Verifying Execution Traces

Klaus Havelund
NASA JPL, California Inst. of Technology, USA

Joint work with:
Ylies Falcone
University of Grenoble, France
Giles Reger
University of Manchester, UK

Markoberdorf Summer School, August 2012

Acknowledgements

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.

Copyright 2012. All rights reserved.
Overview of lectures

- Introduction: what is runtime verification (RV)?

- How to manage without an RV system
  - Writing monitors using AspectJ and JAVA.

- Survey of four RV systems with different characteristics:
  - TraceMatches:
    - an extension of AspectJ with regular expressions.
  - JavaMOP:
    - supports many different logics as plugins.
  - Ruler:
    - a form of rule-based programming.
  - TraceContract:
    - an internal DSL extending Scala.

Part I
Introduction to Runtime Verification
System verification

- **Static**: based on complete analysis of code/models of code
  - static analysis / abstract interpretation
  - theorem proving
  - model checking

- **Dynamic**: based on single executions of program/system
  - testing
  - runtime verification
    - a focus of research on analysis of program executions

Attempting a definition of “Runtime Verification”

**Definition (Runtime Verification)**
Runtime Verification is the discipline of computer science dedicated to the analysis of system executions (possibly leveraged by static analysis) by studying specification languages and logics, dynamic analysis algorithms, system instrumentation, and system guidance.

**Definition (Runtime Verification - wider version)**
Runtime Verification is the study of how to get as much out of your runs possible.
Attempting a definition of “Runtime Verification”

**Definition (Runtime Verification)**

Beyond `assert` and `print`.

We focus on runtime verification of user-provided specifications.

**One field - many names**

- Runtime verification
- Runtime monitoring
- Runtime checking
- Runtime analysis
- Dynamic analysis
- Trace/log analysis
- Fault protection
- Runtime enforcement
Runtime verification applications

- Testing
- Fault protection
- Intrusion detection
- Program understanding
- Profiling
- Execution visualization

Testing, runtime verification, fault protection

- runtime verification focuses on the oracle problem
The different approaches compared

<table>
<thead>
<tr>
<th>technique</th>
<th>automated</th>
<th>scalable</th>
<th>coverage</th>
<th>properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>model checking</td>
<td>yes</td>
<td>no</td>
<td>finite</td>
<td>complex</td>
</tr>
<tr>
<td>theorem proving</td>
<td>no</td>
<td>no</td>
<td>complete</td>
<td>complex</td>
</tr>
<tr>
<td>static analysis</td>
<td>yes</td>
<td>yes</td>
<td>complete</td>
<td>simple</td>
</tr>
<tr>
<td>runtime verification</td>
<td>yes</td>
<td>yes</td>
<td>low</td>
<td>complex</td>
</tr>
</tbody>
</table>

Signature:
- automated: once model/program, input, and specification is provided
- scalable: applies to realistic systems without too much pain
- coverage: to what extent all possible executions are explored
- properties: how complex properties can be expressed and how elegantly

System verification

Several attempts at combining static and dynamic analysis. For example:

- **From static to dynamic:**
  - prove as much as possible with static techniques.
  - leave the rest for dynamic techniques.

- **From dynamic to static (a dual view):**
  - decide set of program locations to instrument to drive monitors.
  - use static analysis to reduce that set.

These two views most likely represent the same problem.
Runtime verification in theory

- *Events* record runtime behavior
  - snapshots of state or actions performed

- A finite sequence of events is a *trace* $\tau$

- A *property* $\phi$ denotes a *language* $L(\phi)$ (a set of traces)

- $\tau$ satisfies $\phi$ iff $\tau \in L(\phi)$

Viewing the execution as a *trace*

A trace $\sigma$ is a formal view of a discretized execution:

- a sequence of program’s states
- a sequences of program’s events
- a mix states/events

At any time during execution:

- we are in the present moment *now*
- *past* in known
- *future* is unknown - many possible
Giving verdicts along the way

- Should detect success/failure as soon as possible
- Standard approach is to use \textit{four-valued verdict domain}
- Consider all possible extensions of a trace

<table>
<thead>
<tr>
<th>current trace $\tau$</th>
<th>all suffixes $\sigma$</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 $\tau \in \mathcal{L}(\varphi)$</td>
<td>$\tau \sigma \in \mathcal{L}(\varphi)$</td>
<td>stop with Success $T$</td>
</tr>
<tr>
<td>2 $\tau \in \mathcal{L}(\varphi)$</td>
<td>unknown</td>
<td>carry on monitoring $T_{\text{still}}$</td>
</tr>
<tr>
<td>3 $\tau \notin \mathcal{L}(\varphi)$</td>
<td>$\tau \sigma \notin \mathcal{L}(\varphi)$</td>
<td>stop with Failure $F$</td>
</tr>
<tr>
<td>4 $\tau \notin \mathcal{L}(\varphi)$</td>
<td>unknown</td>
<td>carry on monitoring $F_{\text{still}}$</td>
</tr>
</tbody>
</table>

Runtime verification in practice

- Start with a system to monitor.
Runtime verification in practice

- *Instrument* the system to record relevant events.

- *Provide* a monitor.
Runtime verification in practice

- Dispatch each received event to the monitor.

Runtime verification in practice

- Compute a verdict for the trace received so far.
Runtime verification in practice

- Possibly generate *feedback* to the system.

Runtime verification in practice

- We might possibly have synthesized monitor from a *property*. 

![Diagram](image-url)
Generating the trace

The concrete execution of the program needs to be abstracted:

- **Discrepancy:**
  - events of the program in $\Sigma_c$
  - events of the specification in $\Sigma_a$

- Define a function $\Sigma_c \rightarrow \Sigma_a \cup \{\text{irrelevant}\}$

- It is the role of the **instrumentation** phase

Monitor Placement: how the monitor is integrated

- **Offline:** the trace is analyzed *aposteriori*
  e.g., analyzing log file/trace dump

- **Online:** the trace is analyzed in a *lock-step* manner
  - **external:** monitor runs in parallel with the system e.g., communication over pipes, sockets
    - synchronous (system waits for response)
    - asynchronous (buffered communication)
  - **internal:** monitor’s code is embedded into the application
Monitor placement

About reaction

Reaction can take several forms:

1. Display an error message
2. Throw an exception in the monitored program, and monitored program then deals with it
3. Launch some (recovery) code: the effect depends on monitor’s placement
How is the monitor specified?

- Program (built-in algorithm focused on specific problem)
  - data race detection
  - atomicity violation
  - deadlock detection
- Programming language
- Design by contract (pre/post conditions), JML for example
- Logic (formal system)
  - state machines
  - regular expressions
  - grammars (context free languages)
  - temporal logic (past time, future time)
  - rule-based logics

From propositional to parametric monitoring

- Field started with *propositional* monitoring
  - events are just strings
- Recently moved to *parametric* monitoring
  - events carry data values
- Solutions exist spanning the two classical dimensions
  - Expressiveness of specification language
  - Efficiency of monitoring algorithm
- We shall see solutions in both dimensions.
The propositional approach: an example

- Record *propositional* events, for example
  - open, close
- Define a property over propositional events, for example
  - LTL (finite-trace) \(\Box (\text{open} \rightarrow \bigcirc (\neg \text{open} \cup \text{close}))\)
  - RE (open.close)*
  - DFA
- Check if each trace prefix is in the language of the property

---

Using Projection

- With the property
- Take the trace

\[\text{open.read.write.close.open.read.close}\]

- What do we do with read and write?
- Filter out irrelevant events / Project on relevant events

\[\text{open.close.open.close}\]
Going parametric

- Consider the code
  
  ```java
  File f1 = new File("manual.pdf");
  File f2 = new File("readme.txt");
  f1.open();
  f2.open();
  f2.close();
  f1.close();
  ```

- Say we just focus on propositional events
  
  `open.open.close.close`

- Not good, we want to *parameterize* events with data values and use those values in the specification

- Instead record the parametric trace
  

---

Parametric properties

- Using the events
  
  - `open(f)` when file f is opened
  - `close(f)` when file f is closed

- the property becomes

    ![Diagram](image-url)
From parametric to quantified

Quantify over variables in parametric property:

\[ \forall f \quad \text{open}(f) \quad \text{close}(f) \]

Some instrumentation techniques

Manual:
- assertions
- pre/post conditions in design by contract solutions

Automated:
- instrumentation of source code
  - CIL (C) [http://sourceforge.net/projects/cil](http://sourceforge.net/projects/cil)
- instrumentation of byte/object code
  - Valgrind (C) [http://valgrind.org](http://valgrind.org)
  - BCEL (Java) [http://jakarta.apache.org/bcel](http://jakarta.apache.org/bcel)
- aspect-oriented programming (AOP):
  - AspectC (C) [https://sites.google.com/a/gapp.msrg.utoronto.ca/aspectc](https://sites.google.com/a/gapp.msrg.utoronto.ca/aspectc)
  - AspectC++ (C++) [http://www.aspectc.org](http://www.aspectc.org)
  - AspectJ (Java) [http://www.eclipse.org/aspectj](http://www.eclipse.org/aspectj)
Detection of concurrency errors

Debugging is hard to achieve on multi-threaded systems, due to:
- the large number of possible behaviors
- the difficulty to establish causality between events

An RV approach is to use predictive analysis:
- Turn a hard to test property into an easy to test property (footprints).
- Violation is only suggestive, indicating the potential for an error in some other trace of the monitored system.

\[ \text{report}(\sigma) \Rightarrow \exists \sigma' \in \text{Exec(}System) : \text{error}(\sigma') \ldots \text{or not} \ldots \]

Existing approaches for detection of deadlocks, dataraces, atomicity errors

Challenges
- Code instrumentation
- Definition of specification languages
  - expressive
  - elegant
- Synthesis of efficient monitors
- Low impact monitoring
- Integration of static and dynamic analysis
- How to control application in case of violation/validation
- Programming language design that supports RV
- Learning specifications from traces
- Program visualization
- ...
Summary

- RV aims to answer the word problem for executions of a program wrt. a specification.
- Also sometimes coined the oracle problem.
- Efficient monitoring of parametric properties is a main challenge.
- Practical and effective technique.
- Growing community with a lot of research opportunities.

Recap

Last lecture we looked at

- General definition of what runtime verification is.
- Decided to focus on checking executions against formalized requirements.

- Challenges:
  - Instrumentation techniques.
  - Specification languages for writing monitors. Propositional versus parametric monitors.
In this lecture

- We will demonstrate how program monitors can be written in Java using AspectJ for code instrumentation
- A through-going example: a planetary rover
- Design-by-contract: programming with pre- and post-conditions
- From design-by-contract to temporal specifications
- Code instrumentation with AspectJ: separating concerns
- Specification with Java: while instrumenting with AspectJ

Rover example

- A rover contains various instruments, each represented as a task
- **Command execution:**
  - Command sequences are received from ground
  - Commands get dispatched to instruments
  - Commands either succeed or fail on instruments
- **Resource management:**
  - Tasks need resources, and sends resource requests to an arbiter
  - Some resources can be in conflict and some have higher priority
  - The arbiter can grant or deny resources
  - Or ask tasks to rescind (cancel) resources if another task asks for higher priority resources.
- **File system:**
  - Results get stored in file system
  - Logged data gets sent back to earth
Example of resource acquisition sequence
Design-by-Contract of a method

- We first consider a method for requesting a resource.
- We will specify its pre- and post-condition.
- Subsequently we shall discuss the limitations of pre/post conditions.

The requestResource method
The requestResource method

- An actor (task) calls this method when requesting a resource by name.
- The object representing the resource is looked up.
- The method declares a result variable representing the response, updates it, and finally returns it.

```java
1  Response requestResource(Actor actor, String name) {
2      Resource resource = resources.get(name);
3      Response result = null;
4      ...
5      return result;
6  }
```

The returned result is of type Response

```java
1  public class Response {
2      private Resource resource = null;
3      private List<RescindOrder> rescinds =
4          new ArrayList<RescindOrder>();
5
6  public Response(Resource resource,
7          List<RescindOrder> rescinds) {
8          this.resource = resource;
9          this.rescinds = rescinds;
10     }
11
12  public Resource getResource() {
13      return resource;
14  }
15
16  public List<RescindOrder> getRescinds() {
17      return rescinds;
18  }
19  }
```
Informal requirement statement

```java
Response requestResource(Actor actor, String name) {
    Resource resource = resources.get(name);
    Response result = null;
    ...
    return result;
}
```

**Requirement CorrectResponse**

- **Pre-condition:** *the name should correspond to an existing resource.*
- **Post-condition:** *the Response object returned by requestResource must be well-formed. For example:*
  ```java
  result.getResource() != null and result.getRescinds() == null
  iff: the resource is not owned by another task, and it is not in conflict with other resources.
  ```

Design by contract with JML (Java Modeling Language)

```java
/*@ requires resources.containsKey(name);
@ ensures
@ \result.getResource() != null &&
@ \result.getRescinds() == null
@ ==
@ \exists Resource resource;
  resource == resources.get(name) &&
  \result.getResource() == resource &&
  \old(isAvailable(resource)) &&
  \old(getActiveConflicts(resource).isEmpty()));
@*/
```

```java
Response requestResource(Actor actor, String name) {
    Resource resource = resources.get(name);
    Response result = null;
    ...
    return result;
}
```
Pre- and post-conditions expressed as assertions

- Most programming languages, however, do not explicitly support DBC. So we use assertions.
- Post-condition is complex, so we call a post-condition method.

```java
Response requestResource(Actor actor, String name) {
    assert resources.containsKey(name);
    Resource resource = resources.get(name);
    boolean oldIsAvailable = isAvailable(resource);
    List<Resource> oldInConflict = getActiveConflicts(resource);
    Response result = null;
    ...
    assert post_requestResource(actor, name, oldIsAvailable, oldInConflict, result);
    return result;
}
```

The post-condition function

- Parameterized with:
  - arguments to original function (actor, name)
  - old values of variables needed in post-condition (old...))
  - returned value of original function (result)

```java
private boolean post_requestResource(
    Actor actor, String name,
    boolean oldIsAvailable, List<Resource> oldInConflict,
    Response result)
{
    Resource resultResource = result.getResource();
    List<RescindOrder> resultRescinds = result.getRescinds();

    Resource resource = resources.get(name);

    if (resultResource != null && resultRescinds == null)
        return oldIsAvailable && oldInConflict.isEmpty();
    ...
}
```
Design By Contract: discussion

- Assertions only check current state.

- DBC, like JML, also allows reference to previous state (old(...)). In addition DBC also supports checking invariants, for example at method boundaries.

- If we want to check temporal properties we need to build data structures to represent - at any point in the execution the following information:
  - the past: selected facts about what happened so far in the execution.
  - the future: obligations indicating what should and what should not happen in the future execution.

The 2 monitor dimensions: temporal and remoteness

<table>
<thead>
<tr>
<th></th>
<th>temporal assertions</th>
<th>design by contract</th>
<th>temporal logic</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>assert</td>
<td>JML</td>
<td>JML + TL</td>
</tr>
<tr>
<td>remoteness</td>
<td>internal to code</td>
<td>external to code</td>
<td></td>
</tr>
<tr>
<td></td>
<td>AspectJ + assert</td>
<td>AspectJ + assert</td>
<td>AspectJ + assert + history</td>
</tr>
</tbody>
</table>

- In the rest of this lecture we shall study how to write monitors external to the code in AspectJ.
Aspect-oriented programming

- An alternative module concept compared to object oriented programming.
- Emphasis on separating cross-cutting concerns. Logging for example. That is, code for one aspect of the program is collected together in one place.
- We use it for monitoring, and do not focus on the broader application of AOP as a programming paradigm.
- Key idea: define hooks into program and indicate code to be executed when they are reached during execution.
- Enables to capture data as well, such as method arguments and returned results.

The problem with object-oriented programming

- code tangling: one module handles many concerns.
  - Flow of core logic gets obscured.

- code scattering: one concern is handled in many modules.
  - lots of typing, searching, ...
  - increases probability of consistency errors
  - big picture is missing

Example: logging
Monitoring-oriented examples of cross-cutting code

- Logging (tracking program behavior)
- Verification (checking program behavior)
- Policy enforcement (correcting behavior)
- Security management (preventing attacks)
- Profiling (exploring where a program spends its time)
- Visualization (of program executions)

Aspect-oriented programming systems

- **AspectJ** - for **Java**:
  http://www.eclipse.org/aspectj
- **AspectC++** - for **C++**:
  http://www.aspectc.org
- **ACC** - for **C**:
  https://sites.google.com/a/gapp.msrg.utoronto.ca/aspectc
- **Arachne** - for **C**:
  http://www.emn.fr/x-info/arachne/index.html
- **Aspicere** - for **C**:
  http://mcis.polymtl.ca/~bram/aspicere
- **InterAspect** - for **GCC**:
  http://www.fsl.cs.stonybrook.edu/interaspect
Aspect-oriented programming with AspectJ

- Url: http://www.eclipse.org/aspectj
- Works well with Eclipse: http://www.eclipse.org

- An extension of JAVA.
- Launched 1998 at Xerox PARC.
- The AspectJ compiler is free and open source, very mature.
- Outputs .class files compatible with the JVM.

Hello world AspectJ example

```java
public class Hello {
    public static void main(String[] args) {
        System.out.println("this is");
        System.out.println("hello");
    }
}

public aspect World {
    after(String s) :
        call(void java.io.PrintWriter.print(String))
        && args(s)
        && if (s.equals("hello"))
            { System.out.println("world"); }
}

this is
hello
world
```
Hello world AspectJ example

```java
public class Hello {
    public static void main(String[] args) {
        System.out.println("this is");
        System.out.println("hello");
    }
}

public aspect World {
    after(String s) : // advice
        call(void java.io.PrintWriter.println(String)) // pointcut
        && args(s) // pick up argument
        && if (s.equals("hello")) // condition
    {
        System.out.println("world"); // do this after such a call
    }
}
```

The core idea

- Use AspectJ to instrument code.
- Use Java to fill in data declarations and advice bodies to perform monitoring.
AspectJ terminology

- **join point**: point in a program that one can “join on”
- **pointcut**: specifies a set of join points + picks out values at those points
- **advice**:
  - pointcut
  - + advice code to be inserted
  - + insertion point (before, after, around)
- **aspect**: a modular unit for cross cutting behavior
  - normal Java definitions (fields, methods)
  - + list of pointcut definitions
  - + advice definitions

Three kinds of advice

- **before(...)**: `<pointcut> { <adviceCode> }`
  inserts *advice code* before join points that match pointcut.

- **after(...)**: `<pointcut> { <adviceCode> }`
  inserts *advice code* after join points that match pointcut.

- **T around(...)**: `<pointcut> { <adviceCode> }`
  replaces join points that match pointcut with *advice code*.

In the *advice code* the **proceed(...)** construct represents the original matching join point, taking arguments and returning a value if it is a method call.
Recall pre- and post-condition for method requestResource

```java
/*@ requires resources.containsKey(name);
@ ensures 
(\result.getResource() != null && \
\result.getRescinds() == null) 
== 
(\exists Resource resource;
 resource == resources.get(name) && \
\result.getResource() == resource && \
old(isAvailable(resource)) && \
old(getActiveConflicts(resource).isEmpty())); 
@*/

1 Response requestResource(Actor actor, String name) {
2 Resource resource = resources.get(name);
3 Response result = null;
4 ... 
5 return result;
6 }
```
Same pre- and post-condition as an aspect

```java
privileged public aspect PrePostRequestResource {
  ...
  ...
  pointcut requestResource(ResourceTable table, String name) :
    call(Response ResourceTable.requestResource(Actor, String))
    && args(∗, name) && target(table);

  before(ResourceTable table, String name) :
    requestResource(table, name) {
      ...
    }

  after(ResourceTable table, String name)
  returning (Response result) :
    requestResource(table, name) {
      ...
    }
}
```

Declaring data in aspect

```java
privileged public aspect PrePostRequestResource {
  ...
  ...
  Resource resource;
  boolean oldIsAvailable;
  List<Resource> oldInConflict;

  pointcut requestResource(ResourceTable table, String name) :
    call(Response ResourceTable.requestResource(Actor, String))
    && args(∗, name) && target(table);

  before(ResourceTable table, String name) :
    requestResource(table, name) {
      ...
    }

  after(ResourceTable table, String name)
  returning (Response result) :
    requestResource(table, name) {
      ...
    }
}
```
The pre-condition, and preparing for post-condition

```java
privileged public aspect PrePostRequestResource {
    Resource resource;
    boolean oldIsAvailable;
    List<Resource> oldInConflict;

    pointcut requestResource(ResourceTable table, String name) :
        call(Response ResourceTable.requestResource(Actor, String))
        && args(∗, name) && target(table);

    before(ResourceTable table, String name) :
        requestResource(table, name) {
            assert table.resources.containsKey(name);
            resource = table.resources.get(name);
            oldIsAvailable = table.isAvailable(resource);
            oldInConflict = table.getActiveConflicts(resource);
        }

    after(ResourceTable table, String name) . . . { . . . }
}
```

The post-condition

```java
privileged public aspect PrePostRequestResource {
    Resource resource;
    boolean oldIsAvailable;
    List<Resource> oldInConflict;

    pointcut requestResource(ResourceTable table, String name) :
        call(Response ResourceTable.requestResource(Actor, String))
        && args(∗, name) && target(table);

    before(ResourceTable table, String name) . . . { . . . }

    after(ResourceTable table, String name)
        returning (Response result) :
            requestResource(table, name) {
                if (result.getResource() != null && result.getRescinds() == null) {
                    assert oldIsAvailable && oldInConflict.isEmpty();
                }
            }
}
```
To sum up

- We saw how a pre- and post-conditions can be specified using AspectJ and Java.
- Although somewhat verbose it does separate code from specification, which becomes essential if substantial specifications are written.
- We shall now extend this approach to temporal properties.
- For this purpose we shall illustrate:
  - The join points were are interested in for temporal specification.
  - An aspect Instrument that instruments the program to generate events at those points.
  - A class Monitor that every specific monitor shall extend (sub-class).
  - Selected properties of resource management defined as sub-classes of class Monitor.

Join points (red and underlined) of interest

- The arbiter receives, processes and sends messages.

```java
public class Arbiter extends Actor {
    ResourceTable table = new ResourceTable();

    public void run() {
        while (true) {
            Message message = receive();
            if (message instanceof Request) {
                Response response = table.requestResource(requester, resourceName);
                . . .
                if (resource == null && rescinds == null) {
                    requester.sendDeny();
                } else if (rescinds == null) {
                    requester.sendGrant(resource);
                } else . . .
            } else else if (message instanceof Cancel) { . . . }
        }
    }
}
```
The instrumentation aspect

```java
privileged aspect Instrument {
  // List of all monitors to be notified on each new event
  List<Monitor> monitors = new ArrayList<Monitor>();

  // Creating the list of monitors in aspect constructor
  public Aspect() {
    monitors.add(new monitors.GrantCancel(), ...);
  }

  // Advice for the sendGrant event
  after(Resource resource, Actor receiver) :
    call(void missioncontrol.Task+.sendGrant(Resource))
      && args(resource) && target(receiver)
    {
      // activate event on each monitor (not efficient)
      for (Monitor monitor : monitors)
        monitor.sendGrant(resource, receiver);
    }

  // Advice for other events
}
```

The class Monitor

- Every specific monitor must extend class `Monitor`.
- Each event is represented by a method that will get called from the instrumentation aspect when the corresponding join point is reached.
- Method bodies are empty. An extending specific monitor must override methods relevant to the property.

```java
public class Monitor extends MonitorUtils {
  void sendRequest(Actor sender, String resource) {};
  void sendRequest(Actor sender, Resource resource) {};
  void sendCancel(Actor sender, Resource resource) {};
  void sendGrant(Resource resource, Actor receiver) {};
  void sendRescind(Resource resource, Actor receiver) {};
  void sendDeny(Actor receiver) {};
  void addConflict(Resource resource1, Resource resource2) {};
  void addPriority(Resource resource1, Resource resource2) {};
  void requestResource(Actor actor, String name,
                       Response response) {};
  void cancelResource(Actor actor, Resource resource) {};
  void end() {}
}
```
The class *MonitorUtils*

- Auxiliary methods used for writing monitors.

```java
public class MonitorUtils {

    protected void verify(boolean condition, String message) {
        if (!condition) error(message);
    }

    protected void error(String message) {
        System.out.println("***error:");
    }

    protected void fail(String message) {
        try {
            throw new RuntimeException(message);
        } catch (Exception e) {
            e.printStackTrace();
        }
        System.exit(0);
    }
}
```

JavaDoc documentation of methods representing events

<table>
<thead>
<tr>
<th>Method Summary</th>
<th>Monitor</th>
<th>Constructor Summary</th>
<th>Monitor()</th>
</tr>
</thead>
<tbody>
<tr>
<td>addConflict(Resource resource1, Resource resource2)</td>
<td>Monitor</td>
<td>Represents the addition of a conflict between two resources.</td>
<td></td>
</tr>
<tr>
<td>addPriority(Resource resource1, Resource resource2)</td>
<td>Monitor</td>
<td>Represents the addition of a priority between two resources.</td>
<td></td>
</tr>
<tr>
<td>end()</td>
<td>Monitor</td>
<td>Signals the end of the program.</td>
<td></td>
</tr>
<tr>
<td>executeAction(actor, java.lang.String name, Resource resource)</td>
<td>Monitor</td>
<td>Represents a call of the method executeAction().</td>
<td></td>
</tr>
<tr>
<td>executeForcedAction(actor, java.lang.String name, Resource resource)</td>
<td>Monitor</td>
<td>Represents a call of the method executeForcedAction().</td>
<td></td>
</tr>
<tr>
<td>executeForcedTask(actor, java.lang.String name, Resource resource)</td>
<td>Monitor</td>
<td>Represents a forced task on a resource.</td>
<td></td>
</tr>
<tr>
<td>executeTask(actor, java.lang.String name, Resource resource)</td>
<td>Monitor</td>
<td>Represents a task on a resource.</td>
<td></td>
</tr>
<tr>
<td>freeResource(ResourceManager resource)</td>
<td>Monitor</td>
<td>Represents the task executing a resource.</td>
<td></td>
</tr>
<tr>
<td>freeResource(ResourceManager resource, Action resource)</td>
<td>Monitor</td>
<td>Represents a resource being freed.</td>
<td></td>
</tr>
<tr>
<td>freeTask(ResourceManager resource, Action resource)</td>
<td>Monitor</td>
<td>Represents a task being freed.</td>
<td></td>
</tr>
<tr>
<td>hasConflict()</td>
<td>Monitor</td>
<td>Represents a conflict that has been detected.</td>
<td></td>
</tr>
<tr>
<td>hasConflict(Resource resource1, Resource resource2)</td>
<td>Monitor</td>
<td>Represents a conflict between two resources.</td>
<td></td>
</tr>
<tr>
<td>hasPriority()</td>
<td>Monitor</td>
<td>Represents a priority between two resources.</td>
<td></td>
</tr>
<tr>
<td>hasTask()</td>
<td>Monitor</td>
<td>Represents a task that has been detected.</td>
<td></td>
</tr>
<tr>
<td>restart()</td>
<td>Monitor</td>
<td>Represents the restart of the program.</td>
<td></td>
</tr>
<tr>
<td>restart()</td>
<td>Monitor</td>
<td>Represents the restart of the program.</td>
<td></td>
</tr>
<tr>
<td>restart()</td>
<td>Monitor</td>
<td>Represents the restart of the program.</td>
<td></td>
</tr>
</tbody>
</table>

```

Class Monitor

- java.lang.Object
- Monitor

Direct Known Subclasses
- ConsensusMonitor, MonitorGeneral, RequestControl, RequestPriority, UnilateralResponse

public class Monitor extends BoundedBuffer

All classes that define a monitor must subclass this class. The class provides a method for each kind of event monitored. These methods have empty bodies. If a monitor needs some action to be performed when a method corresponding to an event is called, it needs to override the method.

Constructor Summary

Monitor()
Let us specify some properties over these events

- **GrantCancel**: For a given resource, grants and cancellations should alternate, starting with a grant. Furthermore: a cancellation should be performed by the same task that was last granted the resource.

- **OnlyRescindGranted**: Only ask a task to rescind the resource if it is currently owned by the task. That is: it has been granted, and it has not yet been cancelled.

- **RespectConflicts**: Conflicts must be respected. For every pair of resources, if they conflict then only one can be granted at any one time.

- **RespectPriorities**: Let priorities sort conflicts. If there is a conflict and the requested resource has the highest priority then the other priority should be rescinded before the resource is granted.
Resource Management: grant and cancel should alternate

**Requirement GrantCancel**

For a given resource, grants and cancellations should alternate, starting with a grant. Furthermore: a cancellation should be performed by the same task that was last granted the resource.

Granting and cancelling resources
Algorithm for checking GrantCancel

- **Declarations:**
  - `allocated`: map from resource to actor if allocated by the actor.

- **Operations:**
  - When `r` is granted to an actor `a`:
    - Verify that \( \text{allocated}(r) = \text{undefined} \).
    - \( \text{allocated}(r) := a \).
  - When `a` cancels `r`:
    - Verify that \( \text{allocated}(r) = a \).
    - \( \text{allocated}(r) := \text{undefined} \).

---

The GrantCancel Monitor

```java
public class GrantCancel extends Monitor {
    HashMap<Resource, Actor> allocated = new HashMap<Resource, Actor>();

    @Override
    void sendGrant(Resource resource, Actor receiver) {
        verify(!allocated.containsKey(resource));
        allocated.put(resource, receiver);
    }

    @Override
    void sendCancel(Actor sender, Resource resource) {
        verify(allocated.get(resource) == sender);
        allocated.remove(resource);
    }
}
```
Resource Management: only rescind granted

**Requirement OnlyRescindGranted**

Only ask a task to rescind the resource if it is currently owned by the task. That is: it has been granted, and it has not yet been cancelled.

Granting, rescinding and cancelling resources
Algorithm for checking OnlyRescindGranted

- **Declarations:**
  - *allocated*: map from resource to actor if allocated by the actor.

- **Operations:**
  - When *r* is granted to an actor *a*:
    * \( \text{allocated}(r) := a \).
  - When *a* cancels *r*:
    * \( \text{allocated}(r) := \text{undefined} \).
  - When a rescind message is sent to an actor *a* to cancel *r*:
    * Verify that \( \text{allocated}(r) = a \).

---

The OnlyRescindGranted Monitor

```java
public class OnlyRescindGranted extends Monitor {
    HashMap<Resource, Actor> allocated = new HashMap<Resource, Actor>();

    @Override
    void sendGrant(Resource resource, Actor receiver) {
        allocated.put(resource, receiver);
    }

    @Override
    void cancelResource(Actor actor, Resource resource) {
        allocated.remove(resource);
    }

    @Override
    void sendRescind(Resource resource, Actor receiver) {
        verify(allocated.get(resource) == receiver);
    }
}
```
Resource Management: respect conflicts

**Requirement RespectConflicts**

Conflicts must be respected. For every pair of resources, if they conflict then only one can be granted at any one time.

Granting and cancelling resources with conflicts
Algorithm for checking RespectConflicts

- **Declarations:**
  - `conflicts`: map from resource to set of resources it is in conflict with.
  - `allocated`: map from resource to actor if allocated by the actor.

- **Operations:**
  - When the pair $(r_1, r_2)$ are declared as conflicting:
    - Add $r_2$ to the set $conflicts(r_1)$ and vice versa.
  - When $r$ is granted to an actor $a$:
    - Verify that no resource in set $conflicts(r)$ is in domain of $allocated$.
    - $allocated(r) := a$.
  - When $a$ cancels $r$:
    - Verify that $allocated(r) = a$.
    - $allocated(r) := undefined$.

The RespectConflicts Monitor 1

```java
public class RespectConflicts extends Monitor {
  MapToSet<Resource, Resource> conflicts = new MapToSet<Resource, Resource>();
  Map<Resource, Actor> allocated = new HashMap<Resource, Actor>();

  @Override
  void addConflict(Resource resource1, Resource resource2) {
    conflicts.getSet(resource1).add(resource2);
    conflicts.getSet(resource2).add(resource1);
  }
}
```
The RespectConflicts Monitor 2

```java
public class RespectConflicts extends Monitor {
    MapToSet<Resource, Resource> conflicts = ...
    Map<Resource, Actor> allocated = ...

    @Override
    void sendGrant(Resource resource, Actor receiver) {
        Set<Resource> intersection =
            intersect(conflicts.getSet(resource), allocated.keySet());
        verify(intersection.isEmpty());
        allocated.put(resource, receiver);
    }

    @Override
    void sendCancel(Actor actor, Resource resource) {
        verify(allocated.get(resource) == actor);
        allocated.remove(resource);
    }
}
```

Resource Management: respect priorities

**Requirement RespectPriorities**

Let priorities sort conflicts. If there is a conflict and the requested resource has the highest priority then the other priority should be rescinded before the resource is granted.
Requesting, rescinding, cancelling and granting resources with prioritized conflicts

Algorithm for checking RespectPriorities

- **Declarations:**
  - \( \text{conflicts} \): map from resource to set of resources it is in conflict with.
  - \( \text{priorities} \): map from resource to set of resources with lower priority.
  - \( \text{allocated} \): map from resource to actor if allocated by the actor.
  - \( \text{toRescind} \): map from resource \( r \) to the pairs \( (r', a') \) of resources \( r' \) that have to be cancelled by actor \( a' \) before the resource \( r \) can be granted.

- **Operations:**
  - When the pair \( (r_1, r_2) \) are declared as conflicting:
    * Add \( r_2 \) to the set \( \text{conflicts}(r_1) \) and vice versa.
  - When \( r_1 \) is declared as having higher priority than \( r_2 \):
    * Add \( r_2 \) to the set \( \text{priorities}(r_1) \).
  - When \( r \) is granted to an actor \( a \):
    * Verify that \( \text{toRescind}(r) \) is empty.
    * \( \text{allocated}(r) := a \).
  - When \( a \) cancels \( r \):
    * \( \text{allocated}(r) := \text{undefined} \).
  - When \( a \) requests \( r \):
    * If winning conflict, add loosing conflicts to \( \text{toRescind} \).
  - When an actor \( a \) is asked to rescind \( r \):
    * Verify that the pair \( (a, r) \) is mapped to by some resource in \( \text{toRescind} \).
    * Remove the pair \( (a, r) \) from \( \text{toRescind} \).
The RespectPriorities Monitor 1

```java
public class RespectPriorities extends Monitor {
    MapToSet<Resource, Resource> conflicts = ...;
    MapToSet<Resource, Resource> priorities = ...;
    HashMap<Resource, Actor> allocated = ...;
    MapToSet<Resource, Pair<Actor, Resource>> toRescind = ...;

    @Override
    void addConflict(Resource resource1, Resource resource2) {
        conflicts.getSet(resource1).add(resource2);
        conflicts.getSet(resource2).add(resource1);
    }

    @Override
    void addPriority(Resource resource1, Resource resource2) {
        priorities.getSet(resource1).add(resource2);
    }
}
```

The RespectPriorities Monitor 2

```java
public class RespectPriorities extends Monitor {
    MapToSet<Resource, Resource> conflicts = ...;
    MapToSet<Resource, Resource> priorities = ...;
    HashMap<Resource, Actor> allocated = ...;
    MapToSet<Resource, Pair<Actor, Resource>> toRescind = ...;

    @Override
    void sendGrant(Resource resource, Actor actor) {
        Set<Pair<Actor, Resource>> rescinds = toRescind.getSet(resource);
        verify(rescinds.isEmpty());
        allocated.put(resource, actor);
    }

    @Override
    void cancelResource(Actor actor, Resource resource) {
        allocated.remove(actor);
    }
}
```
The RespectPriorities Monitor 3

@Override void sendRequest(Actor sender, Resource resource) {
    Set<Resource> conflicting =
        intersect(conflicts.getSet(resource), allocated.keySet());
    if (!conflicting.isEmpty()) {
        boolean winningConflict = false;
        for (Resource conflict : conflicting) {
            if (priorities.get(resource).contains(conflict))
                winningConflict = true;
            if (priorities.get(conflict).contains(resource)) {
                winningConflict = false; break;
            }
        } // a conflict is winning if none have higher priority and this has higher priority than at least one
        if (winningConflict) {
            Set<Pair<Actor, Resource>> rescinds =
                new HashSet<Pair<Actor, Resource>>();
            for (Resource conflict : conflicting)
                rescinds.add(new Pair<Actor, Resource>
                    (allocated.get(conflict), conflict));
            toRescind.put(resource, rescinds);
        }
    }
}

The RespectPriorities Monitor 4

public class RespectPriorities extends Monitor {
    MapToSet<Resource, Resource> conflicts = ...;
    MapToSet<Resource, Resource> priorities = ...;
    HashMap<Resource, Actor> allocated = ...;
    MapToSet<Resource, Pair<Actor, Resource>> toRescind = ...;

    @Override
    void sendRescind(Resource resource, Actor receiver) {
        for (Pair<Actor, Resource> pair : toRescind.keySet()) {
            Set<Pair<Actor, Resource>> rescinds =
                toRescind.get(resource);
            for (Pair<Actor, Resource> pair : rescinds) {
                if (pair.second == resource) {
                    verify(pair.first == receiver);
                    rescinds.remove(pair);
                    break;
                }
            }
        }
    }
}
Summary

- AspectJ is powerful for instrumenting code.
- AspectJ extends Java, and Java can therefore be used for specifying properties.
- This combination works!
- However, at times it requires a lot of code to write down properties. We have to invent internal data structures and update them on each incoming event.
- The challenge is whether we can provide a simpler way of writing properties.

References

Recap

- We have looked at the general principles behind Runtime Verification.
- We have looked at how to specifying programs using Java and monitor them using Aspectj.
Parametric runtime verification

- Propositional Runtime Verification for Finite-State properties can be carried out using Finite State Automata over a set of states $S$.
- Given a set of events $\Sigma$, the next state is computed using a transition function $\delta \in (\Sigma \times S) \rightarrow S$.
- In Parametric Runtime Verification we consider events carrying data values drawn from a set of objects $O$.
- We wish to associate a monitor with each set of related objects.
- Therefore, these objects should take part in the transition function, which we may be tempted to write as

$$\delta \in (\Sigma \times 2^O \times S) \rightarrow S$$

- However, using the parameter object values in the transition function in this way is ambiguous.
- For example, we want to be able to differentiate between the events $\text{priority}(\text{wheels}, \text{camera})$ and $\text{priority}(\text{camera}, \text{wheels})$. 
Producing bindings

- Instead of using the set of objects in a parametric events we construct bindings - which give unique names to parameter object values.
- Let $\text{Bind} = \text{Var} \rightarrow \mathcal{O}$ be the set of all bindings (partial maps), given some set of variables $\text{Var}$.
- Let $\text{name} : (\Sigma \times \mathcal{O}^*) \rightarrow \text{Bind}$ be a function that creates bindings from a parametric event, how this is implemented is not important here.
- For example, a possible implementation of $\text{name}$ might give
  
  $\text{name}(\text{priority}(\text{wheels}, \text{camera})) = [r_1 \mapsto \text{wheels}, r_2 \mapsto \text{camera}]$
  
  $\text{name}(\text{priority}(\text{camera}, \text{wheels})) = [r_1 \mapsto \text{camera}, r_2 \mapsto \text{wheels}]$

- It is now possible to differentiate between events $\text{priority}(\text{wheels}, \text{camera})$ and $\text{priority}(\text{camera}, \text{wheels})$.
- The parametric transition function should therefore be:
  
  $\delta \in (\Sigma \times \text{Bind} \times S) \rightarrow S$

Different approaches

- For efficiency reasons, it is desirable to organise the computation of $\delta$, and there are three different approaches to this:
  
  - Object-based
    
    $\delta \in \text{Bind} \rightarrow (S \times \Sigma) \rightarrow S$

  - State-based
    
    $\delta \in S \rightarrow (\Sigma \times \text{Bind}) \rightarrow S$

  - Event-based
    
    $\delta \in \Sigma \rightarrow (S \times \text{Bind}) \rightarrow S$

- Each approach requires the monitoring process to be structured in a different way, leading to different implementations and potential for optimisations.
- In this section we discuss TraceMatches, a state-based approach.
- In the next section we discuss JavaMOP, an object-based approach.
TraceMatches : An Overview

What is TraceMatches?
- An extension of the AspectJ language.
- Was first introduced in a 2005 OOPSLA paper.
- Implemented in the abc compiler.

What are its defining principles?
- Allows a user to write properties involving the history of computation.
- Uses *regular expressions* over *pointcuts*.
- Uses free variables in events to capture parameters.

Syntax

A tracematch consists of:
- (free) variable declarations
- symbol declarations (using pointcuts)
- a regular expression over symbols
- a piece of Java code to be executed on a match
Availability

- **TraceMatches** is available as an extension to the abc compiler found at [http://www.sable.mcgill.ca/abc/](http://www.sable.mcgill.ca/abc/).

Executing TraceMatches

One approach to weaving a tracematch into your code is:

- compile code to be woven into a folder at bin
- (download and) place abc jars into a folder at abc_home_path
- place source for tracematch(s) into a folder at src/tracematch
- run the command

```java
java -classpath "abc_home_path/abc-complete.jar;bin"
   -Xmx256M -Dabc.home=abc_home_path abc.main.Main
   -ext abc.tm -source 1.5 -d bin
   -inpath bin -sourceroots src/tracematch
```

- This weaves the tracematches into the compiled Java code.
- If you have included any libraries in your Java code you will need to include them in the classpath here too.
The GrantCancel example

- Recall this requirement - we are going to use it to illustrate how TraceMatches works.

**Requirement GrantCancel**

For a given resource, grants and cancellations should alternate, starting with a grant. Furthermore: a cancellation should be performed by the same task that was last granted the resource.

Reporting failure

- We will use this utility method to report failure. This prints out a stack trace to enable the user to locate the error.

```java
public class Util {
    protected static void fail(String message) {
        try {
            throw new RuntimeException(message);
        } catch (Exception e) {
            e.printStackTrace();
        }
        System.exit(0);
    }
}
```
The symbols (events) as pointcuts

- We first define pointcuts to capture the events we are interested in. These are:
  - `grant(requester, resource)`
  - `cancel(owner, resource)`

```plaintext
1 pointcut grant(Actor requester , Resource resource) :
2   call(void missioncontrol.Task+.sendGrant(Resource))
3   && args(resource) && target(requester);

4 pointcut cancel(Actor owner , Resource resource) :
5   call(void missioncontrol.Arbitr .sendCancel(Task, Resource))
6   && args(owner, resource);
```

Grant and cancel should alternate

- A `tracematch` defines a regular expression over pointcuts.
- There is a match if any `suffix` of the trace matches the expression.
- We say `TraceMatches` is `suffix-matching` / uses suffix semantics.
- Here this is whenever either of the two events fail to alternate.
- This is parameterised with a `Resource r`.

```plaintext
1 tracematch(Resource r)
2 {
3   sym grant after : grant(∗ , r);
4   sym cancel after : cancel(∗ , r);
5
6   (grant grant) | (cancel cancel)
7   {
8     Util.fail("Calls of \texttt{grant} and \texttt{cancel} on \texttt{resource} \+\texttt{\text{"do not alternate"}}");
9   }
10 }
```
Detecting a match

- Translate the regular expression into a Finite State Automaton.
  - A well understood transformation.
  - Easy to manipulate at runtime.

![Finite State Automaton Diagram]

Detecting a match on a propositional trace

- Let us consider the trace:

  grant.cancel.grant.grant

- Remember - we want to match a trace suffix.
- Mark reached states.
- A state is marked if and only if it can be reached using the current event from a state marked on the previous step.
- Initial states are always marked.
Detecting a match on a propositional trace

\[ \varepsilon \]

Detecting a match on a propositional trace

\[ \varepsilon.\text{grant} \]
Detecting a match on a propositional trace

$\epsilon\cdot\text{grant}$

Detecting a match on a propositional trace

$\epsilon\cdot\text{grant}\cdot\text{cancel}$
Detecting a match on a propositional trace

\[ \epsilon.\text{grant} \cdot \text{cancel} \]

Detecting a match on a propositional trace

\[ \epsilon.\text{grant} \cdot \text{cancel} \cdot \text{grant} \]
Detecting a match on a propositional trace

\[ \epsilon, \text{grant, cancel, grant} \]

1
  Grant
2
   Grant
   4

1
   Cancel
3
   Cancel
   5

Detecting a match on a propositional trace

\[ \epsilon, \text{grant, cancel, grant, grant} \]

1
   Grant
2
   Grant
   4

1
   Cancel
3
   Cancel
   5
Detecting a match on a propositional trace

- We have a match

\[ \epsilon.\text{grant.canceled.grant.grant} \]

Because this suffix matched the regular expression

\[ \epsilon.\text{grant.canceled.grant.grant} \]

- Because this suffix matched the regular expression
Detecting a match on a parametric trace

- Works fine when there is no data - but data is useful!
- Consider the observations:
  
  grant(driving_task, wheels)
  grant(driving_task, antenna)
  cancel(driving_task, wheels)
  grant(camera_task, antenna)

- The antenna resource is granted without first being cancelled.
- For resource ‘antenna’ this trace matches the regular expression on the fourth event.
- Our propositional approach would flag an error on the second event.
- Need a new approach - label states with constraints.

Detecting a match on a parametric trace

- Label the initial state as true and all other states as false.
Detecting a match on a parametric trace

- Let \( r \) stand for resource, \( a \) for antenna and \( w \) for wheels.
- Label state 2 with constraint \( (r=w) \).
- Note that our symbol only binds the resource to \( r \) - the task is ignored.

\[
\text{grant(driving\_task,wheels)}
\]

\[
\text{grant(driving\_task,antenna)}
\]

Detecting a match on a parametric trace

- Add constraint \( (r=a) \) to state 2 (in disjunction).
Detecting a match on a parametric trace

- Remove the constraint \( (r=w) \) from state 2, add this to state 3.

\[
\begin{align*}
\text{grant}(\text{driving task, wheels}) \\
\text{grant}(\text{driving task, antenna}) \\
\text{cancel}(\text{driving task, wheels})
\end{align*}
\]

Detecting a match on a parametric trace

- Add constraint \( (r=a) \) to state 4.

\[
\begin{align*}
\text{grant}(\text{driving task, wheels}) \\
\text{grant}(\text{driving task, antenna}) \\
\text{cancel}(\text{driving task, wheels}) \\
\text{grant}(\text{camera task, antenna})
\end{align*}
\]
Detecting a match on a parametric trace

- \((r=a)\) is a solution to the constraint labelling (final) state 4.
- Therefore, we execute the method body for \((r=a)\).

The details

- We have informally illustrated how \textsc{TraceMatches} operates using an example, we will now:
  - Formalise the components of a tracematch.
  - Capture the (declarative operational) semantics formally.
  - Consider how this translates into an implementation.
  - Discuss efficiency issues.
Symbols

- Let $A$ be the alphabet of symbols declared in the tracematch.
- Let $P$ be the regular expression over $A$ declared in the tracematch.
- A symbol is modeled as a function in $\text{Event} \to \text{Constraint}$.
- The constraint captures a binding for the variables in the symbol.
- For example:

\[
\begin{align*}
\text{grant}(\text{grant}(\text{driving}\_\text{task},\text{wheels})) &= \left(r = \text{wheels}\right) \\
\text{grant}(\text{cancel}(\text{driving}\_\text{task},\text{wheels})) &= \left(\text{false}\right)
\end{align*}
\]

- A trace of (parametric) events matches a trace of symbols if the constraints produced are consistent:

\[
\text{match}(a_1\ldots a_n, e_1\ldots e_n) = \begin{cases} \\
\wedge_i a_i(e_i) & \text{if } m = n \\
\text{false} & \text{otherwise}
\end{cases}
\]

Bindings

- Recall that a binding is a partial map from variables to values.
- Here the variables are the free variables in the tracematch.
- A binding can be applied to a constraint to get a truth value e.g.,

\[
[r \mapsto w]((r=w) \lor (r=a)) = (w=w) \lor (w=a) = \text{true} \lor \text{false} = \text{true}
\]

- Therefore a binding applied to an event creates a predicate on events:

\[
\theta(a) = \lambda e.\theta(a(e)) \in \text{Event} \to \mathbb{B}
\]

*here we assume that bindings bind all free variables*

- Let $P(\theta)$ be the regular expression constructed by applying $\theta$ to each symbol in the regular expression $P$. 

Declarative semantics

- The aim is to find the bindings (of free variables) for which the code should be executed i.e., those that we match on, given a trace $\tau$.
- We start by defining which events are relevant to a binding $\theta$:

$$relevant(\theta) = \{ e \mid \exists a \in A : \theta(a(e)) = true \}$$

- We can filter irrelevant events out of a trace:

$$\epsilon \upharpoonright_{\theta} = \epsilon \quad \tau e \upharpoonright_{\theta} = \begin{cases} (\tau \upharpoonright_{\theta})e & \text{if } e \in relevant(\theta) \\ (\tau \upharpoonright_{\theta}) & \text{otherwise} \end{cases}$$

- A trace satisfies $P$ for $\theta$ if it matches with a word in $P(\theta)$ and its last event is relevant - otherwise irrelevant events cause matching:

$$satisfy(\tau, \theta) = \bigvee_{\sigma \in L(P(\theta))} match(\sigma, \tau \upharpoonright_{\theta}) \land last(\tau) \in relevant(\theta)$$

- The bindings to execute the code for given trace $\tau$ are:

$$\{ \theta \mid \tau' \text{ is a suffix of } \tau \land satisfy(\tau', \theta) \}$$

Operational semantics

- We define a regular expression that captures the declarative semantics.
- Let this be the regular expression $Pat$, such that the code is executed for every solution to

$$\bigvee_{\sigma \in L(Pat)} match(\sigma, \tau)$$

- As we saw previously, we want a trace to match $Pat$ if:
  - a relevant suffix of that trace matches $P - \Sigma^*(P \| skip^*)$
  - the last event of that suffix is in $A - (\Sigma^*A)$

- Let $Pat = \Sigma^*(P \| \text{skip}^*) \cap (\Sigma^*A)$ where:
  - $\Sigma$ is the set of all symbols
  - $\|$ is the interleaving operation
  - $\text{skip}$ is a special symbol that matches irrelevant events
Defining \texttt{skip}

- The \texttt{skip} symbol has two functions:
  - Match any event not in \( A \)
  - Ensure that we do not skip relevant events
- So what constraint should \texttt{skip} produce?
- An event is relevant (where \( C \) is the current constraint) iff
  \[ \exists a \in A : (a(e) \land C) \neq false \]
  Therefore, let \texttt{skip} symbol be defined as:
  \[ \texttt{skip}(e) = \bigwedge_{a \in A} \neg a(e) \]
- For example:
  \[
  \texttt{skip}(\texttt{grant}(d\_task,w)) = \neg\texttt{grant}(\texttt{grant}(d\_task,w)) \land \\
  \neg\texttt{cancel}(\texttt{grant}(d\_task,w)) \\
  = \neg(r = w) \land \neg false \\
  = (r \neq w)
  \]

Defining \texttt{skip}

- The \texttt{skip} symbol has two functions:
  - Match any event not in \( A \)
  - Ensure that we do not skip relevant events
- So what constraint should \texttt{skip} produce?
- An event is relevant (where \( C \) is the current constraint) iff
  \[ \exists a \in A : (a(e) \land C) \neq false \]
  Therefore, let \texttt{skip} symbol be defined as:
  \[ \texttt{skip}(e) = \bigwedge_{a \in A} \neg a(e) \]
- This satisfies the two functions:
  - If \( e \) is not in \( A \) then this will be \texttt{true}
  - If \( e \) is relevant to \( C \) then this will contradict \( C \) - therefore not allowing us to match with \texttt{skip}
From semantics to implementation

- Construct an automaton for $Pat = \Sigma^*(P \parallel \text{skip}^*) \cap (\Sigma^* A)$.
- We missed out some transitions earlier:

```
1
  ^
  |  grant
  ↓  cancel
  → Σ

2
  ^
  |  grant
  ↓  cancel
  → 4

3
  ^
  |  cancel
  ↓  skip
  → 5
```

- Associate a label (of constraints) with each state.
- Update this label for state $i$ as follows:

$$label_i' = \left( \bigvee_{j \xrightarrow{a} i} (label_j \land a(e)) \right) \lor \left( label_i \land \bigwedge_{a \in A} \neg a(e) \right)$$

- Partial matches compatible with $a(e)$ at state $j$ transition to state $i$.
- Remove a partial match if any transition can be taken.
- This moves constraints representing bindings so that they label states the trace filtered with respect to that binding would reach.
Considering efficiency

Removing memory leaks:

- Observations
  1. If labels store monitored objects directly we will get space leaks
  2. The structure of the automaton can be used to identify objects no longer required for monitoring

- Optimisations
  1. Store objects using forms of weak references (where applicable)
  2. Categorise how the variable should be stored at each state into
     - `collectableWeakRefs` - bound on every path from current to final state
     - `weakRefs` - not in the above and not used in action
     - `strongRefs` - not in either of the above

Indexing:

- Observation: An event is only relevant to a small part of a label
- Optimisation: Index labels via a set of variables i.e.

\[ label = Val^n \rightarrow Constraint \]

\( n \) variables selected automatically from those guaranteed to be bound

Finishing the requirement

- Our requirement was:

Requirement **GrantCancel**

For a given resource, grants and cancellations should alternate, starting with a grant. Furthermore: a cancellation should be performed by the same task that was last granted the resource.

- However we have only covered:

For a given resource, grants and cancellations should alternate.

- We now finish implementing this requirement in **TraceMatches**.
For a given resource, grants and cancellations should start with a grant.

- We need to detect the start of the trace.
- It is important that the grant symbol is present.

```
1 tracematch(Resource r)
2 {
3    sym start before:
4        execution(missioncontrol.Main.main(String[]))
5    sym grant after: grant(*,r);
6    sym cancel after: cancel(*,r);
7
8    start cancel
9    {
10       Util.fail("The resource \texttt{"}+r+"\texttt{ was cancelled before}"
11          +"\texttt{ it was granted}"");
12    }
```
Starts with a grant

For a given resource, grants and cancellations should \[\text{start}\] with a grant.

- We need to detect the start of the trace.
- It is important that the \textit{grant} symbol is present.
- With it in the alphabet we do not match on \textit{start}.\textit{grant}.\textit{cancel}.
- Otherwise, \textit{grant} would be filtered out of the trace and it would match:

```
1 start 2 cancel 3
```

The owner of a resource cancels it

Furthermore: a cancellation should be performed by the same task that was last granted the resource.

- Our variables are the resource, the owner of the resource at some stage and another actor who may try and cancel the resource.

```python
1 trace_match( Resource r, Actor owner, Actor other )
2 {
3     sym grant_owner after: grant( owner, r );
4     sym cancel_owner after: cancel( owner, r );
5     sym other_cancel after: cancel( other, r );
6
7     grant_owner other_cancel
8     {
9       if ( other!=owner )
10          Util.fail("Resource \textit{r} cancelled by \textit{other} when held by \textit{owner} + owner");
11     }
12 }
```
Respect conflicts

We can also define this property using TraceMatches.

**Requirement RespectConflicts**

Conflicts must be respected. For every pair of resources, if they conflict then only one can be granted at any one time.

RespectConflicts : Defining Pointcuts

- We first define pointcuts to capture the events we are interested in:
  - `conflict(resource_1, resource_2)`
  - `grant_r1(resource_1)`
  - `cancel_r1(resource_1)`
  - `grant_r2(resource_2)`

```java
1  pointcut conflict(Resource resource1, Resource resource2) :
2      call(void missioncontrol.ResourceTable.addConflict
3          (Resource, Resource)) && args(resource1, resource2);
4
5  pointcut grant(Resource resource) :
6      call(void missioncontrol.Task+.sendGrant(Resource))
7          && args(resource);
8
9  pointcut cancel(Resource resource) :
10     call(void missioncontrol.Arбитр.sendCancel(Task, Resource))
11     && args(owner, resource);
```
RespectConflicts: defining the property

- We define our tracematch using these events.
- A trace matches 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 \( \text{cancel}_r_2 \) is not defined here so the ordering matters.

```plaintext
traceMatch( Resource r1, Resource r2 )
{
    sym conflict after: ( conflict(r1,r2) || conflict(r2,r1) );
    sym grant_r1 after: grant(r1);
    sym cancel_r1 after: cancel(r1);
    sym grant_r2 after: grant(r2);

    conflict+ (grant_r1 | cancel_r1 | grant_r2)*
        grant_r1 grant_r2

    if (r1 != r2)
        Util.fail("Conflicting resources: +r1 and +r2 were granted at the same time");
}
```

Summary

- **TraceMatches** is an extension to AspectJ that allows us to write suffix-matching regular expressions over pointcuts.
- We can effectively quantify over variables in these pointcuts.
- The approach is defined in terms of labelling states with constraints.
- **Weak References** and **Indexing** are used to improve performance.
Adding Trace Matching with Free Variables to AspectJ: C. Allan, P. Avgustinov, A. Simon Christensen, L. Hendren, S. Kuzins, O. De Moor, D. Sereni, G. Sittampalam, J. Tibble. In OOPSLA 2005

Recap

- In the last section we saw the **TraceMatches** tool.
- This was
  - defined as an extension to Aspect
  - suffix-matching
  - implemented via labelling states of a state machine with constraints
In this section we will look at the JavaMOP tool, considering:

- The syntax of JavaMOP
- An illustrative example
- The semantics
- A discussion of algorithms and efficiency

What is JavaMOP

- A language in the Monitoring-Oriented Programming family (MOP).
- MOP is an attempt to formalise the process of monitoring programs as a programming methodology.
- similar to how AOP is a methodology for cross-cutting concerns.
- A stand-alone tool that compiles JavaMOP specifications into AspectJ advice.
- Combines parametric trace slicing with logic plugins to give a generic framework for parametric runtime monitoring.
**Syntax**

- **JAVAMOP syntax** is a special instance of MOP syntax

\[
\text{(Specification)} ::= \{\langle\text{Instance Modifier}\rangle\} \langle\text{Id}\rangle \langle\text{Instance Parameters}\rangle \{\text{“}\}
\{\langle\text{Instance Declaration}\rangle\}
\{\langle\text{Event}\rangle\}
\{\langle\text{Property}\rangle\}
\{\langle\text{Property Handler}\rangle\}
\text{“}\}
\]

\[
\text{(Event)} ::= \{\text{“}creation\text{”}\} \{\text{“}event\text{”} \langle\text{Id}\rangle \langle\text{Instance Event Definition}\rangle \{\text{“}\} \langle\text{Instance Action}\rangle \text{“}\}
\]

\[
\text{(Property)} ::= \langle\text{Logic Name}\rangle \text{“} : \text{”} \langle\text{Logic Syntax}\rangle
\]

\[
\text{(Property Handler)} ::= \text{“}@\text{”} \langle\text{Logic State}\rangle \langle\text{Instance Handler}\rangle
\]

- At a high level, a **JAVAMOP specification** contains the same components as a **TraceMatches specification**:
  - parameter/variable declaration
  - event declarations
  - a propositional property
  - code to execute

- It also includes additional modifiers to modify the semantics and gives optimisation hints (see later).

**Availability**

- Here you can
  - See all publications related to **JAVAMOP**
  - Download the tool.
  - View examples
  - Interact with the tool
Running JavaMOP

- Follow instructions to set appropriate paths.
- Save specification in `<spec-name>.mop` file.
- Run `javamop <spec-name>.mop`.
- This will produce an AspectJ file - weave this as normal with `javamoprt.jar` on the classpath.
- You may wish to add `javamoprt.jar` to your Java extension libraries.

The resource lifecycle

- Let us consider the lifecycle of a resource.
- It can be in one of three states:
  1. Unowned
  2. Requested
  3. Owned
- We would like to make sure it only transits between these states in certain ways i.e.

```
  1  2  3
  ↓  ↓  ↓
request  deny  cancel
          grant  rescind
```

- We will call this the Resource Lifecycle requirement.
Defining a specification

- We first capture the variables used in the specification.
- Here this is just the resource whose lifecycle we are monitoring.
- We can think of this as being universally quantified.
- i.e. For all resources.

\[
\text{ResourceLifeCycle}(\text{Resource } r) \ 
\]

Defining the events

- Events are defined in terms of AspectJ pointcuts

\[
\begin{align*}
\text{event request after}(\text{Resource } r) : \\
& \quad \text{call}(\text{void } *.\text{Task+}.\text{sendRequest}(\text{Resource})) && \text{args}(r) \\
\text{event grant after}(\text{Resource } r) : \\
& \quad \text{call}(\text{void } *.\text{Task+}.\text{sendGrant}(\text{Resource})) && \text{args}(r) \\
\text{event rescind}(\text{Resource } r) : \\
& \quad \text{call}(\text{void } *.\text{Task+}.\text{sendRescind}(\text{Resource})) && \text{args}(r) \\
\text{event cancel after}(\text{Resource } r) : \\
& \quad \text{call}(\text{void } *.\text{Arbiter+}.\text{sendCancel}(\text{Task},\text{Resource})) \\
& \quad && \text{args}(*,r) \\
\text{event deny after}(\text{Resource } r) : \\
& \quad \text{call}(\text{void } *.\text{Arbiter+}.\text{sendDeny}(\text{Task},\text{Resource})) \\
& \quad && \text{args}(*,r)
\end{align*}
\]
Describing the property

- We describe the property with a Finite State Machine (fsm).
- We directly implement the fsm on the previous slide.

```plaintext
fsm :
  start [
    request  ->  requested
  ]
  requested [
    deny  ->  start
    grant  ->  owned
  ]
  owned [
    rescind  ->  owned
    cancel  ->  start
  ]
```

Defining actions

- We can perform different actions when reaching different states, allowing us to record different kinds of error.
- For example, we could log when a resource is granted.
- And we should record errors.
- The `fail` action is called when no transition can be made.

```plaintext
@owned{
    log.print("Resource\"+r+" granted");
}

@fail{
    Util.fail("Resource\"+r+
    " was used incorrectly");
}
```
Matching

- Let us consider how to match against the trace:
  - request(wheels)
  - request(antenna)
  - grant(antenna)
  - deny(wheels)
  - cancel(antenna)
  - request(wheels)
  - rescind(antenna)

- We are already given the finite state machine for our property.

```
request
1 -> 2
    deny
    cancel
2       grant
3         rescind
```

Capturing full behaviour

- We can add in the implicit fail state:

```
grant
cancel
deny
rescind
1
  request
  deny
  cancel
  rescind
2
    grant
    deny
3         rescind
fail
```

Processing the trace

- We will associate bindings with states from this state machine.
- i.e. We will build a map

\[
\text{Bind} \rightarrow \text{State}
\]

- We start with an empty binding and initial state.

\[
\begin{array}{c}
\text{r} \\
\text{state} \\
- \\
1
\end{array}
\]

Processing the trace

- We will associate bindings with states from this state machine.
- i.e. We will build a map

\[
\text{Bind} \rightarrow \text{State}
\]

- We start with an empty binding and initial state.

\[
\text{request(wheels)}
\]

\[
\begin{array}{c}
\text{r} \\
\text{state} \\
- \\
1 \\
\text{wheels} \\
2 \\
\end{array}
\] (requested)
Processing the trace

- We will associate bindings with states from this state machine.
- i.e. We will build a map

\[ \text{Bind} \rightarrow \text{State} \]

- We start with an empty binding and initial state.

<table>
<thead>
<tr>
<th>request(wheels)</th>
<th>r</th>
<th>state</th>
</tr>
</thead>
<tbody>
<tr>
<td>request(antenna)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>grant(antenna)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\( r \) state

- wheels: 2 (requested)
- antenna: 2 (requested)

Processing the trace

- We will associate bindings with states from this state machine.
- i.e. We will build a map

\[ \text{Bind} \rightarrow \text{State} \]

- We start with an empty binding and initial state.

<table>
<thead>
<tr>
<th>request(wheels)</th>
<th>r</th>
<th>state</th>
</tr>
</thead>
<tbody>
<tr>
<td>request(antenna)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>grant(antenna)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\( r \) state

- wheels: 2 (requested)
- antenna: 3 (owned)
We will associate bindings with states from this state machine.
i.e. We will build a map

\[ \text{Bind} \rightarrow \text{State} \]

We start with an empty binding and initial state.

<table>
<thead>
<tr>
<th>Request/Command</th>
<th>r</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>request(wheels)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>request(antenna)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>grant(antenna)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>deny(wheels)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cancel(antenna)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>grant(antenna)</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>wheels</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>antenna</td>
<td>3</td>
<td>(owned)</td>
</tr>
</tbody>
</table>

Processing the trace

We will associate bindings with states from this state machine.
i.e. We will build a map

\[ \text{Bind} \rightarrow \text{State} \]

We start with an empty binding and initial state.

<table>
<thead>
<tr>
<th>Request/Command</th>
<th>r</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>request(wheels)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>request(antenna)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>grant(antenna)</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>wheels</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>antenna</td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>
We will associate bindings with states from this state machine.
i.e. We will build a map

$$\text{Bind} \rightarrow \text{State}$$

We start with an empty binding and initial state.

<table>
<thead>
<tr>
<th>r</th>
<th>state</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>wheels</td>
<td>2  (requested)</td>
</tr>
<tr>
<td>antenna</td>
<td>1</td>
</tr>
</tbody>
</table>

request(wheels)
request(antenna)
grant(antenna)
deny(wheels)
cancel(antenna)
request(wheels)
rescind(antenna)
The details

- We have now seen informally how **JavaMOP** operates using an example, we will now look at how it works.
- **JavaMOP** is built in two halves.
  - The *parametric trace slicing* technique slices an input parametric trace into a set of propositional traces, each associated with a binding.
  - A *logic plugin* defines how to interpret each propositional trace.
- **JavaMOP** can run in different modes - which tells us how to interpret the results from the logic plugin.

### Parametric Trace Slicing

- Parametric trace slicing defines a trace ‘slice’ (subtrace) for a particular binding of the variables.
- **JavaMOP** implicitly translates parametric events of the form $e(\overline{v})$ to parameterised events of the form $e(\theta)$ by associating an event name $e$ with a parameter signature $\overline{x}$ - it is assumed this occurs during event extraction.
- An event $e(\theta')$ is relevant to binding $\theta$ if it only includes things mentioned in $\theta$ - $\theta'$ is a submap of $\theta$.
- Therefore, Parametric Trace Slicing is defined as
  
  $$\epsilon \downarrow_\theta = \epsilon$$

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

- Note the similarity with filtering in **TraceMatches**.
Slicing our trace

- Let us call this trace \( \tau \).

\[
\begin{align*}
&\text{request(wheels)} \\
&\text{request(antenna)} \\
&\text{grant(antenna)} \\
&\text{deny(wheels)} \\
&\text{cancel(antenna)} \\
&\text{request(wheels)} \\
&\text{rescind(antenna)}
\end{align*}
\]

- We can slice it with respect to wheels and antenna:

\[
\begin{align*}
\tau \downarrow_{[r \mapsto w]} &= \text{request(w).deny(w).request(w)} \\
\tau \downarrow_{[r \mapsto a]} &= \text{request(a).grant(a).cancel(a).rescind(a)}
\end{align*}
\]

Limitations of this approach

- We do not illustrate JavaMOP using our GrantCancel and RespectPriorities examples
- As JavaMOP cannot capture these properties without resorting to programming as we did with AspectJ
- The main limitation that prevents us from doing this is that we each program event may only relate to a single event in the specification
- For example, we cannot define the events grant\_r\_1 and grant\_r\_2 both related to the same pointcut (but with different values) as we did with TraceMatches
- By restricting expressiveness, JavaMOP is able to carry out monitoring more efficiently
Logic plugins

- Logic Plugins provide a function

\[ \text{Propositional} \text{Trace} \rightarrow \text{Verdict} \]

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

- Let us consider how we might write different properties using these plugins.

Extended regular expressions

- We can use the ere plugin to model the first part of the GrantCancel property, similar to how we did this in the TraceMatches approach.
- This defines the property as matching on a trace suffix by using the emphsuffix mode.
- Code is executed when the ere is matched.

```
1 suffix GrantCancel (Resource r) {
2      ... 
3      ere : (grant grant) | (cancel cancel)
4
5      @match{
6          ... 
7      }
8  }
```

Grant and Cancel should alternate
Extended regular expressions

- Alternatively we can rewrite this so that code is executed when the expression is not matched.
- This can be more intuitive and leads us to write validation rather than violation properties.

```java
GrantCancel(Resource r) {
  ...
  ere : (grant cancel)*
  @fail{
    ...
  }
}
```

Grant and Cancel should alternate.

Linear temporal logic

- We have all the standard LTL operators.
- Code is executed when the property is violated.
- We have future time operators.

```java
GrantCancel(Resource r) {
  ...
  ltl : []( grant => () cancel)
  @violation{
    ...
  }
}
```

When you see a Grant the next event is a Cancel
Linear temporal logic

- We have all the standard LTL operators.
- Code is executed when the property is violated.
- And past time operators.

```plaintext
GrantCancel(Resource r) {
    ...
    pltl : cancel => (*) grant
    @violation{
        ...
    }
}
```

A cancel must be preceded by a grant.

Context free grammars

- Grammars follow the standard syntax.
- Again we can either match a grammar.

```plaintext
GrantCancel(Resource r) {
    ...
    cfg : S -> W | G | C
        W -> start cancel
        G -> grant grant
        C -> cancel cancel
    @match{
        ...
    }
}
```

Grants and cancel should alternate, starting with a grant
Context free grammars

- Grammars follow the standard syntax.
- Or fail to match it.

```
1 GrantCancel(Resource r) {
2   ...
3   cfg : S -> grant cancel S | epsilon
4
5   @fail{
6     ...
7   }
8 }
```

Grants and cancel should alternate, starting with a grant.

Putting it together

- We have Parametric Trace slicing as a function:
  \[ \text{pts} \in \text{ParametricTrace} \times \text{Bind} \rightarrow \text{PropositionalTrace} \]

- And a Logic Plugin as a function:
  \[ \text{plugin} \in \text{PropositionalTrace} \rightarrow \text{Verdict} \]

- So the verdict for a trace given a binding is:
  \[ \text{check} \in \text{ParametricTrace} \times \text{Bind} \rightarrow \text{Verdict} = \text{plugin} \circ \text{pts} \]

- We can model an action as a predicate that checks the verdict and a function that takes a binding and returns code:
  \[ \text{Action} = (\text{Verdict} \rightarrow \mathbb{B}) \times (\text{Bind} \rightarrow \text{Code}) \]

- Therefore, the code to execute on observing parametric trace \( \tau \) is:
  \[ \{ \text{act.snd}(\theta) \mid \text{act} \in \text{Actions} \land \theta \in \text{Bind} \land \text{act.fst} (\text{check}(\tau, \theta)) \} \]
JavaMOP modes

JavaMOP has some additional modes which alter these semantics:

- **suffix**
  - Performs suffix matching (as in `TraceMatches`) rather than complete matching.

- **perthread**
  - Constructs a separate trace per program thread.

- **full-binding**
  - Actions are only fired by bindings that bind all variables in the specification.

- **unsynchronized**
  - Access to the monitor state is not synchronized - faster but may introduce data races.

- **decentralized**
  - Indexing is decentralized - see later for details.

JavaMOP modes (cont)

JavaMOP has some additional modes which alter these semantics:

- **maximal-bindings**
  - A binding $\theta$ can only cause an action to fire if there does not exist a binding $\theta'$ such that $\theta \subseteq \theta'$ and $\text{pts}(\theta', \tau) \neq \epsilon$.
  - i.e. only match on the largest relevant binding

- **connected**
  - Only connected bindings may cause an action to fire. A binding is connected if all bound values are connected (transitively) by events.
  - We may wish to define behaviours for objects related by events.
  - For example - for every enumeration constructed from some collection.
From semantics to implementation

- Now let us consider how we move from the semantics of parametric trace slicing to an implementation.
- We will consider only logic plugins which can be translate to fsm (and give no details of this translation).
- We will not consider alterations required for different modes.
  (See published work for these details)
- We will present an algorithm for parametric trace slicing.
- And refine this based on considerations of efficiency.

An API example

- We turn to an example of correct API usage to demonstrate JAVAMOP's implementation.

Requirement UnsafeMapIterator

When a collection (i.e. key or value set) is created from a map and an iterator is created for this collection, do not use the iterator after the original map is updated.

<table>
<thead>
<tr>
<th>keySet</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set&lt;K&gt; keySet()</td>
</tr>
<tr>
<td>Returns a set view of the keys contained in this map. The set is backed by the map, so changes to the map are reflected in the set, and vice-versa. If the map is modified while an iteration over the set is in progress (except through the iterator’s own remove operation), the results of the iteration are undefined. The set supports element removal, which removes the corresponding mapping from the map, via the iterator's remove, set, remove, removeAll, retainAll, and clear operations. It does not support the add or addAll operations.</td>
</tr>
<tr>
<td>Returns: a set view of the keys contained in this map</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Collection&lt;V&gt; values()</td>
</tr>
<tr>
<td>Returns a collection view of the values contained in this map. The collection is backed by the map, so changes to the map are reflected in the collection, and vice-versa. If the map is modified while an iteration over the collection is in progress (except through the iterator's own remove operation), the results of the iteration are undefined. The collection supports element removal, which removes the corresponding mapping from the map, via the iterator's remove, Collection.remove, collection.remove, retainAll, and clear operations. It does not support the add or addAll operations.</td>
</tr>
<tr>
<td>Returns: a collection view of the values contained in this map</td>
</tr>
</tbody>
</table>
The JavaMOP specification

```java
import java.util.*;

full-binding UnsafeMapIterator(Map m, Collection c, Iterator i){

event createC after(Map m) returning(Collection c):
    (call(* Map.values()) || call(* Map.keySet()))
    && target(m) {};

event createI after(Collection c) returning(Iterator i):
    call(* Collection.iterator()) && target(c) {};

event use before(Iterator i):
    call(* Iterator.next()) && target(i) {};

event update after(Map m):
    (call(* Map.put(..)) || call(* Map.putAll(..))
    || call(* Map.clear()) || call(* Map.remove(..)))
    && target(m) {};

eres: createC update* createI use* update update* use

@match{ System.out.println("unsafe_iterator_usage!");}
}
```

An automaton

- We can construct an automaton for the expression
  
  createC update* createI use* update update* use

- A match is detected if we reach state 5.
An example trace

- Let us consider the trace:
  
  \[
  \begin{align*}
  &\text{createC}(M_1, C_1) \\
  &\text{createC}(M_1, C_2) \\
  &\text{createI}(C_1, I_1) \\
  &\text{update}(C_1) \\
  &\text{createI}(C_2, I_2) \\
  &\text{use}(I_1)
  \end{align*}
  \]

- According to the theory, the code should be executed for binding 
\[ [c \mapsto C_1, m \mapsto M_1, i \mapsto I_1] \] as the slice

\[
\text{createC createI update use}
\]

matches the expression

A basic algorithm

\begin{align*}
\text{Input: } &\text{ a parametric trace } \tau \\
\text{Output: } &\text{ a map from bindings to propositional traces}\\
1 \quad &\Delta : [\text{Bind } \rightarrow \text{PropositionalTrace}] ; \\
2 \quad &\Theta : \text{Bind} ; \\
3 \quad &\Delta \leftarrow [\bot \rightarrow \epsilon] ; \\
4 \quad \text{foreach } &e(\theta) \in \tau \text{ in order do} \\
5 \quad &\Theta \leftarrow \text{dom}(\Delta) ; \\
6 \quad \text{foreach } &\theta' \in \Theta \text{ do} \\
7 \quad &\text{if } \theta \text{ is consistent with } \theta' \text{ then} \\
8 \quad &\quad \theta_{\text{max}} \leftarrow [ ] ; \\
9 \quad &\quad \text{foreach } \theta_{\text{alt}} \in \Theta \text{ do} \\
10 \quad &\quad \quad \text{if } \theta_{\text{max}} \subseteq \theta_{\text{alt}} \subseteq \theta \uparrow \theta' \text{ then } \theta_{\text{max}} = \theta_{\text{alt}} \\
11 \quad &\quad \Delta(\theta \uparrow \theta') \leftarrow \Delta(\theta_{\text{max}})e \\
12 \quad \text{return } &\Delta
\end{align*}
What it does

1. Initialise $\Delta$
2. For each event $e(\theta) \in \tau$ in order do
3. $\Theta \leftarrow \text{dom}(\Delta)$;
4. foreach $\theta' \in \Theta$ do
5. if $\theta$ is consistent with $\theta'$ then
6. $\theta_{\text{max}} \leftarrow \theta'$;
7. foreach $\theta_{\text{alt}} \in \Theta$ do
8. if $\theta_{\text{max}} \sqsubseteq \theta_{\text{alt}} \sqsubseteq \theta \rightarrow \theta'$ then $\theta_{\text{max}} = \theta_{\text{alt}}$
9. $\Delta(\theta \rightarrow \theta') \leftarrow \Delta(\theta_{\text{max}}) e$
10. return $\Delta$

First inefficiency

- It is inefficient to store the propositional traces directly.
- We can use the assumption that our property can be presented by a FSM to update the algorithm.
- We map bindings to states.
- Let $q_0$ and $\delta$ be the initial state and transition function.
- Therefore, we compute the check function.

**Input**: a parametric trace $\tau$

**Output**: a map from bindings to monitors

1. $\text{Check}(\tau)\{$
2. $\Delta : [\text{Bind} \rightarrow \text{State}]; \Theta : \text{Bind}$;
3. $\Delta \leftarrow [\bot \rightarrow q_0]$;
4. foreach $e(\theta) \in \tau$ in order do
5. $\Theta \leftarrow \text{dom}(\Delta)$;
6. foreach $\theta' \in \Theta$ do
7. if $\theta$ is consistent with $\theta'$ then
8. $\theta_{\text{max}} \leftarrow \theta'$;
9. foreach $\theta_{\text{alt}} \in \Theta$ do
10. if $\theta_{\text{max}} \sqsubseteq \theta_{\text{alt}} \sqsubseteq \theta \rightarrow \theta'$ then $\theta_{\text{max}} = \theta_{\text{alt}}$
11. $\Delta(\theta \rightarrow \theta') \leftarrow \delta(\Delta(\theta_{\text{max}}), e)$
12. return $\Delta$ return $\theta \in \text{dom}(\Delta)$ where $\Delta(\theta)$ is final
13. $\}$
How it works

- Bindings can be represented in a lattice using the submap relation \( \subseteq \).
- For example, let us represent the bindings computed for our trace where we use \((x,y,z)\) to represent the binding \([m \mapsto x, c \mapsto y, i \mapsto z]\).
- Let us also label the bindings with states (with F for fail).

![Diagram]

- On receiving create\((M_1,C_1)\) we construct a new binding

\[
\begin{array}{c}
\text{createC} & \xrightarrow{\text{update}} & \text{createI} & \xrightarrow{\text{use}} & \text{update} & \xrightarrow{\text{update}} & \text{use} & \rightarrow \text{5} \\
\end{array}
\]

\[
\text{Trace} \\
\text{createC}(M_1,C_1)
\]

\[
(M_1,C_1,-):2 \\
(-,-,-):1
\]
We initialise new bindings with the state from the maximal binding (here (-,-,-)) and then apply the event.

\[ \text{createC}(M_1, C_1) \]
\[ \text{createC}(M_1, C_2) \]

We extend the existing \((M_1, C_1, -)\) to get \((M_1, C_1, I_1)\).

As \((M_1, C_1, -)\) is maximal, we initialise this binding with state 2 and make the transition to state 3.

There is no \text{createI} transition from state 1 so \((-, C_1, I_1)\) fails.
How it works

1. createC
2. createI
3. update
4. use
5. 

- update(C₁) is relevant to (-,C₁,-), (-,C₁,I₁), (M₁,C₁,-) and (M₁,C₁,I₁)
- (M₁,C₁,-) does a self transition and (M₁,C₁,I₁) moves to state 4

Trace

createC(M₁,C₁)
createC(M₁,C₂)
createI(C₁,I₁)
update(C₁)

(M₁,C₁,I₁):4
(M₁,C₂,-):2
(-,C₁,I₁):F
(M₁,C₁,-):2
(-,C₁,-):F
(-,-,-):1

How it works

1. createC
2. createI
3. update
4. use
5. 

- (M₁,C₂,I₂) extends (M₁,C₂,-) and (-,C₂,I₂) extends (-,-,-)

Trace

createC(M₁,C₁)
createC(M₁,C₂)
createI(C₁,I₁)
update(C₁)
createI(C₂,I₂)

(M₁,C₂,I₂):3
(M₁,C₁,I₁):4
(-,C₂,I₂):F
(M₁,C₂,-):2
(-,C₁,I₁):F
(M₁,C₁,-):2
(-,C₁,-):F
(-,-,-):1
How it works

Recall that we are in full-binding mode.
We only match if a full-binding reaches a relevant state (here 5).
Therefore, code is executed for \((M_1,C_1,I_1)\) as an improper usage was detected.

### Trace

<table>
<thead>
<tr>
<th>Action</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>create(C) ((M_1,C_1))</td>
<td>1</td>
</tr>
<tr>
<td>create(C) ((M_1,C_2))</td>
<td>2</td>
</tr>
<tr>
<td>create(I) ((C_1,I_1))</td>
<td>3</td>
</tr>
<tr>
<td>update ((C_1))</td>
<td>4</td>
</tr>
<tr>
<td>create(I) ((C_2,I_2))</td>
<td>5</td>
</tr>
<tr>
<td>use ((I_1))</td>
<td>6</td>
</tr>
</tbody>
</table>

**Applying use for binding \((M_1,C_1,I_1)\) takes it to state 5**
Second inefficiency

- Let $n$ be the number of bindings in $\text{dom}(\Delta)$ at a particular step.
- We access $\Delta$ $n^2$ times on each step - we check every binding to see if it is relevant to the event.
- This is inefficient.
- Instead, we can directly lookup relevant events by storing in a map, for each binding, those existing bindings that are relevant.

What should $\mathcal{U}$ be?

- Let $\mathcal{U} : \text{Bind} \to 2^{\text{Bind}}$ be such a map
- We want $\mathcal{U}$ to help us update $\Delta$
- $\Delta$ should be ‘union-closed’ - if two compatible bindings are in $\Delta$, their union should also be in $\Delta$:

$$\forall \theta, \theta' \in \text{dom}(\Delta) : \text{compatible}(\theta, \theta') \Rightarrow \theta \cup \theta' \in \text{dom}(\Delta)$$

- $\mathcal{U}$ should be ‘submap-closed’ - every submap of a binding in $\Delta$ should be in $\mathcal{U}$:

$$\forall \theta \in \text{dom}(\Delta), \forall \theta' \in \text{Bind} : \theta' \sqsubseteq \theta \Rightarrow \theta' \in \text{dom}(\mathcal{U})$$

- $\mathcal{U}$ should be ‘relevance-closed’ - every entry in $\mathcal{U}$ should point to the relevant bindings in $\Delta$:

$$\forall \theta, \theta' \in \text{dom}(\Delta) : \theta \sqsubseteq \theta' \Rightarrow \theta' \in \mathcal{U}(\theta)$$
A refined algorithm

\[ \Delta : [\text{Bind} \to \text{State}]; \mathbb{U} : \text{Bind} \to 2^{\text{Bind}} \]

\[ \Delta \leftarrow \{
\bot \to q_0\}; \mathbb{U} \leftarrow \emptyset \text{ for any } \theta \in \text{Bind} \]

\foreach \ e(\theta) \in \tau \text{ in order do}
  \begin{align*}
    & \text{if } \theta \notin \text{dom}(\Delta) \text{ then} \\
    & \quad \text{foreach } \theta_m \sqsubseteq \theta \text{ (big to small) do} \\
    & \quad \quad \text{if } \theta_m \in \text{dom}(\Delta) \text{ then break} \\
    & \quad \quad \text{defTo}(\theta, \theta_m) \text{ } \\
    & \quad \text{foreach } \theta_m \sqsubseteq \theta \text{ (big to small) do} \\
    & \quad \quad \text{foreach } \theta' \in \mathbb{U}(\theta_m) \\
    & \quad \quad \quad \text{compatible with } \theta \text{ do} \\
    & \quad \quad \quad \quad \text{if } (\theta' \sqcup \theta) \notin \text{dom}(\Delta) \text{ then defTo}(\theta' \sqcup \theta, \theta') \\
    & \text{foreach } \theta' \in \{\theta\} \cup \mathbb{U}(\theta) \text{ do} \text{ } \\
    & \quad \Delta(\theta') \leftarrow \sigma(\Delta(\theta'), e) \\
  \end{align*}

\text{return } \Delta

\textbf{Initialisation}
\textbf{If } \theta \text{ not in } \mathbb{U} \text{ add it and ensure closure properties}
\textbf{We will look at how this is done next}
\textbf{Update states for relevant bindings}

\textbf{Closing } \mathbb{U}

\begin{enumerate}
  \item \text{if } \theta \notin \text{dom}(\Delta) \text{ then} \\
  \quad \text{foreach } \theta_m \sqsubseteq \theta \text{ (big to small) do} \\
  \quad \quad \text{if } \theta_m \in \text{dom}(\Delta) \text{ then break} \\
  \quad \quad \text{defTo}(\theta, \theta_m) \text{ } \\
  \quad \text{foreach } \theta_m \sqsubseteq \theta \text{ (big to small) do} \\
  \quad \quad \text{foreach } \theta' \in \mathbb{U}(\theta_m) \\
  \quad \quad \quad \text{compatible with } \theta \text{ do} \\
  \quad \quad \quad \quad \text{if } (\theta' \sqcup \theta) \notin \text{dom}(\Delta) \text{ then defTo}(\theta' \sqcup \theta, \theta') \\
  \text{defTo}(\theta, \theta'): \text{ } \\
  \Delta(\theta) \leftarrow \Delta(\theta') \\
  \text{foreach } \theta'' \sqsubseteq \theta \text{ do} \\
  \quad \mathbb{U}(\theta'') \leftarrow \mathbb{U}(\theta'') \cup \{\theta\}
\end{enumerate}

\textbf{We only need to update } \mathbb{U} \text{ if } \theta \text{ is not in } \mathbb{U}
\textbf{We first find the maximal binding in } \Delta \text{ (might be } \bot) \text{ }
\textbf{Use it to add } \theta \text{ }
\textbf{Ensures closure properties}
\textbf{Consider all submaps}
\textbf{Attempt to create all unions}
\textbf{defTo uses the state from the maximal binding to initialise } \theta \text{ }
\textbf{Relevance-closes } \mathbb{U} \text{ for } \theta \text{ i.e. adds it to the } \mathbb{U}-\text{entry for all smaller existing bindings}
Why is this better?

1. foreach $e(\theta) \in \tau$ in order do
   2. if $\theta \notin \text{dom}(\Delta)$ then
   3. foreach $\theta_m \sqsubseteq \theta$ (big to small) do
   4. if $\theta_m \in \text{dom}(\Delta)$ then break
   5. defTo$(\theta, \theta_m)$
   6. foreach $\theta_m \sqsubseteq \theta$ (big to small) do
   7. foreach $\theta' \in U(\theta_m)$ compatible with $\theta$ do
   8. if $(\theta' \sqcup \theta) \notin \text{dom}(\Delta)$ then defTo$(\theta' \sqcup \theta, \theta')$
   9. foreach $\theta' \in \{\theta\} \cup U(\theta)$ do
   10. $\Delta(\theta') \leftarrow \sigma(\Delta(\theta'), e)$
11. return $\Delta$

- We only update $U$ if we haven’t seen the event’s objects before.

**Optimise Common Case**

- Only iterate over small collections - we expect $U(\theta)$ to be small compared to $\text{dom}(\Delta)$.

**How it works**

- We begin with $\Delta$ containing the empty binding and initial state, and $U$ empty

<table>
<thead>
<tr>
<th>Trace</th>
<th>$\Delta$</th>
<th>$U$</th>
</tr>
</thead>
<tbody>
<tr>
<td>(-,-,-)</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>
### How it works

- Adding \((M_1, -, -)\) and \((-, C_1, -)\) to \(\mathbb{U}\) allows us to find \((M_1, C_1, -)\) in the future whenever we see an event using just \(C_1\) or \(M_1\) with

<table>
<thead>
<tr>
<th>Trace</th>
<th>(\Delta)</th>
<th>(\mathbb{U})</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{createC}(M_1, C_1))</td>
<td>((-,-,-))</td>
<td>((-,-,)) ((M_1, C_1, -))</td>
</tr>
<tr>
<td></td>
<td>((M_1, C_1, -))</td>
<td>((M_1, C_1, -))</td>
</tr>
<tr>
<td></td>
<td>(1)</td>
<td>(2)</td>
</tr>
</tbody>
</table>

- \((M_1, C_2, -)\) is also added to the entry in \(\mathbb{U}\) for \((M_1, -, -)\) - this relates to the ‘above-of’ relation in the lattice we were building earlier.

<table>
<thead>
<tr>
<th>Trace</th>
<th>(\Delta)</th>
<th>(\mathbb{U})</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{createC}(M_1, C_1))</td>
<td>((-,-,-))</td>
<td>((-,-,)) ((M_1, C_1, -))</td>
</tr>
<tr>
<td>(\text{createC}(M_1, C_2))</td>
<td>((-,-,-))</td>
<td>((-,-,)) ((M_1, C_1, -))</td>
</tr>
<tr>
<td></td>
<td>((M_1, C_1, -))</td>
<td>((M_1, C_1, -))</td>
</tr>
<tr>
<td></td>
<td>(1)</td>
<td>(2)</td>
</tr>
<tr>
<td></td>
<td>((M_1, C_1, -))</td>
<td>((M_1, C_1, -))</td>
</tr>
<tr>
<td></td>
<td>((-,-))</td>
<td>((-,-,)) ((M_1, C_1, -))</td>
</tr>
<tr>
<td></td>
<td>((M_1, C_2, -))</td>
<td>((M_1, C_2, -))</td>
</tr>
<tr>
<td></td>
<td>((-,-,-))</td>
<td>((-,-,)) ((M_1, C_1, -))</td>
</tr>
<tr>
<td></td>
<td>((M_1, C_1, -))</td>
<td>((M_1, C_1, -))</td>
</tr>
<tr>
<td></td>
<td>(1)</td>
<td>(2)</td>
</tr>
<tr>
<td></td>
<td>((-,-,)) ((M_1, C_1, -))</td>
<td>((M_1, C_2, -))</td>
</tr>
<tr>
<td></td>
<td>((M_1, C_1, -))</td>
<td>((M_1, C_1, -))</td>
</tr>
</tbody>
</table>
How it works

- \((-,C_1,I_1)\) is added from \((-,-,-)\)
- \((M_1,C_1,-)\) in \(\mathbb{U}((-,-,-))\) is used to add \((M_1,C_1,I_1)\)

<table>
<thead>
<tr>
<th>Trace</th>
<th>(\Delta)</th>
<th>(\mathbb{U})</th>
</tr>
</thead>
<tbody>
<tr>
<td>create\text{C}(M_1,C_1)</td>
<td>(-,-,-)</td>
<td>((-,,-)) ((M_1,C_1,-)) ((M_1,C_2,-))</td>
</tr>
<tr>
<td>create\text{C}(M_1,C_2)</td>
<td>(M_1,C_1,-)</td>
<td>2 ((-,C_1,I_1)) ((M_1,C_1,I_1))</td>
</tr>
<tr>
<td>create\text{I}(C_1,I_1)</td>
<td>(\cdot,C_1,I_1)</td>
<td>3 ((-,C_1,-)) ((M_1,C_1,-)) ((M_1,C_2,-))</td>
</tr>
<tr>
<td></td>
<td>(M_1,C_2,-)</td>
<td>2 ((-,C_2,-)) ((M_1,C_2,-)) ((-,C_2,l_2))</td>
</tr>
<tr>
<td></td>
<td>(\cdot,C_1,l_1)</td>
<td>4 ((-,l_1)) ((M_1,C_1,I_1)) ((M_1,C_1,l_1))</td>
</tr>
<tr>
<td></td>
<td>(M_1,-,-)</td>
<td>((-,C_1,I_1)) ((M_1,C_1,l_1)) ((M_1,C_1,l_1))</td>
</tr>
<tr>
<td></td>
<td>((-,C_1,-))</td>
<td>((-,C_1,-)) ((M_1,C_1,-)) ((M_1,C_1,I_1))</td>
</tr>
<tr>
<td></td>
<td>((-,-,l_1))</td>
<td>((-,l_1)) ((M_1,C_1,I_1)) ((M_1,C_1,l_1))</td>
</tr>
<tr>
<td></td>
<td>((-,l_1))</td>
<td>((-,l_1)) ((M_1,C_1,I_1)) ((M_1,C_1,l_1))</td>
</tr>
<tr>
<td></td>
<td>((-,l_1))</td>
<td>((-,l_1)) ((M_1,C_1,I_1)) ((M_1,C_1,l_1))</td>
</tr>
<tr>
<td></td>
<td>((-,l_1))</td>
<td>((-,l_1)) ((M_1,C_1,I_1)) ((M_1,C_1,l_1))</td>
</tr>
<tr>
<td></td>
<td>((-,l_1))</td>
<td>((-,l_1)) ((M_1,C_1,I_1)) ((M_1,C_1,l_1))</td>
</tr>
<tr>
<td></td>
<td>((-,l_1))</td>
<td>((-,l_1)) ((M_1,C_1,I_1)) ((M_1,C_1,l_1))</td>
</tr>
</tbody>
</table>

How it works

- \(\theta_m\) is \((-,-,-)\) therefore \(\text{defTo}((-,-,-),(-,-,-))\) sets \((-,C_1,-)\) to state 1 which is updated to \(F\) by \(\sigma\)
- As expected \(\mathbb{U}((-,-,-)) = \{(M_1,C_1,-),(-,C_1,l_1),(M_1,C_1,l_1)\}\)
How it works

- $\theta_m$ is $(-,-,-)$ so $\text{defTo}((-,-,\text{I}_2),(-,-,-))$ adds this to $\Delta$ with state 1 and applying $\sigma$ updates this to $F$
- We consider $(-,\text{C}_2,-) \sqsubseteq (-,\text{C}_2,\text{I}_2)$ and use $\text{U}((-,-,\text{I}_1))$ to add $(\text{M}_1,\text{C}_2,\text{I}_2)$

<table>
<thead>
<tr>
<th>Trace</th>
<th>$\Delta$</th>
<th>$\text{U}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>createC($\text{M}_1,\text{C}_1$)</td>
<td>$(-,-,-)$</td>
<td>$(\text{M}_1,\text{C}_1,-)(\text{M}_1,\text{C}_2,-)$</td>
</tr>
<tr>
<td>createC($\text{M}_1,\text{C}_2$)</td>
<td>$(-,\text{C}_1,-)$</td>
<td>$(-,\text{C}_1,\text{I}_1)(\text{M}_1,\text{C}_1,\text{I}_1)$</td>
</tr>
<tr>
<td>createI($\text{C}_1,\text{I}_1$)</td>
<td>$(\text{M}_1,\text{C}_2,\text{I}_2)$</td>
<td>$(-,-,-)$</td>
</tr>
<tr>
<td>update($\text{C}_1$)</td>
<td>$(-,\text{C}_1,-)$</td>
<td>$(-,\text{C}_1,-)$</td>
</tr>
<tr>
<td>createI($\text{C}_2,\text{I}_2$)</td>
<td>$(-,\text{C}_2,\text{I}_2)$</td>
<td>$(-,\text{C}_2,\text{I}_2)$</td>
</tr>
<tr>
<td>createC($\text{M}_1,\text{C}_1$)</td>
<td>$(\text{M}_1,\text{C}_1,-)$</td>
<td>$(-,\text{C}_2,\text{I}_2)$</td>
</tr>
<tr>
<td>createC($\text{M}_1,\text{C}_2$)</td>
<td>$(\text{M}_1,\text{C}_2,\text{I}_2)$</td>
<td>$(\text{M}_1,\text{C}_2,\text{I}_2)$</td>
</tr>
<tr>
<td>createI($\text{C}_1,\text{I}_1$)</td>
<td>$(-,\text{C}_1,-)$</td>
<td>$(-,\text{C}_2,\text{I}_2)$</td>
</tr>
<tr>
<td>update($\text{C}_1$)</td>
<td>$(-,\text{C}_1,-)$</td>
<td>$(-,\text{C}_1,-)$</td>
</tr>
<tr>
<td>createI($\text{C}_2,\text{I}_2$)</td>
<td>$(-,\text{C}_2,\text{I}_2)$</td>
<td>$(-,\text{C}_2,\text{I}_2)$</td>
</tr>
<tr>
<td>use($\text{I}_1$)</td>
<td>$(\text{M}_1,\text{C}_1,\text{I}_1)$</td>
<td>$(\text{M}_1,\text{C}_1,\text{I}_1)$</td>
</tr>
</tbody>
</table>

How it works

- We can use the $(-,-,\text{I}_1)$ entry in $\text{U}$ to find the two relevant bindings
- Previously we would have had to compare $(-,-,\text{I}_1)$ with every binding in $\Delta$
How it works

<table>
<thead>
<tr>
<th>Trace</th>
<th>Δ</th>
<th>( \Delta )</th>
<th>( \cup )</th>
</tr>
</thead>
<tbody>
<tr>
<td>createC((M_1,C_1))</td>
<td>((-,-,-))</td>
<td>1</td>
<td>((-,-,-)) ( (M_1,C_1,-)(M_1,C_2,-) )</td>
</tr>
<tr>
<td>createC((M_1,C_2))</td>
<td>((M_1,C_1,-))</td>
<td>F</td>
<td>((-,-,-)) ( (M_1,C_1,-)(M_1,C_2,-) )</td>
</tr>
<tr>
<td>createI((C_1,I_1))</td>
<td>((-,C_1,I_1))</td>
<td>2</td>
<td>((-,-,-)) ( (M_1,C_2,I_2)(-,I_1) )</td>
</tr>
<tr>
<td>update((C_1))</td>
<td>((M_1,C_1,I_1))</td>
<td>5</td>
<td>((-,-,-)) ( (M_1,C_1,-)(M_1,C_2,-) )</td>
</tr>
<tr>
<td>createI((C_2,I_2))</td>
<td>((-,C_1,-))</td>
<td>F</td>
<td>((-,-,-)) ( (M_1,C_1,I_1)(M_1,C_2,I_2) )</td>
</tr>
<tr>
<td>use((I_1))</td>
<td>((-,-,-))</td>
<td>3</td>
<td>((-,-,-)) ( (M_1,C_1,I_1) )</td>
</tr>
</tbody>
</table>

Further Inefficiencies

There are two other main methods for reducing inefficiency

- **Minimising Garbage**
  - Introduce *creation events*
  - Introduce *enable* and *co-enable* sets
  - Use *Weak References*

- **Indexing**
  - Create and maintain an index map per set of variables
  - Decentralise indexing - storing index maps in monitored objects via weaving
Summary

- The JavaMOP automatically generates AspectJ code.
- It is based on the concept of parametric trace slicing.
- Specifications can be written in a number of different formalisms using logic plugins.
- The tool has a number of optimisations including algorithms which use complex data structures to remove redundant work

References

Recap

- We have seen the **TraceMatches** tool, defined as an extension to AspectJ.
- We have seen the stand-alone **JavaMOP** tool based on the concept of parametric trace slicing.
- Both of these have efficient monitoring algorithms.
- But both are limited in terms of expressiveness.
Contents

In this section we introduce the RuleR tool.

RuleR focuses on expressiveness.

We consider:
- The syntax of RuleR
- How to use RuleR
- An illustrative example
- The RuleR algorithm

What is RuleR?

- Based on the concept of rewrite rules.
- The monitor is represented by a set of states, each consisting of a set of rule activations.
- Rules rewrite this state.
- Rules can be parameterized with data.
- Written as a stand-alone system - events are explicitly dispatched to a RuleR monitor.
A `RULER` specification has the following parts:

- a name
- observation definitions
- rule definitions
- initial rules
- acceptance conditions

A rule consists of:

- a modifier
- a name
- variable definitions
- a list of "condition → obligations" parts

Syntax: high level

```
  ruler <name>{
    observes : <obs-def> list
    <mod><rule-name>(<var-def> list){
      condition → obligation1
      " | " obligation2
    ...}
    ...}
  initials : <rule-name> list
  [forbidden : <rule-name> list]
  [succeeded : <rule-name> list]
  [assert : <rule-name> list]
}
```

Syntax: low level

- This is only part of the full `RULER` syntax.

```plaintext
<spec> ::= ruler <name> { <body> }
<body> ::= observes : <obs-def> list
             <rule> list
             initials : <rule-name> list
             [ <rules-cond> : <rule-name> list ] list
<rule-cond> ::= succeed | forbidden | assert
<rule> ::= <rule-mod><rule-name>(<vardef> list) { <rule-part> list }
<rule-mod> ::= step | state | always
<rule-part> ::= <literal> list → <obligation>
               | <literal> list {<rule-part> :}
               | <literal> list {<rule-part> |}
<obligation>::= <p-literal> list "|" <obligation> | <epsilon>
<p-literal> ::= <atom> | !<atom>
<atom> ::= <rule-name>(<dsymb> list) | <obs-name>(<dsymb> list) | <p-atom>
<p-atom> ::= <rule-name>(<symb> list) | <obs-name>(<symb> list)
<obs-def> ::= <obs-name>(<vardef> list)
<dsymb> ::= <var> | !<vardef>
<symb> ::= <var> | <val> | <p-atom>
<vardef> ::= <var> ":" <type>
<type> ::= int | long | string | boolean | obj
```
Using RuleR

- **RuleR** is a stand-alone tool that presents the following to the user:
  - a monitor constructor
    ```java
    new Ruler("spec.ruler", "out-file.txt", false)
    ```
  - a dispatch method of the form
    ```java
    dispatch("grant", new Object[]{wheels})
    ```
  - A verdict enumeration for returning results
    ```java
    public enum Verdict
    {TRUE, STILL_TRUE, STILL_FALSE, FALSE, UNKNOWN}
    ```

- These can be called from an AspectJ file containing instrumentation.

The RespectPriorities example

- We will use this example to demonstrate how RuleR works.
- We assume all conflicts and priorities are declared at the beginning.

**Requirement RespectPriorities**

Let priorities sort conflicts. If there's a conflict and the requested resource has the highest priority then the granted resource should be rescinded before any shutdown of the system.
Recording events

- We need to write instrumentation to:
  - construct the RuleR monitor

```java
RuleR monitor = new RuleR(
    "specs/RespectPriorities.ruler",
    "out/RespectPriorities.out",
    false);
```

Recording events

- We need to write instrumentation to:
  - construct the RuleR monitor
  - record the relevant events and dispatch these to the monitor

```java
29  ... 
30  after(Resource resource) : 
31    call(void missioncontrol.Task+.sendRescind(Resource))
32      && args(resource){
33        handle(monitor.dispatch("rescind",
34            new Object[]{resource}));
35    }
36  }
37  
38  after(Resource resource1, Resource resource2) : 
39    call(void missioncontrol.ResourceTable.addConflict
40      (Resource, Resource)) && args(resource1, resource2){
41      handle(monitor.dispatch("conflict",
42            new Object[]{resource1,resource2}));
43    }
44  }
45  ...
```
Recording events

- We need to write instrumentation to:
  - construct the RuleR monitor
  - record the relevant events and dispatch these to the monitor
  - deal with verdicts appropriately
  - capture system shutdown

```java
55 private static handle(Signal verdict) {
56   switch (verdict) {
57     case TRUE : 
58       case STILL_TRUE : break;
59     case STILL_FALSE : log.println("Waiting for a Rescind");
60       break;
61     case FALSE : log.println("Failed:
62       "outstanding Rescind Request");
63   }
64 }
```

Recording events

- We need to write instrumentation to:
  - construct the RuleR monitor
  - record the relevant events and dispatch these to the monitor
  - deal with verdicts appropriately
  - capture system shutdown

```java
after():
execution(void missioncontrol.Main.main(String[])){
   handle(monitor.dispatch("shutdown", new Object[]{}));
}
```
A note on verdicts

- **RULER** has a 4/5 valued logic.
- TRUE and FALSE mean this is the verdict and it cannot change.
- STILL_X means that the verdict is currently X but it may change.

```java
private static handle(Signal verdict) {
    switch (verdict) {
    case TRUE :
    case STILL_TRUE : break;
    case STILL_FALSE : log.println("Waiting for Rescind");
    break;
    case FALSE : log.println("Failed: outstanding Rescind Request");
    }
}
```

The specification: declaring events

- We first declare the events used.
- An event signature consists of a name and a tuple of types.
- A type is either **obj** or a primitive Java type.

```java
ruler RespectPriorities{
    observes conflict(obj,obj), priority(obj,obj),
    request(obj), grant(obj), cancel(obj),
    rescind(obj), shutdown;
```
The specification: declaring rules

- Record conflicts and Priorities.
- Track when each resource is granted and cancelled.
- If a conflicting resource of greater priority is requested and no higher priority granted resource conflicts with y then require a rescind.
- If a rescind is received that’s okay but if we finish first that’s not.

```plaintext
always Start()
conflict(x:obj,y:obj) -> C(x,y),C(y,x);
priority(x:obj,y:obj) -> P(x,y);
grant(x:obj) -> G(x);
}
state C(x:obj,y:obj){}
state P(x:obj,y:obj){}
always G(x:obj){
cancel(x) -> !G(x);
request(y:obj), C(x,y), P(y,x)
{: 
P(z:obj,y), G(z) C(y,z) -> Ok;
default -> Res(x);
:}
}
state Res(x:obj){
rescind(x) -> Ok; shutdown -> Fail;
}
```
The specification: declaring rules

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

  always Start()
  {
    conflict(x: obj, y: obj) -> C(x, y), C(y, x);
    priority(x: obj, y: obj) -> P(x, y);
    grant(x: obj) -> G(x);
  }

  state C(x: obj, y: obj) {}
  state P(x: obj, y: obj) {}
  always G(x: obj) {
    cancel(x) -> !G(x);
    request(y: obj), C(x, y), P(y, x)
    {
      P(z: obj, y), G(z), C(y, z) -> Ok;
      default -> Res(x);
    }
  }

  state Res(x: obj) {
    rescind(x) -> Ok; shutdown -> Fail;
  }

  initials Start;
  forbidden Res;
}

- The rules fit in here.
- Declare initial rule(s).
- Declare forbidden rule(s).
- Define a monitor.

monitor {
  uses M: RespectPriorities;
  run M .
}

Monitoring a parametric trace

- Consider the trace:

  conflict(wheels, antenna) STILL_TRUE
  priority(antenna, wheels) STILL_TRUE
  request(wheels) STILL_TRUE
  grant(wheels) STILL_TRUE
  request(antenna) STILL_FALSE
  rescind(wheels) STILL_TRUE
  cancel(wheels) STILL_TRUE
  grant(antenna) STILL_TRUE
  shutdown STILL_TRUE

- These are the expected results.
- We are waiting for a rescind.
- We have reached the end with no outstanding rescinds required.
Building rule activations

- As we go through the trace we build up a set of rule activations.

<table>
<thead>
<tr>
<th>Trace</th>
<th>Rule Activations</th>
</tr>
</thead>
<tbody>
<tr>
<td>conflict(w,a)</td>
<td>Start()</td>
</tr>
<tr>
<td></td>
<td>C(w,a)</td>
</tr>
<tr>
<td></td>
<td>C(a,w)</td>
</tr>
</tbody>
</table>

always Start(){
  conflict(x:obj,y:obj)
  -> C(x,y), C(y,x);
  priority(x:obj,y:obj)
  -> P(x,y);
  grant(x:obj) -> G(x);
}

- We match with conflict(x:obj,y:obj).
Building rule activations

- As we go through the trace we build up a set of rule activations.

<table>
<thead>
<tr>
<th>Trace</th>
<th>Rule Activations</th>
<th>always Start()</th>
</tr>
</thead>
<tbody>
<tr>
<td>conflict(w,a)</td>
<td>Start()</td>
<td>conflict(x:obj,y:obj)</td>
</tr>
<tr>
<td></td>
<td>C(w,a)</td>
<td>-&gt; C(x,y),C(y,x);</td>
</tr>
<tr>
<td></td>
<td>C(a,w)</td>
<td>priority(x:obj,y:obj)</td>
</tr>
<tr>
<td></td>
<td>P(a,w)</td>
<td>-&gt; P(x,y);</td>
</tr>
<tr>
<td>priority(a,w)</td>
<td></td>
<td>grant(x:obj) -&gt; G(x);</td>
</tr>
</tbody>
</table>

- We match with priority(x:obj,y:obj).

Building rule activations

- As we go through the trace we build up a set of rule activations.

<table>
<thead>
<tr>
<th>Trace</th>
<th>Rule Activations</th>
</tr>
</thead>
<tbody>
<tr>
<td>conflict(w,a)</td>
<td>Start()</td>
</tr>
<tr>
<td>priority(a,w)</td>
<td>C(w,a)</td>
</tr>
<tr>
<td>request(w)</td>
<td>C(a,w)</td>
</tr>
<tr>
<td></td>
<td>P(a,w)</td>
</tr>
</tbody>
</table>

- Nothing matches.
Building rule activations

- As we go through the trace we build up a set of rule activations.

<table>
<thead>
<tr>
<th>Trace</th>
<th>Rule Activations</th>
<th>always Start(){}</th>
</tr>
</thead>
<tbody>
<tr>
<td>conflict(w,a)</td>
<td>Start()</td>
<td>conflict(x:obj,y:obj) -&gt; C(x,y),C(y,x);</td>
</tr>
<tr>
<td>priority(a,w)</td>
<td>C(w,a)</td>
<td>priority(x:obj,y:obj) -&gt; P(x,y);</td>
</tr>
<tr>
<td>request(w)</td>
<td>C(a,w)</td>
<td>Grant(x:obj) -&gt; G(x);</td>
</tr>
<tr>
<td>grant(w)</td>
<td>P(a,w)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>G(w)</td>
<td></td>
</tr>
</tbody>
</table>

- We match with grant(x:obj).

Building rule activations

- As we go through the trace we build up a set of rule activations.

<table>
<thead>
<tr>
<th>Trace</th>
<th>Rule Activations</th>
<th>always G(w){</th>
</tr>
</thead>
<tbody>
<tr>
<td>conflict(w,a)</td>
<td>Start()</td>
<td>cancel(w) -&gt; !G(w);</td>
</tr>
<tr>
<td>priority(a,w)</td>
<td>C(w,a)</td>
<td>request(y:obj), C(w,y), P(y,w) {:</td>
</tr>
<tr>
<td>request(w)</td>
<td>C(a,w)</td>
<td>P(z:obj,y),G(z),C(y,z)</td>
</tr>
<tr>
<td>grant(w)</td>
<td>P(a,w)</td>
<td>-&gt; true;</td>
</tr>
<tr>
<td>request(a)</td>
<td>G(w)</td>
<td>default -&gt; Res(w);</td>
</tr>
<tr>
<td></td>
<td>Res(w)</td>
<td>:</td>
</tr>
</tbody>
</table>

- We match with request(y:oby).
- C(w,a) and P(a,w) exist.
- Cannot find z such that P(z,w), G(z) and C(a,z).
- Therefore, add Res(w).
Building rule activations

- As we go through the trace we build up a set of rule activations.

<table>
<thead>
<tr>
<th>Trace</th>
<th>Rule Activations</th>
</tr>
</thead>
<tbody>
<tr>
<td>conflict(w,a)</td>
<td>Start()</td>
</tr>
<tr>
<td>priority(a,w)</td>
<td>C(w,a)</td>
</tr>
<tr>
<td>request(w)</td>
<td>C(a,w)</td>
</tr>
<tr>
<td>grant(w)</td>
<td>P(a,w)</td>
</tr>
<tr>
<td>request(a)</td>
<td>G(w)</td>
</tr>
<tr>
<td>rescind(w)</td>
<td>Res(w)</td>
</tr>
</tbody>
</table>

```
state Res(w) {
  rescind(w) -> Ok;
  shutdown -> Fail;
}
```

- We match with `rescind(w)` directly.
- **state** rule activations are removed when they fire.

---

Building rule activations

- As we go through the trace we build up a set of rule activations.

<table>
<thead>
<tr>
<th>Trace</th>
<th>Rule Activations</th>
</tr>
</thead>
<tbody>
<tr>
<td>conflict(w,a)</td>
<td>Start()</td>
</tr>
<tr>
<td>priority(a,w)</td>
<td>C(w,a)</td>
</tr>
<tr>
<td>request(w)</td>
<td>C(a,w)</td>
</tr>
<tr>
<td>grant(w)</td>
<td>P(a,w)</td>
</tr>
<tr>
<td>request(a)</td>
<td>G(w)</td>
</tr>
<tr>
<td>rescind(w)</td>
<td>Res(w)</td>
</tr>
</tbody>
</table>

```
always G(w) {
  cancel(w) -> !G(w);
  request(y:obj), C(w,y), P(y,w) {:
    P(z:obj,y), G(z), C(y,z) -> true;
    default -> Res(w);
  };
}
```

- We match directly on `cancel(w)`.
- We can explicitly remove rule activations.
Building rule activations

As we go through the trace we build up a set of rule activations.

<table>
<thead>
<tr>
<th>Trace</th>
<th>Rule Activations</th>
</tr>
</thead>
<tbody>
<tr>
<td>conflict(w,a)</td>
<td>Start()</td>
</tr>
<tr>
<td>priority(a,w)</td>
<td>C(w,a)</td>
</tr>
<tr>
<td>request(w)</td>
<td>C(a,w)</td>
</tr>
<tr>
<td>grant(w)</td>
<td>P(a,w)</td>
</tr>
<tr>
<td>request(a)</td>
<td></td>
</tr>
<tr>
<td>rescind(w)</td>
<td></td>
</tr>
<tr>
<td>cancel(w)</td>
<td></td>
</tr>
<tr>
<td>grant(a)</td>
<td></td>
</tr>
</tbody>
</table>

always Start()

```java
conflict(x:obj,y:obj)
  -> C(x,y),C(y,x);
priority(x:obj,y:obj)
  -> P(x,y);
grant(x:obj) -> G(x);
}
```

We match with grant(x:obj).

Building rule activations

The final set does not contain Res - therefore we have a success.

<table>
<thead>
<tr>
<th>Trace</th>
<th>Rule Activations</th>
</tr>
</thead>
<tbody>
<tr>
<td>conflict(w,a)</td>
<td>Start()</td>
</tr>
<tr>
<td>priority(a,w)</td>
<td>C(w,a)</td>
</tr>
<tr>
<td>request(w)</td>
<td>C(a,w)</td>
</tr>
<tr>
<td>grant(w)</td>
<td>P(a,w)</td>
</tr>
<tr>
<td>request(a)</td>
<td></td>
</tr>
<tr>
<td>rescind(w)</td>
<td></td>
</tr>
<tr>
<td>cancel(w)</td>
<td></td>
</tr>
<tr>
<td>grant(a)</td>
<td></td>
</tr>
<tr>
<td>shutdown</td>
<td></td>
</tr>
</tbody>
</table>

G(a)
The details

- We have seen how a RuleR rule system is evaluated for a given trace.
- We now look at the underlying algorithm.

Structure

- The monitor keeps track of a set of states called a frontier.
- Each state consists of a set of rule activations.
- The monitor has access to a rule system containing rule definitions.
- The monitor uses an observation to update the frontier, and computes a result based on this.
Rules and rule activations

- **A Rule Definition**
  - Has a name.
  - Has a modifier.
  - Is parameterised by (typed) variables.
  - Associates conditions with obligations i.e., \( \text{cancel}(x) \rightarrow \text{!G}(x) \).

- **A Rule Activation**
  - Is associated with a Rule Definition by its name.
  - Contains a binding for the rule’s variables.
  - We can think of this as an instantiation of the rule with the binding.

```plaintext
always G(x:obj){
  cancel(x) -> !G(x);
  request(y:obj), C(x,y),
P(y,x) {:
    P(z:obj,x),G(z),
    C(y,z) -> true;
  default -> Res(x);
  :}
}
```

```plaintext
always G(wheels){
  cancel(wheels) -> !G(wheels);
  request(y:obj), C(wheels,y),
P(y,wheels) {:
    P(z:obj,wheels),G(z),
    C(y,z) -> true;
  default -> Res(wheels);
  :}
}
```

The frontier of states

- We call a set of rule activations a **State**.
- A specification can contain non-determinism through a choice of obligations.
- Therefore, the current configuration of the monitor is represented by a set of states called a **Frontier**.
- Conceptually a state is in conjunction, whereas a frontier is in disjunction.
- A rule system represents an infinite state machine.
- The approach of expanding the frontier is a method for non-deterministically searching this state machine.
High level algorithm

At a high level, we can view the RuleR algorithm as adding the observation to the frontier, firing all activated rules and then checking for inconsistency.

1. create an initial frontier with initials rule activations
2. FOREACH observation
3. -Merge observation state across the frontier
4. -use activated rules to generate a successor set of states
5. -union successor sets to form the new frontier
6. -if no self-consistent state exists we have failed
7. -if a state reduces to true we have succeeded

The algorithm : setup

1. frontier : Set[State];
2. frontier := { initials } ;
3. RS : Map[String, Rule];
4. RS := rule definitions;
5. foreach obs ∈ trace do
   6. frontier := PROCESS(obs);
   7. output CHECK(frontier)

- We store the initial set of rule activations in the frontier.
- The rule system is represented as a map from rule names (strings) to rules.
- For each observation update the frontier and compute the appropriate result.
The algorithm : processing observations

1. \textsc{Process} (obs):

2. \texttt{newF} = \emptyset;

3. \foreach \texttt{s} \in \texttt{frontier} do

4. \quad \texttt{S} = \{ \{ \texttt{ra} \in \texttt{s} \mid \texttt{RS(ra.name).mod} \neq \texttt{step} \} \};

5. \quad \foreach \texttt{ra} \in \texttt{s} do

6. \quad \quad \texttt{rule} = \texttt{RS(ra.name)};

7. \quad \quad \foreach \texttt{(c} \rightarrow \texttt{O)} \in \texttt{rule.body} do

8. \quad \quad \quad \foreach \texttt{b} \in \texttt{unify(ra, c, s)} do

9. \quad \quad \quad \quad \texttt{S} = \{ \texttt{s'} \cup \texttt{b(o)} \mid \texttt{s'} \in \texttt{S}, \texttt{o} \in \texttt{O} \};

10. \quad \quad \texttt{if \ r.mod=state} then

11. \quad \quad \quad \quad \texttt{S} = \{ \texttt{s'} \in \texttt{S} \mid \texttt{assert} \neq \emptyset \};

12. \quad \texttt{return newF};

- For each state
- Create a new set of states of persistent rule activations
- For each rule activation and each rule part
- For each binding that would make the condition true, expand the new states with the instantiated obligations
- If it is a state rule activation, remove it
- Remove any states that do not have an assert rule

The algorithm : checking the frontier

1. \textsc{Check} (frontier):

2. \texttt{collapse frontier};

3. \texttt{if \ frontier= \emptyset} then

4. \quad \texttt{return False}

5. \texttt{if \ \exists \texttt{s} \in \texttt{frontier} \ s = \emptyset \ or \ s \cap \texttt{Success} \neq \emptyset} then

6. \quad \texttt{return True}

7. \texttt{if \ \exists \texttt{s} \in \texttt{frontier}. \ s \cap \texttt{Forbidden} == \emptyset} then

8. \quad \texttt{return Still_True;}

9. \texttt{else}

10. \quad \texttt{if \ \forall \texttt{s} \in \texttt{frontier}. \ s \cap \texttt{Forbidden} \neq \emptyset} then

11. \quad \quad \texttt{return Still.False;}

12. \quad \texttt{else}

13. \quad \quad \texttt{return Unknown;}

- Collapse the frontier by removing inconsistent states
- If the frontier is now empty then there are no paths on which we have met our obligations
- If a state in the frontier is empty or contains a Success rule activation then we have met all obligations on that path
- If there is a state with no forbidden rule activations we are currently meeting obligations
- If all states have forbidden rule activations no paths meet obligations
- The result may be unknown
A second example

- Let us consider a second example.
- This involves the command subsystem of the Rover.
- A command has a name and an id i.e. $\text{command}(\text{name}, \text{id})$.

**Requirement Commands**

A Command should be successful before the end of the system and no other command with the same name may be issued before it is successful. Commands ids should be strictly increasing and all replies should be received within one minute of sending.

The instrumentation

- We instrument the code to record the events:
  - $\text{command}(\text{name}, \text{id})$
  - $\text{fail}(\text{name}, \text{id})$
  - $\text{succeed}(\text{name}, \text{id})$
- When constructing the monitor we set timed mode to true.
- In timed mode the dispatch method adds a timestamp to the event - this adds an additional long parameter.
The specification

```plaintext
ruler Commands{
  observes command(string,int,long), success(string,int,long),
  fail(string,int,long), shutdown;

  state Start(max_id:int) {
    command(name:string,id:int,time:long){:
      max_id>id -> Fail;
      default -> Com(name,id,time), Start(id);
    }
  }

  state Com(name:string,id:int,time:long){
    command(name,x:int,t:long) -> Fail;
    success(name,id,t){:
      t> time-1000 -> Fail;
      default -> Ok;
    }
    fail(name,id,t){:
      t> time-1000 -> Fail;
      default -> Com(name,id,time);
    }
    shutdown -> Fail;
  }

  initials Start(0);
  forbidden Com;
}
```

Summary

- **RULER** is a very expressive system that captures specifications via rule systems.
- For example, we can embed METATEM-like quantified temporal logic in RULER.
- Rules are used to rewrite sets of rule activations (facts) and a set of conditions on these sets determines the verdict given.
- Rules can be parameterised and activated and deactivated.
- There are additional features that have not been demonstrated here including non-determinism, monitor chaining and parameterising rules with rule activations.
References


Rule Systems for Runtime Verification: A Short Tutorial H. Barringer, K. Havelund, D. Rydeheard, A. Groce
Recap

- We have seen three DSLs (Domain Specific Languages) for RV:
  - TraceMatches
  - JavaMOP
  - RuleR
- TraceMatches and JavaMOP focus on efficiency while RuleR focuses on expressiveness.
- These DSLs are so-called external DSLs requiring special parsers.
- Developing and modifying such an external DSL is time consuming.
- Expressive power is limited by logic.
In this lecture

- We shall explore an alternative to external DSLs: internal DSLs.
- Specifically the TraceContract internal DSL written in Scala.
- Introduction to TraceContract.
- Implementation of TraceContract.
- Specification of resource management properties.

External versus internal DSL

- **External DSL**
  - small language typically with very focused functionality
  - specialized parser
  - **pros:**
    - can be optimally succinct
    - “easy” to learn for person not familiar with programming language
    - analyzable: a spec can be analyzed easily, visualized, etc.

- **Internal DSL**
  - an extension of an existing programming language
  - typically an API - using base language’s features only
  - **pros:**
    - easier to develop and later adapt
    - expressive, the programming language is never far away
    - allows use of existing tools such as type checkers, IDEs, etc.
Examples

- **External DSLs:**
  - JavaMOP
  - TraceMatches
  - RuleR

- **Internal DSL:**
  - TraceContract

- **Hybrid:**
  - AspectJ – a syntactic extension of Java

The broader perspective

- Programming languages are becoming increasingly advanced, approaching formal specification languages, such as VDM, ASML, Z, etc.
- It is natural to express specifications in a high-level programming language/scripting language.
- It is furthermore natural to also allow for temporal specifications to be expressed in the programming language.
- The combination of high-level programming and logic is powerful for runtime verification.
- Programmers feel comfortable if they have a real programming language underneath the logic, as “plan B”.
Developed in the Scala programming language.

An internal DSL (API), hence an extension of Scala.

Sandbox experimental combination of parameterized:
  ▶ state machines allowing named as well as un-named states.
  ▶ future time Linear Temporal Logic (LTL).
  ▶ rule-based programming for past time properties.

Expressive and easy to implement and modify.

LTL part is based on formula rewriting.
  ▶ □ p = p ∧ □(□ p)
  ▶ ♦ p = p ∨ ♦(♦ p)
  ▶ p U q = q ∨ (p ∧ □(p U q))

Scala is a high-level unifying language

Object-oriented
Functional
Strongly typed
Script-like, semicolon inference, type inference
Sets, list, maps, iterators, comprehensions
Lots of libraries
Compiles to JVM
Current applications of TraceContract

- **LADEE**
  - “Lunar Atmosphere and Dust Environment Explorer”.
  - developed at: NASA Ames Research Center.
  - purpose: to assess the Lunar atmosphere and the nature of dust above the surface.
  - **TraceContract**: used for checking command sequences against flight rules before sent to spacecraft.

- **SMAP**
  - “Soil Moisture Active Passive”.
  - developed at: NASA’s Jet Propulsion Laboratory.
  - purpose: will provide global measurements of soil moisture on Earth.
  - **TraceContract**: used for checking logs produced by running system against requirements.
Commands must succeed

- We are analyzing log files containing information about commands being issued, and their success and failure respectively.

**Requirement CommandMustSucceed**
An issued command must succeed, without a failure to occur before then.

**Events in TraceContract**

- First we need to define the events we observe:
  - commands being issued, each having a name and a number
  - successes of commands
  - failures of commands
- Each event type sub-classes a type: Event
- **case**-classes allow for pattern matching over objects of the class

```scala
abstract class Event

3 case class Command(name: String, nr: Int) extends Event
4 case class Success(name: String, nr: Int) extends Event
5 case class Fail (name: String, nr: Int) extends Event
```
Property in LogScope

- For comparison we first show the specification in the external DSL: LogScope, which was inspiration for TraceContract.
- A **hot** state must be exited before end of log (non-final state).

    ```scala
    monitor CommandMustSucceed {
      always {
        Command(n, x) => RequireSuccess(n, x)
      }
    }

    hot RequireSuccess(name, number) {
      Fail(name, number) => error
      Success(name, number) => ok
    }
    }
    ```

Property in TraceContract - looks very similar

- Uses partial functions: `{case ... => ...}` defined with pattern matching as arguments to DSL functions (require and hot) defined in Monitor class. RequireSuccess is a user-defined function representing a state.
- A quoted name, such as `'name'` represents the value of that name.

    ```scala
    class CommandMustSucceed extends Monitor[Event] {
      require {
        case Command(n, x) => RequireSuccess(n, x)
      }

      def RequireSuccess(name: String, number: Int) =
      hot {
        case Fail('name', 'number') => error
        case Success('name', 'number') => ok
      }
    }
    ```
Inlining the call of \textit{RequireSuccess}(n, x)

- Since \textit{RequireSuccess}(n, x) is a function, the call of it can be inlined.
- After all, this is “just” a program and standard program transformation works.
- The result is an interesting temporal logic like specification with an un-named hot state.

```
1 class CommandMustSucceed extends Monitor[Event] {
2     require {
3         case Command(n, x) =>
4             hot {
5                 case Fail('n', 'x') => error
6                 case Success('n', 'x') => ok
7             }
8         }
9 }
```

Same property in LTL

- \textsc{TraceContract} also offers future time linear temporal logic (LTL).
- allowing to write events as formulas, negations, propositional formulas, and temporal.
- $\phi$ until $\psi$ means: $\psi$ must eventually hold, and until then $\phi$ must hold.

```
1 class CommandMustSucceed extends Monitor[Event] {
2     require {
3         case Command(n, x) =>
4             not(Fail(n, x)) until (Success(n, x))
5     }
6 }
```

- note mix of Scala’s pattern matching (to catch arguments of command) and LTL.
10 first commands must succeed

**Requirement First10CommandsMustSucceed**

The first 10 issued commands must succeed, without a failure to occur before then.

---

**Counting: first 10 commands must succeed**

- Code (here counting and testing on counter) can be mixed with logic.
- That is: increase counter and return LTL formula.

```java
class First10CommandsMustSucceed extends Monitor[Event] {
    var count = 0
    require {
        case Command(n, x) if count < 10 =>
            count = count + 1
            not(Fail(n, x)) until (Success(n, x))
    }
}
```
Max one success for a command

**Requirement MaxOneSuccess**

An issued command can succeed at most once.

**Using the state formula**

- Previously we saw the *hot* state: we stay in a hot state until a transition fires. It is an error to end up in a hot state at the end of the log (it is a non-final state).
- The *state* state has the same semantics, except that it is a final state.

```java
1 class MaxOneSuccess extends Monitor[Event] {
2     require {
3         case Success(_, number) =>
4             state {
5                 case Success(_, 'number') => error
6             }
7     }
8 }
```
Alternation

Requirement AlternatingCommandSuccess

Commands and successes should alternate.

State machine solution

```java
class AlternatingCommandSuccess extends Monitor[Event] {

    property(s1)

    def s1: Formula =
        state {
            case Command(n, x) => s2(n, x)
            case _ => error
        }

    def s2(name: String, number: Int) =
        state {
            case Success('name', 'number') => s1
            case _ => error
        }
}
```
A past time property

- Properties so far have been future time properties: from some event, the future behavior must satisfy some property.
- The following requirement refers to the past of some event (success).

**Requirement SuccessHasAReason**

A success must be caused by a previously issued command.

TraceContract offers limited rule-based programming

- State logic and LTL cannot express this property.
- TraceContract offers a limited form of rule-based programming, were a fact \( f \) (sub-classing class Fact) can be queried \( (f?) \), created \( (f+) \), and deleted \( (f-) \). The result in the latter two cases is True.

```scala
class SuccessHasAReason extends Monitor[Event] {
  case class Commanded(name: String, nr: Int) extends Fact

  require {
    case Command(n, x) => Commanded(n, x) +
    case Success(n, x) =>
      if (Commanded(n, x) ?) Commanded(n, x) −
    else
      error
  }
}
```
The ?- abbreviation

- We can make this monitor simpler by using test-and-set: $f \ ?\$, for a given fact $f$, meaning: *return true iff. the fact $f$ is recorded, delete the fact in any case.*

```scala
1 class SuccessHasAReson extends Monitor[Event] { 
2     case class Commanded(name: String, nr: Int) extends Fact 
3     require { 
4         case Command(n, x) => Commanded(n, x) + 
5         case Success(n, x) => Commanded(n, x) ?– 
6     } 
7 } 
```

Making monitors of monitors

- We can create a new monitor which includes other monitors as sub-monitors. Useful for organizing properties.
- The semantics is the obvious one of conjunction: all monitors will get checked individually.

```scala
1 class CommandRequirements extends Monitor[Event] { 
2     monitor( 
3         new CommandMustSucceed, 
4         new MaxOneSuccess, 
5         new SuccessHasAReson 
6     ) 
7 } 
```
Analyzing a complete trace (log analysis)

- To verify a trace: first create it, then instantiate monitor, and call `verify` method on monitor with trace as argument.

```scala
object TraceAnalysis extends Application {
  val trace: List[Event] = List(
    Command("STOP_DRIVING", 1),
    Command("TAKE_PICTURE", 2),
    Fail("STOP_DRIVING", 1),
    Success("TAKE_PICTURE", 2),
    Success("SEND_TELEMETRY", 42))

  val monitor = new CommandRequirements
  monitor. verify ( trace )
}
```

Alternatively: analyzing event by event (online monitoring)

- To verify a sequence of events: instantiate monitor, and call `verify` method on monitor for each event, and call `end()` if event flow terminates.

```scala
object TraceAnalysis extends Application {
  val monitor = new CommandRequirements
  monitor. verify (Command("STOP_DRIVING", 1))
  monitor. verify (Command("TAKE_PICTURE", 2))
  monitor. verify (Fail("STOP_DRIVING", 1))
  monitor. verify (Success("TAKE_PICTURE", 2))
  monitor. verify (Success("SEND_TELEMETRY", 42))
  monitor. end()
}
```
Result

CommandMustSucceed property violated
Violating event number 3: Fail(STOP_DRIVING,1)
Error trace:
  1=Command(STOP_DRIVING,1)
  3=Fail(STOP_DRIVING,1)

SuccessHasAReason property violated
Violating event number 5: Success(SEND_TELEMETRY,42)
Error trace:
  5=Success(SEND_TELEMETRY,42)
ScalaDoc documentation of API

```scala
def eventuallyP(n: Int)(formula: Formula): Formula
  Eventually true after n steps.

def eventuallyQ(n: Int)(formula: Formula): Formula
  Eventually true in maximally n steps.

def notP(n: Int)(formula: Formula): Formula
  Eventually true in less than n steps.

def factExists[pred: PartialFunction[Fact, Boolean]]: Boolean
  Tests whether a fact exists in the fact database, which satisfies a predicate.

val getMonitorResults: MonitoringResult[Event]
returns the result of a trace analysis for the monitor.

val getSubMonitors: List[Monitor[Event]]
returns the sub-monitor of a monitor.

def globallyQ(formula: Formula): Formula
  Globally true (an LTL formula).

def hot(block: PartialFunction[Event, Formula]): Formula
  A hot state waiting for an event to eventually match a transition (required) between m and n steps.

def not(block: PartialFunction[Event, Formula]): Formula
  A hot state waiting for an event to eventually match a transition (required). The state remains active until the incoming event matches the block, that is, until block.isDefinedAt(x) == true, in which case the state formula evaluates to (not x).

class Requirement extends Monitor[Event] {
  require {
    case COMMAND(x) =>
      hot {
        case SUCCESS(success) => ok
      }
  }
}
```

ScalaDoc documentation of API
Features by category: state functions

```python
class Monitor[Event] {
    class Formula {
        ...
    }
    type Block = Event =?=> Formula

    // waiting states:
    def state (block: Block): Formula
    def hot (block: Block): Formula
    def always (block: Block): Formula

    // next state:
    def weak (block: Block): Formula
    def strong (block: Block): Formula
    def step (block: Block): Formula
    ...
}
```

Error and success

```python
...  
def error : Formula
  def error (message: String): Formula

def ok : Formula
  def ok (message: String): Formula
  ...
```
Temporal logic

... // propositional logic:

```python
def matchEvent (predicate: Event =?= Boolean): Formula
object True extends Formula
object False extends Formula

def not (formula: Formula): Formula
```

... // temporal operators:

```python
def globally (formula: Formula): Formula
def eventually (formula: Formula): Formula
def eventuallyLe (n: Int )(formula: Formula): Formula
def weaknext (formula: Formula): Formula
def strongnext (formula: Formula): Formula
```

Infix operators defined in class Formula

... // propositional logic:

```python
class Formula {
    // propositional logic:
def and (that: Formula): Formula
def or (that: Formula): Formula
def implies (that: Formula): Formula
```

... // temporal operators:

```python
def unless (that: Formula): Formula
def until (that: Formula): Formula
def upto (block: Block): Formula
```
Rule-based programming

```scala
class Fact {
  ...
}

def matchFact (pred: Fact =?=> Boolean): Boolean

implicit def convFact2FactOps (fact: Fact): FactOps

class FactOps {
  def + : Unit
  def - : Unit
  def ? : Boolean
  def ~ : Boolean
  def ?- : Boolean
  def ~+ : Boolean
}
```

Other implicit conversions

- An implicit function is applied to a value by the compiler if the value is not type correct, but the application is.
- Two implicit functions convert events and Booleans to formulas. An event and a Boolean expression can therefore occur as a formula.
- One implicit function converts the unit value to a formula. This means one can write code as a formula (equals True).

```scala
implicit def convEvent2Formula (event: Event): Formula
implicit def convBoolean2Formula (cond: Boolean): Formula
implicit def convUnitToFormula (unit: Unit): Formula
```

- We will see the implementation of these implicit functions later.
Declaring and verifying properties

// declaring properties inside a monitor
def informal (explanation: String): Unit
def property (formula: Formula): Unit
def property (name: Symbol)(formula: Formula): Unit
def require (block: Block): Unit
def requirement (name: Symbol)(block: Block): Unit
def monitor (monitors: Monitor[Event]*): Unit

// verification:
def select (filter: Event =?=> Boolean): Unit
def verify (trace: Trace): MonitorResult[Event]
def verify (event: Event): Unit
def end (): Unit
def getMonitorResult : MonitorResult[Event]

Towards a TraceContract calculus

\[ \langle TC \rangle ::= \text{matchEvent}\{\langle EventPred \rangle \}
\]
\| \text{matchFact}\{\langle FactPred \rangle \}
\| \neg \langle TC \rangle
\| \langle TC \rangle \lor \langle TC \rangle
\| \circ \langle TC \rangle
\| \langle TC \rangle \cup \langle TC \rangle
\| \langle StateKind \rangle \{\langle TransitionBlock \rangle \}
\| \langle BoolExp \rangle
\| \langle Stmt \rangle
\| \langle Fact \rangle (\text{‘+’ | ‘-‘})\]

\[ \langle StateKind \rangle ::= \text{state} | \text{hot} | \text{always} | \text{weak} | \text{strong} | \text{step} \]

\[ \langle EventPred \rangle ::= \text{partial function of type: } \langle Event \rangle \xrightarrow{\sim} \{\text{true, false}\} \]

\[ \langle FactPred \rangle ::= \text{partial function of type: } \langle Fact \rangle \xrightarrow{\sim} \{\text{true, false}\} \]

\[ \langle TransitionBlock \rangle ::= \text{partial function of type: } \langle Event \rangle \xrightarrow{\sim} \langle TC \rangle \]
The following slides represent a complete implementation of a mini-\texttt{TraceContract}, focusing only on conceptual ideas. It is sufficient for the examples provided on previous slides.

\textbf{Rule system}

```scala
trait RuleSystem {
  trait Fact
  var facts: Set[Fact] = Set()
  var toRecord: Set[Fact] = Set()
  var toRemove: Set[Fact] = Set()
  implicit def convE(fact: Fact) = new {
    def + : Unit = { toRecord += fact }
    def − : Unit = { toRemove += fact }
    def ? : Boolean = facts contains fact
    def ¬ : Boolean = !(facts contains fact)
  }
  def matchFact(pred: Fact =?= > Boolean): Boolean = {
    facts exists (pred orElse { case _ => false })
  }
  def updateFacts() {
    toRemove foreach (facts −= _)
    toRecord foreach (facts + = _)
    toRecord = Set()
    toRemove = Set()
  }
}
```
Class Monitor

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

  type Block = Event => Option[Formula]

  trait Formula {
    def apply(e: Event): Formula
    def reduce: Formula = this

    def and(that: Formula) = And(this, that) reduce
    def or(that: Formula) = Or(this, that) reduce
    def until (f: Formula) = Until(this, f)
    def upto(b: Block) = Upto(this, b)
  }

  ... // what follows on the next slides goes here
}
```

True and false

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

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

def error = False
def ok = True
```
Propositional logic

```scala
implicit def convEvent2Formula(e: Event): Formula = Now(e)
implicit def convBoolean(b: Boolean): Formula = if (b) True else False
implicit def convUnit(u: Unit): Formula = True

case class Now(expect: Event) extends Formula {
  def apply(e: Event): Formula =
    if (expect == e) True else False
}

case class matchEvent(p: Event => Boolean) extends Formula {
  def apply(event: Event): Formula =
    if (p.isDefinedAt(event)) p(event)
    else false
}

case class not(f: Formula) extends Formula {
  def apply(e: Event): Formula = not(f(e)).reduce()

  override def reduce(): Formula =
    f match {
      case True => False
      case False => True
      case _ => this
    }
}

case class And(f1: Formula, f2: Formula) extends Formula {
  def apply(e: Event) = And(f1(e), f2(e)) reduce

  override def reduce: Formula =
    (f1, f2) match {
      case (False, _) | (_, False) => False
      case (True, _) => f2
      case (_, True) => f1
      case _ => this
    }
}

case class Or(f1: Formula, f2: Formula) extends Formula {
  ... // same idea
}
```
Temporal Logic

case class Until(f1: Formula, f2: Formula) extends Formula {
  def apply(e: Event): Formula = 
    Or(f2(e), And(this, f1(e)).reduce).reduce
}

case class strongnext(f: Formula) extends Formula {
  def apply(e: Event): Formula = f(e)
}

case class globally(f: Formula) extends Formula {
  def apply(e: Event): Formula = {
    val formula_ = f(e)
    if (formula_ == f)
      this
    else
      And(this, formula_).reduce
  }
}

case class eventually(f: Formula) extends Formula {
  def apply(e: Event): Formula = 
    Or(this, f(e)).reduce
}

State logic

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

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

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

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

case class Upto(f: Formula, b: Block) extends Formula {
  override def apply(e: Event): Formula = {
    if (b.isDefinedAt(e)) {
      isFinal(f) and b(e)
    } else {
      val formula_ = f(e).reduce
      formula_ match {
        case False => False
        case True => True
        case 'f' => this
        case _ => Upto(formula_, b)
      }
    }
  }
}

Property declaration and verification

def monitor(monitorList: Monitor<Event]*) {
  monitors ++= monitorList.toList
}
def property(f: Formula) {
  current = f
}
def require(b: Block) = property(always(b))
def verify(e: Event) {
  val current_ = current(e)
  if (current_ == False && current_ != False)
    println("*** safety violation " + this.getClass().getSimpleName)
  current = current_
  for (monitor <− monitors) monitor.verify(e)
  updateFacts()
}
def verify(trace: List[Event]) {
  for (e <− trace) {
    println("−−− " + e)
    verify(e)
  }
  end()
The end of the trace

```java

def end() {
    if (!isFinal(current)) println("*** liveness violation " + this.getClass().getSimpleName + " on " + e)
    monitors foreach (_end())
}

def isFinal(f: Formula): Boolean = {
    f match {
        case hot(_)
        | strong(_)
        | Now(_)
        | matchEvent(_)
        | Until(_, _)
        | strongnext(_)
        | eventually(_) => false
        case Upto(f, _) => isFinal(f)
        case not(f) => !isFinal(f)
        case And(f1, f2) => isFinal(f1) && isFinal(f2)
        case Or(f1, f2) => isFinal(f1) || isFinal(f2)
        case True
        | False
        | state(_)
        | always(_)
        | globally(_) => true
    }
}
```

TraceContract
applied to resource management
AspectJ, Java, Scala

- The following slides represent TraceContract specifications of resource management properties previously expressed in raw Java.

- Since Scala and Java integrates well, AspectJ can in principle be used together with TraceContract for instrumentation and monitoring of Java code.

Recall the resource management requirements

- **GrantCancel**: For a given resource, grants and cancellations should alternate, starting with a grant. Furthermore: a cancellation should be performed by the same task that was last granted the resource.

- **OnlyRescindGranted**: Only ask a task to rescind the resource if it is currently owned by the task. That is: it has been granted, and it has not yet been cancelled.

- **RespectConflicts**: Conflicts must be respected. For every pair of resources, if they conflict then only one can be granted at any one time.

- **RespectPriorities**: Let priorities sort conflicts. If there is a conflict and the requested resource has the highest priority then the other priority should be rescinded before the resource is granted.
Informative type names

- First some types introduced only for their descriptive names.
- Scala’s class `Any` corresponds to Java’s class `Object`.
- To make types available we “open up” the object `Util` by importing its contents.

```scala
object Util {
  type Actor = Any
  type Resource = Any
  type Response = Any
}
import Util.
```

Declaring events

- As before event classes are declared as sub-classing a type: `Event`

```scala
abstract class Event

case class SendRequest(a: Actor, r: Resource) extends Event
case class SendCancel(a: Actor, r: Resource) extends Event
case class SendGrant(r: Resource, a: Actor) extends Event
case class SendRescind(r: Resource, a: Actor) extends Event
case class SendDeny(a: Actor) extends Event
case class AddConflict(r1: Resource, r2: Resource) extends Event
case class AddPriority(r1: Resource, r2: Resource) extends Event
case class CancelResource(a: Actor, r: Resource) extends Event
case object End extends Event
```
Analyzing a trace, assuming monitor: Requirements

```scala
object TraceAnalysis extends Application {
  val trace: List[Event] =
  List(
    AddConflict('antenna, 'wheels),
    AddConflict('camera, 'wheels),
    AddConflict('drill, 'wheels),
    AddPriority('antenna, 'wheels),
    SendGrant('antenna, 'commandTask),
    SendGrant('wheels, 'drivingTask),
    CancelResource('commandTask, 'antenna),
    CancelResource('cameraTask,'antenna)
  )

  val monitor = new ResourceRequirements
  monitor.verify (trace)
}
```

The requirements

- We build a monitor containing four sub-monitors corresponding to the four requirements.

```scala
class ResourceRequirements extends Monitor[Event] {
  monitor(
    new GrantCancel,
    new OnlyRescindGranted,
    new RespectConflicts,
    new RespectPriorities
  )
}
```

- Let's build the four monitors
Resource Management: grant and cancel alternate

Requirement **GrantCancel**
For a given resource, grants and cancellations should alternate, starting with a grant. Furthermore: a cancellation should be performed by the same task that was last granted the resource.

Splitting into future and past time property

- **Future time**: when observing a grant of a resource to an actor, then no other grant of that resource should be given weak-until it has been cancelled by the same actor:
  \[
  \square(SendGrant(r, a) \Rightarrow \oplus(\neg SendGrant(r, _) \ \mathcal{W} \ \text{CancelResource}(a, r))
  \]

- **Past time**: when observing a cancel of a resource, it should have been granted in the past to that actor, and not yet cancelled since then:
  \[
  \square(CancelResource(a, r) \Rightarrow \ominus(\neg CancelResource(a, r) \ \mathcal{S} \ SendGrant(r, a))
  \]
Future time with states

- **Requirement:** no grant of a resource upon a grant, without a cancel in between.

```scala
class GrantCancel extends Monitor[Event] {
  require {
    case SendGrant(resource, receiver) =>
      state {
        case SendGrant('resource', _) => error
        case CancelResource('receiver', 'resource') => ok
      }
  }
}
```

Future time with LTL

- **Requirement:** no grant of a resource upon a grant, without a cancel in between.
- **The TraceContract expression:** `mathes(f)` returns true iff. the partial function `f : Event => Boolean` is defined for the current event `e` and `f(e) = true`. The function has the type `matches : (Event => Boolean) => Formula`

```scala
class GrantCancel extends Monitor[Event] {
  require {
    case SendGrant(resource, receiver) =>
      not(mathes {
        case SendGrant('resource', _) => true
      }) unless
      CancelResource(receiver, resource)
  }
}
```
LTL and a predicate

- To make the specification simpler to read we can define a function representing any event that grants a particular resource.

```scala
class GrantCancel extends Monitor[Event] {
  def grant(resource: Resource): Formula =
    matches { case SendGrant('resource ', _) => true }

  require {
    case SendGrant(resource, receiver) =>
      not(grant(resource)) unless CancelResource(receiver, resource)
  }
}
```

Past time with rules

- **Requirement:** no cancel of a resource without a previous grant, and no cancel in between.
- Upon event `sendGrant(r,a)` a fact `Granted(r,a)` is stored.
- Upon event `CancelResource(a,r)` it is tested that a fact `Granted(r,a)` exists, upon which this fact is deleted.

```scala
class GrantCancel extends Monitor[Event] {
  case class Granted(resource: Resource, receiver: Actor) extends Fact

  require {
    case SendGrant(resource, receiver) =>
      Granted(resource, receiver) +
    case CancelResource(sender, resource) =>
      Granted(resource, sender) ?–
  }
}
```
Past and future mixing states and rules

```scala
class GrantCancel extends Monitor[Event] {
  case class Granted(resource: Resource, receiver: Actor) extends Fact

  require {
    case SendGrant(resource, receiver) =>
      Granted(resource, receiver) +;
    state {
      case SendGrant('resource', _) => error
      case CancelResource('receiver', 'resource') => ok
    }
    case CancelResource(sender, resource) =>
      Granted(resource, sender) ?–
  }
}
```

Past and future with rules only

- This monitor is similar to the original AspectJ monitor, but eliminates the need for inventing data structures, such as a map from resources to actors.

```scala
class GrantCancel extends Monitor[Event] {
  case class Granted(resource: Resource, receiver: Actor) extends Fact

  require {
    case SendGrant(resource, receiver) =>
      Granted(resource, receiver) +
    case CancelResource(sender, resource) =>
      Granted(resource, sender) ?–
  }
}
```
Resource Management: only rescind granted

Requirement OnlyRescindGranted

Only ask a task to rescind the resource if it is currently owned by the task. That is: it has been granted, and it has not yet been cancelled.

This is a past time logic property, so we store facts

```scala
class OnlyRescindGranted extends Monitor[Event] {
  case class Granted(resource: Resource, receiver: Actor) extends Fact

  require {
    case SendGrant(resource, receiver) =>
      Granted(resource, receiver) +
    case CancelResource(sender, resource) =>
      Granted(resource, sender) -
    case SendRescind(resource, receiver) =>
      Granted(resource, receiver) ?
  }
}
```
Resource Management: respect conflicts

**Requirement RespectConflicts**

Conflicts must be respected. For every pair of resources, if they conflict then only one can be granted at any one time.

**Using states**

```java
1  class RespectConflictsState extends Monitor[Event] {
2    require {
3      case AddConflict(resource1, resource2) =>
4        respectConflict (resource1, resource2) and
5        respectConflict (resource2, resource1)
6    }
7
8    def respectConflict (resource1: Resource, resource2: Resource) =
9      state {
10        case SendGrant('resource1', receiver) =>
11          state {
12            case SendGrant('resource2', _) => error
13            case CancelResource('receiver', 'resource1') => ok
14          }
15      }
16  }
```
Using facts

```scala
class RespectConflictsFacts extends Monitor[Event] {
  case class Granted(resource: Resource) extends Fact
  case class Conflict(resource1: Resource, resource2: Resource)
    extends Fact

  require {
    case AddConflict(resource1, resource2) =>
      Conflict(resource1, resource2) +;
      Conflict(resource2, resource1) +
    case SendGrant(resource, receiver) =>
      Granted(resource) +;
      ! factExists {
        case Conflict('resource', resource2) =>
          Granted(resource2) ?
      }
    case CancelResource(sender, resource) =>
      Granted(resource) –
  }
}
```

Programming it

```scala
class RespectConflictsProgramming extends Monitor[Event] {
  var conflicts: Set[(Resource, Resource)] = Set()
  var granted: Set[Resource] = Set()

  require {
    case AddConflict(resource1, resource2) =>
      conflicts += Set((resource1, resource2), (resource2, resource1))
    case SendGrant(resource, receiver) =>
      granted += resource
      ! conflicts.exists {
        case (r1, r2) => r1 == resource && granted.contains(r2)
      }
    case CancelResource(sender, resource) =>
      granted -= resource
  }
}
```
Resource Management: respect priorities

**Requirement RespectPriorities**

Let priorities sort conflicts. If there is a conflict and the requested resource has the highest priority then the other priority should be rescinded before the resource is granted.

Assume 2 resources

```java
class RespectPriorities extends Monitor[Event] {
    require {
        case AddConflict(r1, r2) => state {
            case AddPriority(hi, lo) if (hi, lo) == (r1, r2) =>
                always {
                    case SendGrant('lo', actorLo) => state {
                        case CancelResource('actorLo', 'lo') => ok
                        case SendRequest(actorHi, 'hi') => state {
                            case SendGrant('hi', 'actorHi') => error
                            case SendRescind('lo', 'actorLo') => ok
                        }
                    }
                }
            }
        }
    }
}
```
Curious about how `==` is defined?

```scala
implicit def conv[A](pair1: (A, A)) = new {
  def `==`(pair2: (A, A)) = {
    val (a1, a2) = pair1
    val (b1, b2) = pair2
    ((a1 == b1) && (a2 == b2)) || ((a1 == b2) && (a2 == b1))
  }
}
```

Summary

- `TraceContract` is an API.
- Very expressive and convenient for programmers to use.
- For this reason mainly it has been adopted by practitioners.
- Has very simple implementation, which is easy to modify.
- Change requests are easy to process.
- It is, however, difficult to analyze a `TraceContract` specification since it fundamentally is a Scala program - requires some form of reflection or interaction with compiler.
- It will not be suitable for non-Scala programmers.
References

