Specification of Parametric Monitors
Quantified Event Automata versus Rule Systems

Klaus Havelund[1]* and Giles Reger[2]

[1] Jet Propulsion Laboratory, California Inst. of Technology, USA
[2] University of Manchester, UK

Abstract. Specification-based runtime verification is a technique for monitoring program executions against specifications formalized in formal logic. Such logics are usually temporal in nature, capturing the relation between events occurring at different time points. A particular challenge in runtime verification is the elegant specification and efficient monitoring of streams of events that carry data, also referred to as parametric monitoring. This paper presents two parametric runtime verification systems representing two quite different approaches to the problem. QEA (Quantified Event Automata) is a state machine approach based on trace-slicing, while LogFire is a rule-based approach based on the Rete algorithm, known from AI as being the basis for many rule systems. The presentation focuses on how easy it is to specify properties in the two approaches by specifying a collection of properties gathered during the 1st International Competition of Software for Runtime Verification (CSRV 2014), affiliated with RV 2014 in Toronto, Canada.

1 Introduction

Ensuring the correctness or security of a software system is traditionally approached in two ways, with *static analysis* and with *dynamic analysis*. By static analysis we shall broadly understand any approach that does not execute the program using a traditional execution platform, in contrast to dynamic analysis, where the program is executed. Static analysis techniques include for example abstract interpretation, theorem proving and model checking. The distinction is somewhat vague. Some techniques are difficult to classify, for example software model checkers which execute a program using a specialized virtual machine. Dynamic analysis includes testing, which is concerned with generating test inputs for the system, and applying test oracles (monitors) that can determine whether a particular run is satisfactory. However, dynamic analysis can also be applied after deployment of the software in the field, for example to profile behavior, load, etc. under realistic conditions. The concept of cyber-physical system (CPS) is receiving increased attention in the research community. A CPS is

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

a system of collaborating computational elements controlling physical entities. The correctness of such systems is extremely difficult to ensure statically. It is therefore important to ensure that at least observed executions of such systems satisfy certain properties.

Runtime verification [30, 41, 49] (RV) is a subfield of dynamic analysis focusing on *analyzing executions*, including collections thereof, either during test (test oracles), or after deployment. The field is not concerned with test case generation. Even though the field can appear rather narrow, by tradition it includes the following sub-fields. *Specification-based* monitoring, the topic of this paper, is concerned with checking a program execution against a formal specification of one or more requirements, expressed in some form of logic, for example state machines, regular expressions, temporal logic, grammars, or rule-systems. That is, given a program P, and a specification ψ, and an execution trace τ of P, an RV system will be able to determine whether τ satisfies ψ, also formalized as: $\tau \models \psi$. Logics must be expressive and monitors must be efficient. In *runtime analysis*, program executions are analyzed with specialized algorithms. Examples include algorithms for detecting concurrency problems such as deadlock potentials and data races. In *fault protection*, a monitored system will when violating a property be brought from an unsafe state to a safe state. *Specification learning* covers learning (mining) specifications from execution traces, which are then used for either system comprehension, or fed back into a verification system for further analysis, or used for monitoring future revisions of the software. *Trace visualization* of execution traces serves the purpose of comprehending what the system does. Finally, *program instrumentation* is concerned with how to instrument programs to emit events to a monitor, which then analyzes the event stream. For example aspect-oriented programming can be used for this purpose to automatically insert event emitting code at positions relevant for the properties being verified. Static analysis can be used to minimize the number of instrumentation points. RV technology can be stand-alone or incorporated into programming languages, simple assertions being a standard example.

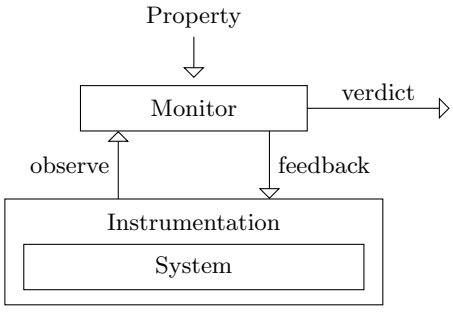

Fig. 1. A typical monitoring infrastructure.

Figure 1 shows a typical monitoring infrastructure, which supports the following activities. *Monitor creation*: a monitor is created, potentially from a formal property. *Instrumentation*: the system is instrumented to generate events for the monitor. *Execution*: The system is executed, generating events that are *observed* by the monitor. These events are either monitored online, as they are generated, or they are stored in logs, which are later analyzed by the monitor. *Responses*: the monitor produces for each consumed event a *verdict* indicating the status of the property, depending on the event sequence seen so far. In the case of online monitoring, the monitor may send *feedback* to the system, so that corrective actions can be taken by the system.

In this paper we shall focus on specification-based runtime verification, and in particular on what is referred to as *parametric monitoring*: the monitoring of event streams where events can carry data. That is, say we are monitoring a file system, then an event may have the form `read`(f) where f is a file identifier, a parameter to the `read` event. Parametric monitoring is particularly challenging as, for each incoming event, it involves the efficient lookup of information about the previous part of the execution trace that is relevant for that event's particular parameter (or set of parameters), in order to determine what the appropriate action of the monitor should be. We shall describe and apply two parametric runtime verification systems, representing two seemingly different main approaches of organizing monitors. The systems will be demonstrated on four classes of properties stemming from respectively the Java programming language API, banking, planetary rovers, and finally concurrency in programming languages. These properties were gathered during the 1st Intl. Competition of Software for Runtime Verification (CSRV 2014), affiliated with RV 2014 in Toronto, Canada [2], and also presented in [20]. Our focus is on specification, and we shall therefore assume that programs/systems to be monitored have been instrumented appropriately to emit the necessary events.

The first system, qea (Quantified Event Automata) [4, 53], is a state-machine logic based on a so-called trace-slicing approach, an approach that has shown to yield extremely efficient monitors. The approach involves conceptually slicing a trace into projections, a projection for each parameter combination. Fast indexing makes this very efficient. The state of a monitor is abstractly seen as a mapping from parameter values to monitor states. Note that trace slicing is not to be confused with program slicing. The latter involves slicing a program into projections. The second system, LogFire [40], is a rule-based system, inspired by rule systems as developed within the artificial intelligence community. LogFire's implementation is based on the Rete algorithm, often used for implementing rule engines. A monitor consists of a set of rules, which abstractly seen operate on a (structured) set of facts, where a fact is a named record $F(v_1, \ldots, v_n)$. Here F is a name and $v_1, \ldots, v_n$ are values. A rule's left-hand side can check for the presence or absence of facts, and the right-hand side can add or delete facts. In reality, however, facts are inserted into a network, the Rete network (*rete* means "net" in Latin), making rule evaluation more efficient. qea

is a so-called *external* DSL (Domain Specific Language), whereas LogFire is a so-called *internal* DSL, an API in the Scala programming language.

The rest of the paper is organized as follows. Section 2 gives a brief survey of specification-based runtime verification systems. Section 3 introduces the two logics qea and LogFire. Sections 4, 5, 6, and 7 contain the specification in the two logics of the Java API, banking, rover, and concurrency properties respectively. Section 8 summarizes and discusses the specification experience. Finally Section 9 concludes the paper.

2 Survey of specification-based runtime verification

Numerous runtime verification systems have been developed over the last 15 years, some of which will be mentioned here. We will only focus on specification-based systems that verify whether program executions satisfy user-provided formal specifications. As discussed in the introduction, the field is broader than this. Initial specification-based systems could only handle propositional events. These include for example Temporal Rover [26], Mac [48], and Java PathExplorer [43, 42]. Later work has studied such propositional monitoring logics from a more theoretic point of view, including notions such as 4-valued logics [15] and monitorability [29]. During the last decade there has been an increasing focus on systems for monitoring data parameterized events, so-called parametric monitoring. The first systems to handle parameterized events appeared around 2004, and include such systems as Eagle [7], Hawk [21], Jlo [58], TraceMatches [3], and Mop [18, 51]. Several systems have appeared since then. RV systems usually implement specification languages which are based on formalisms such as state machines [27, 34, 37, 18, 28, 9], regular expressions [3, 18], temporal logic [48, 26, 43, 7, 59, 21, 58, 57, 18, 9, 13, 36, 14, 22], variations over the μ-calculus [7], grammars [18], and rule-based systems [12, 40]. A few of these logics incorporate time as a built-in concept, for example the metric first-order temporal logics in the early TemporalRover [26] and in the more recent MFOTL [13]. If no special concept of time is introduced, time observations can be considered as just data, as is common in rule-based systems [12, 40].

In this paper we focus on two important approaches to parametric monitoring, namely the trace-slicing approach, represented by qea, and the rule-based approach, represented by LogFire. Two other important approaches based on trace-slicing are TraceMatches [3] and Mop [18, 51]. Slicing-based approaches are generally extremely efficient, but at the cost of lack of some expressiveness, as pointed out in [4]. qea is an attempt to augment the expressiveness of slicing-based approaches, without losing too much of the efficiency. Rule systems are expressive.

The Rete-based LogFire is inspired by the Ruler system [11, 12, 1], itself a rule-based system. Ruler, however, is not influenced by Rete. Ruler led to the study of the Rete algorithm (in LogFire), in order to determine its relevance for runtime verification. Ruler itself was inspired by MetateM [5] and Eagle [7], a linear time μ-calculus for monitoring, with past time as well

as future time operators. Although attractive, the implementation of EAGLE appeared complex, leading to the simpler implementation in RULER. Another derivative from RULER is LOGSCOPE [10, 35, 8], which is a data parameterized state machine oriented monitoring logic, implemented as a simplified rule-engine, and applied to log files at Jet Propulsion Laboratory (JPL), for testing of the Mars Curiosity rover. The QEA formalism has similarities with LOGSCOPE. Other rule-based systems include DROOLS [24], JESS [45] and CLIPS [19]. Standard rule systems usually enable processing of facts, which have a life span. In contrast, LOGFIRE additionally implements events, which are instantaneous. DROOLS supports a notion of events, which are facts with a limited life span, inspired by the concept of *Complex Event Processing* (CEP), described by David Luckham in [50]. The DROOLS project has an effort ongoing, defining functional programming extensions to DROOLS [25]. In contrast, by embedding a rule system in an object-oriented and functional language, LOGFIRE can leverage the already existing host language features.

The classification of runtime verification frameworks into slicing-based and rule-based is an idealization, and more research is needed in order to make a clean classification. Some logics, for example, handle data as constraints, including the first-order Linear Temporal Logic LTL^{FO} [14], and the first-order linear temporal logic based on SMT (Satisfiability Modulo Theories) solving described in [22]. Some systems based on Linear Temporal Logic (LTL) [52] apply rewriting of LTL formulas, inspired by [33]. These include for example [26, 43, 7, 59, 58, 9, 36].

QEA, like most of the logics mentioned above, is an *external DSL*, a standalone "small" language equipped with its own customized parser. In contrast, LOGFIRE is an *internal* SCALA *DSL*, essentially an API in SCALA. Two other rule-based internal DSLs for SCALA exist: HAMMURABI [32] and ROOSCALOO [55]. HAMMURABI, which is not RETE-based, achieves efficient evaluation of rules by evaluating these in parallel, assigning each rule to a different SCALA actor. ROOSCALOO [55] is RETE based, but is not documented in any form other than experimental code. Other internal SCALA DSLs for monitoring include TRACECONTRACT [9] and DAUT (Data automata) [38, 39], both of which are based on parameterized state machines. An embedding of LTL in HASKELL is described in [59]. MOPBOX [17] is a JAVA library for monitoring, offering a re-implementation of the efficient trace-slicing algorithms contained in MOP [18, 51], but defining the interface as an API.

3 Introduction to QEA and LogFire

This section introduces the two logics QEA (Section 3.2) and LOGFIRE (Section 3.3). The logics are illustrated using a file usage example, presented below.

3.1 The file usage example

Consider that we can monitor the following file usage events:

– `open`$(f, m, size)$: records that file f is being opened in mode m (R for read and W for write), and $size$ denotes the size of the file in bytes on opening.
– `close`(f): records closing file f.
– `read`(f): records reading a file f.
– `write`(f, b): records writing b bytes to file f.

The correct usage of the file system is captured by the following informal requirements:

– A file starts closed and cannot be opened (closed) if already open (closed).
– A file if opened must eventually be closed.
– A file can only be read or written if it is open in the corresponding mode.
– No file can exceed 16MB.

We will use this example to illustrate QEA and LogFire in the following two sections.

3.2 Introduction to QEA

Quantified event automata (QEA) [4, 53] is an automaton-based specification language for monitoring parameterized events. It is based on the idea of *parametric trace-slicing* [18], which is most prominently used by Mop. The theory behind QEA generalizes this previous notion of slicing in a number of ways that will be discussed below. The MarQ tool [54] takes QEA specifications and can monitor them either online on Java programs or offline on recorded traces.

```
qea{
  Forall(f)
  accept next(closed){
    open(f,'R',_) -> readonly
    open(f,'W',size) -> writeonly
  }
  next(readonly){
    read(f) -> readonly
    close(f) -> closed
  }
  next(writeonly){
    write(f,b) if [size+b <= 16000000 ]
               do [ size+=b ] -> writeonly
    close(f) -> closed
  }
}
```

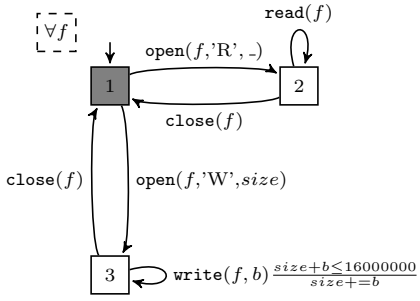

Fig. 2. A QEA model of the file usage property in both text and graphical formats.

Specification of file usage example in QEA. The QEA specification of the file usage property is given in Figure 2 in both a textual and a graphical representation. In other work we have mainly used the graphical representation as we

find it more readable. However, as it is not an input format, it is not appropriate for discussing specification approaches and will use the textual format later.

A QEA consists of some quantifications and an automaton. In this example the quantification is **Forall** `(f)` stating that f is a universally quantified variable. This quantifier list can also include **Exists** quantification and a global guard on quantified variables introduced by the **Where** keyword.

The automaton has three states relating to the three states a file can be in. In the textual format the first state defined is always taken to be the initial state. States have a kind, either **next** or **skip**, which defines what should happen when a transition cannot be taken; **next** means a next event is required, **skip** means the next event will be skipped if no transition can be taken. The states here are all **next** states i.e. from the `closed` state it is only possible to `open` a file as an event is required to make a transition.

Only the `start` is state is an **accept** state. As defined later, a trace is accepted if it has a path to an accepting state. The language also includes implicit **success** and **failure** states which are **skip** states with no outgoing transitions and where the former is an **accept** state.

In the `writeonly` state the `write` transition has a guard `size+b <= 16000000` and assignment `size+=b`, which are used to ensure that the size of the file does not exceed the limit, note how the `size` variable is given the initial size when the file is opened. The **if** keyword introduces guards and the **do** keyword introduces assignments.

Let us consider how we would monitor this property on the following trace:

$$\texttt{open}(A, \text{'R'}, 0).\texttt{open}(B, \text{'W'}, 15999900).\texttt{write}(B, 100).\texttt{read}(A).\texttt{write}(B, 100).\texttt{close}(B)$$

Here there are two files, A and B. The quantification $\forall f$ tells us that we need to *slice* the trace on these values. This gives us the following two trace slices associated with bindings of f:

$$\begin{array}{lcl} [f \mapsto A] & \mapsto & \tau_1 = \texttt{open}(A, \text{'R'}, 0).\texttt{read}(A) \\ [f \mapsto B] & \mapsto & \tau_2 = \texttt{open}(B, \text{'W'}, 15999900).\texttt{write}(B, 100).\texttt{write}(B, 100).\texttt{close}(B) \end{array}$$

The trace slice τ_1 should then be interpreted for the automaton with f replaced by A. In this case there is a path in the automaton from state `closed` to state `readonly`. But, as this state is not accepting, any trace finishing in this state is rejected. Similarly the trace slice τ_2 should be interpreted on the automaton with f replaced by B. Here we can capture the behavior through the rewriting of a configuration containing the current state and binding of free variable *size*:

$$\langle 1, [] \rangle \xrightarrow{\texttt{open}(B,\text{'W'},15999900)} \langle 3, [size \mapsto 15999900] \rangle \xrightarrow{\texttt{write}(B,100)} \langle 3, [size \mapsto 16000000] \rangle$$

After two events we reach state 3 (`writeonly`) but cannot take the `write` transition as the guard `size+b <= 16000000` does not hold. As the state is labelled **next** this leads to failure as **next** states must be able to make a move on the next event. This trace violates the property as at least one, in this case both, of the trace slices are not accepted by the instantiated automaton.

As we consider the trace slices τ_1 and τ_2 separately, we would have had the same result for any interleaving of τ_1 and τ_2. This interleaving can be restricted if two trace slices contain the same ground event, as the slices must 'synchronize' on an occurrence of this event in the full trace.

Defining QEA acceptance through trace-slicing. QEA has been formally defined elsewhere [4, 53], here we give a flavor of the structures and notion of trace acceptance. We begin with the basic definitions. An event in the alphabet $\Sigma(X, Y)$ is of the form $e(\overline{z})$ where $\overline{z} \in (X \cup Y \cup \text{Val})^*$ for values Val and disjoint variable sets X and Y. We separate the variables into those that will be quantified X and those that will remain free Y; this distinction will be clarified below. An event is ground if $\overline{z} \in \text{Val}^*$. A trace is a finite sequence of ground events. A binding is a map (partial function with finite domain) from variables to values. Bindings can be applied to events to rewrite their variables. A guard is a predicate on bindings. An assignment is a (partial) function on bindings. An event $e(\overline{z})$ *matches* a ground event $e(\overline{v})$ if there is a binding θ such that $\theta(e(\overline{z})) = e(\overline{v})$ and $\texttt{match}(e(\overline{z}), e(\overline{v}))$ is the minimal such binding with respect to the sub-map relation (if such a binding exists, undefined otherwise).

An event automaton (EA) is a (potentially non-deterministic) finite state machine with alphabet $\Sigma(X, Y)$ where transitions can be labelled with guards and assignments. Recall that states can be either **next** or **skip** as described above. An EA can be *grounded* with a binding with domain X. A grounded EA has an acceptance relation for traces (over its alphabet) defined for configurations (pairs of states and bindings). We do not give this relation here but it is the standard transition relation extended for guards, assignments and the notion of **next** and **skip** states. For an EA $\mathcal{E}$ and a grounding binding θ, the acceptance relation defines a ground language $\mathcal{L}(\theta(\mathcal{E}))$ over $\Sigma(\theta(X), \text{Val})$ as the set of all traces that can reach a configuration with an accepting state. Note that free variables (Y) are replaced by Val as they can take any value that is acceptable to the transition relation i.e. satisfies the guard and assignment structures.

For example, the automaton in Fig. 2 has alphabet $\Sigma(\{f\}, \{size, b\})$. The grounded language for binding $[f \mapsto A]$ would contain traces such as

$$\texttt{open}(A, \text{'R'}, 0).\texttt{read}(A).\texttt{close}(A).\texttt{open}(A, \text{'R'}, 0).\texttt{read}(A).\texttt{close}(A)$$
$$\texttt{open}(A, \text{'W'}, 12000).\texttt{write}(A, 100).\texttt{write}(A, 100).\texttt{close}(A)$$

as they reach an accepting state and satisfy all guards.

A QEA is a pair consisting of a quantifier list $\Lambda(X)$ and an EA over alphabet $\Sigma(X, Y)$. The quantifier list consists of universal and existential quantification over variables X, can include a global guard over X and can be negated. The domain of a quantified variable $x \in X$ is given as those values that can be bound to x when matching events from the EA's alphabet with events in the trace i.e.

$$\mathcal{D}(\tau, x) = \{\texttt{match}(e(\overline{z}), e(\overline{v}))(x) \mid e(\overline{z}) \in \Sigma(X, Y), e(\overline{v}) \in \tau\}$$

In the previous section we extract the domain $\mathcal{D}(\tau, f) = \{\mathrm{A}, \mathrm{B}\}$ as, for example,

$$\begin{array}{ll} \texttt{match}(\texttt{open}(A, \text{`}R\text{`}, 0), \texttt{open}(f, \text{`}R\text{`}, size)) & = [f \mapsto A, size \mapsto 0] \\ \texttt{match}(\texttt{close}(B), \texttt{close}(f)) & = [f \mapsto B] \end{array}$$

The notion of trace acceptance is defined using *trace-slicing*. We first consider pure universal quantification. Given a QEA with $\Lambda = \forall x_1 ... \forall x_n$ and EA $\mathcal{E}$, the trace τ is accepted if for *every* binding θ such that $\theta(x_i) \in \mathcal{D}(\tau, x_i)$ the trace $\tau \downarrow_{\Sigma(\theta(X),Y)}$ is in $\mathcal{L}(\theta(\mathcal{E}))$ where the slicing (projection) operation $\downarrow_{\mathcal{A}}$ is defined as

$$\begin{array}{lll} \epsilon \downarrow_{\mathcal{A}} & = \epsilon & \\ \tau.e(\overline{v}) \downarrow_{\mathcal{A}} & = \begin{array}{l} \tau \downarrow_{\mathcal{A}} .e(\overline{v}) \\ \tau \downarrow_{\mathcal{A}} \end{array} & \begin{array}{l} \text{if } \exists e(\overline{z}) \in \mathcal{A} \text{ such that } \texttt{matches}(e(\overline{z}), e(\overline{v})) \\ \text{otherwise} \end{array} \end{array}$$

i.e. for the binding θ we *slice* the trace to give only events relevant to θ, then we check if that trace slice is accepted by the EA grounded with θ. This can be appropriately modified for existential and alternating quantification. For example, the quantifier alternation $\exists x \forall y$ says that there is a value $d \in \mathcal{D}(\tau, x)$ such that $\tau \downarrow_{\Sigma(\theta(X),Y)}$ is in $\mathcal{L}(\theta(\mathcal{E}))$ for every binding θ where $\theta(x) = d$.

Note that this slicing can place ground events in multiple slices i.e. if we had an alphabet $\{\texttt{create}(c, i), \texttt{update}(c), \texttt{use}(i)\}$ for quantified c and i, the event $\texttt{create}(A)$ would be relevant to bindings $[c \mapsto A, i \mapsto 1]$ and $[c \mapsto A, i \mapsto 2]$. Also free variables are ignored for slicing, so an event in the trace matching an event using only free variables would be relevant to all bindings.

Monitoring algorithm. Whilst the above gives a reasonable definition of trace acceptance, this is not a pragmatic method for runtime monitoring as it requires multiple passes of the trace. Instead, an incremental notion of acceptance has been introduced [4, 53] which maintains a map from bindings of quantified variables to sets of configurations. As not all information is available at the start it is necessary to track partial bindings of quantified variables, which requires careful treatment to preserve the semantics described above. The introduction of this map from bindings to configurations lends itself to forms of *indexing*, which is what has made tools that use trace-slicing, such as MOP, highly efficient. MARQ implements a symbol-based form of indexing that uses the alphabet of each (partially) instantiated EA to locate the relevant configurations.

Free versus quantified variables. One aspect of QEA that deserves clarification is the difference between *quantified* and *free* variables. Recall that an EA has two sets of variables, X and Y, and the quantifications range of X, leaving those in Y free. The QEA in Figure 2 has one quantified variable f and two free variables b and $size$. As we saw in the earlier examples, we use values for f to *slice* the trace and then fix this value when evaluating the trace. The variables b and $size$ are *rebound* whenever they match a new value and can be checked in guards and updated in assignments.

```
class FileUsage extends Monitor {
  "r1" -- 'open('f, 'm, 'size) & not('Open('f, '_, '_)) |->
    insert('Open('f, 'm, 'size))
  "r2" -- 'Open('f, '_, '_) & 'open('f, '_, '_) |-> fail()
  "r3" -- 'Open('f, '_, '_) & 'close('f) |-> remove('Open)
  "r4" -- 'Open('f, 'm, '_) & 'read('f) |-> ensure('m.string == "R")
  "r5" -- 'read('f) & not('Open('f, '_, '_)) |-> fail()
  "r6" -- 'Open('f, 'm, 'size) & 'write('f, 'b) |-> {
    ensure('m.string == "W" && 'size + 'b <= 16000000)
    update('Open('f, 'm, 'size + 'b))
  }
  "r7" -- 'write('f) & not('Open('f, '_, '_)) |-> fail()

  hot('Open)
}
```

Fig. 3. An LogFire model of the file usage property.

Actual guards and assignments. Whilst the qea language theoretically supports arbitrary guards and assignments, the MarQ monitoring tool currently only supports those relating to arithmetic and sets. For arithmetic we have the standard comparison operations (i.e. $=, \leq$), arithmetic operators (i.e. $+, \times$) and update operations (i.e. +=,++). For sets we have set definition, addition and removal, and the contains (in) predicate.

3.3 Introduction to LogFire

LogFire [40] is a rule-based specification language specifically developed for monitoring streams of parameterized events. It is developed as an internal Scala DSL (an API in Scala), allowing a user to freely mix rule programming with traditional programming. It is based on the Rete algorithm [31], specifically as described in [23]. The Rete algorithm is the basis for various rule-based systems. LogFire augments it with a distinction between facts and events and implements an indexing algorithm for optimizing rule evaluation. LogFire can be used for online as well as offline monitoring, although offline monitoring (log analysis) has been the main focus. In the case of online monitoring, the tool would have to be connected to the application via some instrumentation tool, such as AspectJ [46].

Specification of file usage example in LogFire The LogFire specification of the file usage property is given in Fig. 3. LogFire only provides a textual language. One could imagine a graphical notation for rule-based systems, similar to (but yet different from) the visualization of qea models, as illustrated in Figure 2. However, this is not explored in this work. A LogFire monitor is defined as a Scala class that extends (is a sub-class of) the class `Monitor`, which defines all the LogFire primitives (constants, variables and functions), which allows one to write rules. A monitor extending this class must define zero or more rules, each of the form:

```
"name" -- 'cond1 & ... & condn |-> action
```

A rule has a name (a string), followed by a left-hand side, which is a list of conditions separated by conjunction (‘&’); and a right-hand side following the arrow (|->), this is the action. The state of a monitor at any time during the evaluation of an event stream is conceptually (simplified) a set of facts of the form $F(v_1, \ldots, v_n)$, where F is a name and $v_1, \ldots, v_n$ is a list of ground values. This set is referred to as the *fact memory*. Observed events are also facts, which, however, only exist a brief moment when observed. By convention, names of observed events consist of all small letters, whereas names of internally generated facts start with a capital letter. A condition in a rule's left-hand side can check for the presence or absence of a particular fact (including events), and the action on the right-hand side of the rule can add or delete facts, produce error messages, or cause other side effects. Generally, an action can be any SCALA code. Left-hand side matching against the fact memory usually requires unification of variables occurring in conditions. In case all conditions on a rule's left-hand side match (become true), the right-hand side action is executed. This model is very well suited for processing data rich events, and is simple to understand by nature of its very operational semantics. It is interesting to note that finite-state machines can be mapped into rule systems.

The monitor in Figure 3 should be understood as follows. The monitor operates with one fact: $Open(f, m, size)$, representing the fact that file f has been opened in mode m, and currently has size $size$. There are seven rules, named `r1` ... `r7`. Rule `r1` states that upon the occurrence of an `open(f, m, size)` event, if the file has not been opened yet (there is no fact in the fact memory that matches the pattern `Open(f, _, _)`), then an `Open(f, m, size)` fact is inserted into the fact memory. Free identifiers on the left-hand side, appearing as SCALA quoted symbols, such as `'size`, are bound when matched against facts, including events. Rule `r2` states that if a file has been opened, an open event will cause a failure to be reported. Rule `r3` handles the closing of a file, causing the fact matching `Open(f, _, _)` to be removed from the fact memory. Rule `r4` ensures that a file can only be read if it has been opened in read-mode. An alternative, also allowed, formulation of this rule would have been:

```
"r4" -- 'Open('f, "W", '_) & 'read('f) |-> fail()
```

Rule `r5` states that reading a file that is not open is reported as a failure. Rule `r6` captures writing to a file, and ensures that the file has been opened in write-mode, and that the new size does not exceed the upper allowed bound. In addition, the `Open` fact is updated to record the increased size. Rule `r7` states that it is illegal to write to a non-opened file. Finally, the fact `Open` is recorded to be a so-called *hot* fact, which is equivalent to a non-acceptance state in a state machine. When monitoring terminates there should be no hot facts in the fact memory. Figure 4 shows how the fact memory evolves as the events are consumed in the previous trace used to illustrate QEA.

Implementation of LogFire In the presentation above, the configuration of LOGFIRE was described as being a set of facts, those active at any moment during monitoring. Facts can be added or deleted from this set. This explanation is valid

$$
\begin{aligned}
&\{\} \xrightarrow{\texttt{open}(A,\text{`R'},0)} \\
&\{\texttt{Open}(\text{`A'},\text{`R'},0)\} \xrightarrow{\texttt{open}(B,\text{`W'},15999900)} \\
&\{\texttt{Open}(\text{`A'},\text{`R'},0),\texttt{Open}(\text{`B'},\text{`W'},15999900)\} \xrightarrow{\texttt{write}(B,100)} \\
&\{\texttt{Open}(\text{`A'},\text{`R'},0),\texttt{Open}(\text{`B'},\text{`W'},16000000)\} \xrightarrow{\texttt{read}(A)} \\
&\{\texttt{Open}(\text{`A'},\text{`R'},0),\texttt{Open}(\text{`B'},\text{`W'},16000000)\} \xrightarrow{\texttt{write}(B,100)} \text{error}
\end{aligned}
$$

Fig. 4. Evolution (simplified) of LogFire fact memory.

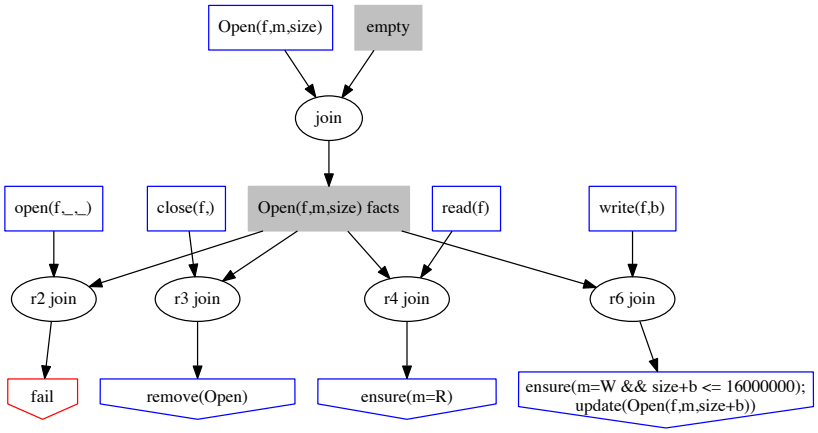

Fig. 5. Rete network for rules `r2`, `r3`, `r4`, and `r6`.

for understanding specifications. However, such a set implementation would be potentially inefficient when evaluating rules against an incoming event. Consider for example the configuration:

$$\{\texttt{Open}(\text{`A'},\text{`R'},0),\texttt{Open}(\text{`B'},\text{`W'},15999900)\}$$

and the incoming event $\texttt{write}(B,100)$. We could now (i) evaluate all the seven rules, one by one, and for each we would (ii) scan the set above to examine if it contains a relevant fact $\texttt{Open}(\text{`B'},\text{`W'},\ldots)$. To avoid this, LogFire implements the Rete algorithm, which optimizes (i). In addition, LogFire augments this algorithm with indexing into this set, which optimizes (ii). The Rete algorithm stores rules and facts as a network, which "glues" rules together that have common prefixes of left-hand sides.

A Rete network for rules `r2`, `r3`, `r4`, and `r6` is shown in Figure 5. Suppose rule `r1` fires and adds an `Open`(f,m,size) fact. This will enter the upper left so-called *alpha memory*. Newly added facts (including events) are added to

alpha memories (white boxes). The top *join node* will be activated and merge the incoming fact with previous facts, of which there are none (the initial so-called *beta memory*, the upper right grey box, is empty). The result is stored in a new beta memory. Each of these beta memories represents a prefix of the conditions in a rule. They contain all facts matching a corresponding condition with pointers back to facts in preceding beta memories, establishing a collection of chains of facts. Since `Open`(f,m,size) occurs as the first condition of rules `r2`, `r3`, `r4`, and `r6`, these now are connected to this beta memory. For example, in the case of a `write`$(B, 100)$ event, the event is added to the right-most alpha memory, whereafter rule `r6`'s join node is activated, corresponding to firing of rule `r6`. Thereby we avoid evaluating all the other rules, corresponding to the optimization of case (i) above.

Concerning optimization (ii), recall that a beta memory contains all facts matching a corresponding condition. It can for example be the set above containing the two facts: `Open`('A','R',0) and `Open`('B','W',15999900). On the observation of a `write`$(B, 100)$ event, rule `r6`'s join node will now have to search for an `Open`('B',...,...) fact in this set. With a large number of facts, this can be costly. To optimize this search, the beta memory is organized as an index from file identifiers to sets of facts:

$$A \mapsto \{\texttt{Open}(\text{'A'}, \text{'R'}, 0)\}$$
$$B \mapsto \{\texttt{Open}(\text{'B'}, \text{'W'}, 15999900)\}$$

It is the join node's responsibility to look up the relevant facts in the beta node when it is activated with an event from an alpha node. The first argument to the `write`$(B, 100)$ event tells the join node to look up B in the index. As such the implementation has some similarity with the slicing approach.

LogFire syntax It remains to briefly explain how rules are interpreted. Consider for example rule `r4`. This rule is by the SCALA compiler interpreted as the following chain of method calls:

```
R("r4").--(C('Open).apply(('f, 'm, '_))).&(C('read).apply('f)).|->{
  ensure('m.string == "R")
}
```

SCALA allows dots and parentheses around method arguments to be omitted in calls of methods on objects. In the above expansion these have been inserted. In addition, two so-called *implicit functions* `R` and `C` have been applied by the SCALA compiler. A user-defined implicit function from type T_1 to type T_2 is applied by the compiler when a value of type T_1 occurs in a place where a value of type T_2 is expected (the function must be unique). For example, `R` is defined as follows, returning an (anonymous) object, which defines the method '`--`', which again returns an object defining the methods '`&`' and '`|->`'.

```
implicit def R(name: String) = new {
  def --(c: Condition) = new RuleDef(name, List(c))
}

class RuleDef(name: String, conditions: List[Condition]) {
```

```
  def &(c: Condition) = new RuleDef(name, c :: conditions)

  def |->(stmt: => Unit) {
    addRule(Rule(name, conditions.reverse, Action((x: Unit) => stmt)))
  }
}
```

4 Specification of Java API properties

The first domain we consider is the task of checking compliance with the JAVA library API. These properties are common examples in publications on runtime verification, and some featured multiple times in the competition. In this case they are all about collections but properties of sockets and streams have been discussed elsewhere [53]. There has also been a systematic effort to formalize the informal properties in the JAVA library documentation [47]. Properties about programming APIs, specifically object-oriented languages like JAVA, have common characteristics. Typically they are about a small set of objects, and if more than one object is targeted then the objects are typically *connected* in some way i.e. one is created from another.

4.1 HasNext

This property applies to every `java.util.Iterator` object, and requires that `hasNext`() be called before `next`() and that `hasNext`() returns true. Two events are monitored: `hasNext`(i, r) is triggered when `hasNext` is called on Iterator i with result r and `next`(i) is triggered when `next` is called on Iterator i.

QEA specification. The specification defines a `safe` and `unsafe` state; a `next` event is only allowed in the `safe` state, which is reached by `hasNext` returning true.

```
qea {
  Forall(i)
  accept skip(unsafe){
    hasnext(i,r) if [ r = true ] -> safe
    next(i) -> failure
  }
  accept skip(safe){
    next(i) -> unsafe
  }
}
```

LogFire specification. The monitor uses one fact, `Safe(i)`, to record when `hasnext` has been called returning true. It is required by `next`, as a guard, and then removed. The monitor, as subsequent LOGFIRE monitors, is named `M` in order to keep naming brief. Similarly, rule names are kept short.

```
class M extends Monitor {
  "r1" -- 'hasnext('i, true) |-> insert('Safe('i))
  "r2" -- 'Safe('i) & 'next('i) |-> remove('Safe)
  "r3" -- 'next('i) & not('Safe('i)) |-> fail()
}
```

4.2 Counting iterator

If a `java.util.Iterator` object is created from a collection of size s then we can only call **`next`** on that iterator at most s times. Two events are relevant: **`iterator`**(i,s) records the creation of iterator i from a collection of size s; and **`next`**(i) records call **`next`** on iterator i. As iterators are JAVA objects they can only be created once.

QEA specification. This QEA saves the size of the iterator and decreases this size on each **`next`** event whilst it is safe to do so. As the `iterate` state is a **next** state, a failure will occur if a **`next`** event occurs and the guard `csize > 0` cannot be satisfied.

```
qea{
  Forall(i)
  accept skip(start){ iterator(i,csize)  -> iterate }
  accept next(iterate) { next(i) if [ csize > 0 ] do [ csize-- ] -> iterate }
}
```

LogFire specification. The monitor uses one fact, `Iterate(i,csize)`, to record that there are `csize` elements left in the collection that iterator `i` is derived from. It is decreased on each observation of a `next`.

```
class M extends Monitor {
  "r1" -- 'iterator('i, 'csize) |-> 'Iterate('i, 'csize)
  "r2" -- 'Iterate('i, 'csize) & 'next('i) |-> {
    if ('csize > 0)
      update('Iterate('i, 'csize - 1))
    else
      fail()
  }
}
```

4.3 UnsafeMapIterator

If a collection is created from a `java.util.Map` object (via calls to `values` or `keySet`) and then an `java.util.Iterator` object is created from that collection, then the iterator cannot be used after the original map has been updated. Four events are relevant: **`create`**(m,c) records the creation of collection c from map m; **`iterator`**(c,i) records the creation of iterator i from collection c; **`update`**(m) records m being updated; and **`use`**(i) records the usage of iterator i. As collections and iterators are JAVA objects they can only be created once.

QEA specification. This QEA specifies the path to failure i.e. the sequence of events that reach a non-accepting state. Note that quantification is over all maps, collections and iterators. A naive monitoring algorithm would create all such bindings, even for unrelated objects - this is an inherent inefficiency in trace-slicing that must be avoided through careful implementation and extension of the theory. Note the use of **skip** states to ignore irrelevant events.

```
qea{
  Forall(m,c,i)
  accept skip(start){  create(m,c) -> hascol }
  accept skip(hascol){ iterator(c,i) -> hasit }
  accept skip(hasit){  update(m) -> updatedm }
  accept skip(updatedm){ use(i) -> failure }
}
```

LogFire specification. The monitor uses three facts: `HasCol(m,c)` to record when collection `c` has been created from a map `m`; `HasIt(m,i)` to record that iterator `i` has been created from a collection created from map `m`; and finally `UpdateM(i)` to record that the map that iterator `i` is derived from has been updated, and hence no further iteration (use) is allowed.

```
class M extends Monitor {
  "r1" -- 'create('m, 'c) |-> insert('HasCol('m, 'c))
  "r2" -- 'HasCol('m, 'c) & 'iterator('c, 'i) |-> insert('HasIt('m, 'i))
  "r3" -- 'HasIt('m, 'i) & 'update('m) |-> insert('UpdateM('i))
  "r4" -- 'UpdateM('i) & 'use('i) |-> fail()
}
```

4.4 Hashing persistence

Objects placed in a hashing structure, such as a `HashSet` or `HashMap`, should have persistent hash codes whilst in the structure for the usage to be sound. Otherwise we may have the situation where we add an object and then get the result false when checking for its presence. Three events are relevant: `add`(c,o,h), `observe`(c,o,h) and `remove`(c,o,h) respectively record the addition, observation and removal of object o on hashing collection c using hash code h.

QEA specification. We have chosen not to quantify over collections in this QEA as it is possible to specify the property by quantifying over objects only. Note that monitoring complexity depends on the number of bindings, which can be, in the worst case, exponential in the number of quantified variables. This specification works by tracking how many collections the object is in and ensuring that the hash code remains persistent whilst this is non-zero. This assumes that the `add` event only occurs when the object did not previously exist in the collection, and `remove` only occurs when it did. If these assumptions cannot be enforced (via instrumentation) then the specification would need to quantify over c.

```
qea{
  Forall(o)
  accept skip(out){
    add(c,o,h) do [ count:=1; ] -> in
  }
  accept next(in){
    add(c,o,h2) if [ h = h2 ] do [ count++ ] -> in
    observe(c,o,h2) if [ h = h2 ] -> in
    remove(c,o,h2) if [ count > 1 and h = h2 ] do [ count-- ] -> in
    remove(c,o,h2) if [ count = 1 and h = h2 ] -> out
  }
}
```

LogFire specification. The monitor uses one fact, `In(c,o,h)`, to record that collection `c` contains object `o`, which had hash code `h` when first added. This should remain the hash code as long as the object is in that collection.

```
class M extends Monitor {
  "r1" -- 'add('c, 'o, 'h) |-> insert('In('c, 'o, 'h))
  "r2" -- 'In('c, 'o, 'h1) & 'observe('c, 'o, 'h2) |-> ensure('h1.i == 'h2.i)
  "r3" -- 'In('c, 'o, 'h1) & 'add('c, 'o, 'h2) |-> ensure('h1.i == 'h2.i)
  "r4" -- 'In('c, 'o, 'h1) & 'remove('c, 'o, 'h2) |-> {
    ensure('h1.i == 'h2.i)
    remove('In)
  }
}
```

5 Specification of banking properties

The next application domain we consider is that of banking. The following properties are concerned with accounts and transfers. There is a focus on timed properties i.e. ensuring that an action occurs within a given timeframe. In both logics time is modeled as time stamps, which are just data.

5.1 Unique accounts

An account approved by the administrator may not have the same account number as any other already existing account in the system. The event `approve`(*id*) indicates that an account with *id* has been approved.

QEA specification. To specify this property in a pure trace-slicing style we would quantify over ids and record a failure if two events occur with the same quantified id.

```
qea{
  Forall(id)
  accept skip(start){ approve(id) -> once }
  accept skip(once) { approve(id) -> failure }
}
```

However, with QEAs we can also maintain a set S of previously seen account ids. This is more like the fact-based approach taken by LogFire. It does not utilize the trace-slicing mechanisms but will be more efficient for this very simple property as trace-slicing, and the associated indexing, is not required.

```
qea{
  accept next(safe){ approve(id) if[ not(id in S) ] do[ S.add(id) ] -> safe }
}
```

LogFire specification. The monitor uses one fact, `Approved(id)`, to record that account `id` has been approved.

```
class M extends Monitor {
  "r1" -- 'approve('id) |-> insert('Approved('id))
  "r2" -- 'Approved('id) & 'approve('id) |-> fail()
}
```

5.2 Greylisting

Once grey-listed, a user must perform at least three incoming transfers before being white-listed. There are three relevant events. `greyList`(*user*) and `whiteList`(*user*) indicate that *user* was grey and white-listed respectively and `transfer`(*user*) records the fact that *user* performed a transfer.

QEA specification. This specification has two states indicating the status of the user. The number of transfers are counted in the `grey` state (and zeroed on a `greyList` event) so that this count can be checked at a `whiteList` event.

```
qea{
  Forall(u)
  accept skip(white){
    greyList(u) do [ count:=0 ] -> grey
  }
  accept next(grey){
    transfer(u) do [ count++ ] -> grey
    greyList(u) do [ count:=0 ] -> grey
    whiteList(u) if [count >= 3 ] -> white
  }

}
```

LogFire specification The monitor uses one fact, `Grey(u,count)`, to record that user `u` has been grey-listed, and since then has had `count` incoming money transfers, initially 0.

```
class M extends Monitor {
  "r1" -- 'greyList('u) & not('Grey('u, '_)) |-> insert('Grey('u, 0))
  "r2" -- 'Grey('u, '_) & 'greyList('u) |-> update('Grey('u, 0))
  "r3" -- 'Grey('u, 'count) & 'transfer('u) |-> update('Grey('u, 'count + 1))
  "r4" -- 'Grey('u, 'count) & 'whiteList('u) |-> {
    if ('count < 3)
      fail()
    else
      remove('Grey)
  }
}
```

5.3 Reconciling accounts

The administrator must reconcile accounts every 1000 attempted external money transfers or an aggregate total of one million dollars in attempted external transfers. There are two events: `reconcile` is a propositional event that indicates that all accounts have been reconciled; and `transfer`(*amount*) indicates that *amount* was transferred. Invalid traces have two forms. The first is where there are more than 1000 events between `reconcile` events and the second is where the sum of amounts exceeds one million dollars.

QEA specification. Here the `safe` state indicates some safe amount has been transferred and we can only stay in this state if each transfer is safe. To ensure this a running count and total is kept.

```
qea{
  accept next(start){
    transfer(amount) do[ count:=1; total:=amount ] -> safe
  }
  accept next(safe){
    transfer(amount) if[ count < 1000 and total < 1000000 ]
                     do[ count++; total += amount ] -> safe
    reconcile do[ count:=0; total:= 0] -> safe
  }
}
```

LogFire specification The monitor uses one fact, `Sums(count,total)`, carrying two sums: `count`, which is the number of transfers since the last reconciliation, and `total`, which is the total sum of money transferred since the last reconciliation. An initial `Sums(0,0)` fact is added to the fact memory (`addFact('Sums)(0,0)`) before monitoring starts.

```
class M extends Monitor {
  "r1" -- 'Sums('_, '_) & 'reconcile() |-> update('Sums(0, 0))
  "r2" -- 'Sums('count, 'total) & 'transfer('a) |-> {
    if ('count + 1 > 1000 || 'total + 'a > 1000000)
      fail()
    else
      update('Sums('count + 1, 'total + 'a))
  }

  addFact('Sums)(0,0)
}
```

The monitor above purely uses rules to implement the property. However, we can, as in the QEA monitor, use global variables `count` and `total`. This is shown in the following version, illustrating how LogFire monitors can mix rules and programming:

```
class M extends Monitor {
  var count: Int = 0
  var total: Int = 0

  "r1" -- 'reconcile() |-> {count = 0; total = 0}
  "r2" -- 'transfer('a) |-> {
    if (count + 1 > 1000 || total + 'a > 1000000)
      fail()
    else {
      count += 1; total += 'a
    }
  }
}
```

5.4 Maximum withdrawals

The number of withdrawal operations performed within 10 minutes before a customer logs off is less than or equal to the allowed limit of 3. It is assumed that a trace records the activities of a single user. A user is not required to log off. There are two events of interest: `withdraw`(*time*) records the time that

a customer made a withdrawal and `logoff`(*time*) records the time that the customer logged off.

QEA specification. This QEA can be read as 'we do not reach failure' as it is a negated QEA accepting invalid traces. The QEA is non-deterministic, creating a new configuration per time window. A failure is detected if one of these configurations reaches the **success** state, as the specification is negated. The combination of non-determinism and negation is required as a trace is accepted if *at least one path* reaches an accepting state, therefore we use this approach if we want failure when *all paths* are required to reach an accepting state.

On each `withdraw` event a new configuration is created in the `safe` state with count 1. On each subsequent `withdraw` event (within the time window) this count is increased until it reaches 3 and the configuration is transferred to the `unsafe` state. Any `logoff` occurring within the time window for a configuration in the `unsafe` state will lead to failure. Note that, due to the use of **next** states, configurations will be removed if an event occurs that does not satisfy any transitions i.e. one that happens outside of that configuration's time window.

```
qea{
  Negated
  skip(start){
    withdraw(t1) do[ count:=1 ] -> safe
    withdraw(_) -> start
  }
  next(safe){
    withdraw(t2) if[ t2-t1 <= 10 and count < 3 ] do [ count++ ] -> safe
    withdraw(t2) if[ t2-t1 <= 10 and count = 3 ] do [ count++ ] -> unsafe
  }
  next(unsafe){
    withdraw(t2) if[ t2-t1 <= 10 ] -> unsafe
    logoff(t2)   if[ t2-t1 <= 10 ] -> success
  }
}
```

LogFire specification The monitor uses one fact, `Count(time,count)`, which is created upon each withdrawal event, using the same form of non-determinism as used in the QEA specification. It tracks the number of withdrawals since (and including) that withdrawal, should a logoff occur. It carries two pieces of data: `time`, which is the time of the withdrawal, and `count`, which is the number of withdrawals since then.

```
class M extends Monitor {
  "r1" -- 'withdraw('time) |-> insert('Count('time, 1))
  "r2" -- 'Count('time, 'count) & 'withdraw('time2) |-> {
    if ('time2 - 'time <= 10)
      update('Count('time, 'count + 1))
    else
      remove('Count)
  }
  "r3" -- 'Count('time, 'count) & 'logoff('time2) |-> {
    ensure('time2 - 'time > 10 || 'count <= 3)
    remove('Count)
  }
}
```

5.5 Transaction limit reporting

The property requires that a client's executed transactions must be reported within at most 5 days if the transferred amount exceeds a given threshold of $2,000. A transaction only occurs once. The event `trans`(c, t, a, ts) denotes that the client c performs transaction t, transferring the amount a at timestamp ts. The `report`(t, ts) event denotes that transaction t is reported at timestamp ts.

QEA specification. In this specification, making a transfer for an amount more than 2000 moves us into an `unsafe` state for a transaction. A `report` event within 5 days takes us to a `safe` state. To ensure that failures are captured early, this specification only allows `transfer` events for other clients to occur within the 5 day waiting period.

```
qea{
  Forall(t)
  accept skip(safe){ trans(t,a,ts1) if [ a > 2000 ] -> unsafe }
  skip(unsafe){
    trans(_,_,ts2) if[ ts2-ts1 > 5 ] -> failure
    report(t,ts2)  if[ ts2-ts1 <= 5 ] -> success
  }
}
```

LogFire specification The monitor uses one fact, `Unsafe(t,ts)`, representing the fact that a transaction `t` occurred at time `ts` of more than $2,000$ dollars. This fact is declared hot, meaning that monitoring should not terminate with such a fact in the fact memory – it has to eventually be reported (and within 5 days).

```
class M extends Monitor {
  "r1" -- 'transfer('t, 'a, 'ts) |-> {
    if ('a > 2000) insert('Unsafe('t, 'ts))
  }
  "r2" -- 'Unsafe('t, 'ts1) & 'report('t, 'ts2) |-> {
    ensure('ts2 - 'ts1 <= 5)
    remove('Unsafe)
  }

  hot('Unsafe)
}
```

5.6 Transaction limit authorization

The property requires that executed transactions of any customer must be authorized by some employee before they are executed if the transferred money exceeds a given threshold of $2,000. Authorization only lasts for 21 days and a transaction must be authorized at least 2 days before it is made. A transaction can only occur once. The `trans`(t,a,ts) indicates transaction t occurred at time ts for amount a. The event `auth`(t, ts), indicates the authorization of the transaction t at timestamp ts.

QEA specification. In the `unauth` state transactions can only be below 2,000. The **`auth`** event takes us to the `auth` state, where any transaction can occur as long as we are inside the authorized period.

```
qea{
  Forall(t)
  accept skip(unauth){
    trans(t,a,_) if [ a < 2000 ] -> success
    trans(t,a,_) if [ a >= 2000 ] -> failure
    auth(t,auth_ts) -> auth
  }
  accept skip(auth){
    trans(t,a,ts) if [ ts-auth_ts >= 2 and ts-auth_ts < 21 ] -> success
    trans(t,a,ts) if [ ts-auth_ts < 2  and ts-auth_ts > 21 ] -> failure
    auth(t,auth_ts) -> auth
  }
}
```

LogFire specification The monitor uses one fact, `Auth(t,ts)`, representing the fact that transaction `t` has been authorized at time `ts`. Note how authorizations are updated in case such exist.

```
class M extends Monitor {
  "r1" -- 'auth('t, 'ts) & not('Auth('t, '_)) |-> insert('Auth('t, 'ts))
  "r2" -- 'Auth('t, '_) & 'auth('t, 'ts) |-> update('Auth('t, 'ts))
  "r3" -- 'Auth('t, 'ts1) & 'trans('t, 'a, 'ts2) |-> {
    ensure('a < 2000 || ('ts2 - 'ts1 >= 2 && 'ts2 - 'ts1 < 21))
    remove('Auth)
  }
  "r4" -- 'trans('t, 'a, '_) & not('Auth('t, '_)) |-> {
    ensure('a < 2000)
  }
}
```

5.7 Report approval

The property represents an approval policy for publishing business reports within a company. The property requires that any report must be approved prior to its publication. Furthermore, the property asks that the person who publishes the report must be an accountant and the person who approves the publication must be the accountants manager at the time of the approval. Finally, the approval must happen within at most 10 days before the publication.

The event $\texttt{publish}(a, f, ts)$ denotes the publication of the report f by the accountant a at timestamp ts. The event $\texttt{approve}(m, f, ts)$ denotes the publishing approval of the report f by the manager m at timestamp ts. The event $\texttt{mgr\_S}(m, a)$ marks the time when m starts being manager of accountant a, and the event $\texttt{mgr\_F}(m, a)$ marks the corresponding finishing time. Analogously, $\texttt{acc\_S}(a)$ and $\texttt{acc\_F}(a)$ mark the starting and finishing times when a is an accountant

QEA specification. This specification keeps track of the current managers of accountant a in set M. There are then two states: `nota` and `isa` indicate that the accountant is not approved or is. Then `ts_app` tracks the most recent

time at which the report was legitimately approved so that this can be checked on a `publish` event. In QEA it is assumed that sets are empty on their first occurrence. The **defined** predicate is true if a variable has been given a value.

```
qea{
  Forall(f,a)
  accept next(nota){
    mgr_S(m,a) do [ M.add(m) ] -> nota
    mgr_F(m,a) do [ M.remove(m) ] -> nota
    approve(m,f,ts_app) if [ m in M ] -> nota
    acc_S(a) -> isa
  }
  accept next(isa){
    mgr_S(m,a) do [ M.add(m) ] -> isa
    mgr_F(m,a) do [ M.remove(m) ] -> isa
    approve(m,f,ts_app) if [ m in M ] -> isa
    acc_F(a) -> nota
    publish(a,f,ts_pub) if [ defined(ts_app) and ts_pub - ts_app < 10 ] -> isa
  }
}
```

LogFire specification The monitor uses the following facts: `Mgr(m,a)` to record that manager `m` is manager of accountant `a`; `Acc(a)` to record that `a` is an accountant; and finally `Appr(a,f,ts)` to record that accountant `a` at time `ts` has been approved by one of his/her managers to publish report `f`. Note how a `Appr(a,f,ts)` fact is generated for each accountant `a` that a manager is manager of when he/she approves a report `f`. Note also how approvals are updated in case such exist.

```
class M extends Monitor {
  "r1" -- 'mgr_S('m, 'a) |-> 'Mgr('m, 'a)
  "r2" -- 'Mgr('m, 'a) & 'mgr_F('m, 'a) |-> remove('Mgr)
  "r3" -- 'acc_S('a) |-> 'Acc('a)
  "r4" -- 'Acc('a) & 'acc_F('a) |-> remove('Acc)
  "r5" -- 'approve('m, 'f, 'ts) & 'Mgr('m, 'a) & not('Appr('a, 'f, '_)) |->
    insert('Appr('a, 'f, 'ts))
  "r6" -- 'Appr('a, 'f, '_) & 'approve('m, 'f, 'ts) & 'Mgr('m, 'a) |->
    update('Appr('a, 'f, 'ts))
  "r7" -- 'publish('a, '_, '_) & not('Acc('a)) |-> fail()
  "r8" -- 'publish('a, 'f, '_) & not('Appr('a, 'f, '_)) |-> fail()
  "r9" -- 'Appr('a, 'f, 'ts1) & 'publish('a, 'f, 'ts2) |-> ensure('ts2 - 'ts1 <= 10)
}
```

5.8 Withdrawal limit

The property is rooted in the domain of fraud detection. The property requires that the sum of withdrawals of each user in the last 28 days does not exceed the limit of $10,000. The event `withdraw`$(u, a, ts)$ denotes the withdrawal of the amount a by the user u at timestamp ts.

QEA specification. This is another specification of the form 'there does not exist a path to failure'. This time we capture the behavior required to perform a bad withdraw. On each `withdraw` event we create a new configuration that represents the start of a new 28 day period.

```
qea  {

  Negated
  Exists(u)

  skip(start){
    withdraw(u,a,_) -> start
    withdraw(u,a,ts) do[s:=a] -> withdrawn
    withdraw(u,a,_) if[a > 10000] -> success
  }
  next(withdrawn){
    withdraw(u,a,ts2) if[ ts2-ts < 28 and s+a >  10000 ] -> success
    withdraw(u,a,ts2) if[ ts2-ts < 28 and s+a <= 10000 ]
                      do[s+=a] -> withdrawn
  }
}
```

LogFire specification The monitor uses one fact, `Withdrawn(u,a,ts)`, to record that user `u` withdrew `a` dollars at time `ts` (not including amounts greater than $10,000$). Such a fact is produced for each withdrawal, and is used to monitor the next 28 days from that point, accumulating the withdrawn amounts.

```
class M extends Monitor {
  "r1" -- 'withdraw('u, 'a, 'ts) |-> {
    if ('a <= 10000)
      insert('Withdrawn('u, 'a, 'ts))
    else
      fail()
  }
  "r2" -- 'Withdrawn('u, 'sum, 'ts1) & 'withdraw('u, 'a, 'ts2) |-> {
    if ('ts2 - 'ts1 <= 28) {
      val newSum = 'sum + 'a
      if (newSum <= 10000)
        update('Withdrawn('u, newSum, 'ts1))
      else
        fail()
    } else remove('Withdrawn)
  }
}
```

6 Specification of rover properties

In this set of properties we consider the operation of planetary rovers. The first two properties consider communication and the last three consider internal resource management.

6.1 Rover coordination

This property relates to the self-organization of communicating rovers and captures the situation where (at least) one rover is able to communicate with all other (known) rovers. The property states that there exists a leader (rover) who has pinged every other (known) rover and received an acknowledgement. The events `ping`(*from*,*to*) and `ack`(*to*,*from*) indicate that *from* pinged *to* and *to* acknowledged *from* respectively. The leader does not need to have pinged itself. The set of known rovers are those that ping/ack or have been pinged/acked.

QEA specification. This specification uses quantifier alternation to capture the property that there is one rover `r1` that sends `ping` and receives an `ack` from every other different rover `r2`. The **Join** declaration indicates that the domains of `r1` and `r2` should be joined i.e. if a value is considered for `r1` it should also be considered for `r2` and vice-versa.

```
qea{
  Exists(r1) Forall(r2) Where(r1!=r2) Join(r1,r2)
  skip(start) { ping(r1,r2) -> pinged  }
  skip(pinged){ ack(r2,r1)  -> success }
}
```

LogFire specification. The monitor uses the following facts: `Ping(r1,r2)` to record that rover `r1` pings rover `r2`; `Node(r)` to record that `r` is a node; `Reach(r1,r2)` to record that node `r1` has pinged node `r2`, and that `r2` has acknowledged back; `Cand(r)` to record that rover `r` is a candidate as leader, all nodes are declared as candidates; and finally `End()` records the end of the trace, at which point the final computation is performed. At that point candidates are removed if there is some node they do not reach. If no candidates are left at some point an error is reported, there is no leader.

```
class M extends Monitor {
  "r1" -- 'ping('r1, 'r2) |-> {
    insert('Node('r1))
    insert('Node('r2))
    insert('Ping('r1, 'r2))
  }
  "r2" -- 'ack('r1, 'r2) |-> {
    insert('Node('r1))
    insert('Node('r2))
  }
  "r3" -- 'Node('r) |-> {
    insert('Cand('r))
    insert('Reach('r, 'r))
  }
  "r4" -- 'Ping('r1, 'r2) & 'ack('r2, 'r1) |-> {
    insert('Reach('r1, 'r2))
    remove('Ping)
  }
  "r5" -- 'end() |-> 'End()
  "r6" -- 'End() & 'Cand('r1) & 'Node('r2) & not('Reach('r1, 'r2)) |->
    remove('Cand)
  "r7" -- 'End() & not('Cand('_)) |-> fail()
}
```

The LogFire specification is clearly more complicated than the QEA automaton. In LogFire we are limited since we cannot directly model a universal quantifier nested under an existential quantifier as in the following quantified linear temporal logic formula: $\exists leader \bullet \forall n \bullet \Diamond(ping(leader, n) \wedge \Diamond ack(n, leader))$, where $\Diamond\psi$ has the classical meaning: eventually ψ for some temporal formula ψ.

6.2 Command nesting

If a command with identifier B starts after a command with identifier A has started, then command B must succeed before command A succeeds (last issued – first to succeed). A command can only be started and succeed once. The events `com`(id) and `suc`(id) record the issuing and success of command id respectively.

QEA specification. This specification captures the property that command `c2` is nested inside command `c1`. Note that, due to the symmetry of `c1` and `c2`, for every two values there will be two instances of the QEA considering each command being nested inside the other. As the trace is considered after instantiation, the events `com(c1)` and `com(c2)` are distinct as they will be instantiated with different values. The states capture different stages in each command. Note that the events `com(c2)` and `suc(c2)` on the initial state are necessary if `c2` is not nested inside `c1`. Similarly, the `finishedEarly` state is entered if `c2` only occurs after `c1` succeeds.

```
qea{
  Forall(c1,c2)
  accept next(none){
    com(c2) -> none; suc(c2) -> none
    com(c1) -> startedOne
  }
  accept next(startedOne){
    com(c2) -> startedTwo
    suc(c2) -> finishedEarly
  }
  accept next(startedTwo){
    suc(c2) -> finishedTwo
  }
  accept next(finishedTwo){
    suc(c1) -> finished
  }
  accept next(finishedEarly){
    com(c2) -> finishedEarly; suc(c2) -> finishedEarly
  }
  accept next(finished){}
}
```

LogFire specification. The monitor uses the following facts: `Com(x)` to record that command `x` has been issued (this fact is never deleted); `Suc(x)` to record that command `x` has succeeded (this fact is never deleted); and finally `Ord(x,y)` to record that command `y` has been issued after command `x`, and therefore must succeed before command `x`.

```
class M extends Monitor {
  "r1" -- 'com('x) |-> insert('Com('x))
  "r2" -- 'Com('x) & 'com('x) |-> fail()
  "r3" -- 'Com('x) & 'suc('x) |-> insert('Suc('x))
  "r4" -- 'suc('x) & not('Com('x)) |-> fail()
  "r5" -- 'Suc('x) & 'suc('x) |-> fail()
  "r6" -- 'Com('x) & not('Suc('x)) & 'com('y) |-> 'Ord('x, 'y)
  "r7" -- 'Ord('x, 'y) & 'suc('y) |-> remove('Ord)
  "r8" -- 'Ord('x, 'y) & 'suc('x) |-> fail()
}
```

6.3 Resource lifecycle

This property represents the lifecycle of a resource with respect to a task, as managed by a planetary rover's internal resource management system - or any resource management system in general. The lifecycle goes as follows:

– A resource may be requested by the task

– A requested resource may be denied or granted to the task
– A granted resource may be rescinded or cancelled
– A resource may only be requested by a task if that task does not currently hold the resource
– A granted resource must eventually be cancelled

We use the events $\texttt{request}(t, r)$, $\texttt{deny}(t, r)$, $\texttt{grant}(t, r)$, $\texttt{rescind}(t, r)$ and $\texttt{cancel}(t, r)$ for a task t and resource r.

QEA specification. The specification captures the three valid states for a resource, with respect to a task, and the valid transitions for each state.

```
qea{
  Forall(t,r)
  accept next(free){
    request(t,r) -> requested
  }
  accept next(requested){
    deny(t,r) -> free
    grant(t,r) -> granted
  }
  accept next(granted){
    cancel(t,r) -> free
    rescind(t,r) -> granted
  }
}
```

LogFire specification. The monitor uses the following two facts: `Req(t,r)` to record that resource `r` has been requested by task `t`; and `Grant(t,r)` to record that resource `r` has been granted to task `t`.

```
class M extends Monitor {
  "r1" -- 'request('t,'r) |-> insert('Req('t,'r))
  "r2" -- 'Req('t,'r) & 'deny('t,'r) |-> remove('Req)
  "r3" -- 'Req('t,'r) & 'grant('t,'r) |-> {
    remove('Req)
    insert('Grant('t,'r))
  }
  "r4" -- 'Grant('t,'r) & 'cancel('t,'r) |-> remove('Grant)
  "r5" -- 'deny('t,'r) & not('Req('t,'r)) |-> fail()
  "r6" -- 'grant('t,'r) & not('Req('t,'r)) |-> fail()
  "r7" -- 'request('t,'r) & 'Grant('t,'r) |-> fail()
  "r8" -- 'rescind('t,'r) & not('Grant('t,'r)) |-> fail()
  "r9" -- 'cancel('t,'r) & not('Grant('t,'r)) |-> fail()

  hot('Grant)
}
```

6.4 Resource management

Every resource should only be held by at most one task at any one time. If a resource is granted to a task it should be cancelled before being granted to another task. This is therefore a mutual exclusion property. The event $\texttt{grant}(t, r)$ captures that task t is granted resource r, similarly $\texttt{cancel}(t, r)$ captures that task t releases resource r.

QEA specification. This specification captures the notion of a bad grant. A bad grant occurs when a resource has been granted to a task and is then granted again to any task (including the task holding the resource). It also uses a guard to ensure that the task granted the resource is the task that cancels the resource.

```
qea{
  Forall(r)
  accept next(free){ grant(t1,r) -> granted }
  accept next(granted){
    grant(_,r) -> failure
    cancel(t2,r) if [ t1 = t2 ] -> free
  }
}
```

LogFire specification. The monitor uses one fact, `Granted(t,r)`, representing that task `t` has been granted resource `r`.

```
class M extends Monitor {
  "r1" -- 'grant('t, 'r) & not('Granted('_, 'r)) |-> insert('Granted('t, 'r))
  "r2" -- 'Granted('_, 'r) & 'grant('_, 'r) |-> fail()
  "r3" -- 'Granted('t, 'r) & 'cancel('t, 'r) |-> remove('Granted)
  "r4" -- 'cancel('t, 'r) & not('Granted('t, 'r)) |-> fail()
}
```

6.5 Resource conflict management

This property represents the management of conflicts between resources as managed by a planetary rovers internal resource management system - or any resource management system in general. It is assumed that conflicts between resources are declared at the beginning of operation. After this point resources that are in conflict with each other cannot be granted at the same time. A conflict between resources r_1 and r_2 is captured by the event `conflict`(r_1,r_2) and a conflict is symmetrical. Resources are granted and cancelled using `grant`(r) and `cancel`(r) respectively.

QEA specification. The specification quantifies over two resources and has two separate states representing each resource being granted (after being put in conflict). Note the symmetry of the `conflict` events required to capture the relationship. Elsewhere [53] this property has been specified differently with efficiency in mind; an encoding of the property that would be more efficient to monitor would replace the `r2` quantification with a set that collects all resources in conflict with `r1`. This would be more efficient as the monitoring algorithm is exponential in the number of quantified variables.

```
qea{
  Forall(r1,r2)
  accept skip(start){
    conflict(r1,r2) -> free
    conflict(r2,r1) -> free
  }
  accept skip(free){
    grant(r1) -> granted1
    grant(r2) -> granted2
```

```
  }
  accept next(granted1){
    cancel(r1) -> free
  }
  accept next(granted2){
    cancel(r2) -> free
  }
}
```

LogFire specification. The monitor uses the following facts: `Conflict(r1,r2)` to record that there is a conflict between resources `r1` and `r2` – for each such conflict added its symmetric fact is also added; `Granted(r)` to record that resource `r` has been granted; and finally `Locks(r1,r2)` to record that resource `r1` has been granted, which is in conflict with `r2`, which therefore is locked from being granted also. The `Lock` predicate is needed since LogFire currently does not permit negation of conjunctions in conditions.

```
class M extends Monitor {
  "r1" -- 'conflict('r1, 'r2) |-> {
    insert('Conflict('r1, 'r2))
    insert('Conflict('r2, 'r1))
  }
  "r2" -- 'grant('r) & not('Granted('r)) & not('Locks('_, 'r)) |->
    insert('Granted('r))
  "r3" -- 'Granted('r) & 'grant('r) |-> fail()
  "r4" -- 'Locks('_, 'r) & 'grant('r) |-> fail()
  "r5" -- 'Granted('r1) & 'Conflict('r1, 'r2) |-> 'Locks('r1, 'r2)
  "r6" -- 'Granted('r) & 'cancel('r) |-> remove('Granted)
  "r7" -- 'Locks('r1,'r2) & 'cancel('r1) |-> remove('Locks)
  "r8" -- 'cancel('r) & not('Granted('r)) |-> fail()
}
```

7 Specification of concurrency properties

Finally we consider two properties related to concurrency and synchronization via locks.

7.1 Lock nesting

A thread should release a lock as many times as it acquires the lock. Additionally, locks taken within a call to a method should be released during that call. This property therefore represents a *double nesting* of method calls and lock taking. Abstractly this could be viewed as parenthesis matching for two kinds of parenthesis. The four events of interest are `begin`(t) and `end`(t), which respectively record the beginning and end of a method for thread t, and `lock`(t,l) and `unlock`(t,l), which respectively record the locking and unlocking of lock l by thread t.

QEA specification. This QEA specifies paths to failure for a thread and lock using the negated existential pattern seen earlier. The first path to the `failed` state via the `locked` state is followed when the lock is held when exiting the

method it was taken in. There also exists a set of paths that finish in the `inside` or `locked` states when either a method is not exited or a lock not unlocked. Finally, unlocking a lock that is not locked will also lead to failure. The behavior is captured using the counter `depth` to track the depth of the thread's call stack. Each **`begin`** event creates a new configuration `inside` a method call; this effectively attempts to find a failing path for each suffix of the trace starting with a **`begin`** event. The `count` counter tracks the lock depth.

```
qea{
 Negated
 Exists(t,l)
 skip(outside){
    begin(t) do [ depth:=1 ] -> inside
    begin(t) -> outside
  }
  accept skip(inside){
    begin(t) do [ depth++ ] -> inside
    end(t) if [ depth = 1 ] do [ depth:=0] -> finished
    end(t) if [ depth > 1 ] do [ depth -- ] -> inside
    lock(t,l) do [ count:=1 ] -> locked
    unlock(t,l) -> failed
  }
  skip finished {}
  accept skip (locked){
    lock(t,l) do [ count++ ] -> locked
    unlock(t,l) if [count > 1 ] do [ count  ] -> locked
    unlock(t,l) if [ count =1 ] do [ count:=0 ] -> inside
    begin(t) do [ depth++ ] -> locked
    end(t) if [ depth > 1 ] do [ depth  ] -> locked
    end(t) if [ depth = 1 ] -> failed
  }
  accept skip(failed){}
}
```

LogFire specification The monitor uses the following two facts: `Inside(t,d)` to record that thread `t` is currently at a method activation depth of `d` (it has called methods nested `d` times without returning from any of them); and `Locked(t,l,d,c)` to record that thread `t` has taken lock `l` a total of `c` times while at activation depth level `d`.

```
class M extends Monitor {
  "r1" -- 'begin('t) & not('Inside('t, '_)) |-> insert('Inside('t, 1))
  "r2" -- 'Inside('t, 'd) & 'begin('t) |-> update('Inside('t, 'd + 1))
  "r3" -- 'Inside('t, 'd) & 'end('t) |-> {
    if ('d.int > 1)
      update('Inside('t, 'd - 1))
    else
      remove('Inside)
  }

  "r4" -- 'Inside('t, 'd) & 'lock('t, 'l) & not('Locked('t, 'l, 'd, '_)) |->
    insert('Locked('t, 'l, 'd, 1))
  "r5" -- 'Inside('t, 'd) & 'Locked('t, 'l, 'd, 'c) & 'lock('t, 'l) |->
    update('Locked('t, 'l, 'd, 'c + 1))
  "r6" -- 'Inside('t, 'd) & 'Locked('t, 'l, 'd, 'c) & 'unlock('t, 'l) |-> {
    if ('c > 1)
      update('Locked('t, 'l, 'd, 'c - 1))
    else
      remove('Locked)
  }
  "r7" -- 'Inside('t, 'd) & 'unlock('t, 'l) & not('Locked('t, 'l, 'd, '_)) |->
```

```
    fail()
  "r8" -- 'Inside('t, 'd) & 'Locked('t, '_, 'd, '_) & 'end('t) |-> fail()

  hot('Locked)
}
```

7.2 Lock ordering

This property represents a conservative deadlock-avoidance strategy that prevents cycles between locks by enforcing a partial ordering on locks: a thread can only take a lock L_2 while holding a lock L_1 if L_1 precedes L_2 in the partial ordering. The property states that for every two (different) locks, if they are taken in one order in one part of the system, then they are not taken in the opposite order in another part of the system. The events `lock`(t,l) and `unlock`(t,l) respectively capture the locking and unlocking of lock l by thread t.

QEA specification. This specification quantifies over a pair of threads and a pair of (distinct) locks. If the locks are taken by the first thread in one order, then they cannot be taken in a different order by the second thread.

```
qea{
  Forall(t1,t2,l1,l2)
  Where(l1 != l2)

  accept skip(start){ lock(t1,l1) -> lock1 }
  accept skip(lock1){
    unlock(t1,l1) -> start
    lock(t1,l2) -> lock12
  }
  accept skip(lock12){ lock(t2,l2) -> lock122 }
  accept skip(lock122){
    unlock(t2,l2) -> lock12
    lock(t2,l1) -> failure
  }
}
```

LogFire specification The monitor uses the following two facts: `Locked(t,l)` to record that thread `t` has taken lock `l` and not yet released it; and `Edge(l1,l2)` to record that thread `t` at some point held lock `l1`, while nested acquiring lock `l2`.

```
class M extends Monitor {
  "r1" -- 'lock('t, 'l) |-> insert('Locked('t, 'l))
  "r2" -- 'Locked('t, 'l) & 'unlock('t, 'l) |-> remove('Locked)
  "r3" -- 'Locked('t, 'l1) & 'lock('t, 'l2) |-> insert('Edge('l1, 'l2))
  "r4" -- 'Edge('l1, 'l2) & 'Edge('l2, 'l1) |-> fail()
}
```

8 Summary and discussion

In this section we summarize and discuss our experience specifying the various properties in the two different logics. We reflect on the two logics from a linguistic perspective and make suggestions for improvements to each logic. We also discuss some issues relating to the pragmatic differences between the two methods.

8.1 Relationship to temporal logic

Two approaches to specification of property monitors have been presented, QEA which is automaton-based, and LOGFIRE, which is rule-based. From the point of view of writability and readability it is clear that both logics are low-level in the sense that specifications are somewhat verbose and can be hard to read. The standard alternative approach is some form of temporal logic or regular expressions. In many cases it is likely that these more abstract logics can make specifications easier to write and read. However, note that some properties will not benefit from the abstractions of these higher level logics. For example, the resource lifecycle property in Section 6.3 is suited to a low-level specification style and will likely have a highly convoluted specification in temporal logic.

Note that it is common folklore that temporal logic can be difficult for users to write and read. However, we do believe that many properties can more easily be stated in a combination of temporal logic and regular expressions, as for example found in the SALT language [16], itself influenced by PSL [60]. Occasionally when understanding a property we would draw a time line and plot in events on the time line, not far from the time line notation proposed in TIMEEDIT [56].

The classical way of giving semantics to temporal logic within the model checking community is to translate temporal formulas to automata [33, 44]. Similarly, it has been shown how to translate temporal formulas (LTL) to rules [6]. Temporal logics for parametric monitoring are, however, typically not translated into automata (or rules), but are rather interpreted over the structure of the formulas, which evolve as events are consumed. One reason for this discrepancy in approach within the model checking and runtime verification communities is in part due to lack of automata concepts that involve data. Note that extended state machines (state machines with variables that can be checked in transition guards and updated in transition actions) are not sufficient since variables are global, in contrast to QEA where they are local to a slice. Both QEA and LOGFIRE can be considered as target for translations from parametric temporal logic.

LTL (Linear Temporal Logic) is a more realistic candidate for such translations than CTL (Computation Tree Logic) since a single trace is linear. However, one can imagine monitoring CTL on sets of traces. A transformation of LTL into either logic (QEA or LOGFIRE) would be straightforward, assuming a version of LTL with a finite-trace semantics, as RV is an activity carried out on finite traces only.

In the case of LOGFIRE, since it really is a SCALA API, such translation can be defined as templates in SCALA, as described in [40]. This allows a mix of rule-based programing and temporal logic (in addition to traditional programming). A similar mix of state machines and temporal logic can be found in TRACECONTRACT [9] and DAUT (Data automata) [38, 39]. Similarly, temporal logic can be translated into QEA. We believe that a combination of low-level automata/rules and high-level temporal logic/regular expressions would be a convenient specification formalism. It is interesting to observe that if one allows states/facts to be anonymous (un-named) one obtains systems much related to temporal logic.

This corresponds to having transitions labelled with sequences of events, in contrast to just single events. This can be viewed as a basic abstraction mechanism.

8.2 A few notes on specification styles

Several QEA automata have been stated in *positive form*: describing only valid transitions. This is in contrast to a *negative form*, in which erroneous transitions are called out explicitly, in QEA by leading to a **failure** state and in LogFire by leading to an **error** state. As an example, consider the introductory file usage example. The QEA specification in Figure 2 expresses that a read or write operation is not allowed on a file unless it has been opened, by simply not containing such a transitions out of the `closed` state. In contrast, the LogFire specification in Figure 3 expresses this explicitly as two failure transitions (rules `r5` and `r7`). All properties in LogFire are stated in negative form since positive form formulations are not possible. In QEA one has a choice. It can be debated which of the two forms (positive or negative) that in general is more readable.

Another difference is that QEA supports negation of automata, i.e. the QEA specifies erroneous behavior as success with the understanding that this verdict should be negated. This can be hard to read as one must translate success into failure. The negation is needed for non-deterministic QEA where no paths should lead to failure, as the acceptance condition for the automaton is defined as: *there exists a path to an accepting state*. In LogFire *all paths must lead to an acceptance state*.

Neither QEA nor LogFire support the specification of time as a built-in concept. In both cases time is modeled as time stamps, which are just data like any other data. This leads to a difficulty in determining when time bounds get exhausted since it requires an event with a new time stamp. Time violations are not necessarily detected as soon as they happen but rather when the next event arrives. This is demonstrated in the transaction limit reporting property in Section 5.5.

8.3 Expressiveness and complexity

The expressiveness and complexity of formalisms presented here have not been formally studied at the time of writing. However, it seems plausible that both formalisms are Turing complete, hence equally expressive, and able to express any form of verifiable properties. We here consider LogFire without including all of Scala for writing actions. Of course, if we allow any Scala code to be executed the answer to this question is obvious.

The complexity of monitoring in general depends on the property being monitored and the trace. In the worst case, the monitor may end up storing the entire trace, and in each step search this. For this reason it is important that such search is optimized. In general, however, a monitor stores an abstraction of the so-far observed trace, as a function of the property monitored. This makes monitoring pragmatically possible.

In trace-slicing based techniques such as QEA the number of bindings used in trace-slicing is, in the worst case, exponential in the length of the trace. In practice, values are reused and (for online monitoring) garbage collection (see Sec. 8.5) reduces the set of bindings during monitoring. However, it is often possible to define a specification with fewer quantified variables at the cost of introducing additional guards and assignments. This can dramatically improve the monitoring performance.

8.4 Comments on logics

QEA. One of QEA's main features is that it allows arbitrary interleaving of quantifiers, universal as well as existential. For example, a universal quantification can be nested underneath an existential quantification, which was specifically useful in the specification of the rover coordination property in Section 6.1. Another useful feature of QEA is its support for variables that are local to a slice. LogFire also allows for declaration of variables (since LogFire is a Scala API). However, these are global to the monitor. In LogFire variables local to what corresponds to a slice are modeled as parameters to facts.

One significant drawback of the slicing-based approach of QEA is that a guard on a transition only can test on variables within one slice, and not across slices. The consequence of this restriction is not clear. In LogFire all facts are visible to all rules referring to them. A related disadvantage is that QEA must declare a finite quantifier list, making it difficult to specify either second-order properties (i.e., for all subsets) or properties over a variable number of parameters (i.e., for n rovers for variable n). Both kinds of properties can be handled by the more flexible rule-based approach. Another minor drawback is that the automaton approach in QEA considers states to be distinct. This means that if some variables need updating in several states, such update transitions are needed in all those states, hence causing a repetition of specification. This issue does not occur in LogFire.

During the specification exercise it was recognized that QEA specifications could be simplified by the introduction of pre-defined **success** and **failure** states. Other possible modifications could include the following. QEA specifications contain many state modifiers. A choice could be to introduce defaults, such that for example states by default were accept states. Similarly, one could choose a default amongst the skip/next state modifiers. In TraceContract [9] for example, by default all states are accept and skip states. As we have learned from programming, an if-then-else construct would be useful in order to avoid repeating conditions on transitions.

LogFire. LogFire allows a rule's left-hand side conditions to refer to numerous facts and negations thereof. This yields an expressive power, which in QEA can be partially emulated by introducing sets. Furthermore, LogFire supports transitive closure on facts. The Rete engine will execute rules on a set of facts until a fixed point is reached. This feature is not available in classical automata.

It can be useful for expressing for example reachability properties. This is needed for example if generalizing the lock order property in Section 7.2 to N threads, where N is unknown before monitoring.

LogFire's disadvantages include the following. Due to the fact that it is an API in Scala, user-defined names (event names and fact names) are quoted symbols. This is somewhat inconvenient, and is a consequence of names not being first-class citizens in Scala, as they are not in most programming languages. Finally, facts have to be removed explicitly, in contrast to state machines when taking a transition out of a state.

During the specification exercise it was recognized that LogFire specifications could be simplified by the introduction of **hot** facts (non-accept facts). Other possible modifications could include the following. A fact could be declared as *transient*, meaning that if it occurs in a rule that fires it is removed. LogFire currently does not allow conditions that are negations of conjunctions of facts, as is allowed in the original Rete algorithm. This would be a useful addition. Finally, also in LogFire would an if-then-else construct be useful. The Ruler system [11, 12, 1] has hot and transient facts, as well as an if-then-else construct.

8.5 Under the hood

With respect to implementations, both the slicing algorithm in qea and the augmented Rete algorithm in LogFire use indexing to access states/facts relevant for an incoming event. Future research will expose the exact relationship between the two approaches. Furthermore, there are three pragmatic issues related to monitoring that we have not discussed in depth. They are instrumentation, object identity, and garbage collection. We will briefly explain their relevance. Instrumentation techniques must be used to extract events from running programs/systems; the extracted events might be passed directly to an online monitor or recorded in log files for later processing. To deal with data values, they must have a notion of object identity, i.e., an object such as an Iterator should be consistently recorded using the same identifier. In languages such as Java it is possible to use either an object's *reference identity* (i.e., ==) or *semantic identity* (i.e., `equals`). Usually a property is written with one in mind and getting the correct verdict will depend on using the intended notion of identity. The reference identity of an object is consistent over time, whilst the semantic identify can change, i.e. the semantic identity of a collection may change as its contents changes, making semantic identity inappropriate for monitoring in some cases. Using reference identity requires storing the reference in the monitor (as normally also does semantic identity, but one can get around it, see below). Storing the references can be problematic in garbage-collected languages such as Java, where storing (in the monitor) a reference to an object can prevent the object from being garbage collected when the monitored application no longer refers to it (introducing a memory leak). A monitor can, however, appropriately clean its own data structures to prevent this, which can speed up monitoring. One way to do this in Java is to use reference identity in combination with *weak*

references. Alternatively, one can represent the monitored object by some new object with the same semantic identity, and use semantic identity on this new object thereafter.

9 Conclusion

We have presented two monitoring logics, QEA, which is automaton-based, and LOGFIRE, which is rule-based. These logics can be used for writing monitors directly, or they can be the target of translations from temporal logics. The logics are comparable. However, the distinguishing features of QEA are that it allows existential as well as universal quantification, arbitrarily mixed, as well as variables that are local to slices. The distinguishing features of LOGFIRE are its rule system, where rule conditions can refer to multiple facts and their negations, and where a repeated fixed point evaluation strategy allows for rules to compute the transitive closure, useful for expressing certain reachability properties. QEA is an external DSL whereas LOGFIRE is an internal DSL (and API in SCALA). We showed the application of the two logics to the specification of properties obtained from the 1st international runtime verification competition. Future work includes fully understanding how the two monitoring algorithms relate to each other; improvements on the notations; as well as merging these respective logics with temporal logic.

Acknowledgements We would like to thank the organizers, Ezio Bartocci, Borzoo Bonakdarpour, and Yliès Falcone, of the 1st International Competition of Software for Runtime Verification (CSRV 2014), affiliated with RV 2014 in Toronto, Canada [2]. In addition we would like to thank the participants contributing the properties to the competition.

References

1. RuleR website. http://www.cs.man.ac.uk/∼howard/LPA.html.
2. CSRV 2014. http://rv2014.imag.fr/monitoring-competition.
3. C. Allan, P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, O. Lhoták, O. de Moor, D. Sereni, G. Sittamplan, and J. Tibble. Adding trace matching with free variables to AspectJ. In *OOPSLA'05*. ACM Press, 2005.
4. H. Barringer, Y. Falcone, K. Havelund, G. Reger, and D. Rydeheard. Quantified Event Automata - towards expressive and efficient runtime monitors. In *18th International Symposium on Formal Methods (FM'12), Paris, France, August 27-31, 2012. Proceedings*, volume 7436 of *LNCS*. Springer, 2012.
5. H. Barringer, M. Fisher, D. M. Gabbay, G. Gough, and R. Owens. Metatem: An introduction. *Formal Asp. Comput.*, 7(5):533–549, 1995.
6. H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Program monitoring with LTL in Eagle. In *Parallel and Distributed Systems: Testing and Debugging (PADTAD'04), Santa Fee, New Mexico, USA*, volume 17 of *IEEE Computer Society*, April 2004.

7. H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-based runtime verification. In *VMCAI*, volume 2937 of *LNCS*, pages 44–57. Springer, 2004.
8. H. Barringer, A. Groce, K. Havelund, and M. Smith. Formal analysis of log files. *Journal of Aerospace Computing, Information, and Communication*, 7(11):365–390, 2010.
9. H. Barringer and K. Havelund. TraceContract: A Scala DSL for trace analysis. In *17th International Symposium on Formal Methods (FM'11), Limerick, Ireland, June 20-24, 2011. Proceedings*, volume 6664 of *LNCS*, pages 57–72. Springer, 2011.
10. H. Barringer, K. Havelund, D. Rydeheard, and A. Groce. Rule systems for runtime verification: A short tutorial. In *Proc. of the 9th Int. Workshop on Runtime Verification (RV'09)*, volume 5779 of *LNCS*, pages 1–24. Springer, 2009.
11. 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.
12. H. Barringer, D. E. Rydeheard, and K. Havelund. Rule systems for run-time monitoring: from Eagle to RuleR. *J. Log. Comput.*, 20(3):675–706, 2010.
13. D. A. Basin, F. Klaedtke, and S. Müller. Policy monitoring in first-order temporal logic. In T. Touili, B. Cook, and P. Jackson, editors, *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, Proceedings*, volume 6174 of *LNCS*, pages 1–18. Springer, 2010.
14. A. Bauer, J.-C. Küster, and G. Vegliach. From propositional to first-order monitoring. In *Runtime Verification - 4th Int. Conference, RV'13, Rennes, France, September 24-27, 2013. Proceedings*, volume 8174 of *LNCS*, pages 59–75. Springer, 2013.
15. A. Bauer, M. Leucker, and C. Schallhart. The good, the bad, and the ugly, but how ugly is ugly? In *Proc. of the 7th Int. Workshop on Runtime Verification (RV'07)*, volume 4839 of *LNCS*, pages 126–138, Vancouver, Canada, 2007. Springer.
16. A. Bauer, M. Leucker, and J. Streit. SALT – structured assertion language for temporal logic. In Z. Liu and J. He, editors, *Formal Methods and Software Engineering*, volume 4260 of *Lecture Notes in Computer Science*, pages 757–775. Springer Berlin Heidelberg, 2006.
17. E. Bodden. MOPBox: A library approach to runtime verification. In *Runtime Verification - 2nd Int. Conference, RV'11, San Francisco, USA, September 27-30, 2011. Proceedings*, volume 7186 of *LNCS*, pages 365–369. Springer, 2011.
18. F. Chen and G. Roşu. Parametric trace slicing and monitoring. In *Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'09)*, volume 5505 of *LNCS*, pages 246–261, 2009.
19. Clips website: http://clipsrules.sourceforge.net.
20. H. C. Cruz. Optimisation techniques for runtime verification. Master's thesis, University of Manchester, 2014.
21. M. D'Amorim and K. Havelund. Event-based runtime verification of Java programs. In *Workshop on Dynamic Program Analysis (WODA'05)*, volume 30(4) of *ACM Sigsoft Software Engineering Notes*, pages 1–7, 2005.
22. N. Decker, M. Leucker, and D. Thoma. Monitoring modulo theories. In E. Ábrahám and K. Havelund, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Grenoble, France, April 7-11, 2014. Proceedings*, volume 8413 of *LNCS*, pages 341–356. Springer, 2014.
23. R. B. Doorenbos. *Production Matching for Large Learning Systems.* PhD thesis, Carnegie Mellon University, Pittsburgh, PA, 1995.

24. Drools website: http://www.jboss.org/drools.
25. Drools functional programming extensions website: https://community.jboss.org/wiki/FunctionalProgrammingInDrools.
26. D. Drusinsky. The temporal rover and the ATG rover. In *SPIN Model Checking and Software Verification*, volume 1885 of *LNCS*, pages 323–330. Springer, 2000.
27. D. Drusinsky. *Modeling and Verification using UML Statecharts*. Elsevier, 2006. ISBN-13: 978-0-7506-7949-7, 400 pages.
28. Y. Falcone, J.-C. Fernandez, and L. Mounier. Runtime verification of safety-progress properties. In *Proc. of the 9th Int. Workshop on Runtime Verification (RV'09)*, volume 5779 of *LNCS*, pages 40–59. Springer, 2009.
29. Y. Falcone, J.-C. Fernandez, and L. Mounier. What can you verify and enforce at runtime? *J Software Tools for Technology Transfer*, 14(3):349–382, 2012.
30. Y. Falcone, K. Havelund, and G. Reger. A tutorial on runtime verification. In M. Broy and D. Peled, editors, *Summer School Marktoberdorf 2012 - Engineering Dependable Software Systems, to appear*. IOS Press, 2013.
31. C. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19:17–37, 1982.
32. M. Fusco. Hammurabi - a Scala rule engine. In *Scala Days 2011, Stanford University, California*, 2011.
33. R. Gerth, D. Peled, M. Y. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In P. Dembinski and M. Sredniawa, editors, *In Protocol Specification Testing and Verification (PSTV)*, volume 38, pages 3–18. Chapman & Hall, 1995.
34. J. Goubault-Larrecq and J. Olivain. A smell of ORCHIDS. In *Proc. of the 8th Int. Workshop on Runtime Verification (RV'08)*, volume 5289 of *LNCS*, pages 1–20, Budapest, Hungary, 2008. Springer.
35. A. Groce, K. Havelund, and M. H. Smith. From scripts to specifications: the evolution of a flight software testing effort. In *32nd Int. Conference on Software Engineering (ICSE'10), Cape Town, South Africa*, ACM SIG, pages 129–138, 2010.
36. S. Hallé and R. Villemaire. Runtime enforcement of web service message contracts with data. *IEEE Transactions on Services Computing*, 5(2):192–206, 2012.
37. K. Havelund. Runtime verification of C programs. In *Proc. of the 1st TestCom/-FATES conference*, volume 5047 of *LNCS*, Tokyo, Japan, 2008. Springer.
38. K. Havelund. Data automata in Scala. In M. Leucker and J. Wang, editors, *8th International Symposium on Theoretical Aspects of Software Engineering, TASE 2014, Changsha, China, September 1-3. Proceedings*. IEEE Computer Society Press, 2014.
39. K. Havelund. Monitoring with data automata. In T. Margaria and B. Steffen, editors, *6th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation. Track: Statistical Model Checking, Past Present and Future (organized by Kim Larsen and Axel Legay), Corfu, Greece, October 8-11. Proceedings*, volume 8803 of *LNCS*, pages 254–273. Springer, 2014.
40. K. Havelund. Rule-based runtime verification revisited. *Software Tools for Technology Transfer (STTT)*, April 2014. Published online.
41. K. Havelund and A. Goldberg. Verify your runs. In *Verified Software: Theories, Tools, Experiments, VSTTE 2005*, pages 374–383, 2008.
42. K. Havelund and G. Roşu. Efficient monitoring of safety properties. *Software Tools for Technology Transfer*, 6(2):158–173, 2004.
43. K. Havelund and G. Rosu. Monitoring programs using rewriting. In *16th ASE conference, San Diego, CA, USA*, pages 135–143, 2001.

44. G. J. Holzmann. *The Spin Model Checker – Primer and Reference Manual*. Addison-Wesley, 2004.
45. Jess website: http://www.jessrules.com/jess.
46. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In J. L. Knudsen, editor, *Proc. of the 15th European Conference on Object-Oriented Programming*, volume 2072 of *LNCS*, pages 327–353. Springer, 2001.
47. C. Lee, D. Jin, P. O. Meredith, and G. Roşu. Towards categorizing and formalizing the JDK API. Technical Report http://hdl.handle.net/2142/30006, Department of Computer Science, University of Illinois at Urbana-Champaign, March 2012.
48. I. Lee, S. Kannan, M. Kim, O. Sokolsky, and M. Viswanathan. Runtime assurance based on formal specifications. In *PDPTA*, pages 279–287. CSREA Press, 1999.
49. M. Leucker and C. Schallhart. A brief account of runtime verification. *Journal of Logic and Algebraic Programming*, 78(5):293–303, may/june 2008.
50. D. Luckham, editor. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley, 2002.
51. P. Meredith, D. Jin, D. Griffith, F. Chen, and G. Roşu. An overview of the MOP runtime verification framework. *Software Tools for Technology Transfer (STTT)*, 14(3):249–289, 2012.
52. A. Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science*, pages 46–57. IEEE Computer Society, 1977.
53. G. Reger. *Automata Based Monitoring and Mining of Execution Traces*. PhD thesis, University of Manchester, 2014.
54. G. Reger, H. C. Cruz, and D. Rydeheard. MARQ: monitoring at runtime with QEA. In *Proceedings of the 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'15)*, 2015.
55. Rooscaloo website: http://code.google.com/p/rooscaloo.
56. M. Smith, G. Holzmann, and K. Ettessami. Events and constraints: a graphical editor for capturing logic properties of programs. In *5th Int Sym. on Requirements Engineering, Toronto, Canada*, volume 55(2), pages 14–22. 2001.
57. V. Stolz. Temporal assertions with parameterized propositions. In *Proc. of the 7th Int. Workshop on Runtime Verification (RV'07)*, volume 4839 of *LNCS*, pages 176–187, Vancouver, Canada, 2007. Springer.
58. V. Stolz and E. Bodden. Temporal assertions using AspectJ. In *Proc. of the 5th Int. Workshop on Runtime Verification (RV'05)*, volume 144(4) of *ENTCS*, pages 109–124. Elsevier, 2006.
59. V. Stolz and F. Huch. Runtime verification of concurrent Haskell programs. In *Proc. of the 4th Int. Workshop on Runtime Verification (RV'04)*, volume 113 of *ENTCS*, pages 201–216. Elsevier, 2005.
60. M. Vardi. From Church and Prior to PSL. In O. Grumberg and H. Veith, editors, *25 Years of Model Checking*, volume 5000 of *Lecture Notes in Computer Science*, pages 150–171. Springer Berlin Heidelberg, 2008.