Monitoring with Data Automata

Klaus Havelund[⋆]

Jet Propulsion Laboratory
California Institute of Technology
California, USA

Abstract. We present a form of automaton, referred to as *data automata*, suited for monitoring sequences of data-carrying events, for example emitted by an executing software system. This form of automata allows states to be parameterized with data, forming named records, which are stored in an efficiently indexed data structure, a form of database. This very explicit approach differs from other automaton-based monitoring approaches. Data automata are also characterized by allowing transition conditions to refer to other parameterized states, and by allowing transitions sequences. The presented automaton concept is inspired by rule-based systems, especially the Rete algorithm, which is one of the well-established algorithms for executing rule-based systems. We present an optimized external DSL for data automata, as well as a comparable unoptimized internal DSL (API) in the Scala programming language, in order to compare the two solutions. An evaluation compares these two solutions to several other monitoring systems.

1 Introduction

Runtime verification (RV) is a sub-field of software reliability focused on how to monitor the execution of software, checking that the behavior is as expected, and if not, either produce error reports or modify the behavior of the software as it executes. The executing software is instrumented to emit a sequence of events in some formalized event language, which is then checked against a temporal specification by the monitor. This can happen during test before deployment, or during deployment in the field. Orthogonally, monitoring can occur online, simultaneously with the running program, or offline by analyzing log files produced by the running program. Many RV systems have appeared over the last decade. The main challenges in building these systems consist of defining expressive specification languages, which also makes specification writing attractive (simple properties should have simple formulations), as well as implementing efficient monitors for such. A main problem is how to handle data-carrying events efficiently in a temporal setting. Consider for example the following event stream consisting of three $grant(t, r)$ events (resource r is granted to task t):

[⋆] 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.

$\langle grant(t_1, a), grant(t_2, b), grant(t_3, a)\rangle$, and consider the property that no resource should be granted to more than one task at a time. When receiving the third event $grant(t_3, a)$, the monitor has to search the relevant history of seen events, which, if one wants to avoid looking at the entire history, in the presence of data ends up being a data indexing problem in some form or another.

RV systems are typically based on variations of state machines, regular expressions, temporal logics, grammars or rule-based systems. Some of the most efficient RV systems tend to be limited wrt. expressiveness [2], while very expressive systems tend to not be competitive wrt. efficiency. Our earlier work includes studies of rule-based systems, including Ruler [6] and LogFire [15]. As example of a rule in a rule-based system, consider: $Granted(t, r) \wedge grant(t', r) \Rightarrow Error(t, t', r)$. 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 named 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. 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. The rule above states that if the fact memory contains a fact that matches $Granted(t, r)$ for some task t and resource r, and a $grant(t', r)$ event is observed, then a new fact $Error(t, t', r)$ is added to the fact memory. A well-established algorithm for efficiently executing rule-based systems is the Rete algorithm [12], which we implemented in the LogFire system [15] as an internal DSL (API essentially) in the Scala programming language, while adopting it for runtime verification (supporting events in addition to facts), and by optimizing fact search using indexing.

While an interesting solution, the Rete algorithm is complex. Our goal is to investigate a down-scaled version of Rete to an automaton-based formalism, named *data automata* (Daut), specifically using the indexing approach implemented in [15]. Two alternative solutions are presented and compared. First, data automata are presented as a so-called *external DSL*, a stand-alone formalism, with a parser and interpreter implemented in Scala. The formalism has some resemblance to process algebraic notations, such as CSP and CCS. Second, we present an unoptimized *internal DSL* (Daut^{int}), an API in the Scala programming language, with a very small implementation, an order of magnitude smaller compared to the external DSL (included in its entirety in Appendix A). An internal DSL has the advantage of offering all the features of the host programming language in addition to the features specific to the DSL itself. We compare these two solutions with a collection of other monitoring systems.

The paper is organized as follows. Section 2 outlines related work. Section 3 presents data automata, as the external DSL named Daut, including their pragmatics, syntax, and semantics. Section 4 presents an indexing approach to obtain more efficient monitors for data automata. Section 5 presents the alternative internal Scala DSL named Daut^{int}, which also implements the data automaton concept. Section 6 presents an evaluation, comparing performance with other systems. Section 7 concludes the paper.

2 Related Work

The inspiration for this work has been our work on the rule-based LogFire system [15], which again was inspired by the Ruler system [6]. The external DSL is closely related to LogScope [4]. The internal Scala DSL is a modification of the internal Scala DSL TraceContract [5]. As such this work can be seen as presenting a reflection of these four pieces of work.

The first systems to handle parameterized events appeared around 2004, and include such systems as Eagle [3] (a form of linear μ-calculus), Jlo [18] (linear temporal logic), TraceMatches [1] (regular expressions), and Mop [16] (allowing for multiple notations). Mop seems the most efficient of all systems. The approach applied is referred to as *parametric trace slicing*. 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. This indexing approach results in an impressive performance. However, this is at the price of some lack of expressiveness in that properties cannot relate different slices, as also pointed out in [2]. MopBox [10] is a modular Java library for monitoring, implementing Mop's algorithms.

Quantified Event Automata [2] is an automaton concept for monitoring parameterized events, which extends the parametric trace slicing approach used in 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, similar to what is attempted in the here presented work. The work is also closely related to Orchids [13], which is a comprehensive state machine based monitoring framework created for intrusion detection.

Several systems have appeared that monitor first order extensions of propositional linear temporal logic (LTL). A majority of these are inspired by the classical rules (Gerth et. al) for rewriting LTL. These extensions include [17], an extension of LTL with a binding operator, and implemented using alternating automata; LTL-FO^{+} [14], for parameterized monitoring of Xml messages communicated between web-services; Mfotl [7], a metric first-order temporal logic for monitoring, with time constraints as well as universal and existential quantification over data; LTL$^{\text{FO}}$ [8], based on spawning automata; and [11], which

Listing 1. Monitor for requirements R_1 and R_2

```
monitor R1R2 {
  init always Start {
    grant(t, r) → Granted(t,r)
    release(t, r) :: ¬Granted(t,r) → error
  }

  hot Granted(t,r) {
    release(t,r) → ok
    grant(_,r) → error
  }
}
```

uses a combination of classical monitoring of propositional temporal properties and SMT solving.

3 The Daut Calculus

3.1 Illustration by Example

We shall introduce Daut by example. Consider a scenario where we have to write a monitor that monitors sequences of $grant(t, r)$ and $release(t, r)$ events, representing respectively granting a resource r to a task t, and task t releasing resource r. Consider furthermore the two requirements R_1: *"a grant of a resource to a task must be followed by a release of that resource by the same task, without another grant of that resource in between (to the same task or any other task)"*, and R_2: *"a resource cannot be released by a task, which has not been granted the resource"*. These requirements can be formalized in Daut as shown in Listing 1. The monitor has the name *R1R2*. It contains two states *Start* and *Granted*, the latter of which is parameterized with a task t and a resource r. The *Start* state is the initial state indicated by the modifier **init**. Furthermore, it is an **always** state, meaning that whenever a transition is taken out of the state, an implicit self-loop keeps the state around to monitor further events. In the *Start* state when a $grant(t, r)$ event is observed, a $Granted(t, r)$ state is created. If a $release(t, r)$ event is observed, and the condition occurring after :: is *true*, namely that there is no $Granted(t, r)$ state active, it is an error. The state $Granted(t, r)$ is a so-called hot state, which essentially is a non-final state. It is an error to remain in a hot state at the end of a log analysis.

The formalism allows for various abbreviations. For example, it is possible to write transitions at the top level, as a shorthand for introducing a state with modifiers **init** and **always**. This is illustrated by the monitor in Listing 2, which is semantically equivalent to the monitor in Listing 1. Also, target states can

Listing 2. Simplified monitor

```
monitor R1R2 {
  grant(t, r) → Granted(t,r)
  release(t, r) :: ¬Granted(t,r) → error

  hot Granted(t,r) {
    release(t,r) → ok
    grant(_,r) → error
  }
}
```

Listing 3. Monitor for requirement R_1

```
monitor R1 {
  grant(t, r) → hot {
    release(t,r) → ok
    grant(_,r) → error
  }
}
```

be "inlined", making it possible to write sequences of transitions without mentioning intermediate states. This is a shorthand for the longer form where each intermediate state is named. As an example, requirement R_1 can be stated succinctly as shown in Listing 3. Such nesting can be arbitrarily deep, corresponding to time lines. This makes it possible to write monitors that resemble temporal logic, as also was possible in TraceContract [5].

In general, states can be parameterized with arbitrary values represented by expressions in an expression language (not just identifiers as in some RV approaches, for example Mop). The formalism allows counting, as an example. The right-hand sides of transitions can for brevity also be conditional expressions, where conditions can refer to state and event parameters, as well as other states. To summarize, this automaton concept supports parameterized events, parameterized states, transition conditions involving state and event parameters as well as other parameterized states, expressions as arguments to states, and conjunction of conditional target states. What is not implemented from classical rule-based systems is disjunction of target states (as in Ruler), variables and general statements as actions, deletion of facts in general (only the state from which a transition leads is deleted when taking the transition, except if it is an **always** state), and general unification across conditions. A further extension of this notation (not pursued in this work) could allow declaration of variables local to a monitor, reference to such in conditions, as well as arbitrary statements with side-effects on these variables in right-hand side actions. The internal SCALA DSL DAUT^{int} presented in Section 5 does support these extensions.

3.2 Syntax

The presentation of data automata shall focus on the syntax of such, as used in the specifications seen in the previous subsection. The full grammar for DAUT is shown in Figure 1, using extended BNF notation, where $\langle N \rangle$ denotes a non-terminal, $\langle N \rangle$ ::= ... defines the non-terminal $\langle N \rangle$, S^* denotes zero or more occurrences of S, S^{**} denotes zero or more occurrences of S separated by commas (','), $S \mid T$ denotes the choice between S and T, $\lceil S \rceil$denotes optional S, **bold** text represents a keyword, and finally '...' denotes a terminal symbol.

⟨*Specification*⟩ ::= ⟨*Monitor*⟩*

⟨*Monitor*⟩ ::= **monitor** ⟨*Id*⟩ '{' ⟨*Transition*⟩* ⟨*State*⟩* '}'

⟨*State*⟩ ::= ⟨*Modifier*⟩* ⟨*Id*⟩ ⌈ (⟨*Id*⟩**) ⌉ ⌈ '{' ⟨*Transition*⟩* '}' ⌉

⟨*Modifier*⟩ ::= **init** | **hot** | **always**

⟨*Transition*⟩ ::= ⟨*Pattern*⟩ '::' ⟨*Condition*⟩ '→' ⟨*Action*⟩**

⟨*Pattern*⟩ ::= ⟨*Id*⟩ '('⟨*Id*⟩**')'

⟨*Condition*⟩ ::= ⟨*Condition*⟩ '∧' ⟨*Condition*⟩
  | ⟨*Condition*⟩ '∨' ⟨*Condition*⟩
  | '¬' ⟨*Condition*⟩
  | '('⟨*Condition*⟩')'
  | ⟨*Expression*⟩ ⟨*relop*⟩ ⟨*Expression*⟩
  | ⟨*Id*⟩ ⌈ '('⟨*Expression*⟩**')' ⌉

⟨*Action*⟩ ::= **ok**
  | **error**
  | ⟨*Id*⟩ ⌈ '('⟨*Expression*⟩**')' ⌉
  | **if** '(' ⟨*Condition*⟩ ')' **then** ⟨*Action*⟩ **else** ⟨*Action*⟩
  | ⟨*Modifier*⟩* '{' ⟨*Transition*⟩* '}'

Fig. 1. Syntax of DAUT

The syntax can briefly be explained as follows. A ⟨*Specification*⟩ consists of a sequence of monitors, each representing a data automaton. A ⟨*Monitor*⟩ has a name represented by an identifier ⟨*Id*⟩, and a body enclosed by curly brackets. The body contains a sequence of transitions and a sequence of states.

The transitions are short for an **init**ial **always** state containing these transitions. A $\langle \mathit{State} \rangle$ is prefixed with zero or more modifiers (**init**, **always**, or **hot**), has a name, and an optional list of (untyped) formal parameters, and an optional body of transitions leading out of the state. A $\langle \mathit{Transition} \rangle$ consists of a pattern that can match (or not) an incoming event, where already bound formal parameters must match the parameters of the event, followed by a condition. If the pattern matches and the condition evaluates to *true*, the action is executed, leaving the enclosing state unless it is an **always** state. A $\langle \mathit{Condition} \rangle$ conforms to the standard Boolean format including relations over values of expressions. The last alternative $\langle \mathit{Id} \rangle \lceil$ '(' $\langle \mathit{Expression} \rangle^{**}$')' $\rceil$ allows to write state expressions as conditions. A state expression of the form $id(exp_1, \ldots, exp_n)$ is true if there is a state active with parameters equal to the value of the expressions. This specifically allows to express past time properties. An $\langle \mathit{Action} \rangle$ is either **ok**, meaning the transition is taken without further action (a skip), **error**, which causes an error to be reported, the creation of a new state (target state), a conditional action, useful in practice, or the derived form of a modifier-prefixed block of transitions, avoiding to name the target state.

3.3 Semantics

Basic Concepts The semantics is defined as an operational semantics. We first define some basic concepts. We shall assume a set Id of identifiers and a set V of values. An environment $env \in Env = Id \overset{m}{\to} V$ is a finite mapping from identifiers to values. An event $e \in Event = Id \times V^*$ is a tuple consisting of an event name and a list of values. We shall write an event $(id, \langle v_1, \ldots, v_n \rangle)$ as: $id(v_1, \ldots, v_n)$. A trace $\sigma \in Trace = Event^*$ is a list of events. A state identifier id is associated with a sequence of formal parameters $id_1, \ldots, id_n$. A particular state $s \in State = id(v_1, \ldots, v_n)$, for $v_1, \ldots, v_n \in V$, represents an instantiation of the formal parameters. For such a state we can extract the environment with the following notation: $s.env$ of type Env, formed from the binding of the formal parameter ids to the values: $s.env = [id_1 \mapsto v_1, \ldots, id_n \mapsto v_n]$.

The semantics of each single monitor in a specification is a labeled transition system: $LTS = (Config, Event, \to, i, F)$. Here $Config \subseteq State$ is the set of all possible states (possibly infinite depending on the value domain). $Event$ is a set of parameterized events. $\to \subseteq Config \times (Event \times \mathbb{B}) \times Config$ is a transition relation, which defines transitions from a configuration to another as a result of an observed event, while "emitting" a Boolean flag being *false* iff. an error has been detected. $i \subseteq Config$ is the set of initial states, namely those with modifier **init** (these cannot have arguments). Finally, $F \subseteq Config$ is the set of final states $id(v_1, \ldots, v_n)$ where id is not declared with modifier **hot**.

The operational semantics to be presented defines how a given configuration con evolves to another configuration con' on the observation of an event e. In addition, since such a move can cause an **error** state to be entered, a Boolean flag, the *status flag*, will indicate whether such an error state has been entered in that particular transition. The result of transitions will hence be pairs of the form $(flag, con) \in Boolean \times Config$, also called *results* (res). Furthermore, we shall use the value $\bot$ to indicate that an evaluation has failed, for example if no transitions are taken out of a state. Consequently we need to be able to compose results, potentially being $\bot$, where combination of two proper results is again a result consisting of the conjunction of flags and union of configurations. We define two operators, $\oplus_\bot$ (for combining results that can potentially be $\bot$), and $\oplus$ (for combining proper results):

$$\begin{array}{l}
\mathit{res}_\bot \oplus_\bot \mathit{res}'_\bot = \\
\quad \textbf{case}\ (\mathit{res}_\bot, \mathit{res}'_\bot)\ \textbf{of} \\
\qquad (\bot, r) \Rightarrow r \\
\qquad (r, \bot) \Rightarrow r \\
\qquad (r_1, r_2) \Rightarrow r_1 \oplus r_2
\end{array}
\qquad\qquad
\begin{array}{l}
(b_1, \mathit{con}_1) \oplus (b_2, \mathit{con}_2) = \\
\quad (b_1 \wedge b_2, \mathit{con}_1 \cup \mathit{con}_2)
\end{array}$$

Note that this semantics will yield a status (*true* or *false*) for each observed event depending on whether an error state has been entered in that specific transition. This status does not reflect whether an error state has been entered *so far* from the beginning of the event stream. This form of non-monotonic result computation allows the result to switch for example from *false* in one step to *true* in the next, and is useful for online monitoring, where it is desirable to know whether the *current* event causes an error. The result across the trace can simply be computed as the conjunction over all emitted status flags. In case a 4-valued logic is desired [9], this is easily calculated on the basis of the contents of the current configuration (*false*: if **error** reached, and if not, *true*: if it contains no states, *possibly false*: if it contains at least one non-final state, and *possibly true*: if it contains only final states, one or more).

Operations Semantics The LTS denoted by a monitor is defined by the operational semantics presented in Figure 2. The semantics is defined for the kernel language not including (i) **always** states, (ii) transitions at the outermost level, and (iii) inlined states (all states have to be explicitly named).

Rule E (Evaluate) is the top-level rule, and reads as follows. A configuration con evolves ($\xrightarrow{e,b}$ below the line) to a configuration con' on observation of an event e, while emitting a status flag b, if ($\overset{e}{\hookrightarrow}$ above the line): $\mathit{con}, \mathit{con}$, where the second con functions as an iterator, yields the status b and configuration con'.

Rule E-ss$_1$ (Evaluate set of states) defines how the state iterator set is traversed ($\overset{e}{\hookrightarrow}$), here in the situation where the state iterator set has become empty. Rule E-ss$_2$ defines the evaluation in the case where the state iterator is not empty, by selecting a state s, which then is evaluated using $\overset{e}{\longmapsto}$, and then evaluating the remaining states ss recursively with $\overset{e}{\hookrightarrow}$.

Rule E-s$_1$ (Evaluate state) defines the evaluation of a state ($\overset{e}{\longmapsto}$) by evaluating ($\overset{e}{\Longrightarrow}$) its transitions $t.ts$ in the configuration and in the environment $t.env$ associated with the state. Here in the situation where none of the transitions fire, represented above the line by the result of $\overset{e}{\Longrightarrow}$ being the value $\bot$. Rule E-s$_2$ defines the evaluation in the situation where at least one of the transitions fire.

$$\boxed{\text{E}}\ \frac{con, con \overset{e}{\hookrightarrow} b, con'}{con \xrightarrow{e,b} con'}$$

$$\boxed{\text{E-ss}_1}\ \frac{}{con, \{\} \overset{e}{\hookrightarrow} (true, \{\})}$$

$$\boxed{\text{E-ss}_2}\ \frac{\begin{array}{c} con, s \overset{e}{\longmapsto} res \\ con, ss \overset{e}{\hookrightarrow} res' \end{array}}{con, s \cup ss \overset{e}{\hookrightarrow} res \oplus res'}$$

$$\boxed{\text{E-s}_1}\ \frac{con, s.env, s.ts \overset{e}{\Longrightarrow} \bot}{con, s \overset{e}{\longmapsto} true, \{s\}}$$

$$\boxed{\text{E-s}_2}\ \frac{con, s.env, s.ts \overset{e}{\Longrightarrow} res}{con, s \overset{e}{\longmapsto} res}$$

$$\boxed{\text{E-ts}_1}\ \frac{}{con, env, Nil \overset{e}{\Longrightarrow} \bot}$$

$$\boxed{\text{E-ts}_2}\ \frac{\begin{array}{c} con, env, t \overset{e}{\rightharpoondown} res_\bot \\ con, env, ts \overset{e}{\Longrightarrow} res'_\bot \end{array}}{con, env, \langle t \rangle ^\frown ts \overset{e}{\Longrightarrow} res_\bot \oplus_\bot res'_\bot}$$

$$\boxed{\text{E-t}_1}\ \frac{\begin{array}{c} t\ \textbf{is}\ \text{`}pat :: cond \to rhs' \\ [\![pat]\!]^P env\ e = \bot \end{array}}{con, env, t \overset{e}{\rightharpoondown} \bot}$$

$$\boxed{\text{E-t}_2}\ \frac{\begin{array}{c} t\ \textbf{is}\ \text{`}pat :: cond \to rhs' \\ [\![pat]\!]^P env\ e = env' \\ [\![cond]\!]^C con\ env' = false \end{array}}{con, env, t \overset{e}{\rightharpoondown} \bot}$$

$$\boxed{\text{E-t}_3}\ \frac{\begin{array}{c} t\ \textbf{is}\ \text{`}pat :: cond \to rhs' \\ [\![pat]\!]^P env\ e = env' \\ [\![cond]\!]^C con\ env' = true \\ [\![rhs]\!]^R con\ env' = res \end{array}}{con, env, t \overset{e}{\rightharpoondown} res}$$

Fig. 2. Operational semantics of DAUT

Rule E-ts$_1$ (Evaluate transitions) defines the evaluation ($\overset{e}{\Longrightarrow}$) of a list of transitions in the environment of the current state being evaluated. Here in the situation where this list is empty. In this case $\perp$ is returned to indicate that no transitions fired. Rule E-ts$_2$ defines the evaluation in the case where there is at least one transition t to be evaluated using $\overset{e}{\rightharpoondown}$ to a result, potentially $\perp$, and then evaluating the remaining transitions ts recursively with $\overset{e}{\Longrightarrow}$.

Finally, rule E-t$_1$ (Evaluate transition) defines the evaluation ($\overset{e}{\rightharpoondown}$) of a monitor transition t, which has the syntactic format: $pat :: cond \rightarrow rhs$, in the environment of the current state being evaluated. Recall that a transition consists of a pattern *pat* against which an observed event is matched. If successfully matched, the condition *cond* is evaluated, and if *true*, the right-hand side action *rhs* is executed. Rule E-t$_1$ defines the evaluation in the situation where the pattern *pat* does not match the event, either because the event names differ or because the actual parameters of the event do not match the assignments to the formal parameters defined by *env*. In this case $\perp$ is returned to indicate that no transitions fired. The semantics of patterns is defined by the evaluator $[\![\_]\!]^P$ in Figure 3. Rule E-t$_2$ defines the evaluation in the case where the pattern does match, but where the condition, evaluated by $[\![\_]\!]^C$ in Figure 3, evaluates to *false*. Rule E-t$_3$ defines the evaluation in the case where the pattern matches and the condition evaluates to *true*. In this case the right-hand side *rhs* is evaluated with $[\![\_]\!]^R$ in Figure 4.

Semantic Functions The semantic functions referenced in Figure 2 are defined in Figures 3 and 4. The semantics of expressions is the obvious one and is not spelled out. The semantics of conditions is also the obvious one, except for the semantics of state predicates of the form $id(exp_1, \ldots, exp_n)$: the expression arguments are evaluated and the result is *true* if and only if the resulting state is contained in the configuration *con*[1]. The semantics of the right-hand side, a comma separated list of actions of type $Action^{**}$, is obtained by evaluating each action to a result, and then 'and' ($\wedge$) the flags together and 'union' ($\cup$) the configurations together. The semantics of an action is a pair consisting of a status flag and a configuration, the flag being *false* if the action is **error**.

4 Optimization

The operational semantics presented in Figure 2 in the previous section is based on iterating through the configuration, a set of states, (rules E-ss$_1$ and E-ss$_2$), state by state, evaluating the event against each state. This is obviously costly. A better approach is to arrange the configuration as an indexed structure which makes it efficient for a given event to extract exactly those states that have

[1] The actually implemented semantics is a little more complicated by allowing selected arguments to the state predicate to be the "don't care" value '_', meaning that the search will not care about the values in these positions. However, the automaton concept is meaningful without this additional feature.

```
$[\![\_]\!]^P : Pattern \rightarrow Env \rightarrow Event \rightarrow Env_\perp$
$[\![pat]\!]^P env\ id(v_1, \ldots, v_n) =$
    case $pat$ of
        "_" $\Rightarrow env$ // don't care pattern matches all
        $id(id_1, \ldots, id_n) \Rightarrow$ // event names match
            let $env' = \{id_1 \mapsto v1, \ldots, id_n \mapsto v_n\}$ in
                if $(\forall id \in (dom(env) \cap dom(env')) \bullet env(id) = env'(id)))$
                    then $env \oplus env'$
                    else $\perp$ // bindings do not match
        $id'(\ldots)$ where $id \neq id' \Rightarrow \perp$ // event names do not match

$[\![\_]\!]^C : Cond \rightarrow Config \rightarrow Env \rightarrow \mathbb{B}$
$[\![cond]\!]^C con\ env =$
    case $cond$ of
        ...
        $id(exp_1, \ldots, exp_n) \Rightarrow id([\![exp_1]\!]env, \ldots, [\![exp_n]\!]env)) \in con$

$[\![\_]\!]^E : Exp \rightarrow Env \rightarrow \mathbb{B}$
...
```

Fig. 3. Semantics of patterns, conditions and expressions

transitions labeled with event patterns where the event name is the same, and where the formal parameters are bound to values (in the state's environment *env*) that match those in the corresponding positions in the incoming event. We here ignore "don't care" patterns, which match any event (the actually implemented algorithm deal with these as well). In the following we highlight some of the classes implementing such an optimization in the Scala programming language. First the top-level *Monitor* class:

```
class Monitor(automaton: Automaton) {
  val config = new Config(automaton)
  ...
  def verify(event: Event) {
    var statesToRem: Set[State] = {}
    var statesToAdd: Set[State] = {}
    for (state ∈ config.getStates(event)) { // efficient search for states
      val (rem, add) = execute(state, event)
      statesToRem ++= rem
      statesToAdd ++= add
    }
    statesToRem foreach config.removeState
```

```
$[\![\_]\!]^R : Action^{**} \rightarrow Config \rightarrow Env \rightarrow Result$
$[\![act_1, \ldots, act_n]\!]^R con\ env =$
    **let**
        $results = \{[\![act_i]\!] con\ env \mid i \in 1..n\}$
        $status = \bigwedge\{b \mid (b, con') \in results\}$
        $con'' = \bigcup\{con' \mid (b, con') \in results\}$
    **in**
        $(status, con'')$

$[\![\_]\!]^A : Action \rightarrow Config \rightarrow Env \rightarrow Result$
$[\![act]\!]^A con\ env =$
    **case** $act$ **of**
        **ok** $\Rightarrow (true, \{\})$
        **error** $\Rightarrow (false, \{\})$
        $id(exp_1, \ldots, exp_n) \Rightarrow (true, \{id([\![exp_1]\!]env, \ldots, [\![exp_n]\!]env)\})$
        **if** $(cond)$ **then** $act_1$ **else** $act_2 \Rightarrow$
            **if** $([\![cond]\!]con\ env)$**then** $[\![act_1]\!]con\ env$ **else** $[\![act_2]\!]con\ env$
```

Fig. 4. Semantics of transition right-hand sides

```
      statesToAdd foreach config.addState
    }
  }
```

The monitor (parameterized with the abstract syntax tree, *automaton*, representing the monitor) contains a instantiation of the *Configuration* class. The *verify* method is called for each event. It maintains two sets, one containing states to be removed from the configuration as a result of taking transitions, and one for containing states to be added. These sets are used to update the configuration at the end of the method. The essential part of this method is the expression: *config.getStates(event)*, which extracts only the relevant states for a given event.

The *Configuration* class is defined next. The core idea is to maintain two kinds of nodes: *state nodes* and *event nodes*. There is one state node for each named state. It contains at any point in time an index of all the states with that name, only distinguished by their parameters. Likewise, there is one event node for each transition, representing the event pattern on that transition. The event node is linked to the source state of the transition. The state nodes and event nodes are mapped to by their names. Since an event name can occur on several transitions, an event name is mapped to a list of event nodes:

```
  class Config(automaton: Automaton) {
```

```
  var stateNodes: Map[String, StateNode] = Map()
  var eventNodes: Map[String, List[EventNode]] = Map()
   ...
  def getStates(event: Event): Set[State] = {
    val (eventName, values) = event
    var result: Set[State] = Set()
    eventNodes.get(eventName) match {
      case None ⇒
      case Some(eventNodeList) ⇒
        for (eventNode ∈ eventNodeList) {
          result ++= eventNode.getRelevantStates(event)
        }
    }
    result
  }
}
```

The method *getStates* returns the set of states relevant for a given event. It does this by first looking up all the event nodes for that event (those with the same name), each corresponding to a particular transition, and for each of these it retrieves the relevant states in the corresponding state node. The details of how this works is given by the classes *EventNode* and *StateNode*, where sets and maps are mutable (updated point wise for efficiency reasons). The class *EventNode* is as follows:

```
case class EventNode(stateNode: StateNode,
  eventIds: List[Int], stateIds: List[String]) {
   ...
  def getRelevantStates(event: Event): Set[State] = {
    val (_, values) = event
    stateNode.get(
      stateIds,
      for (eventId ∈ eventIds) yield values(eventId)
    )
  }
}
```

An event node contains a reference to the state node it is connected to (the source state of the transition the event pattern occurs on), a list of parameter positions in the event that are relevant for the search of relevant states, and a list of the formal parameter names in the associated state these parameter positions correspond to. To calculate the states relevant for an event, the state node's *get* method is called with two arguments: the list of formal state parameters that are relevant, and the list of values they have in the observed event. The state node is as follows:

Listing 4. A monitor with a cancel option

```
monitor R3 {
  grant(t, r) → Granted(t,r)

  hot Granted(t,r) {
    release (t,r) → ok
    cancel(r) → ok
  }
}
```

```
case class StateNode(stateName: String, paramIdList: List[String]) {
  var index: Map[List[String], Map[List[Value], Set[State]]] = Map()
   ...
  def get(paramIdList: List[String], valueList: List[Value]): Set[State] =
  {
    index(paramIdList).get(valueList) match {
      case None ⇒ emptySet
      case Some(stateSet) ⇒ stateSet
    }
  }
}
```

A state node defines the name of the state, as well as its parameter identifier list (formal parameters). It contains an index, which maps a projection of the parameter identifiers to yet a map, which maps lists of values for these parameters to states which bind exactly those values to those parameters. A similar *put* method is defined, which inserts a state in the appropriate slot.

As an example, consider the monitor in Listing 4, where a depletable resource (can be assigned simultaneously to more than one task) either can get released by the task that it was granted to, or it can be canceled for all tasks that currently hold it. Suppose we observe the events $\langle grant(t_1, a), grant(t_2, a)\rangle$. Then the index for the state node for *Granted* will look as follows:

$$\begin{array}{lcl} \langle t, r\rangle & \mapsto & [\ \langle t_1, a\rangle \mapsto \{Granted(t_1, a)\},\ \langle t_2, a\rangle \mapsto \{Granted(t_2, a)\}\ ] \\ \langle r\rangle & \mapsto & [\ \langle a\rangle \mapsto \{Granted(t_1, a), Granted(t_2, a)\}\ ] \end{array}$$

5 Internal DSL

The internal DSL, DAUT^{int}, is defined as an API in SCALA. SCALA offers various features which can can make an API look and feel like a DSL. These include implicit functions, possibility to omit dots and parentheses in calls of methods on objects (although not used here), partial functions, pattern matching, and

Listing 5. Events (DAUT^{int})

```
trait Event
case class grant(task: String, resource: String) extends Event
case class release(task: String, resource: String) extends Event
```

case classes. DAUT^{int} is a variation of TRACECONTRACT, presented in [5], which explains in more detail how to use SCALA for defining a domain specific language for monitoring. TRACECONTRACT is a larger DSL, also including an embedding of linear temporal logic. However, it does in its pure form not support specification of past time properties (additional rule-based constructs had to be added to support this). DAUT^{int} is much simpler, just focusing on data automata, and it supports specification of past time properties by allowing transition conditions to refer to other parameterized states. This is achieved by defining states as **case** classes. A main advantage of an internal DSL is the ability to mix the DSL with code. Although not shown here, monitors can freely mix DSL constructs and programming constructs, such as variable declarations and assignment statements. For example, the right-hand side of a transition can include SCALA statements.

The complete implementation of DAUT^{int} is shown in Appendix A. As shown in Section 6, this simple DSL is surprisingly efficient compared to many other systems (except for MOP), which is interesting considering that it consists of very few lines of code. We shall not here explain the details, and refer to [5] for the general principles of implementing a similar DSL. Instead we shall illustrate what the DAUT monitors presented in Section 3 look like in DAUT^{int}. First we need to define the events of interest, see Listing 5. This is done by introducing the trait (similar to an abstract class) of events *Event* and then defining each type of event as a case class subclassing *Event*. In contrast to normal classes, case classes allow pattern matching over objects of the class, including its parameters. The monitors in listings 2 and 3 can be programmed as shown in Listing 6.

Note the similarity with the corresponding DAUT monitors. A monitor extends the *Monitor* class, which is parameterized with the event type. The method *whenever* takes a partial function as argument and creates an initial **always** state from it. A partial function can in SCALA be defined with a sequence of **case** statements using pattern matching over the events, defining the domain of the partial function. A state is modeled as a class that subclasses one of the pre-defined classes: *state*, *hot*, or *always*, defining respectively normal final states, non-final states, and final states with self-loops. The transitions in a state are declared with the *when* method which, just as the *whenever* method, takes a partial function representing the transitions as argument. Note that in order to enforce a pattern to match on values bound to an identifier, the identifier has to be quoted, as in ‘t‘. Finally, DAUT^{int} allows to combine monitors in a hierarchical manner, for the purpose of grouping monitors together. A monitor can be

Listing 6. Monitors (DAUT^{int})

```
class R1R2 extends Monitor[Event] {
  whenever {
    case grant(t, r) ⇒ Granted(t, r)
    case release(t, r) if !Granted(t, r) ⇒ error
  }

  case class Granted(t: String, r: String) extends hot {
    when {
      case release(‘t‘, ‘r‘) ⇒ ok
      case grant(_, ‘r‘) ⇒ error
    }
  }
}

class R1 extends Monitor[Event] {
  whenever {
    case grant(t, r) ⇒ hot {
      case release(‘t‘, ‘r‘) ⇒ ok
      case grant(_, ‘r‘) ⇒ error
    }
  }
}
```

Listing 7. Applying a monitor (DAUT^{int})

```
object Main {
  def main(args: Array[String]) {
    val obs = new R1R2

    obs.verify(grant("t1", "A"))
    obs.verify(release("t1", "A"))

    obs.end()
  }
}
```

applied as shown in Listing 7, creating an instance and subsequently submitting events to it.

6 Evaluation

This section describes the benchmarking performed to evaluate DAUT, the abstract operational semantics DAUT^{sos}, and the internal DSL, DAUT^{int}. The systems are evaluated against seven other RV systems, also evaluated in [15], which also explains the evaluation setup in details. The experiments focus on analysis of logs (offline analysis), since this has been the focus of our application of RV. 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. The systems compared are explained in [15]. All monitors check requirements R_1 and R_2 (page 4), formalized in DAUT in Listing 2 and in DAUT^{int} in Listing 6 (first monitor). Logs can abstractly be seen as sequences of events $grant(t, r)$ and $release(t, r)$, where t and r are integer values. The logs are represented as CSV files, and parsed with a CSV-parsing script.

Table 1. Results of tests 1-7. For each test is shown the *memory* of the test, length of the trace, and time taken to parse the log (subtracted in the following numbers). 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
memory	1	1	5	30	100	500	5000
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
DAUT	$\frac{49}{631}$	$\frac{84}{23:847}$	$\frac{86}{24:338}$	$\frac{89}{22:432}$	$\frac{90}{22:298}$	$\frac{86}{23:287}$	$\frac{80}{12:612}$
DAUT^{sos}	$\frac{102}{302}$	$\frac{192}{10:435}$	$\frac{79}{26:438}$	$\frac{24}{1:22:727}$	$\frac{8}{4:19:697}$	$\frac{2}{16:27:990}$	$\frac{0.18}{92:2:26}$
DAUT^{int}	$\frac{233}{133}$	$\frac{1715}{1:166}$	$\frac{770}{2:729}$	$\frac{373}{5:368}$	$\frac{195}{10:236}$	$\frac{54}{36:929}$	$\frac{5}{3:6:560}$
MOP	$\frac{595}{52}$	$\frac{1381}{1:448}$	$\frac{1559}{347}$	$\frac{1341}{1:491}$	$\frac{7143}{280}$	$\frac{7096}{282}$	$\frac{847}{1:193}$

The experiment consists of analyzing seven different logs: one log, numbered 1, generated from the Mars Curiosity rover during 99 (Mars) days of operation on Mars, together with six 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). 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: $\{1, 5, 30, 100, 500, 5000\}$. As an example, a memory value of 500 means that the log contains 500 $grant(t, r)$ events for all different values of (t, r), before any resources are released, resulting a memory of size 500, which then has to be indexed. The results are shown in Table 1.

The table shows that MOP outperforms all other systems by orders of magnitude. This fundamentally illustrates that the indexing approach used, although leading to limited expressiveness, has major advantages when it comes to efficiency. A more surprising result, however, is that the internal DSL DAUT^{int} outperforms all other tools, except MOP, for lower memory values. Furthermore, as a positive result, the optimized DAUT presented in this paper performs better than the other systems (again except MOP) for high memory values.

7 Conclusion

We have presented data automata, their syntax, semantics and efficient implementation. We consider data automata as providing a natural solution to the monitoring problem. The formalism and indexing algorithm have been motivated based on our experiences with rule-based systems, hence exploring the space between standard propositional automata and fully general rule-based systems. The algorithm is much less complex than the RETE algorithm, often used in rule-based systems, and appears to be more efficient. However, the implementation is not as efficient as the state-of-the-art RV system MOP. On the other hand, the notation is more expressive. We have shown an implementation in SCALA of an internal DSL which models data automata, but with the additional advantage of providing all of SCALA's features. The implementation is very simple, but moderately competitive wrt. efficiency.

References

1. 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.

2. 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.
3. H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-based runtime verification. In *VMCAI*, volume 2937 of *LNCS*, pages 44–57. Springer, 2004.
4. H. Barringer, A. Groce, K. Havelund, and M. Smith. Formal analysis of log files. *J. of Aerospace Computing, Information, and Communication*, 7(11):365–390, 2010.
5. 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.
6. 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.
7. D. A. Basin, F. Klaedtke, and S. Müller. Policy monitoring in first-order temporal logic. In T. Touili, B. Cook, and P. Jackson, editors, *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, Proceedings*, volume 6174 of *LNCS*, pages 1–18. Springer, 2010.
8. 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*, volume 8174 of *LNCS*, pages 59–75. Springer, 2013.
9. 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.
10. 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.
11. N. Decker, M. Leucker, and D. Thoma. Monitoring modulo theories. In E. Ábrahám and K. Havelund, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Grenoble, France, April 7-11, 2014. Proceedings*, volume 8413 of *LNCS*, pages 341–356. Springer, 2014.
12. C. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19:17–37, 1982.
13. 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.
14. S. Hallé and R. Villemaire. Runtime enforcement of web service message contracts with data. *IEEE Transactions on Services Computing*, 5(2):192–206, 2012.
15. K. Havelund. Rule-based runtime verification revisited. *Software Tools for Technology Transfer (STTT)*, April 2014. Published online.
16. 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.
17. 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.
18. 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.

A The Internal Scala DSL DAUT^{int}

```
class Monitor[E <: AnyRef] {
  val monitorName =
    this.getClass().getSimpleName()

  var monitors: List[Monitor[E]] = List()
  var states: Set[state] = Set()

  var statesToAdd: Set[state] = Set()
  var statesToRemove: Set[state] = Set()

  def monitor(monitors: Monitor[E]*) {
    this.monitors ++= monitors
  }

  type Transitions =
    PartialFunction[E, Set[state]]

  def noTransitions: Transitions =
  {
    case _ if false ⇒ null
  }

  class state {
    var transitions: Transitions =
      noTransitions

    def when(ts: Transitions) {
      this.transitions = ts
    }

    def apply(event: E): Option[Set[state]] =
      if (transitions.isDefinedAt(event))
        Some(transitions(event)) else None
  }

  class always extends state
  class hot extends state
  case object error extends state
  case object ok extends state

  def stateExists(
    pred: PartialFunction[state, Boolean]):
      Boolean =
  {
    states exists (pred orElse {
      case _ ⇒ false })
  }

  def state(ts: Transitions): state =
  {
    val e = new state
    e.when(ts)
    e
  }

  def always(ts: Transitions): state =
  {
    val e = new always
    e.when(ts)
    e
  }

  def hot(ts: Transitions): state =
  {
    val e = new hot
    e.when(ts)
    e
  }

  def error(msg: String): state =
  {
    println("\n*** " + msg + "\n")
    error
  }

  def whenever(ts: Transitions) {
    states += always(ts)
  }


  implicit def stateToBoolean(s: state): Boolean =
    states contains s

  implicit def unitToSet(u: Unit): Set[state] =
    Set(ok)

  implicit def stateToSet(s: state): Set[state] =
    Set(s)

  implicit def statePairToSet(
    ss: (state, state)): Set[state] =
      Set(ss._1, ss._2)

  implicit def stateTripleToSet(
    ss: (state, state, state)): Set[state] =
      Set(ss._1, ss._2, ss._3)


  def verify(event: E) {
    for (s ∈ states) {
      s(event) match {
        case None ⇒
        case Some(stateSet) ⇒
          if (stateSet contains error) {
            println("\n*** error!\n")
          } else {
            for (state ∈ stateSet) {
              if (state != ok) {
                statesToAdd += state
              }
            }
          }
          if (!s.isInstanceOf[always]) {
            statesToRemove += s
          }
      }
    }
    states −−= statesToRemove
    states ++= statesToAdd
    statesToAdd = Set()
    statesToRemove = Set()
    for (monitor ∈ monitors) {
      monitor.verify(event)
    }
  }

  def end() {
    val hotStates =
      states filter (_.isInstanceOf[hot])
    if (!hotStates.isEmpty) {
      println("*** hot states in " + monitorName)
      hotStates foreach println
    }
    for (monitor ∈ monitors) {
      monitor.end()
    }
  }
}
```