A Scala DSL for Rete-based Runtime Verification

Klaus Havelund*

Jet Propulsion Laboratory
California Institute of Technology, California, USA

Abstract. Runtime verification (RV) consists in part of checking execution traces against formalized specifications. Several 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 RETE-based runtime verification system, named LOGFIRE (originally intended for offline log analysis but also applicable to online analysis), as an internal DSL in the SCALA programming language, using SCALA's support for defining DSLs. This combination appears attractive from a practical point of view. Our contribution is in part conceptual in arguing that such rule-based frameworks originating from AI may be suited for RV.

1 Introduction

Runtime Verification (RV) consists of monitoring the behavior of a system, either online as it executes, or offline after its execution, for example by analyzing log files. Although this task seems easier than verification of all possible executions, this task is challenging. From an *algorithmic* point of view the challenge consists of efficiently processing events that carry data. 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. 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. Most of these systems implement specification languages which are based on state machines, regular expressions, temporal logic, or grammars. The most efficient of these, for example [11], however, tend to have limited expressiveness as discussed in [3].

* The work described in this publication was carried out at Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

It can be observed that rule-based programming seems like an attractive approach to monitoring, as exemplified by the RULER system [5]. Rules are of the form $lhs \Rightarrow rhs$, where the left-hand side are conditions on a memory of *facts*, and the right-hand side is an action that can add or remove facts, or execute other code, including yielding error messages. This model seems very well suited for processing data rich events, and is simple to understand due to its operational flavor. 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. The RETE algorithm [6] is a well-established efficient pattern-matching algorithm for implementing such production rule systems. It maintains a network of nodes through which facts filter to the leaves where actions (rule right-hand sides) are executed. It avoids re-evaluating conditions in a rule's left hand side each time the fact memory changes. This algorithm has acquired a reputation for "extreme difficulty". Our primary goal with this work has been to understand how well this algorithm serves to solve the runtime verification task, and hence attempt to bridge two communities: formal methods and artificial intelligence. An initial discussion of this work was first presented in [8]. We have specifically implemented a rule-based system, named LOGFIRE, based on the RETE algorithm in the SCALA programming language as an internal DSL, essentially extending SCALA with rule-based programming. We have made some modifications to the 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. Early results show that the algorithm performs reasonably, although not as optimal as specialized RV algorithms, such as MOP [11].

There are several well-known implementations of the RETE algorithm, including DROOLS [1]. These systems offer external DSLs for writing rules. The DROOLS project has an effort ongoing, defining functional programming extensions to DROOLS. In contrast, by embedding a rule system in an object-oriented and functional language such as SCALA, 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 required by runtime verification. The event concept in DROOLS is inspired by the concept of *Complex Event Processing* [10]. Two rule-based internal DSLs for SCALA exist: HAMMURABI [7] and ROOSCALOO [2]. HAMMURABI is not RETE-based, and instead evaluates rules in parallel. ROOSCALOO [2] is RETE based, but is not documented in any form other than experimental code. A RETE-based system for aspect-oriented programming with history pointcuts is described in [9]. The system offers a small past time logic, which is implemented with a modification of the RETE algorithm. This is in contrast to our approach where we maintain the core of the RETE algorithm, and instead write or generate rules reflecting specifications. In previous work we designed the internal SCALA DSL TRACECONTRACT for automaton and temporal logic monitoring [4]. An internal SCALA DSL for 'design by contract' is presented in [12].

2 The LogFire DSL

In this section we shall illustrate LogFire by specifying a monitor for the resource management system for a planetary rover. Subsequently we will briefly explain the operational meaning of the specification.

2.1 Specification

Consider a rover that runs a collection of tasks in parallel. A resource arbiter manages resource allocation, ensuring for example that a resource is only used by one task at a time. Consider that we monitor logs containing the events:

$grant(t, r)$: task t is granted resource r.
$release(t, r)$: task t releases resource r.
$end()$: the end of the log is reached.

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

> *"A resource can only be granted to one task (once) at any point in time, and must eventually be released by that task."*

We shall now formalize this requirement as a LogFire monitor. The main component of LogFire is the class *Monitor*, which any user-defined monitor must extend to get access to constants and methods provided by LogFire. User-defined monitors will contain rules of the form:

$$name \texttt{ -- } condition_1 \texttt{ \& } \ldots \texttt{ \& } condition_n \texttt{ |-> } action$$

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 much 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. Our monitor becomes:

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

  "r1" -- grant('t, 'r) & not(Granted('t, 'r)) |-> Granted('t, 'r)
  "r2" -- Granted('t, 'r) & release('t, 'r) |-> remove(Granted)
  "r3" -- Granted('t, 'r) & grant('_, 'r) |-> fail("double grant")
  "r4" -- Granted('t, 'r) & end() |-> fail("missing release")
  "r5" -- release('t,'r) & not(Granted('t,'r)) |-> fail("bad release")
}
```

Value definitions introduce event and fact names. Rule r_1 formalizes that if a *grant*($'t$,$'r$) is observed, and no *Granted*($'t$,$'r$) fact exists in the fact memory (with the 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. Rule r_2 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 remaining rules express the error situations - r_3: granting an already granted resource, r_4: ending monitoring with a non-released resource, and r_5: releasing a resource not granted to the releasing task. LogFire allows to write any Scala code on the right-hand side of a rule, just as any Scala definitions are allowed in LogFire monitors, including local variables and methods. We can create an instance of this monitor and submit events to it (not shown here), which then get verified for conformance with the rules. Any errors will be documented with an error trace illustrating what events caused what rules to fire.

2.2 Meaning

Each rule definition in the class *ResourceProperties* is effectively a method call, or rather: a chain of method calls (commonly referred to as *method chaining*), which get called when the class gets instantiated (a Scala class body can contain statements). Note that Scala allows to omit dots and parentheses in method calls. As an example, the definition of rule r_2 is equivalent to the statement:

```
R("r2").--(C('Granted)('t, 'r)).&(C('release)('t, 'r)). |-> {
  remove('Granted)
}
```

The functions *R* (standing for *Rule*) and *C* (standing for *Condition*) are so-called *implicit functions*. An implicit function in Scala is defined as part of the program (in this case in the class *Monitor*), but is not explicitly called. Such functions are instead applied by the compiler in cases where type checking fails but where it succeeds if one such (unique) implicit function can be applied. In the statement above we have inserted them explicitly for illustration purposes, as the compiler will do. The function *R* takes a string as argument and returns an object of a class, which defines a function `--`, which as argument takes a condition, and returns an object, which defines a method `&`, which takes a condition, and returns an object defining a method `|->`, which takes a Scala statement (passed call by name, hence not yet executed), and finally creates a rule internally.

Creating the rules internally means building the Rete network as an internal data structure in the instantiated *ResourceProperties* object, representing the semantics of the rules. Figure 1 illustrates the network created by the definitions of rules r_2, r_3, and r_4 (rules r_1 and r_5 contain negated conditions which are slightly complicated, and therefore ignored in this short exposition). When events and facts are added to the network, they sift down from the top. For example, a *Granted*$(7, 32)$ event will end up in the lower grey buffer, from which three *join nodes* lead to different actions depending on what the next event is: *release*, *grant*, or *end*. The join nodes perform matching on arguments.

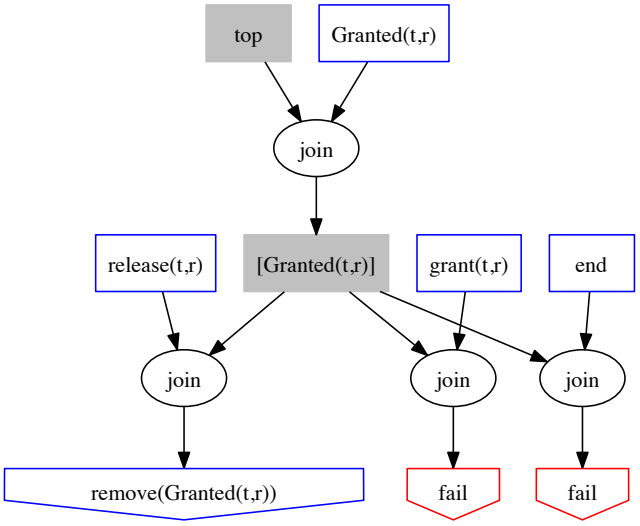

Fig. 1. The RETE network for rules r_2, r_3, and r_4.

2.3 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, the more succinct a specification is, the better. We have as an example implemented a specification pattern in 50 lines of SCALA code in the class *PathMonitor* (not shown here). In a path 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 requirement that a resource should not be granted to a task if it is already granted:

```
class DoubleGrant extends PathMonitor {
  when("double grant")(grant('t, 'r), no(release('t, 'r)), grant('_, 'r)) {
    fail()
  }
}
```

The property states that when a $grant('t,'r)$ is observed, and then subsequently another $grant('\_,'r)$ of the same resource, without a $release('t,'r)$ in between,

then the code provided as the last argument is executed, in this case just the reporting of a failure. The function *when* is itself defined as a sequence of rule definitions.

3 Conclusion

We have illustrated how rule-based programming based on the RETE algorithm, integrated in a high-level programming language, can be used for runtime verification. The initial experiments show that the system is very expressive and convenient, and is acceptable from a performance point of view, although not as efficient as optimized specialized RV algorithms.

References

1. Drools website. http://www.jboss.org/drools.
2. Rooscaloo website. http://code.google.com/p/rooscaloo.
3. 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.
4. 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.
5. 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.
6. C. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19:17–37, 1982.
7. M. Fusco. Hammurabi - a Scala rule engine. In *Scala Days 2011, Stanford University, California*, 2011.
8. K. Havelund. What does AI have to do with RV? (extended abstract). In *5th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation, Heraclion, Greece, October 15-18.*, volume 7610 of *LNCS*. Springer, 2012. Invited presentation.
9. 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.
10. D. Luckham. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley, 2002.
11. 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.
12. M. Odersky. Contracts for Scala. In *Runtime Verification - First Int. Conference, RV'10, St. Julians, Malta, November 1-4, 2010. Proceedings*, volume 6418 of *LNCS*, pages 51–57. Springer, 2010.