

# Specification-based Monitoring in C++

Klaus Havelund\*

Jet Propulsion Laboratory  
California Institute of Technology  
Pasadena, California, USA

**Abstract.** Software systems cannot in general be assumed proven correct before deployment. Testing is still the most common approach to demonstrate a satisfactory level of correctness. However, some errors will survive verification efforts, and it is therefore reasonable to monitor a system after deployment, to determine whether it executes correctly. Both for testing and post-deployment monitoring, it may be desirable to be able to formalize correctness properties that can be monitored against program executions. This is also referred to as runtime verification. We present a specification language and a monitoring system for monitoring such specifications against event streams. The monitoring engine front-end, written in Scala, translates the specification to C++, whereas the back-end (the monitoring engine), written in C++, interprets the generated C++ monitor on an event stream. This makes it feasible to monitor the execution of C and C++ programs online.

## 1 Introduction

The correctness of software is usually demonstrated through extensive testing. A test suite usually consists of test cases, where each test case consists of a test input vector and a *test oracle*, which determines whether the test case executes properly. Since testing does not provide 100% coverage of all execution paths, there is also a need after deployment to monitor the software as it executes. For this the concept of *monitors* is needed. We present LOGSCOPE (available at [24]), a system for monitoring event streams (traces) against formal specifications, also referred in literature as Runtime Verification (RV). A formal specification, written by a user, is translated into a monitor in C++, which can be used both as a test oracle before deployment, or for monitoring the system as it executes after deployment. A monitor generated from a formal specification is event-driven. It receives events, one by one, modifying its internal state for each observed event, and emitting an error message or calling a callback function in case a violation of the specification is encountered. Such a system can be used *offline*, analyzing log files, or *online*, monitoring the system real-time as it executes. The system solves the following problem:

---

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

*Given a sequence of events emitted from a software application, how can we assure that the event stream satisfies a set of desired properties?*

The *frontend*, written in the SCALA programming language, parses a specification and translates it to an Abstract Syntax Tree (AST) in C++. The *backend*, the monitoring engine itself, written in C++, imports the C++ AST generated by the frontend, and interprets it over a sequence of events emitted to it. The fact that the backend is implemented in C++ makes it possible to monitor applications online that themselves are written in C or C++, which is important for embedded systems. It is for example unthinkable to call a monitor in JAVA or PYTHON from a C/C++ program in an embedded application, unless monitoring is performed remotely and asynchronously. Numerous RV systems have been developed in the past, as discussed in Section 2, but most are implemented in high-level languages such as JAVA. 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.

The specification language allows to state properties about events that can carry data, e.g. relating data occurring in events at different positions in the trace. We refer to this as *parametric monitoring*. What makes the specification language stand out compared to most related work is that it combines a rule-based language and a state machine language. A rule-based system consists of a set of rules operating on a set of facts in a memory, where a fact is a named data record. Each rule has a condition and an action. The condition of a rule can refer to the memory and query whether certain facts are present or not, including the data they carry, which is used to support parametric monitoring. The action of a rule can add or delete facts from the memory. As such several facts can be active at any moment in time. The ability to refer to the presence/absence of facts can be used to model past time temporal properties by letting facts be generated when certain events occur that need to be remembered in the future. LOGSCOPE is fundamentally a rule-based system but its specification language has a state machine look and feel by supporting the definition of event triggered transitions as part of fact definitions.

The paper is organized as follows. Section 2 discusses related work. Section 3 provides an overview of the tool. Section 4 explains the notation through a series of examples. Section 5 outlines how to use the tool. Section 6 briefly outlines aspects of the implementation. Section 7 describes experimentation performed. Finally, Section 8 concludes the paper. Appendix A contains visualizations of the monitors presented throughout the paper, generated by LOGSCOPE.

## 2 Related Work

LOGSCOPE's concepts can be traced back to the early RULER system [2], where the idea of merging rule-based programming with state machine notation was first explored, and implemented in JAVA. That work was followed by the first ver-

sion of LOGSCOPE<sup>1</sup> [4, 23], implemented and executing monitors in PYTHON, in contrast to the here presented system where monitors are generated in the higher performing C++. The algorithmic approach, however, was the same, namely parsing a monitor specification into an Abstract Syntax Tree (AST) and then *interpreting* it over the trace, as we shall discuss later on. The earlier system did not allow to query the fact memory for presence/absence of facts. It was, however, richer in a number of other ways, including providing a temporal logic layer translated to the automaton layer, and allowing PYTHON code to be written as part of monitors, such as in e.g. conditions and actions. The here presented version does not yet offer similar capabilities for including C++ code in monitor specifications.

We have developed other runtime verification systems based on the same idea of merging rule-based programming with state machines, including DAUT [17, 10] (in SCALA) and PYCONTRACT [8, 27] (in PYTHON). Both are so-called *internal* DSLs, in contrast to LOGSCOPE, which is an *external* DSL with its own grammar and parser. In an internal DSL monitors are written directly in the host language, which we in Section 7 shall see can have important impact on performance. We also developed the purely rule-based runtime verification system LOGFIRE [18, 22], also an internal SCALA DSL, based on the RETE algorithm [13, 12] traditionally used in expert systems, in order to explore the applicability of this algorithm for runtime verification. The RMOR monitoring framework for C [16] generates C code from an external state machine DSL that includes aspect-oriented programming for instrumenting code to be monitored.

Numerous other RV systems have been developed over time providing external DSLs. Many are written in high-level application programming languages, such as e.g. JAVA, SCALA, and OCAML, which makes them less suited for online monitoring of embedded systems. JAVA-MAC [21] was an early system supporting a past time temporal logic, allowing for a clear separation between the definition of the primitive events of a system and the system properties. It enables automatic instrumentation of JAVA code to generate events for the monitor. JAVA-MAC does not support parametric monitoring, however.

A number of systems support efficient parametric monitoring through slicing: the idea of splitting the trace of events carrying data into several subtraces of propositional events, each of which is then submitted to a propositional monitor. MOP [25] offers several data parametric specification formalisms as separate plugins, including state machines, past and future temporal logics, regular expressions, grammars, etc. The logics are separated in the sense that any property is expressed in one of the logics. The parameterization is based on slicing, which is very efficient, but which offers a somewhat limited expressiveness. MOP supports automated code instrumentation using aspect-oriented programming (via ASPECTJ). The QEA system [28] is based on extended finite state machines, and improves the expressiveness of the slicing approach compared to MOP by allowing so-called free variables that can be updated in the monitors. LARVA [6]

---

<sup>1</sup> The original system was focused on Log analysis, hence the name LOGSCOPE (Scope as telescope).

offers a specification language for writing Dynamic Automata with Events and Timers (DATEs), also a form of extended finite state machines, and similar to timed automata enriched with stopwatches. It supports a basic form of parametric monitoring with slicing. Various other specification formalisms are translated into DATEs. DATEs can communicate via channels and global variables. The system also supports automated code instrumentation using aspect-oriented programming (via ASPECTJ).

A different branch of formalisms include those supported in stream-based systems. LOLA [9] is a synchronous stream-based language which allows the user to specify the properties of a program in past and future linear time temporal logic. The language guarantees bounded memory to perform online monitoring, but differs from most other synchronous languages in that it is able to refer to future values in a stream. It allows the user to collect statistics at runtime and to express numerical queries. The COPILOT specification language [26] is an internal HASKELL DSL from which monitors in C are generated for monitoring hard real-time reactive systems. It supports a past time temporal logic and a bounded future time temporal logic, both mapped into stream expressions. It supports data parameterization, which is bounded due to the real-time constraints requiring statically bounded execution time and memory usage.

Closely related are systems resembling variants of the linear  $\mu$ -calculus, using recursion. EAGLE [3] implements a recursive data parametric calculus with past and future time operators. HAWK [7] extends EAGLE with constructs for capturing parameterized program events such as method calls and method returns. Parameters can be executing thread, the objects that methods are called upon, arguments to methods, and return values. The tool automates program instrumentation of JAVA programs (via ASPECTJ). DETECTER [1] implements a future time data parametric Hennessy-Milner logic with recursion for monitoring ERLANG programs.

Several systems have been developed specifically supporting forms of first-order linear time temporal logic as the core logic. MONPOLY [5] supports a first-order linear time temporal logic with future and past time temporal operators. The logic also supports aggregation operators (e.g., sum and average), increasing the expressiveness of the logic. BEEPBEOP [15] permits writing first-order linear time temporal logic properties over the data in a trace of XML messages. In [11] is presented a framework that lifts monitor synthesis procedures for propositional temporal logics to a temporal logic over structures within a given first-order theory. To evaluate such specifications, SMT solving and classical monitoring of propositional temporal properties are combined. DEJAVU [19] supports a first-order past linear time temporal logic and represents data occurring in events with BDDs. In [20] is described an extension that augments that logic with rules.

### 3 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. 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
  }
}
```

Figure 1 illustrates the architecture of LOGSCOPE. A monitor specification, written in the SCOPE specification language by a user, is by the **frontend** (written in the SCALA programming language) translated to an AST in C++, representing the structure of the specification, and 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  $E_1, E_2, \dots$  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`.

For each monitor in `contract.cpp` is maintained an internal memory, called the *frontier*, which is a set of active *states*  $S_1, S_2, \dots, S_k$ . States in LOGSCOPE are similar to those of state machines, however, in contrast to traditional state machines the frontier can contain more than one state, each parameterized with its own data. 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  $S_1, S_2, \dots, S_k$  in the frontier<sup>2</sup>, causing states to be removed, states to be added, and/or error messages to be issued.

<sup>2</sup> Optimizations similar to slicing can avoid examining all states.



## 4.1 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:

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

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 5, we shall see how such events are concretely created with the backend C++ API.

## 4.2 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*  $\Rightarrow$  *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**.

Textual monitors are visualized by LOGSCOPE using GRAPHVIZ’s dot-format, see Appendix A. This can help in convincing the specification writer that the specification expresses the intended property.

```

monitor M1 {
  event command, succeed

  init Command {
    command ⇒ Succeed
  }

  hot Succeed {
    succeed ⇒ Command
    command ⇒ error
  }
}

```

Fig. 2: Monitor M1 for property  $P_1$ .

### 4.3 Some Alternative Monitors

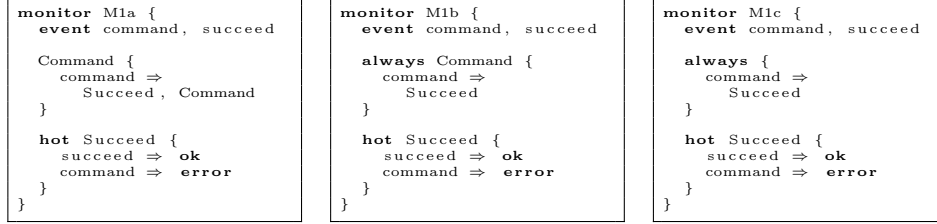
Figure 3 shows some alternative monitors for property  $P_1$ , illustrating different aspects of the language. In the monitor M1a, instead of alternating between the states **Command** and **Succeed**, whenever a **command** event is observed in the **Command** state, in addition to creating a **Succeed** 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.

### 4.4 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 traditional





(a) Monitor M1a.

(b) Monitor M1b.

(c) Monitor M1c.

Fig. 3: Alternative monitors M1a, M1b, and M1c for property  $P_1$ .

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:

- `command(name : c, kind : k)` : command being issued
- `succeed(name : c)` : successful termination

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 4. 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<sup>3</sup>. The anonymous initial **always** state contains a single transition, which on the left-hand side of the arrow  $\Rightarrow$  matches any `command` event where the `kind` field is the string "FSW". On such a match the command

<sup>3</sup> An extension of the language can allow different types of values.

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  $\text{command}(\text{name} : \text{"TURN"}, \text{kind} : \text{"FSW"})$  is observed, then a  $\text{Succeed}(c : \text{"TURN"})$  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**.

The general rule for when a variable name in a transition is *bound* versus *matched against* is the following. A variable, such as  $x$  in the **always** state, is *bound* to an incoming value of an event if  $x$  is not occurring as parameter to the state. A variable, such as  $c$  in the **Succeed** state, is *matched against* if occurs as parameter to the state.

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 {
  event command(cmd, kind), succeed(c)

  always {
    command(cmd : x, kind : "FSW")  $\Rightarrow$  Succeed(c : x)
  }

  hot Succeed(c) {
    succeed(cmd : c)  $\Rightarrow$  ok
    command(cmd : c)  $\Rightarrow$  error
  }
}
```

Fig. 4: Monitor M2 for property  $P_2$ .

#### 4.5 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 5 monitors this property. The monitor is the same as in Figure 4 except that in the initial **always** state we have added an extra transition:

$$\text{succceed}(\text{name} : x) \text{ @ } !\text{Succeed}(c : x) \Rightarrow \text{error}$$

In addition to the event pattern `succceed(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 {
  event command(cmd, kind), succceed(cmd)

  always {
    command(cmd : x, kind : "FSW")  $\Rightarrow$  Succeed(c : x)
    succceed(cmd : x) @ !Succeed(c : x)  $\Rightarrow$  error
  }

  hot Succeed(c) {
    succceed(cmd : c)  $\Rightarrow$  ok
    command(cmd : c)  $\Rightarrow$  error
  }
}
```

Fig. 5: Monitor M3 for property  $P_3$ .

## 4.6 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)
- cancel(name : c, nr : n)
- dispatch(name : c, nr : n)
- fail(name : c, nr : n)
- succeed(name : c, nr : n)
- close(name : c, nr : n)

```

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 6. 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 software command instance with the name `sc` will cause an error in this state, we don't care what the command number is. In this monitor, the 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
```

Whether we use a don't care symbol  $\_$  or not does, however, have an effect in case we do not declare our events at the beginning of the monitor: 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 {
  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")  $\Rightarrow$  Dispatch(dc : c, dn : n)
    succeed(cmd : c, nr : n) @ !Succeed(sc : c, sn : n)  $\Rightarrow$  error
  }

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

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

  NoMoreSuccess(nc,nn) {
    succeed(cmd : nc, nr : nn)  $\Rightarrow$  error
  }

  hot Close(cc,cn) {
    close(cmd : cc, nr : cn)  $\Rightarrow$  ok
  }
}

```

Fig. 6: Monitor M4 for property  $P_4$ .

#### 4.7 The Complete Grammar

The complete grammar for the SCOPE language is defined in Figure 7. In addition to traditional grammar notation we also use  $A^*$  denoting  $A$  zero or more times, separated by commas, and likewise for  $A^+$  denoting  $A$  one or more times, separated by commas.

Note that one does not need to define events. In that case the events are inferred from the state transitions. Providing event definitions, however, 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 explained below.

The non-terminal ‘Modifier’ introduces two modifiers we have not explained before. A **step** state will be deleted from the frontier at the next event if none of

its transitions fire. A **next** state will cause an error at the next event if none of its transitions fire. In the definition of the non-terminal ‘Trans’ (transition), if a pattern on the right-hand side of  $\Rightarrow$  is negated, the corresponding state is removed. Such patterns must be grounded with no undefined identifiers. Note that when taking a transition, the source state containing the transition is removed from the frontier, unless the state is annotated with the **always** modifier.

```

Specification ::= Monitor*
Monitor ::= monitor Id '{' EventDef* State* '}'
EventDef ::= event Event,+
Event ::= Id [ '(' Id,+ ')' ]
State ::= Modifier+ '{' Trans+ '}' | Modifier* Id [ '(' Id,* ')' ] [ '{' Trans+ '}' ]
Modifier ::= init | always | hot | step | next
Trans ::= Pat [ '!' Pat,+ ] '=>' Pat,*
Pat ::= [ '(' ] 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

```

Fig. 7: Grammar for the LOGSCOPE language.

## 5 Usage

The front-end is delivered as a jar-file and a script referring to the jar-file. The **logscope** script is applied as follows to  $n$  files (for  $n \geq 1$ ) containing monitor specifications in the SCOPE language:

$$\text{logscope } \langle file_1 \rangle \dots \langle file_n \rangle$$

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. In case the input specification passes the parsing and type checking, LOGSCOPE will generate a directory **tool-generated**, containing the file **contract.cpp** with the generated C++ AST (which will be used to monitor the specified monitors) and visualizations of monitors in **.png** format, one for each monitor.

The user writes a program, say **main.cpp**, which is compiled with the generated **contract.cpp** file, and which instantiates the specification in **contract.cpp** and feeds it events. Figure 8 shows a program, **main.cpp**, using the **contract.cpp** file (by importing **contract.h**) that is generated by the frontend from the monitor specification in Figure 5.

In line 4 we create a `SpecObject` object via a call of the `makeContract` function, that the generated file `contract.cpp` defines. The `SpecObject` contains the AST of the specification as a C++ object tree. We then create a trace, a list of four events, in lines 5-10. In our case, the specification in Figure 5 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 6, 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 11-13 iterates through each event `e` in the trace, and feeds it to the contract via a call of the `eval` method. Finally, in line 14, we end the monitoring. This can cause additional error messages to be produced in case any hot states remain in the frontier of monitor. Note that during online monitoring this method may never be called. As shown in this example, the monitor must be fed events `e` via calls of the form `contract.eval(e)`. For the purpose of a simple presentation we created an explicit trace. When monitoring an application in *online* mode (as it executes) it is up to the implementer of the application to invoke these calls. When monitoring an application in *offline* mode, it is up to the implementer of the application to write events to a file, which can then later be processed.

```

1  #include "contract.h"
2
3  int main() {
4      SpecObject contract = makeContract();
5      list<Event> events = {
6          Event(10,"command",{{"cmd","TURN"},{"kind","FSW"}}),
7          Event(20,"command",{{"cmd","TRACK"},{"kind","FSW"}}),
8          Event(30,"succeed",{{"cmd","TURN"}}),
9          Event(40,"succeed",{{"cmd","SEND"}}),
10     };
11     for (Event &e : events) {
12         contract.eval(e);
13     };
14     contract.end();
15 }

```

Fig. 8: The main program in `main.cpp`.

Monitoring the above trace causes two errors to be detected, the first due to the 4th event (line 9 in Figure 8), which is a success of a command that has not been issued. At the end of monitoring (at the call `contract.end()` in line 14), an additional error is detected, indicating that the monitor M3 ends in the **hot** Succeed state, since the TRACK command has not succeeded.

## 6 Implementation

The front-end, implemented in SCALA, parses the SCOPE specification using SCALA's parser combinator library and produces an AST in SCALA. The AST is

type checked, and then written out as an AST in C++ to the file `contract.cpp`. Monitors are visualized with GRAPHVIZ [14]. The backend is implemented in C++14 (the 2014 version).

The monitoring engine can be conceived as operating with objects of three main classes. An object of the `SpecObject` class represents the specification, which is a collection of monitors, each of which is represented by an object of the `MonitorObject` class. Each such monitor, at any moment in time during monitoring, contains a set of active states, each represented by an object of the `StateObject` class. Each of these three classes defines an `eval(Event &e)` method, evaluating a single event, as shown in Figure 9.

```

void SpecObject::eval(Event &e) {
    for (MonitorObject *m : monitors) m->eval(e);
}

void MonitorObject::eval(Event e) {
    if (isRelevant(e)) {
        event = e;
        statesToDelete.clear();
        statesToAdd.clear();
        for (auto state : states) state.eval();
        for (auto &state : statesToDelete) states.erase(state);
        for (auto &state : statesToAdd) states.insert(state);
    }
}

void StateObject::eval() {
    bool fired = false;
    for (ast::Transition *trans : stateAST->transitions) {
        TransitionResult result = evalTransition(trans);
        if (result == TransitionResult::FAILED) error();
        fired = fired || (result != TransitionResult::SILENT);
    };
    if (stateAST->isNext && !fired) error();
    if (!stateAST->isAlways &&
        (stateAST->isNext || stateAST->isStep || fired)) {
        monitor->recordStateToDelete( *this );
    }
}

```

Fig. 9: Evaluation methods.

The `SpecObject::eval` method calls the `eval` method on each of its monitors. The method `MonitorObject::eval` only handles events relevant to the monitor (either declared, or if no event declarations: used). It calls the `eval` method on each of its states. It operates on two variables storing states to be deleted and states to be added due to the effect of the transitions in the monitor. Finally, the `StateObject::eval` method walks through the transitions. Next-states must fire. The state is removed if it is not an always state, and it is a next or step state, or a transition fired.



## 7 Experiment

### 7.1 Setting Up the Experiment

In order to evaluate LOGSCOPE’s performance, we compared it to two other monitoring systems: DAUT [17, 10] and PYCONTRACT [8, 27]. As previously mentioned, while the LOGSCOPE specification language is an external DSL – a stand-alone language, where a monitor specification is *translated* to an AST (in C++) and then interpreted; both DAUT and PYCONTRACT are internal DSLs, in respectively SCALA and PYTHON, implementing automata concepts very similar to the one of LOGSCOPE. An internal DSL is effectively a library in the host language. A user is therefore not limited to the expressiveness of the DSL, but can use the entire host language for writing monitors if needed. This can e.g. be useful for performing computations on the data appearing in events. Generally, high-level languages such as PYTHON, and especially SCALA, are well suited for defining internal DSLs. Both internal DSLs use the host language’s pattern matching to match events against transition left hand sides. This turns out to be a crucial difference from the C++ implementation of LOGSCOPE, where this pattern matching had to be implemented. The aging earlier version of LOGSCOPE [4, 23], implemented in PYTHON, and executing in PYTHON, was not performing well, and has for this reason been eliminated from the numeric comparison.

We monitored a slight modification of property M4 in Figure 6, shown in Figure 10, which requires less memory, avoiding the accumulating memory needed by the NoMoreSuccess state for example. The DAUT version of this property is shown in Figure 11 and the PYCONTRACT version is shown in Figure 12. We applied each monitor to 8 logs (traces) with varying length and shape. Table 1 shows the execution times in seconds and milliseconds. Each log is generated by an artificial log generator developed specifically for the experiment, parameterized with a shape  $S = (P, R)$  of positive integers, where  $P$  denotes how many commands are active in parallel, from issuing the command, dispatching, succeeding, and closing (four events), and  $R$  indicates how many times such a parallel execution is repeated.

### 7.2 Result and Interpretation

The surprise here is that LOGSCOPE, with a backend implemented in C++, is slower than the other two systems implemented in SCALA and PYTHON. The SCALA version has the best performance, which in part likely can be contributed to the JVM’s just-in-time compilation. We believe the reason for LOGSCOPE’s weaker performance to be the following. LOGSCOPE supports an external DSL, where a monitor specification is translated (by the SCALA frontend) into an AST (in C++). This AST is subsequently during monitoring *interpreted* on the stream of input events. The interpreter includes our own implementation of *pattern matching* of events against transition left-hand sides. In contrast, both DAUT and PYCONTRACT are internal shallow DSLs, where monitors are written directly in the host language, and specifically using the pattern matching provided

by the host languages, yielding an advantage wrt. performance. Note that this performance advantage of internal shallow DSLs also holds when compared to interpretation of *internal deep* DSLs, which are similar to external DSLs, in that a program/specification in the DSL is represented as a data structure (AST). Note finally that the C++ implementation uses dynamic memory allocation (as do DAUT and PYCONTRACT). This should in a revised version for embedded software monitoring be replaced by a static memory pool, potentially providing efficiency improvements wrt. time and memory.

The presented C++ version of LOGSCOPE, however, performs better than the earlier version described in [4, 23]. This is perhaps not so surprising since they both use fundamentally the same technique of interpreting the monitor specification AST over the trace, and C++ is a high performance programming language compared to Python.

```

monitor M4_modified {
  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")  $\Rightarrow$  Dispatch(dc : c, dn : n)
  }

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

  hot Succeed(sc,sn) {
    succeed(cmd : sc, nr : sn)  $\Rightarrow$  Close(cc : sc, cn : sn)
    command(cmd : sc, nr : _, kind : "FSW")  $\Rightarrow$  error
    fail(cmd : sc, nr : sn)  $\Rightarrow$  error
  }

  hot Close(cc,cn) {
    succeed(cmd : cc, nr : cn)  $\Rightarrow$  error
    close(cmd : cc, nr : cn)  $\Rightarrow$  ok
  }
}

```

Fig. 10: Monitor M4 modified, used in experiment.

## 8 Conclusion

We have presented a framework in C++ for monitoring event streams, with a frontend written in SCALA supporting an external DSL. We compared the implementation with two other monitoring systems implemented in respectively SCALA and PYTHON. It turns out that those systems perform better, likely due to the fact that they are internal DSLs benefiting from the host language's pattern matching constructs.

```

class M4_modified extends Monitor[Event] {
  always {
    case Com(c, n, "FSW") => Dispatch(c, n)
  }

  case class Dispatch(cmd:String, nr:String) extends state {
    hot {
      case Can('cmd', 'nr') => ok
      case Dis('cmd', 'nr') => Succeed(cmd, nr)
    }
  }

  case class Succeed(cmd:String, nr:String) extends state {
    hot {
      case Suc('cmd', 'nr') => Close(cmd, nr)
      case Com('cmd', -, "FSW") => error()
      case Fai('cmd', 'nr') => error()
    }
  }

  case class Close(cmd:String, nr:String) extends state {
    hot {
      case Suc('cmd', 'nr') => error()
      case Clo('cmd', 'nr') => ok
    }
  }
}

```

Fig. 11: Monitor M4 modified, in the SCALA DSL DAUT.

Table 1: Result of experiment. For each log is shown shape  $S$  of the log: a pair  $(P, R)$  of constants, where  $P$  denotes how many commands are active in parallel, from issuing the command, dispatching, succeeding, and closing (four events), and  $R$  indicates how many times such a parallel execution is repeated. The number of events in a log is  $4 * P * R$ .

log nr.	1	2	3	4	5	6	7	8
$S=(P,R)$	(1,12500)	(50,250)	(1,50000)	(5,10000)	(10,5000)	(20,2500)	(1,125000)	(5,25000)
nr. of events	50,000	50,000	200,000	200,000	200,000	200,000	500,000	500,000
LOGSCOPE	5.172 s	42.974 s	20.571 s	14.884 s	21.857 s	35.537 s	23.471 s	37.256
PYCONTRACT	0.373 s	4.600 s	1.567 s	3.425 s	5.120 s	8.811 s	4.511 s	9.483 s
DAUT	0.595 s	0.733 s	0.927 s	0.938 s	1.000 s	1.221 s	1.482 s	1.431 s

The current version of the tool is a prototype. A continuation of this work would include the following activities. There is a need to obtain a better understanding of the performance differences. In general, there is a need to modify the backend C++ code to match embedded programming practices. This includes most importantly to avoid dynamic memory allocation, using instead statically sized object pools. The backend monitoring engine can be further optimized. One can e.g. use indexing to quickly identify what states an event is relevant for. Various extensions to the specification language can be considered, such as e.g. allowing the use of C++ code as part of specifications. Currently arguments to a state are passed by referring to the formal parameter names. Allowing positional arguments without referring to the formal parameter names might be desirable. Finally, a more optimal way to translate a monitoring specification is to translate

```

class M4_modified(Monitor):
    def transition(self, event):
        match event:
            case {'name': 'command', 'cmd': c, 'nr': n, 'kind': "FSW"}:
                return self.Dispatch(c, n)

    @data
    class Dispatch(HotState):
        cmd: str
        nr: str

        def transition(self, event):
            match event:
                case {'name': 'cancel', 'cmd': self.cmd, 'nr': self.nr}:
                    return ok
                case {'name': 'dispatch', 'cmd': self.cmd, 'nr': self.nr}:
                    return self.Succeed(self.cmd, self.nr)

    @data
    class Succeed(HotState):
        cmd: str
        nr: str

        def transition(self, event):
            match event:
                case {'name': 'succeed', 'cmd': self.cmd, 'nr': self.nr}:
                    return self.Close(self.cmd, self.nr)
                case {'name': 'command', 'cmd': self.cmd, 'nr': _, 'kind': "FSW"}:
                    return error()
                case {'name': 'fail', 'cmd': self.cmd, 'nr': self.nr}:
                    return error()

    @data
    class Close(HotState):
        cmd: str
        nr: str

        def transition(self, event):
            match event:
                case {'name': 'succeed', 'cmd': self.cmd, 'nr': self.nr}:
                    return error()
                case {'name': 'close', 'cmd': self.cmd, 'nr': self.nr}:
                    return ok

```

Fig. 12: Monitor M4 modified, in the PYTHON DSL PYCONTRACT.

it to a C++ program that is specialized for the monitor (rather than translating it to an AST in C++). Partial evaluation could be considered as an approach to achieve this, partially evaluating the interpreter given a specification, yielding a program that now only takes the event trace as input.

## References

1. Duncan Paul Attard, Ian Cassar, Adrian Francalanza, Luca Aceto, and Anna Ingólfssdóttir. A runtime monitoring tool for actor-based systems. In Simon Gay and Antonio Ravara, editors, *Behavioural Types: from Theory to Tools*, chapter 3, pages 49–76. River Publishers, 2017.
2. 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.

3. Howard Barringer, Allen Goldberg, Klaus Havelund, and Koushik Sen. Rule-based runtime verification. In *VMCAI*, volume 2937 of *LNCS*, pages 44–57. Springer, 2004.
4. 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.
5. David A. Basin, Felix Klaedtke, Srdjan Marinovic, and Eugen Zălinescu. Monitoring of temporal first-order properties with aggregations. *Formal Methods in System Design*, 46(3):262–285, 2015.
6. Christian Colombo, Gordon J. Pace, and Gerardo Schneider. LARVA — safer monitoring of real-time Java programs (tool paper). In *Proceedings of the 2009 Seventh IEEE International Conference on Software Engineering and Formal Methods*, SEFM ’09, pages 33–37, Washington, DC, USA, 2009. IEEE Computer Society.
7. Marcelo d’Amorim and Klaus Havelund. Event-based runtime verification of Java programs. In *Proceedings of the Third International Workshop on Dynamic Analysis*, WODA ’05, page 1–7, New York, NY, USA, 2005. Association for Computing Machinery.
8. Dennis Dams, Klaus Havelund, and Sean Kauffman. A Python library for trace analysis. In *Runtime Verification - 22nd Int. Conference, RV’22, Tbilisi, Georgia, September 28-30, 2022. Proceedings*, LNCS. Springer, 2022.
9. Ben D’Angelo, Sriram Sankaranarayanan, César Sánchez, Will Robinson, Bernd Finkbeiner, Henny B. Sipma, Sandeep Mehrotra, and Zohar Manna. LOLA: Runtime monitoring of synchronous systems. In *Proceedings of TIME 2005: the 12th International Symposium on Temporal Representation and Reasoning*, pages 166–174. IEEE, 2005.
10. Daut. <https://github.com/havelund/daut>, 2022.
11. Normann Decker, Martin Leucker, and Daniel Thoma. Monitoring modulo theories. *Software Tools for Technology Transfer (STTT)*, 18(2):205–225, 2016.
12. Robert B. Doorenbos. *Production Matching for Large Learning Systems*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, 1995.
13. Charles Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19:17–37, 1982.
14. Graphviz. <https://graphviz.org>, 2022.
15. Sylvain Hallé and Roger Villemaire. Runtime enforcement of web service message contracts with data. *IEEE Transactions on Services Computing*, 5(2):192–206, 2012.
16. Klaus Havelund. Runtime verification of C programs. In *Proc. of the 1st Test-Com/FATES conference*, volume 5047 of *LNCS*, Tokyo, Japan, 2008. Springer.
17. 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.
18. Klaus Havelund. Rule-based runtime verification revisited. *Software Tools for Technology Transfer (STTT)*, 17(2):143–170, 2015.
19. Klaus Havelund and Doron Peled. Runtime verification: From propositional to first-order temporal logic. In Christian Colombo and Martin Leucker, editors, *Runtime Verification - 18th International Conference, RV 2018, Limassol, Cyprus, November 10-13, 2018, Proceedings*, volume 11237 of *LNCS*, pages 90–112. Springer, 2018.
20. Klaus Havelund and Doron Peled. An extension of LTL with rules and its application to runtime verification. In Bernd Finkbeiner and Leonardo Mariani, editors, *Runtime Verification - 19th International Conference, RV 2019, Porto, Portugal, November 8-11, 2019, Proceedings*, LNCS. Springer, 2019.

21. M. Kim, S. Kannan, I. Lee, and O. Sokolsky. Java-MaC: a run-time assurance tool for Java. In *Proc. of the 1st Int. Workshop on Runtime Verification (RV'01)*, volume 55(2) of *ENTCS*. Elsevier, 2001.
22. LogFire. <https://github.com/havelund/logfire>.
23. LogScope in Python. <https://github.com/havelund/logscope>, 2022.
24. LogScope in Scala/C++. <https://github.com/logscope>, 2022.
25. Patrick O'Neil Meredith, Dongyun Jin, Dennis Griffith, Feng Chen, and Grigore Roşu. An overview of the MOP runtime verification framework. *International Journal on Software Techniques for Technology Transfer (STTT)*, pages 249–289, 2011.
26. Lee Pike, Nis Wegmann, Sebastian Niller, and Alwyn Goodloe. Copilot: Monitoring embedded systems. *Innovations in Systems and Software Engineering*, 9(4):235–255, 2013.
27. PyContract. <https://github.com/pyrv/pycontract>, 2022.
28. Giles Reger, Helena Cuenca Cruz, and David Rydeheard. MarQ: Monitoring at runtime with QEA. In Christel Baier and Cesare Tinelli, editors, *Proceedings of the 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2015)*, pages 596–610. Springer, 2015.

## A Visualization of Monitors

Textual monitors are automatically visualized using GRAPHVIZ’s dot-format [14]. This appendix shows the visualization of the textual monitors presented in Section 4.

**Monitor M1** The monitor M1 in Figure 2 is visualized in Figure 13. 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 horizontal line.

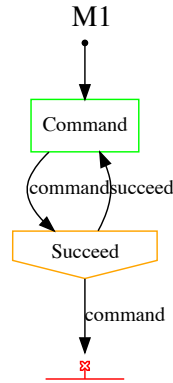


Fig. 13: Monitor M1 visualized.

**Monitors M1a, M1b, and M1c** The three monitors M1a, M1b, and M1c in Figure 3 are visualized in Figure 14. Figure 14a, 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 14b 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 14c is only that the initial state in Figure 14c has no name.

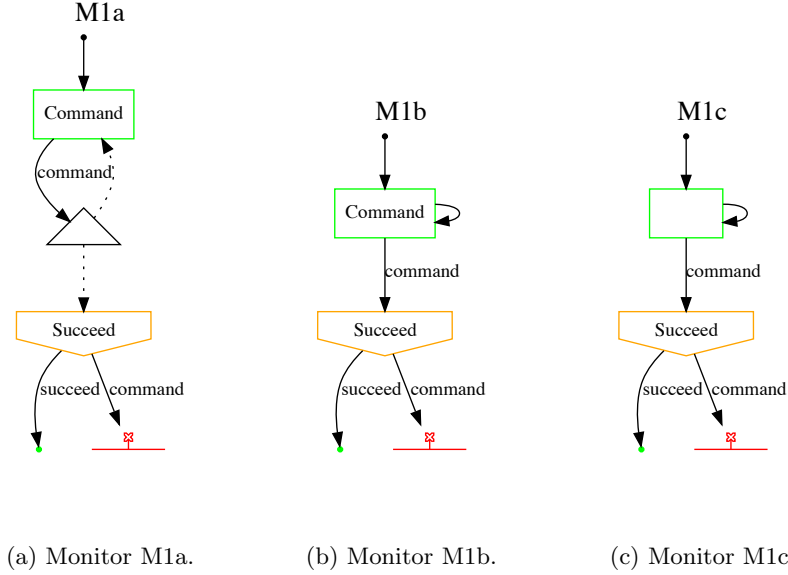


Fig. 14: Monitors M1a, M1b, and M1c visualized.

**Monitor M2** Monitor M2 in Figure 4 is visualized in Figure 15. 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 ‘ $\text{command}(\text{name} : x, \text{kind} : \text{"FSW"}) \Rightarrow \text{Succeed}(\text{c} : x)$ ’ from the initial **always** state is shown as an edge labelled with  $\text{command}(\text{cmd} : x, \text{kind} : \text{"FSW"})$ , and below it the binding of the  $\text{c}$  field of the **Succeed** state (see its definition) to the  $x$  that was bound on the left of the  $\Rightarrow$  symbol.

**Monitor M3** Monitor M3 in Figure 5 is visualized in Figure 16. 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  $\text{succeed}(\text{name} : x)$  but also with the condition pattern  $!\text{Succeed}(\text{c} : x)$  underneath.

**Monitor M4** Monitor M4 in Figure 6 is visualized in Figure 17. Recall that by observing the color scheme one can from the graph quickly understand the



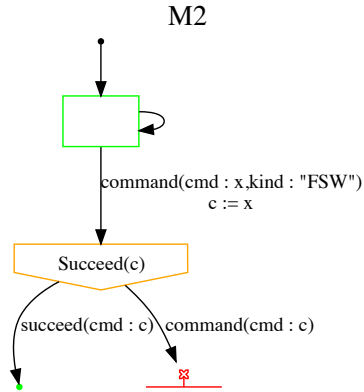


Fig. 15: Monitor M2 visualized.

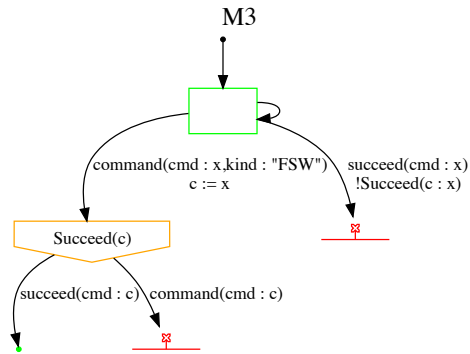


Fig. 16: Monitor M3 visualized.

violations being checked for: orange means terminating here is a violation, and red means an occurred violation.

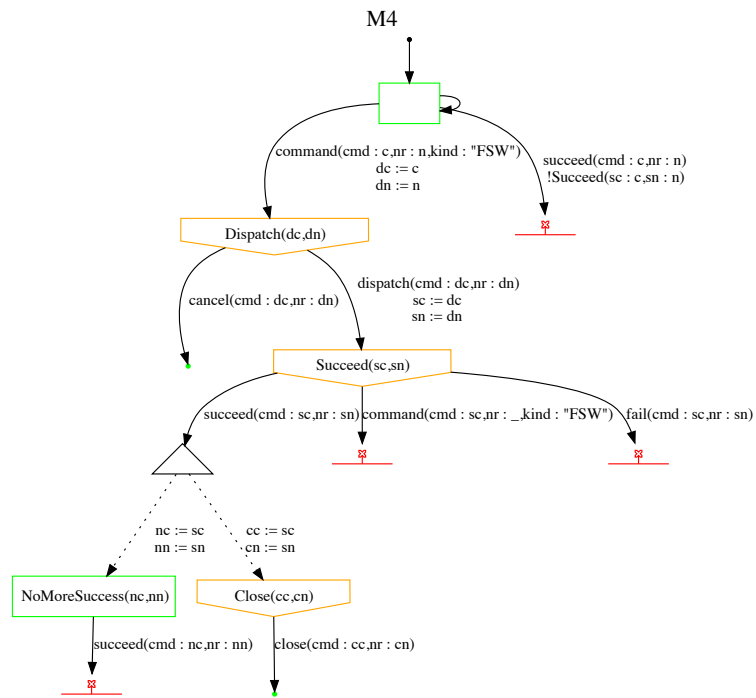


Fig. 17: Monitor M4 visualized.