A Tutorial on Runtime Verification

Yliès FALCONE [a], Klaus HAVELUND [b,1] and Giles REGER [c,2]
[a] *University of Grenoble I (UJF), Laboratoire d'Informatique de Grenoble, France*
[b] *Jet Propulsion Laboratory, California Institute of Technology, USA*
[c] *University of Manchester, UK*

Abstract. This tutorial presents an overview of the field referred as to *runtime verification*. Runtime Verification is the study of algorithms, data structures, and tools focused on analyzing executions of systems. The performed analysis aims at improving the confidence in systems behavior, either by improving program understanding, or by checking conformance to specifications or algorithms. This chapter focuses specifically on checking execution traces against requirements formalized in terms of monitors. It is first shown on examples how such monitors can be written using aspect-oriented programming, exemplified by ASPECTJ. Subsequently four monitoring systems are illustrated on the same examples. The systems cover such formalisms as regular expressions, temporal logics, state machines, and rule-based programming, as well as the distinction between external and internal DSLs.

Keywords. Runtime verification, code instrumentation, temporal logic, regular expressions, state machines, rule-based programming, DSL.

1. Introduction

The rise of ubiquitous, embedded, safety-critical systems introduces a requirement of high-level confidence in their behavior. Numerous formal verification techniques provide mathematical guarantees on the reliability of software models and systems. However, reality is that the silver-bullet verification method does not exist yet. Proving millions of lines of code is not yet feasible. Usually, two kinds of verification techniques can be distinguished: static and dynamic analysis. Ideally, one would only have to apply *static analysis* to verify the system behavior prior to its execution. Examples of such techniques include model checking, theorem proving, and static analysis. However, these techniques suffer from shortcomings. Model checking suffers from the state-explosion problem: as the size of the system grows, the computational power required to verify systems grows beyond the capabilities of state-of-the-art computers. Theorem proving requires manual effort to carry out proofs (invariant discovery). Static code analysis scales well but the expressiveness of the properties that can be checked is limited, and many interesting behavioral properties remain out of scope of this technique. Techniques based on *dynamic analysis* inspect single executions of the system under scrutiny, and are thus incomplete (yield false negatives). However, this incompleteness allows to face-off the limitations

[1] Part of 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. The work of this author was furthermore supported by AFOSR Grant FA9550-09-1-0481 and NSF Grant CCF-0926190.

[2] The work of this author was supported by EPSRC Grant EP/P505208/1.

of static verification methods, and, renders them interesting complementary verification methods.

Runtime Verification (RV) is a dynamic analysis method aiming at checking whether a run of the system under scrutiny satisfies a given correctness property[3]. The inputs to an RV system are: (1) a system to be checked, and (2) a set of properties to be checked against the system execution. The properties can be expressed in a formal specification language (e.g., automata-based or logic-based formalism), or even as a program in a general-purpose programming language. A runtime verification process typically consists of the following three stages. First, from a property is generated a *monitor*, i.e., a decision procedure for the property. This step is often referred as to *monitor synthesis*. The monitor is capable of consuming *events* produced by a running system and emits *verdicts* according to the current satisfaction of the property based on the history of received events. Second, the system under scrutiny is *instrumented*. The purpose of this stage is to be able to generate the relevant events to be fed to the monitor. This step is often referred as to *system instrumentation*. Third, the system's execution is analyzed by the monitor. This analysis can occur either during the execution in a lock-step manner, or after the execution has finished assuming that events have been written to a log. This step is often referred as to *execution analysis*. Instrumenting the system and checking a property over the execution induces *overhead*. The overhead depends on many factors, including which program points are instrumented as well as data structures and algorithms used in the monitor. Generally, the more expressive an RV domain-specific formalism is, the more costly (in terms of overhead) the monitoring is. A goal is to find a balance between efficiency and expressiveness. Specification conciseness and elegance is an additional dimension concerned with how easy specifications are to write and read.

This tutorial presents an overview of the of field runtime verification by first providing a theoretic framework, then by illustrating how program instrumentation can be performed, and by subsequently illustrating five different notations for writing monitors, including the use of the general-purpose programming language JAVA, and four RV domain-specific languages. Compared to the original references introducing some of these systems, we take the liberty to sometimes present concepts in a different way. Section 2 introduces general mathematical preliminaries. Section 3 defines a unifying theoretic framework for runtime verification. Section 4 introduces the through-going example of a planetary rover platform written in JAVA, for which monitors shall be written. Section 5 explains how one can instrument the rover software using ASPECTJ, an aspect-oriented programming framework for JAVA, and Section 6 explains how to write monitors in the general-purpose programming language JAVA, using ASPECTJ for instrumentation. Sections 7-10 present the RV frameworks TRACEMATCHES [3], JAVAMOP [10,15], RULER-lite [4,8], and TRACECONTRACT [7]. Each system is illustrated by examples, as well as by presenting formal semantics and/or algorithm concepts.

Many important runtime verification frameworks have been left out of this tutorial due to space limitations, including for example [4,18,17,16,9,5,11]. The interested reader can also consult some previous attempts to overviewing runtime verification [13,14]. Moreover, the important question of monitorability, i.e., the issue of determining what properties can be monitored at runtime is not discussed. The interested reader can consult [11] for an extensive study.

[3] The field in fact is broader, and generally covers any technique used to analyze or explain program executions. Such techniques may for example include specification learning and trace visualization.

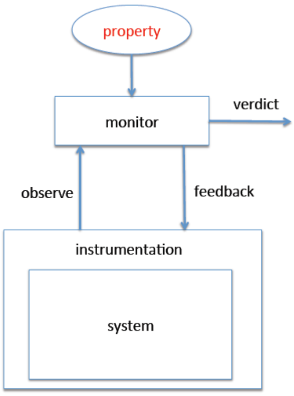

Figure 1. An overview of the runtime verification process.

2. Preliminaries

We introduce some mathematical notation that will be used in the chapter. We use $X \to Y$ and $X \rightharpoondown Y$ to denote sets of total and partial functions between sets X and Y, respectively. A map (a partial function with finite domain) where variable x_i is bound to value v_i, $1 \leq i \leq n$, is noted $[x_1 \mapsto v_1, \ldots, x_n \mapsto v_n]$. The empty map is denoted by $\bot$. The domain of a map $M : X \rightharpoondown Y$ (subset of X) is noted $\underline{dom}(M)$. We will often refer to maps as *bindings*. Given two maps A and B, the map override operator is defined as:

$$(A \dagger B)(x) = \begin{cases} B(x) & \text{if } x \in \underline{dom}(B), \\ A(x) & \text{if } x \notin \underline{dom}(B) \text{ and } x \in \underline{dom}(A), \\ \text{undefined otherwise.} \end{cases}$$

Maps are partially ordered: M_1 is a submap of M_2, noted $M_1 \sqsubseteq M_2$, if and only if:

$$\underline{dom}(M_1) \subseteq \underline{dom}(M_2) \wedge \forall x \in \underline{dom}(M_1) : M_1(x) = M_2(x).$$

3. A Unified Setting

This section presents a unified setting for runtime verification for use in this chapter.

3.1. The Runtime Verification Phases

Before giving formal definitions of runtime verification concepts, we give a brief overview of the stages involved in monitoring a system, captured in Figure 1:

1. *Monitor creation*: A monitor is created, potentially from a formal property.
2. *Instrumentation*: The system is instrumented to generate events for the monitor.
3. *Execution*: The system is executed, generating events for the monitor.
4. *Responses*:
   - The monitor produces for each consumed event a *verdict* indicating the status of the property depending on the event sequence seen so far.
   - The monitor sends *feedback* to the system - this may give further information to the system, so that more specific corrective actions can be taken.

Each of these stages are discussed in further detail in the rest of this section.

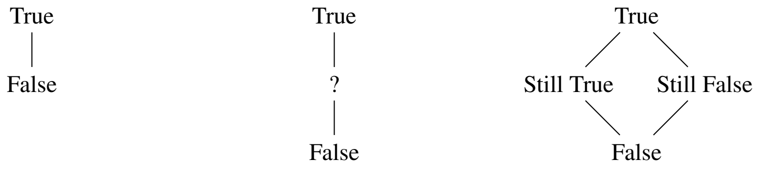

Figure 2. Possible verdict domains.

3.2. Events, Traces and Properties

The behaviour of a system for a given execution can be captured as a finite sequence of *events* describing selected actions taken by the system, or selected states of the system. In general, an event consists of a name and sequence of data values. We assume a set of data values $\mathcal{V}$, which will depend on the context of the runtime verification system.

Definition 1 (Events and traces) *An* event *is a pair $\langle e, \overline{v} \rangle$ where e is an event name and $\overline{v}$ is a finite sequence of values in $\mathcal{V}$. From here onwards we will write $e(\overline{v})$ for the event $\langle e, \overline{v} \rangle$. Let Event be the set of all events. A* trace *is a finite sequence of events. Let ϵ be the empty trace. Let $Trace(\mathcal{A})$ be the set of all traces over a given set of events $\mathcal{A}$.*

In runtime verification it is common to consider only *finite* traces, as monitoring a continuously running systems can be considered as analyzing finite sequence snapshots. All runs of a system can be described by a (possibly infinite) set of finite traces. In general, a property is a syntactic object belonging to some particular logic or language, however in this section we are interested in the abstract denotation (semantics) of the property. Here we model properties as functions from traces to a given verdict domain.

Definition 2 (Property) *Let $Prop(\mathcal{A}, \mathbb{D}) = Trace(\mathcal{A}) \to \mathbb{D}$ be the set of all properties from traces over the set of events $\mathcal{A}$ to the verdict domain $\mathbb{D}$.*

Different runtime verification systems use different formalisms or logics, and thus can express different sets of properties. We say that one system is more *expressive* than another if the properties the first can express properly contain those the second can express.

3.3. Verdicts and Feedback

One advantage of verifying a system at runtime is that the system can take corrective action if a property is violated, using the results of verification to steer itself towards more desirable behaviors. To achieve this, monitors communicate with the system through *verdicts* and *feedback*. Verdicts give the status of the monitored system with respect to a property and feedback provides additional information to the monitored system.

A runtime verification system will return a verdict from some verdict domain $\mathbb{D}$ after processing each event, or the whole trace. In the most simple case this verdict domain would be $\mathbb{B}$ - either true or false. However, many runtime verification systems use verdict domains containing three or more values to give a more fine-grained result as illustrated in Figure 2 and described elsewhere [9]. The first step is to introduce a third value to indicate that the system has not succeeded or failed yet. An alternative is to extend $\mathbb{B}$ with two new values, which we call *still true* and *still false* here. These indicate that the monitored system is currently succeeding/failing but its status may change in the future.

The area of feedback has not been widely explored within the context of runtime verification. There are relations to automatic program repair, program steering, fault protection, self-healing systems, planning, and runtime enforcement to mention just a few topics.

3.4. System Instrumentation

So far we have discussed traces, verdicts and feedback but have not considered how these are communicated from and to the monitored system. In the runtime verification process this is handled via *instrumentation*. It is a matter of discussion whether this instrumentation forms part of the runtime verification system itself or not - some tools intermix scripts defining instrumentation and properties, whereas others keep them firmly separate. In the theoretical presentation given in this chapter we will separate the two, however instrumentation is a key part of the runtime verification process.

The instrumentation approach used depends on the system being instrumented. A majority of the tools discussed in this chapter target JAVA programs and therefore ASPECTJ, an aspect-oriented programming language for JAVA, is used. ASPECTJ can be used directly as a runtime monitoring system, as discussed in Section 6. At the very least, an *interface* should be defined between a runtime verification tool and monitored system.

3.5. Runtime Verification Systems

A runtime verification system defines a method for synthesizing *monitors* - objects which consume events and produce verdicts. We do not include the notion of feedback - assuming this is handled by some additional external system.

Definition 3 (Monitor) *Let $\langle \mathbb{D}, \mathcal{A}, Q, q_0, \Delta, \Gamma \rangle$ be a monitor where $\mathbb{D}$ is a (possibly infinite) verdict domain, $\mathcal{A}$ is a (possibly infinite) set of events, Q is a (possibly infinite) set of monitor states, q_0 is the initial monitor state, $\Delta : (Q \times \mathcal{A}) \to Q$ is a transition function and $\Gamma : Q \to \mathbb{D}$ is the verdict function. Let $Monitor(\mathbb{D}, \mathcal{A})$ be the set of all monitors with verdict domain $\mathbb{D}$ for events in $\mathcal{A}$.*

At runtime we store a monitor's current state and then process each event by computing the next monitor state using Δ and finding the verdict to return using Γ. We restrict Δ to be deterministic to make it possible to know the exact next state whilst monitoring at runtime - methods such as backtracking are not usually seen in runtime verification. We can emulate non-determinism by defining a monitor state to be a set.

We let the set of events and verdict domain both be possibly infinite. For events this makes sense as their parameters may be taken from an infinite data domain. However the notion of an infinite verdict domain is not instantly straight-forward - previously we mentioned only small fixed domains of truth values. But there has been some work on monitoring logics that return statistics or probabilities as verdicts, for example [12].

A runtime verification system conceptually captures a verdict domain, a set of possible events, a set of possible properties, and a method for generating monitors from properties.

Definition 4 (Runtime verification system) *A* runtime verification system *is a tuple $\langle \mathbb{D}, \mathcal{A}, \mathcal{P}, \mathcal{G} \rangle$ where $\mathbb{D}$ is a (possibly infinite) verdict domain, $\mathcal{A}$ is a (possibly infinite) set of* possible *events, $\mathcal{P} \subseteq \bigcup_{\mathcal{B} \subseteq \mathcal{A}} Prop(\mathcal{B}, \mathbb{D})$ is a (possibly infinite) set of properties, and $\mathcal{G} : \mathcal{P} \to Monitor(\mathbb{D}, \mathcal{A})$ is a monitor-generation function.*

A runtime verification system will typically provide a *Domain-Specific Language* (DSL) for describing properties, the implementation of which will implicitly capture the monitor-generation function $\mathcal{G}$. The different approaches for defining a DSL may be categorized as follows [6]:

1. *External.* The DSL is a stand-alone language.
   (a) *Compilation.* A property is parsed and translated into a program, representing a monitor for that property, which is then *executed*.
   (b) *Interpretation.* A property is parsed and translated into a data structure, representing a monitor for that property, which is then *interpreted*.
2. *Internal.* The DSL is embedded in an existing General-Purpose Language (GPL) and is therefore directly executable.
   (a) *Shallow.* A property can make use of features in the GPL.
   (b) *Deep.* A property is represented as a data structure in the GPL.

3.5.1. Getting Bindings from Events

Many runtime verification systems use so-called *symbolic* events in specifications, which are then matched against events in the trace to produce *bindings*. We quickly introduce some general concepts which will be used in later sections.

Definition 5 (Symbolic event) *A* symbolic event *is a pair* $\langle e, \overline{s}\rangle$ *where* $e \in \Sigma$ *is an event name and* $\overline{s} \in (\mathcal{X} \cup \mathcal{V})^*$ *is a sequence of variables or values.*

We will write symbolic events in the same way as we write events i.e., $e(\overline{s})$. A symbolic event in a specification is matched against an event in a trace to produce a binding.

Definition 6 (Matching) *Given a symbolic event* $\mathbf{a} = e(s_1, \ldots s_n)$ *and an event* $\mathbf{b} = e(v_1, \ldots, v_n)$ *then* $\texttt{matches}(\mathbf{a}, \mathbf{b})$ *is true if and only if there exists a binding* θ *such that if* s_i *is a variable then* $\theta(s_i) = v_i$*, otherwise* $s_i = v_i$*, for* $1 \leq i \leq n$*. Let* $\texttt{match}(\mathbf{a}, \mathbf{b})$ *denote the smallest such binding (with respect to* $\sqsubseteq$*) if it exists and undefined otherwise.*

3.6. Classifying Runtime Verification Approaches

We do not attempt to present a classification of runtime verification systems but for completeness we mention some dimensions which have been used previously:

- When monitoring occurs. This can either be *online*, monitoring occurs whilst the system is running, or *offline*, monitoring occurs after the system has run - applied to a log file.
- Where the monitor is placed. This can either be *inline*, the monitor is included in the code of the system, or *outline*, the monitor exists as an external entity.
- When are verdicts returned. This can either be *violation*, when the monitored property becomes false, or *validation*, when the monitored property becomes true.

3.7. A Quick Note on Usability

As runtime verification systems are intended as tools they must be usable. Here we briefly discuss some aspects of runtime verification systems that will impact this. We believe that there are three main aspects important to runtime verification systems.

Efficiency. For a tool to be practical it must work on large traces - not only should a monitoring algorithm be fast, it should also scale with the trace size. Efficiency requirements depend on context - are we monitoring a running program or a log file produced by one. In the former case it is likely that we would want an *incremental* monitoring algorithm, that introduces minimal overhead with respect to memory and running time. In the latter case it is likely that we are interested mostly in running time. Overhead depends on the monitoring algorithm used, the complexity of the specification and the makeup of the trace.

Expressiveness. What specifications can we write? This is defined by the specification language of a particular system. Comparing the expressiveness of these languages is not straightforward. Whilst there are well known results relating to language theory (i.e. context-free languages are more expressive than regular languages) there are many aspects here which complicate this issue, for example how events are specified.

Elegance. How easy is it to write a specification. The syntax and operations of a particular specification language may make certain specifications easier or harder to write and/or read. Conciseness is an important measurable aspect of this.

4. A Rover System and its Requirements

In this section we briefly introduce a simple rover example, which will be referred to throughout the paper, and for which we will formulate requirements and write monitors in the various notations. The assumed programming language is Java. The example concerns a planetary rover platform. The rover runs *tasks* that use *resources*. A *scheduler* manages the task execution and resource allocation. A task may handle an instrument on board of the rover, such as a camera. Tasks are commanded by the scheduler and report back whether they succeed or fail. In addition, the scheduler manages resources (the antenna for example), which can be requested by tasks, and granted if available and are not in conflict with other granted resources. Tasks must eventually cancel (hand back) resources they have been granted. Conflicting resources can have different priorities. A request for a resource with a higher priority than an already granted resource causes the lower priority resource to be rescinded (the task owning it is asked to cancel it). We shall assume a class *Res* representing various resources. The following class models a task:

```
public class Task {
  public int id;
  void sendCommand(String name, int number) {...}
  void sendGrant(Res res) {...}
  void sendRescind(Res res) {...}
}
```

A task can be commanded by the scheduler to perform a job by a call of the *sendCommand* method on the task. A command is identified by a name and a running job number. A task can be granted a resource, or asked to cancel it again (the resource is rescinded). The following class models the scheduler:

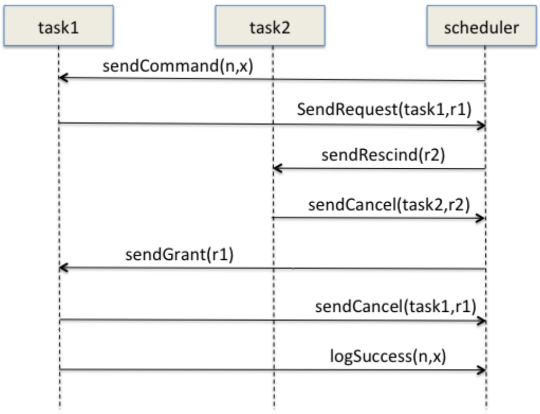

Figure 3. Message sequence diagram illustrating a possible interaction between scheduler and two tasks.

```
public class Scheduler {
  void logSuccess(String name, int number) {...}
  void logFailure(String name, int number) {...}
  void setConflict(Res res1, Res res2) {...}
  void setPriority(Res res1, Res res2) {...}
  void sendRequest(Task task, Res res) {...}
  void sendCancel(Task task, Res res) {...}
  void sleep() {...}
}
```

Tasks report success or failure back to the scheduler, as well as request and cancel resources. Two resources can be declared as being in conflict with each other. Furthermore, one resource r_1 can be declared as having higher priority than another conflicting resource r_2. Finally, the scheduler can be put to sleep during the dark nights on the foreign planet. The message sequence diagram in Figure 3 illustrates a possible sequence of communication events: the scheduler commands a task to perform a job, the task then requests a resource r_1 which has higher priority than a conflicting resource r_2 already granted to another task, the other task is therefore asked to rescind r_2, which it does, after which the resource r_1 is granted to the first task, which subsequently after done job then hands back the resource r_1, and reports success. We will consider four properties for which we will write monitors in the various notations. These are:

1. Exactly One Success: *An issued command, identified by a name and a running job number, shall eventually succeed exactly once, and must not fail before then.*

2. Respect Conflicts: *A resource should not be granted if it is in conflict with another already granted resource.*

3. Respect Priorities: *If a resource r_1 is requested that has a priority higher than an already granted resource r_2, and there is not another granted resource with yet higher priority than r_1, then r_2 should be rescinded before r_1 is granted.*

4. Release Resource: *If a resource is granted during the execution of a command then it should be released before the command succeeds.*

We will not write monitors for all four properties in each notation, but we need a range of examples due to the varying expressiveness of the systems presented.

5. Instrumentation with ASPECTJ

ASPECTJ is an implementation of the *aspect-oriented programming* (AOP) paradigm for the JAVA programming language. We present general principles and motivations of AOP and provide a brief overview of ASPECTJ's main features needed for our RV purposes.

5.1. Aspect-Oriented Programming

The main motivation for AOP arises from the following observation: in software systems, some concerns (e.g., logging, policy enforcement, security management, profiling, trace visualization, and verification) are typically not implemented in a modular fashion. Such concerns are referred as to *cross-cutting* concerns because they affect several modules of the system. Cross-cutting code suffers from:

- *code scattering*: similar code is distributed throughout many modules;
- *code tangling*: two or more concerns are implemented in the same module.

AOP aims at solving these issues by allowing cross-cutting concerns to be defined in a special kind of cross-cutting modules called *aspects*. An "aspectized" software system usually comprises of a main application, implementing the basic functionalities, written using traditional modules (classes in the case of JAVA), and then a set of aspects implementing cross-cutting concerns. Code instrumentation is a cross-cutting concern that can be implemented as aspects. These aspects *observe* the execution of events of interest in the target system. Observations of these events trigger the execution of code that *updates* the state of monitors.

The main concepts of AOP are those of: *join points*, *pointcuts*, *advice*, and *aspects*. A *join point* is a well-identified point in the execution of a program. For instance, it can be a method call or the access to an attribute. A *pointcut* is used to select some join points and access their execution context. For instance, a pointcut can select the calls of methods used to perform a given operation of interest. An *advice* consists of a pointcut and a body of code, i.e., the code to be executed when join points matching the pointcut are reached during the execution of the program. Finally, an *aspect* is defined as a collection of pointcuts and advice. Aspects can also contain fields and methods, similar to a class.

5.2. Pointcuts

The general syntax of a pointcut is (ignoring access modifiers):

```
pointcut pointcut_name ([arguments]) : pointcut_expression
```

where *pointcut_name* is the pointcut name being defined, and *pointcut_expression* is the actual pointcut that the name will denote, and which can take various forms depending on the kinds of join points that must be selected by this named pointcut. When observing system execution, we are for example often interested in method calls. ASPECTJ also allows us to observe many other events in programs such as class initialization, read/write accesses to attributes, exception handler executions, etc. Let us consider an example of a pointcut:

```
pointcut com(String name, int number, Task task) :
   call(void Task.sendCommand(String, int))
  && args(name,number) && target(task);
```

This pointcut, named *com*, has three parameters: *name*, *number*, and *task*. It selects join points which are calls to methods matching the signature **void** Task.sendCommand (String, **int**), i.e., calls to the (unique) method *sendCommand* of the class *Task* that has parameters of type String and **int**, and does not return a value. The directive **args**(name,number) indicates that the parameter runtime values should be bound to *name* and *number*, allowing the values of parameters to be accessed at runtime. The directive **target**(task) binds the object on which the selected methods are called to *task*. Conjunction and disjunction of pointcuts is achieved using the operators && and ||, respectively.

5.3. Advice and Aspects

Recall that an advice consists of a pointcut and a code snippet that is executed when a join point matching the pointcut is reached. The general syntax of an advice is:

```
advice_kind([arguments]) : pointcut_expression {
   advice_body
}
```

where *advice_kind* indicates when the advice body, a statement, should execute when the pointcut is matched by a join point. There are three kinds of advice: a *before* (resp. *after*) advice executes the advice body before (resp. after) the join point; while with an *around* advice, the advice body replaces the join point. In this later case, it is possible to use a **proceed** statement in the advice body to execute the selected join point (hence the term *around*). Arguments of the advice (in *advice_kind([arguments])*) should be those selected by the pointcut. These parameters are used by the advice body.

An aspect is a collection of pointcuts, advice, and JAVA class body declarations. The following aspect checks that command job numbers increase by one. It consists of the *com* pointcut and a before-advice that executes before any call to the *Task.sendCommand* method. It also declares and uses a variable *current* for keeping track of the last command number seen. Pointcut parameters are retrieved by the advice and used in the code body to check the condition, as an assertion, followed by an update of the variable *current*.

```
public aspect IncreasingNumbers {
 int current = 0;

 pointcut com(String name, int number, Task task) :
  call(void Task.sendCommand(String, int))
  && args(name,number) && target(task);

 before(String name, int number, Task task) : com(name, number, task) {
   assert(number == current + 1) :
     "wrong job number for " + name + " (" + task.id + ")";
   current = number;
 }
}
```

6. Designing Monitors with ASPECTJ and JAVA

This section illustrates the considerable amount of effort required to write monitors for the properties *Exactly One Success* and *Respect Priorities* in JAVA, while instrumenting the code with ASPECTJ.

6.1. Defining Pointcuts

There are many approaches and styles possible when writing monitors in an aspect-oriented framework such as ASPECTJ. One approach is for each property to be modeled as a single self-contained aspect, including pointcut definitions. This may, however, require repetition of pointcuts across aspects. A second approach is to have aspects be concerned with instrumentation only, and let them communicate with separate *property objects*, which then check the properties. Each such property object can either have a method declared for each relevant event, or it can have a single *submit(Event event)* method, where different events are defined as sub-classes of an abstract *Event* class. In the following we have chosen a third approach, namely to first define an abstract aspect which only defines all the pointcuts of interest. One can imagine this as a result of an individual activity of identifying all events of interest in the rover code. Each property is then defined as an aspect that *extends* this aspect, referring to a subset of these pointcuts, depending on the property. The following aspect defines the pointcuts of interest:

```
abstract aspect Pointcuts extends MonitorUtils {
  pointcut com(String name, int number, Task task) :
    call(void Task.sendCommand(String, int))
   && args(name,number) && target(task);

  pointcut suc(String name, int number) : ...;
  pointcut fail(String name, int number) : ...;
  pointcut conflict(Res res1, Res res2) : ...;
  pointcut priority(Res res1, Res res2) : ...;
  pointcut request(Task task, Res res) : ...;
  pointcut grant(Res res, Task task) : ...;
  pointcut cancel(Task task, Res res) : ...;
  pointcut rescind(Res res, Task task) : ...;
  pointcut end() : call(void Scheduler.sleep());
}
```

6.2. Specifying Exactly One Success

The *Exactly One Success* property is modeled by the following aspect. It models a state machine with four states: an initial state, and the states *issued* (the command has been issued), *succeeded* (the command has succeeded), and *failed* (either the command failed, or succeeded more than once). The class *Command*, not shown, models a command identified by a name and a number. The field *states* maps each command to a monitor for this particular command; more specifically: to the state the monitor is in. Any command identity not mapped to a state is considered in the initial state. Each advice instruments one of the events (method calls) relevant to this property and performs an action based on the active state for the particular command.

```
aspect ExactlyOneSuccess extends Pointcuts {
  enum State {issued, succeeded, failed};
  Map<Commmand, State> states = new HashMap<Commmand, State>();

  after(String name, int number) : com(name,number,*) {
    Commmand command = new Commmand(name, number);
    State current = states.get(command);
    if (current == null) states.put(command, State.issued);
  }

  after(String name, int number) : suc(name,number) {
    Commmand command = new Commmand(name, number);
    State current = states.get(command);
    switch (current) {
      case issued: states.put(command, State.succeeded); break;
      case succeeded:
        error(”command succeeded more than once ” + command);
        states.put(command, State.failed);
      default:
    }
  }

  after(String name, int number) : fail(name,number) {
    Commmand command = new Commmand(name, number);
    State current = states.get(command);
    switch (current) {
      case issued:
        error(”command failed: ” + command);
        states.put(command, State.failed);
      default:
    }
  }

  after() : end() {
    for (Commmand command : states.keySet()) {
      State current = states.get(command);
      switch (current) {
        case issued: error(”command not succeeded ” + command);
        default:
      }
    }
  }
}
```

A *com(name,number,task)* event causes an entry to be inserted in *states* pointing to the *issued* state (if in the initial state before). Note that in the pointcut *com(name,number,*)* we ignore the task, indicated by a ∗. A *suc(name,number)* event will make a transition from *issued* to *succeeded*, or from *succeeded* to *failure*. The *end()* event causes an error to be issued for all commands that are in the *issued* state, which is considered as a non-final state, modeling that they have not yet succeeded.

6.3. Specifying Respect Priorities

The *Respect Priorities* property is modeled by the following aspect using three fields as internal state. The field *priorities* maps a resource to the set of resources that have lower priority, and is updated at occurrence of a *priority(res1,res2)* event. The *allocated* field is a set of resources currently granted to tasks, and is updated at the occurrence of *grant(res,task)* and *cancel(task,res)* events. The field *toRescind* maps a resource r to the set of resources that need to be rescinded before r can be granted, and is updated at the occurrence of *request(task, res)* and *rescind(res,task)* events, and queried at the occurrence of a *grant(res,task)* event.

```
public aspect RespectPriorities extends Pointcuts {
  MapToSet<Res, Res> priorities = new MapToSet<Res, Res>();
  Set<Res> allocated = new HashSet<Res>();
  MapToSet<Res, Res> toRescind = new MapToSet<Res, Res>();

  after(Res res1, Res res2) : priority(res1,res2) {
    priorities.getSet(res1).add(res2);
  }

  after(Res res) : request(*, res) {
    Set<Res> lower = new HashSet<Res>(priorities.getSet(res));
    lower.retainAll(allocated);
    if (!lower.isEmpty()) {
      boolean higherExists = false;
      for (Res r : allocated) {
        if (priorities.getSet(r).contains(res)) {
          higherExists = true; break;
        }
      }
      if (!higherExists) {
        Set<Res> rescinds = toRescind.getSet(res);
        rescinds.addAll(lower);
      }
    }
  }

  after(Res res) : grant(res,*) {
    Set<Res> rescinds = toRescind.getSet(res);
    if (!rescinds.isEmpty()) error(res + ” granted before rescinds”);
    allocated.add(res);
  }

  after(Res res) : cancel(*,res) {allocated.remove(res);}

  after(Res res) : rescind(res,*) {
    for (Res r : toRescind.keySet()) toRescind.getSet(r).remove(res);
  }
}
```

The class *MapToSet⟨A, B⟩* models a mapping from *A* to *Set⟨B⟩*, and defines a method *getSet(A x)* that in case x is not defined updates the map with the empty set, which is then returned.

7. TRACEMATCHES

7.1. Overview

In the previous section we saw how ASPECTJ could be used to write monitors using pointcuts denoting single points in the execution. TRACEMATCHES [3] is an extension of the ASPECTJ language, implemented in the abc compiler [2], giving users the ability to write monitors involving the history of computation. Regular expressions are given over pointcuts with *free variables* in events used to capture parameters.

7.2. Introductory Example

Let us use TRACEMATCHES to specify our running *Exactly One Success* requirement. We use the pointcuts introduced in Section 6 to capture the events of interest, here referred to as *symbols*, and then specify a regular expression over these:

```
public aspect ExactlyOneSuccess extends Pointcuts{
  tracematch(String name, Integer number)
  {
    sym com after: com(name,number,*);
    sym suc after: suc(name,number);
    sym fail after: fail(name,number);

    com (fail | suc suc)
    {
        error("Command "+name+" with id "+number+" failed.");
     }
  }
}
```

The regular expression matches if any *suffix* of the trace matches the expression - we say TRACEMATCHES is *suffix-matching*. Here this is whenever a command is followed by a failure or two successes. This specification is parameterised with $name$ and $number$ - we can consider these variables universally quantified.

Let us consider what suffix-matching means by considering an alternative example. We might wish to specify that grant and cancel should alternative for a particular resource. To specify this in TRACEMATCHES we need to consider what the end of a bad trace would look like - in this case this is where we see a grant next to a grant or a cancel next to a cancel. Assuming grant and cancel symbols relating to the relevant pointcuts we would specify this property using the following regular expression:

```
    grant grant | cancel cancel
```

A trace matches this tracematch if the subtrace relevant to a particular resource has a suffix which matches the regular expression, for example the trace

`grant`(wheels). `cancel`(wheels) . `grant`(camera) . `cancel`(camera). `cancel`(wheels)

would match on the last event, and would therefore violate the property.

7.3. Semantics

In this section we will briefly present both a declarative and operational semantics of TRACEMATCHES.

7.3.1. Core Concepts

We begin by formalizing the components of a tracematch. The **sym** keyword introduces symbolic events, note that here we deviate from the presentation given in [3]. Let P be the regular expression in the tracematch, defined over the set of symbolic events S.

Given an event $e(v_1, \ldots, v_k)$ and a symbolic event $e(n_1, \ldots, n_k)$ we can compute the constraint: $n_1 = v_1 \wedge \ldots \wedge n_k = v_k$. This constraint represents the fact that an event has been observed with these bindings. Since many events can be observed over time, in general there is a need to operate with disjunctions of such constraints. Hence the general form of a constraint is a disjunction: $C_1 \vee \ldots \vee C_m$, where each C_i is a conjunction of the form: $E_1 \wedge \ldots \wedge E_n$ of (in)equalities of the form $n = v$ or $n \neq v$ for some parameter n and value v. Let $Constraint$ be the type of such constraints. Note that a constraint captures a set of bindings (solutions), the set of bindings satisfying the constraint.

From the set of symbolic events S we can extract a function $M : Matcher$ of the type $Matcher = SymbolicEvent \rightarrow (Event \rightarrow Constraint)$, that produces a constraint from a symbolic event and an event. The generated constraint captures the binding of the parameters of the symbol to the argument values in the event. For example, with symbolic events $\texttt{com}(n, x)$ and $\texttt{suc}(n, x)$, we get a M satisfying:

$$M(\texttt{com}(n, x))(\texttt{com}(\mathrm{A},1)) = (n = \mathrm{A} \wedge x = 1) \tag{1}$$

$$M(\texttt{com}(n, x))(\texttt{suc}(\mathrm{A},1)) = false \tag{2}$$

Note the relation between this matcher and the match function defined in Section 3.5.1 - in both cases a structure to represent bindings is constructed from a symbolic event and an event. A tracematch can be considered as denoting a tuple $\langle P, M, C \rangle$ consisting of a regular expression P, a matcher M, and a code block C to be executed when P matches an execution trace.

We can now define the constraint resulting from matching a sequence of symbolic events (a word over the regular expression P) against a trace of events. Given a matcher M, a sequence of symbolic events $a_1 \ldots a_m$ matched against a trace of events $e_1 \ldots e_n$ results in the constraint defined by the following function:

$$\mathsf{match}_M([a_1 \ldots a_m], [e_1 \ldots e_n]) = \begin{cases} \bigwedge_i M(a_i)(e_i) & \text{if } m = n \\ \mathit{false} & \text{otherwise} \end{cases}$$

Given a particular binding, the concept of a matcher can be further constrained to respect that binding. Observe first that a binding (binding all of the tracematch's parameters) can be applied to a constraint to get a truth value (itself a constraint) e.g.,

$$\begin{aligned}[\mathrm{n} \mapsto \mathrm{A}, \mathrm{x} \mapsto 1]\,((\mathrm{n{=}A} \wedge \mathrm{x{=}1}) \vee (\mathrm{n{=}B} \wedge \mathrm{x{=}2})) &= (\mathrm{A{=}A} \wedge 1{=}1) \vee (\mathrm{A{=}B} \wedge 1{=}2) \\ &= \mathit{true} \vee \mathit{false} \\ &= \mathit{true}\end{aligned}$$

We can define a function that for a given constraint C yields the solutions (bindings) to the constraint:

$$\mathsf{solutions}(C) = \{\theta \mid \theta(C) = \mathit{true}\}$$

Given a matcher M, and a binding θ, we can now define the binding-constrained matcher: $M^{\theta} : \mathit{Matcher} = \lambda a.\lambda e.\theta(M(a)(e))$.

7.3.2. *Declarative Semantics*

The objective of a declarative semantics of TRACEMATCHES is to determine for a given trace of events τ (observed so far) what bindings of the tracematch's parameters, if any, make this trace satisfy the regular expression P. We start by defining which events are relevant to a binding θ, given a matcher M:

$$\mathsf{relevant}_M(\theta) = \{e \mid \exists a \in S : \theta(M(a)(e)) = \mathit{true}\}$$

We can filter irrelevant events out of a trace, given a matcher M and a binding θ:

$$\epsilon \downharpoonright_{\theta}^{M} = \epsilon \qquad\qquad \tau e \downharpoonright_{\theta}^{M} \begin{cases} (\tau \downharpoonright_{\theta}^{M})e & \text{if } e \in \mathsf{relevant}_M(\theta) \\ (\tau \downharpoonright_{\theta}^{M}) & \text{otherwise} \end{cases}$$

We can now define a predicate $\mathsf{satisfy}_M^P(\tau, \theta)$ determining whether a trace τ satisfies a tracematch $\langle P, M, C\rangle$, given a specific binding θ. Briefly this is the case if τ matches with a word in $\mathcal{L}(P)$ and its last event is relevant. This last condition is important as otherwise irrelevant events could cause matching:

$$\mathsf{satisfy}_M^P(\tau, \theta) = \bigvee_{\sigma \in \mathcal{L}(P)} \mathsf{match}_{M^{\theta}}(\sigma, \tau \downharpoonright_{\theta}^{M}) \wedge \mathsf{last}(\tau) \in \mathsf{relevant}_M(\theta)$$

We can now define the set of bindings generated from a trace τ, each of which will cause execution of the code associated with the tracematch:

$$\mathsf{bindings}_M^P(\tau) = \{\theta \mid \tau' \text{ is a suffix of } \tau \wedge \mathsf{satisfy}_M^P(\tau', \theta)\}$$

The intention is that the function bindings is applied at each step during monitoring where τ represents the current trace of events. If the set of bindings is empty in a particular step no code will get executed. The above semantics is of course very inefficient (even not computable as presented) and an equivalent operational more efficient semantics is needed, as introduced in the following section.

7.3.3. *Operational Semantics*

Given tracematch $\langle P, M, C\rangle$, we shall construct a state machine labelled with constraints that are updated as events are consumed. If a constraint labelling a final state has solutions (bindings satisfying the constraint) then the code C will be executed for each solution. The approach is to use P to generate a regular expression P^{+}, which can be used to construct the state machine. P^{+} should be such that, according to the declarative semantics, a trace matches P^{+} if: (i) a relevant suffix of that trace matches P and (ii) the last event of the trace is relevant. Formally P^{+} should satisfy the following equation:

$$\mathsf{bindings}^P_M(\tau) = \mathsf{solutions}(\bigvee_{\sigma\in\mathcal{L}(P^+)} \mathsf{match}_M(\sigma,\tau))$$

This equation formalizes the idea that for a given trace τ, we follow each word through the automaton for P^+, and collect the solution(s) to the generated constraint. Note that in contrast to the definition of $\mathsf{bindings}^P_M(\tau)$ in the declarative semantics, P^+ takes care of (i) and (ii) above. It is for example a regular expression on the *full* trace. To achieve this, P^+ is defined as follows:

$$P^+ = \underbrace{U^*(P \parallel \mathbf{skip}^*)}_{(i)}\underbrace{\cap(\Sigma^* S)}_{(ii)}$$

Where U is the set of all symbolic events, $\parallel$ is the interleaving operation and **skip** is a special symbolic event that matches irrelevant events. We define this special symbolic event **skip** in terms of how it is interpreted by M, its main function is to ensure that we do not skip relevant events. We will achieve this by making it produce a constraint that will remove any existing constraints relevant to the event. Therefore, for a constraint to remain on a state it must be reintroduced by an incoming transition. As n event is relevant to constraint C iff $\exists a \in S : (M(a)(e) \wedge C) \neq false$ we let **skip** be defined by:

$$M(\mathbf{skip})(e) = \bigwedge_{a\in S} \neg M(a)(e)$$

This achieves our aim - if e is relevant to some constraint C then $M(\mathbf{skip})(e)$ will contradict C. For example $M(\mathbf{skip})(\mathtt{com}(\text{A,1}))$ would be:

$$\neg M(\mathtt{com}(n,x))(\mathtt{com}(\text{A,1})) \wedge \neg M(\mathtt{suc}(n,x))(\mathtt{com}(\text{A,1})) \wedge \neg M(\mathtt{fail}(n,x))(\mathtt{com}(\text{A,1}))$$
$$= \neg(n = A \wedge x = 1) \wedge \neg \mathit{false} \wedge \neg \mathit{false}$$
$$= (n \neq A \vee x \neq 1)$$

Which contradicts $(n = A \wedge x = 1)$. Note that if e is not in S this will be true.

Based on this operational semantics, we can consider an implementation which would realise TRACEMATCHES. Firstly we construct an automaton for $P+$ - this adds self-looping transitions for **skip** to P, except in the final states. We then associate a label (of constraints) with each state and update this label for state i as follows:

$$\mathit{label}_i{}' = \left(\bigvee_{j \xrightarrow{a} i} (\mathit{label}_j \wedge M(a)(e))\right)$$

As we have self-looping skip transitions on non-final states, the part of a state's constraint which is not relevant to e is kept. We also add (or 'move') the relevant (to e) part of any constraint labelling a state with a transition to i.

7.4. Worked Example

Let us consider the *Respect Conflicts* property. A trace should match if there is a conflict between resource r_1 and resource r_2 and they are (at some point) granted at the same time - note that `cancel_r2` is not defined here so the ordering matters.

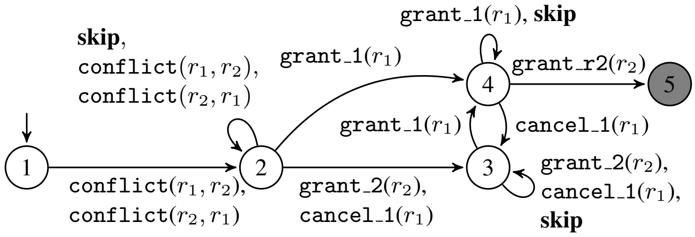

Figure 4. A finite state machine for the *Respect Conflicts* property.

Trace	State constraints Let C_1 = (r$_1$=w $\wedge$ r_2=c) and C_2 = (r_1=c $\wedge$ r_2=w)				
	1	2	3	4	5
	true	false	false	false	false
conflict(w,c)	true	$C_1 \vee C_2$	false	false	false
grant(w)	true	false	C_2	C_1	false
cancel(w)	true	false	$C_1 \vee C_2$	false	false
grant(c)	true	false	C_1	C_2	false
grant(c)	true	false	C_1	false	C_2

Figure 5. Evaluating a trace for the *Respect Conflicts* example.

```
public aspect RespectConflicts extends Pointcuts{
  tracematch(Res r1, Res r2)
  {
    sym conflict after: (conflict(r1,r2) || conflict(r2,r1));
    sym grant_1 after: grant(r1,*);
    sym cancel_1 after: cancel(r1,*);
    sym grant_2 after: grant(r2,*);

    conflict+ (grant_1 | cancel_1 | grant_2)* grant_1 grant_2
    {
      if(r1!=r2)
        error("Conflicting resources "+r1 + " and "+r2+" both granted");
    }
  }
}
```

We first generate the state machine in Figure 4 and associate the $true$ constraint to state 1 and $false$ to every other state. Note that when extracting S from the tracematch the **sym** command for `conflict` introduces two events as it is a disjunction (although we can only refer to them together in P). Figure 5 then demonstrates how constraints are updated for each state for a given trace dealing with the wheels (w) and camera (c). Consider, for example, the evaluation of the second event. The constraint C_1 is removed from state 2 by **skip** as it is relevant to the event, it is also added to state 4 as

$$(C_1 \vee C_2) \wedge M(\texttt{grant}(r_1))(\texttt{grant}(\mathrm{w})) = (C_1 \vee C_2) \wedge (r_1 = \mathrm{w}) = C_1$$

After evaluating the trace there exists a constraint, C_2, labelling the accepting state 5. We then execute the code for the solution to this constraint - the binding $[r_1 \mapsto \mathrm{c}, r_2 \mapsto \mathrm{w}]$.

8. JavaMOP

8.1. Overview

JavaMOP [15,10,1] is a tool and an associated specification language from the *Monitoring-Oriented Programming* (MOP) approach - an attempt to formalise the process of monitoring programs as a programming methodology. JavaMOP is packaged as a stand-alone tool that compiles specifications into AspectJ aspects. Aspects can then be directly weaved into the monitored system. JavaMOP allows a user to embed code (actions) into the specification, making feedback part of the system. In this section we will see how JavaMOP combines *parametric trace slicing* with *logic plugins* to give a generic framework for parametric runtime monitoring.

8.2. Introductory Example

Let us consider our running *Exactly One Success* property. JavaMOP does not allow us to import pointcuts so we would have to expand those defined in Section 6 in the following:

```
ExactlyOneSuccess(String n, Integer x) {

  event com after(String n, Integer x) : com(n,x,*){}
  event suc after(String n, Integer x) : suc(n,x){}
  event fail after(String n, Integer x) : fail(n,x){}

  // the specification using some logic plugin
}
```

JavaMOP provides different *logic plugins* to specify properties over the given events. We can use regular expressions in the same way as TraceMatches - describing the behaviour leading to failure. By default this is matched against the whole trace, however JavaMOP can also run in *suffix-matching* mode to give the same behaviour as TraceMatches. A code segment can be associated with the matching of a regular expression:

```
  ere : com (fail | (suc suc))
  @match{ error(”Command (”+n+”,”+x+”) failed!”); }
```

Here **ere** indicates that the ‘extended regular expression’ logic plugin is being used. Alternatively, it is possible to define correct behaviour and look for deviations. In this case we associate the code snippet with the failure to match a regular expression:

```
  ere : com suc
  @fail{ error(”Command (”+n+”,”+x+”) failed!”); }
```

Alternatively we can use the ‘finite state machine’ plugin to take remedial action on command failure by keeping track of the current state of the property and associating actions with each state:

```
fsm :
   start [
      com -> active
   ]
   active [
      suc -> done
      fail -> failed
   ]
   done [
      suc -> twoSuccess
   ]
   failed [ ]
   twoSuccess [ ]

   @fail{ error(n,x,"unexpected action"); }
   @active{ log.print("Command ("+n+","+x+") issued."); }
   @failed{
       log.print("Command ("+n+","+x+") failed, resending.");
       resend(n,x);
   }
   @twoSuccess{
       log.print("Command ("+n+","+x+") has succeeded twice.");
       error(n,x,"twoSuccess");
   }
```

The **fsm** logic assumes that if no transition can be taken then a failure occurs, implicitly 'completing' the state machine to a hidden *fail* state (distinct from the *failure* state).

Let us now consider how we should interpret these specifications. To do this we must understand how data values are dealt with. JAVAMOP uses the notion of *parametric trace slicing* to *slice* the trace into a set of propositional traces, one for each set of parameters. As an example, let us consider the following trace:

$$\texttt{com}(\text{“move”},1).\texttt{suc}(\text{“move”},1).\texttt{com}(\text{“stop”},1).\texttt{suc}(\text{“move”},1).$$

We have two bindings of the parameters and, in this simple example, we can easily identify the propositional traces associated with each of them:

$$\begin{array}{ll}[\text{name} \mapsto \text{“move”}, \text{x} \mapsto 1] \mapsto \texttt{com.suc.suc} \\ [\text{name} \mapsto \text{“stop”}, \text{x} \mapsto 1] & \mapsto \texttt{com}\end{array}$$

A propositional 'plugin' can then be used to check each of these propositional traces. Generally this involves constructing a state machine, directly or indirectly, from the propositional specification and finding the state reached by the propositional trace. For each binding, the appropriate action can then be selected based on this state.

8.3. The Underling Theory behind MOP

JAVAMOP consists of two parts. First, the *parametric trace slicing* technique slices a parametric trace into a set of propositional traces, each associated with a binding. Second, a *logic plugin* defines how to interpret each propositional trace. Furthermore, JAVAMOP can run in different modes - changing how the specification should be interpreted.

The theory behind JAVAMOP is defined in terms of *parameterized events*, which are distinct from the events introduced in Section 3.2. A parameterized event is a pair in the set $\Sigma \times Binding$, where Σ is a set of event names. To translate events into parameterized events we introduce a function based on concepts introduced in section 3.5.1 to give the unique binding for an incoming event. To ensure this binding is unique the set of symbolic events S cannot contain two events with the same name. Given a set of symbolic events S and an event $e(\overline{v})$ we define the translation as follows:

$$\mathsf{match}(S, e(\overline{v})) = e(\texttt{match}(a, e(\overline{v}))) \text{ where } a \in S \text{ and } \texttt{matches}(a, e(\overline{v}))$$

This restriction on S is limiting. Furthermore, in JAVAMOP one cannot define the events `grant_r1` and `grant_r2` both related to the same pointcut (but with different variables), as we did with TRACEMATCHES, and we can therefore not specify the *Respect Conflicts* example using JAVAMOP without resorting to programming (as in Section 5).

Let us introduce *parametric trace slicing* as an implementation of the function:

$$\mathsf{pts} \in \mathit{Trace}(\mathit{Event}) \times \mathit{Binding} \to \mathit{Trace}(\Sigma).$$

For a given binding of variables, a trace 'slice' (subtrace) is defined as the largest relevant subtrace. An event $e(\theta')$ is relevant to a binding θ if it only includes things mentioned in θ, i.e., θ' is a submap of θ. Therefore, we define our function as $\mathsf{pts}(\tau, \theta) = \tau \downarrow_\theta$ where

$$\epsilon \downarrow_\theta = \epsilon \qquad e(\theta')\tau \downarrow_\theta = \begin{cases} e(\tau \downarrow_\theta) \text{ if } \theta' \sqsubseteq \theta \\ \tau \downarrow_\theta \quad \text{otherwise} \end{cases}$$

Note the similarity with filtering in TRACEMATCHES on page 16.

The propositional traces produced by parametric trace slicing are then interpreted by a *logic plugin*, which provides the following function for a given set of verdicts:

$$\mathsf{plugin} \in \mathit{Trace}(\Sigma) \to \mathit{Verdict}$$

The logic plugins currently provided include: Finite State Machines (**fsm**), Extended Regular Expressions (**ere**), Context Free Grammars (**cfg**), Linear Temporal Logic (**ltl**) and String Rewriting Systems (**srs**).

Code snippets (actions) are associated with verdicts produced by the logic plugin - match, fail or (sets of) states in a finite state machine. This code can contain variables which need to be instantiated - which can be provided by a binding. Given a piece of code γ we write its execution for binding θ as $\gamma(\theta)$. We define an action as follows:

$$\mathit{Action} = \mathit{Verdict} \times (\mathit{Code}$$

We can then execute the appropriate code using the two functions previously introduced to give a verdict for a trace given a binding and then determine the code to execute on observing a parametric trace τ:

$$\mathsf{check} \in \mathit{Trace}(\mathit{Event}) \times \mathit{Binding} \to \mathit{Verdict} = \mathsf{plugin} \circ \mathsf{pts}$$
$$\mathsf{execute}(\tau) = \mathit{act.snd}(\theta) \text{ for any } \theta \text{ where } \mathit{act} \in \mathit{Actions} \wedge \mathit{act.fst} = \mathsf{check}(\tau, \theta)\}$$

Finally, JAVAMOP can be run in different modes. Let us discuss those modes which alter the semantics. Firstly, two modes determine the traces passed into the system:

- suffix - Performs suffix matching rather than total matching.
- perthread - Constructs a separate trace per program thread.

Then three modes place filters on the set execute(τ) to contain:

- full-binding - Only bindings that bind all variables in the specification.
- maximal-bindings - Only bindings maximal in that set with respect to $\sqsubseteq$.
- connected - Only connected bindings. A binding is connected if all bound values are connected (transitively) by events.

The last mode is interesting as it allows us to define behaviors for objects related by events. For example - for every resource used by some task.

8.4. A Basic Monitoring Algorithm

Figure 6 gives an algorithm for monitoring a trace given a specification using a set of symbolic events $\mathcal{A}$ and presented as a finite state machine with transition function δ and set of final states F. We will consider only logic plugins which can be translated to finite-state machines (and give no details of this translation). We will not consider alterations required for different modes. These details are discussed in further details in [15].

A map Δ, from bindings to states in the finite state machine is constructed, first being initialised with the empty binding ($\bot$) mapped to the initial state (q_0). Then, for each event in the trace, the binding θ is extracted from the event (line 4) and each previously seen binding θ' is examined. If θ' is consistent with θ (they agree on all shared variables) then we find the *maximal* existing consistent binding θ_m (lines 9 and 10) and use this to create an entry in Δ (line 11). Note that if θ has been previously seen then $\theta_m = \theta$. We consider this process further in the next section where we examine a worked example.

```
   Input: a parametric trace $\tau$
   Output: Matching bindings
1  $\Delta : [Bind \rightharpoonup State]$; $\Theta : Bind$;
2  $\Delta \leftarrow [\bot \rightarrow q_0]$ ;
3  foreach event $e(\overline{v}) \in \tau$ in order do
4      $\theta \leftarrow \texttt{match}(S, e(\overline{v}))$;
5      $\Theta \leftarrow \underline{dom}(\Delta)$;
6      foreach $\theta' \in \Theta$ do
7          if $\theta$ is consistent with $\theta'$ then
8              $\theta_{max} \leftarrow \theta'$;
9              foreach $\theta_{alt} \in \Theta$ do
10                 if $\theta_{max} \sqsubseteq \theta_{alt} \sqsubseteq \theta \dagger \theta'$
                   then $\theta_{max} = \theta_{alt}$
11             $\Delta(\theta \dagger \theta') \leftarrow \delta(\Delta(\theta_{max}), e)$

12 return $\{\theta \in \underline{dom}(\Delta) \mid \Delta(\theta) \in F\}$
```

Figure 6.: A basic monitoring algorithm.

8.5. Worked Example

Let us examine the *Release Resource* property. We define this property using the **ere** logic plugin as follows:

```
ReleaseResource(String n, int x, int t, Res r){

  event com before(String n, int x, int t) : com(n,x,t){}
  event suc before(String n, int x) : suc(n,x) { }
  event grant before(Res r, int t) : grant(r,t) { }
  event cancel before (Res r, int t) : cancel(r,t) { }
  ere : com (grant cancel)* grant suc

  @match{
    System.err.println("Resource "+r+" not released before command "+n+":"
                                        +x+" completed by task "+t);
   }
}
```

The first step requires us to translate the regular expression into a finite state machine, as given in Figure 7. Figure 8 illustrates monitoring of the trace using the algorithm in Figure 6. Let us consider how Δ is updated. To begin with, Δ contains only the empty binding mapped to the initial state. On receiving the first event, the extracted binding is $\theta_1 = [n \mapsto \text{A}, x \mapsto 1, t \mapsto 10]$ and as the only existing binding is the empty binding a new entry is created in Δ for θ_1, using the state associated with the empty binding. On receiving the second event, the binding $\theta_2 = [t \mapsto 10, r \mapsto \text{Z}]$ is extracted. By considering the empty binding an entry for θ_2 is added to Δ. The interesting step which captures the main intuition behind this algorithm occurs when consider what happens when we consider the existing binding θ_1. The bindings θ_1 and θ_2 are consistent and the maximal binding is θ_1 - as demonstrated in the lattice on the right of Figure 8. Therefore an entry is created for the binding $\theta_2 \dagger \theta_1$ using the state associated with θ_1. The rest of the trace is processed as above and we reach an accepting state on the final event, indicating that the specification has been violated.

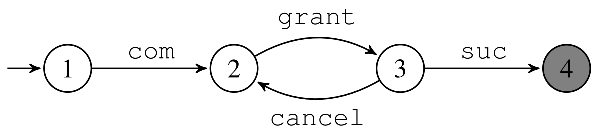

Figure 7. A finite state machine for the regular expression used in the *Release Resource* property.

Δ									
Binding				Trace					
n	x	t	r	com(A,1,10)	grant(10,Z)	grant(10,Y)	cancel(10,Y)	suc(A,1)
-	-	-	-	1	1	1	1	1
A	1	10	-	2	2	2	2	1
A	1	10	Z	-	3	3	3	**4**
-	-	10	Z	-	F	F	F	F
A	1	10	Y	-	-	3	2	2
-	-	10	Y	-	-	F	F	F

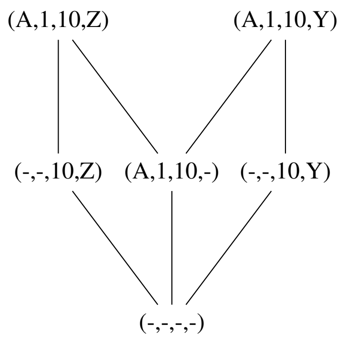

Figure 8. Monitoring a trace with the algorithm in Figure 6. The table on the left gives Δ for each event. The lattice on the right demonstrates the relationship between the bindings in Δ.

9. RULER-lite

9.1. Overview

In this section we introduce RULER-lite, a cut-down version of the RULER [8] system. RULER is a highly expressive rule-based runtime verification system. It has its roots in EAGLE [4] - a general-purpose, rule-based, temporal system for defining, or programming, monitors for complex temporal behavioral patterns. The EAGLE logic was highly expressive, yet an efficient monitoring algorithm remained elusive, partly due to the complex constructs used in EAGLE's logic. RULER was developed as a low-level system, into which specifications written in languages such as EAGLE could be compiled. However, it became clear that the low-level formulation of rule systems was an intuitive and powerful approach, and RULER became a runtime verification system in its own right.

RULER-lite is a cut-down version of RULER developed for this chapter. It removes many of the powerful features of RULER, whilst remaining highly expressive. The main features that are excluded are parameterization of rules with facts, multiple events in a single step, run-to-completion evaluation and monitor combination (leading to a four, rather than five valued logic as monitor combination can introduce an 'unknown' result').

9.2. RULER-*lite by Example*

```
ruler ExactlyOneSuccess{
  observes
    com(string ,int ),
    suc(string ,int ),
    fail(string ,int );

  always Start(){
    com(n:string ,x:int)
      -> Active(n,x);
  }
  state Active(n:string ,x:int){
    suc(n,x) -> Done(n,x);
    fail(n,x) -> fail;
  }
  state Done(n:string ,x:int){
    suc(n,x) -> fail;
  }
  initials Start;
  forbidden Active;
}
```

Figure 9.: RULER-lite specification for the *Exactly One Success* property.

Figure 9 gives a RULER-lite specification for our running *Exactly One Success* example property. The specification consists of three main parts - a list of event definitions, a list of rules, and a list of verdict conditions.

We first introduce the three events that can be observed with given parameter types. Next, we have a list of rule definitions. RULER-lite's main idea is to use events to rewrite sets of facts about the monitored system, with verdicts based on the (non) existence of certain facts. Properties are captured by a *rule system*, where a rule indicates how a certain type of fact should be rewritten.

A rule system describes an infinite state system - a set of facts is a *state* and an observed event causes a state to be rewritten into a new state. Figure 10 gives a partial unrolling of the system described by our property i.e., *Active(A,1)* and *Start* are facts. This rewriting process may be non-deterministic - one concrete event may lead to more than one state. Therefore a monitor stores the current set of active states, called a *frontier* i.e., a *monitor state* is a set of states.

These facts either record past behaviour or introduce *obligations* of future behavior. A rule consists of a name, a list of parameters and a list of rewrite rules of the form

$$c_1 \wedge \ldots \wedge c_n \rightarrow (o_1 \wedge \ldots \wedge o_j) \vee \ldots \vee (o_k \wedge \ldots o_{k+l})$$

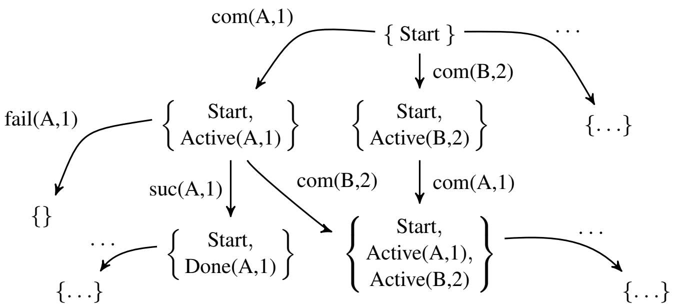

Figure 10. Part of infinite state machine described by the *Exactly One Success* property as given in Figure 9.

Verdict	Indicates	Condition on frontier.
True	Ultimate success	An empty state or a state containing a success fact.
False	Ultimate failure	An empty frontier.
Still True	Current success	One state with non-forbidden facts.
Still False	Current failure	Otherwise - all states contain forbidden facts.

Table 1. An explanation of verdicts.

where c_i is a *condition* and o_i is an *obligation*. This can be read as *if all conditions are true in the present then the future must satisfy one set of obligations*. Conditions and obligations are written using facts and conditions can refer to the incoming event.

The last part of the specification gives the initial frontier, here the fact Start, and then gives sets of rule names for deciding verdicts - there are three possible sets of rule names - if a state contains a **forbidden** rule name then it is currently failing, if a state contains a **succeed** rule name then it is completely successful and if a state doesn't fire an **assert**ed rule on each step it is completely failing.

RuleR-lite uses a four-valued verdict logic, summarized in Table 1, which is based on the frontier. If a state is empty then all obligations have been met (the property cannot fail). If the frontier is empty then no state met their obligations (the property cannot succeed). If a state contains no forbidden facts then the property is currently being met - but this is not final. If all states contain forbidden facts the property is currently failing, but this is not final.

In the specification given in Figure 9 we return the verdict *Still False* if an issued command has not yet succeeded, *False* if a command has failed or succeeded more than once, and *Still True* otherwise. *Still False* is returned because of the **forbidden** command and *Fail* because of the fail obligation in the Active and Done rules. The fail obligation is one of two special obligations that can be added to a state - *fail* causes the state to be removed from the frontier and *ok* causes the state to collapse, leaving an empty state.

Trace	Frontier
	{{Start }}
com(move,1)	{{Start, Active(move,1)}}}
suc(move,1)	{{Start, Done(move,1)}}
com(stop,2)	{{Start, Done(move,1), Active("stop",2)}}}
suc(move,1)	{ }

Table 2.: Processing a trace.

Let us consider how the rule system in Figure 9 should be interpreted using the example trace in Table 2. We consider this process an event at a time. The initial frontier consists of a single state containing a Start fact - there is no non-determinism so we will only be dealing with a single state. For the first event, com(move,1), we consider the only fact in the state - *Start*. We unify com(move,1) with com(n:**string**,x:**int**) to give us a binding $[\mathrm{n} \mapsto \mathrm{move}, \mathrm{x} \mapsto 1]$ and use this to add a new fact *Active(move,1)* to the state. We add the *Start* fact back into the state due to its **always**-modifier. Each rule has a modifier indicating what should happen to a fact of that kind when it fires. This can be either **step** - only be active for a single step, **state** - active until 'fired', or **always** - active until removed by negation. Therefore, in this case *Active* facts are removed when fired, but the *Start* fact is never removed. Finally, we return the *Still False* verdict as there is a forbidden *Active* fact in the state - we have a command that has not yet succeeded. For the next observed event, suc(move,1), we check if each fact in the state can fire. By applying the binding in the fact *Active(move,1)* we match the condition success(move,1) in the *Active* rule's first rewrite rule and add the fact *Done(move,1)* to the state. The **state** modifier tells us to remove the *Active(move,1)* fact. We return the *Still True* verdict. The third event is processed in the same way as the first, adding another fact to the state and returning *Still False*. Finally, another suc(move,1) event is received. The *Done* rule is the only active rule which fires, adding the *fail* obligation to the state, causing it to be removed and leaving an empty frontier. As indicated in Table 1, an empty frontier represents failure and *False* is returned as the given trace violates our property - the move command with id 1 succeeds twice.

9.3. Rule Systems

We present the objects described in the previous sections more formally. We assume a set of rule names $Name$. As rules may refer to facts we introduce these first.

Definition 7 (Fact) *A fact $\langle r, \varphi \rangle$ is a pair consisting of a rule name and a binding i.e. $Fact = Name \times Binding$.*

Events and facts may appear in the conditions of rewrite rules with values, bound variables or free variables in their parameter lists. To capture free variables we introduce *introductions*, for example com(n:**string**,x:**int**) is an event introduction.

Definition 8 (Event and fact introductions) *An* introduction *is a pair $x : t$ where x is a variable and t is a type. An* event introduction *is a pair $\langle e, \overline{z} \rangle$ where $e \in \Sigma$ and $\overline{z}$ is a list of values, variables, or introductions. A* fact introduction *$\langle n, \varphi, \Gamma \rangle$ extends the concept of a fact with a map from variables to introductions.*

Conditions and obligations are statements over events and facts. Conditions may introduce new variables, whereas obligations cannot.

Definition 9 (Conditions and obligations) *An obligation is a (possibly negated) fact. A condition may be either a (possibly negated) event or fact introduction, or a well-typed boolean expression over variables.*

A list of obligations os can be instantiated with a binding θ by replacing all variables in os with their associated value in θ. We use conditions and obligations to build rules.

Definition 10 (Rule) *A* rule *is a tuple $\langle \mathbf{modifier}, name, (x_1 : t_1, \ldots x_n : t_n), R \rangle$ where $\mathbf{modifer} \in \{\mathbf{step}, \mathbf{state}, \mathbf{always}\}$ is a persistence modifier, $name \in Name$, $(x_1 : t_1, \ldots x_n : t_n)$ are the rule's parameters and R is a set of rewrite rules of the form $cs \rightarrow os_1 \vee \ldots \vee os_k$ where cs is a condition list and os_i is an obligation list.*

Definition 11 (Rule system) *A* rule system *is a map (partial function) from rule names to rules i.e.* $RuleSystem = String \rightharpoonup Rule$*.*

Defining a rule system as a map from rule names to rules allows us to easily access the rule associated with a given a fact. Facts are gathered together in states in a frontier.

Definition 12 (State and frontier) *A state is a set of facts and a frontier is a set of states i.e., State ⊂ Fact and Frontier ⊂ State.*

There are a number of well-formedness conditions. A fact $\langle n, \varphi\rangle$ is well-formed iff $\underline{dom}(\varphi)$ equals the parameters of the associated rule. A fact introduction $\langle n, \varphi, \Gamma\rangle$ is well formed iff $\underline{dom}(\varphi) \cap \underline{dom}(\Gamma) = \emptyset$ and $\underline{dom}(\varphi) \cup \underline{dom}(\Gamma)$ equals the parameters of the associated rule. A condition list is well formed if an introduced variable is not used before it is introduced. A rule $\langle m, n, p, R\rangle$ is well-formed if for each rewrite rule $cs \rightarrow os_1 \vee \ldots \vee os_k \in R$ the non-introduced variables in cs are contained in p, and the variables of each os_i are contained in either p or the variables introduced in cs.

9.4. Monitoring Algorithm

Figure 11 gives a monitoring algorithm for RULER-lite which takes a rule system and computes a verdict for each event in a trace, making use of algorithms given in Figures 12 and 13. The algorithm consists of a main loop over the given trace and two functions - one to process an event, and one to check the frontier. The frontier is initialized using the **initials** set.

The PROCESS function builds a new frontier by rewriting each state in the old frontier into a set of states. For each state we use P to keep track of the persistent facts from the previous state and N to build up the new states to be added. A fact is persistent if it is an **always** rule (lines 13-4) or an unfired **state** rule (lines 13-4, 20-1).

For each fact in a state and each rewrite rule in the associated rule, we use the UNIFY function in Figure 13 to build up a set of bindings which satisfy the condition list of that rewrite rule (line 16), if this set is empty then the condition cannot be satisfied and the rule is not fired, otherwise the rule is fired for each binding. To fire a rule we use the bindings to instantiate the rewrite rule's obligations (line 18). The computed states are filtered for asserted rule names if given (lines 23-4) and then combined with the persistent rules and added to the new frontier (line 25).

```
1  frontier : Frontier := initials ;
2  RS : RuleSystem := rule system;

3  foreach event ∈ trace do
4      frontier := PROCESS(event);
5      output CHECK(frontier)

6  PROCESS(event):
7  begin
8      newF = ∅;
9      foreach $s$ ∈ frontier do
10         P := ∅; N := { ∅ };
11         foreach $\langle n, \theta\rangle \in s$ do
12             $\langle m, n, \_, R\rangle$ = RS(n);
13             if $m \neq$ step then
14                 P = P + $\langle n, \theta\rangle$
15             foreach $(cs \rightarrow Os) \in R$ do
16                 B :=
                   UNIFY($\theta$, $cs$, $s$,event);
17                 foreach $\theta' \in B$ do
18                     N = {$s' \cup os(\theta')$ |
19                             $s'$ ∈ N, $os \in Os$};
20                     if $m$=state then
21                         P := P - $\langle n, \theta\rangle$

22         if assert is not empty then
23             N := {$s' \in N$ | $\exists\langle n, \theta\rangle \in s$ :
24                                     $n$ ∈ assert }
25         newF $\cup=$ {$P \cup s'$ | $s' \in N$}
26     return newF
27 end
```

Figure 11.: A monitoring algorithm.

The CHECK function implements Table 1. This introduces the collapse function to resolve any negated facts in a state - if $\neg\langle n, \theta\rangle \in s$ then all instances of $\langle n, \theta\rangle$ and $\neg\langle n, \theta\rangle$ are removed from s. This may cause a state to become empty, and is the only way in which a fact with **always** persistence is removed.

```
CHECK(frontier):
begin
    frontier := collapse(frontier);
    if frontier= $\emptyset$ then return False;
    else if $\emptyset$ $\in$frontier or $s \cap$ succeed $\neq \emptyset$
    then return True;
    else if $\forall s$ $\in$frontier. $s \cap$forbidden $\neq \emptyset$
    then return Still False;
    else return Still True;
end
```

Figure 12.: Computing a verdict.

```
UNIFY($\theta$, $cs$, $s$,event):
if $cs$ is empty then return $\{\theta\}$ else
    c := head($cs$);
    if c is $\neg$c$'$ and UNIFY($\theta$,[c$'$],$s$,event) = $\emptyset$ then return UNIFY($\theta$,tail($cs$),$s$,event)
    else if c is boolean expression $\psi$ and $\theta \models \psi$ then return UNIFY($\theta$,tail($cs$),$s$,event)
    else if c is event introduction $e(\ldots, z_i, \ldots)$ and matches($e(\ldots, y_i, \ldots)$, event) where
    $y_i = \begin{cases} v & \text{if } y_i \text{ is value } v \\ \theta(x) & \text{if } y_i \text{ is variable } x \\ x & \text{if } y_i \text{ is introduction } x : t \end{cases}$ then
        return UNIFY($\theta \dagger$ match($e(\ldots, y_i, \ldots)$), event),tail($cs$),$s$,event)
    else if c is fact introduction $\langle n, \varphi, \Gamma\rangle$ then
        B := $\{[\ldots, x_i \mapsto \varphi'(x_i), \ldots] \mid \langle n, \varphi'\rangle \in s \wedge \varphi' \sqsubseteq \theta \dagger \varphi \wedge x_i \in \mathsf{dom}(\Gamma)\}$
        return $\bigcup_{\varphi' \in B}$ UNIFY($\theta \dagger \varphi'$,tail($cs$),$s$,event)
    else return $\emptyset$
```

Figure 13. Unify function

9.5. A Worked Example

Figure 14 gives the RULER-lite specification for the *Respect Priorities* property. We show in Figure 15 how this rule system is used to rewrite an initial state for a trace of events dealing with the camera and wheel resources - we abbreviate these to c and w respectively.

The first event causes *P(c,w)* to be added, recording that the camera has higher priority than the wheels, and the second event introduces *G(w)* to record that the wheels have been granted. On receiving the third event we satisfy the conditions of a rewrite rule in the *G* rule. The {: A;B;... :} notation is syntactic sugar for saying if A does not fire try B and so on - note that we use a common condition prefix. Unifying `request`(c) with request(x:**obj**) under $[y \mapsto$ c$]$ gives us $[x \mapsto$ w$, y \mapsto$ c$]$ so we check if the state contains *P(c,w)*, which it does. As we cannot find a fact in the state matching P(z:**obj**,c) we fall through to the default behaviour of adding the fact *Res(w,c)* to the state.

```
ruler RespectPriorities{
  observes
    priority(obj,obj),
    request(obj), grant(obj),
    cancel(obj), rescind(obj);

  always Start(){
    priority(x:obj,y:obj)
        -> P(x,y);
    grant(x:obj) -> G(x);
  }
  state P(x:obj,y:obj){}
  always G(x:obj){
    cancel(x) -> !G(x);
    request(y:obj), P(y,x)
    {:
      P(z:obj,y),G(z) -> Ok;
      default -> Res(x,y);
    :}
  }
  state Res(x:obj,y:obj){
   rescind(x) -> Ok;
   grant(y) -> Fail;
  }
  initials Start;
}
```

Figure 14.: Specifying the Respect Priorities property.

$$\{\text{Start}\} \xrightarrow{\text{priority(c,w)}} \left\{\begin{array}{c}\text{Start,}\\ \text{P(c,w)}\end{array}\right\} \xrightarrow{\text{grant(w)}} \left\{\begin{array}{c}\text{Start,}\\ \text{P(c,w),}\\ \text{G(w)}\end{array}\right\} \xrightarrow{\text{request(c)}} \left\{\begin{array}{c}\text{Start,}\\ \text{P(c,w),}\\ \text{G(w),}\\ \text{Res(w,c)}\end{array}\right\} \xrightarrow{\text{grant(c)}} \{\}$$

Figure 15. Rewriting a state for the *Respect Priorities* property.

The final event matches with the grant(y) condition in the *Res* rule, adding the fail obligation and collapsing the state. The state is empty and the *False* verdict is returned.

10. TraceContract

10.1. Overview

In the previous sections we have seen three examples of DSLs (Domain-Specific Languages) for writing runtime monitors. These DSLs are so-called *external* in the sense that they are stand-alone, usually small, languages (compared to a full programming language), implemented with special parsers. An alternative is what is often referred to as *internal* DSLs. An internal DSL is a suggested way of writing code in a general-purpose programming language, the *host language*, specific for a particular domain. Usually an internal DSL is implemented as an API. No parser is needed beyond that of the host language. The advantages of an internal DSL include: (i) expressiveness due to access to the host language, (ii) ease of development and maintenance/change, (iii) use of existing tools such as type checkers, IDEs, etc. The advantages of an external DSL include: (i) it can be optimally succinct, easing writing and reading programs, (ii) it can be easy to learn, (ii) programs can be analyzed more easily since we have access to the parse trees. TraceContract [7] is an internal DSL for writing runtime monitors, an API in the Scala programming language. As we shall see, Scala provides good support for the creation of internal DSLs. TraceContract offers an experimental combination of parameterized state machines with anonymous states, referred to as state logic, future time linear temporal logic (LTL) - not further discussed in this section, and rule-based programming; as well as free combinations of these forms.

10.2. Examples

Let us illustrate TraceContract by specifying the two through-going requirements *Exactly One Success* and *Respect Priorities*. We first need to define the type of events to be monitored by defining an abstract type *Event*, and defining each individual event type as a sub-class (**case**-classes allow for pattern matching over objects of the class and do not require use of the **new** keyword to generate objects):

```
abstract class Event
case class Command(name: String, number: Int) extends Event
case class Success(name: String , number: Int ) extends Event
case class Fail (name: String , number: Int ) extends Event
case class Priority (r1: Resource, r2: Resource) extends Event
case class Grant(r: Resource, t: Task) extends Event
case class Request(t: Task, r: Resource) extends Event
case class Rescind(r: Resource, t: Task) extends Event
case class Cancel(t: Task, r: Resource) extends Event
```

The monitor for the property *Exactly One Success* is shown below. The monitor, looking very similar to the formalization of the same property in RuleR-lite, is expressed as a class *ExactlyOneSuccess* that sub-classes a class *Monitor*, which is part of the TraceContract API. Class *Monitor* is parameterized with the event type and offers methods needed for writing monitors. The monitor reads as follows: (always) require that whenever a command is observed, enter a *hot Active* state. A hot state has to be exited before the end of the analysis, corresponding to *forbidden* states in RuleR-lite. A success leads to the *Done* state to become active, where we just watch that another success does not occur. It is acceptable to be in a *state* at the end of the analysis (it is a final state).

```
class ExactlyOneSuccess extends Monitor[Event] {
  require {
    case Command(n, x) => Active(n, x)
  }

  def Active(name: String, number: Int) =
    hot {
      case Fail(`name`, `number`) => error
      case Success(`name`, `number`) => Done(name, number)
    }

  def Done(name: String, number: Int) =
    state {
      case Success(`name`, `number`) => error
    }
}
```

The main observation to make here is that this monitor only uses four methods from the DSL: *require*, *hot*, *state* and *error*. The remaining part of the monitor is standard Scala code. Specifically *Active* and *Done* are Scala functions, and for example `Active(n, x)` is a function call. The TraceContract functions *require*, *hot*, and *state* take as argument a Scala *partial function*: {**case** ... => ...} defined with pattern matching (one or more **case**-statements). A quoted variable, such as ‘name‘ represents: match the *value of* the variable rather than binding to it.

Since in this case the function definitions are not recursive, calls of these can be inlined as shown in the temporal logic flavored version of the above monitor below:

```
class ExactlyOneSuccess extends Monitor[Event] {
  require {
    case Command(n, x) =>
      hot {
        case Fail(`n`, `x`) => error
        case Success(`n`, `x`) =>
          state {
            case Success(`n`, `x`) => error
          }
      }
  }
}
```

State logic is not capable of expressing past time properties. For this we introduce rule-based programming, where facts are added to and deleted from a fact database, which can then queried for contents. The property *Respect Priorities* is expressed in a combination of state logic and rule-based programming:

```
class RespectPriorities extends Monitor[Event] {
  case class P(r1: Resource, r2: Resource) extends Fact
  case class G(r: Resource) extends Fact

  require {
    case Priority (x, y) => P(x, y).+;
    case Grant(x, _) =>
      G(x).+;
      always {
        case Request(_, y) if P(y, x).? &&
          ! factExists {case G(z) => P(z, y).?} =>
          state {
            case Rescind(‘x‘, _) => ok
            case Grant(‘y‘, _) => error
          }
      } upto {
        case Cancel(_, x) => G(x) −
      }
  }
}
```

The monitor declares two facts: P(r1:Resource,r2:Resource), representing that resource *r1* has priority over resource *r2*, and G(r:Resource), representing that resource *r* has been granted. These classes are defined as sub-classes of the class *Fact*, defined in the *Monitor* class. A fact can be added to the database by the suffix $+$ operator, deleted by the $-$ suffix operator, or queried by the suffix $?$ operator. When a resource is granted, we record that fact and then go into an *always*-state. Here, if we see a request for a new resource *y*, which has higher priority than *x* (we see here the use of SCALA's conditional **if**-patterns), and if there does not exist a fact G(z) where *z* has higher priority than *y*, then we disallow a grant of *y* before a rescind of *x*. The *always*-state is active *upto* a cancel of *x* is observed.

Monitors can be hierarchically composed, semantically equivalent to a conjunction:

```
class Requirements extends Monitor[Event] {
  monitor(new ExactlyOneSuccess, new RespectPriorities )
}
```

As illustrated by the following program, a monitor can be fed events by repeated calls of the verify (event: Event) method, optionally followed by a call of the end() method:

```
object TraceAnalysis extends Application {
  val monitor = new Requirements
  monitor. verify (Command("STOP_DRIVING", 245))
  monitor. verify (Success( "STOP_DRIVING", 245))
   ...
  monitor.end()
}
```

Events can either be read from a log file or be generated by a running program properly instrumented. A monitor returns a data structure with results, which can be examined by

```
trait Monitor[Event] extends RuleSystem {
  private var formula: Formula = True
  private var monitors: List[Monitor[Event]] = List()

  def property(f: Formula) {formula = f}
  def monitor(monitorList: Monitor[Event]*) {monitors ++= monitorList.toList}

  trait Formula {
    def apply(e: Event): Formula
     ...
  }

  def verify(e: Event) {
    val formula_ = formula(e)
    if (formula_ == False && formula != False)  reportSafetyError()
    formula = formula_
    for (monitor <- monitors) monitor.verify(e)
    updateFacts()
  }

  def end() { ... }
   ...
}
```

Figure 16. TRACECONTRACT implementation of Monitor.

the calling program, including an error trace for each violated property, consisting of the events that caused the error to occur.

10.3. Implementation

A monitor can contain one formula and a list of sub-monitors, updated with the methods *property* and *monitors* respectively, as shown in Figure 16. Every formula type is a sub-class of the type *Formula*[4]. On each *Formula* is defined a method '`apply(e: Event): Formula`'. Given a formula f, this notation allows us to write $f(e)$ for a given event as a shorthand for $f.apply(e)$. For a given formula f and incoming event e, the expression $f(e)$ denotes the new formula obtained by rewriting f. If the formula becomes false a violation has occurred.

The basic form of a formula is either *True* (also referred to as *ok*) and *False* (also referred to as *error*), each of which stays unchanged when evaluated on a new event, as shown in Figure 17. The more interesting formulas are the state logic formulas, which include for example *state*, *hot*, and *always*, see Figure 18. Each state formula is modeled as a **case** class, which as argument takes a partial function from events to formulas, represented by the type *Block*. A block can be thought of as the set of transitions leading out of the state. A partial function can be tested for defined-ness on an argument. We furthermore introduce a short-hand for defining state logic properties:

```
def require(b: Block) = property(always(b))
```

[4] *Formula* is defined as **trait**, which is a form of abstract class, allowing undefined entities.

```
case object True extends Formula {def apply(e: Event) = this}
case object False extends Formula {def apply(e: Event) = this}
```

Figure 17. TRACECONTRACT implementation of True and False.

```
type Block = PartialFunction [Event,Formula]

case class state (b: Block) extends Formula {
  def apply(e: Event): Formula = if (b.isDefinedAt(e)) b(e) else this
}

case class always(b: Block) extends Formula {
  def apply(e: Event): Formula =
    if (b.isDefinedAt(e)) And(b(e), this ).reduce else this
}
```

Figure 18. TRACECONTRACT implementation of states.

```
def end() {
  if (! isFinal (formula)) reportLivenessError ()
  for (m <− monitors) m.end()
}

def isFinal (f: Formula): Boolean = {
  f match {
     case hot(_)  => false
     case state (_) | always(_) => true
     case Or(f1, f2) => isFinal(f1) || isFinal (f2)
     ...
  }
}
```

Figure 19. TRACECONTRACT implementation of end.

We finally need to define the function *end*, see Figure 19, called when a trace analysis terminates. It determines whether there are any formulas that remain to be satisfied. These represent (bounded) liveness properties.

It remains to explain how rule-based programming is implemented. Recall that a user can define facts as objects of classes that sub-class a class *Fact*, and that these facts can be added to or deleted from a database of facts, as well as queried for presence in the database. The class *RuleSystem* shown in Figure 20 implements this functionality. The variable *facts* contains the current set of facts; while *toRecord* and *toRemove* are used to store facts to be added or to be removed while processing a single event. At the end of processing an event, by a call of *updateFacts*, the variable *facts* is updated by adding and removing these facts respectively. This ensures that rules do not interfere in a non-deterministic manner. The implicit function *fact2FactOps* converts a fact to an object of an anonymous class containing a collection of argument-less methods (suffix operators). For example, if F is a fact, then $F+$ will add F to the database.

```
trait RuleSystem {
  trait Fact
  private var facts : Set[Fact] = Set()
  private var toRecord: Set[Fact] = Set()
  private var toRemove: Set[Fact] = Set()

  implicit def fact2FactOps( fact : Fact) = new {
    def + : Unit = { toRecord += fact }
    def − : Unit = { toRemove += fact }
    def ? : Boolean = facts contains fact
    def ~ : Boolean = !( facts contains fact )
  }

  def updateFacts () {
    toRemove foreach ( facts −= _);  toRecord foreach ( facts += _)
    toRecord = Set (); toRemove = Set()
  }
}
```

Figure 20. TRACECONTRACT implementation of rule system.

11. Summary

The diversity of specification formalisms and their comparisons presented in this tutorial suggests, not too surprisingly, that the silver-bullet specification formalism to express properties for runtime verification does not exist. A specification formalism should combine *efficiency* of the algorithms as well as *expressiveness* and *elegance* of the specification formalism.

Concerning *efficiency*, the most optimal monitors (for JAVA code) can be hard-coded in ASPECTJ, and carefully constructed aspects are usually used as gold-standards for measuring the efficiency on RV systems. In this case the monitors are essentially programmed in JAVA. The most efficient of the mentioned RV specific systems is JAVAMOP, followed by (in that order) TRACEMATCHES, RULER-lite, and TRACECONTRACT.

Concerning *expressiveness*, JAVAMOP appears to be less expressive than TRACEMATCHES as TRACEMATCHES allows several events to be generated by the same join point, which is not the case for JAVAMOP. These systems again are less expressive than RULER-lite and TRACECONTRACT, both of which are Turing complete, as is ASPECTJ. It should be said that JAVAMOP allows code to be executed when events are received as well as when monitors reach certain states. Similarly, TRACEMATCHES is an extension of ASPECTJ, and hence allows code to be executed. In this sense both these systems are Turing complete as well. Here we try to only compare the raw specification formalisms.

Concerning *elegance*, this is partly a subjective criterion, although it includes the concept of conciseness of specifications, a concept that can be measured. JAVAMOP and TRACEMATCHES are the most concise languages within their expressive power. In some cases, however, we have found that "programming" in RULER-lite and TRACECONTRACT can be easier although not as succinct. This is perhaps due to their simple to understand execution models. TRACECONTRACT is an internal DSL, an API, extending SCALA. As a consequence, a user can "dive into" SCALA and program monitors, should the API be too weak for a particular problem. In our experience this is attractive to some users, and has been the main reason for adoption in two real projects at NASA.

In addition to the three dimensions just mentioned, ease of *implementation* is important. TRACECONTRACT likely has the simplest implementation of all the systems discussed. This means that user change requests can be reacted to quickly. The shortest path to a trace analysis tool is likely an API in a high-level programming language, combined with an instrumentation framework such as ASPECTJ.

Runtime verification as a field has several unsolved problems. We do not believe, that there is consensus on what the right formalisms for trace analysis are. Monitoring parameterized events adds complexity that requires existing logics to be adapted. Monitoring algorithms and data structures remain research topics. An interesting topic is how specifications are created, for example how are these techniques integrated with requirements engineering? - how can learning techniques be applied to infer specifications from traces? In general, how does runtime verification integrate with the other parts of the software engineering process, such as for example unit testing.

References

[1] JAVAMOP website. `http://fsl.cs.uiuc.edu/index.php/MOP`.

[2] abc compiler website. `http://www.sable.mcgill.ca/abc/`.

[3] C. Allan, P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. Adding trace matching with free variables to AspectJ. *SIGPLAN Not.*, 40:345–364, October 2005.

[4] H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-based runtime verification. In *VMCAI*, volume 2937 of *LNCS*, pages 44–57. Springer, 2004.

[5] 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.

[6] H. Barringer and K. Havelund. Internal versus external DSLs for trace analysis. In *Proc. of the 2nd Int. Conference on Runtime Verification (RV'11)*, volume 7186 of *LNCS*, pages 1–3. Springer, 2011.

[7] H. Barringer and K. Havelund. TraceContract: a Scala DSL for trace analysis. In *Proc. of the 17th international conference on formal methods*, volume 6664 of *LNCS*, pages 57–72. Springer, 2011.

[8] H. Barringer, D. Rydeheard, and K. Havelund. Rule systems for run-time monitoring: from EAGLE to RuleR. *J Logic Computation*, 20(3):675–706, June 2010.

[9] A. Bauer, M. Leucker, and C. Schallhart. The good, the bad, and the ugly, but how ugly is ugly? In *Proceedings of the 7th international workshop on runtime verification*, volume 4839 of *LNCS*, pages 126–138. Springer, 2007.

[10] 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. Springer, 2009.

[11] 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.

[12] B. Finkbeiner, S. Sankaranarayanan, and H. B. Sipma. Collecting statistics over runtime executions. In *Proceedings of the 2nd international workshop on runtime verification*, pages 36–55. Elsevier, 2002.

[13] K. Havelund and A. Goldberg. Verify your runs. In *Verified Software: Theories, Tools, Experiments, VSTTE 2005*, pages 374–383, 2008.

[14] M. Leucker and C. Schallhart. A brief account of runtime verification. *Journal of Logic and Algebraic Programming*, 78(5):293–303, may/june 2008.

[15] P. Meredith, D. Jin, D. Griffith, F. Chen, and G. Roşu. An overview of the MOP runtime verification framework. *J Software Tools for Technology Transfer*, pages 1–41, 2011.

[16] V. Stolz. Temporal assertions with parametrized propositions. *J. Log. and Comput.*, 20:743–757, June 2010.

[17] V. Stolz and E. Bodden. Temporal assertions using AspectJ. In *Proceedings of the 5th international workshop on runtime verification*, volume 144, pages 109–124, Amsterdam, 2005. Elsevier.

[18] V. Stolz and F. Huch. Runtime verification of concurrent Haskell programs. In *Proceedings of the 4th international workshop on runtime verification*, volume 113, pages 201–216, Amsterdam, 2004. Elsevier.