Formal Methods in System Design (2015), http://link.springer.com/article/10.1007/s10703-015-0222-7
6. Broy, M., Havelund, K., Kumar, R.: Towards a unified view of modeling and programming. In: Margaria, T., Steffen, B. (eds.) 7th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation, ISoLA 2016, Corfu, Greece, October 10-14. LNCS, vol. 9953. Springer (2016)
7. 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. pp. 154–164. PLDI '15, ACM, New York, NY, USA (2015), http://doi.acm.org/10.1145/2737924.2737996
8. Desai, A., Gupta, V., Jackson, E., Qadeer, S., Rajamani, S., Zufferey, D.: P: Safe asynchronous event-driven programming. In: Proceedings of PLDI '13. pp. 321–332 (2013), http://doi.acm.org/10.1145/2491956.2462184
9. Drusinsky, D.: Modeling and Verification using UML Statecharts. Elsevier (2006), iSBN-13: 978-0-7506-7949-7, 400 pages
10. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-oriented Software. Addison-Wesley, Boston, MA, USA (1995)
11. Havelund, K.: Closing the gap between specification and programming: VDM^{++} and Scala. In: Korovina, M., Voronkov, A. (eds.) HOWARD-60: Higher-Order Workshop on Automated Runtime Verification and Debugging. EasyChair Proceedings, vol. 1 (December 2011), manchester, UK.
12. Havelund, K.: Data automata in Scala. In: Proc. of the 8th International Symposium on Theoretical Aspects of Software Engineering (TASE'14) (2014)
13. Havelund, K.: Rule-based runtime verification revisited. International Journal on Software Tools for Technology Transfer 17(2), 143–170 (2015)
14. Havelund, K., Joshi, R.: Modeling and monitoring of hierarchical state machines in Scala. In preparation
15. Havelund, K., Visser, W.: Program model checking as a new trend. STTT 4(1), 8–20 (2002)
16. Kauffman, S., Havelund, K., Joshi, R.: nfer – a notation and system for inferring event stream abstractions. In: 16th Int. Conference on Runtime Verification (RV'16), Madrid, Spain. LNCS, vol. 10012, pp. 235–250 (2016)

17. Meredith, P., Jin, D., Griffith, D., Chen, F., Roşu, G.: An overview of the MOP runtime verification framework. J Software Tools for Technology Transfer pp. 1–41 (2011), http://dx.doi.org/10.1007/s10009-011-0198-6
18. Müller, P., Schwerhoff, M., Summers, A.J.: Viper: A verification infrastructure for permission-based reasoning. In: Jobstmann, B., Leino, K.R.M. (eds.) Verification, Model Checking, and Abstract Interpretation (VMCAI). LNCS, vol. 9583, pp. 41–62. Springer-Verlag (2016)
19. Samek, M.: Practical UML Statecharts in C/C++, Second Edition: Event-Driven Programming for Embedded Systems. Newnes, MA, USA, 2 edn. (2009)