Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

Rule-based Runtime Verification Revisited

Klaus Havelund

Jet Propulsion Laboratory
California Institute of Technology
California, USA

Received: date / Revised version: date

Abstract. Runtime verification (RV) consists in part of checking execution traces against user-provided formalized specifications. Throughout the last decade many new systems have emerged, most of which support specification notations based on state machines, regular expressions, temporal logic, or grammars. The field of Artificial Intelligence (AI) has for an even longer period of time studied rule-based production systems, which at a closer look appear to be relevant for RV, although seemingly focused on slightly different application domains, such as for example business processes and expert systems. The core algorithm in many of these systems is the Rete algorithm. We have implemented a rule-based system, named LogFire, for runtime verification, founded on the Rete algorithm, as an internal DSL in the Scala programming language (in essence a library). Using Scala's support for defining DSLs allows to write rules elegantly as part of Scala programs. This combination appears attractive from a practical point of view. Our contribution is part conceptual in arguing that such rule-based frameworks originating from AI are suited for RV. Our contribution is technical by implementing an internal rule DSL in Scala; by illustrating how specification patterns can easily be encoded that generate rules, and by adapting and optimizing the Rete algorithm for RV purposes. An experimental evaluation is performed comparing to six other trace analysis systems. LogFire is currently being used to process telemetry from the Mars Curiosity rover at NASA's Jet Propulsion Laboratory.

1 Introduction

Runtime Verification (RV) consists of monitoring the behavior of a system, either on-the-fly as it executes, or post-mortem after its execution, for example by analyzing log files. Although this task sounds much easier than full-fledged formal verification of all possible executions of a program, this task is challenging. From an *efficiency* (algorithmic) point of view the challenge consists of fast processing of events that carry data. For example, if a lock is released, the data are the particular lock object; if a task is being terminated, the data are the particular task id. When a monitor receives an event, it has to efficiently locate what part of the monitor is relevant to activate, as a function of the data carried by the event. The monitor has to *match* the right sub-monitor. This is called the *matching problem.* From an *expressiveness* point of view, a logic should be as expressive as possible. From a *elegance* point of view a logic should be easy to use and succinct for simple properties.

The problem has been addressed in several monitoring systems within the last years. These systems usually implement specification languages which are based on formalisms such as state machines, regular expressions, temporal logic, or grammars. The most efficient of these, however, tend to have limited expressiveness as discussed in [15]. Our own work has focused on expressiveness. It can be observed that rule-based programming seems like an attractive approach to monitoring, as exemplified by the Ruler system [23,10]. In its simplest form, a rule-based specification consists of a set of rules of the form:

$$condition_1, \ldots, condition_n \Rightarrow action$$

The state of a rule system can abstractly be considered as consisting of a set of *facts*, referred to as the *fact memory*, where a fact is a data record, a mapping from field names to values. A fact represents a piece of observed information about the monitored system. A condition in a rule's left-hand side can check for the presence or absence of a particular fact, and the action on the right-hand side of the rule can add or delete facts, produce error messages, or cause other side effects. 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.

Within the field of Artificial Intelligence (AI) rule-based production systems have been well studied, for example in the context of expert systems and business rule systems. In such systems a specification[1] is likewise a set of rules, although with a different interpretation that in RULER. The RETE algorithm [37] is the base of many AI rule systems. This algorithm has acquired "a reputation for extreme difficulty" [59], making up close to 1000 lines of pseudo-code (described in [30]), which is considerable for a theoretic algorithm. The core idea is, however, simple. The algorithm maintains a network of facts, avoiding to re-evaluate all conditions in each rule's left-hand side each time the fact memory changes. Our primary goal with this work has been to understand how well this algorithm serves to solve the runtime verification task, how it relates to the RULER algorithm, and hence attempt to bridge the two communities, one anchored in artificial intelligence and one anchored in formal methods. A discussion of this work in its initial phase was first presented in [46] and later in [47], both short papers.

A secondary goal has been to study integration of an RV formalism into a high-level programming language, in this case SCALA, which is well suited for defining so-called *internal* DSLs. A notation for writing monitors can be conceived as a DSL (Domain Specific Language) [38]. Most frameworks developed for runtime verification are *external DSLs*: separate, standalone, languages, with their own parsers. In contrast, an *internal* DSL extends an existing programming language. A typical way of achieving this is to define the DSL as an API in the host programming language. Generally, the arguments for an internal DSL are: limited implementation effort due to direct executability of DSL constructs, feature richness through inheriting the host language's constructs, and tool inheritance, i.e. it becomes possible to directly use all the tool support available for the host language, such as IDEs, editors, debuggers, static analyzers, and testing tools. Often one wants to write advanced properties for which a simple logic does not suffice, including counting and collecting statistics. In a programming language this all becomes straightforward.

Our contributions are as follows. First of all, we have implemented a rule-based system based on the RETE algorithm in the SCALA programming language as an internal DSL, essentially extending SCALA with rule-based programming. Second, we show how it is relatively straight-forward to define specification patterns, such as fragments of temporal logic and time lines, instances of which are translated to rules. An interesting nuance is that these templates allow data parameterized events. Third, we have made some modifications to the RETE algorithm to make it suitable for the RV problem, including fitting it for event processing (as opposed to fact processing) and optimizing it with fast indexing to handle commonly occurring RV scenarios. Finally, we have performed experiments comparing the resulting implementation with six other runtime verification and rule-based systems. Among the test data are log data from the Mars rover Curiosity, developed as part of the MSL (Mars Science Laboratory) mission at NASA's Jet Propulsion Laboratory (JPL) [8]. Our conclusion is, that with such modifications the RETE algorithm is suited for RV, although some of its advantages might not be of critical importance for the RV problem. The implemented algorithm appears competitive relative to the state-of-the-art rule-based system DROOLS [5], but is not as fast as the, to our knowledge, fastest existing state-of-the-art runtime verification system MOP [58].

The paper is organized as follows. Section 2 outlines related work. Section 3 illustrates the LOGFIRE DSL through the specification of a monitor for a set of requirements concerning a simple planetary rover resource management system. Section 4 shows how specification patterns conveniently can be programmed and used. Section 5 describes the RETE algorithm, based on the presentation in [30]. Section 6 discusses the suitability of the RETE algorithm for the RV problem, and then describes the modifications and optimizations we have made to the algorithm to suit the RV problem. Section 7 explains how the internal SCALA DSL has been implemented. Section 8 presents experiments made, comparing performance with other systems. Section 9 concludes the paper. The paper contains SCALA program text, but although knowledge of SCALA is an advantage, it is not a prerequisite for reading the paper.

2 Related Work

2.1 Related Work on Monitoring

Runtime verification [48,56] as a field has delivered several systems over the last decade. Initial systems could only handle propositional events (such as *GateOpen* and *GateClose*), not carrying data: Temporal Rover [31], MAC [55], and JAVA PathExplorer [50,49]. Later work has studied such propositional monitoring logics from a more theoretic point of view, including notions such as 4 valued logics [26] and monitorabillity [35]. More recently there has been a growing interest in so-called *parametric* properties where events carry data values. The challenges here are expressiveness of logics and efficiency of

[1] The term *program* is normally used within the AI community to refer to such a set of rules. We shall use the term *specification* in this paper to be consistent across the systems discussed.

monitoring algorithms. The first systems to handle parameterized events appeared around 2004, and include such systems as EAGLE [17], HAWK [29], JLO [62], TRACEMATCHES [14], and MOP [28,58]. Several systems have appeared since then. RV systems usually implement specification languages which are based on formalisms such as state machines [32,42,45,28,34,19], regular expressions [14,28], temporal logic [55,31,50,17,63,29,62,61, 28,19,24,44,25], or grammars [28]. Some systems based on Linear Temporal Logic (LTL) [60] apply rewriting of LTL formulas, inspired by [41]. These include for example [31,50,17,63,62,19,44]. An example of a rewrite rule is $p \mathsf{U} q = q \vee (p \wedge \bigcirc(p \mathsf{U} q))$, meaning: $p \mathsf{U} q$ (p until q) is true if q is true now, or: p is true now, and in the next step, $p \mathsf{U} q$ holds. Each new event causes the current LTL term to be rewritten into a new term representing the formula that must hold in the next step. This form of rewriting is different from the form of rewriting taking place in rule-based systems. Other LTL-based systems translate temporal logic formulas to state machines before monitoring starts [28].

The TRACEMATCHES (regular expressions) and MOP (logic independent) systems stand out as being amongst the most efficient of these systems, with MOP claimed to be the fastest [58]. In MOP an approach is applied referred to as *parametric trace slicing*. As described in [28, 58], and also further discussed in [15], here a trace of data carrying events is, from a semantic point of view, sliced to a set of propositional traces containing propositional events not carrying data (one trace for each binding of data parameters), which are then fed to propositional monitors. In practice, however, the state of a monitor contains, simplified viewed, a mapping from bindings of parameter values to propositional monitor states. Consider as an example a specification stating that locks should be acquired and released in alternating order. Specifically, we are interested in monitoring events of the form: $\mathit{acquire}(L)$ and $\mathit{release}(L)$, for some universally quantified parameter L, against the regular expression: $(\mathit{acquire}(L)\ \mathit{release}(L))^*$. Consider now a trace of the form (events are separated by dots '.'):

$$\mathit{acquire}(l_{42}).\mathit{release}(l_{42}).\mathit{release}(l_{68})$$

Simplified viewed, MOP works as follows. After the first event is processed, the binding $[L \mapsto l_{42}]$ is mapped to a state where a release is expected next. Upon processing the next event $\mathit{release}(l_{42})$, the binding $[L \mapsto l_{42}]$ can be immediately constructed just from (i) this event and (ii) the quantified parameter L of the event, and then be used directly to look up the monitoring state of the corresponding propositional property, that in this case luckily expects a propositional release event. As we shall see in Section 8, this indexing approach results in an impressive performance.

From an expressiveness point of view, however, as pointed out in [15], parametric trace slicing suffers from two limitations. First of all, it is not possible to write a property where an event refers to two or more different variables. There has to be a unique such, like L in this case. This is because the indexing approach relies on being able to create the binding $[L \mapsto l_{42}]$ just from the observed event $\mathit{release}(l_{42})$. It is for example therefore not possible to express the property that at most one lock should be acquired at any one time, since this would require to formulate a property over events $\mathit{acquire}(L_1)$ and $\mathit{acquire}(L_2)$ (also referred to as the *"Talking Philosophers"* problem in [15]). Second, the approach relies on the fact that arguments to an event do not change in a property, preventing for example counting (also referred to as the *"Auction Bidding"* problem in [15]).

A separate line of work has focused on more expressive logics, able to handle such properties mentioned above. This work has taken two forms: external DSLs (with focus on the logic) and internals DSLs (with focus also on the integration of the logic into a programming language). EAGLE [17] is an external DSL, a linear time mu-calculus for monitoring, with past time as well as future time operators. Although attractive, the implementation appears complex. This observation lead to the design of the RULER system [22,23,10], which supports a rule-based specification language. RULER, however, is inspired by METATEM [16] and not by RETE implementations. RULER has subsequently lead us to study the RETE algorithm as described in this paper, in order to determine the relevance of RETE for runtime verification. LOGSCOPE [21,43,18] is a data parameterized state machine oriented derivative of RULER, implemented for analyzing log files at JPL, for testing of the Mars Curiosity rover.

Our first internal DSL was TRACECONTRACT [19, 20], an embedding in the object-oriented and functional programming language SCALA, supporting data parameterized state machines, temporal logic, and a simple form of rule-based programming. State machines are in TRACECONTRACT defined as a so-called *shallow* internal DSL, where as many of the host language's language constructs as possible are made part of the DSL. For example, SCALA's pattern matching and notion of partial functions are used to define the notion of state transitions. Temporal logic, on the other hand, is represented in TRACECONTRACT as a *deep* embedding. In a deep embedding, data structures in the host language are used to represent DSL constructs in an explicit manner (Abstract Syntax), such that they can be processed. In [40] it is argued that the advantage of a deep embedding is that "We 'know' the code of the term, for instance we can print it, compute its length, etc", whereas the advantage of a shallow embedding is that "we do not know the code, but we can run it". The advantage of a shallow internal DSL of course is the re-use of programming language constructs. LOGFIRE, described in this paper, is a mixed *deep* and *shallow* internal DSL, deep in the sense that some core syntactic features (specifically left-hand sides of rules) are defined as data structures. This allows us to more easily analyze and optimize such specifications, although forcing us to re-invent the notion of for example pattern matching.

A *deep* embedding of LTL in HASKELL is described in [63]. In contrast to TRACECONTRACT, it handles data parameterization by a concept called formula templates instantiated for all possible permutations of propositions. MOPBOX [27] is a JAVA library for monitoring, hence a shallow embedding, offering a re-implementation of efficient indexing algorithms contained in MOP [28,58], but defining the interface as an API, as in TRACECONTRACT and LOGFIRE. The user can for example define a state machine as a sequence of JAVA statements updating a state machine data structure. The tool is modular, allowing to experiment with different monitoring/indexing algorithms.

Quantified Event Automata [15] is an automaton concept for monitoring parameterized events, which extends the parametric trace slicing approach taken by MOP by allowing event names to be associated with multiple different variable lists (not allowed in MOP), by allowing non-quantified variables to vary during monitoring, and by allowing existential quantification in addition to universal quantification. This results in a strictly more expressive logic. This work arose from an attempt to understand, reformulate and generalize parametric trace slicing, and more generally from an attempt to explore the spectrum between MOP and more expressive systems such as EAGLE and RULER.

ORCHIDS [42] is a comprehensive state machine based monitoring framework created for intrusion detection. It dynamically spawns state machine monitors as events are processed. Abstract interpretation is used to safely kill useless monitors, including monitors which will not detect anything, as well as monitors that are subsumed by others that will report shorter runs.

An extension of future time Linear Temporal Logic (LTL) with a binding operator ($pattern \rightarrow \varphi$, for some LTL formula φ) is presented in [61]. This work was preceded and inspired by [62]. The binding operator has similarities with a rule, with a binding left-hand side, that has to match an incoming event before the right-hand side formula becomes active. This form of binding operator is also found in systems such as LOGSCOPE [21,43,18] and TRACECONTRACT [19,20]. The logic offers universal and existential quantification of variables bound. During monitoring an alternating automaton is constructed on-the-fly. A state is a control state and a binding, as is also the case in LOGFIRE. Note that LOGFIRE can refer to such states in conditions, and hence express past time properties.

A first-order future time linear temporal logic, LTL-FO$^+$, for paramererized monitoring of web-services is presented in [44]. The logic permits universal and existential quantification over data occurring in XML messages, which are being transmitted between a server and clients. The logic is particular in allowing path expressions over the XML structures. An LTL-FO$^+$ formula is interpreted as a so-called *watcher* automaton, which is built on-the-fly by rewriting formulas, inspired by [41], and adapted to first-order quantification; similar to the other rewriting-based systems mentioned earlier. The quantification in these logics is over the data occurring in the trace, as is the case in LOGFIRE.

A metric first-order temporal logic, MFOTL, for monitoring, with time constraints as well as universal and existential quantification over data, is presented in [24]. The logic includes past time operators in addition to future time operators, and is described as an expressive fragment of a temporal logic, which can be effectively monitored. A formula is rewritten to a formula in a monitorable fragment, permitting matching against data parameterized events. This logic, as well as the other first-order logics mentioned, has limitations in not being able to express aggregation of data, such as for example counting. In general the logic operates only with events, in contrast to LOGFIRE, which can also operate with parameterized facts, and hence aggregations.

Another first-order linear temporal logic, LTL$^{\mathrm{FO}}$, for parameterized monitoring is presented in [25]. The temporal logic, offering first-order quantification, is converted to what is referred to as *spawning automata*. A spawning automaton is in principle an automaton associated with a set of constraints representing the binding of quantified variables. This is very similar to the concept of automata in LOGSCOPE and TRACECONTRACT, where, however, the binding of variables is represented by concrete mappings from names to values, similar also to how bindings are represented in LOGFIRE and RULER. Several complexity results are presented in [25]. The automaton solution does not seem to be optimized.

2.2 Related Work on Rule Processing

There are several state-of-the-art implementations of the RETE algorithm, including DROOLS [5], JESS [7] and CLIPS [2]. These systems offer external DSLs for writing rules. In this work we have focused on DROOLS for comparison, since it is implemented in JAVA and since it is freely available from the web. DROOLS is interesting since it can perform backward chaining (from conclusions, as PROLOG) as well as forward chaining (from assumptions, as RETE, and as implemented in LOGFIRE). DROOLS also performs indexing as we do, although details on this have to our knowledge not been published in a detailed form. For a discussion of DROOLS optimizations the reader is referred to [3]. The DROOLS project has an effort ongoing, defining functional programming extensions to DROOLS [4]. In contrast, by embedding a rule system in an object-oriented and functional language, as done in LOGFIRE, we can leverage the already existing host language features.

DROOLS supports a notion of events, which are facts with a limited life time. These events, however, are not as short-lived as possibly desirable in runtime verification. This leads to problems in defining certain properties. Consider for example the following two rules, where e is an event, and F is a fact:

$$\begin{array}{lll} r_1 & : & e, F \Rightarrow \mathit{remove}(F) \\ r_2 & : & e, \neg F \Rightarrow \mathit{error} \end{array}$$

Assume that event e occurs, and that fact F is present in the fact memory. The first rule handles the case where fact F is present - in which case the action is to delete it. The second rule handles the case where F is not present, and will immediately get triggered as a result of the fact F being deleted by the first rule firing (within the same event cycle). As a result an error will be reported although intuitively this should not be the case. The problem here is that the event e "stays around" for too long, thereby causing damage. In DROOLS the specification will consequently have to be formulated differently (which is possible to do). We have modified RETE to process events with immediate removal (before right-hand side are evaluated essentially), such that such properties can be stated in their natural form. The event concept in DROOLS is inspired by the concept of *Complex Event Processing* (CEP), described by David Luckham in 2002 [57]. This concept in many ways is very much related to RV. CEP is concerned with processing streams of events in (near) real time, where the main focus is on the correlation and composition of atomic events into complex (compound) events. Note that a rule-based system in general, including LOGFIRE and DROOLS, allows for such abstraction, as discussed elsewhere in this paper.

Two rule-based internal DSLs for SCALA exist: HAMMURABI [39] and ROOSCALOO [9]. 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 [9] is RETE based, but is not documented in any form other than experimental code. Our implementation is not based on any of these systems. A RETE-based system for aspect-oriented programming with history pointcuts is described in [51]. The system offers a small past time logic, which is implemented as a modification of the RETE algorithm. This is in contrast to our approach, where we maintain the RETE algorithm, and instead write rules that represent the desired properties.

3 The LogFire DSL

In this section we shall illustrate LOGFIRE by stating a collection of requirements for the resource management system for a planetary rover, and then formalizing them as a LOGFIRE monitor. The requirements as well as formalizations are inspired by real systems as well as other literature, such as [52] and [36]. However, parts of the examples are artificially made up in order to not reveal protected information.

3.1 Requirements for a Resource Management System

A planetary rover runs a collection of tasks in parallel. Each task handles a specific application, such as imaging, controlling the robot arm, communication with earth, and driving. Tasks use resources, for example motors. A *resource arbiter* manages resource allocation, ensuring for example that a resource is only used by one task at a time, and that resource deadlocks do not occur.

A task requests a resource, upon which it is either granted or denied by the arbiter, and if granted, the task eventually is expected to release the resource at a later point in time. In order to avoid deadlocks, a partial order on resources is imposed, which has to respected: if a resource r_1 is ordered before a resource r_2, then a task should not be granted r_1 after having been granted r_2, until r_2 is released again.

We first settle on the events we expect to monitor. Events are short-lived instantaneous observations, which will trigger the evaluation of rules. The event types are as follows (all occurrences of parameter s is a time stamp):

$\mathit{before}(r_1, r_2)$	:	resource r_1 is ordered before r_2
$\mathit{request}(s, t, r)$	:	task t requests resource r
$\mathit{deny}(s, t, r)$	:	task t is denied resource r
$\mathit{grant}(s, t, r)$	:	task t is granted resource r
$\mathit{release}(s, t, r)$	:	task t releases resource r
$\mathit{end}()$	:	the end of the log is reached

Consider next the following four informal requirements that logs containing instances of these event types have to satisfy:

- ***Release***: A resource granted to a task should eventually be released by that task.
- ***NoRelease***: A resource can only be released by a task, if it has been granted to that task, and not yet released.
- ***NoGrant***: As long as a resource is granted to a task, it cannot be granted again, neither to that task nor to any other task. Also, a resource cannot be granted to a task if it is ordered before another resource already granted to that task.
- ***Deny***: A task requesting a resource that is already granted, or which is ordered before a resource already granted to that task, should be denied the resource within 10 seconds. There should be no more than three denials in a log.

3.2 Formalizing Requirements as Rules

We shall now formalize these four requirements. The main component of LOGFIRE is the trait[2] *Monitor*, which

[2] A *trait* in SCALA is a module concept closely related to the notion of an *abstract class*, as for example found in JAVA. Traits,

any user-defined monitor must extend to get access to the constants and methods provided by the rule DSL.

3.2.1 Declaration of Events and Facts

As already mentioned, *events* are short-lived instantaneous observations about the system being monitored. In contrast, *facts* are long-lived pieces of information stored in the fact memory of the rule system. They are added and removed explicitly by rule right-hand sides. As it turns out, we shall use the same representation of events and facts, only their life-span distinguishes them. Events can be considered short-lived facts. The first task is to declare event and fact names. These are in LogFire represented as values of the Scala type *Symbol*, which contains quoted identifiers, such as *'a*, *'f42*, *'request*, etc. In order to avoid writing quotes we allow to declare unquoted event and fact names. The following trait declares all events and facts that we will need:

```
trait ResourceMonitor extends Monitor {
  val before, request, deny,
      grant, release, end = event
  val Before, Granted, Denied = fact
}
```

This allows us to for example write *request* instead of *'request* in our monitors[3].

3.2.2 General Form of Rules

All subsequent monitors will extend *ResourceMonitor*, and will contain rules of the form:

name `--` *condition*$_1$ & ... & *condition*$_n$ `|->` *action*

A rule is defined by a name, a left-hand side consisting of a conjunction of conditions, and a right-hand side consisting of an action to be executed if all the conditions match the fact memory. A condition is a pattern matching facts or events in the fact memory, or, as we shall later see, the negation of a pattern, being true if such a fact does not exist in the fact memory. Arguments to conditions are variables (quoted identifiers of the type *Symbol*) or constants. The first occurrence of a variable in a left-hand side condition is binding, and subsequent occurrences in that rule must match this binding. An action can be adding facts, deleting facts, or generally be any Scala code to be executed when a match for the left-hand side is found. Rules for the requirements will be formulated as traits extending *ResourceMonitor*. These traits will later conveniently be combined into one monitor.

3.2.3 Resource Ordering

Our first set of rules[4] specify how *before*(r_1, r_2) events are captured and turned into facts of the form *Before*(r_1, r_2):

```
trait Ordering extends ResourceMonitor {
  "r1" --
  before('r1,'r2) |-> Before('r1,'r2)

  "r2" --
  Before('r1,'r2) & Before('r2,'r3) |->
    Before('r1,'r3)
}
```

Rule r_1 states that a *before*(r_1, r_2) event causes a fact *Before*(r_1, r_2) to be generated. Note that events only exist briefly while facts exist until explicitly deleted. Rule r_2 creates facts expressing the transitive nature of the ordering relation. For example, consider the trace:

$$before(a, b).before(b, c)$$

The first event will create the fact *Before*(a, b) via rule r_1. Likewise, the second event will create the fact *Before*(b, c), also via rule r_1. Rule r_2 will as a consequence immediately fire and create the fact *Before*(a, c), building the transitive closure of the ordering relation. We shall refer to the continued execution of rules until none can fire as *inference*. This building of facts from other facts can be a very powerful tool for writing monitors, as will be discussed later.

3.2.4 Requirement***Release***

The next set of rules formalize the ***Release*** requirement:

```
trait Release extends ResourceMonitor {
  "r3" --
  grant('_,'t,'r) & not(Granted('t,'r)) |->
    Granted('t,'r)

  "r4" --
  Granted('t,'r) & release('_,'t,'r) |->
    remove(Granted)

  "r5" --
  end() & Granted('t,'r) |->
     fail("missing release")
}
```

Rule r_3 formalizes that if a *grant*('_,'t,'r) is observed, and no *Granted*('t,'r) fact exists in the fact memory (with same task *'t* and resource *'r*), then a *Granted*('t,'r) fact is inserted in the fact memory to record that the *grant* event occurred. The '_ symbol represents a *"don't care"* pattern, matching, in this case, any time stamp, without binding it to an identifier.

however, differ by allowing a more flexible way of composition called *mixin composition*, an alternative to multiple inheritance. A trait can be thought of as "just" a collection of definitions.

[3] This procedure is a consequence of the fact that in Scala, like in most programming languages with an exception in Lisp, names are not first-class citizens.

[4] Due to the two-column format each rule is typically defined on several lines, with the rule name on the first line.

Rule r_4 expresses that if a *Granted*('t,'r) fact exists in the fact memory, and a *release* event occurs with matching arguments, then the *Granted* fact is removed. The function *remove* takes as argument a symbol, in this case the symbol 'Granted (which the identifier *Granted* stands for - introduced in a **val**-declaration in the trait *ResourceMonitor*). This symbol represents a reference to the exact fact that matches the condition and makes the rule trigger. This approach works if there is only, as in this case, one *Granted* fact mentioned on the left-hand side. In the more general case, it is possible to label facts and use these labels to remove facts, as in the following rule where the *Granted*('t,'r) fact is labelled *'l*:

```
"r4" --
Granted('t,'r) == 'l & release('_,'t,'r) |->
  remove('l)
```

Finally, rule r_5 expresses that when an *end*() event is observed at the end of the log, and a resource still remains granted (that is: not yet released), it is a failure.

3.2.5 Requirements ***NoRelease*** and ***NoGrant***

The next two traits represent the requirements ***NoRelease*** and ***NoGrant***, and should be self-explanatory after the above explanations.

```
trait NoRelease extends ResourceMonitor {
  "r6" --
  release('_,'t,'r) & not(Granted('t,'r)) |->
    fail("bad release")
}

trait NoGrant extends ResourceMonitor {
  "r7" --
  Granted('_,'r) & grant('_,'_,'r) |->
    fail("bad double grant")

  "r8" --
  Before('r1,'r2) &
  Granted('t,'r2) & grant('_,'t,'r1) |->
    fail("bad grant order")
}
```

3.2.6 Requirement ***Deny***

The ***Deny*** property is defined by the following trait:

```
trait Deny extends ResourceMonitor {
  val Counter = fact

  "r9" --
  Granted('_,'r) & request('s,'t,'r) |->
    Deny('s,'t,'r)

  "r10" --
  Before('r1,'r2) &
  Granted('t,'r2) & request('s,'t,'r1) |->
    Deny('s,'t,'r1)

  "r11" --
  Deny('s1,'t,'r) & Counter('n) &
  deny('s2,'t,'r) |-> {
    if (('s2 - 's1) > 10000 || 'n >= 3) fail()
    remove(Deny)
    remove(Counter)
    insert(Counter('n + 1))
  }

  "r12" --
  end() & Deny('s,'t,'r) |-> fail("missing deny")

  addFact(Counter)(0)
}
```

The rules r_9 and r_{10} generate a *Deny* fact upon a request if the requested resource is already granted, or is ordered before a resource that is already granted. The time stamp *'s* indicates the time the resource was requested. The *Deny* fact represents the obligation that the resource arbiter eventually must inform the requesting task, that it has been denied the resource. Rule r_{12} expresses that the *Deny* fact represents such an obligation, that has to be fulfilled (by the triggering of rule r_{11}) before the end of the log is detected.

Rule r_{11} describes what happens when a deny event occurs at time *'t*2. We assume a *Counter* fact in the fact memory, which carries an integer, counting the number of *deny* events occurring so far. The fact memory is initialized to contain *Counter*(0). The right-hand side of rule r_{11} is a code block, a SCALA statement. First a failure is reported in case the denial comes too late, or if there are more than 3 denials observed in total. Note that symbols standing for bound variables can in limited contexts be referred to as variables, in this case integers. This is achieved by defining *implicit functions*, that by the SCALA compiler gets applied to lift values from one type to another. The concept of *implicit functions* will be explored in later sections to explain other DSL constructs. Alternatively, a variable *'x* assumed to be bound to a value of type T can be accessed as get[T]('x). Abbreviated dot-notation forms furthermore exist for selected primitive types, such as for example'x.i, which is equivalent to get[Int]('x). Next, the *Deny* and *Counter* facts are removed, and a new updated *Counter*('n + 1) fact is inserted[5].

LOGFIRE offers functions (*ensure* and *update*) to make specification writing slightly more convenient. For example rule r_{11} can instead be written as:

```
"r11" --
```

[5] A right-hand side of the form ... |->insert(f) can be written as ... |->f, as we have seen for example in rule r_1.

```
Deny('s1,'t,'r) & Counter('n) &
deny('s2,'t,'r) |-> {
  ensure(('s2 - 's1) <= 10000 & 'n < 3)
  remove(Deny)
  update(Counter('n + 1))
}
```

The call update(Counter('n +1)) is equivalent to the *remove-insert* statement pair in the first version of the rule. The use of a fact to count is shown to illustrate how facts can be instantiated with arbitrary expressions as arguments. In this case, however, it may be easier to introduce a trait-local counter variable as shown in the following version (rules r_9, r_{10} and r_{12} stay unchanged):

```
trait Deny extends ResourceMonitor {
  var counter: Int = 0
  ...

  "r11" --
  Deny('s1, 't, 'r) & deny('s2, 't, 'r) |-> {
    ensure(('s2 - 's1) <= 10000 & counter < 3)
    remove(Deny)
    counter += 1
  }

  ...
}
```

This example illustrates how DSL and regular Scala code can be elegantly merged.

3.3 Finalization and Application of Monitor

3.3.1 Composing Rules into a Monitor

The traits above can now be combined using what is referred to as Scala's *mixin-class composition*, which is a syntactic approach to multiple inheritance, where in this case the class *ResourceManagement* is defined to combine (inherit from) all the rules from the individual rule traits shown above:

```
class ResourceManagement
  extends Ordering
  with Release
  with NoRelease
  with NoGrant
  with Deny
```

3.3.2 Applying the Monitor

We can now finally create an instance of this monitor and apply it to a sequence of events. In the following object the events are fed explicitly. In general one would likely read the events from a log file; or generate events from a running program, using some form of program instrumentation, such as aspect-oriented programming.

```
object ApplyResourceManagement {
  def main(args: Array[String]) {
    val m = new ResourceManagement

    m.addEvent('before)('wheel1,'wheel2)             // 1
    m.addEvent('before)('wheel2,'wheel3)             // 2
    m.addEvent('request)(1012,"drive",'wheel3)  // 3
    m.addEvent('grant)(1402,"drive",'wheel3)    // 4
    m.addEvent('request)(3451,"drive",'wheel1)  // 5
    m.addEvent('grant)(4435,"drive",'wheel1)    // 6
    m.addEvent('release)(9002,"drive",'wheel3)  // 7
    m.addEvent('release)(9409,"drive",'wheel1)  // 8
    m.addEvent('end)()                                // 9
  }
}
```

The events are numbered for reference. The trace above violates the property ***NoGrant*** by granting wheel 1 to the drive task (event 6), although wheel 1 is ordered before the already granted wheel 3. This resource request should instead have been denied, which results in the ***Deny*** property to be violated (detected at the end of the log). Error reports are stored internally in the monitor in an instance of a class *MontiorResult*, which can be obtained by a call of m.getMonitorResult in case monitor results need to processed automatically, for example as part of a testing framework. Error messages are also printed on standard out. An error message consists of an error trace reporting what events caused rules to fire that were relevant for the violation of the property. For example, the error trace generated due to the above mentioned *grant* event number 6 has the following format:

```
*** ERROR bad grant order

[1] 'before('wheel1,'wheel2) -->
      'Before('wheel1,'wheel2)
    rule: "r1" --
    'before('r1,'r2) |-> 'Before('r1,'r2)

[2] 'before('wheel2,'wheel3) -->
      'Before('wheel2,'wheel3)
    rule: "r1" --
    'before('r1,'r2) |-> 'Before('r1,'r2)

[2] 'before('wheel2,'wheel3) -->
      'Before('wheel1,'wheel3)
    rule: "r2" --
    'Before('r1,'r2) & 'Before('r2,'r3) |->
      'Before('r1,'r3)

[4] 'grant(2000,"drive",'wheel3) -->
      'Granted("drive",'wheel3)
    rule: "r3" --
    'grant('_,'t,'r) & not('Granted('t,'r)) |->
      'Granted('t,'r)
```

```
[6] 'grant(4000,"drive",'wheel1) -->
      'Fail("ERROR bad grant order")
    rule: "r8" -- 'Before('r1,'r2) &
    'Granted('t,'r2) & 'grant('_,'t,'r1) |-> {
      ...
    }
```

The trace shows that events 1, 2, 4 and 6 contributed to the error. The first line in the error trace:

```
[1] 'before('wheel1,'wheel2) -->
      'Before('wheel1,'wheel2)
    rule: "r1" --
    'before('r1,'r2) |-> 'Before('r1,'r2)
```

states that event number 1, *'before('wheel1,'wheel2)*, caused the fact *'Before('wheel1,'wheel2)* to be generated, by executing rule r_1. The error trace outlines the scenario where (events 1 and 2): $wheel_1$ is ordered before $wheel_3$ via $wheel_2$ due to transitive closure of the *Before* relation, and where $wheel_3$ is first granted to the drive task (event 4), and then $wheel_1$ is granted (event 6), which violates the resource ordering.

3.4 Additional Features

3.4.1 Events and Facts as Maps

In the rules above events and facts have been referred to in what is called positional format, where the argument patterns to event and fact conditions are provided in a manner similar to the way arguments are provided in traditional function calls. For example, the ***Release*** property rule r_3 refers to event *grant('_,'t,'r)*, representing a condition which when matching a *grant* fact in fact memory, will cause the second argument to be bound to 't and the third argument to be bound to 'r.

In general, however, the argument to an event or fact is a map from keys to values. The above condition is in principle short for *grant('_2 → 't, '_3 → 'r)*, meaning that this condition will match a *grant* event carrying a map, that maps the symbol *'_2* to a value, which is then bound to *'t*, and that maps the symbol *'_3* to a value, which is bound to *'r*. This map-oriented style can be used directly in rules and traces. As an example, the following is an alternative formulation of the ***Release*** property using map style for events and positional style for generated facts:

```
trait Release extends ResourceMonitor {
  "r3" --
  grant('task -> 't,'resource -> 'r) &
    not(Granted('t,'r)) |-> Granted('t,'r)

  "r4" --
  Granted('t,'r) &
  release ('task -> 't, 'resource -> 'r) |->
    remove(Granted)

  "r5" --
  end() & Granted('t,'r) |-> fail("missing release")
}
```

Here *grant* and *release* events are assumed to carry maps with at least two fields: *'task* and *'resource*. The positional style is convenient if events carry few arguments. In practice, however, it turns out to be common to encounter events with many arguments, in which case the map-based notation is convenient: conditions can just refer to the fields of relevance to the particular rule, instead of mentioning all the arguments. Assume that all rules have been modified to process map-based events with these fields. The following object illustrates how such events can be submitted to the monitor in map-format. In this case a *grant* event carries a map defining three fields (*'time*, *'task*, and *'resource*):

```
object ApplyResourceManagement {
  def main(args: Array[String]) {
    val m = new ResourceManagement
     ...
    m.addMapEvent('grant)(
      'time -> 2000,
      'task -> "drive",
      'resource -> 'wheel3)
     ...
  }
}
```

3.4.2 Submitting Facts to a Monitor

We have above seen how events are submitted to a monitor. It is also possible to submit (position-based as well as map-based) facts to a monitor, which will remain in fact memory until removed again explicitly on the right-hand side of a rule. For example, in the monitor above we submitted the events *'before('wheel1,'wheel2)* and then *'before('wheel2,'wheel3)*, which caused rule r_1 to generate the two facts *'Before('wheel1,'wheel2)* and *'Before('wheel2,'wheel3)*, whereafter rule r_2 would generate fact *'Before('wheel1,'wheel3)*. We could instead avoid the need for rule r_1 and submit the two facts *'Before('wheel1,'wheel2)* and *'Before('wheel2,'wheel3)* directly. In this case the submission of events 1 and 2 in the *ApplyResourceManagement* monitor above (page 8) would instead become submission of facts:

```
m.addFact('Before)('wheel1, 'wheel2) // 1
m.addFact('Before)('wheel2, 'wheel3) // 2
```

3.4.3 Considering the Fact Memory as a Database

LogFire offers a method for writing the fact memory to persistent store and a method for retrieving it at a later point in time. Both methods take a file name and a predicate on facts as arguments, and writes to, respectively reads from, this file those facts that satisfy the predicate. Writing facts to persistent store can be useful when LogFire is used in sessions distributed over time, either due to log sizes, log availability, or time constraints on the user of LogFire. In addition there are methods for deleting all facts satisfying a predicate, and for extracting all facts satisfying a predicate into a set.

4 Defining and Using Specification Patterns

Rule-based programming, as we have seen demonstrated above, is an expressive and moderately convenient notation for writing monitoring properties. Although specifications are longer than traditional temporal logic specifications, they are simple to construct due to their straight forward and intuitive semantics. However, ideally the more succinct a specification is, the better. In this section we shall illustrate how to define and apply specification patterns, using Scala's general purpose function definitions and support for defining DSLs.

An often cited resource on specification patterns is [33], which illustrates how various typically occurring temporal specification patterns over propositional names can be translated into various forms of temporal logic. In contrast, the patterns presented in this section are over *parameterized* events and are translated into rules. The fact that the patterns are formed over parameterized events makes our approach strictly more expressive. We shall first illustrate the definition of two temporal patterns from [33], and then a pattern inspired by regular expressions as used in TraceMatches [14] and Mop [28].

4.1 Temporal Logic

Two patterns mentioned in [33] are the global response pattern: *'always P implies eventually Q'*; and the global precedence pattern: *'no P before Q'*, which can also be read as: *'always P implies Q in the past'*. In Ltl these patterns can be written as respectively $\Box(P \rightarrow \Diamond Q)$ and $!P \; \mathcal{W} \; Q$ (not P weak-until Q). The following monitor extends a class *TemporalLogic*, where we have defined these patterns. The monitor expresses the properties ***Release*** and ***NoRelease***.

```
class ReleaseProperties extends TemporalLogic {
  val grant, release = event

  "Release" --- grant('_,'t,'r) --> release('_,'t,'r)
  "NoRelease" --- release('_,'t,'r) ==> grant('_,'t,'r)
}
```

The first property states that when a *grant('_,'t,'r))* event is observed then eventually (`-->`) a *release('_,'t,'r)* event must occur, with the same task 't and resource 'r as parameters (ignoring the first time stamp parameter). Similarly for the second property stating that a *release('_,'t,'r)* event must be preceded (`==>`) by an earlier *grant('_,'t,'r)* event. The *TemporalLogic* trait is defined as follows.

```
trait TemporalLogic extends LogicUtil {
  private def response(name: String)
                          (pc1: PC, pc2: PC) {
    val unsafe = newSymbol('unsafe)
    val args = pc1.getVariables.reverse
    newRuleId() -- pc1 |-> unsafe(args: _*)
    newRuleId() -- unsafe(args: _*) & pc2 |->
      remove(unsafe)
    newRuleId() -- unsafe(args: _*) & 'end() |->
       fail (name + " failed")
  }

  private def precedence(name: String)
                            (pc1: PC, pc2: PC) {
    val safe = newSymbol('safe)
    val args = pc2.getVariables.reverse
    newRuleId() -- pc2 |-> safe(args: _*)
    newRuleId() -- pc1 & not(safe(args: _*)) |->
       fail (name + " failed")
  }

  implicit def R(name: String) = new {
    def ---(pc1: PC) = new {
      def -->(pc2: PC) =
        response(name)(pc1, pc2)
      def ==>(pc2: PC) =
        precedence(name)(pc1, pc2)
    }
  }
}
```

The functions *response* and *precedence* define the semantics of respectively the response and the precedence pattern. Each takes as arguments the name of the property, and two so-called *Positive Conditions* (*PC*) representing the two events that should be related, and from these three arguments generate the rules modeling the respective patterns, using the rule syntax we have already seen before. For example, the *response* function first generates a unique new fact name, named *unsafe* (with the function *newSymbol*), extracts the arguments of the first event *pc1*, and then generates three rules. The first rule generates an *unsafe(args)* fact upon observation of *pc1*. The second rule removes this fact upon observation of *pc2*, and the third rule reports a failure upon detection of this fact at the end of the log. Likewise, the *precedence* function generates two rules, one generating a fact *safe(args)* upon observation of a *pc2* event, and one rule checking for the presence of this fact upon observation of a *pc1* event.

The functions *response* and *precedence* can be applied directly to define properties. However, we shall illustrate how Scala's *implicit functions* can be used to define the notation used in the trait *ReleaseProperties* above. An *implicit function* (defined with the keyword **implicit**) $f : A \Rightarrow B$ will be automatically applied by the compiler on any value $a : A$ occurring in a context where a value of type B is expected. Specifically in this case, the function named R (standing for *Rule*) is defined as implicit, meaning that whenever a string *name* occurs followed by `---`, this function will be applied to the string, returning an un-named object defining the `---` method, which itself returns a new object defining the two methods `-->` and `==>`. The definition of the R function corresponds to the following BNF grammar:

⟨*formula*⟩ ::= ⟨*String*⟩ `---` ⟨*PC*⟩ (`-->` ⟨*PC*⟩ | `==>` ⟨*PC*⟩)

Scala allows methods to be called without dot notation and argument parentheses. The following Scala code from the *ReleaseProperties* monitor above:

```
”Release” --- grant('_,'t,'r) --> release('_,'t,'r)
```

has the same meaning as:

```
”Release”.---(grant('_,'t,'r)).-->(release('_,'t,'r))
```

which by the compiler then gets corrected by the insertion of a call of the implicit R function on the rule name, and similarly an implicit function C (standing for *Condition*) to lift event names:

```
R(”Release”).---(C(grant)('_,'t,'r)).-->(
  C(release)('_,'t,'r)
)
```

This statement in turn results in the following rules to be generated with a semantics that at this point should be self-explanatory:

```
”tl1” --
'grant('_,'t,'r) |-> 'unsafe_1('t,'r)

”tl2” --
'unsafe_1('t,'r) & 'release('_,'t,'r) |->
  remove('unsafe_1)

”tl3” --
'unsafe_1('t,'r) & 'end() |-> fail(”Release”)
```

Similarly, the formula:

```
”NoRelease” --- release('_, 't, 'r) ==>
  grant('_, 't, 'r)
```

generates the two rules:

```
”tl4” --
'grant('_,'t,'r) |-> 'safe_2('t,'r)

”tl5” --
'release('_,'t,'r) & not('safe_2('t,'r)) |->
  fail(”NoRelease”)
```

4.2 Path Expressions

Systems such as TraceMatches [14] and Mop [28] support specifications in the form of regular expressions. We have implemented a limited but still useful subset of regular expressions, not including disjunction and repetition, as a pattern in 50 lines of Scala code in the class *PatternExpressions* (not shown here). We have chosen this simple format since it is easy to implement, but also since it is deemed practically sufficient[6]. In a pattern expression one can provide a sequence of events and/or negation of events. A match on such a sequence anywhere in the trace will trigger a user-provided code segment to get executed. As an example, consider the following formulation of the ***NoGrant*** requirement (a resource should not be granted to a task if it is already granted):

```
class NoGrant extends PathExpressions {
  val grant, release = event

  when(”double grant”)(
    grant('_, 't1, 'r),
    no(release('_, 't1, 'r)),
    grant('_, 't2, 'r)
  ) {
    fail(”resource acquired twice”)
  }
}
```

The property states that when a $grant('\_,'t1,'r)$ is observed, and then subsequently another $grant('\_,'t2,'r)$ of the same resource, without a $release('\_,'t1,'r)$ in between, then the code provided as the last argument is executed, in this case just the reporting of a failure. The above call of the function *when* will generate the following rules, assuming that *Initial* is the initial state (the names of generated facts are renamed to ease reading):

```
”pe1” --
'grant('_,'t1,'r) & 'Initial() |-> 'Granted('t1,'r)

”pe2” --
'release('_,'t1,'r) & 'Granted('t1,'r) |->
  'Released('t1,'r)
```

[6] Disjunction and repetition are often used in Mop only because of the special semantics of Mop regular expressions, where left out events in the regular expression mean that such events are not allowed to occur in the trace at those points. Hence, if they can occur in the trace, they have to be mentioned in the regular expression at the appropriate positions. In contrast, our patterns are *relaxed* in the sense that left out events can occur in the trace but are ignored during the conformance check, and therefore in many situations we can avoid the need for disjunction and repetition.

```
”pe3” −−
'grant('_,'t2, 'r) & 'Granted('t1,'r) &
not('Released('t1,'r)) |−> {
  remove('Granted)
   insert ('Matched('t1,'r,'t2))
}

”pe4” −−
'grant('_3,'t2,'r) & 'Granted('t1,'r) &
'Released('t1,'r) |−> {
  remove('Granted)
  remove('Released)
}

”pe5” −−
'Matched('t1,'r,'t2) |−> {
   fail (”resource acquired twice”)
}
```

The translation may seem somewhat surprising and can in this case be optimized. For example, in rule pe_2 we could abort the tracking immediately when a release event is observed, instead of creating a *'Released('t1,' r)* fact in addition to the *'Granted('t1,' r)* fact already present. However, the translation above is generalized to handle past time properties.

5 The Rete Algorithm

This section contains a very condensed presentation of the Rete algorithm that was developed by Charles L. Forgy [37] in the 1970ties. For a full account of the algorithm, which space does not permit here, the reader is referred to [30]. The algorithm has acquired “a reputation for extreme difficulty” [59]. As part of his Ph.D. thesis [30], Robert B. Doorenbos outlines in the mid 1990ties the algorithm in a very thorough and precise manner on 52 pages; and furthermore augments the algorithm with two optimizations, explained on an additional 45 pages. The core of the algorithm is, however, rather simple. Our work is based on Doorenbos's presentation, including his optimizations (referred to as *left-* and *right-unlinking* in [30]).

5.1 The Rete *Algorithm*

From an abstract point of view, the Rete algorithm works on a program state, which is a set of facts, or *working memory elements* (*wme*'s), as they are called in Doorenbos's thesis [30]. A fact in Doorenbos's thesis is a triple: $(id, attr, value)$ reflecting the intuition that an object *id* has an attribute *attr* which has the value *value*. Doorenbos writes such triples as (*id ^attr value*). For example, the fact that a task '*t*' has been granted a resource '*r*' is modeled as: (*t ^Granted r*). We shall in this paper write this as $Granted(t, r)$. We shall generalize this to allow any number of arguments, and also allow general maps as facts, as will be illustrated. A rule program consists of a set of rules of the form: $lhs \Rightarrow rhs$, where a left-hand side is a sequence of conditions composed by conjunction, and the right-hand side is an action (or a composition of such). Conditions can be tests of the presence or absence of facts satisfying certain constraints on the parameters, and actions can be addition or removal of facts to or from the fact memory. As an example, consider the earlier presented rules r_8 (Section 3.2.5) and r_{10} (Section 3.2.6), repeated here for ease of readability:

```
”r8” −−
Before('r1,'r2) &
Granted('t,'r2) & grant('_,'t,'r1) |−>
   fail (”bad grant order”)

”r10” −−
Before('r1,'r2) &
Granted('t,'r2) & request('s,'t,'r1) |−>
  Deny('s,'t,'r1)
```

Recall that events are written with small initial letters, while facts are written with capital initial letters. However, the original Rete algorithm only operates with facts. Consider for now that events are just facts. Their special treatment will be discussed in Section 6. A naive inefficient implementation, given a change in the fact memory, might check each rule in the rule program, and re-evaluate the entire left-hand side, marking the action for execution if the left-hand side evaluates to true. Whether to execute all such marked actions or only selected ones is referred to as *conflict resolution*. Note that looping is possible in case deleted or added facts trigger new rules.

The Rete algorithm optimizes this problem by avoiding this repeated re-evaluation of left-hand sides. The rules r_8 and r_{10} above are modeled as the network shown in Figure 1. The name *Rete* means *network* and reflects the way a set of rules is stored by the algorithm: as a network. This network consists of 4 kinds of nodes:

- *alpha memories*: the white rectangular nodes. Each of these nodes represents a condition (assume for now that conditions are positive, no negations). A condition is the assertion that a certain fact is present in the fact memory. When a fact is added to the fact memory, it is effectively added to the alpha memory corresponding to that condition.
- *beta memories*: the grey rectangular nodes. Each of these represents a prefix of the conditions in a rule. For example in the figure, $beta_3$ will contain a so-called *token* when condition $Before(r_1, r_2)$ and condition $Granted(t, r_2)$ have become true for some resources R_1 and R_2, and task T. A token is simply the

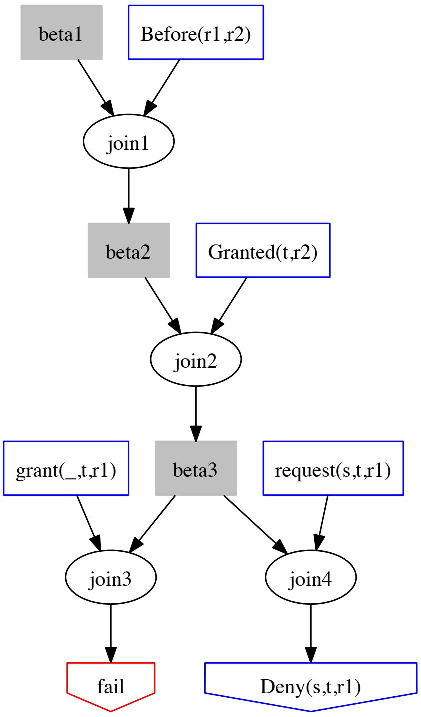


Fig. 1. Simple network for the rules r_8 and r_{10}.

list of facts that matched the previous conditions, for example in this case the list:

$$\langle \mathit{Before}(R_1, R_2), \mathit{Granted}(T, R_2)\rangle$$

Note that a beta memory can contain multiple such if there have been several matches.

- *join nodes*: the round nodes. Each such is connected to an alpha memory and a beta memory, and correspondingly gets activated in two cases:
  1. when the connected *alpha memory* receives a new fact. A common terminology is to say that the join node is *right-activated*, since alpha memories usually are drawn to the right of a join node (this is just a layout issue, and Figure 1 in fact violates this rule, in order to obtain a better layout). In this case the connected beta memory is searched for all its tokens, and for each is it tested if the bindings of that token match that of the fact. If so, a new extended token is created from the old token and the fact, and sent to the child beta memory.
  2. when the connected *beta memory* receives a new token. A common terminology is to say that the join node is *left-activated*. In this case the connected alpha memory is searched for all its facts, and for each is it tested if the bindings of that fact match that of the token. If so, a new extended token is created from the old token and the fact, and sent to the child beta memory.
- *action nodes*: the downwards arrow shaped nodes. Such a node represents an action to get executed if a token arrives.

When a new fact is added, it is put in the matching alpha memory (or rather alpha memories), using a simple indexing approach. This indexing is based purely on the constants occurring in the conditions of rules. For example, assume that a rule contains the condition $\mathit{request}(s, t, \text{“}\mathit{motor}2\text{”})$, where the resource argument is a string literal, namely the name of a particular resource. A fact will only be inserted in the alpha memory represented by this node, and hence possibly causing rule evaluation, if its attribute is *request* and if the third argument is exactly “$\mathit{motor}2$”.

5.2 Rete's Five Optimizations

The Rete algorithm specifically optimizes five situations. These are described in the following, illustrated with references to Figure 1.

5.2.1 Rule Relevance

The Rete network is organized such that when a fact is added, it is placed in the relevant alpha memory. The alpha memory is only linked to those join nodes, and hence rules, for which it is relevant. For example, when a *grant* fact is added, only join node join_3 is evaluated in Figure 1. That is, only rule r_8 is evaluated, and not rule r_{10} (amongst those two).

5.2.2 Minimal Condition Evaluation

The Rete algorithm attempts to (re)evaluate a condition “as little as possible”. Consider rules r_8 and r_{10}, and suppose that the fact $\mathit{Before}(\text{“}\mathit{antenna}\text{”}, \text{“}\mathit{wheels}\text{”})$ is added to the alpha memory. In the network this means that the join node join_1 executes and puts the token $\langle \mathit{Before}(\text{“}\mathit{antenna}\text{”}, \text{“}\mathit{wheels}\text{”})\rangle$ in beta_2. The network will in this case in subsequent steps not keep re-evaluating the condition $\mathit{Before}(r_1, r_2)$ in rules r_8 and r_{10}, unless the $\mathit{Before}(\text{“}\mathit{antenna}\text{”}, \text{“}\mathit{wheels}\text{”})$ fact is removed or other *Before* facts are added.

5.2.3 Prefix Sharing

The Rete algorithm optimizes situations where two or more rules have a common condition prefix. Consider again the rules r_8 and r_{10} above. These two rules share the prefix:

$$Before(r_1, r_2), Granted(t, r_2)$$

The Rete network is in this case laid out to ensure that this prefix is shared between the two rules for evaluation purposes, they share the same sub-graph, and hence is only evaluated once whenever a relevant change occurs.

5.2.4 Prefix Readiness

This is the optimization technique referred to as *right-unlinking* in [30] - an addition to the original RETE algorithm described in [37]. Consider join node $join_2$. In the original algorithm, when a *Granted* fact arrives, a test will be performed on whether $beta_2$ is empty (and if it is, no further work is done). Suppose now that $beta_2$ is empty and contains no tokens. That is, there are no *Before* facts. In this case Doorenbos's optimization avoids the emptyness test by *right-unlinked* $join_2$, meaning that the pointer from the $Granted(t, r_2)$ alpha memory to $join_2$ is removed. This has as consequence that when a *Granted* fact arrives, the algorithm will not even reach and execute $join_2$.

This may not seem like a big gain, but the situation can provide an optimization in case an alpha memory is connected to many join nodes, and many of these have empty beta memories. This occurs in the situation where a condition K occurs in many rules with different prefixes as follows:

$$\begin{array}{lllll} r_1 & : & c^1_1, \ldots, c^1_{n_1}, & K & \Rightarrow & a_1 \\ r_2 & : & c^2_1, \ldots, c^2_{n_2}, & K & \Rightarrow & a_2 \\ \ldots & & & & & \\ r_m & : & c^m_1, \ldots, c^m_{n_m}, & K & \Rightarrow & a_m \end{array}$$

Assume that only in some of these rules are the conditions before K, the prefix, true; then only for these will the corresponding join nodes be executed (right-activated) after this optimization.

5.2.5 Suffix Readiness

This is the second optimization technique contributed in [30], also referred to as *left-unlinking*, and the symmetric case of right-unlinking described above. Consider again join node $join_2$, and suppose that there are no facts in the $Granted(t_1, r_1)$ alpha memory. In this case $join_2$ is *left-unlinked*, meaning that the pointer from $beta_2$ to $join_2$ is removed. It optimizes the situation where a certain condition prefix is repeated in many rules, but in each rule followed by different conditions K_i:

$$\begin{array}{llllll} r_1 & : & c_1, \ldots, c_n, & K_1 & \Rightarrow & a_1 \\ r_2 & : & c_1, \ldots, c_n, & K_2 & \Rightarrow & a_2 \\ \ldots & & & & & \\ r_m & : & c_1, \ldots, c_n, & K_m & \Rightarrow & a_m \end{array}$$

6 Evaluation and Modification of RETE for RV

This section contains a brief discussion of the applicability of the RETE algorithm for runtime verification, and then outlines a number of modifications and optimizations we have made to the algorithm to make it more suitable.

6.1 Evaluation

6.1.1 Events versus Facts

The first observation is that the original RETE algorithm does not distinguish between events and facts. There are only facts. A fact remains in the fact memory until explicitly removed by an action on the right-hand side of a rule. Second, RETE is designed for *logical inference*. That is, when adding (or removing) a fact, the rule engine executes until a fixpoint is reached, and no more inferences can be made[7]. Consider for example the previously introduced rules r_3 (Section 3.2.4) and r_7 (Section 3.2.5):

```
"r3" --
grant('_,'t,'r) & not(Granted('t,'r)) |->
  Granted('t,'r)

"r7" --
Granted('_,'r) & grant('_,'_,'r) |->
  fail ("bad double grant")
```

Consider an initially empty fact memory, and suppose that we add the fact (event) *grant*(344, 42, "*antenna*"). This will cause rule r_3 to fire, which will add the fact *Granted*(42, "*antenna*") to the fact memory. This in turn will trigger rule r_7 since now the fact memory contains *grant*(344, 42, "*antenna*") and *Granted*(42, "*antenna*"). As a result, an error is issued although this cannot be the intention. Rather, the desired behavior should preferably be that after submitting the *grant*(344, 42, "*antenna*") event, rule r_3 would fire, and then no more rules would fire until the next event is submitted.

This form of propagation until a fixpoint is reached can be useful for analyzing logs, as suggested in [53]. For example we can imagine that some higher order abstraction facts can be generated if a number of other sub-facts are generated. This idea is being explored in an application of LOGFIRE to the analysis of telemetry from the Curiosity Mars rover, and will be reported elsewhere. However, an event such as *grant*(...) is considered as a short-lived event. It indeed turns out that for RV purposes we do not want such *inference* for *events*.

6.1.2 Rete's Optimizations

RETE's optimizations listed in Section 5.2 include rule relevance, minimal condition evaluation, prefix sharing,

[7] Note that in the original RETE algorithm, when a left-hand side becomes false, previously added facts on the right-hand side do *not* get retracted, as they would in pure logic inference. DROOLS, however, offers this pure logic interpretation of rules in addition to the standard RETE interpretation.

prefix readiness, and suffix readiness. Of these the first two: *rule relevance* (only evaluate rules concerned with an incoming fact) and *minimal condition evaluation* (only evaluate a condition when a fact concerning that condition is added or removed), appear relevant for runtime verification, whereas the last three might seem less so, as will be discussed in the following.

Prefix Sharing (rules sharing a prefix of conditions share evaluation of these in the network), seems an essential part of the RETE algorithm, and yields the most benefit if there are many rules that share conditions. The question is, however, whether rules are strongly connected like this in RV contexts. This would require that facts generated by rules modeling one property would be relevant for rules for other properties. The properties would have to concern the same artifacts for this to be the case. Indeed, we have seen sharing between rules in the previous specifications. The question is whether this will occur in a larger scale for RV properties. It is possible that the RV community has focused on small isolated properties, since this is how traditional temporal properties look like, and that further case studies will reveal need for larger scale sharing.

Concerning *prefix readiness*, it is not clear to what extent this is an important optimization in practice. Consider for example rule r_7 above. Suppose a *grant* event arrives. Then for this optimization to be useful there should be no resources granted at all to any tasks. In addition *grant* should occur in many other rules, where no tokens satisfy the prefix. The question is how likely this is to occur in RV practice. On the other hand, *suffix readiness* seems to be a more important optimization, since it covers the notion of state machines with states, and from each state many transitions $K_1 - K_m$ leading out of the state.

6.1.3 Missing Indexing

Consider rule r_8 from Section 5.1, and its representation in Figure 1; specifically consider join node $join_3$ in that network. Consider the scenario where many resources have been granted that are ordered *after* other resources, that is, many tokens of the form:

$$\langle Before(r_1, r_2), Granted(t, r_2)\rangle$$

in the beta memory $beta_3$. Assume now that an event $grant(344, 42, \text{“}antenna\text{”})$ arrives in the alpha memory for condition $grant(\_, t, r_1)$, and join node $join_3$ is activated, meaning that we search for a token in $beta_3$ where r_1 has the value “*antenna*”. This is an instance of the matching problem. In the RETE algorithm, as presented in [30], this search is performed linearly: all tokens are scanned from left to right searching for matches. Since several tokens could match, all tokens have to be searched. This is a considerable inefficiency in the case where there are numerous tokens. The same symmetrical argument can be made in the other direction: when a token arrives in a beta memory and the join node has to search all facts in the corresponding alpha memory. An indexing mechanism is needed in each direction. In fact, this is closely related to the form of indexing performed in efficient frameworks such as MOP [58].

6.2 Modifications

The considerations above has lead us to implement specifically two modifications required to make the RETE algorithm appropriate for RV: *event processing* and *indexing*. These modifications are described in the following.

6.2.1 Event Processing

As discussed in Section 6.1, events (in this paper referred to with names in all small letters) should be treated differently than “long lasting” facts (in this paper referred to with names with an initial capital letter). Events are short lived. The question is: how short lived? As the *grant*/*Granted* scenario (Subsection 6.1.1) illustrates, it is not enough to delete the event after a fixpoint is reached, and before the next event is received. It has to be deleted even earlier. The algorithm for submitting an event consequently is modified to be as follows. Assume a set of actions to be executed next: the *next action frontier*, initially empty.

Definition 1 (Event cycle resulting from submitting an event e**).**

1. add event e to the fact memory.
2. apply RETE where only left-hand sides are evaluated. Right-hand sides that are reached are only stored for later execution in the *next* action frontier.
3. remove e from fact memory (eager event removal).
4. make *next* action frontier *current*, and create a new empty *next* action frontier.
5. exit if *current* action frontier is empty.
6. while there are still stored right-hand sides waiting for execution in the *current* action frontier, execute stored right-hand sides, and store new right-hand sides reached in the *next* action frontier.
7. goto step 4.

This algorithm is also illustrated by the following pseudo code:

```
var frontier : List[Action] = Nil

def addFact(fact: Fact) {
  // After fact is added: only left−hand sides are
  // evaluated. Reached right−hand sides are stored
  // in 'frontier'.
   ...
}

def removeFact(fact: Fact) {
    // After fact is removed: same procedure as for
    // addFact.
    ...
  }

  def submit(event: Event) {
    addFact(event)
    removeFact(event)
    while (frontier != Nil) {
      val current = frontier; frontier = Nil
      for (action <- current) {
        action match {
          case ADD(fact: Fact) =>
            addFact(fact)
          case REM(fact: Fact) =>
            removeFact(fact)
        }
      }
    }
  }
```

Note that in each step an event is added and then removed. This removal is most efficient if the event is the last condition on the left-hand side, as in rule r_7 (page 14), corresponding to be at the bottom of the Rete network. This is because then the least amount of information should be retracted in the network. However, when writing rules of the form where an event carry data that are referred to in a negative condition, as in rule r_3 (page 14), the event has to occur *before* the negated condition. This leads to a less optimal solution; we do not currently have a better solution.

6.2.2 Indexing

Matching in Original Algorithm

To motivate our optimized indexing solution, we shall first briefly outline part of the matching process carried out by the original algorithm. However, in contrast to the original algorithm where a fact is a triple, we shall generalize this to let a fact be a mapping from field identifiers (represented by Scala's type *Symbol* that contains quoted identifiers, such as $'x$, $'\_1$, etc) to values (represented by Scala's type *Any* that corresponds to Java's type *Object*):

```
  type Field = Symbol
  type Value = Any
  type Fact = Map[Field,Value]
```

In the real implementation a token (the elements of beta memories) is built by augmenting a previous token with a new fact. This is done by representing a token as a pair consisting of a pointer to the previous token as the first argument, and then the fact as the second argument. However, for presentation purposes we here represent a token as a list facts:

```
  type Token = List[Fact]
```

Each alpha and beta memory contains a collection of respectively facts and tokens:

```
class AlphaMemory {
  var items : Set[Fact]
   ...
}

class BetaMemory {
  var items : List [Token]
   ...
}
```

When a data element arrives in one of the memories, the connected join node searches the other memory sequentially in order to find matches. For example, consider now the rule r_8 (page 12), and its representation in Figure 1. Consider specifically join node $join_3$, which corresponds to the $grant(\_, t, r_1)$ condition. Consider that $beta_3.items$ contains three tokens: $\langle T_1, T_2, T_3\rangle$, where:

$$\begin{array}{lcl} T_1 & = & \langle Before(a_1,a_2), Granted(A,a_2)\rangle \\ T_2 & = & \langle Before(b_1,b_2), Granted(B,b_2)\rangle \\ T_3 & = & \langle Before(c_1,c_2), Granted(C,c_2)\rangle \end{array}$$

That is, for example token T_2 represents the fact that resource b_1 must be granted before resource b_2 plus the fact that b_2 has already been granted to task B. According to rule r_8 it is therefore illegal to grant resource b_1 to task B in this situation. Consider now that task B, however, is granted resource b_1 at time 1206, causing the following fact to be added to the alpha memory: $grant(1206, B, b_1)$. We now need to try to determine whether $beta_3$ contains a matching token, and if it does, it indicates an error situation. Indeed token T_2 is a match since:

- the 2^{nd} field of $grant(1206, B, b1)$, namely B, is equal to the 1^{st} field of the 1^{st} fact $Granted(B, b_2)$ from the right (corresponding to searching from bottom up in the network) in the T_2 token.
- the 3rd field of $grant(1206, B, b_1)$, namely b_1, is equal to the 1'st field of the 2nd fact $Before(b_1, b_2)$ from the right in the T_2 token.

To perform this test, the original algorithm stores in each join node, in addition to a pointer to the beta memory and the alpha memory, a list of *tests* to perform when either a token arrives in the beta memory, or a fact arrives in the alpha memory. A test is represented by the positions to be compared:

```
class JoinNode {
  val alphaMemory: AlphaMemory
  val betaMemory: BetaMemory

  val tests: List[(Field, (Int,Field)]
  def matches(token: Token, fact: Fact): Boolean = {
    tests forall {
      case (field1 ,(number,field2)) =>
        fact( field1 ) = token.ithFact(number)(field2)
    }
  }

  def leftActivation (token: Token) {
    val facts =
      alphaMemory.items filter matches(token,_)
    // for each such fact:
    //   form new token from old token
    ...
  }

  def rightActivation( fact : Fact) {
    val tokens =
      betaMemory.items filter matches(_,fact)
    // for each such token:
    //   form new token from fact
    ...
  }
}
```

Each test $(\mathit{field}_1, (\mathit{number}, \mathit{field}_2)$ in this list (the value *tests*) specifies a field_1 in the alpha memory fact to match against a field_2 in the beta memory token, identified by relative condition *number*, counted from the right and starting from 0. For example, for join node join_3 this variable will contain the list:

$$\mathit{tests} = \langle({}'\_2, (0,{}'\_1)), ({}'\_3, (1,{}'\_1))\rangle$$

Testing the $\mathit{grant}(1206, B, b_1)$ event against each of the tokens T_1, T_2 and T_3, we end up with one match: T_2. The join node produces a new token which is T_2 extended with $\mathit{grant}(1206, B, b_1)$:

$$\langle Before(b_1, b_2), \mathit{Granted}(B, b_2), \mathit{grant}(1206, B, b_1)\rangle$$

which represents an error situation, and which is transmitted to the fail node.

The *matches* function determines whether a token and a fact match with respect to the *tests*. The function *leftActivation* selects all facts from the alpha memory matching a token that has arrived in the beta memory. Likewise, the function *rightActivation* selects all tokens from the beta memory matching a fact that has arrived in the alpha memory.

Introducing Indexing

Our alternative approach consists of using an index in beta and alpha nodes to speed up the search for matches. Let us first define the concept of an indexable fact or token. An indexable object offers a function for looking up a vector of fields (those we need to match on), returning a list of values for those fields (the result is optional representing that the object may not be defined for one or more fields):

```
trait Indexable[I] {
  def lookup(indexVector: List[I] ): Option[List[Value]]
  ...
}
```

Facts and tokens must implement this interface:

```
class Fact extends Indexable[Field]
class Token extends Indexable[(Int, Field)]
```

Note that to look up a value in a token we need to know what fact identified with its position (from the right), and the field in that fact. We now define what a d-index (double-index) is. A d-index is essentially a mapping, that maps a sequence of indexes to another mapping, which maps a sequence of values to a set of indexable elements.

```
class DoubleIndex[I, E <: Indexable[I]] {
  var index: Map[List[I], Map[List[Value], Set[E]]]

  def lookup(indexVector: List[I] ,
             valueVector: List [Value] ): Set[E]
  ...
}
```

Both an alpha memory and a beta memory contains a d-index, which is updated each time a new indexable element is added:

```
class AlphaMemory {
  val items : DoubleIndex[Field, Fact]()
  ...
}

class BetaMemory {
  val items : DoubleIndex[(Int, Field ), Token]
  ...
}
```

For our example, the beta memory for join_3 looks as follows:

```
index =
  [
    $\langle(0,{}'\_1), (1,{}'\_1)\rangle - >$
      [
        $\langle A, a_1\rangle \to \{\langle Before(a_1, a_2), \mathit{Granted}(A, a_2)\rangle\}$
        $\langle B, b_1\rangle \to \{\langle Before(b_1, b_2), \mathit{Granted}(B, b_2)\rangle\}$
        $\langle C, c_1\rangle \to \{\langle Before(c_1, c_2), \mathit{Granted}(C, c_2)\rangle\}$
      ]
  ]
```

The join node contains two variables stating respectively what the index vector is for its associated alpha and beta memories:

```
class JoinNode {
  var alphaVector: List[Field]
  var betaVector: List[(Int,Field)]
  ...
  def leftActivation(token: Token) {
    val values = token.lookup(betaVector)
    val facts =
      alphaMemory.index.lookup(alphaVector,values)
    // for each such fact:
    //   form new token from old token
      ...
  }

  def rightActivation(fact: Fact) {
    val values = fact.lookup(alphaVector)
    val tokens =
      betaMemory.index.lookup(betaVector,values)
    // for each such token:
    //   form new token from fact
      ...
  }
}
```

The method *leftActivation* is called when a token arrives in the connected beta memory. It uses the *betaVector* to look up the values of interest in that token, those that have to match fields in facts in the alpha memory. Those matching alpha memory facts are then looked up using the *alphaVector* and the values extracted from the token. Similarly for the method *rightActivation*. As an example, when the $grant(1206, B, b_1)$ event arrives in the alpha memory, $alphaVector = \langle '\_2, '\_3 \rangle$ is used to extract the relevant $values = \langle B, b_1 \rangle$, which are then together with $betaVector = \langle (0, '\_1), (1, '\_1) \rangle$ used to look up the (in this case singular element) set of matching tokens: $\{\langle Before(b_1, b_2), Granted(B, b_2) \rangle\}$.

7 Implementation of DSL

Sections 5 and 6 outlined the essential algorithmic properties of the standard Rete algorithm and the modifications and optimizations performed in this work. Concerning the core algorithm, it suffices to add that the implementation in Scala is object-oriented, in contrast to the pseudo code provided in [30]. This section focuses on the implementation of DSL for writing rules, using Scala's convenient features for defining DSLs. The presentation is simplified compared to the actual implementation, but illustrates the main principles. Let us recall what a rule may look like, by considering rule r_4 from Subsection 3.4.1, but using map notation for the fact as well as for the event (to simplify the explanation):

```
trait Release extends Monitor {
  val release = event
  val Granted = fact

  "r4" --
    Granted('task -> 't, 'resource -> 'r) &
    release('task -> 't, 'resource -> 'r) |->
      remove(Granted)
}
```

7.1 Names as First-Class Citizens

Class *Monitor* defines the functions *event*, *fact*, '−−', '&', and '|−>', and some additional implicit functions that make this work. When called as above these functions will produce the above rule in an internal format, as an object of a class *Rule*, to be explained below, which is then passed as argument to a method *addRule(rule: Rule)* in the Rete module. This method then creates a Rete network as described in sections 5 and 6.

The methods *event* and *fact* use Java's reflection[8] to bind the symbols (quoted names) 'release and 'Granted to the unquoted value names. The first two lines of class *Release* are equivalent to:

```
val release = 'release
val Granted = 'Granted
```

A problem illustrated here is, that in designing a DSL in Scala, and in most programming languages with an exception in Lisp, names are not first-class citizens. In supporting a DSL one will have to introduce a way of writing user defined names (if not using Scala's already existing features for introducing such, such as class, function, variable and constant definitions). LogFire uses symbols, single quoted names, for this purpose, which are easier to type than strings requiring two double quotes. However, even single quoted names are undesirable in a DSL, and therefore some approach to define names is useful, as attempted above.

7.2 Abstract Syntax for Rules

As mentioned, a rule is internally represented by an object of the class *Rule*, defined as follows, here simplified slightly for presentation purposes by eliminating some levels and by not presenting methods defined in these classes.

```
case class Rule(
  name: String,
  conditions: List[Condition],
  action: Action
)

trait Condition
case class PC(constraints: Map[Symbol, Pattern])
```

[8] We plan to use Scala's recently introduced macro features instead, and augment with parameter constraints.

```
  extends Condition
case class NC(constraints: Map[Symbol, Pattern])
  extends Condition

trait Pattern
case class Variable(s: Symbol) extends Pattern
case class Constant(s: Any) extends Pattern

case class Action(code: Unit => Unit)
```

These classes define the abstract syntax of rules: a rule consists of a name; a left-hand side, which is a list of conditions; and then a right-hand side, which is an action. A condition can either be a Positive Condition (PC) or a negated condition, here called a Negative Condition (NC)[9]. A condition is defined by a map from field names to field constraints: patterns. A pattern is either a variable (to be bound on first occurrence in a rule, and to be matched if not the first occurrence), or a constant that has to be matched. An action is a block of Scala code, here represented as a function from type Unit to Unit, Scala's void-type. The above rule is for example represented as the following *Rule* object (Abstract Syntax Tree):

```
  Rule(
    ”r4”,
    List(
      PC(
        Map(
          'kind -> Constant('Granted),
          'task -> Variable('t),
          'resource -> Variable('r)
        )
      ),
      PC(
        Map(
          'kind -> Constant('release),
          'task -> Variable('t),
          'resource -> Variable('r)
        )
      )
    ),
    Action((x: Unit) => remove('Granted))
  )
```

7.3 From Concrete to Abstract Syntax

It remains to be explained how this internal representation is generated from the rule r_4 shown above. The approach is based on the same techniques used in Section 4 for defining specification patterns, namely implicit functions lifting values of one type to values of another type, and the option of omitting dot-notation and parentheses around arguments when calling a method on an object. That is, for example a call such as `obj.meth(42)` can be written as: `obj meth 42`. Furthermore, method names can be sequences of symbols.

The following definitions define an implicit function *R*, lifting a string (a rule name) to an anonymous object, which defines the '−−' operator, which when applied to a condition returns an object of the class *RuleDef*. This class in turn defines the condition conjunction operator '&' and the action operator '|−>' defining the transition from left-hand side to right-hand side of the rule. This operator calls *addRule*, which adds the rule to the Rete network.

```
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)
      )
    )
  }
}
```

The implicit function *R* gets invoked by the compiler automatically when a string is followed by the symbol '−−', in order to resolve the type error (since there is no '−−' operator directly defined on strings). The individual conditions in a rule are similarly constructed with the help of the following implicit function, which lifts a symbol (the name of an event or fact) to a *Cond* object, which defines an *apply* function:

```
implicit def C(kind: Symbol) = new Cond(kind)

class Cond(kind: Symbol) {
  def apply(args: (Symbol, Any)*): PC = {
    val constraints =
      for ((field, value) <- args.toMap) yield {
        val pattern = value match {
          case symbol: Symbol => Variable(symbol)
          case _ => Constant(value)
        }
        (field -> pattern)
      }
    PC(constraints + ('kind -> Constant(kind)))
```

[9] LogFire also supports negation of a conjunction of conditions, as described in [30], although at this point we are unsure about the correctness of the algorithm for handling such provided in [30].

```
  }
}
```

The *apply* method in Scala has special interpretation: if an object *O* defines a such, the object can be applied to a list of arguments using function application syntax: *O(. . .)*, equivalent to calling the *apply* method: *O.apply(. . .)*. The *apply* method in this case takes a sequence of (Symbol,Any) pairs as argument, where a pair (s,a) in Scala can be written as s −>a. This makes it possible to write conditions such as:

```
  release ('task −> 't, 'resource −> 'r)
```

which by the compiler is translated into:

```
  C('release ). apply(('task,'t ),(' resource,' r))
```

The *apply* method returns a Positive Condition (PC) with the constraints formed by mapping values to patterns (symbols are mapped to *Variable*s and other values are mapped to *Constant*s). Negation of conditions is made possible with the following function:

```
  def not(pc: PC): Condition = NC(pc.constraints)
```

The complete interpretation by the Scala compiler of the rule definition:

```
  "r4" −−
    Granted('task −> 't, 'resource −> 'r) &
    release ('task −> 't, 'resource −> 'r) |−>
      remove(Granted)
```

finally becomes:

```
  R("r4").−−(
    C('Granted).apply(('task,'t ),  ('resource,' r))
  ).&(
    C('release ). apply(('task,'t ),  ('resource,' r))
  ). |−> (
    remove('Granted)
  )
```

7.4 Program Understanding with Data Visualization

The Rete algorithm has a reputation of being complicated. Comprehending the structure of the network generated from a set of rules, and how it evolves as events are processed, can be non-trivial. To ensure that the implementation works as desired is therefore a challenge. For the purpose of *program understanding*, we have developed a data structure visualization package in Scala, based on GraphViz [6]. This package is used to visualize the Rete network in a manner similar to Figure 1, although with more details.

The GraphViz layout program takes as input a representation of a graph in a simple text language, and generates a two-dimensional graph in a desired format. We implemented a Scala class *Graph*, an object of which represents the internal structure of a GraphViz graph, supporting all of the GraphViz notation. Given such a graph object, one can call methods such as for example *addNode*, *addEdge* (from one node to another), *addFields* (a data record: field-name/value pairs, associated with a node), *shapeNode*, and *colorNode*, with appropriate arguments, which have as side effect to build up this data structure. The data structure can finally be printed as a `.dot` file and visualized with GraphViz.

Each class, objects of which should be visualized, must extend the trait *Visual*, which provides two methods. The first method, *draw(file: String)*, produces a graph visualizing the object and writes it in GraphViz's text format to the file indicated as parameter. The second method, *toGraph(graph: Graph)*, is called from *draw*, and must be defined in each class extending trait *Visual*. It will update the argument graph with nodes and edges stemming from that object[10]:

```
trait Visual {
  def draw( file : String)
  def toGraph(graph: Graph)
}
```

As an example, below is shown a fragment of the code for the class AlphaMemory:

```
class AlphaMemory extends Visual {
  val items = new DoubleIndex[Field, Fact]()
  var successors: List [Joiner] = List()
  var referenceCount: Int = 0
   ...
  def toGraph(config: Config) {
    config .colorNode(this, "blue")
    config .addEdge(this, 'items, items)
    config .addEdges(this, 'successors, successors)
  }
   ...
}
```

As an example, the graph for the rule r_4 on page 18 is shown in Figure 2. The figure contains two join nodes (with rounded corners; purple in color print) corresponding to the two left-hand side conditions. The top right join node executes when a *Granted* fact is matched in the fact memory, while the second bottom left join node executes when in addition a *release* event is matched, where the *TestAtJoinNode* indicates the test to be performed on the attributes. With a complete match of the left-hand side the *PNode* executes the right-hand side.

[10] Note that this is similar to Java's and Scala's *toString* method, although different by updating an argument rather than returning a result (*toString* returns a string).

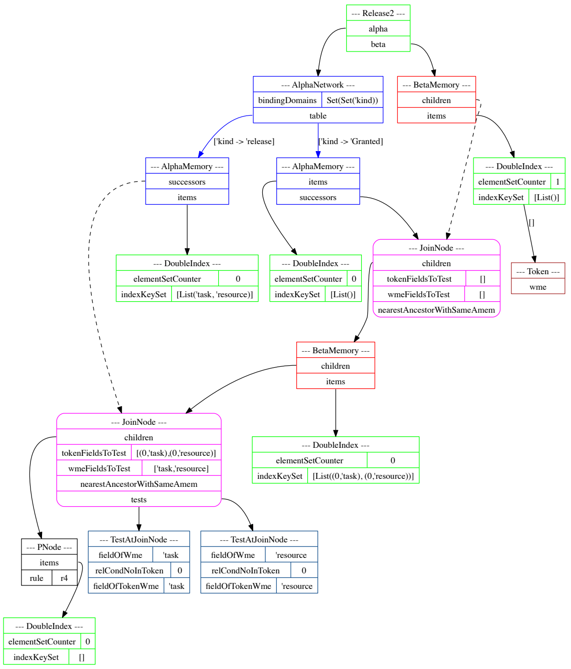


Fig. 2. Graph for the rule r_4 on page 18.

7.5 Testing

Performing automated testing of a language processing system like LogFire is difficult for two reasons: (i) inputs include specifications, which are rather complex objects to generate automatically; and (ii) one would need an oracle, which for every specification/trace pair tells us whether the trace satisfies the specification, and if not yielding an explanation of why not. The latter would essentially require implementing a second, preferably simpler, system with similar functionality, a *reference implementation*. It is perhaps here worth noting, that LogFire does not seem suitable for testing itself.

Instead, we have developed support for writing test cases (using ScalaTest [12]), where a test case consists of a specification, a trace, and an oracle for that specification/trace pair. The following test case illustrates this. A monitor is created and in this case fed two events. After each event it is asserted what facts should be in the fact memory. Finally it is asserted which errors the trace results in, in this case one violation and its error trace.

```
class TestResourceManagement extends Test {
  val m = new ResourceManagement
  setMonitor(m, false)

  add('grant(1, "A", 'motor3))
  assertFacts('Granted("A", 'motor3))

  add('grant(2, "B", 'motor3))
  assertFacts(
    'Granted("A",'motor3),
    'Granted("B",'motor3)
  )

  assertResult(
    Report(
      "ERROR bad double grant",
      (1, "r3", 'Granted("A", 'motor3)),
      (2, "r7", 'Fail("ERROR bad double grant")))
  )
}
```

If the test case fails to satisfy the oracle when executed, for example the first time it is executed with an empty oracle, the correct oracle is printed (if the second argument to *setMonitor* is true), and can be applied in future tests, assuming that its correctness is first manually confirmed. Any later modifications of the implementation will now be checked against each such test case.

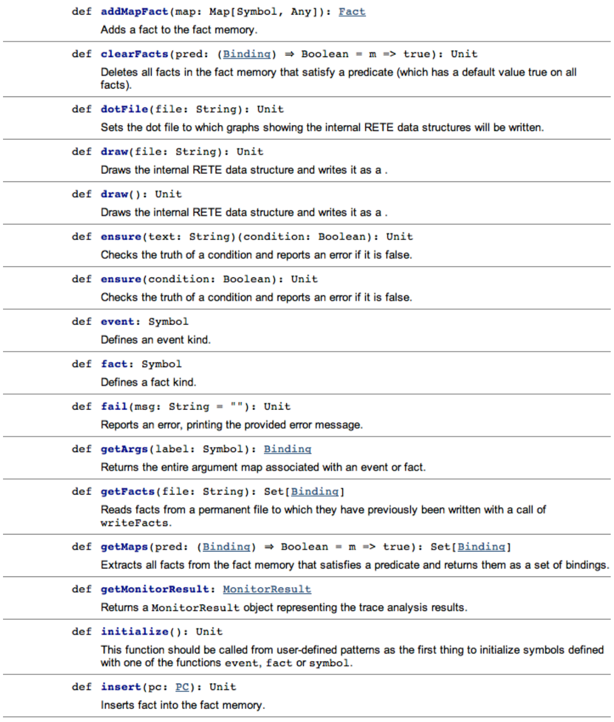
7.6 Documentation

LogFire has been documented as an API using ScalaDoc [11]. Figure 3 shows a page of the documentation. Clicking on an element (class, function, etc.) opens an explanation of that element. It should be noted that when documenting an internal Scala DSL as an API, one has in some cases to explain a grammar as a set of functions, classes and methods. This can in some cases seem somewhat inconvenient from a documentation point of view, as perhaps best illustrated by the definitions in Subsection 7.3, defining the "syntax" of rule definitions as a collection of implicit functions, objects and methods to be called in a chained manner.

8 Experiments

This section describes the benchmarking performed to evaluate LogFire and various other rule-based and/or runtime verification systems. The experiments have focused on analysis of logs (offline analysis), since this has been the focus of our application of RV, and since this must be considered an important practical RV application domain. The focus on logs does, however, not suggest that these techniques only can be used for log analysis. In fact, all the systems discussed and analyzed can be run in online mode, monitoring a system as it executes.

Offline and online analysis may require different techniques, including for example how garbage collection is handled. In a system such as Mop [58] handling of garbage collection is an important algorithmic problem. For example, if an object is monitored, and this object at some point is no longer used by the monitored application, then the monitor will allow the object to be garbage



```
def addMapFact(map: Map[Symbol, Any]): Fact
    Adds a fact to the fact memory.

def clearFacts(pred: (Binding) ⇒ Boolean = m => true): Unit
    Deletes all facts in the fact memory that satisfy a predicate (which has a default value true on all
    facts).

def dotFile(file: String): Unit
    Sets the dot file to which graphs showing the internal RETE data structures will be written.

def draw(file: String): Unit
    Draws the internal RETE data structure and writes it as a .

def draw(): Unit
    Draws the internal RETE data structure and writes it as a .

def ensure(text: String)(condition: Boolean): Unit
    Checks the truth of a condition and reports an error if it is false.

def ensure(condition: Boolean): Unit
    Checks the truth of a condition and reports an error if it is false.

def event: Symbol
    Defines an event kind.

def fact: Symbol
    Defines a fact kind.

def fail(msg: String = ""): Unit
    Reports an error, printing the provided error message.

def getArgs(label: Symbol): Binding
    Returns the entire argument map associated with an event or fact.

def getFacts(file: String): Set[Binding]
    Reads facts from a permanent file to which they have previously been written with a call of
    writeFacts.

def getMaps(pred: (Binding) ⇒ Boolean = m => true): Set[Binding]
    Extracts all facts from the fact memory that satisfies a predicate and returns them as a set of bindings.

def getMonitorResult: MonitorResult
    Returns a MonitorResult object representing the trace analysis results.

def initialize(): Unit
    This function should be called from user-defined patterns as the first thing to initialize symbols defined
    with one of the functions event, fact or symbol.

def insert(pc: PC): Unit
    Inserts fact into the fact memory.
```

Fig. 3. LogFire API documentation with ScalaDoc.

collected, usually using what is referred to as *weak references*. Future studies will reveal the need for this form of technique for log analysis. None of the systems compared, except Mop, handles this problem.

The evaluation was carried out on an Apple Mac Pro, 2 × 2.93 GHz 6-Core Intel Xeon, 32GB of memory, running Mac OS X Lion 10.7.5. Applications were run in Eclipse JUNO 4.2.2, running Scala IDE version 3.0.0/2.10 and Java 1.6.0.

8.1 The Systems Compared

The systems compared are the following (previously introduced in the related work Section 2):

- LogFire: the rule-based system presented in this paper, which is a augmentation of the Rete algorithm with event processing and an indexing scheme.
- Rete/UL: our Scala implementation of the Rete algorithm directly from [30], with the addition of event processing. LogFire is an optimization of this implementation with indexing. Comparing LogFire with Rete/UL offers perhaps the best insight into the benefits of the indexing.
- Drools [5]: one of the main state-of-the-art rule-based systems tailored for the Java language, and freely available from the JBoss community, where it is also referred to as their *Business Rules Management System (BRMS)* solution. Drools is based on an enhanced implementation of the Rete algorithm, for example with indexing (similar to our indexing optimization, although details are not available), and is therefore an interesting data point for comparison.
- Ruler [22,23,10]: is a runtime verification system implemented in Java, and created as a rule-based system, although with a quite different implementation than the Rete algorithm. Ruler removes a fact in the next step unless explicitly commanded to keep it by a rule. In contrast, Rete removes a fact only when explicitly commanded to do so by a rule.
- LogScope [43,18]: is a down-scaled and slightly modified version of Ruler, emphasizing data parameterized state machines, and implemented in Scala. LogScope, in contrast to Ruler, considers a fact to survive unless explicitly removed by a rule. LogScope was developed to explore language features and is not optimized.
- TraceContract [19]: an internal Scala DSL for state machines (shallow DSL) and LTL (deep), as well as a simple rule-based notation (shallow). It applies re-writing and normalization of formulas. TraceContract was developed to explore language features and is not optimized.
- Mop [58]: is amongst the most efficient state-of-the-art runtime verifications system in existence to our knowledge. Mop supports many different logics, provided as plugins, all using the same indexing algorithm to access monitors fast based on arguments to parameterized events. Normally Mop is designed to monitor programs as they execute (online monitoring). For example, JavaMop verifies events generated by a running Java program, instrumented with AspectJ [54,1]. Log analysis is in this evaluation performed by inserting a log reader program, which for each event read from the log, calls a method defined for the particular kind of event, and which has an empty method body (it does nothing). The calls of these empty-body methods are then instrumented with AspectJ to drive the monitors.

Of these systems, the author has contributed to/developed the following: Ruler (contributed to its design), LogScope, TraceContract, and LogFire, see [13].

8.2 Specification of Requirements

The systems are compared by specifying the same small set of requirements in each system's specification language, and subsequently analyze a collection of log files with each system against this specification. The requirements are closely related to the resource grant-and-release example presented in Subsections 3.2.4 and 3.2.5. We assume logs, which contain events such as: *grant*(t, r) (task t is granted resource r) and *release*(t, r) (task t releases resource r). Over such logs we formulate the following requirements:

- ***Release***: A resource granted to a task should eventually be released by that task.

- ***NoRelease***: A resource can only be released by a task, if it has been granted to that task, and not yet released.
- ***NoGrant***: As long as a resource is granted to a task, it cannot be granted again, neither to that task nor to any other task.

These requirements are formulated as follows in LogFire (and should be self-explanatory based on the descriptions provided in Subsections 3.2.4 and 3.2.5):

```
class ResourceRequirements extends Monitor {
  val grant, release, end = event
  val Granted = fact

  "r1" --
  grant('task -> 't, 'resource -> 'r) &
  not(Granted('t, 'r)) |-> Granted('t, 'r)

  "r2" --
  Granted('t, 'r) &
  release('task -> 't, 'resource -> 'r) |->
    remove(Granted)

  "r3" --
  Granted('t, 'r) &
  grant('task -> '_, 'resource -> 'r) |->
    fail("double grant")

  "r4" --
  Granted('t, 'r) & end() |-> fail("missing release")

  "r5" --
  release('task -> 't, 'resource -> 'r) &
  not(Granted('t, 'r)) |-> fail("bad release")
}
```

We shall not in this paper show the specifications formulated in the other systems.

8.3 The Logs

As will be illustrated in a moment, logs are for this experiment concretely represented as CSV files, the dominant representation form of MSL logs. The logs can, however, abstractly be seen as sequences of events $grant(t, r)$ and $release(t, r)$, where t and r are integer values. As an example, the following can be seen as an abstract log of 4 events:

```
grant(1,1)
grant(2,1)
release(1,1)
release(1,2)
```

This log by the way violates all three requirements (the interested reader may investigate why). Concretely, the logs are represented as CSV files, and parsed with a CSV-parsing script (a modified version of a script developed by MSL personal [53]) based on Scala's *parser combinator* library. Hence the above log is concretely represented as:

```
kind, task, resource
grant, 1, 1
grant, 2, 1
release, 1, 1
release, 1, 2
```

```
grant(1,1)    |
grant(2,2)    | G=3 grants
grant(3,3)    |
---
release(1,1)  | R=1 releases |
grant(1,1)    | R=1 grants   | L=2
---                          | release-grant
release(1,1)  | R=1 releases | blocks
grant(1,1)    | R=1 grants   |
---
release(1,1)  |
release(2,2)  | G=3 releases
release(3,3)  |
```

Fig. 4. Example log with shape $S = (G = 3, L = 2, R = 1)$

The experiment consists of analyzing 7 different logs: one log, numbered 1, generated from the Mars Curiosity rover during 99 (Mars) days of operation on Mars, together with 6 artificially generated logs, numbered 2-7, that are supposed to stress test the algorithms for their ability to handle particular situations requiring fast indexing. The MSL log contains a little over 2.4 million events, of which 30.933 are relevant grant and release events, which are extracted before analysis. The shape of this log is a sequence of paired grant and release events, where a resource is released in the step immediately following the grant event (after all other events have been filtered out). The log has the form (task ids are not relevant):

```
grant(1,1)
release(1,1)
grant(1,2)
release(1,2)
grant(1,3)
release(1,3)
...
```

In this case we say that the required *memory* is 1: only one ($task$, $resource$) association needs to be remembered at any point in time. In this sense there is no need for indexing since only one resource is held at any time. This might be a very realistic scenario in many cases. The artificially generated logs experiment with various levels of *memory* amongst the values: $\{1, 5, 30, 100, 500, 5000\}$. More specifically, each artificial log is characterized by a shape S:

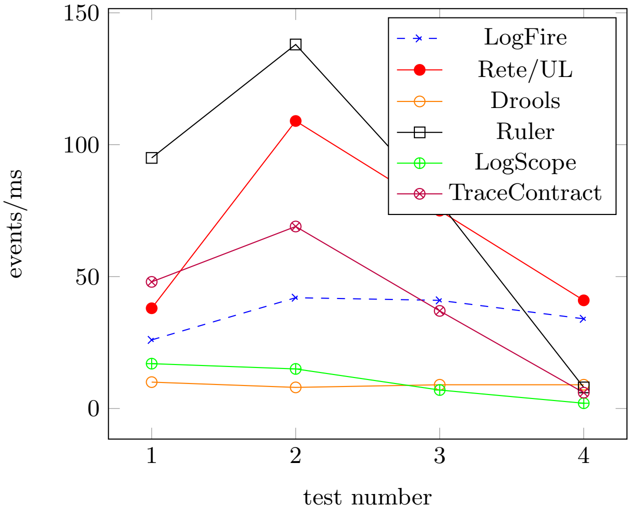


Fig. 5. Results of first 4 tests requiring light indexing.

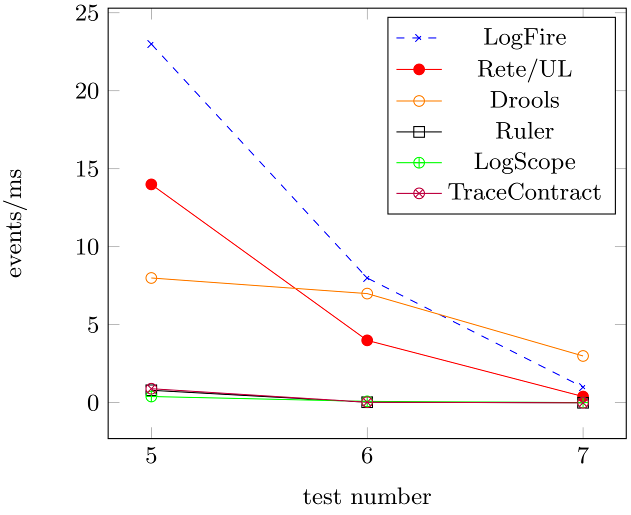


Fig. 6. Results of last 3 tests requiring heavy indexing.

$$S = (G, L, R)$$

consisting of 3 parameters:

- G: the number of distinct grants at the beginning of the log. This defines the *memory* of the log. The log is ended with releases of all these granted resources. What is in between is referred to as the *mid* section.
- L: the *mid* section consists of L *release-grant blocks*.
- R: a *release-grant block* in the mid section is charac-terized R releases of different resources followed by R grants of the same resources.

As an example, a log with shape $S = (G = 3, L = 2, R = 1)$, with comments inserted, is shown in Figure 4.

8.4 Results

The results of running all 7 systems on the 7 logs are shown in Table 1. The tools are grouped into three sections - truly rule-based systems: LogFire, Rete/UL, Drools, and Ruler; experimental un-optimized systems: LogScope and TraceContract, and the highly optimized system Mop. For each log is indicated: log number, shape $S = (G, L, R)$, length in terms of number events, and the time it takes on average across systems to parse the log and generate events for the systems to process (those times are approximately the same across systems for one particular log). For each system the parsing time is subtracted from the total time to process the log, yielding two numbers for each system: number of events processed per millisecond (above line), and time consumed monitoring (below line) measured in *min:sec:ms*, with minutes and seconds left out if 0.

The table shows that Mop outperforms all other systems by orders of magnitude. The difference is major. This fundamentally illustrates that the indexing approach used in a state-of-the-art system such as Mop is much faster than a standard algorithm from AI such as Rete for runtime verification purposes. This is an important observation in itself.

In order to better understand the differences between the remaining systems (excluding Mop for which the result is clear), the events per millisecond numbers per tool/log combination are also shown graphically in Figures 5 and 6. Figure 5 shows the results for logs 1-4, for relatively low values of G (memory), whereas Figure 6 shows the results for logs 5-7, for relatively high values of G. Two figures are shown in order to make the presentation clearer.

From Figure 5 (low values of G) it becomes clear that systems not implementing any form of advanced indexing (such as Ruler, Rete/UL, and even TraceContract) perform relatively well. LogFire only performs average on such logs. However, it performs better than the state-of-art rule system Drools. We do not know the reason for this. Drools performs approximately as well as the completely unoptimized LogScope.

From Figure 6 (high values of G) we see that LogFire performs better than the other systems, including Drools, except for the extreme value $G = 5000$ where Drools suddenly performs better. Again, we are uncertain about the reason for this behavior. LogFire performs better than Rete/UL, which would be expected. Ruler, LogScope, and TraceContract all underperform for large values of G. For $G = 5000$ (test number 7), Ruler and TraceContract did not finish within 72 hours, and had to be forcefully terminated.

In summary, Mop's indexing solution outperforms the rule-based algorithms for this set of typical runtime verification properties. LogFire generally performs better than the state-of-art rule-based system Drools, but does only average for low values of G. A main lesson to be learned is that the indexing approach used in a sys-

Table 1. Results of tests 1-7. For each test is shown shape of the test (m stands for million and k stands for thousand), length of the trace, and time taken to parse the log. For each tool two numbers are provided - above line: number of events processed by the monitor per millisecond, and below line: time consumed monitoring (minutes:seconds:milliseconds, with minutes and seconds left out if 0). DNF stands for 'Did Not Finish'.

trace nr.	1	2	3	4	5	6	7
S=(G,L,R)	msl log	(1,1m,1)	(5,350k,3)	(30,100k,10)	(100,100k,10)	(500,10k,100)	(5000,5k,100)
length	30,933	2,000,002	2,100,010	2,000,060	2,000,200	2,001,000	1,010,000
parsing	3 sec	45 sec	47 sec	46 sec	46 sec	46 sec	24 sec
LogFire	$\frac{26}{1:190}$	$\frac{42}{47:900}$	$\frac{41}{50:996}$	$\frac{34}{58:391}$	$\frac{23}{1:27:488}$	$\frac{8}{3:55:696}$	$\frac{1}{15:54:769}$
Rete/UL	$\frac{38}{816}$	$\frac{109}{18:428}$	$\frac{75}{28:141}$	$\frac{41}{48:524}$	$\frac{14}{2:26:983}$	$\frac{4}{8:25:867}$	$\frac{0.4}{43:33:366}$
Drools	$\frac{10}{3:97}$	$\frac{8}{4:1:758}$	$\frac{9}{3:47:535}$	$\frac{9}{3:34:648}$	$\frac{8}{4:14:497}$	$\frac{7}{4:36:608}$	$\frac{3}{5:4:505}$
Ruler	$\frac{95}{326}$	$\frac{138}{14:441}$	$\frac{78}{27:77}$	$\frac{8}{4:5:593}$	$\frac{0.8}{41:39:750}$	$\frac{0.034}{977:20:636}$	DNF
LogScope	$\frac{17}{1:842}$	$\frac{15}{2:11:908}$	$\frac{7}{4:54:605}$	$\frac{2}{21:42:389}$	$\frac{0.4}{76:17:341}$	$\frac{0.09}{369:25:312}$	$\frac{0.01}{2074:43:470}$
TraceContract	$\frac{48}{645}$	$\frac{69}{28:851}$	$\frac{37}{57:428}$	$\frac{6}{5:58:497}$	$\frac{0.9}{36:29:594}$	$\frac{0.036}{919:5:134}$	DNF
Mop	$\frac{595}{52}$	$\frac{1381}{1:448}$	$\frac{1559}{347}$	$\frac{1341}{1:491}$	$\frac{7143}{280}$	$\frac{7096}{282}$	$\frac{847}{1:193}$

tem such as Mop may be a useful technique to apply to rule-based systems.

9 Conclusion and Future Work

Rule-based systems seem natural for runtime verification/program monitoring. From a specification notation point of view rule-based systems appear quite suitable for expressing the kind of properties the runtime verification community normally writes. Specifications written in a rule system have an operational flavor, which can be seen as a disadvantage or an advantage, depending on the view point. The operational flavor makes specifications longer than in declarative temporal logic or regular expressions. However, they are natural to write. Once the core idea is mastered, writing rules is straight forward, like programming. More declarative specifications can be more tricky to get right. This observation is similar to the observation, that it may be easier to formulate a non-trivial property as a state machine than as a temporal logic formula or a regular expression. As we have seen, one can in addition define specification templates allowing for more succinct specifications. Rule-based systems are fully expressive, in fact Turing complete, making them for example strictly more expressive than several declarative notations.

From a performance point of view, however, a system like Mop clearly is superior. The difference is substantial, suggesting a future study of the relationship between the indexing approaches used in runtime verification (in Mop in particular) and the algorithms behind the more expressive rule-based systems, such as Rete. For low *memory* (small values of G), Ruler interestingly performs better than Rete/UL, which performs better than LogFire. For high *memory* (large values of G) the results are turned around, and here LogFire performs better than the other rule-based systems, except for $G = 5000$ where Drools outperforms LogFire.

LogFire is developed as an internal DSL in Scala. We believe that an internal DSL has advantages over an external (stand-alone) DSL for log processing. Under practical circumstances log processing usually requires complex operations to be performed in addition to the pure detection of event patterns. Having access to a general purpose high-level programming language offering object-oriented as well as functional programming features seems very attractive for test engineers for example. LogFire is a mixture of a deep and a shallow internal DSL. Right-hand sides of rules are formed in a shallow DSL: directly from Scala code (any Scala statement with return type *Unit*). Left-hand sides of rules are formulated as a deep internal DSL, meaning: all constructs are elements of explicitly defined Scala data types. This makes it possible to process left-hand sides in order to generate the Rete network.

Concerning current and future work, we are currently evaluating LogFire on a more comprehensive set of logs. This work will include log abstraction ([53], see discussion below) and visualization. We are also considering various extensions and modifications to the framework. One of the issues encountered is the need for using quoted names for events and facts. These may be better handled with Scala's new macro concept. The ideal would be to have events and facts be Scala objects, as is the case in Drools and TraceContract. However, this is made complicated by the need for pattern matching over such objects, which is difficult when defining a deep internal DSL, as LogFire in part is (what concerns left-hand sides). TraceContract is a shallow internal DSL in contrast, making it possible to re-use Scala's pattern matching over objects. We plan to explore the boundary between shallow and deep internal DSLs, as represented by these two languages. More broadly we intend to explore some of the state machine notations existing in TraceContract, Ruler, and LogScope, for example state machines with anonymous (un-named) states.

None of the evaluated tools take explicit advantage of multi-threading. Additional efficiency can be obtained by exploring multi-threading solutions to the monitoring problem, as for example performed in Hammurabi [39].

A topic only briefly mentioned so-far in Section 2 is the ability of using a rule-based system to specify event abstraction. The idea is to infer facts from events, while having additional rules which from facts infer other facts. This idea was in part illustrated with the *Before* fact in Subsection 3.2.3. Here events such as $\mathit{before}(r_1, r_2)$ and $\mathit{before}(r_2, r_3)$ would generate facts such as $\mathit{Before}(r_1, r_2)$ and $\mathit{Before}(r_2, r_3)$, from which a rule of the form:

```
"r2" -- Before('r1, 'r2) & Before('r2, 'r3) |->
    Before('r1, 'r3)
```

generates a new fact $\mathit{Before}(r_1, r_3)$. This is an example of event abstraction: from single events are derived facts, from which other facts are derived. This form of specification allows to build abstractions over the log, as suggested in [53], which can be useful for processing complex information. In current work this idea is being applied to logs from the MSL Mars mission.

Acknowledgements

We thank Howard Barringer (University of Manchester, UK) for numerous fruitful discussions. Also thanks to Giles Reger (University of Manchester, UK) and Patrick Meredith (Mop project, University of Illinois at Urbana-Champaign, IL, USA) for their input on how to write specifications optimally in Mop, and to Patrick Meredith for his general support in using the Mop system. Thanks to Mark Proctor, Davide Sottara, and Edson Tirelli (the Drools project) for their support in using Drools. Thanks to Rajeev Joshi (Jet Propulsion Laboratory, California, USA) for his help in parsing and analyzing MSL logs, including helping formulating properties over these logs. Rajeev Joshi also came up with the suggestion that abstraction over logs could be useful. As it turns out, LogFire (more generally: any Rete-based system) offers this functionality. The work was carried out at Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration. The work was furthermore supported by NSF Grant CCF-0926190.

References

1. AspectJ website. http://www.eclipse.org/aspectj.
2. Clips website. http://clipsrules.sourceforge.net.
3. Drools blog. http://blog.athico.com/2013/01/life-beyond-rete-rip-rete-2013.html.
4. Drools functional programming extensions website. https://community.jboss.org/wiki/FunctionalProgrammingInDrools.
5. Drools website. http://www.jboss.org/drools.
6. Graphviz website. http://www.graphviz.org.
7. Jess website. http://www.jessrules.com/jess.
8. Mars Science Laboratory (MSL) mission website. http://mars.jpl.nasa.gov/msl.
9. Rooscaloo website. http://code.google.com/p/rooscaloo.
10. RuleR website. http://www.cs.man.ac.uk/∼howard/LPA.html.
11. Scaladoc website. https://wiki.scala-lang.org/display/SW/Scaladoc.
12. Scalatest website. http://www.scalatest.org.
13. Website for various runtime verification tools, including: Eagle, RuleR, LogScope, TraceContract, and LogFire. http://www.havelund.com/tools.
14. 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.
15. 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.
16. H. Barringer, M. Fisher, D. M. Gabbay, G. Gough, and R. Owens. Metatem: An introduction. *Formal Asp. Comput.*, 7(5):533–549, 1995.
17. H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-based runtime verification. In *VMCAI*, volume 2937 of *LNCS*, pages 44–57. Springer, 2004.
18. 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.
19. 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.
20. H. Barringer, K. Havelund, E. Kurklu, and R. Morris. Checking flight rules with TraceContract: Application of a Scala DSL for trace analysis. In *Scala Days 2011, Stanford University, California*, 2011.
21. 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.
22. 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.
23. 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.

24. 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 *Lecture Notes in Computer Science*, pages 1–18. Springer, 2010.
25. 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.
26. 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.
27. 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.
28. 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.
29. 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.
30. R. B. Doorenbos. *Production Matching for Large Learning Systems.* PhD thesis, Carnegie Mellon University, Pittsburgh, PA, 1995.
31. 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.
32. D. Drusinsky. *Modeling and Verification using UML Statecharts.* Elsevier, 2006. ISBN-13: 978-0-7506-7949-7, 400 pages.
33. M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. In B. W. Boehm, D. Garlan, and J. Kramer, editors, *Proceedings of the 1999 International Conference on Software Engineering, ICSE' 99, Los Angeles, CA, USA, May 16-22, 1999*, pages 411–420. ACM, 1999.
34. 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.
35. 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.
36. Y. Falcone, K. Havelund, and G. Reger. A tutorial on runtime verification. In M. Broy, D. Peled, and G. Kalus, editors, *Engineering Dependable Software Systems*, volume 34 of *NATO Science for Peace and Security Series - D:Information and Communication Security*, pages 141–175. IOS Press, 2013.
37. C. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19:17–37, 1982.
38. M. Fowler and R. Parsons, editors. *Domain-Specific Languages.* Addison-Wesley, 2010.
39. M. Fusco. Hammurabi - a Scala rule engine. In *Scala Days 2011, Stanford University, California*, 2011.
40. F. Garillot and B. Werner. Simple types in type theory: Deep and shallow encodings. In *20th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'07), Kaiserslautern, Germany. Proceedings*, volume 4732 of *LNCS*, pages 368–382. Springer, 2007.
41. 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.
42. 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.
43. 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.
44. S. Hallé and R. Villemaire. Runtime enforcement of web service message contracts with data. *IEEE Transactions on Services Computing*, 5(2):192–206, 2012.
45. K. Havelund. Runtime verification of C programs. In *Proc. of the 1st TestCom/FATES conference*, volume 5047 of *LNCS*, Tokyo, Japan, 2008. Springer.
46. K. Havelund. What does AI have to do with RV? (extended abstract). In T. Margaria and B. Steffen, editors, *5th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation. Track: Runtime Verification - the Application Perspective (organized by Ylies Falcone and Lenore Zuck), Heraclion, Greece, October 15-18. Proceedings*, volume 7610 of *LNCS*. Springer, 2012.
47. K. Havelund. A Scala DSL for Rete-based runtime verification. In *Runtime Verification - 4th Int. Conference, RV'13, Rennes, France, September 24-27, 2013. Proceedings*, volume 8174 of *LNCS*, pages 322–327. Springer, 2013.
48. K. Havelund and A. Goldberg. Verify your runs. In *Verified Software: Theories, Tools, Experiments, VSTTE 2005*, pages 374–383, 2008.
49. K. Havelund and G. Roşu. Efficient monitoring of safety properties. *Software Tools for Technology Transfer*, 6(2):158–173, 2004.
50. K. Havelund and G. Rosu. Monitoring programs using rewriting. In *16th ASE conference, San Diego, CA, USA*, pages 135–143, 2001.
51. C. Herzeel, K. Gybels, and P. Costanza. Escaping with future variables in HALO. In *Proc. of the 7th Int. Workshop on Runtime Verification (RV'07)*, volume 4839 of *LNCS*, pages 51–62. Springer, 2007.
52. G. J. Holzmann and R. Joshi. Model-driven software verification. In *Model Checking Software - the 11th International SPIN Workshop, Barcelona, Spain*, volume 2989 of *LNCS*, pages 76–91. Springer, 2004.
53. R. Joshi. Resources for analyzing MSL logs, personal communication. 2013.
54. 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.

55. 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.
56. M. Leucker and C. Schallhart. A brief account of runtime verification. *Journal of Logic and Algebraic Programming*, 78(5):293–303, may/june 2008.
57. D. Luckham, editor. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley, 2002.
58. 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.
59. M. Perlin. Topologically traversing the Rete network. *Applied Artificial Intelligence*, 4(3):155–177, 1990.
60. A. Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science*, pages 46–57. IEEE Computer Society, 1977.
61. 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.
62. 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.
63. 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.