The LogScope Log File Analyzer
User Manual

Klaus Havelund

Laboratory for Reliable Software
Jet Propulsion Laboratory
California, USA

December 15, 2008

Abstract

This manual explains how to use the LOGSCOPE log file analysis tool. LOGSCOPE has been developed to specifically assist in testing JPL's Mars Science Laboratory (MSL) flight software, but is very generic in nature and can in principle be applied to any application that produces some form of logging information (which almost any software does). The intended use is for offline post-processing of such logs, after execution of the system under test. LOGSCOPE processes logs in a specific format (PYTHON lists of events). The generation of logs in this format from original logcs is beyond the scope of the manual, but can typically be done with a small PYTHON script. LOGSCOPE can in principle also be used to monitor systems online, during their execution. However, this will require a different setup of the tool than described in this manual. Logs are checked against specifications written in a formalized specification language that includes a powerful parameterized extension of state machines and a practical and user friendly temporal logic. Specifications can furthermore be learned from log files. The manual explains the specification language and the API.

Contents

Chapter 1

Introduction

LOGSCOPE is a PYTHON program that supports analysis of log files for testing purposes. The tool in principle takes as input a log file and a specification of expectations wrt. the format of the log file, and produces as output a report on violations of the specification. A log file is assumed to be a PYTHON sequence containing PYTHON dictionaries as events, as explained in this report. LOGSCOPE has been developed to specifically assist in testing Mars Science Laboratory (MSL) flight software, but is very generic in nature and can in principle be applied to other applications.

The LOGSCOPE Specification Language (LOGSCOPE/SL) consists of two parts: (i) a high-level *pattern language*, much resembling a temporal logic, and (ii) a lower-level, but more expressive, *rule language* (we shall often refer to it as the automata language since specifications in this core language resemble familiar automata). Patterns are automatically translated to automata. It is the intention that the user should mostly write patterns, but automata can become necessary in certain cases where the extra expressive power is needed. Appendix A contains the full grammar of the total specification language, while Appendix B contains the list of predefined predicates, which can be used to constrain events in `where`-clauses.

Related Work and Sources of Inspiration

The LOGSCOPE system has specifically been influenced by the RULER system [1, 2, 3], and in particular wrt. the rule-based core language. LOGSCOPE shall really be seen as an adaptation of RULER to the specific needs of JPL's MSL project by adding temporal logic, making some adjustments, and re-programming in PYTHON. LOGSCOPE has, however, also been influenced by the state machine-based systems RCAT system [10] and RMOR [7]. A substantial amount of work has been done on runtime verication within the last decade [11], such as for example [9], [5], [4], and [8], all of which has guided in the design of RULER as well as LOGSCOPE.

Chapter 2

Log Files

LOGSCOPE can issue statements about the well-formedness of logs, where a log is a sequence of events. The kind of log that LOGSCOPE can process is more specifically a (PYTHON) sequence of events, where an event is assumed to be a PYTHON dictionary: a mapping from field names (strings) to values (strings, integers, floats, even lists). A special field named `"OBJ_TYPE"` must be defined in all events, and must be mapped to one of the five string values `"COMMAND"`, `"EVR"`, `"CHANNEL"`, `"CHANGE"`, and `"PRODUCT"`, indicating what kind of event it concerns. These five event kinds correspond to the five kinds of event reports generated by MSL[1], and are explained as follows:

- `COMMAND`: commands issued to the spacecraft (input to the system).
- `EVR`: internal transitions, usually generated by logging statements in the code.
- `CHANNEL`: periodic samplings of the spacecraft state.
- `CHANGE`: delta-changes to the spacecraft state.
- `PRODUCT`: science results produced by the software/hardware (output of the system).

Although these different event kinds have special meaning for MSL, LOGSCOPE is agnostic about their meaning. No other constraints are put on the remaining fields of an event. That is, any event can have any fields from the perspective of LOGSCOPE. How log files are generated is beyond the scope of this report. They can for example be generated from log files or program output generated in some other format, using an intermediate translating script that converts from the other log file format to the one expected by LOGSCOPE. Such a script would typically be written in PYTHON.

The following is an example of a *command* event.

[1] A soon-to-come version of the tool will be agnostic to the particular kinds of events generated in a log, and hence will not be MSL specific.

```
{
  "OBJ_TYPE" : "COMMAND",
  "EventNumber" : 12673,
  "String" : "REMS_FP_DMP,0",
  "EventTime" : "2008-295T18:47:52.000",
  "Stem" := "REMS_FP_DMP",
  "SCET" := None,
  "Arguments" := [’0’],
  "Number" := 2,
  "Type" := "FlightSoftwareCommand"
}
```

We shall as an example consider the following simplified log file consisting of 5 events as basis for a through-going example:

```
log =
 [
  {"OBJ_TYPE" : "COMMAND",  "Type" : "FSW",
      "Stem" : "PICT", "Number" : 231},
  {"OBJ_TYPE" : "EVR",  "Dispatch" : "PICT", "Number" : 231},
  {"OBJ_TYPE" : "CHANNEL",  "DataNumber" : 5},
  {"OBJ_TYPE" : "EVR",  "Success" : "PICT", "Number" : 231},
  {"OBJ_TYPE" : "PRODUCT", "ImageSize" : 1200}
 ]
```

The log could correspond to the command `"PICT"` to be fired, followed by a dispatch of that command, then a CHANNEL observation, then a success of the command, and finally a data product.

Chapter 3

Writing Patterns

3.1 The Simplest Possible Pattern

Assume the we want to express the following property:

> R_1: *Whenever a flight software command is issued, then eventually an EVR should indicate success of that command.*

Before we can formalize this property, it needs to be refined to refer to the specific fields of events. The following is such a refinement:

> $R_1^{refined}$: *Whenever a COMMAND is issued with the* `Type` *field having the value* `"FSW"`[1]*, the* `Stem` *field having some unknown value which we name x, and the* `Number` *field having some unknown value y, then eventually an EVR should occur, with the field* `Success` *mapped to x and the field* `Number` *mapped to y.*

Our log file defined in Section 2 in fact satisfies this specification since event number 1 (the command) is matched by event number 4, the success. Let's try to formalize this requirement in a specification. Specifications are written in separate text files, so we open a text editor. A specification file may contain zero or more specification units, each of which is either a temporal logic *pattern*, or a *rule system*, also referred to as an *automaton*. For now we shall focus on temporal logic patterns. Our property can be stated formally in LOGSCOPE/SL as follows:

```
pattern P1:
  COMMAND{Stem: x, Type : "FSW", Number: y} =>
     EVR{Success: x, Number: y}
```

[1] For the purpose of this presentation abbreviated from `"FlightSoftware"`

This pattern (‘pattern’ is a keyword) has the name P1 and states that *if* a command is observed in the log file at a position i, with the Stem field having some unknown value x, the Type field having the exact string value ”FSW”, and the Number field having some unknown value y; *then* later in that log file, at a position $j > i$, an EVR should occur with a Success field having x as value and a Number field having y as value. The pattern has the form:

```
'pattern' NAME ':' event => consequence
```

The event triggering the pattern is a command event (and will typically be), but can in general be of one of five forms:

- COMMAND$\{field_1 : range_1, \ldots, field_n : range_n\}$:
  matches events where OBJ_TYPE = "COMMAND"
- EVR$\{field_1 : range_1, \ldots, field_n : range_n\}$:
  matches events where OBJ_TYPE = "EVR"
- CHANNEL$\{field_1 : range_1, \ldots, field_n : range_n\}$:
  matches events where OBJ_TYPE = "CHANNEL"
- CHANGE$\{field_1 : range_1, \ldots, field_n : range_n\}$:
  matches events where OBJ_TYPE = "CHANGE"
- PRODUCT$\{field_1 : range_1, \ldots, field_n : range_n\}$:
  matches events where OBJ_TYPE = "PRODUCT"

In between the $\{\ldots\}$ brackets occur zero, one or more constraints, each consisting of a field name (without quotes), and a range specification. We saw two forms of range specifiations: the string "FSW" for the field Type and the names x and y for the other fields. A string constant represents a concrete constraint: the field in the event has to match this value exactly (by PYTHON equality ==). One can also provide an integer as such a concrete range constraint. A name (x and y in this case) on the left of => indicates that we do not constrain the values, we don’t even know what they may be, but we bind the values to these names so that they can be referred to in the consequence.

The consequence in our case is just an EVR event, referring to the names x and y. These names are now constraining: the corresponding fields now have to have the values these names were bound to by the triggering event.

3.2 Negation of Events

A consequence can also be the negation (’!’) of an event. Suppose we want to state the following property:

> *R_2: Whenever a flight software command is issued, then thereafter no EVR indicating failure of that command should occur.*

This requirement can as before be refined to be more specific:

> *$R_2^{refined}$: Whenever a COMMAND is issued with the* `Type` *field having the value* `"FSW"`*, the* `Stem` *field having some value x, and the* `Number` *field having some value y, then there should thereafter not occur an EVR, with the field* `Failure` *mapped to x and the field* `Number` *mapped to y.*

This can be expressed by the following property:

```
pattern P2:
  COMMAND{Type : "FSW", Stem: x, Number: y} =>
    ! EVR{Failure: x, Number: y}
```

Note the negation sign (‘`!`’) in front of the EVR. Again, our logfile from Section 2 satisfies this property.

3.3 Ordered and Unordered Consequences

We have seen that the consequence of a pattern can be an event or the negation of an event. There are two more forms: ordered and unordered sequences of consequences (a recursive definition). The syntax for consequences is as follows:

```
consequence ::=
   ['!'] event
 | '[' consequence_1,...,consequence_n ']'
 | '{' consequence_1,...,consequence_n '}'
```

Above in patterns `P1` and `P2` we saw instances of the first alternative: an event (in the case of `P2` it was negated). The next two alternatives are used when we want to compose consequences. There are two forms: `[...]` indicates that the consequences should occur *in exact order*, while `{...}` indicates that they may occur *in any order*. This is explained in the following two subsections.

3.3.1 Ordered Consequences

As an example, consider the following requirement:

> *R*$_3$*: Whenever a flight software command is issued, there should follow a dispatch of that command (with that number), and no dispatch failure before that, followed by a success of that command (with that number), and no failure before that, and no more successes of that command (exactly one success).*

This property can be stated as follows.

```
pattern P3 :
  COMMAND{Type: "FSW", Stem: x, Number: y} =>
    [
      ! EVR{DispatchFailure: x},
        EVR{Dispatch: x, Number: y},
      ! EVR{Failure : x, Number : y},
        EVR{Success: x, Number: y},
      ! EVR{Success: x, Number: y}
    ]
```

The consequence consists of a sequence (in square brackets `[...]`) of consequences, in this case events and negations of events. The ordering means that the dispatch should occur before the success, and the negations state what should *not* happen in between the non-negated events.

3.3.2 Unordered Consequences

It is also possible to indicate an un-ordered arrangement of events. For example, suppose we want to state the following slightly more relaxed property:

> *R*$_3$*: Whenever a flight software command is issued, there should follow a dispatch of that command (with that number), and also a success, but the two events can occur in any order (although this may not have much meaning). In addition, we don't* ever *want to see a dispatch failure or a failure of that command. Finally, after! a success there should not follow another success for that same command and number.*

This can be formulated in LogScope/SL as follows:

```
pattern P4 :
  COMMAND{Type: "FSW", Stem: x, Number: y} =>
    {
        EVR{Dispatch: x, Number: y},
        [
            EVR{Success: x, Number: y},
          ! EVR{Success: x, Number: y}
        ],
      ! EVR{DispatchFailure: x},
      ! EVR{Failure : x, Number : y}
  }
```

The curly brackets {...} indicate an un-ordered collection of consequences. The fact that they are un-ordered means that the non-negated events can occur in any order, and negations have to hold all the time. However, nested inside the {...} construct we have the ordered sequence:

```
[EVR{Success: x, Number: y}, !EVR{Success: x, Number: y}]
```

expressing that *after!* a success there should not occur another success.

Limitation

Note: a current limitation means that one cannot write a consequence *after* a {...} construct. That is, after an un-ordered sequence one cannot indicate an event to happen after all the un-ordered events have occurred. This functionality will be provided in a future version of LOGSCOPE.

3.4 Advanced Ranges

In the previous sections we have seen examples of field constraints of the form: *field* : *string* or *field* : *variable_name*, the first meaning that the field should have that exact value and the second that the value of the field is stored in the variable (a *binding*) if it occurs on the left hand side of =>, or that the field has to have whatever value the variable is bound to (a *constraint*) if it occurs on the right hand side of =>. It is also possible to indicate a number as range, an interval between two numbers, or indexing in case the value is indexible: that is, a list, string, dictionary or bit vector. This will be illustrated below.

Recall our log file that contained a CHANNEL with a DataNumber integer field and a PRODUCT with an integer ImageSize field. Assume we want to express the following property:

R_3: Whenever a flight software `"PICT"` *(take picture) command is issued, there should follow a* `CHANNEL` *event with DataNumber containing a bit vector where the three first bit positions from right to left have the values 1, 0, and 1, and there should thereafter follow a data* `PRODUCT` *with an ImageSize between 1000 and 2000.*

This property can be stated as follows using interval and indexing ranges:

```
pattern P5 :
    COMMAND{Type: "FSW", Stem: "PICT"} =>
    [
       CHANNEL{DataNumber : {0 : 1, 1 :0, 2 :1}},
       PRODUCT{ImageSize : [1000,2000]}
    ]
```

The `DataNumber` is constrained by the range `{0 :  1, 1 :0, 2 :1}` meaning that bit number 0 should have the value 1, bit number 1 the value 0 and bit number 2 the value 1, counted from the right. this matches exactly the number 5 occurring in the example log file (5 = 101 in binary). The format of an indexing range is a little mini constraint on values in various positions:

{ *value*$_1$: *range*$_1$,...,*value*$_n$: *range*$_n$ }

where a value can be an integer or a string. The indexing constraint can also be used to index into a string, a list (in both cases indexes would be integers, counting from the left) or a dictionary. The ranges can again be full ranges, including binding names. The PRODUCT constraint: `[1000,2000]` expresses that the image size should be between 1000 and 2000 in an obvious manner.

3.5 General Event Predicates

The notion of range constraints as illustrated in the previous section can be generalized to general predicates on events. LogScope/SL allows event definitions of the form:

```
Kind{field_1:range_1,...,field_n:range_n} where predicate
```

where `Kind` as usual is one of `COMMAND`, `EVR`, `CHANNEL`, `CHANGE` and `PRODUCT`, and where the `predicate` constraints the fields. The predicate can be an atomic predicate, like $P(x_1,\ldots,x_n)$ where P is either a predefined predicate (see Appendix B), or P is a user defined predicate (to be explained), and $x_1,\ldots,x_n$ are bound in the context. An atomic predicate can also be an arbitrary Python expression delimited by the symbol '`|`' on either side, as in: `|x>6|`. Predicates can be composed using the traditional Boolean operators: `and`, `or`, `not`, and brackets `(...)`. The user can define and/or import predicates at the beginning of a specification file delimited by the symbols `:::` ... `:::`. The following example illustrates how a variant of property `P5` above can be written using predicates instead of ranges.

```
:::

def bit(p,n):
    '''
    Get bit at position p of n.
    Binary count starts at position 0 from the right.
    '''
    return (int(n) >> int(p)) & 1

:::

pattern P6 :
  COMMAND{Type: "FSW", Stem: y} where |y.startswith("PIC")| =>
  [
    CHANNEL{DataNumber: d}
      where |bit(0,d)==1| and |bit(1,d)==0| and |bit(2,d)==1|,
    PRODUCT{ImageSize : s}
      where less_equal(1000,s) and less_equal(s,2000)
  ]
```

The specification starts with a definition in Python of the function `bit(p,n)`. The Python code must be enclosed with the symbol `:::` on either side (and occur at the beginning of the specification file). The pattern `P6` triggers on commands where the stem y is a string that starts with the prefix string `"PIC"`. The predicate:

```
|y.startswith("PIC")|
```

is a Python expression enclosed with `|` on either side. The other predicates used to constrain channel events are also Python expressions, however referring to the `bit(p,n)` function defined as part of the specification. The predicate `less_equal` can be called directly since it is pre-defined (Appendix B). Predicates that return a Boolean can be called directly without entering full Python code enclosed with `|...|`.

Chapter 4

Writing Automata

The patterns introduced above are translated into the kernel rule-based language, which includes automata/state machines. Thus rule-based language is more expressive than the pattern language and can also be used for writing specifications when the pattern language is not sufficient. This rule-based language is described in this chapter. It is not necessary to read this chapter in order to use the pattern language introduced above, but it is useful in order to understand the visualization of specification units described in the subsequent chapter.

We shall refer to a rule system as an automaton, although the concept is somewhat more general than the traditional concept of automaton. An automaton is expressed in terms of states and transitions between states triggered by events. Events are exactly as in patterns. Just as events can be parameterized with values as we have seen above, states can be parameterized too, hence carrying values produced by incoming transitions. Also, an automaton can be in several states at the same time, all of which have to lead to success[1]. We shall illustrate what some of the patterns we have seen above look like as automata, namely the automata they are translated to.

4.1 Automaton for P1

Consider the property P1 above (page 6). This property is by LOGSCOPE translated to the following automaton before monitoring (normally it would get the same name as the pattern, but for purposes of presentation we assign it a different name):

[1] Hence, an automaton is an AND-automaton in contrast to traditional OR-automata. LOGSCOPE will eventually be extended also with OR-states

```
automaton A_P1 {
  always S1 {
    COMMAND{Type : "FSW", Stem : x, Number : y} => S2(x,y)
  }

  state S2(x,y) {
    EVR{Success : x, Number : y} => done
  }

  initial S1
  hot S2
}
```

The automaton consists of two states: `S1` and `S2`. There is one transition exiting the `S1` state: this transition is triggered by a flight software command and enters state `S2(x,y)` with x and y now bound to the actual values in the event that matches. This is an example of a state being parameterized with data. The state `S1` is not parameterized. The state is an *always-state* meaning that it is always active, waiting for any command being observed. That is, event if the transition is taken into state `S2`, state `S1` is still active. State `S2` is a *normal state* meaning that when an exiting transition is taken we leave that state. The exiting transition is matched by an EVR where the x and y values have to match the values of the parameters of the state, which again was determined by the transition from S1 to S2. In other words, the names x and y in S1 are bound there, while the names x and y in state S2 are constraints. When the transition is taken we are finished monitoring, indicated by the `done` “state”.

The initial state is `S1`. The state `S2` is a *hot* state, meaning that it must be left before the end of the log file is recognized. If not it is regarded as an error. One can indicate more than one initial state and zero or more hot states. By default, if no initial state is indicated, the first state mentioned is initial. It is also possible to indicate the keyword `initial` in front of initial states instead of listing them at the end. Likewise, the hot states can be marked with the keyword `hot` inserted in front of the state (instead of lisiting them at the end of the automaton). This is illustrated in the next specification, which is the automaton corresponding to pattern P3 (page 9).

4.2 Automaton for P3

```
automaton A_P3 {
  always S1 {
    COMMAND{Type : "FSW",Number : y,Stem : x} => S2(x,y)
  }

  hot state S2(x,y) {
    EVR{DispatchFailure : x} => error
    EVR{Dispatch : x,Number : y} => S3(x,y)
  }

  hot state S3(x,y) {
    EVR{Failure : x,Number : y} => error
    EVR{Success : x,Number : y} => S4(x,y)
  }

  state S4(x,y) {
    EVR{Success : x,Number : y} => error
  }
}
```

This example illustrates how several different transitions can leave a state, enabled by different conditions. The `error` target state causes an error to be reported. The initial state is by `S1` since nothing else is stated. Here it waits for a command at which point it enters `S2` while binding x and y to the actual event values of `Stem` and `Number` respectively. In state `S2`, in case a dispatch failure occurs an error is reported and the monitoring stops tracking that specific automaton instance. Note, it would be possible to start this monitor in state `S2("PICT",231)` by providing the following initialization declaration:

```
automaton A_P3 {
  ...
  initial S2("PICT",231)
}
```

4.3 Automaton for P4

The final automaton we shall show is the one for pattern P4 (page 9). This automaton is characterized by having to express that after a command, we want to see a dispatch and a success, but in no particular order, just as there should not at any time be a dispatch failure or a failure. After a success, however, there should not be another success. The un-ordering of all but the last double success check is in the pattern on page 9 expressed by the curly bracket construct : `{...}`. In the automata world this is expressed by letting the transition triggered by a command in the initial `Watch` state enter several states: `wD`, `wS`, `noDF` and `noF`, all of which now have to lead to success (AND-semantics). State `wD` waits for a dispatch. State `wS` waits for a success whereafter it enters state `noS` where another success is not allowed. States `noDF` and `noF` check that no dispatch failure and failure occur. The states can be said to "*execute in parallel*".

```
automaton A_P4 {
  always Watch {
    COMMAND{Type : "FSW",Stem : x,Number : y} =>
         wD(x,y),wS(x,y),noDF(x,y),noF(x,y)
  }

  hot state wD(x,y) {
    EVR{Dispatch : x,Number : y} => done
  }

  hot state wS(x,y) {
    EVR{Success : x,Number : y} => noS(x,y)
  }

  state noS(x,y) {
    EVR{Success : x,Number : y} => error
  }

  state noDF(x,y) {
    EVR{DispatchFailure : x} => error
  }

  state noF(x,y) {
    EVR{Failure : x,Number : y} => error
  }
}
```

4.4 Step States and Success States

Consider the automaton `A_P3` on page 15. It expresses that after a command should follow a dispatch and then a success. There can be any number of events in between these required events. The automaton is formulated using a hot state. LogScope/SL also offers the dual notion of a *success* state[2]. A success state has to be reached by the end of the monitored log. We shall express the automaton using success states instead of hot states. We shall also strengthen the property and require that the command, the dispatch and the success occur right after each other with no other events in between. This can be achieved by declaring the states as *step* states, an alternative to declaring them *always* or *state* states. A *step* state is only active for one cycle and "goes away" unless left (or re-entered) in the next step. The following automaton expresses this requirement using *step* states and a *success* state. It is clear that if the automaton does not move forward in each step, we do not reach the success state.

```
automaton A_P3.2 {
  step S1 {
    COMMAND{Type : "FSW",Stem : x, Number : y} => S2(x,y)
  }

  step S2(x,y) {
    EVR{Dispatch : x,Number : y} => S3(x,y)
  }

  step S3(x,y) {
    EVR{Success : x,Number : y} => S4
  }

  step S4 {}

  initial S1
  success S4
}
```

4.5 Other Matters

LogScope offers two forms of comments:

[2] Often referred to as `final` states in automata theory. Hot states and success states are dual (can replace each other in the context of finite traces).

```
# this is a one-line comment

/* this is
a multi-line
comment
*/
```

It is possible to ignore specification units by prefixing the declaration with the keyword `ignore`, as in:

```
  ignore pattern P1:
    COMMAND{Stem: x, Type : "FSW", Number: y} =>
      EVR{Success: x, Number: y}
```

This simply ignores the specification unit during monitoring. This can be useful for experimenting with a set of specification units, or simply for abandoning a unit without directly deleting it. This form is simpler to use than to comment it out.

Chapter 5

Visualizing Automata

Automata, hand-written as well as translations of patterns, can be visualized with GraphViz [6]. Automata are stored in GraphViz's dot format, an internal representation for graphical diagrams. Figure 5.1 illustrates the diagram for automaton `A_P3`. The error states are black and hot states have a red edge and are pointing downwards as to indicate that once entering such a state, we have to leave the state before the end of the log in order to avoid an error message. *always*-states are annotated with '`@`' whereas *step*-states are annotated with '`#`' (`error` and `done` states are for technical reasons step states). It should be straight forward to see the relationship between this diagram and the textual format of `A_P3`.

Figure 5.2 illustrates the diagram for automaton `A_P4`. It illustrates a transition having several new target states, namely the transition from state `Watch` to the four different states: `wD` (wait for a Dispatch), `wS` (wait for a Success), `noDF` (no Dispatch Failure) and `noF` (no Failure). The branching is indicated with a upwards pointing blue-edged triangle, symbolizing and AND-symbol: '$\wedge$': all the four states must lead to satisfaction.

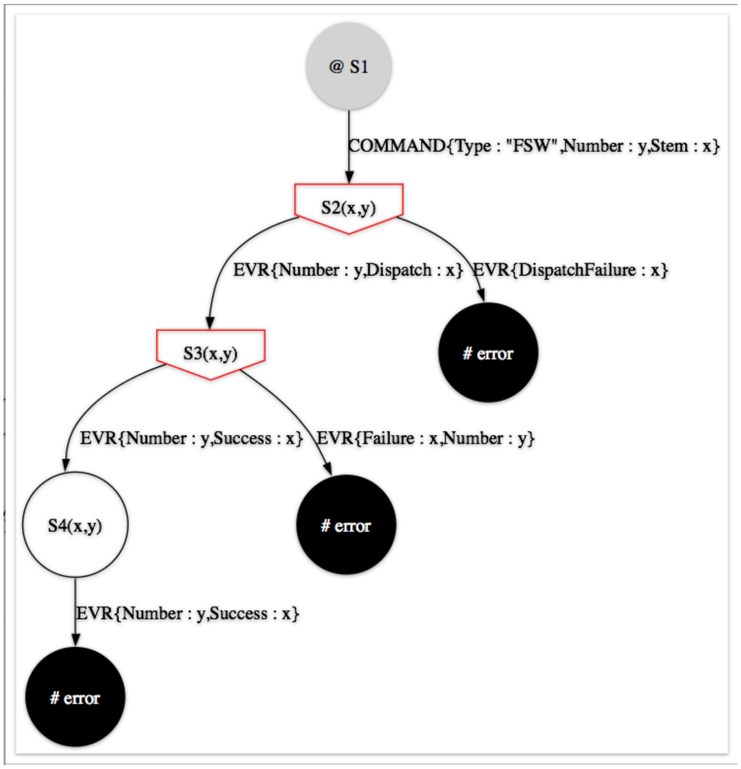
@ S1
COMMAND{Type : "FSW",Number : y,Stem : x}
S2(x,y)
EVR{Number : y,Dispatch : x}
EVR{DispatchFailure : x}
S3(x,y)
# error
EVR{Number : y,Success : x}
EVR{Failure : x,Number : y}
S4(x,y)
# error
EVR{Number : y,Success : x}
# error

Figure 5.1: GraphViz view of Automaton A_P3

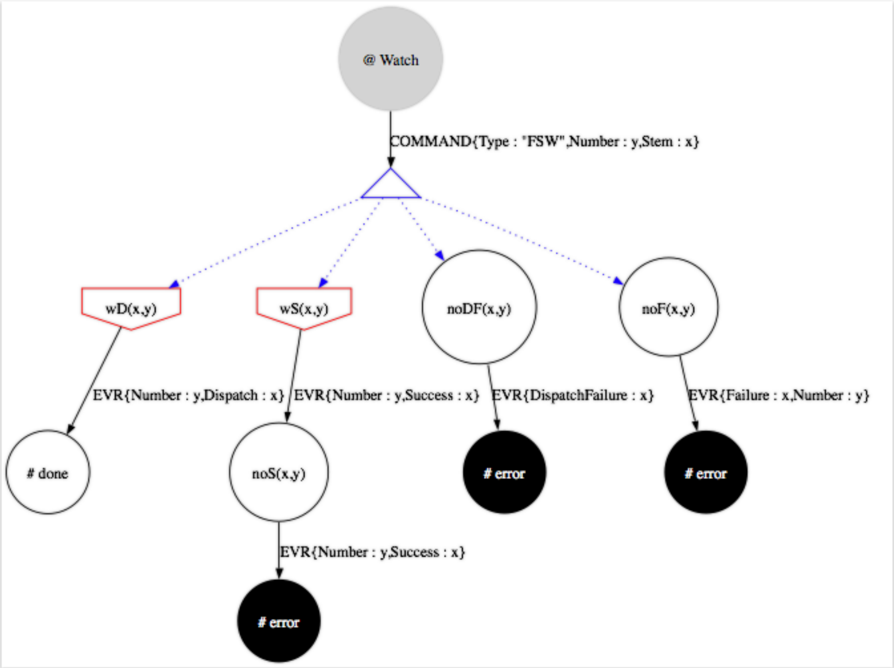
@ Watch
COMMAND{Type : "FSW",Number : y,Stem : x}
wD(x,y)
wS(x,y)
noDF(x,y)
noF(x,y)
EVR{Number : y,Dispatch : x}
EVR{Number : y,Success : x}
EVR{DispatchFailure : x}
EVR{Failure : x,Number : y}
# done
noS(x,y)
# error
# error
EVR{Number : y,Success : x}
# error

Figure 5.2: GraphViz view of Automaton A_P4

Chapter 6

The API and its Use

6.1 The API

The LogScope API for monitoring is very simple and consists of three classes: `Observer`, `Results` and `Error` as indicated below (parameter and result types are indicated after colons, where '`+`' means '*or*' and '`seq[X]`' means '*sequence of X*' - this is not correct Python, which is untyped, but helps illustrate the API):

The LogScope ***API***

```
def setResultDir(dir : string)

class Observer:
  def __init__(self,monitorThis : string+list[string])
  def monitor(self,log : list[dict]) : list[Results]
  def getResults(self) : list[Results]

class Results:
  def getSpecName(self) : string
  def getErrors(self) : list[Error]

class Error:
  def getLocation(self)
  def getMessage(self)
```

The `Observer` class gets instantiated with one or more specification file names, and thereafter provides a method `monitor` which can be applied to an event log, which it will check against the specifications. The `monitor` method returns a list of `Results` objects, one for each specification unit monitored. Each such `Results` object gives access to the name of the specification unit as well as a (possibly empty) list of `Errors`. The list of `Results` objects returned by the `monitor` method can also be accessed after monitoring by calling the `getResults` method.

The `monitorThis` argument to the `Observer` constructor must be either:

- a string : denoting an absolute name of a file containing a specification in LOGSCOPE/SL.
- a sequence of strings (in square brackets – `[...]`): denoting absolute names of files containing specifications. These specifications will then be joined into one.

Note that the results of a monitoring run are stored in the text file `RESULTS` in the result directory set by the `setResultDir(dir : string)` function. This access to the results is sufficient in many cases. However, the API does provide the possibility of accessing the results as data objects (the `getResults` method for example), which may be useful in regression testing where lots of scripts are run automatically.

6.2 An Example Script

Below is an example of using this API.

An example Python script

```
import lsm.lsm as lsm

# create an event log, in this case hand-made, but can also be
# extracted with log file extractor:

log = [
    {"OBJ_TYPE" : "COMMAND",
         "Type" : "FSW", "Stem" : "PICT", "Number" : 231},
    {"OBJ_TYPE" : "EVR", "Dispatch" : "PICT", "Number" : 231},
    {"OBJ_TYPE" : "CHANNEL", "DataNumber" : 5},
    {"OBJ_TYPE" : "EVR", "Success" : "PICT", "Number" : 231},
    {"OBJ_TYPE" : "PRODUCT", "ImageSize" : 1200}
  ]

# specify absolute path of where results should be stored
# (.dot files and RESULT file):

lsm.setResultDir("$EXAMPLES/results")

# instantiate the Observer class providing a total path name of
# specification file:

observer = lsm.Observer("$EXAMPLES/manual-examples")

# call the observer's monitor function on the log:

observer.monitor(log)
```

6.3 The Output Generated by LogScope

This chapter explains the output generated by LOGSCOPE and how to access it. The results of monitoring a log is by default printed to standard output. In addition, the results are written to a file named `RESULTS`, stored in the directory indicated by the directory indicated as parameter to the `setResultDir` function.

6.3.1 No Errors Detected

Our original log file on page 5 (same as in the script above) satisfies all the properties `P1`,...,`P5` and `A_P1`, `A_P3` and `A_P4`. Here we shall focus on properties `P1` and `P2` (by inserting the keyword `ignore` in front of all the other properties). When checking the log file against these two properties the following output is generated.

```
=================================
   parsed specification units:
=================================

pattern P1 :
  COMMAND{Type : "FSW", Stem : x, Number : y} =>
    EVR{Success : x, Number : y}

pattern P2 :
  COMMAND{Type : "FSW", Stem : x, Number : y} =>
    !EVR{Failure : x, Number : y}

=====================================
   translated specification units:
=====================================

automaton P1 {
  always S1 {
    COMMAND{Type : "FSW",Number : y,Stem : x} => S2(x,y)
  }

  state S2(x,y) {
    EVR{Number : y,Success : x} => S3(x,y)
  }

  state S3(x,y) {}

  initial S1
  hot S2
}

automaton P2 {
  always S1 {
    COMMAND{Type : "FSW",Number : y,Stem : x} => S2(x,y)
  }

  state S2(x,y) {
    EVR{Failure : x,Number : y} => error
  }

  initial S1
}

=====================================
   monitoring new log of length 5:
=====================================
============================
       RESULTS FOR P1:
============================

No errors detected!

Statistics {
  COMMAND :
      {’Type’: ’FSW’, ’Number’: 231, ’Stem’: ’PICT’} -> 1
  EVR :
      {’Number’: 231, ’Success’: ’PICT’} -> 1
}

============================
       RESULTS FOR P2:
============================

No errors detected!

Statistics {
  COMMAND :
      {’Type’: ’FSW’, ’Number’: 231, ’Stem’: ’PICT’} -> 1
}

========================
   Summary of Errors:
========================

P1  :  0
P2  :  0

specification was satisfied

5 events processed in 0 minutes and 0.00110 seconds (4545 events/sec)

Results are now being written to the file:
/Users/khavelun/Desktop/MSLRESULT/RESULTS

End of session!
```

That is, the output consists of the following elements:

- parsed specification units
- automata for all specification units (hand-written as well as generated from pat-

terns)

- information during monitoring on the progress, in this case nothing is printed since there are so few events. Normally is printed for every 100 event: number of events processed so far / total number of events, and the percentage this corresponds to.
- the result of the verification for each specification unit
- a summary of errors for all specification units and other statistics

The result in this case is that no errors were detected for any of the two patterns. For each specification unit is indicated statistics information indicating for each kind of event how many events matched conditions in the specification unit for each set of arguments relevant to the specification unit.

In the result directory is in addition to the `RESULT` file also stored two GrapViz dot files with the names `P1.dot` and `P2.dot`. These can be visualized by GraphViz (see instructions [6] for your system), as can be seen in Figure 6.1, using Mac's GraphViz package.

6.3.2 Errors Detected

Consider now that we modify the log file in the script on page 23 by changing event number 4 to become a *Failure* event instead of a Success event. Note, this is in this example done by having a `Failure` field map to the command name, instead of having a `Success` field map to the command name.

Log with event number 4 indicating failure

```
  log =
   [
    {"OBJ_TYPE" : "COMMAND",  "Type" : "FSW",
        "Stem" : "PICT", "Number" : 231},
    {"OBJ_TYPE" : "EVR",  "Dispatch" : "PICT", "Number" : 231},
    {"OBJ_TYPE" : "CHANNEL",  "DataNumber" : 5},
->  {"OBJ_TYPE" : "EVR",  "Failure" : "PICT", "Number" : 231},
    {"OBJ_TYPE" : "PRODUCT", "ImageSize" : 1200}
   ]
```

The output of running LOGSCOPE on this log file and properties `P1` and `P2` as before is as follows (omitting the listing of the patterns and automata as these are the same):

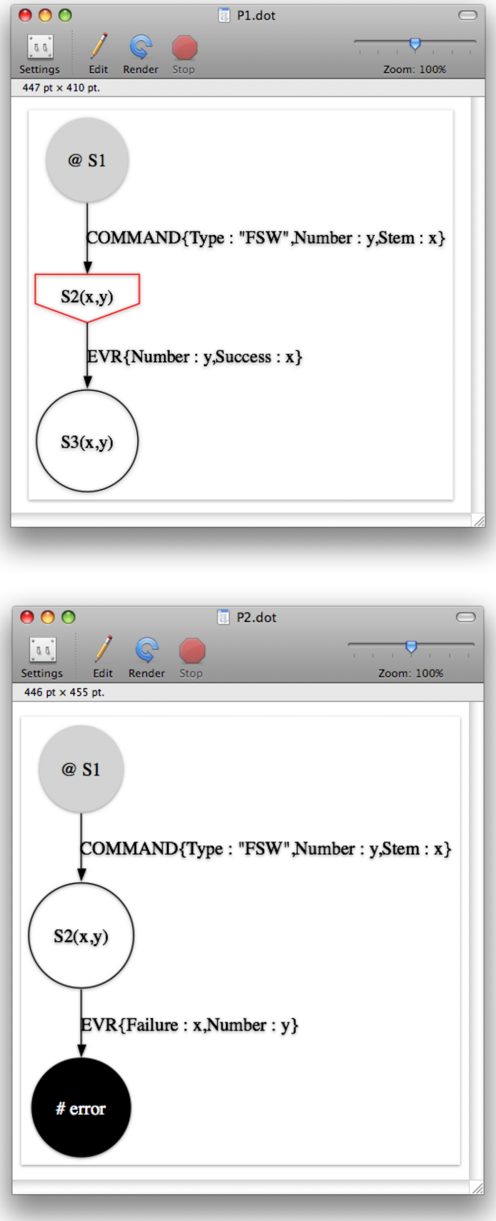

Figure 6.1: GraphViz view of `P1.dot` and `P2.dot`

```
=====================================
   monitoring new log of length 5:
=====================================

*** P2 violated: by event 4 in state:

  state S2(x,y) {
    EVR{Failure : x,Number : y} => error
  }
  with bindings: {'y': 231, 'x': 'PICT'}

by transition 1 : EVR{'Failure': 'PICT', 'Number': 231} => error

violating event:

EVR 4 {
  Failure := "PICT" - str
  OBJ_TYPE := "EVR" - str
  Number := 231 - int
}

--- error trace: ---

COMMAND 1 {
  OBJ_TYPE := "COMMAND" - str
  Type := "FSW" - str
  Number := 231 - int
  Stem := "PICT" - str
}

EVR 4 {
  Failure := "PICT" - str
  OBJ_TYPE := "EVR" - str
  Number := 231 - int
}

============================
       RESULTS FOR P1:
============================

Errors: 1

*** violated: in hot end state:

  state S2(x,y) {
    EVR{Number : y,Success : x} => S3(x,y)
  }
  with bindings: {’y’: 231, ’x’: ’PICT’}

--- error trace: ---

COMMAND 1 {
  OBJ_TYPE := "COMMAND" - str
  Type := "FSW" - str
  Number := 231 - int
  Stem := "PICT" - str
}

Statistics {
  COMMAND :
      {’Type’: ’FSW’, ’Number’: 231, ’Stem’: ’PICT’} -> 1
}

============================
       RESULTS FOR P2:
============================

Errors: 1

*** violated: by event 4 in state:

  state S2(x,y) {
    EVR{Failure : x,Number : y} => error
  }
  with bindings: {’y’: 231, ’x’: ’PICT’}

by transition 1 : EVR{’Failure’: ’PICT’, ’Number’: 231} => error

violating event:

EVR 4 {
  Failure := "PICT" - str
  OBJ_TYPE := "EVR" - str
  Number := 231 - int
}

--- error trace: ---

COMMAND 1 {
  OBJ_TYPE := "COMMAND" - str
  Type := "FSW" - str
  Number := 231 - int
  Stem := "PICT" - str
}

EVR 4 {
  Failure := "PICT" - str
  OBJ_TYPE := "EVR" - str
  Number := 231 - int
}

Statistics {
  COMMAND :
      {’Type’: ’FSW’, ’Number’: 231, ’Stem’: ’PICT’} -> 1
  EVR :
      {’Failure’: ’PICT’, ’Number’: 231} -> 1
}

========================
   Summary of Errors:
========================

P1  :  1 error
P2  :  1 error

specification was violated 2 times

5 events processed in 0 minutes and 0.00121 seconds (4128 events/sec)

Results are now being written to the file:
/Users/khavelun/Desktop/MSLRESULT/RESULTS

End of session!
```

We can now see that after event number 4 a violation of P2 is detected: the automaton for this property is in state `S2` – with parameters x=”PICT” and y=231, when the listed error transition fires: a failure of the `PICT` command. The *violating failure event* is listed, and then an *error trace* showing the sequence of event from the start of monitoring that drove the monitor into an error state, leaving out events that did not influence the monitor (in this case events 2 and 3).

At the end of monitoring the results for each property are listed. This includes two kind of errors:

- *safety properties*: error transitions caused by execution of transitions with `error` on the right hand side. This means that events that should not happen, happened.
- *liveness properties*: staying in a *hot* state at the end of the log. This means that some events that should happen, did not happen.

For property P2 the safety property violation that was reported above is repeated, including violating event and error trace. For property P1 a liveness property violation is reported. The automaton ends in the hot state S2 (see automaton generated from the pattern) with x bound to "PICT" and y bound to 231. The error trace illustrates the events that brought the automaton to this state.

Chapter 7

Specification Learning

Writing specifications can be difficult and time consuming, the key problem being to identify what properties to check - even just formulated in English prose. In some cases we may want to learn a specification from one or more "*well behaved*" log files, and then later turn this specification into a monitor to check subsequent log files as part of a regression test suite. LOGSCOPE provides functionality supporting learning of specifications from logs.

Two forms of learning is supported: *concrete learning* and *abstract learning*. During *concrete learning* LOGSCOPE learns the exact log files it sees up to equivalence on a set of field names that is provided to the learner, either a default set, or field names (for each kind of event) provided by the user. This results in very large automata, in principe representing the set of all logs seen during learning, but projected to the fields of interest. This closely resembles the process one would apply by comparing log files with UNIX's `diff` command after irrelevant event fields have been eliminated for example with UNIX's `grep` command. One will for example be able to perform some tests on the flight simulator, learning the spec, and then later go into the testbed and carry out the same test, now comparing the new test with the learned spec. In abstract learning mode LOGSCOPE learns high level properties about consequences of individual commands, yielding typically small human readable specification units.

Note: the abstract learning capability is not yet fully implemented and remains to be documented.

7.1 Concrete Learning

7.1.1 The API

The concrete Learner API consists of the class `ConcreteLearner`:

The concrete learner API

```
class ConcreteLearner:
  def __init__(self,automatonName : string,
                    fileName : string+None = None)
  def learnlog(self,log : list[dict])
  def dumpSpec(self,filename : string)
```

The constructor takes two arguments, the second of which is optional. The first argument is the name of the automaton that should be learned. The second argument indicates the name of a file. If the second file name argument is absent (or `None`), a new automaton will be constructed. If on the other hand the second file name argument is present (a string), the automaton will be read in from that file, and further refined. In the latter case the file must contain an automaton/pattern with the name given as the first argument.

The `learnlog` method takes as argument a log (list of events) and updates the specification accordingly, adding what it sees in the log. This method can be called repeatedly on different logs before the accumulated specification is finally written to a file with the `dumpSpec` method, taking an absolute file name as argument. This filename is what subsequently may be passed as the second argument to another call of the `ConcreteLearner` constructor in order to refine the specification through additional learning.

7.1.2 An Example Script

The following script illustrates a session using the `ConcreteLeaner` class.

```
import lsm.lsm as lsm

# create event logs, in this case hand-made for illustration;
# will typically be extracted with log file extractor:

log1 = [
  {"OBJ_TYPE" : "COMMAND", "Stem" : "PICT"},
  {"OBJ_TYPE" : "EVR", "Dispatch" : "PICT", "EventId" : 2,
      "Module" : "dispatcher", "Message" : "dispatch done!"},
  {"OBJ_TYPE" : "CHANNEL", "ChannelId" : 3, "DataNumber" : 5},
  {"OBJ_TYPE" : "EVR", "Success" : "PICT", "EventId" : 3,
      "Message" : "command succeeded!"},
  {"OBJ_TYPE" : "PRODUCT", "Name" : "Image", "ImageSize" : 1200}
]

log2 = [
 {"OBJ_TYPE" : "COMMAND", "Stem" : "PICT"},
 {"OBJ_TYPE" : "EVR", "Dispatch" : "PICT", "EventId" : 2,
     "Module" : "dispatcher", "Message" : "dispatch done!" },
 {"OBJ_TYPE" : "CHANNEL", "ChannelId" : 3, "DataNumber" : 6}, # change data number
 {"OBJ_TYPE" : "EVR", "Success" : "PICT", "EventId" : 3,
     "Message" : "command succeeded!"},
 {"OBJ_TYPE" : "PRODUCT", "Name" : "Image", "ImageSize" : 1200}
]

log3 = [
  {"OBJ_TYPE" : "COMMAND", "Stem" : "PICT"},
  {"OBJ_TYPE" : "EVR", "Dispatch" : "PICT", "EventId" : 2,
      "Module" : "dispatcher", "Message" : "dispatch done!"},
  # leave out CHANNEL
  {"OBJ_TYPE" : "EVR", "Success" : "PICT", "EventId" : 3,
      "Message" : "command succeeded!"},
  {"OBJ_TYPE" : "PRODUCT", "Name" : "Image", "ImageSize" : 1200}
]

log4 = [
  {"OBJ_TYPE" : "COMMAND", "Stem" : "PICT"},
  {"OBJ_TYPE" : "EVR", "Dispatch" : "PICT", "EventId" : 2,
      "Module" : "dispatcher", "Message" : "dispatch done!"},
  {"OBJ_TYPE" : "CHANNEL", "ChannelId" : 3, "DataNumber" : 5},
  {"OBJ_TYPE" : "EVR", "Failure" : "PICT", "EventId" : 3,
      "Message" : "***command failed!"},
  {"OBJ_TYPE" : "PRODUCT", "Name" : "Image", "ImageSize" : 1200}
]

# specify absolute path of where results should be stored
# (.dot files and RESULT file):

resultdir = "/Users/khavelun/Desktop/MSLRESULT"
lsm.setResultDir(resultdir)

# default fields learned from:
# - for COMMAND : "Stem"
# - for EVR : "EventId","Module","Message","EventNumber"
# - for CHANNEL : "ChannelId","Module","DataNumber"
# - for CHANGE : "ChannelId","Module","DataNumber"
# - for PRODUCT : "Name"

# specify if this should be changed:

lsm.fieldsEVR(["EventId","Module","Message"])

# learn a new automaton based on log1 and log2 in the same run:
```

```
learner = lsm.ConcreteLearner("LogPattern")
learner.learnlog(log1)
learner.learnlog(log2)
learner.dumpSpec(resultdir + "/LogPattern1.spec")

# refine the spec just stored in LogPattern1.spec by learning from
# a new log file. store result in LogPattern2.spec.

learner = lsm.ConcreteLearner("LogPattern",resultdir + "/LogPattern1.spec")
learner.learnlog(log3)
learner.dumpSpec(resultdir + "/LogPattern2.spec")

# monitor now a good (log1) and a bad (log4) log file:

obs = lsm.Observer(resultdir + "/LogPattern2.spec")
obs.monitor(log1)
obs.monitor(log4)
```

7.1.3 The Output Generated by LogScope/Learner

The learned specification is shown below. It describes the combined behavior of `log1`, `log2` and `log3`. Figure 7.1 shows the GraphViz view of the dot files of the two learned specifications (the script contains two learning sessions). The top automaton illustrates the result of learning from `log1` and `log2`, while the bottom automaton (equivalent to the one shown in textual format) illustrates the result of additionally learning from `log3`. States colored red have been learned during the most recent learning session. This can also be seen from the state names, which have the form `L<session number>_<state number>`, where the session number increases by 1 for each new learning session.

```
automaton LogPattern {
  step L0_1 {
    COMMAND{Stem : "PICT"} => L0_2
  }

  step L0_2 {
    EVR{EventId : 2,Message : "dispatch done!",
          Module : "dispatcher"} => L0_3
  }

  step L0_3 {
    CHANNEL{DataNumber : 5,ChannelId : 3} => L0_4
    CHANNEL{DataNumber : 6,ChannelId : 3} => L0_7
    EVR{EventId : 3,Message : "command succeeded!"} => L1_1
  }

  step L0_4 {
    EVR{EventId : 3,Message : "command succeeded!"} => L0_5
  }

  step L0_5 {
    PRODUCT{Name : "Image"} => L0_6
  }

  step L0_6 {}

  step L0_7 {
    EVR{EventId : 3,Message : "command succeeded!"} => L0_8
  }

  step L0_8 {
    PRODUCT{Name : "Image"} => L0_9
  }

  step L0_9 {}

  step L1_1 {
    PRODUCT{Name : "Image"} => L1_2
  }

  step L1_2 {}

  initial L0_1
  success L0_6,L0_9,L1_2
}
```

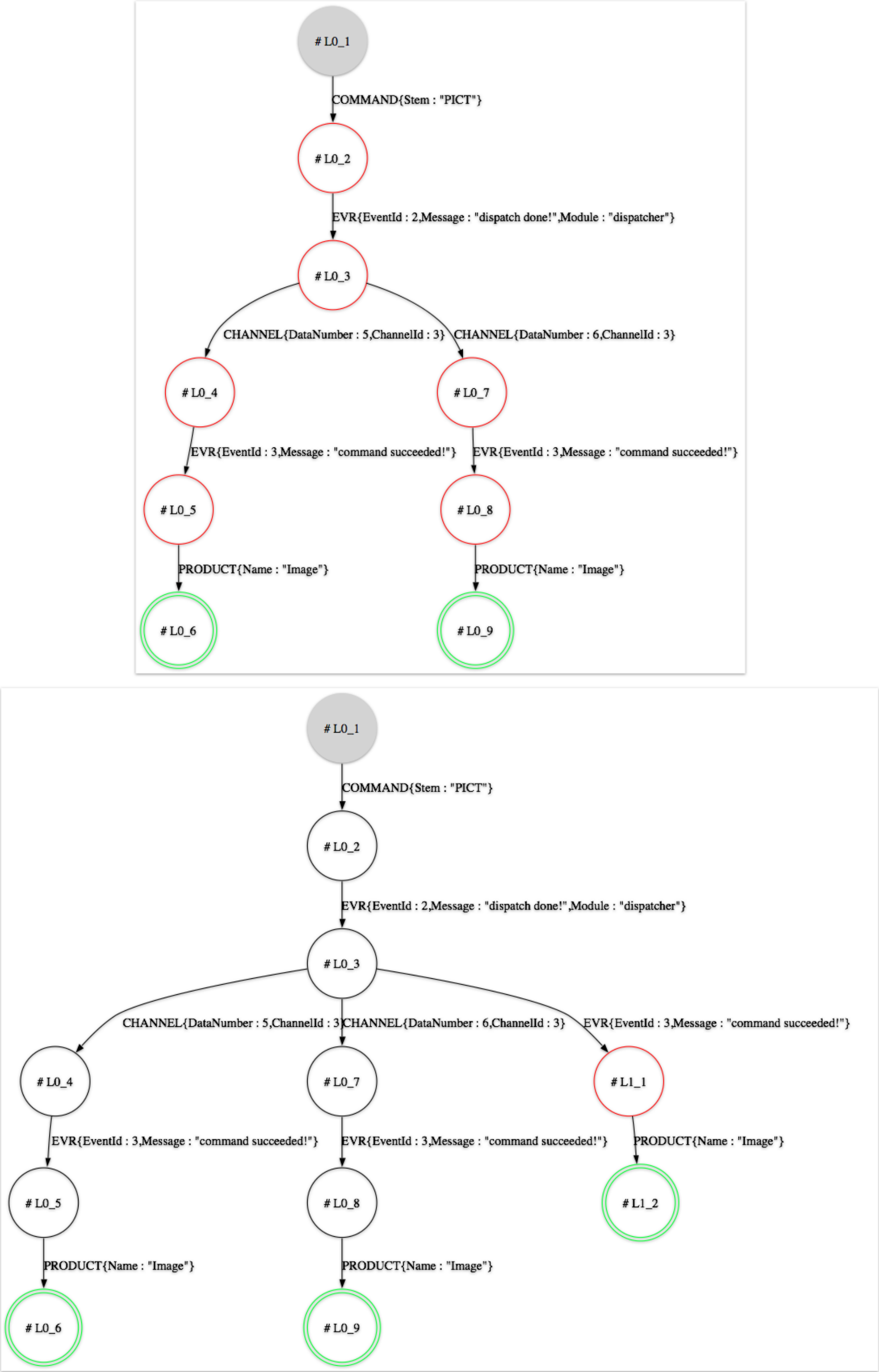
# L0_1
COMMAND{Stem : "PICT"}
# L0_2
EVR{EventId : 2,Message : "dispatch done!",Module : "dispatcher"}
# L0_3
CHANNEL{DataNumber : 5,ChannelId : 3}
CHANNEL{DataNumber : 6,ChannelId : 3}
# L0_4
# L0_7
EVR{EventId : 3,Message : "command succeeded!"}
EVR{EventId : 3,Message : "command succeeded!"}
# L0_5
# L0_8
PRODUCT{Name : "Image"}
PRODUCT{Name : "Image"}
# L0_6
# L0_9
# L0_1
COMMAND{Stem : "PICT"}
# L0_2
EVR{EventId : 2,Message : "dispatch done!",Module : "dispatcher"}
# L0_3
CHANNEL{DataNumber : 5,ChannelId : 3}
CHANNEL{DataNumber : 6,ChannelId : 3}
EVR{EventId : 3,Message : "command succeeded!"}
# L0_4
# L0_7
# L1_1
EVR{EventId : 3,Message : "command succeeded!"}
EVR{EventId : 3,Message : "command succeeded!"}
PRODUCT{Name : "Image"}
# L0_5
# L0_8
# L1_2
PRODUCT{Name : "Image"}
PRODUCT{Name : "Image"}
# L0_6
# L0_9

Figure 7.1: GraphViz view of `LogPattern1.spec.dot` and `LogPattern2.spec.dot`

7.2 Abstract Learning

TO BE WRITTEN.

Appendix A

LogScope/SL Grammar

```
/* --- lexical elements --- */

NAME = [a-zA-Z_.][a-zA-Z0-9_.]*

NUMBER = [0-9]+

STRING = "..."


/* --- specifications */


specification ::= [pythoncode] monitor+

pythoncode ::=
  ’:::’
   python code defining predicates, including imports
  ’:::’

monitor ::= [’ignore’] monitorspec

monitorspec ::= pattern | automaton


/* --- patterns --- */

pattern ::= ’pattern’ NAME ’:’ event ’=>’ consequence

consequence ::= [’!’] event
              | ’[’ consequence+, ’]’
              | ’{’ consequence+, ’}’
```

```
event ::= type ’{’ constraint*, ’}’ [’where’ predicate]

type ::= ’COMMAND’ | ’EVR’ | ’CHANNEL’ | ’CHANGE’ | ’PRODUCT’

constraint ::= NAME ’:’ range

range ::=  NUMBER
         | STRING
         | NAME
         | ’[’ NUMBER ’,’ NUMBER ’]’
         | ’{’ bitvalue*, ’}’

bitvalue ::= value ’:’ range

value ::= NUMBER | STRING

predicate ::= NAME ’(’ argument*, ’)’
           | ’|’ arbitrary boolean valued python expression ’|’
           | predicate ’and’ predicate
           | predicate ’or’ predicate
           | ’not’ predicate
           | ’(’ predicate ’)’

argument ::= NUMBER | STRING | NAME


/* --- automata --- */

automaton ::=
  ’automaton’ NAME ’{’
     state*
     [’initial’ action+]
     [’hot’ name+,]
     [’success’ name+,]
  ’}’

state ::= modifier* statekind NAME [’(’ name+, ’)’] ’{’ rule* ’}’

modifier ::= HOT | INITIAL

statekind ::= ’always’ | ’state’ | ’step’

rule ::= event ’=>’ action+,

action ::= NAME [’(’ argument+, ’)’] | ’done’ | ’error’
```

Appendix B

Predefined Predicates

```
### built-in predicates: ###

def less(x,y):
    return x < y

def equal(x,y):
    return x == y

def less_equal(x,y):
    return x <= y

def greater(x,y):
    return x > y

def greater_equal(x,y):
    return x >= y

def contains(s,t):
    return s.find(t) != -1

### shorter versions: ###

lt = less
eq = equal
le = less_equal
gt = greater
ge = greater_equal
```

Bibliography

[1] H. Barringer, D. Rydeheard, and K. Havelund. Rule Systems for Run-Time Monitoring: from Eagle to RuleR. In *Proc. of the 7th International Workshop on Runtime Verification (RV'07)*, volume 4839 of *LNCS*, Vancouver, Canada, 2007. Springer.

[2] H. Barringer, D. Rydeheard, and K. Havelund. Rule Systems for Run-Time Monitoring: from Eagle to RuleR. *Journal of Logic and Computation*, 2008. Extended version of [1], to appear.

[3] H. Barringer, D.E Rydeheard, and K. Havelund. RuleR: A Tutorial Guide. Available at: `http://www.cs.man.ac.uk/˜howard/LPA.html`, 2008.

[4] F. Chen and G. Roşu. MOP: An Efficient and Generic Runtime Verification Framework. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'07)*, 2007.

[5] S. Eckmann, G. Vigna, and R. A. Kemmerer. STATL Definition. Reliable Software Group, Department of Computer Science, University of California, Santa Barbara, CA 93106, June 2001.

[6] GraphViz. `http://www.graphviz.org`.

[7] K. Havelund. Runtime Verification of C Programs. In *Proc. of the 1st TestCom/FATES conference*, volume 5047 of *LNCS*, Tokyo, Japan, June 2008. Springer.

[8] K. Havelund and G. Roşu. Efficient Monitoring of Safety Properties. *Software Tools for Technology Transfer*, 6(2):158–173, 2004.

[9] M. Kim, S. Kannan, I. Lee, and O. Sokolsky. Java-MaC: a Run-time Assurance Tool for Java. In *Proc. of the 1st International Workshop on Runtime Verification (RV'01)*, volume 55(2) of *ENTCS*. Elsevier, 2001.

[10] M. Smith and K. Havelund. Requirements Capture with RCAT. In *16th IEEE International Requirements Engineering Conference (RE'08)*, IEEE Computer Society, Barcelona, Spain, September 2008.

[11] Runtime Verification. `http://www.runtime-verification.org`.