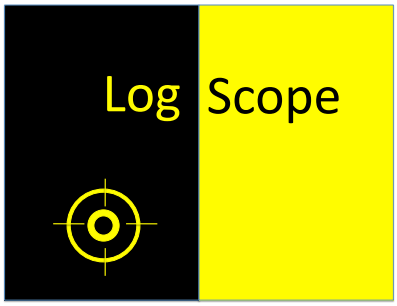


LogScope
A Tool for Specification-based Event Analysis

Klaus Havelund
klaus.havelund@jpl.nasa.gov

Laboratory for Reliable Software
Jet Propulsion Laboratory
California Institute of Technology
California, USA

May 1, 2021

Contents

1 Introduction

This document explains how to use the LOGSCOPE event analysis tool. It can be used for analysis of telemetry streams, as e.g. emitted from the HPSC software. LOGSCOPE solves the following problem:

> *Given a sequence of events emitted from a software application, e.g. a telemetry stream from a spacecraft or rover, how can we assure that the event stream satisfies a set of properties?*

LOGSCOPE allows to formalize the *properties* (requirements) as *monitors* in the formal specification language SCOPE. LOGSCOPE takes as input a formal SCOPE specification, and a sequence of events, one by one, and reports on any discrepancies between the specification and the event sequence. Violations are by default printed on standard output. The **frontend**, written in the SCALA programming language, parses a specification and translates it to a C++ module. The **backend**, the monitoring engine itself, written in C++, imports the C++ module generated by the frontend, and monitors a sequence of events emitted to it. LOGSCOPE can be used for online monitoring, monitoring events as they are generated by a running application in C or C++; or it can be used for offline monitoring, analyzing logs produced by an application at an earlier time.

The specification language is a mix of a state machine language and a rule-based language. An earlier version of LOGSCOPE for offline monitoring of logs was developed in Python to specifically assist in testing JPL's Mars Science Laboratory (MSL) flight software [2, 8]. LOGSCOPE is, however, very generic in nature and can in principle be applied to any logging software application. In most cases there will be a need to convert events generated by the software to the event format of LOGSCOPE. This is, however, a straightforward task.

LOGSCOPE is influenced by previous work in this field, including specifically the following works: RULER [1], a rule-based framework, and the later LOGFIRE [6], another rule-based framework based on the RETE algorithm, as a SCALA library, see also the github repository [7]. A PYTHON version of a variant of LOGSCOPE is described in [2], see also the github repository [8]. Other relevant work includes the RMOR monitoring framework for C [4], as well as the DAUT monitoring framework in SCALA [5], see also the github repository [3].

The LOGSCOPE website, from which all material is accessible is located at:

`https://github.jpl.nasa.gov/pages/logscope/logscope.github.io`

The LOGSCOPE JPL internal github repository containing all material can be found at:

`https://github.jpl.nasa.gov/logscope`

The document is organized as follows. Section 2 provides an overview of the tool. Section 3 is a language tutorial, explaining the notation though a series of examples. Section 4 is a language reference, explaining the grammar of the notation. Section 5

describes how to install and run the frontend. Section 6 describes how to install and run the backend. Section 7 concludes the report with some remarks on the work, including possible future extensions. Appendix A contains the full grammar for the specification language without comments.

2 Overview

LOGSCOPE supports formal analysis of event (telemetry) streams. The tool takes as input:

- a formal specification in the SCOPE language, expressing the properties that the event stream has to satisfy. The specification consists of a collection of monitor specifications.
- an event stream.

LOGSCOPE produces on standard output a report describing where (if at all) the event stream violates the specification. The results of monitoring can also be accessed as a data structure for further processing. Part of LOGSCOPE is written in C++, and supports *online* monitoring of executing C and C++ programs. However, it can also be used in *offline* mode, reading events from a previously produced log file.

The SCOPE specification Language merges rule-based programming with state machines. An example of a monitor specification in the SCOPE language is the following, formalizing the property that: *"Every command (with some apriori unknown name, bound to the variable 'x') must eventually succeed, without a failure before"* (the language will be explained in detail in subsequent sections):

```
monitor CommandsMustSucceed {
  always {
    COMMAND(name : x) ⇒ RequireSuccess(x)
  }

  hot RequireSuccess(cmdName) {
    FAIL(name : cmdName) ⇒ error
    SUCCESS(name : cmdName) ⇒ ok
  }
}
```

LOGSCOPE consists of two parts: a **frontend** and a **backend**:

- The **frontend** parses a specification written in SCOPE, and translates it to a C++ file, named `contract.cpp`, which implements a monitor for each monitor specification.
- The **backend** imports the generated C++ file `contract.cpp` and performs the actual event monitoring from the main program in `main.cpp`.

Figure 1 illustrates the architecture of LOGSCOPE. The monitor specifications, written in the SCOPE specification language by a user, is by the **frontend** (written in the SCALA programming language) translated into C++, stored in the file `contract.cpp`. The **backend** compiles with the `contract.cpp` file, as well as with a main program

in the `main.cpp` file, also written by a user. This main program is responsible for obtaining events $E_1, E_2, \ldots$ from the *System Under Observation*, referred to as SUO, and forwarding them to the backend, which then monitors them using the contract in `contract.cpp`.

Each monitor in `contract.cpp` maintains an internal memory, called the *frontier*, which is a set of active *states* $S_1, S_2, ..., S_k$. We have not yet introduced the concept of a state, but think of a state as a state in a state machine for now. The frontier of a monitor can contain more than one state, each parameterized with its own data, which is one difference from traditional state machines. As we shall see, a state can have transitions out of the state, which can delete states, create new states, and/or issue error messages to a report. For each incoming event E_i, a monitor conceptually applies the event to each state $S_1, S_2, \ldots, S_k$ in the frontier[1], causing states to be removed, states to be added, and/or error messages to be issued.

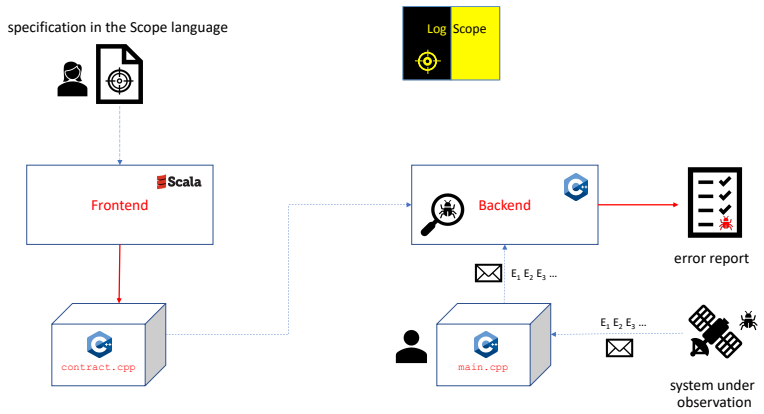


Figure 1: The LogScope architecture.

In *online* monitoring, where the monitor continuously monitors the SUO as it executes, there is conceptually no end to the monitoring, it theoretically continues "forever". However, in *offline* monitoring, analyzing e.g. a log file, monitoring terminates after the last event in the log file has been processed. The specification language contains language constructs, which only have meaning when/if end of monitoring occurs. Specifically it is checked that there are no remaining unfulfilled obligations: events that should occur but did not.

In the following we first present the SCOPE specification language, and subsequently the tool iteself in terms of the frontend and backend.

[1]Optimizations can avoid examining all states.

3 Language Tutorial

A *specification* consists of one or more files, each containing zero or more *monitor specifications*, also referred to as *monitors* when there is no confusion, since that term *monitor* technically refers to the implementation (in C++) of a *monitor specification*.

Each monitor represents a *property* that must hold on an event sequence. We shall illustrate the LOGSCOPE specification language, named SCOPE, though a sequence of examples, that combined cover the different aspects of the language. All the examples concern the commanding of a planetary rover.

3.1 Events

Conceptually, an event is a named record, with a *name* and a *mapping* from fields to values, where both fields and values are strings. We can think of an event to have the following form:

$$name(field_1 : value_1, \ldots, field_n : value_n)$$

In case the map is empty we just refer to the *name*. Some examples are:

- *reboot*
- $command(name : "TURN", kind : "FSW", sol : "125")$

This description suffices to understand the specification language. Later, in Section 6, we shall see how such events are concretely created with the backend C++ API.

3.2 A Simple State Machine

Let us assume that the SUO repeatedly emits two events:

– command : command being issued to, and received by, rover
– succeed : successful termination of command execution on rover

Note that for this first example we do not care about the fact that there are different kinds of commands. We also do not care about the data that events carry. We want to monitor the following property:

> **Property P_1:**
> *After submission of a command, a success of the command must follow, and no other command can be submitted in between.*

The monitor for this property is shown in Figure 2. The monitor, named M1, first declares which events it will monitor, namely command and succeed. Declaring such events has the main purpose of reducing the risk of making specification mistakes by e.g. misspelling event names when defining the states. Then two states are defined: Command and Succeed. The state Command is the initial state of the state machine, indicated by the

state modifier **init**. The state contains one transition: command ⇒ Succeed, expressing that if a command event is observed, then we leave (remove from the frontier) the Command state and enter (add to the frontier) the Succeed state. The Succeed state is annotated with a **hot** modifier, with the meaning that this state must be left (removed) before *end of monitoring* occurs (if it occurs). Leaving the Succeed state can happen in one of two ways. Either a succeed event occurs, in which case we return to the Command state, or another command event occurs, in which case we report an **error**.

```
monitor M1 {
  event command, succeed

  init Command {
    command ⇒ Succeed
  }

  hot Succeed {
    succeed ⇒ Command
    command ⇒ error
  }
}
```

Figure 2: Monitor M1 for property P_1.

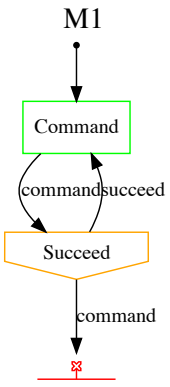


Figure 3: Monitor M1 visualized.

Textual monitors are visualized by LOGSCOPE using GRAPHVIZ's dot-format. This can help in convincing the specification writer that the specification expresses the intended property. The monitor in Figure 2 is visualized in Figure 3. Hot states (annotated in text with the modifier **hot**) are visualized as orange arrow shaped pentagons. Orange means danger: this state has to be left eventually. Non-hot states are visualized as green rectangles, we can stay in those "forever" (terminating monitoring in such a state is ok). The initial state Command is pointed to by an arrow leaving a black point. Transitions are labelled with events (and additional *conditions* as we shall see later). The color red in general indicates error. For example a command issued in the Succeed state causes an error, symbolized with a red cross on a flat line.

3.3 Some Alternative Monitors

Figure 4 shows some alternative monitors for property P_1, illustrating different aspects of the language. In the monitor M1a, instead of alternating between the states Command and Succeed, whenever a command event is observed in the Command state, in addition to creating a Succeed state, we immediately re-create the Command state. This is done by listing the Command state on the right-hand side of the transition arrow $\Rightarrow$, in addition to the Succeed state, separated by a comma. In the Succeed state itself, instead of creating a Command state on observing a succeed event, the **ok** state is entered, which effectively means that we are leaving the Succeed state successfully. This approach is, however, semantically slightly different in the sense that this monitor will keep looking for successes of commands, even after a failure due to a command being issued while waiting for a success. In the monitor M1, such an extra command will cause the frontier to become empty. Note that we have not annotated the Command state with the modifier **init**. In case no states are annotated with **init**, the first state is by default initial (unless the monitor contains *anonymous states* as shown in monitor M1c).

Monitor M1b shows how we can annotate a state with the modifier **always** to obtain the same effect as the transition in the Command state in the M1a monitor. The **always** modifier causes the state to always persist, even when transitions out of the state are taken. It is common for such **always** states to be anonymous, by not giving them a name. This is shown in the monitor M1c, which is the recommended (most convenient) way to write this monitor. If there are anonymous states in a monitor they become initial states in addition to states explicitly annotated with the modifier **init**. Only if there are no states annotated with **init** and there are no anonymous states, the first state becomes the initial state.

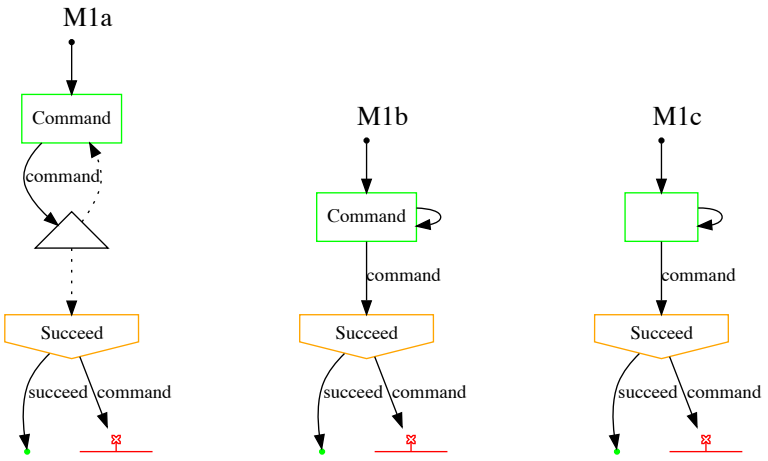
The three alternative monitors for property P_1 are visualized (again by LOGSCOPE) in Figure 5. Figure 5a, the visualization of M1a, shows how multiple target states are visualized: the transition of the Command state triggered by a command event creates a Succeed and a Command state. This is visualized with a black triangle (symbolizing a Boolean 'and': $\wedge$) with dashed lines leading to the target states. Note how in the Succeed state, a succeed event leads to **ok** which in the visualization is shown as a green dot. The visualization of monitor M1b in Figure 5b illustrates how an **always** state is visualized: with an unlabelled self loop. The difference between the visualization of this monitor and of M1c in Figure 5c is only that the initial state in Figure 5c has no name.

```
monitor M1a {
  event command, succeed

  Command {
    command ⇒ Succeed, Command
  }

  hot Succeed {
    succeed ⇒ ok
    command ⇒ error
  }
}

monitor M1b {
  event command, succeed

  always Command {
    command ⇒ Succeed
  }

  hot Succeed {
    succeed ⇒ ok
    command ⇒ error
  }
}

monitor M1c {
  event command, succeed

  always {
    command ⇒ Succeed
  }

  hot Succeed {
    succeed ⇒ ok
    command ⇒ error
  }
}
```

Figure 4: Alternative monitors M1a, M1b, and M1c for property P_1.

3.4 Monitoring Events that Carry Data

We shall now monitor events that carry data, represented as maps from fields to string values. This is where LOGSCOPE distinguishes itself from more traditional state machines. In our next example along the same theme, we shall specifically distinguish between different commands identified by name and kind. That is, our events have the form:



(a) Monitor M1a. (b) Monitor M1b. (c) Monitor M1c.

Figure 5: Monitors M1a, M1b, and M1c visualized.

– command(name : c, kind : k) : command being issued
– succeed(name : c) : successful termination of command execution

The kind k can for example be the string "FSW" (Flight Software, in contrast to Flight Hardware). We shall now modify the property P_1 slightly. The property stated that *"After submission of a command, a success of the command must follow, and no other command can be submitted in between"*. We shall now distinguish between different commands, identified by their names, and relax the property to:

> **Property P_2:**
> *After submission of a flight software command with a name x, a success of the command named x must follow (with the same name), and that command x cannot be re-submitted in between.*

Note that x is a variable representing any command name observed. This means that in between a command named x and its success, another command named y can be submitted as long as $x \neq y$. A monitor for this property is shown in Figure 6. Now the events are declared to carry maps (data). The event command carries a map defining two fields, name, a string denoting the name of the command, and kind, the kind of the command. The succeed command carries its name. Note that all data are strings[2]. The anonymous initial **always** state contains a single transition, which on the left-hand

[2] An extension of the language can allow different types of values.

side of the arrow $\Rightarrow$ matches any command event where the kind field is the string "FSW". On such a match the command name itself is *bound* to the variable x. This x is then referred to on the right-hand side state, where it is bound to the c field of the created Succeed state. So, e.g. if the command $command(name : "TURN", kind : "FSW")$ is observed, then a $Succeed(c : "TURN")$ state is created.

The Succeed state itself is parameterized with a map with a single field c. This field is referred to in the transitions. For example, the first transition states that if a succeed event is observed with a map, which maps the field name to the value of c that was passed as parameter, then we successfully leave the Succeed state by creating an **ok** state. On the other hand, if a command event is observed where the name is c, it is an **error**. Note, that we do not refer to the kind field of the command event in the second transition of the Succeed state (even though all commands monitored by this monitor defines a kind field in their associated maps). The intent is that we do not want any command of any kind with the name c to occur while waiting for a success. We could alternatively have narrowed it down to flight software commands by adding a kind : "FSW". Note that the order of arguments are not important since we are dealing with maps. The format of monitor M2 is typical, many properties will have this form.

Monitor M2 is visualized in Figure 7. The difference from previous visualizations is that now events carry data maps, which is shown. It is also shown how bindings to fields in target state maps are created. Specifically, the transition from the initial **always** state:

```
command(name : x, kind : "FSW") ⇒ Succeed(c : x)
```

is shown as an edge labelled with command(cmd : x, kind : "FSW"), and below it the binding of the c field of the Succeed state (see its definition) to the x that was bound on the left of the $\Rightarrow$ symbol.

```
monitor M2 {
  event command(cmd,kind), succeed(cmd)

  always {
    command(cmd : x, kind : "FSW") ⇒ Succeed(c : x)
  }

  hot Succeed(c) {
    succeed(cmd : c) ⇒ ok
    command(cmd : c) ⇒ error
  }
}
```

Figure 6: Monitor M2 for property P_2.

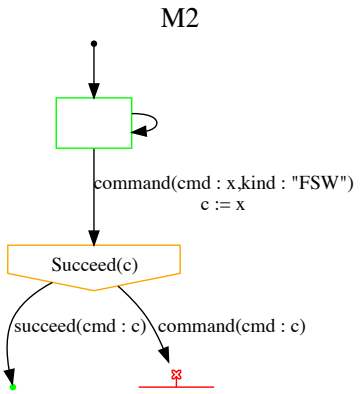


Figure 7: Monitor M2 visualized.

3.5 Referring to the Past

The properties we have seen so far are what we call a *future time* properties. They have the general form: "*if some event occurs, then some other events have to occur in the future and/or other events should not occur in the future*". It is, however, also useful sometimes to refer to things that happened in the past, and specifically to things that did not happen. Let us add a constraint to property P_2, namely that *a command is only allowed to succeed, if it has been commanded in the past and not yet succeeded*. The added constraint refers to the past. That is our property now becomes:

> **Property P_3:**
> *After submission of a flight software command with a name x, a success of the command named x must follow (with the same name), and that command x cannot be re-submitted in between. Furthermore, a command is only allowed to succeed if it has been commanded in the past and not yet succeeded.*

The monitor M3 in Figure 8 monitors this property. The monitor is the same as in Figure 6 except that in the initial **always** state we have added an extra transition:

```
succeed(name : x) @ !Succeed(c : x) ⇒ error
```

In addition to the event pattern succeed(name : x), after the symbol @ follows a condition !Succeed(c : x) stating that there does not (! is negation) exist a state in the *frontier* with a map that maps the field c to the value x bound in the event on the left-hand side of the

@ symbol. If there is no such state, hence the command x is not expected to succeed, then an error is reported. In general, after the @ symbol, a comma separated list of conditions can occur (negated or not), which each have to be true for the transition to be taken. The conditions can bind variables, exactly as does our event here. Bindings can be seen in patterns occurring to the right of the bindings.

Monitor M3 is visualized in Figure 9. The only new visualization concept here is that the transition from the initial **always** state to the error state is now labelled not only with the event pattern succeed(name : x) but also with the condition pattern !Succeed(c : x) underneath.

```
monitor M3 {
  event command(cmd, kind), succeed(cmd)

  always {
    command(cmd : x, kind : "FSW") ⇒ Succeed(c : x)
    succeed(cmd : x) @ !Succeed(c : x) ⇒ error
  }

  hot Succeed(c) {
    succeed(cmd : c) ⇒ ok
    command(cmd : c) ⇒ error
  }
}
```

Figure 8: Monitor M3 for property P_3.

3.6 A Complex Property

The following final example does not introduce essential new language features (except a less essential one), but illustrates how a more complex monitor can look like. We expand the scenario with additional events. We here assume that when a command is received on the rover, it is not immediately executed, but rather it is stored in a queue. While in the queue the command can be cancelled. If not cancelled it is then eventually dispatched for execution. The execution can fail or it can succeed. After successful execution, the command has to be closed (e.g. cleaning up). Each command, in addition to having a name, is now also associated with a command number, increased by 1 for each submitted command. We consider the following events:

- command(name : c, nr : n, kind : k) : command being issued
- cancel(name : c, nr : n) : cancelling of the command
- dispatch(name : c, nr : n) : dispatch of the command
- fail(name : c, nr : n) : failure of the command
- succeed(name : c, nr : n) : successful termination of command execution
- close(name : c, nr : n) : closing of the command

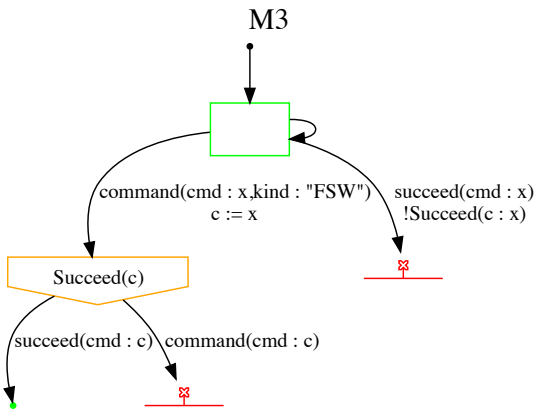


Figure 9: Monitor M3 visualized.

We shall refer to the combination of a command name and its number as a *command instance*. Our new property is as follows.

> **P4:**
> *After submission of a flight software command instance, a dispatch of the command instance must follow, unless it is cancelled first. Once dispatched, it must succeed, without any failure occurring before. In between the dispatch and the success of a command instance, we should observe no re-submission of that command (any command instance with that name). A command instance is not allowed to succeed unless it has been dispatched. Once a command instance has succeeded, it must be closed, and it is not allowed to succeed again.*

The monitor for property P_4 is shown in Figure 10. The second transition in the initial **always** state uses a condition to catch command successes that are not expected. The first transition in the Succeed state creates two new states, a NoMoreSuccess state and a Close state. The second transition in the Succeed state uses a wildcard symbol _ to indicate that any flight softare command instance with the name sc will in this state cause an error, we don't care what the command number is. A perhaps surprising observation is, that that transition has the same meaning as the following transition where we do not mention the nr field at all:

```
command(cmd : sc, kind : "FSW") ⇒ error
```

This is because, as indicated before, these field declarations are map constraints: the command must *at least* have these fields and conform with the range values. Whether we use a don't care symbol _ or not does have an effect in case we do not declare our events at the beginning of the monitor though: using a don't care symbol, as in `nr : _`, does require that there is an `nr` field in the command's data map. If not an error is issued.

Monitor M4 is visualized in Figure 11. The reader should try to co-relate it with the textual specification in Figure 10. Recall that by observing the color scheme one can from the graph quickly understand the violations being checked for: orange means terminating here is a violation, and red means an occurred violation.

```
monitor M4 {
  event command(cmd,nr,kind), cancel(cmd,nr), dispatch(cmd,nr),
        fail(cmd,nr), succeed(cmd,nr), close(cmd,nr)

  always {
    command(cmd : c, nr : n, kind : "FSW") ⇒ Dispatch(dc : c, dn : n)
    succeed(cmd : c, nr : n) @ !Succeed(sc : c, sn : n) ⇒ error
  }

  hot Dispatch(dc,dn) {
    cancel(cmd : dc, nr : dn) ⇒ ok
    dispatch(cmd : dc, nr : dn) ⇒ Succeed(sc : dc, sn : dn)
  }

  hot Succeed(sc,sn) {
    succeed(cmd : sc, nr : sn) ⇒
      NoMoreSuccess(nc : sc, nn : sn), Close(cc : sc, cn : sn)
    command(cmd : sc, nr : _, kind : "FSW") ⇒ error
    fail(cmd : sc, nr : sn) ⇒ error
  }

  NoMoreSuccess(nc,nn) {
    succeed(cmd : nc, nr : nn) ⇒ error
  }

  hot Close(cc,cn) {
    close(cmd : cc, nr : cn) ⇒ ok
  }
}
```

Figure 10: Monitor M4 for property P_4.

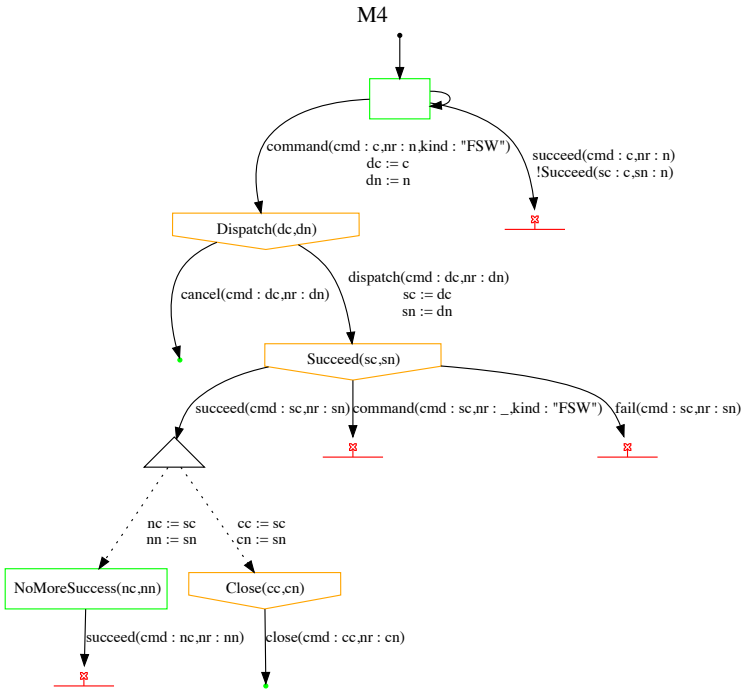


Figure 11: Monitor M4 visualized.

4 Language Reference

In this section we will go over the precise grammar rules defining the grammar of the language and explain these. The full grammar of the language (the union of the grammar rules in this Section) is included in Appendix A. The grammar uses a few notational conventions, explained in Table 1.

4.1 Specifications

$$\langle Specification \rangle ::= \langle Monitor \rangle *$$

We start at the top level. A specification is zero, one, or more monitors. Note that the implementation, as discussed in Section 5, allows a specification to be contained in one or more files. However, the grammar does not reflect this practical detail.

Table 1: Grammar notation.

$\langle Name \rangle ::= \ldots$	definition of a nonterminal
$\ldots ::= \ldots \langle Name \rangle \ldots$	reference to a nonterminal
keyword	a keyword
$A_1 \vert \ldots \vert A_n$	choice between alternatives $A_1, \ldots, A_n$
A^*	A zero or more times
A^+	A one or more times
$A,^*$	A zero or more times, separated by commas
$A,^+$	A one or more times, separated by commas
$[A]$	optional A

4.2 Monitors

$\langle Monitor \rangle$ ::= **monitor** $\langle Id \rangle$ '{' $\langle EventDef \rangle$* $\langle State \rangle$* '}'

A monitor is introduced by the keyword **monitor**, followed by its name, followed by a body in between curly brackets, consisting of zero or more event definitions and zero or more states. If there are no states the monitor does not perform any monitoring. Also note, that one does not need to define events. In that case the events are inferred from the state transitions. Providing event definitions serves two purposes: (1) to offer an additional well-formedness check on the state transitions, that they refer to events declared, and (2) if no events are declared, then only events used in the state transitions are submitted to the monitor, otherwise all declared events are submitted. This can make a difference when using **step** and **next** modifiers, as explained later.

4.3 Event Definitions

$\langle EventDef \rangle$ ::= **event** $\langle Event \rangle$,+

$\langle Event \rangle$ ::= $\langle Id \rangle$ ['(' $\langle Id \rangle$,+ ')']

An event definition is the keyword **event** followed by one or more events. An event is a name followed by an optional list of arguments in between parentheses. Note that the arguments define the fields in a map (the domain/key-values of the map). For example, declaring an event as follows:

```
event command(name, nr, kind)
```

indicates that command events submitted to the monitor will have *at least* these three fields. E.g. an event may be the following (having an extra *msg* field that we don't care about):

$$command(name : "TURN", nr : "223", kind : "FSW", msg : "Point\ to\ Earth")$$

4.4 States

⟨*State*⟩ ::= ⟨*Modifier*⟩+ '{' ⟨*Transition*⟩+ '}'
| ⟨*Modifier*⟩* ⟨*Id*⟩ ['(' ⟨*Id*⟩,* ')'] ['{' ⟨*Transition*⟩+ '}']

A state can have one of two forms, either an *anonymous state* without a name, or a *named state* with a name, in which case it can also have an optional set of parameters. In the anonymous case a modifier must be provided, typically the **always** modifier. In the case the state has a name, it can have an optional parameter list. This means that the state is parameterized with a map, the domain of which are these parameter names. Within the state these names can be referred to. The body of a state is an optional sequence of transitions in between curly brackets. Note that a state can have no transitions. This can make sense if the state's only purpose is to occur as a condition in some other transitions. The initial state of a monitor is defined as follows. Any anonymous state is initial. Any state with modifier **init** is initial. If there are no anonymous states and no states with modifier **init**, then the first state is the initial state.

4.5 Modifiers

⟨*Modifier*⟩ ::= **init** | **always** | **hot** | **step** | **next**

Each state can be annotated with one or more modifiers separated by spaces. Table 2 summarizes the modifiers available for annotating states and their meaning. The **step** and **next** modifiers have not been explained yet. Their use is more rare. A **step** state will be deleted from the frontier in the next step if none of its transitions fire. An **next** state will cause an error in the next step (if there is a next step) if none of its transitions fire. Some combinations are illegal: Modifier **always** cannot occur together with **hot** or **step**. Modifier **step** cannot occur together with **next** or **hot**.

4.6 Transitions

⟨*Transition*⟩ ::= ⟨*Pattern*⟩ ['@' ⟨*Pattern*⟩,+] '⇒' ⟨*Pattern*⟩,*

A transition consists of an event pattern, matched against events submitted to the monitor, and an optional list of conditions (also patterns) after the @ symbol, all matched against the *frontier*. If the event pattern matches the submitted event, and the condition patterns (if any) match states in the frontier (or don't match if they are negated), the transition fires, and the *action* patterns occurring to the right of the ⇒ symbol are "executed", meaning that states are created corresponding to these patterns. If an action pattern is negated, it is removed. Note that when taking a transition, the source state containing the transition is removed from the frontier, unless the state is annotated with then **always** modifier.

4.7 Patterns

⟨*Pattern*⟩ ::= ['!'] ⟨*Id*⟩ ['(' ⟨*Constraint*⟩,* ')']

⟨*Constraint*⟩ ::= ⟨*Id*⟩ ':' ⟨*Range*⟩

⟨*Range*⟩ ::= ⟨*Value*⟩ | ⟨*Id*⟩ | '_'

⟨*Value*⟩ ::= ⟨*String*⟩ | ⟨*Number*⟩

A pattern consists of an optional negation, a name, and an optional list of constraints enclosed by parentheses. A pattern can occur in one of three places in a transition: as an event pattern (the first required pattern), as a condition pattern after @ (optional), or as an action pattern after ⇒ :

```
event_pat @ cond_pat_1, ..., cond_pat_m ⇒ action_pat_1,..., action_pat_n
```

An *event pattern* matches an event if the event has the same name as the pattern, and its map satisfies the constraints. A negated pattern matches an event if the pattern does not match. A *condition pattern* matches if there exists a state in the frontier that matches. A negated condition pattern matches if no such matching state exists in the frontier. The purpose of an *action pattern* is to create a state, with a map indicated by the list of constraints. Hence its purpose is constructive rather than matching. Special cases of action patterns are **ok** and **error** (considered as keywords).

For event patterns and condition patterns, a constraint *id* : *range* matches a map from an event or state, if the map contains an entry *id* : *value*, where the *value* satisfies the *range*. A *range* is either a string literal, a number, an identifier, or the *don't care* symbol _. In the case of an identifier it is either binding if it is new, or constraining if it already has been introduced as a state parameter or in a pattern to the left. A Number matches a string containing that number.

4.8 Identifiers, Numbers, and Strings

⟨*Id*⟩ ::= ⟨*Letter*⟩ (⟨*Letter*⟩ | ⟨*Digit*⟩ | '_')*

⟨*Letter*⟩ ::= 'a' - 'z' | 'A' - 'Z'

⟨*Digit*⟩ ::= '0' - '9'

⟨*Number*⟩ ::= ⟨*Digit*⟩ ⟨*Digit*⟩*

⟨*String*⟩ ::= text between double quotes

An identifier is a letter followed by letters, digits, and underscores. A number is a sequence of digts, hence a natural number. A string is a sequence of characters enclosed in double quotes "...".

Table 2: State modifiers

init	initial state.
always	state is always part of the frontier.
hot	a hot state must not be in the frontier at the end of monitoring.
next	a transition must fire in next step, if there is a next step.
step	if a transition does not fire next, the state is deleted, this is not an error.

5 Frontend

5.1 Installing the Frontend

The frontend is located in the JPL internal github repository:

`https://github.jpl.nasa.gov/logscope/frontend`

The directory `out` contains the following files needed for running the LOGSCOPE frontend:

- `logscope` : script to run the translator from monitor specifications written in the SCOPE specification language to C++.
- `artifacts/frontend_jar/frontend.jar` : the LOGSCOPE jar file (generated from a SCALA program) used by the script above.

The LOGSCOPE frontend is implemented in SCALA and uses GRAPHVIZ for visualizing specifications.

Perform the following tasks to install the frontend:

1. Install SCALA: `https://www.scala-lang.org/download`. The frontend is developed in SCALA version 2.13.4.
2. Install GRAPHVIZ: `https://graphviz.org`.
3. Place the files `logscope` and `frontend.jar` mentioned above in some directory `LOGSCOPE_DIR` (standing for the total path to this directory). The name and location of the directory is not important.
4. `cd` to `LOGSCOPE_DIR` and make the `logscope` script executable:

   `chmod +x logscope`.
5. Make the `logscope` script reachable from anywhere, e.g. by updating the `PATH` variable in your `.bash_profile` file (or similar) to include the `LOGSCOPE_DIR` directory:

   `export PATH=LOGSCOPE_DIR:$PATH`

5.2 Running the Frontend

The `logscope` script is applied as follows to n files (for $n \geq 1$) containing monitor specifications in the SCOPE language:

$$\texttt{logscope}\ \langle file_1 \rangle\ \ldots.\ \langle file_n \rangle$$

LOGSCOPE will merge the monitor specifications in the different files into one specification and translate it to C++. It does not matter in which order the files are provided, or how monitors are distributed over the files.

Parsing and type checking

The specification is parsed, type checked, and finally translated to C++ if parsing and type checking succeeds. In case of parsing or type checking errors, information about these will be printed on standard out.

Generated files

In case the input specification passes the parsing and type checking, LOGSCOPE will generate:

- a directory `logscope-generated` containing:
  - a file `contract.cpp` containing generated C++ code, which will monitor all the specified monitors.
  - a directory `monitors-visualized` containing visualizations of monitors in `.png` format, one for each monitor.

6 Backend

The backend is located in the JPL internal github repository:

`https://github.jpl.nasa.gov/logscope/backend`

6.1 Installing the Backend

1. The backend is implemented in C++14 (the 2014 version). It has been developed with IntelliJ's Clion IDE for C++, but can be used in command line mode, which is recommended.
2. Install CMake: `https://cmake.org`. CMake makes life easier, as it generates a `Makefile` from much simpler data, consisting just of the paths to files to be compiled.
3. Download the backend repository as a zip file and unzip it. This creates a directory named: `backend_master`. We are now ready to compile and run.

6.2 Preparing the Backend for a Run

The installed LOGSCOPE package is ready to compile and run with a particular example. You may skip to the next step (Compiling the Backend) to see how it works.

However, when you want to create your own specifications and monitor your own event streams, the backend needs adjustment of two files at the top level:

- `contract.cpp` : generated by the frontend from a specification. Once generated by the frontend, move/copy this file to here. **This file should not be edited**.
- `main.cpp` : this is the main program, which instantiates the monitor in `contract.cpp` and feeds it events. **The user us supposed to edit this file.**

These files are currently populated with meaningful contents. The file `contract.cpp` is generated by the frontend from the monitor specification in Figure 8 (monitor M3 in `examples/example6/spec.scope`). The file `main.cpp` contains the main program shown in Figure 12 (with additional comments).

6.3 Compiling the Backend

1. Do: `cd backend_master`.
2. Do: `cmake CMakeLists.txt`. This creates a `Makefile` (and three other files that can be ignored).
3. Do: `make`. This compiles the C++ code.

6.4 Running the Backend

1. Do: `./logscope`. This runs `main.cpp` of the compiled code. Note that `./logscope` is the binary generated by the compilation, and not the frontend script.

```
1  #include "contract.h"
2
3  int main() {
4      SpecObject contract = makeContract();
5
6      contract.options.SHOW_PROGRESS = 0;
7
8      list<Event> events = {
9              Event(10, "command", {{"cmd",  "TURN"}, {"kind", "FSW"}}),
10             Event(20, "command", {{"cmd",  "TRACK"}, {"kind", "FSW"}}),
11             Event(30, "succeed", {{"cmd", "TURN"}}),
12             Event(40, "command", {{"cmd",  "PICT"}, {"kind", "FSW"}}),
13             Event(50, "succeed", {{"cmd", "SEND"}}),
14             Event(60, "command", {{"cmd",  "PICT"}, {"kind", "FSW"}})
15     };
16
17     for (Event &e : events) {
18         contract.eval(e);
19     };
20
21     contract.end();
22
23     // Process detected errors manually if needed:
24
25     cout << endl << endl << "Processing the result:" << endl << endl;
26     cout << contract.getEventCount() << " events processed" << endl;
27     cout << contract.getErrorCount() << " errors encountered" << endl;
28     cout << endl;
29     vector<ErrorReport> errors = contract.getErrorReports();
30     for (ErrorReport error : errors) {
31         cout <<
32               error.kind << ", " <<
33               error.monitorName << ", " <<
34               error.stateName << ", " <<
35               error.binding << ", " <<
36               error.eventCount << ", " <<
37               error.transitionCount <<
38               endl;
39     }
40 }
```

Figure 12: The main program in `main.cpp`.

6.5 The Main program, its Output, and Debugging

6.5.1 The Main Program

- We first, in line 1, include the header file `contract.h` for the contract, `contract.cpp`, that was generated by the frontend. This header file is always the same, independent of the generated contract.

- The header file allows us to create an instance of the contract in line 4 via a call of the `makeContract` function. It returns an object of the `SpecObject` class. We can now feed events to this object.
- Line 6 defines the value of the variable `SHOW_PROGRESS`, which controls the amount of output generated during monitoring. For now we set it to 0, which means minimal output including violations of the specification.
- For this presentation we first create a trace, a list of events, in lines 8-15. Each event is an instance of the class:

```
class Event {
public:
    Event();
    Event(int time, string name, unordered_map<string,string> data);
    friend ostream& operator<<(ostream& out, const Event& ev);
    int time;
    string name;
    Binding binding;
};
```

The `time` is a time stamp. It is not currently used by LOGSCOPE, but an extension would. The `name` is a name of the event. The `data` is a mapping from fields (strings) to values (strings).

In our case, the specification in Figure 8 processes the events: command(cmd,kind) and succeed(cmd). Here command and succeed are the names, and cmd and kind are the fields. If we look at the first event in line 9, it represents a command at time 10, with the "cmd" field having the value "TURN" and the "kind" field having the value "FSW".

- The for-loop in lines 17-19 iterates through each event e in the events variable (the trace) and feeds them to the contract via a call of the eval method. This causes the monitors in the contract to process the event, producing violation reports in case of monitor violations.
- Finally, in line 21, we end the monitoring. This can cause additional error messages to be produced in case any hot states remain in the frontiers of monitors. Note that during online monitoring this method may never be called.
- In case the user wants to process the errors manually, the `SpecObject` class offers some methods for doing this, as demonstrated in lines 25-39. Specifically, the method `getErrorReports`, called in line 29, returns a vector of all reported errors, each represented by an object of the `ErrorReport` class. In this case we just print out the reachable fields to demonstrate how they are accessed. Note that for some error reports the `eventCount` and `transitionCount` are not relevant. In those cases these fields have the value -1.

6.5.2 The Output

We shall go through the output generated by the main program. First, it prints out the specification with transitions numbered for later reference in error messages:

```
Monitoring the specification:

========================
monitor M3{
  event command, succeed

  always init {
    [1] command(cmd : x,kind : "FSW") => Succeed(c : x)
    [2] succeed(cmd : x) & !Succeed(c : x) => error
  }

  hot Succeed(c){
    [1] succeed(cmd : c) => ok
    [2] command(cmd : c) => error
  }
}
-------------------------
```

Then errors are reported as they are detected. In our case there are two errors detected during the trace evaluation:

```
Monitoring begins!

======= Error 1: =======
event number 5: succeed with Binding{{{"cmd","SEND"}}}
triggers transition:
  [2] succeed(cmd : x) & !Succeed(c : x) => error
in:
monitor M3 {
  ...
  always init {
    [1] command(cmd : x,kind : "FSW") => Succeed(c : x)
    [2] succeed(cmd : x) & !Succeed(c : x) => error
  }
  with: Binding()
  ...
}
----------------------

======= Error 2: =======
event number 6: command with Binding{{{"kind","FSW"},{"cmd","PICT"}}}
triggers transition:
  [2] command(cmd : c) => error
in:
monitor M3 {
  ...
  hot Succeed(c){
    [1] succeed(cmd : c) => ok
    [2] command(cmd : c) => error
  }
  with: Binding{{{"c","PICT"}}}
  ...
}
----------------------
```

The two errors are as follows:

- The first error message states that event number 5 (line 13 in Figure 12) triggers transition number 2 in the monitor `M3`, showing also the state (the anonymous

initial always-state in this case) in which this transition is located. Since each state is associated with a data binding it shows that as well, empty in this case. The error is caused by the fact that the a SEND command succeeds without ever having been commanded.

- The second error message states that event number 6 (line 14 in Figure 12) triggers transition number 2 in the monitor M3, showing also the state (Succeed) in which this transition is located. The data binding for this state maps "c" to "PICT". The error is caused by the fact that the a PICT command is issued twice, without a success in between, which is not allowed.

At the end of monitoring (at the call `contract.end()` in line 21), two additional errors are detected, each indicating that the monitor M3 ends in the non-final **hot** Succeed state, since neither the TRACK command nor the re-issued PICT command have succeeded.

```
Ending monitoring!

======= Error 3: =======
monitoring terminates in non-final state:
monitor M3 {
  ...
  hot Succeed(c){
    [1] succeed(cmd : c) => ok
    [2] command(cmd : c) => error
  }
  with: Binding{{{"c","PICT"}}}
  ...
}
----------------------

======= Error 4: =======
monitoring terminates in non-final state:
monitor M3 {
  ...
  hot Succeed(c){
    [1] succeed(cmd : c) => ok
    [2] command(cmd : c) => error
  }
  with: Binding{{{"c","TRACK"}}}
  ...
}
----------------------
```

Finally, a summary of the analysis is printed, here indicating number of events processed and number of errors detected:

```
=========================
Summary of Trace Analysis:
total number of events : 6

M3 errors : 4

total number of errors : 4
-------------------------
```

The manual printing of the detected errors appear as follows:

```
Processing the result:

6 events processed
```

```
4 errors encountered

TransitionError, M3, INTERNAL__1, Binding(), 5, 2
TransitionError, M3, Succeed, Binding{{{"c","PICT"}}}, 6, 2
EndError, M3, Succeed, Binding{{{"c","PICT"}}}, -1, -1
EndError, M3, Succeed, Binding{{{"c","TRACK"}}}, -1, -1
```

6.5.3 Debugging

In line 6 we set the `SHOW_PROGRESS` variable to 0, meaning minimal output, only monitor violations essentially. We can set the variable as follows:

- value 0 : no debugging information is printed.
- value 1 : every event processed is printed together with internal monitor states. This is useful for understanding how the monitoring engine works.
- value $N > 1$: event count printed for every N'th event. This is useful for knowing how far the monitoring engine has reached when processing very large traces.

If we for example set the value to 2 for our example, we will see the following lines printed amongst the error messages:

```
  event: 2
  event: 4
  event: 6
```

If we set the value to 1, we will see the following output, merged with the minimal output shown above. Each event is printed with the name and binding of fields to values, and the event number after the # symbol. The internal frontier of each monitor is printed, showing the active states and their data bindings (states with names such as `INTERNAL__1` represent anonymous states, typically always-states).

```
========================
command with Binding{{{"cmd","TURN"},{"kind","FSW"}}} #1
------------------------

=== monitor states: ===
monitor M3:
  Succeed with Binding{{{"c","TURN"}}}
  INTERNAL__1 with Binding()
----------------------

========================
command with Binding{{{"cmd","TRACK"},{"kind","FSW"}}} #2
------------------------

=== monitor states: ===
monitor M3:
  Succeed with Binding{{{"c","TRACK"}}}
  Succeed with Binding{{{"c","TURN"}}}
  INTERNAL__1 with Binding()
----------------------

========================
succeed with Binding{{{"cmd","TURN"}}} #3
------------------------

=== monitor states: ===
```

```
monitor M3:
  Succeed with Binding{{{"c","TRACK"}}}
  INTERNAL__1 with Binding()
----------------------

========================
command with Binding{{{"cmd","PICT"},{"kind","FSW"}}} #4
------------------------

=== monitor states: ===
monitor M3:
  Succeed with Binding{{{"c","PICT"}}}
  Succeed with Binding{{{"c","TRACK"}}}
  INTERNAL__1 with Binding()
----------------------

========================
succeed with Binding{{{"cmd","SEND"}}} #5
------------------------

*** Error 1 reported ...

=== monitor states: ===
monitor M3:
  Succeed with Binding{{{"c","PICT"}}}
  Succeed with Binding{{{"c","TRACK"}}}
  INTERNAL__1 with Binding()
----------------------

========================
command with Binding{{{"cmd","PICT"},{"kind","FSW"}}} #6
------------------------

*** Error 2 reported ...

=== monitor states: ===
monitor M3:
  Succeed with Binding{{{"c","PICT"}}}
  Succeed with Binding{{{"c","TRACK"}}}
  INTERNAL__1 with Binding()
----------------------

Ending monitoring!

*** Error 3 reported ...
*** Error 4 reported ...
```

6.6 Running Examples in the `examples` Directory

The `examples` directory contains the specifications presented in this document, together with associated test programs. To run examples in the `examples` directory, e.g. example 7:

1. apply the `logscope` script to the specification file `examples/example7/spec.scope`, and move the generated `contract.cpp` file to the top level of the backend, as previously explained.
2. Add `#include "examples/example7/example7.h"` to the `main.cpp` file.
3. To run one of the tests, e.g. number 1, add the statement: `example7::test1();` as the only statement in the body of the `main` function.

4. To run all tests in example 7, add the statement: `example7::test();` as the only statement in the body of the `main` function.

7 Remarks

The current version of the tool is a prototype. A continuation of this work would include the following activities.

Cleaning up C++ code In general, there is a need to walk through the backend C++ code and clean it up to match flight software practices.

Avoid dynamic memory allocation Specifically, the C++ code in the backend uses dynamic memory allocation. A refinement would consist of instead using statically sized object pools, effectively doing our own memory management.

Optimize monitoring The backend monitoring engine can be optimized to process events faster. Specifically, given a particular incoming event, we can identify which states are interested in this event in a more efficient manner using indexing.

Increase expressiveness of specification language Various extensions to the specification language can be considered, such as e.g. allowing constraints on time stamps, allowing **OR** of states (in addition to the current **AND**), and the use of C/C++ code as part of the specifications. Also, currently all data are strings. A typing discipline distinguishing between numbers, strings, etc. could be added.

Analysis of approach There is a need to discuss the approach with flight software developers. As a prototype, LOGSCOPE forms a data point in the discussion of how to monitor flight software systems. It is not the only solution. It is, however, a starting point from where a collaborative discussion can proceed. This also includes application to HPSC.

A Specification Grammar

The grammar for the LOGSCOPE specification language is defined in Figure 13.

⟨*Specification*⟩ ::= ⟨*Monitor*⟩*

⟨*Monitor*⟩ ::= **monitor** ⟨*Id*⟩ '{' ⟨*EventDef*⟩* ⟨*State*⟩* '}'

⟨*EventDef*⟩ ::= **event** ⟨*Event*⟩,+

⟨*Event*⟩ ::= ⟨*Id*⟩ ['(' ⟨*Id*⟩,+ ')']

⟨*State*⟩ ::= ⟨*Modifier*⟩+ '{' ⟨*Transition*⟩+ '}'
| ⟨*Modifier*⟩* ⟨*Id*⟩ ['(' ⟨*Id*⟩,* ')'] ['{' ⟨*Transition*⟩+ '}']

⟨*Modifier*⟩ ::= **init** | **always** | **hot** | **step** | **next**

⟨*Transition*⟩ ::= ⟨*Pattern*⟩ ['@' ⟨*Pattern*⟩,+] '⇒' ⟨*Pattern*⟩,*

⟨*Pattern*⟩ ::= ['!'] ⟨*Id*⟩ ['(' ⟨*Constraint*⟩,* ')']

⟨*Constraint*⟩ ::= ⟨*Id*⟩ ':' ⟨*Range*⟩

⟨*Range*⟩ ::= ⟨*Value*⟩ | ⟨*Id*⟩ | '_'

⟨*Value*⟩ ::= ⟨*String*⟩ | ⟨*Number*⟩

⟨*Id*⟩ ::= ⟨*Letter*⟩ (⟨*Letter*⟩ | ⟨*Digit*⟩ | '_')*

⟨*Letter*⟩ ::= 'a' - 'z' | 'A' - 'Z'

⟨*Digit*⟩ ::= '0' - '9'

⟨*Number*⟩ ::= ⟨*Digit*⟩ ⟨*Digit*⟩*

⟨*String*⟩ ::= text between double quotes

Figure 13: Grammar for the LOGSCOPE specification language.

References

[1] H. Barringer, D. Rydeheard, and K. Havelund. Rule systems for run-time monitoring: from Eagle to RuleR. In *Proc. of the 7th Int. Workshop on Runtime Verification (RV'07)*, volume 4839 of *LNCS*, pages 111–125, Vancouver, Canada, 2007. Springer.

[2] Howard Barringer, Alex Groce, Klaus Havelund, and Margaret Smith. Formal analysis of log files. *Journal of Aerospace Computing, Information, and Communication*, 7(11):365–390, 2010.

[3] Daut on github. `https://github.com/havelund/daut`.

[4] Klaus Havelund. Runtime verification of C programs. In *Proc. of the 1st TestCom/FATES conference*, volume 5047 of *LNCS*, Tokyo, Japan, 2008. Springer.

[5] Klaus 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.

[6] Klaus Havelund. Rule-based runtime verification revisited. *Software Tools for Technology Transfer (STTT)*, 17(2):143–170, 2015.

[7] LogFire. `https://github.com/havelund/logfire`.

[8] LogScope in Python. `https://github.com/havelund/logscope`.