A Flight Rule Checker for the LADEE Lunar Spacecraft

Elif Kurklu[1] and Klaus Havelund[2](✉)

[1] NASA Ames Research Center, KBR Wyle Services, Moffett Field, USA
[2] Jet Propulsion Laboratory, California Institute of Technology, Pasadena, USA
`klaus.havelund@jpl.nasa.gov`

Abstract. As part of the design of a space mission, an important part is the design of so-called *flight rules*. Flight rules express constraints on various parts and processes of the mission, that if followed, will reduce the risk of failure. One such set of flight rules constrain the format of *command sequences* regularly (e.g. daily) sent to the spacecraft to control its next near term behavior. We present a high-level view of the automated flight rule checker FRC for checking command sequences sent to NASA's LADEE Lunar mission spacecraft, used throughout its entire mission. A command sequence is in this case essentially a program (a sequence of commands) with no loops or conditionals, and it can therefore be verified with a trace analysis tool. FRC is implemented using the TRACECONTRACT runtime verification tool, an internal Scala DSL for checking event sequences against "formal specifications". The paper illustrates this untraditional use of runtime verification in a real context, with strong demands on the expressiveness and flexibility of the specification language, illustrating the advantages of an internal DSL.

1 Introduction

On September 7, 2013, NASA launched the LADEE (Lunar Atmosphere and Dust Environment Explorer) spacecraft to explore the Moon's atmosphere, specifically the occurrence of dust. The mission lasted seven months and ended on April 18, 2014, where the spacecraft was intentionally instructed to crash into the Moon. NASA Ames Research Center designed, built, and tested the spacecraft, and was responsible for its day-to-day operation. This included specifically programming of *command sequences*, which were then uploaded to the spacecraft on a daily basis. A command sequence is, as it says, a sequence of commands, with no loops or conditionals. A command is an instruction to carry out a certain task at a certain time. An obvious problem facing these day-to-day programmers was whether the command sequences were well-formed.

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

V. K. I. Pun et al. (Eds.): ICTAC 2020, LNCS 12545, pp. 3–20, 2020.
https://doi.org/10.1007/978-3-030-64276-1_1

Generally speaking, for each space mission, NASA designs *flight rules* that capture constraints that must be obeyed during the mission to reduce the risk of failures. One particular form of flight rules specifically concern the command sequences sent from ground to a spacecraft or planetary rover on a regular basis during the mission. We present in this paper the actual effort on building a flight rule checker, FRC, for programming (formalize) these flight rules in the SCALA programming language [25] using the TRACECONTRACT [3,27] API. TRACECONTRACT was originally developed in a research effort as an internal Domain-Specific Language (DSL) [14] in SCALA for runtime verification [6], supporting a notation allowing for a mixture of data parameterized state machines and temporal logic. An internal DSL is a DSL represented within the syntax of a general-purpose programming language, a stylized use of that language for a domain-specific purpose [14]. The programming language SCALA has convenient support for the definition of such internal DSLs. It is, however, fundamentally an API in the host language.

TRACECONTRACT is here used for the purpose of code analysis. The rationale for this use lies in the observation that a command sequence can be perceived as an event sequence. Using an internal DSL for such a task has the advantage that the full power of the underlying host programming language is available, which turns out to be critical in this case. This is in contrast to an external DSL, which is a small language with its own syntax, parsing, etc. Other advantages of internal DSLs compared to external DSLs include ease of development and adjustment, and use of existing tooling for the host programming language.

Missions traditionally write flight rule checkers in programming languages such as MATLAB or PYTHON. However, numerous impressive runtime verification systems have been developed over the last two decades. They all attempt to solve the difficult problem of simultaneously optimizing: Expressiveness of formalism, Elegance of properties, and Efficiency of monitoring (the three Es). Many external DSLs have been developed over time [7,8,11,13,15,21,22,24]. Most of these focus on specification elegance and efficiency. Our own external DSLs include [1,2,5,20]. Fewer internal DSLs have been developed [9,10,26]. In addition to TRACECONTRACT, we developed a rule-based internal SCALA DSL for log analysis [17]. TRACECONTRACT itself has evolved into a newer system, DAUT [12,16], which allows for better optimization wrt. efficiency. We might conclude by emphasizing that a language such as SCALA is generally well suited for modeling, as argued in [19].

Although the mission took place several years ago, we found that it was worth reporting on this effort since it represents an actual application of a runtime verification tool in a highly safety critical environment. Before the LADEE mission and before the development of FRC, an initial study of the problem was performed and documented in [4]. Due to restrictions on what can be published about a NASA mission, the presentation is generic, showing used *specification patterns*, without mentioning any mission data.

The rest of this paper is organized as follows. Section 2 outlines the problem statement, defining the concepts of commands, command sequences, and flight rules. Section 3 presents the TraceContract DSL, previously presented in [3]. Section 4 describes the overall high-level architecture of the flight rule checker Frc. Section 5 presents some of the patterns used for writing flight rules. Finally, Sect. 6 concludes the paper.

2 Command Sequences and Flight Rules

A command sequence to be uploaded to a spacecraft is typically generated by a software program referred to as a *command sequencer*. The input to the command sequencer is a high level plan, that describes a sequence of science or engineering activities to be achieved, which itself is produced either by humans or by another program, a *planner*. Command sequences are short, in the order 10s–100s commands. Figure 1 shows the generic format of such a command sequence. Each line represents a command, consisting of a calendar time (year, number of day in year, and a time stamp), the name of the command, and a list of parameters each of the form `name=value`.

2013-103-00:07:00 /Command$_1$ variable$_{11}$=value$_{11}$ ... variable$_{1n_1}$=value$_{1n_1}$
2013-103-00:07:10 /Command$_2$ variable$_{21}$=value$_{21}$ ... variable$_{2n_2}$=value$_{2n_2}$
2013-103-00:07:16 /Command$_3$ variable$_{31}$=value$_{31}$ ... variable$_{3n_3}$=value$_{3n_3}$
2013-103-00:07:17 /Command$_4$ variable$_{41}$=value$_{41}$ ... variable$_{4n_4}$=value$_{4n_4}$
2013-103-00:07:18 /Command$_5$ variable$_{51}$=value$_{51}$ ... variable$_{5n_5}$=value$_{5n_5}$
2013-103-00:07:20 /Command$_6$ variable$_{61}$=value$_{61}$ ... variable$_{6n_6}$=value$_{6n_6}$

...

Fig. 1. Format of a command sequence.

A command sequence must satisfy various flight rules. An example of such a rule pattern is the following. *A maximum of N commands can be issued per second.* That is, no more than N commands can be issued with the same time stamp. Another rule pattern is the following: *Component activation will be performed at least 1 s after application of power to the component, but no more than the upper limit for activation time.* In other words, at least one second must pass from a component has been powered on, but no more than some upper limit, till it is actually activated to perform its task.

As can be seen, a command sequence can be perceived as a sequence of events, each being a command consisting of a name, a time, and a mapping from parameter names to values. Such an event sequence can be verified against flight rules with the TraceContract tool introduced in the next Section.

3 TraceContract

This section introduces the TraceContract DSL through a complete running example and an overview of the implementation.

3.1 A Complete TraceContract Example

TraceContract is a Scala API for writing trace monitors. We also refer to it as an internal (or embedded) DSL since it is a Domain-Specific Language for writing monitors in an existing host language, in this case Scala. A trace is a sequence of events. We don't really care where the event stream comes from, whether it is emitted from a running system, or, as in this case, a sequence of commands (where a command is an event). The DSL supports a flavor of temporal logic combined with state machines, both parameterized with data to support verification of events that carry data. TraceContract can be used for monitoring event streams online (as they are generated) as well as offline (e.g. stored in files as in this case).

We shall illustrate TraceContract with a single complete executable example. Due to lack of space, the reader is referred to [3,27] for a more complete exposition. Figure 2 shows a Scala program using the TraceContract DSL. The line numbers below refer to this figure. It first defines the type of events we are monitoring (line 3), namely commands, a **case** class (allowing pattern matching against objects of the class). The program contains two monitors, DistinctTimes (lines 5–11) and ActivateTimely (lines 13–24), each extending the TraceContract Monitor class, parameterized with the event type.

The DistinctTimes monitor (lines 5–11) checks that commands are issued with strictly increasing stamps. It waits for any Cmd object to be submitted to it (line 7). The require function (line 6) takes as argument a partial function, enclosed in curly brackets (lines 6–10), and defined by **case** statements, in this case one, ready to fire when an event matches one of the case patterns, in which case the "transition" is taken. In this case, when a command arrives, matching the pattern Cmd(_,time1) (line 7), the monitor transitions to the inner anonymous *cold* state, where it waits for the next command to be submitted (line 8), which does not need to occur since it is a cold state (a final state). If a second command is submitted, however, it is asserted that the time stamp of the second command is bigger than that of the first. The require function works like the temporal logic always-operator ($\Box$).

The ActivateTimely monitor (lines 13–24) checks that if a power command is observed then a subsequent activate command must be observed within 30 time units. The activate command must occur, which is modeled by a hot state (lines 21–23). For illustration purposes, we have defined this rule a bit more long-winded than needed by defining a function activateTimely(powerTime: Int) (lines 20–23), which when called (line 17) returns the hot state. This style illustrates how to write state machines using Scala functions. As can be seen, states can be parameterized with data, and can be written in a way resembling temporal logic (lines 7–8) or state machines (lines 17 + 20–23), or even a mixture.

There are different kinds of states inspired by temporal logic operators [23], including the *cold* (line 7) and *hot* states (line 21) as we have seen. These states differ in (1) how they react to an event that does not match any transition (stay in the state, fail, or drop the state), (2) how they react to an event that matches a transition to another state (keep staying in the source state or leave it), and

```
 1  import tracecontract._
 2
 3  case class Cmd(name: String, time: Int)
 4
 5  class DistinctTimes extends Monitor[Cmd] {
 6    require {
 7      case Cmd(_,time1) ⇒ state {
 8        case Cmd(_,time2) ⇒ time2 > time1
 9      }
10    }
11  }
12
13  class ActivateTimely extends Monitor[Cmd] {
14    val upperBound : Int = 30
15
16    require {
17      case Cmd("power", time) ⇒ activateTimely(time)
18    }
19
20    def activateTimely(powerTime: Int): Formula =
21      hot {
22        case Cmd("activate", time) ⇒ time − powerTime < upperBound
23      }
24  }
25
26  class Monitors extends Monitor[Cmd] {
27    monitor(new DistinctTimes, new ActivateTimely)
28  }
29
30  object Run {
31    def main(args: Array[String]) {
32      val monitors = new Monitors
33      val trace = List(Cmd("power",100), Cmd("transmit", 130), Cmd("activate", 150))
34      monitors.verify(trace)
35    }
36  }
```

Fig. 2. A complete TraceContract example.

(3) how they evaluate at the end of the trace (true or false). Beyond the hot state having the three attributes: (stay if no match, leave if match, false at end), there are the following states: state (stay, leave, true), strong (fail, leave, false), weak (fail, leave, true), drop (drop, leave, true), and always (stay, stay, true). A require(f) call (lines 6 and 16) creates and stores an always(f) state. Un-named (anonymous) states are allowed, as shown in the DistinctTimes monitor (lines 7–9), thereby relieving the user from naming intermediate states in a progression of transitions, as shown in the ActivateTimely monitor (lines 17 + 20–23). This gives a flavor of temporal logic. The target of a transition can be a conjunction of

states as well as a disjunction, corresponding to alternating automata (although this is not used in this work).

Since these monitors are classes, we can write Scala code anywhere Scala allows it, e.g. declaring constants, variables, methods, etc., and use these in the formulas. Note that although we in our example associate one property with each monitor, it is possible to define several properties in a monitor. It is also possible to combine monitors for purely organizational purposes, with no change in semantics. E.g. in our example, we define the monitor Monitors (lines 26–28), the only purpose of which is to group the two other monitors into one. In the main method (lines 31–35) we instantiate this parent monitor and feed it a trace of three commands. This event sequence in fact violates the ActivateTimely monitor, causing the following error message to be issued (slightly shortened), showing an error trace of relevant events:

```
Total number of reports: 1
Monitor Monitors.ActivateTimely property violations: 1

Monitor: Monitors.ActivateTimely Property violated
Violating event number 3: Cmd(activate,150)
Trace:
  1=Cmd(power,100)
  3=Cmd(activate,150)
```

3.2 The TraceContract Implementation

In this section we shall very briefly give an idea of how TraceContract is implemented[1], see [3,27] for more details. A monitor is parameterized with an event type Event, and monitors a collection of formulas over such events, each kind of formula sub-classing the class Formula:

```
class Monitor[Event] {
  var formulas: List[Formula] = List()
  ...
}

abstract class Formula {
  def apply(event: Event): Formula
  def reduce(): Formula = this
  def and(that: Formula): Formula = And(this, that).reduce()
  def or(that: Formula): Formula = Or(this, that).reduce()
  ...
}
```

[1] Note that we have made some simplifications for ease of presentation.

The `apply` function allows one to apply a formula f to an event e as follows: $f(e)$, resulting in a new formula. For each new event, each formula is evaluated by applying it to the event, to become a new formula. The function is defined as abstract and is overridden by the different subclasses of `Formula` corresponding to the various kinds of formulas. The `reduce` function, also specific for each kind of formula (by default being the identity), will simplify a formula by rewriting it according to the classical reduction axioms of propositional logic, e.g. $\mathit{true} \wedge f = f$.

In the monitors above, the function require(f) takes as argument a partial function f of type PartialFunction[Event,Formula], which represents a transition function, and creates and stores a formula object always(f) to be monitored, where always(f) is a formula of the class below.

```
type Block = PartialFunction[Event, Formula]

def require(block: Block) {
  formulas ++= List(always(block))
}

case class always(block: Block) extends Formula {
  override def apply(event: Event): Formula =
    if (block.isDefinedAt(event)) And(block(event),this) else this
}
```

Similarly for cold and hot states, we have the following definitions, which are identical (the difference between cold and hot states shows at the end of a trace, as explained below):

```
case class state(block: Block) extends Formula {
  override def apply(event: Event): Formula =
    if (block.isDefinedAt(event)) block(event) else this
}

case class hot(block: Block) extends Formula {
  override def apply(event: Event): Formula =
    if (block.isDefinedAt(event)) block(event) else this
}
```

Other states follow the same pattern, but vary in the definition of the apply function. We saw in Fig. 2 Boolean expressions, such as time2 > time1 occur as formulas on the right-hand side of **case** transitions. This is permitted by implicit functions (applied by the compiler) lifting these values to formulas. E.g., the functions below are applied by the compiler when a value of the argument type occurs in a place where the return type (Formula) is expected. Note that True and False are TraceContract Formula objects. The second implicit function allows code with side-effects, not returning a value (of type Unit), as the result of transitions.

```
implicit def convBoolean2Formula(cond: Boolean): Formula = if (cond) True else False
implicit def convUnitToFormula(unit: Unit): Formula = True
```

Finally, a monitor offers a method for verifying a single event, a method for ending verification (verifying that all active states are cold, e.g. no hot states), and a method for verifying an entire trace, which calls the previous two methods.

```
def verify(event: Event): Unit {...}

def end(): Unit {
  for (formula ← formulas) {
    if (!end(formula)) reportError(formula)
  }
}

def end(formula: Formula): Boolean = {
  formula match {
    case always(_) ⇒ true
    case state(_) ⇒ true
    case hot(_) ⇒ false
    ...
  }
}

def verify(trace: Trace): MonitorResult[Event] = {
  for (event ← trace) verify(event)
  end()
  getMonitorResult
}
```

TraceContract also offers Linear Temporal Logic (Ltl) [23] operators, and rule-based operators for recording facts, useful for checking past time properties. These features were not used in Frc.

TraceContract was developed with expressiveness in focus rather than efficiency. However, for the small command sequences of up to 100s of commands, efficiency is not an issue. As previously mentioned, TraceContract evolved into a newer system, Daut [12,16]. In [18] is described a performance evaluation of Daut, processing a log of 218 million events. Daut is here able to process between 100,000+ - 400,000+ events per second, depending on the property being verified.

4 Architecture of Flight Rule Checker

Figure 3 shows the architecture of the flight rule checker. It takes as input a command sequence, an initial state of the spacecraft (the expected current state), and an identification of which rules to verify. The catalog of rules programmed in TraceContract is provided as a Scala library. It produces a verification report as result. Which rules to execute is selected by a mission operator in a flight rule editor, a GUI showing all LADEE flight rules, see Fig. 4. The initial state is also specified in the GUI. Descriptions of the flight rules are stored as an Xml file, see Fig. 5, grouped into sub-systems. Each rule has an id, a descriptive title, and a class name indicating which is the corresponding TraceContract monitor.

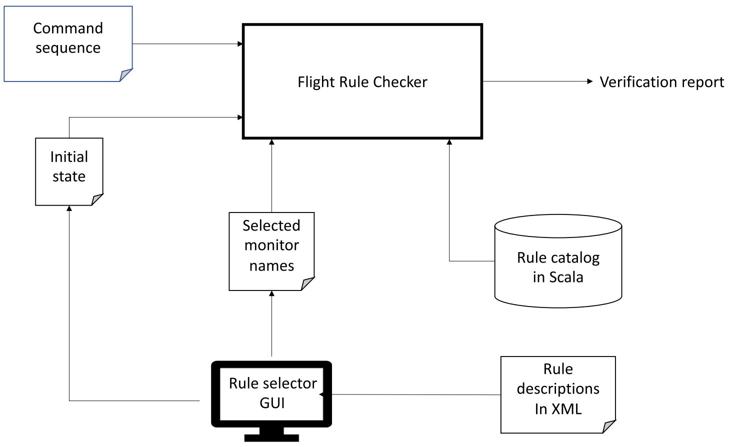
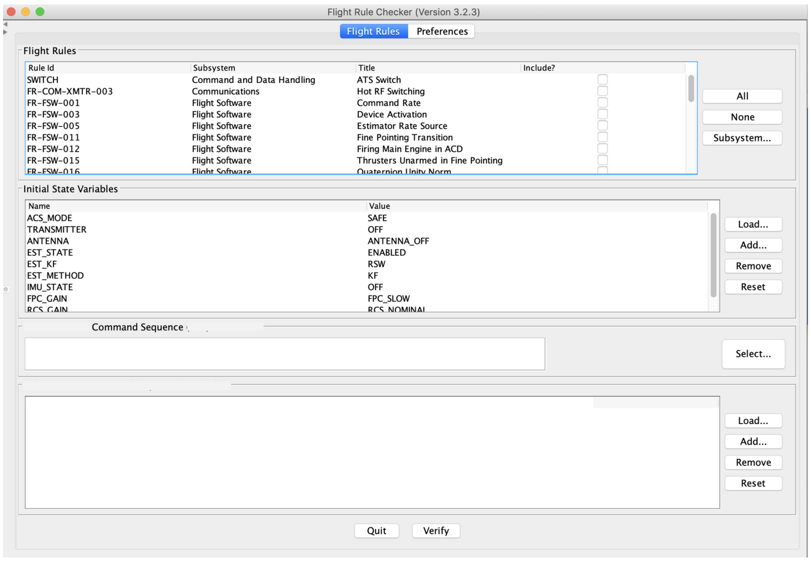


Fig. 3. The flight rule checker architecture.

Figure 6 shows the Scala **case** class FRCCommand, instantiations of which will represent the commands found in the command sequence file (Fig. 1). The figure also shows the flight rule checker's verification function, which takes as arguments a list of names of monitor classes selected, representing the choices made in the GUI in Fig. 4; the location of the command sequence file; and other arguments. The function first locates and instantiates the Scala monitor classes according to their names (using reflection), adding each instance as a sub-monitor to the ruleVerifier parent monitor. It then builds the command sequence, verifies the command sequence, and finally produces an error report based on the data stored in the ruleVerifier monitor.



Flight Rules

Rule Id	Subsystem	Title	Include?
SWITCH	Command and Data Handling	ATS Switch	☐
FR-COM-XMTR-003	Communications	Hot RF Switching	☐
FR-FSW-001	Flight Software	Command Rate	☐
FR-FSW-003	Flight Software	Device Activation	☐
FR-FSW-005	Flight Software	Estimator Rate Source	☐
FR-FSW-011	Flight Software	Fine Pointing Transition	☐
FR-FSW-012	Flight Software	Firing Main Engine in ACD	☐
FR-FSW-015	Flight Software	Thrusters Unarmed in Fine Pointing	☐
FR-FSW-016	Flight Software	Quaternion Unity Norm	☐

Initial State Variables

Name	Value
ACS_MODE	SAFE
TRANSMITTER	OFF
ANTENNA	ANTENNA_OFF
EST_STATE	ENABLED
EST_KF	RSW
EST_METHOD	KF
IMU_STATE	OFF
FPC_GAIN	FPC_SLOW
RCS_GAIN	RCS_NOMINAL

Fig. 4. The flight rule checker GUI for selecting rules.

```
<?xml version='1.0' encoding='UTF−8'?>
<frc>
  <rules>
    <subsystem name="System_1">
        <rule id="ID_1" title="Title_1" class="Monitor_1"></rule>
        <rule id="ID_2" title="Title_2" class="Monitor_2"></rule>
        <rule id="ID_3" title="Title_3" class="Monitor_3"></rule>
      ...
     </subsystem>

     <subsystem name="System_2">
       ...
     </subsystem>
     ...
  </rules>
</frc>
```

Fig. 5. Flight rule catalog in XML format, organized into sub systems.

```
case class FRCCommand(name: String, time: Calendar, params: Map[String, String]) {..}

object FRCService extends ... {
  def verify(rules: List[String], cmdSequence: File, ...): Report = {

    // build monitor containing a sub−monitor for each rule:
    val ruleVerifier = new Monitor[FRCCommand]
    rules.foreach(key ⇒ {
      val ruleInstance = FlightRuleCatalog.getRuleInstance(key)
      ruleVerifier.monitor(ruleInstance)
    })

    // create command sequence:
    val commands: List[FRCCommand] = ...

    // verify command sequence:
    ruleVerifier.verify(commands)

    // create and return report:
    generateReport(ruleVerifier)
  }
  ...
}
```

Fig. 6. The flight rule checker.

5 Flight Rules

A total of 37 flight rules were programmed, out of which 31 were actively used. We will here present the patterns of six of the most generic ones, illustrating different aspects. The rules concern the Flight Software (FSW) subsystem, specifically expressing constraints on how the commands should be sent, etc. Each rule is programmed as a class extending the TraceContract Monitor class.

5.1 Command Rate

Figure 7 shows the '*Command Rate*' monitor. It verifies that no more than MAX = N commands are issued per second, that is: with the same time stamp, for some constant N. For each command, let's call it the *initiator command*, the monitor calls the function count, which itself returns a formula, and which recursively consumes commands until either a command with a bigger time stamp is observed (ok causes the monitor to end tracking this particular initiator command), or the limit of N is reached, in which case an error is issued. Note that the require function initiates this tracking for every observed command. A new counting state machine is created for each command, tracking commands with the same time stamp.

```
// The maximum rate of command issuance is N issued commands per second.

class CommandRate extends Monitor[FRCCommand] {
  val MAX = N

  require {
    case FRCCommand(_, time, _) ⇒ count(time)
  }

  def count(time: Calendar, nr: Int = 1): Formula =
    state {
      case FRCCommand(_, time2, _) ⇒ {
        if (time2 > time) ok else if (nr == MAX) error else count(time, nr + 1)
      }
    }
}
```

Fig. 7. Command Rate monitor.

5.2 Device Activation

Figure 8 shows the '*Device Activation*' monitor. It verifies that if a device is powered on at time t, it must be eventually activated within the time interval $[t + N_1, t + N_2]$ for two positive natural numbers $N_1 < N_2$, stored in the constants activateMin and activateMax. The monitor defines a map from device *power on* commands to their corresponding *activation* commands, which is used in the formula to match them up. There are K such map entries for some not small K. This mapping is manually created and cannot be calculated. The formula itself shows use of SCALA's conditioned **case** statements (a match requires the condition after **if** to hold), and the hot state to indicate that activation must eventually occur. The pattern 'activationCommand' (a variable name in single quotes) means: match the value of the variable activationCommand. The formula states that if we observe an FRCCommand("power", t_1, *params*) command, where *params*("state") = "on" and *params*("device") = d (*power device* d *on*) and deviceMap(d) = a (*d's activation command is* a), then we want to eventually see an FRCCommand(a,t_2,...) such that $t_2 \in [t_1 + N_1, t_1 + N_2]$. An alternative would have been to define a monitor for each pair of *power on* and *activate* commands. Since there are K such pairs this would become heavy handed. This illustrates the advantage of an internal DSL, where maps are available as a data structure.

```
// Component activation will be performed at least $N_1$ seconds
// after application of power to the component, but no more than $N_2$ seconds.

class DeviceActivation extends Monitor[FRCCommand] {
  val activateMin = $N_1$
  val activateMax = $N_2$

  val deviceMap: Map[String, String] = Map(
     "device$_1$_pwr" → "name$_1$_activate",
     "device$_2$_pwr" → "name$_2$_activate",
     "device$_3$_pwr" → "name$_3$_activate",
     ...
     "device$_K$_pwr" → "name$_K$_activate"
  )

  require {
    case FRCCommand("power", powerOnTime, params)
      if deviceMap.keys.toList.contains(params.get("device").get) &&
        params.get("state").get.equals("on") ⇒
           val device = params.get("device").get
           val activationCommand = deviceMap.get(device).get
           hot {
             case FRCCommand(`activationCommand`, activateTime, _) ⇒
                val timeDiff = activateTime.toSeconds − powerOnTime.toSeconds
                timeDiff ≥ activateMin && timeDiff ≤ activateMax
           }
  }
}
```

Fig. 8. Device Activation monitor.

5.3 Time Granularity

Figure 9 shows the '*Time granularity*' monitor. It verifies that the millisecond part of a flight command is 0. That is, the smallest time granularity allowed is seconds. This property shows a simple check on a command argument. It also shows the classification of a monitor as a warning rather than an error if violated.

```
// No stored command sequence shall include commands or command sequences whose
// successful execution depends on command time granularity of less than 1 second.

class TimeGranularity extends Monitor[FRCCommand](Severity.WARNING) {
  require {
    case FRCCommand(_, cmd_time, _) ⇒
       cmd_time.get(Calendar.MILLISECOND) == 0
  }
}
```

Fig. 9. Time granularity monitor.

```
// The y_value will not be changed while the x_mode is M.

class ValueChange extends Monitor[FRCCommand] {
  var xMode = Config.getVarValue("x_mode")
  var yValue = Config.getVarValue("y_value")

  val specialMode = M

  require {
    case FRCCommand("set_x_mode", _, params) ⇒
      xMode = params.get("mode").get
    case FRCCommand("set_y_value", _, params) ⇒ {
      val yValueNew = params.get("value").get
      if (xMode.equals(specialMode) && !yValueNew.equals(yValue) & ...) {
        error
      } else {
        yValue = yValueNew
      }
    }
  }
}
```

Fig. 10. Value Change monitor.

5.4 Value Change

Figure 10 shows the '*Value Change*' monitor. It verifies that some value y will not change while the mode of some component x is M. The monitor is not temporal, but illustrates the use of class variables, xMode for holding the current mode of x, and yValue for holding the current value of y. "set_x_mode" commands change the x mode, and "set_y_value" commands change the value of y.

5.5 Unsafe Activation

Figure 11 shows the '*Unsafe Activation*' property. It verifies that some feature y is not being activated, while some other feature x is being disabled (disabling has a duration). The first outer **case** treats the situation where first an x disabling command ("disable_x") is observed, followed by a y activating command ("activate_y"). In this case it is checked that the time of y activation does not occur within the duration of the x disabling. The second outer **case** treats the situation where we first observe a y activating command and subsequently an x disabling command. y activation will happen normally after x disabling, but if it happens at the same time it could occur before in the command sequence. This monitor shows this more complicated timing constraint.

```
// Feature y will not be activated while feature x is being disabled.

class UnsafeActivation extends Monitor[FRCCommand] {
  require {
    case FRCCommand("disable_x", disableStart, disableParams) ⇒
      state {
        case FRCCommand("activate_y", activateTime, _)
          if activateTime ≥ disableStart &&
            activateTime.toSeconds − disableStart.toSeconds ≤
              disableParams.get("duration").get.toInt ⇒
                error
      }
    case FRCCommand("activate_y", activateTime, _) ⇒
      state {
        case FRCCommand("dispable_x", disableStart, disableParams)
          if disableStart==activateTime ||
            (disableStart.toSeconds + disableParams.get("duration").get.toInt
              == activateTime.toSeconds) ⇒
                error
      }
  }
}
```

Fig. 11. Unsafe Activation monitor.

5.6 Mathematical Constraint

Figure 12 shows the '*Mathematical Constraint*' property. It verifies that for a selection of commands $\mathsf{command}_1 \ldots \mathsf{command}_k$, the constraint W is satisfied on a function F of parameter variables $\mathsf{x}_1 \ldots \mathsf{x}_n$. That is, it verifies that $W(F(\mathsf{x}_1,\ldots,\mathsf{x}_n))$ holds. The formula is not temporal and mainly shows validation of command arguments using a function for performing a non-trivial mathematical computation.

```
// Commands command_1 ... command_k must satisfy a non−trivial mathematical
// well−formedness constraint W on a function F of their arguments.

class MathematicalConstraint extends Monitor[FRCCommand] {
  def wellformed(command: FRCCommand): Boolean = {
    val v_1 = command.valueOf("x_1").get.toDouble
    ...
    val v_n = command.valueOf("x_n").get.toDouble
    val value =F(v_1,...,v_n) // non−trivial mathematical computation
    return W(value)
  }

  require {
    case cmd @ FRCCommand("command_1", _, _) if (!wellformed(cmd)) ⇒ error
    ...
    case cmd @ FRCCommand("command_k", _, _) if (!wellformed(cmd)) ⇒ error
  }
}
```

Fig. 12. Mathematical Constraint monitor.

6 Conclusion

The manual translation of rules expressed in English to monitors in the TraceContract DSL was performed by the first author. The biggest challenges were finding out which command parameters corresponded to those mentioned in the flight rules expressed in natural language, and finding programming patterns in the DSL that would work. Coding errors in Scala was not an issue. We have some testimony to the value of the developed framework, as the person who was responsible for command sequencing and verification back then wrote (in response to the question of how often Frc found errors in command sequences): "*I would actually say it was often, perhaps every other command sequencing cycle.*".

The effort has shown the use of a runtime verification tool, TraceContract, for code analysis. The tool was largely not changed for the purpose, and hence provided all needed features. We attribute this in part to the fact that it was an internal DSL, as opposed to an external DSL. In addition it was reliable with no bugs reported. Many real-life log analysis problems in practice are performed with programming languages, and the reason is in part the attractiveness of Turing completeness. However, internal DSLs do have drawbacks, specifically the difficulty of analyzing "specifications" since they effectively are programs. It is always possible to design an external DSL for a particular problem, also for this application, which more easily can be subject to analysis. However, an external DSL may fit one problem but not another. An internal DSL can be more flexible.

References

1. 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). https://doi.org/10.1007/978-3-540-24622-0_5
2. Barringer, H., Groce, A., Havelund, K., Smith, M.: Formal analysis of log files. J. Aerospace Comput. Inf. Commun. **7**(11), 365–390 (2010)
3. 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). https://doi.org/10.1007/978-3-642-21437-0_7
4. Barringer, H., Havelund, K., Kurklu, E., Morris, R.: Checking flight rules with TraceContract: application of a Scala DSL for trace analysis. In: Scala Days 2011, Stanford University, California (2011)
5. Barringer, H., Rydeheard, D., Havelund, K.: Rule systems for run-time monitoring: from Eagle to RuleR. In: Sokolsky, O., Taşıran, S. (eds.) RV 2007. LNCS, vol. 4839, pp. 111–125. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-77395-5_10
6. Bartocci, E., Falcone, Y., Francalanza, A., Reger, G.: Introduction to runtime verification. In: Bartocci, E., Falcone, Y. (eds.) Lectures on Runtime Verification. LNCS, vol. 10457, pp. 1–33. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-75632-5_1
7. Basin, D.A., Klaedtke, F., Marinovic, S., Zălinescu, E.: Monitoring of temporal first-order properties with aggregations. Formal Methods Syst. Des. **46**(3), 262–285 (2015)
8. Bauer, A., Küster, J.-C., Vegliach, G.: From propositional to first-order monitoring. In: Legay, A., Bensalem, S. (eds.) RV 2013. LNCS, vol. 8174, pp. 59–75. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40787-1_4
9. Bodden, E.: MOPBox: a library approach to runtime verification. In: Khurshid, S., Sen, K. (eds.) RV 2011. LNCS, vol. 7186, pp. 365–369. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-29860-8_28
10. Colombo, C., Pace, G.J., Schneider, G.: LARVA – safer monitoring of real-time Java programs (tool paper). In: SEFM 2009, pp. 33–37. IEEE (2009)
11. D'Angelo, B., et al.: LOLA: runtime monitoring of synchronous systems. In: TIME 2005, pp. 166–174. IEEE (2005)
12. Daut on github. https://github.com/havelund/daut
13. Decker, N., Leucker, M., Thoma, D.: Monitoring modulo theories. Int. J. Software Tools Technol. Transfer **18**(2), 205–225 (2016)
14. Fowler, M., Parsons, R.: Domain-Specific Languages. Addison-Wesley, Reading (2010)
15. Hallé, S., Villemaire, R.: Runtime enforcement of web service message contracts with data. IEEE Trans. Serv. Comput. **5**(2), 192–206 (2012)
16. Havelund, K.: Data automata in Scala. In: TASE 2014, pp. 1–9. IEEE (2014)
17. Havelund, K.: Rule-based runtime verification revisited. Int. J. Software Tools Technol. Transfer **17**(2), 143–170 (2015)
18. Havelund, K., Holzmann, G.: A programming approach to event monitoring. In: Rozier, K. (ed.) Formal Methods for Aerospace Engineering, Progress in Computer Science and Applied Logic. Springer (2021). Draft version, in preparation, to appear
19. Havelund, K., Joshi, R.: Modeling with Scala. In: Margaria, T., Steffen, B. (eds.) ISoLA 2018. LNCS, vol. 11244, pp. 184–205. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-03418-4_12

20. Havelund, K., Peled, D.: Runtime verification: from propositional to first-order temporal logic. In: Colombo, C., Leucker, M. (eds.) RV 2018. LNCS, vol. 11237, pp. 90–112. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-03769-7_7
21. Kim, M., Kannan, S., Lee, I., Sokolsky, O.: Java-MaC: a run-time assurance tool for Java. In: RV 2001, ENTCS, vol. 55, no. 2. Elsevier (2001)
22. Meredith, P.O., Jin, D., Griffith, D., Chen, F., Roşu, G.: An overview of the MOP runtime verification framework. Int. J. Software Tools Technol. Transfer **14**, 249–289 (2011)
23. Pnueli, A.: The temporal logic of programs. In: SFCS 1977, pp. 46–57. IEEE Computer Society (1977)
24. Reger, G., Cruz, H.C., Rydeheard, D.: MarQ: monitoring at runtime with QEA. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 596–610. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46681-0_55
25. Scala. http://www.scala-lang.org
26. Stolz, V., Huch, F.: Runtime verification of concurrent Haskell programs. Electr. Notes Theor. Comput. Sci. **113**, 201–216 (2005)
27. TraceContract on github. https://github.com/havelund/tracecontract