Integrated Modeling and Development of Component-Based Embedded Software in Scala*

Klaus Havelund and Robert Bocchino

Jet Propulsion Laboratory, California Inst. of Technology, Pasadena, USA

Abstract. Developing embedded software requires good frameworks, models, and programming languages. The languages typically used for embedded programming (e.g., C and C++) tend to be decoupled from the models and tend to favor efficiency and low-level expressivity over safety, high-level expressivity, and ease of use. In this work, we explore the use of Scala for integrated modeling and development of embedded systems represented as sets of interconnected components. Although Scala today is not suitable for this domain, several current efforts aim to develop Scala-like embedded languages, so it is conceivable that in the future, such a language will exist. We present four internal Scala DSLs, each of which supports an aspect of embedded software development, inspired by an actual C++ framework for programming space missions. The DSLs support programming of software components, hierarchical state machines, temporal logic monitors, and rule-based test generators. The effort required to develop these DSLs has been small compared to the similar C++ effort.

1 Introduction

Embedded software development. Developing software for embedded systems (for example, robotic vehicles) poses at least three specific challenges. First, without appropriate frameworks, embedded systems programming is difficult. All embedded systems have certain similar behaviors, such as commanding, telemetry, and inter-task communication. It is both tedious and wasteful to encode these behaviors by hand in a general-purpose language for each new application. Domain-specific frameworks such as F Prime (F') [4] and cFS [5] can alleviate this problem. The framework can provide the behavior that is common to many applications, and the developers can focus on the behavior that is specific to their application.

Second, even with a good framework, there is a semantic gap between the domains of design and implementation. For example, developers may express a design in terms of components and their connections, whereas the implementation may be in terms of C++ classes and functions. To bridge this gap, one can

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

express the design in a modeling language and automatically generate code in a general-purpose language for further elaboration by the developers. F Prime uses a domain-specific modeling language named FPP (F Prime Prime, or F") for this purpose [10]. It is also possible to use a general-purpose modeling language such as SysML [36] or AADL [8]. This approach works, but it causes the model to be disconnected from the implementation, because the two are expressed in different languages, and the model does not generate a complete implementation. In particular, some hand-modification of generated code is usually required, and this makes it hard to update the model.

Third, embedded systems usually have tight timing and resource requirements compared to general applications. Therefore embedded developers are restricted in the programming languages they can use. The language must be compiled to efficient machine code, must provide low-level access to machine details, and must provide deterministic timing and scheduling. The traditional players in this space are C, C++, and Ada. More recent entries include D and Rust.

C, while groundbreaking in its day, has not advanced much since the 1980s; by the standards of modern languages, it is woefully primitive. C++ and Ada are better: they provide abstractions such as user-defined class types, and they are advancing with the times. However, these languages are incremental, sometimes awkward evolutions of decades-old designs. Further, C++, like C, has many devious behaviors that can trick even experienced and careful programmers into writing incorrect code. D improves upon C++ in many respects (e.g., cleaned-up syntax, improved type safety), but its design is strongly influenced by C++.

By contrast, Scala [30] is more recently designed from first principles. It has a clean syntax and semantics and is generally easier for programmers to understand and use than traditional embedded languages. Further, Scala's strong static type system rules out many basic errors that can lurk in even well-tested C and C++ programs. Rust is interesting because it adopts modern language design principles while being both efficient and safe. However, it relies on static analysis called "borrow checking" that is notoriously difficult to understand and use.

Overall, while both C++ and Scala can be used for both low- and high-level programming, C++ is more expressive for low-level programming (for example, it provides explicit control over the placement of objects in memory, whereas Scala does not), while Scala is more expressive for high-level programming (for example, Scala has ML-like functions and closures; in C++ functions and closures are separate concepts, and closures are syntactically awkward). Rust is somewhere in the middle.

Our work. We are exploring the use of Scala for modeling and developing embedded systems. Scala is a natural choice for modeling because it is has good high-level expressivity and good support for internal domain-specific languages (DSLs). Scala as it exists today is not suitable for embedded programming: it runs on the Java Virtual Machine, it is garbage collected, and it cannot express low-level machine interaction. However, several efforts are developing Scala-like

embedded languages; we describe some of them below. Although it will take further research and implementation effort, we believe that it is possible to develop such a language. Arguably Swift [35] is already such a language, although its current focus is on iOS app development.

In this work, we imagine that Scala can be used for embedded software, and we explore what we could do if this were true. In particular, we imagine that we can use the same language (Scala) for both modeling and development. To do this, we leverage Scala's strong support for *internal DSLs*. Internal means that the domain-specific language is expressed using only features provided by the host language. By contrast, an *external DSL* has a separate implementation from the host language, with a standalone tool or tools for parsing, analysis, and code generation. Internal DSLs have the advantage that the programmer works with only one language and can use all the tools, such as integrated development environments (IDEs), available for that language. The DSLs are directly executable without the use of external tools.

We have implemented four internal DSLs in Scala 2.13.0. Two of them are inspired by the F′ (F Prime) C++ framework [4], developed at NASA's Jet Propulsion Laboratory (JPL) for programming flight systems as collections of interacting components. F′ has e.g. been used for programming the Mars Helicopter [22]. The four DSLs support programming respectively: components and their port connections; behavior as Hierarchical State Machines (HSMs) in the individual components; runtime monitors; and rule-based test generators. The HSM DSL was previously introduced in [15, 16], and the monitor DSL was introduced in [13, 14]. This work integrates these with the component DSL and the testing DSL in a combined framework.

Threats to validity. The main threat to validity is the considerable gap between Scala and efficient flight software. There are, however, two interesting different attempts in progress to address this problem. Scala Native [31] is a version of Scala, which supports writing low-level programs that compile to machine code. Sireum Kekinian [33] supports programming in the Slang programming language, a subset of Scala 2.13 but with a different memory model. Slang runs on the JVM, natively via Graal, and can be translated to C. It is furthermore supported with contract and proof languages designed for formal verification and analyses. A secondary threat to validity is the unruly nature of internal DSLs. Although internal DSLs are easy to develop and very expressive, they do have drawbacks. Whereas external DSLs have a hard boundary, offering a limited number of options, internal DSLs have a soft boundary, and allow perhaps too many options. Furthermore, whereas external DSLs can be easily analyzed, internal DSLs are harder to analyze and visualize, requiring analysis of the entire host language[1]. Finally, internal DSLs do generally not have as succinct syntax as external DSLs.

[1] This comment concerns *shallow* DSLs as the ones we present in this paper, where the host language constructs are part of the DSL. This is in contrast to *deep* internal DSLs, where a data type is defined, the objects of which are programs in the DSL, and which are analyzable.

Paper outline. Sections 2-5 present the four DSLs for creating respectively components and their connections, hierarchical state machines, runtime monitors, and rule-based test generators. Section 6 outlines related work. Section 7 concludes the paper.

2 Components

The first Scala DSL we shall illustrate allows one to model an embedded software system as a collection of interacting components. The DSL specifically reflects the F′ framework [4] developed at JPL. F′ is a component-based flight software framework. Components can be active or passive. Each active component runs an internal thread. A component communicates with other components through ports. Communication over ports can be asynchronous (via a message placed on a queue) or synchronous (via a direct function call). A component-based system is constructed by defining the components, and subsequently defining a topology: linking them together, connecting each output port of a component with an input port of another component.

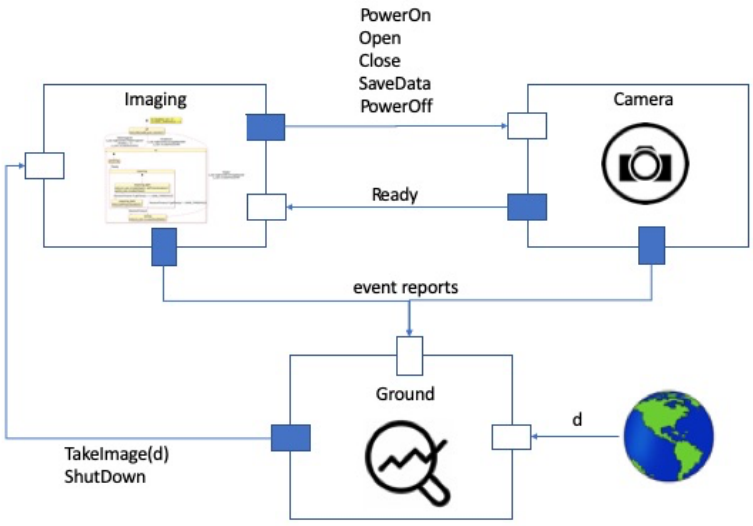

Fig. 1: The F′ Imaging, Camera, and Ground components.

We shall illustrate this DSL (and the other three DSLs) with a single example, shown in Figure 1. The example is an elaboration of the example previously presented in [15], and concerns an imaging application on board a spacecraft,

consisting of three components, Imaging, Camera, and Ground. The Imaging component is given commands from the ground to orchestrate the taking of an image by opening the shutter, controlled by the Camera component, for a certain duration. The Imaging and Camera components send event reports to the ground. Each event report reports an event that occurred on board, such as taking an image. The Imaging component is programmed as a hierarchical state machine (see Section 3), and the Ground component contains a temporal logic monitor (see Section 4).

Importing DSLs To start with, we import the four DSLs into our Scala program where we will build this system, see Figure 2.

```
import fprime._
import hsm._
import daut._
import rules._
```

Fig. 2: Importing the four DSLs.

Defining Message Types Then we define the types of messages that are sent between components. First commands, which are case classes/objects subclassing the pre-defined Command trait (a trait is an interface to one or more concrete classes), see Figure 3. A command either causes an image to be taken, or shuts down the imaging system.

```
trait Command { ... }
case class TakeImage(d: Int) extends Command
case object ShutDown extends Command
```

Fig. 3: Commands.

In this simple system, all ground commands go to the Imaging component, and they all directly extend a single trait Command. In a more realistic system,

commands to several components would be routed through a command dispatch component, and there might be separate command traits for separate components or subsystems.

We then define messages going between the Imaging and the Camera component, see Figure 4. The Imaging component can instruct the Camera component to power on or off, to open or close the shutter, and to save the image. The Camera component can report back that it has followed the various instructions and is ready for a new instruction. For simplicity, we have made the camera's power interface part of the Camera component. In a more realistic system, there could be a separate power component. Also, in a realistic system the ground may command the camera directly.

```
trait CameraControl
case object PowerOn extends CameraControl
case object PowerOff extends CameraControl
case object Open extends CameraControl
case object Close extends CameraControl
case object SaveData extends CameraControl

trait CameraStatus
case object Ready extends CameraStatus
```

Fig. 4: Messages going between Imaging and Camera.

```
trait Event extends Observation
case class  EvrTakeImage(d: Int) extends Event
case object EvrPowerOn extends Event
case object EvrPowerOff extends Event
case object EvrOpen extends Event
case object EvrClose extends Event
case object EvrImageSaved extends Event
case object EvrImageAborted extends Event
```

Fig. 5: Event reports to ground.

Finally we declare the kind of observation messages sent to the ground to report what is happening on board the spacecraft, see Figure 5. These *event reports* (Evr) report on the take-image commands sent from the Ground component to the Imaging component, the instructions being sent from the Imaging component to the Camera component, and whether the image is being saved or aborted in the Camera component.

The Imaging Component The Imaging component is shown in Figure 6. It is defined as a class sub-classing the Component class, and defines two input ports, one for receiving commands from the ground, and one for receiving messages from the camera, and two output ports, one for sending messages to the camera and one for sending observation events to the ground.

```
class Imaging extends Component {
  val i_cmd = new CommandInput
  val i_cam = new Input[CameraStatus]
  val o_cam = new Output[CameraControl]
  val o_obs = new ObsOutput

  object Machine extends HSM[Any] {...}

  object MissedEvents {...}

  override def when: PartialFunction[Any, Unit] = {
    case input ⇒
      if (!Machine(input)) MissedEvents.add(input)
  }
}
```

Fig. 6: The Imaging component.

A component can, as we have seen, have multiple input ports. They are all connected to the same single *message input queue*. That is, when a message arrives at an input port, it is stored in this message queue. Our component contains a state machine Machine, and an auxiliary data structure MissedEvents, both to be described in Section 3.

A component must define the partial function when, which the Component class calls to process a message received on the queue. In this case, it applies the state machine, and if the state machine is not interested in the message (Machine(input) returns false), the event is stored in MissedEvents, to be processed later. Because when is a partial function, one can test whether it is defined for a

certain message msg with the Boolean expression when.isDefinedAt(msg), a feature used to process messages.

The argument type is Any because the type of messages going into the message queue is the union of the messages coming into the input ports. We could statically constrain the type to be the union of the input types. However, writing out the union (say as a collection of Scala case classes) by hand would be inconvenient, since we have already provided this type information on the ports. This issue points to a limitation of using an internal DSL in this case. With an external DSL, we could use the types of the messages to infer and generate the case classes. More work is needed to find the sweet spot between internal and external DSLs meant to augment programmer productivity, in contrast to DSLs targeting non-programmers.

The Camera Component The Camera component is a stub, and is not fully shown in Figure 7. It receives messages from the Imaging component, sends messages back to the Imaging component, and otherwise just sends observation messages to the Ground component, and does not perform any real functions beyond that.

```
class Camera extends Component {
  val i_img = new Input[CameraControl]
  val o_img = new Output[CameraStatus]
  val o_obs = new ObsOutput
  ...
}
```

Fig. 7: The Camera component.

The Ground Component The Ground component issues commands to the Imaging component, and takes as input observations from the Imaging and Camera components, see Figure 8. In addition it takes inputs in the form of integers (i_int) supplied by the ground, where an integer d indicates that a command TakeImage(d) is to be sent to the Imaging component (take an image with the shutter being opened for d milliseconds).

The Ground component contains a monitor SaveOrAbort, formulated as a temporal logic property, and explained in Section 4. Each observation o of type Observation is submitted to the monitor with the call SaveOrAbort.verify(o).

```
class Ground extends Component {
  val i_int = new Input[Int]
  val i_obs = new ObsInput
  val o_cmd = new CommandOutput

  object SaveOrAbort extends Monitor[Observation] {...}

  override def when: PartialFunction[Any, Unit] = {
    case d: Int ⇒ o_cmd.invoke(TakeImage(d))
    case o: Observation ⇒ SaveOrAbort.verify(o)
  }
}
```

Fig. 8: The Ground component.

```
object Main {
  def main(args: Array[String]): Unit = {
    val imaging = new Imaging
    val camera = new Camera
    val ground = new Ground

    imaging.o_cam.connect(camera.i_img)
    imaging.o_obs.connect(ground.i_obs)
    camera.o_img.connect(imaging.i_cam)
    camera.o_obs.connect(ground.i_obs)
    ground.o_cmd.connect(imaging.i_cmd)

    ground.i_int.invoke(1000)
    ground.i_int.invoke(2000)
    ground.i_int.invoke(3000)
  }
}
```

Fig. 9: The Main program.

Connecting the Components The main program makes instances of the components and connects them, see Figure 9. As an example, the statement imaging.o_cam.connect(camera.i_img) connects the output port o_cam of the imaging component with the input port i_img of the camera component. The main program

then asks the ground component to take three images with exposure durations of respectively 1000, 2000, and 3000 milliseconds.

3 Hierarchical State Machines

In this section we present the Scala DSL for writing the Hierarchical State Machine (HSM) that controls the Imaging component. An HSM supports programming with states, superstates, entry and exit actions of states, and transitions between states. The concept corresponds to Harel's state charts [12]. The DSL and this particular HSM has previously been described in [15].

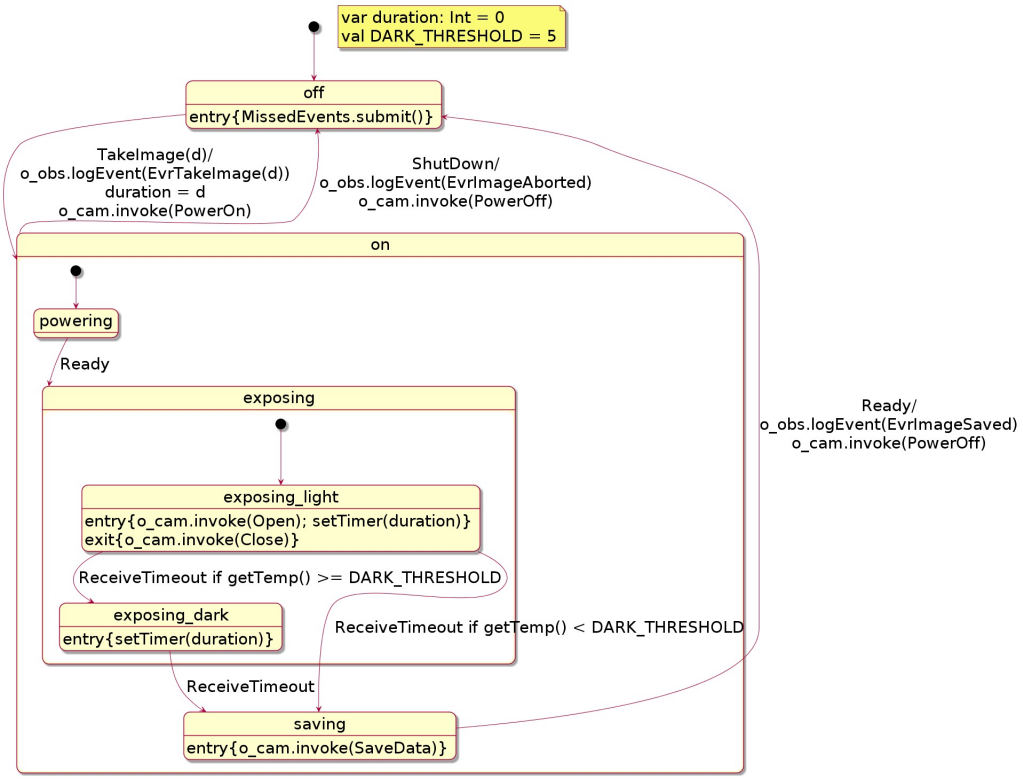

Fig. 10: The Imaging HSM visualized.

The Imaging HSM (referred to as Machine in the Imaging component) is shown graphically in Figure 10. It can be automatically generated from the textual state machine in the corresponding Scala DSL, part of which is shown in Figure 11, using PlantUML [25] and ScalaMeta [32]. The HSM can receive a TakeImage(d)

```
object Machine extends HSM[Any] {
   var duration: Int = 0
   val DARK_THRESHOLD = ...
   def getTemp(): Int = ...

   initial(off)
   ...
   object on extends state() {
     when {
       case ShutDown ⇒ off exec {
         o_obs.logEvent(EvrImageAborted)
         o_cam.invoke(PowerOff)
       }
     }
   }

   object powering extends state(on, true) {
     when { case Ready ⇒ exposing }
   }

   object exposing extends state(on)

   object exposing_light extends state(exposing, true) {
     entry { o_cam.invoke(Open); setTimer(duration) }
     exit {o_cam.invoke(Close) }
     when {
       case ReceiveTimeout ⇒ {
         if (getTemp() ≥ DARK_THRESHOLD) exposing_dark
         else saving
       }
     }
   }
  ...
 }
```

Fig. 11: The Imaging HSM.

command from ground, where `d` denotes the exposure duration. It responds to this request by sending a message to the camera to power on, and waiting until the camera is ready. It then asks the camera to open 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 so-called dark exposure with the shutter closed (but only if the ambient temperature is above

a specified threshold). A dark exposure allows determination of the noise from camera electronics, so that this can be subtracted from the acquired image. Finally, it saves the image data, and powers off the camera.

Following standard HSM notation, see Figure 10, the filled-out black circles indicate the initial substate that is entered whenever a parent state is entered. Thus, for instance, a transition to the on state ends with the HSM being 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 enters the state, whereas the exit action is executed whenever the HSM leaves the 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:

$$\langle event \rangle \textbf{ if } \langle condition \rangle \ / \ \langle action \rangle$$

which denotes that the transition is triggered when the HSM receives the specified $\langle event \rangle$ and the (optional) $\langle condition \rangle$ is true. In response, the HSM transitions to the target state, and executes the specified (optional) $\langle action \rangle$. As an example, suppose the HSM is in state exposing_light , and it receives the event ShutDown (for which a transition is defined from the parent on state). This would cause the HSM to perform the following actions (in order):

1. the exit actions for the states exposing_light , exposing (no action), and on (no action), in that order.
2. the action associated with the transition.
3. the entry action for the state off .

The entry action for the off state is MissedEvents.submit(). This re-submits an event that has been stored in the MissedEvents queue, which the HSM was not able to process in the past when in some state not prepared to process that event. Such "currently unwanted" events are stored for later re-submission. This is an artificial example, showing how one can deal with the fact, that an F′ component only has one input queue to which all input ports of the component connect. The MissedEvents data structure is defined in Figure 12, where the submit function simply re-submits the next missed event to the component's input queue.

4 Monitors

In this section we briefly present the Daut (Data automata) DSL [13, 7] for programming data parameterized temporal runtime monitors. The DSL supports writing event monitors that have either a temporal logic flavor, or a state machine flavor. We specifically program the Ground component to monitor observation events coming down from the Imaging and Camera components. The Ground component, Figure 8, contains an instantiation of the SaveOrAbort monitor, the full definition of which is shown in Figure 13.

The property states that whenever (always) an EvrTakeImage command is observed, then it is an error to observe another EvrTakeImage before either an

```
object MissedEvents {
  private var missedEvents : List[Any] = Nil

  def add(event : Any): Unit = {
    missedEvents ++= List(event)
  }

  def submit(): Unit = {
    missedEvents match {
      case Nil ⇒
      case event :: rest ⇒
        missedEvents = rest
        selfTrigger(event)
    }
  }
}
```

Fig. 12: Data structure for storing missed events.

```
object SaveOrAbort extends Monitor[Observation] {
  always {
    case EvrTakeImage(_) ⇒ hot {
      case EvrTakeImage(_) ⇒ error("not saved or aborted")
      case EvrImageSaved | EvrImageAborted ⇒ ok
    }
  }
}
```

Fig. 13: The SaveOrAbort monitor.

EvrImageSaved or EvrImageAborted is observed. This reflects the property that taking an image should end with the image being saved or aborted before another image is processed.

The DSL for writing monitors is very expressive and convenient. An earlier version (TraceContract [1]) was used throughout NASA's Lunar LADEE mission [2] for checking command sequences against flight rules expressed as monitors, before being sent to the spacecraft.

5 Rule-Based Tests

In developing applications with the F′ flight software framework [4], we have found that it is useful to define tests in terms of *rules*. A rule R consists of a *pre-condition* and an *action*. The pre-condition is a Boolean function on the state of the system, expressing whether R is *enabled* in that state, in which case it can *fire* by executing the action. The action commands the system to do something and checks for the expected behavior. Armed with a set of rules, we can write *scenarios* that use the rules to generate tests, e.g. by randomly selecting enabled rules and firing them. By writing rules and scenarios, one can quickly construct tests that exercise much more behavior than would be practical by manually writing each test. In this section we present such a rule DSL.

Testing the Imaging Component We show how to use the rule DSL to test the Imaging state machine, Figures 10 and 11, explained in Section 3. We use the standard approach to unit testing F′ components. We construct a system consisting of two components:

1. The Imaging component that we want to test.
2. The Test component. This component simulates the rest of the system. It contains a rule-based object that sends input to the Imaging component and checks the resulting output.

Note that although the approach here is used for testing one component, it can be used for testing a collection of components as well.

The Test Component The Test component, see Figure 14, has an input port for each output port of the Imaging component: i_obs for observations, and i_cam for messages the Imaging component normally sends to the Camera component. In addition, the Test component has an i_tck input port. This is used to drive the Test component from the main program: one move at a time in this particular case, to control the speed of rule firing. Correspondingly, the Test component has an output port for each input port of the Imaging component: o_cmd for commands normally coming from ground, and o_cam for messages normally coming from the Camera component.

The when method in the Test component directs incoming "tick" messages (of type Unit) from the main program to the rule engine (TestRules), causing it to fire a single randomly chosen enabled rule. Observation events, on the other hand, are forwarded to the monitor (SaveOrAbort). All other messages are ignored. The SaveOrAbort monitor, Figure 13, is the same that we previously used in the Ground component in Section 4, this time monitoring observation events emitted from the Imaging component only. It monitors that every EvrTakeImage is terminated by a EvrImageSaved or EvrImageAborted before the next EvrTakeImage is observed.

The rule-based tester, TestRules, is defined in Figure 15. It contains three rules, each sending a message to one of the input ports of the Imaging component, taking an image, shutting down the imaging component, or a ready signal

```
class Test extends Component {
  val i_tck = new Input[Unit]
  val i_obs = new ObsInput
  val i_cam = new Input[CameraControl]
  val o_cmd = new CommandOutput
  val o_cam = new Output[CameraStatus]

  object SaveOrAbort extends Monitor[Observation] {...}

  object TestRules extends Rules {...}

  override def when: PartialFunction[Any, Unit] = {
    case _: Unit ⇒ TestRules.fire()
    case o: Observation ⇒ SaveOrAbort.verify(o)
    case _ ⇒
  }
}
```

Fig. 14: The Test component.

(symbolizing that the camera component is ready), with an upper limit on how many messages of each kind can be sent. The execution strategy chosen is 'Pick', which means: whenever the fire () method is called, pick **one** enabled rule randomly and execute it. The Test component is driven by the main program with tick messages: one tick - one rule fired. This way the main program has control over how fast the rule program executes its rules.

The rules DSL offers a sub-DSL for writing rule execution *strategy algorithms*, of type Alg, using the functions shown in Figure 16. Random executes repeatedly a randomly chosen enabled rule, forever, or until no rule applies. A bounded version is provided as well. All executes the rules in sequence. An error is recorded if the pre-condition of a rule fails. Enabled executes enabled rules in sequence. If a pre-condition of a rule is not satisfied, the rule is just skipped. Until executes the rules in sequence, until a rule is reached where the pre-condition is false. First executes the first rule, from left, where the pre-condition evaluates to true. Pick executes a randomly chosen enabled rule once. Seq executes the sequence of algorithms. If executes one of two algorithms depending on a condition. While executes an algorithm as long as some condition is satisfied. Bounded executes the algorithm a bounded number of times.

The Main Test Program The MainTest program in Figure 17 instantiates the Imaging and the Test components, connects their ports, and then repeatedly 1000 times, with 100 ms in between, sends a tick message to the Test component,

```
object TestRules extends Rules {
  val MAX_IMAGES: Int = 1000
  val MAX_SHUTDOWNS: Int = 1000
  val MAX_READY: Int = 1000

  var imageCount: Int = 0
  var shutdownCount: Int = 0
  var readyCount: Int = 0

  rule("TakeImage") (imageCount < MAX_IMAGES) → {
    o_cmd.invoke((TakeImage(imageCount)))
    imageCount += 1
  }

  rule("ShutDown") (shutdownCount < MAX_SHUTDOWNS) → {
    o_cmd.invoke(ShutDown)
    shutdownCount += 1
  }

  rule("Ready") (readyCount < MAX_READY) → {
    o_cam.invoke(Ready)
    readyCount += 1
  }

  strategy(Pick())
}
```

Fig. 15: The TestRules component.

causing a rule to be fired for each tick (the repeat function is provided by the rule DSL).

Detecting a Problem in the Imaging Component Executing the above unit test does not reveal any violations of the SaveOrAbort monitor. However, setting the debugging flag to true yields output, part of which is shown in Figure 18, illustrating two firings of the rule Ready. It demonstrates a problem with the handling of *missed events*: events which arrive in the Imaging component, but which it is not able to handle in the state it is currently in. These are put in the MissedEvents queue (the contents of which is shown as: `stored: [ ... ]`). When the imaging HSM gets back to the off state, it looks for the next event in the missed-queue. If such a one exists it takes it out and re-submits it to itself. The event, however, may not match what is expected even in the off state neither, which is only TakeImage events, and hence it is put back in the missed-

```
  def Random(rules: Rule*): Alg
  def Random(max: Int, rules: Rule*): Alg
  def All(rules: Rule*): Alg
  def Enabled(rules: Rule*): Alg
  def Until(rules: Rule*): Alg
  def First(rules: Rule*): Alg
  def Pick(rules: Rule*): Alg
  def Seq(algs: Alg*): Alg
  def If(cond: ⇒ Boolean, th: Alg, el: Alg): Alg
  def While(cond: ⇒ Boolean, alg: Alg): Alg
  def Bounded(max: Int, alg: Alg): Alg
```

Fig. 16: Functions returning different test strategies.

```
object MainTest {
  def main(args: Array[String]): Unit = {
    val imaging = new Imaging
    val test = new Test

    test.o_cmd.connect(imaging.i_cmd)
    test.o_cam.connect(imaging.i_cam)
    imaging.o_cam.connect(test.i_cam)
    imaging.o_obs.connect(test.i_obs)

    repeat(1000) {
      Thread.sleep(100)
      println("=" * 80)
      test.i_tck.invoke(())
    }
  }
}
```

Fig. 17: The MainTest program.

queue. The result is that the missed-queue grows and grows with Ready and ShutDown events. This can be seen above in that the queue grows from `[Ready]` to `[Ready,Ready]`.

From a functional correctness point of view, the program works since the ShutDown and Ready events probably should be ignored in the off state anyway.

```
...
============================================================
[fpr] ? Test : ()               // Test receives a tick
[rul] executing rule Ready      // Rule Ready executes
[fpr] ! Test -[Ready]-> Imaging // Test sends Ready to Imaging
[fpr] ? Imaging : Ready         // Imaging receives Ready
Ready stored: [Ready]           // Ready unexpected and stored
============================================================
[fpr] ? Test : ()               // Test receives a tick
[rul] executing rule Ready      // Rule Ready executes
[fpr] ! Test -[Ready]-> Imaging // Test sends Ready to Imaging
[fpr] ? Imaging : Ready         // Imaging receives Ready
Ready stored: [Ready,Ready]     // Ready unexpected and stored
============================================================
...
```

Fig. 18: Debug output from running rule-based test.

The problem, however, is that the queue of missed events keeps growing. This problem fundamentally is related to our failed attempt to deal with the fact that a component only has one input queue. It requires the programmer to pay careful attention to how to deal with messages arriving that are not expected in the state the component is currently in. The problem is in particular visible in components programmed as state machines. The F′ C++ team is currently considering how to deal with this problem. Note that this kind of problem also exists in the single input queue actor model [19], but not in the CSP [20] and CCS [24] channel-based models with multiple input queues (channels) that can be selected from.

6 Related Work

Among existing programming languages, there are a few potential alternatives to C and C++ in the embedded domain. Spark Ada [34] and Real-Time Java [27] have existed for some time. More recent languages include Rust [28], Swift [35], Go [11], and D [6]. Spark Ada is interesting due to the support for formal verification. Other languages are emerging supported by formal verification. We have previously mentioned the Scala Native effort [31], and the Slang [33] programming language, based on Scala's syntax, but with a different semantics suited for embedded programming, and supported by formal verification. The PVS theorem prover [26] has been augmented with a translator from PVS to C [9], permitting writing very high-level and verifiable (executable) specifications in PVS using PVS's highly expressive type system, and obtain C's execution speed. With respect to the modeling aspect, the BIP framework [3] supports component-based modeling with components containing C code, and specifically state machines. Interaction between components can be controlled with temporal constraints. The Quantum Framework [29] supports programming with

hierarchical state machines in C and C++, and is used at JPL as target of a translation from statecharts drawn with MagicDraw [21]. Finally, an ongoing effort to design and implement a programming language explicitly supporting hierarchical state machines and monitors is described in [23]. That effort has been directly inspired by the work presented in this paper.

7 Conclusion

We developed four internal DSLs in Scala for modeling and developing embedded systems. The DSLs are inspired by the component-based C++ framework F′ developed at JPL for programming robotic vehicles. The work is part of a broader effort to explore alternatives to C and C++ for programming embedded systems. As part of this effort, we developed three non-trivial multi-threaded applications in both Rust [28] and Scala: an AI plan execution engine for the Deep Space 1 (DS-1) spacecraft, described and verified in [17]; a file transfer protocol, described and verified in [18]; and the F′ component framework, described in [4]. Rust's type checker includes the *borrow checker*, which verifies that memory operations are safe. This borrow checker is challenging to deal with. We are currently exploring features required for programming embedded systems in Scala as an alternative.

References

1. H. Barringer and K. Havelund. TraceContract: A Scala DSL for trace analysis. In *Proc. 17th Int. Symposium on Formal Methods (FM 2011)*, volume 6664 of *LNCS*, pages 57–72, Limerick, Ireland, 2011. Springer.
2. H. Barringer, K. Havelund, E. Kurklu, and R. Morris. Checking flight rules with TraceContract: Application of a Scala DSL for trace analysis. In *Scala Days 2011, Stanford University, California*, 2011.
3. BIP. http://www-verimag.imag.fr/Rigorous-Design-of-Component-Based.html.
4. R. Bocchino, T. Canham, G. Watney, L. Reder, and J. Levison. F Prime: An open-source framework for small-scale flight software systems. In *32nd Annual AIAA/USU Conference on Small Satellites, Utah State University*, 2018.
5. cFS. https://cfs.gsfc.nasa.gov.
6. D. https://dlang.org.
7. Daut on github. https://github.com/havelund/daut.
8. P. Feller and D. Gluch. *Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis and Design Language*. Addison-Wesley, 2012.
9. G. Férey and N. Shankar. Code generation using a formal model of reference counting. In S. Rayadurgam and O. Tkachuk, editors, *NASA Formal Methods (NFM 2016)*, volume 9690 of *LNCS*, pages 150–165. Springer, 2016.
10. FPP. https://github.com/fprime-community/fpp.
11. Go. https://golang.org.
12. D. Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3):231–274, 1987.
13. K. Havelund. Data automata in Scala. In *2014 Theoretical Aspects of Software Engineering Conference, TASE 2014, Changsha, China, September 1-3, 2014*, pages 1–9. IEEE Computer Society, 2014.

14. K. Havelund. Monitoring with data automata. In *Proc. of the 6th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA'14)*, volume 8803 of *LNCS*, pages 254–273. Springer, 2014.
15. K. Havelund and R. Joshi. Modeling and monitoring of hierarchical state machines in Scala. In A. B. Romanovsky and E. Troubitsyna, editors, *Software Engineering for Resilient Systems - 9th International Workshop, SERENE 2017, Geneva, Switzerland, September 4-5, 2017, Proceedings*, volume 10479 of *LNCS*, pages 21–36. Springer, 2017.
16. K. Havelund and R. Joshi. Modeling rover communication using hierarchical state machines with Scala. In *2nd International Workshop on the Timing Performance in Safety Engineering (TIPS 2017)*, volume 10489 of *LNCS*, pages 447–461. Springer, 2017.
17. K. Havelund, M. R. Lowry, and J. Penix. Formal analysis of a space-craft controller using SPIN. *IEEE Trans. Software Eng.*, 27(8):749–765, 2001.
18. K. Havelund and N. Shankar. Experiments in theorem proving and model checking for protocol verification. In M. Gaudel and J. Woodcock, editors, *FME '96: Industrial Benefit and Advances in Formal Methods, Third International Symposium of Formal Methods Europe, Co-Sponsored by IFIP WG 14.3, Oxford, UK, March 18-22, 1996, Proceedings*, volume 1051 of *LNCS*, pages 662–681. Springer, 1996.
19. C. Hewitt, P. Bishop, and R. Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, IJCAI'73, page 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc.
20. C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.
21. MagicDraw. https://www.nomagic.com/products/magicdraw.
22. Mars Helicopter. https://mars.nasa.gov/technology/helicopter.
23. B. McClelland, D. Tellier, M. Millman, K. B. Go, A. Balayan, M. J. Munje, K. Dewey, N. Ho, K. Havelund, and M. Ingham. Towards a systems programming language designed for hierarchical state machines. In *8th IEEE International Conference on Space Mission Challenges for Information Technology, (SMC-IT 2021)*. IEEE, 2021. To appear.
24. R. Milner. *A Calculus of Communicating Systems*, volume 92 of *LNCS*. Springer, 1980.
25. PlantUML. http://plantuml.com.
26. PVS. http://pvs.csl.sri.com.
27. Real-Time Java. https://en.wikipedia.org/wiki/Real_time_Java.
28. Rust. https://www.rust-lang.org.
29. M. Samek. *Practical UML Statecharts in C/C^{++}, Second Edition: Event-Driven Programming for Embedded Systems*. Newnes, MA, USA, 2 edition, 2009.
30. Scala. http://www.scala-lang.org.
31. Scala Native. https://scala-native.readthedocs.io/en/v0.3.9-docs.
32. ScalaMeta. https://scalameta.org.
33. Sireum Kekinian. https://github.com/sireum/kekinian.
34. Spark Ada 2014. http://www.spark-2014.org.
35. Swift. https://developer.apple.com/swift.
36. Systems Modeling Language (SysML). http://www.omg.org/spec/SysML/1.3.