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.

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 an 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

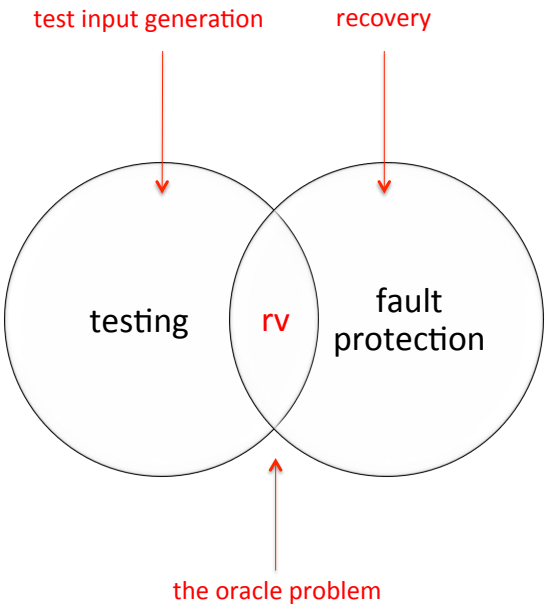
- Runtime verification
- Runtime monitoring
- Runtime checking
- Runtime analysis
- Dynamic analysis
- Trace/log analysis
- Fault protection
- Runtime enforcement

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

technique	automated	scalable	coverage	properties
model checking	yes	no	finite	complex
theorem proving	no	no	complete	complex
static analysis	yes	yes	complete	simple
runtime verification	yes	yes	low	**complex**

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

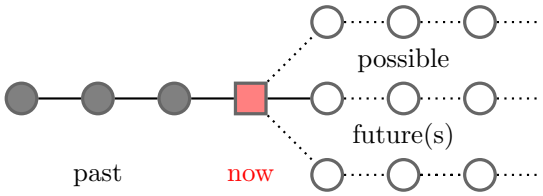
- *Events* record runtime behavior
  - snapshots of state or actions performed
- A finite sequence of events is a *trace* τ
- A *property* ϕ denotes a *language* $\mathcal{L}(\phi)$ (a set of traces)
- τ satisfies ϕ iff $\tau \in \mathcal{L}(\phi)$

Viewing the execution as a *trace*

A trace σ 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

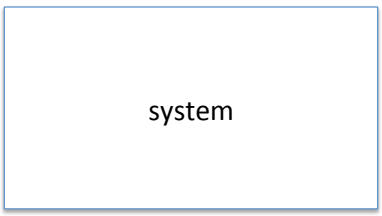
Giving verdicts along the way

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

	current trace τ	all suffixes σ	Action
1	$\tau \in \mathcal{L}(\varphi)$	$\tau\sigma \in \mathcal{L}(\varphi)$	stop with Success T
2	$\tau \in \mathcal{L}(\varphi)$	unknown	carry on monitoring T_{still}
3	$\tau \notin \mathcal{L}(\varphi)$	$\tau\sigma \notin \mathcal{L}(\varphi)$	stop with Failure F
4	$\tau \notin \mathcal{L}(\varphi)$	unknown	carry on monitoring F_{still}

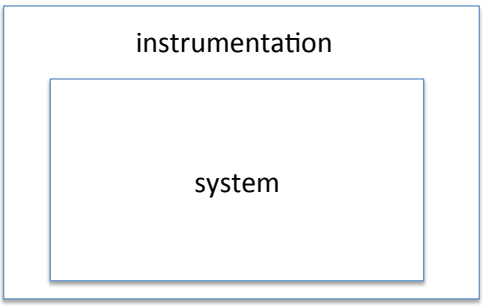
Runtime verification in practice

- Start with a system to monitor.



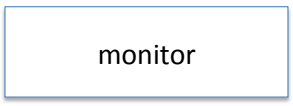
Runtime verification in practice

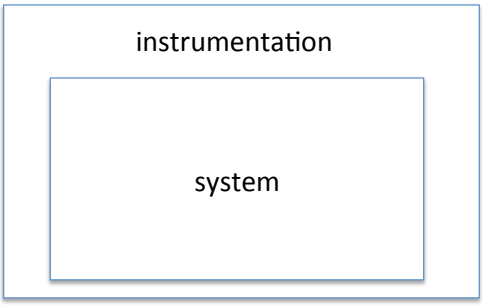
- *Instrument* the system to record relevant events.



Runtime verification in practice

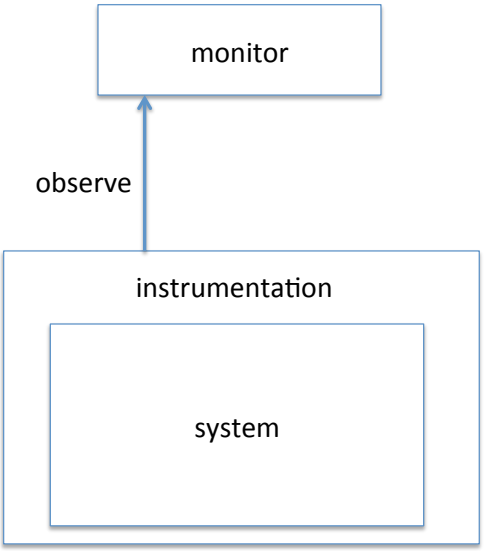
- *Provide* a monitor.





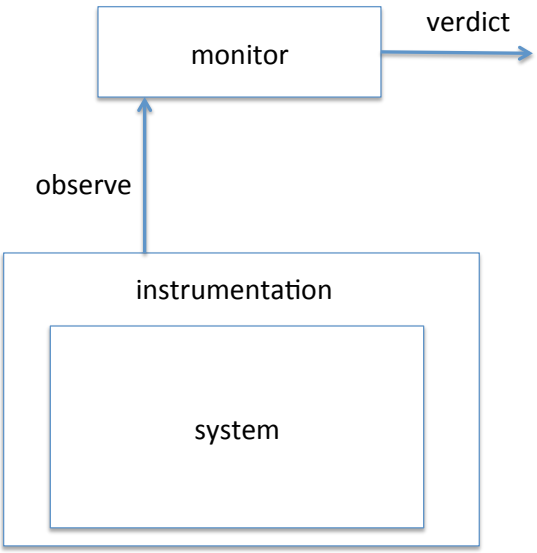
Runtime verification in practice

- *Dispatch* each received event to the monitor.



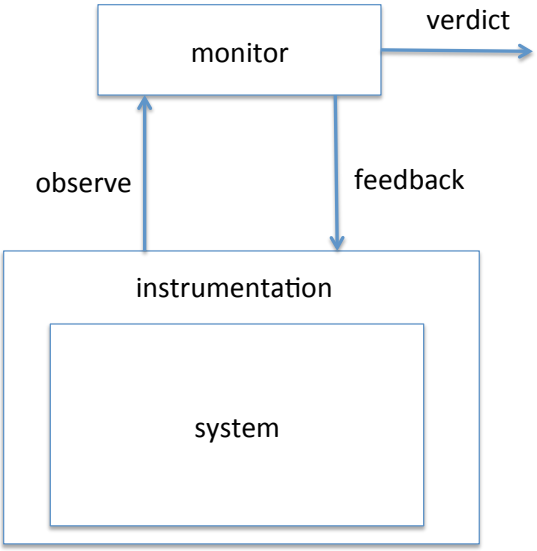
Runtime verification in practice

- Compute a *verdict* for the trace received so far.



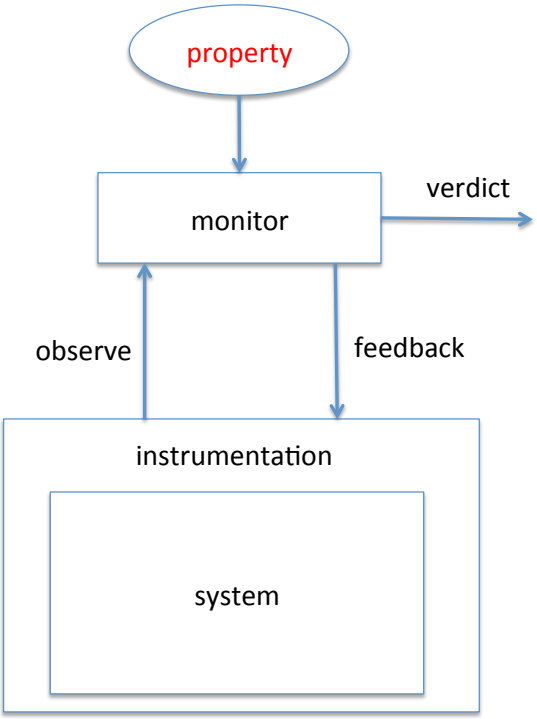
Runtime verification in practice

- Possibly generate *feedback* to the system.



Runtime verification in practice

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



Generating the trace

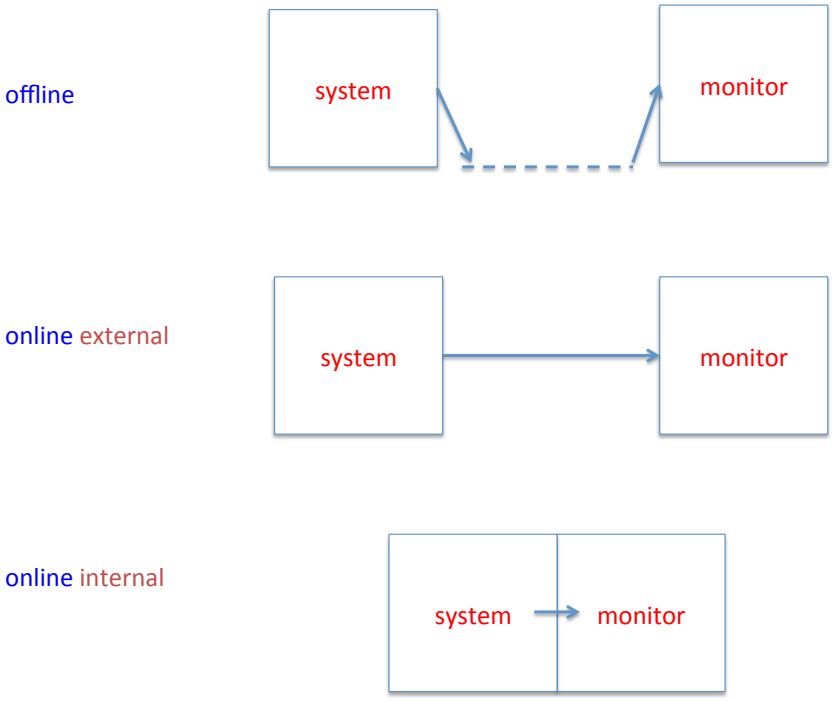
The concrete execution of the program needs to be abstracted:

- Discrepancy:
  - events of the program in Σ_c
  - events of the specification in Σ_a
- Define a function $\Sigma_c \rightarrow \Sigma_a \cup \{irrelevant\}$
- 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?

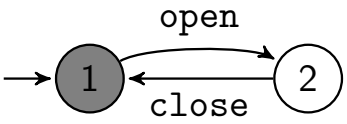
- Program (built-in algorithm focused on specific problem)
  - data race detection
  - atomicity violation
  - deadlock detection
- Programming language
- Design by contract (pre/post conditions), JML for example
- Logic (formal system)
  - state machines
  - regular expressions
  - grammars (context 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

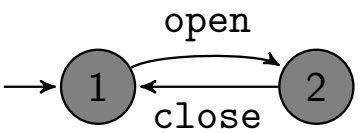
- Record *propositional* events, for example
  - `open`, `close`
- Define a property over propositional events, for example
  - LTL (finite-trace) $\quad \Box(\texttt{open} \rightarrow \bigcirc(\neg\texttt{open}\, \mathcal{U}\, \texttt{close}))$
  - RE $\quad (\texttt{open.close})^*$
  - DFA



- Check if each trace prefix is in the language of the property

Using Projection

- With the property



- Take the trace

`open.read.write.close.open.read.close`

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

`open.close.open.close`

Going parametric

- Consider the code

```
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

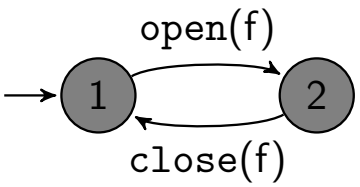
- Not good, we want to *parameterize* events with data values and use those values in the specification
- Instead record the parametric trace

open(manual.pdf).open(readme.txt).close(readme.txt).close(manual.pdf)

Parametric properties

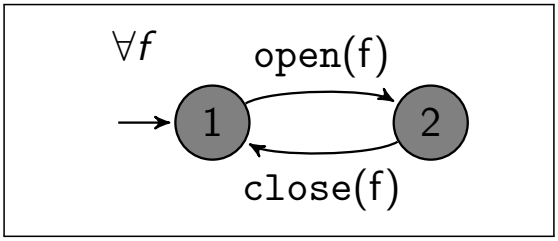
- Using the events
  - open(f) when file f is opened
  - close(f) when file f is closed

- the property becomes



From parametric to quantified

Quantify over variables in parametric property:



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`
- instrumentation of *byte/object code*
  - Valgrind (C) `http://valgrind.org`
  - BCEL (Java) `http://jakarta.apache.org/bcel`
- aspect-oriented programming (AOP):
  - AspectC (C) `https://sites.google.com/a/gapp.msrg.utoronto.ca/aspectc`
  - AspectC++ (C++) `http://www.aspectc.org`
  - AspectJ (Java) `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.

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

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
- ...

Runtime Verification 2012 - Istanbul, Turkey

Keynote Speakers

Jim Larus (MSR) on Programming the Cloud
Martin Rinard (MIT EECS and CSAIL) on From Runtime Verification to Runtime Intervention and Adaptation
Giovanni Vigna (UCSB) on Malware Riding Badware: Challenges in Analyzing (Malicious/Benign) Web Applications

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.

References


Klaus Havelund and Allen Goldberg: *Verify Your Runs*. Verified Software: Theories, Tools, Experiments (VSTTE'05), 2005.


Martin Leucker and Christian Schallhart: *A Brief Account of Runtime Verification*. Journal of Logic and Algebraic Programming, Volume 78, Issue 5, May-June 2009.

Part II

Instrumentation with AspectJ
Specification with Java

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

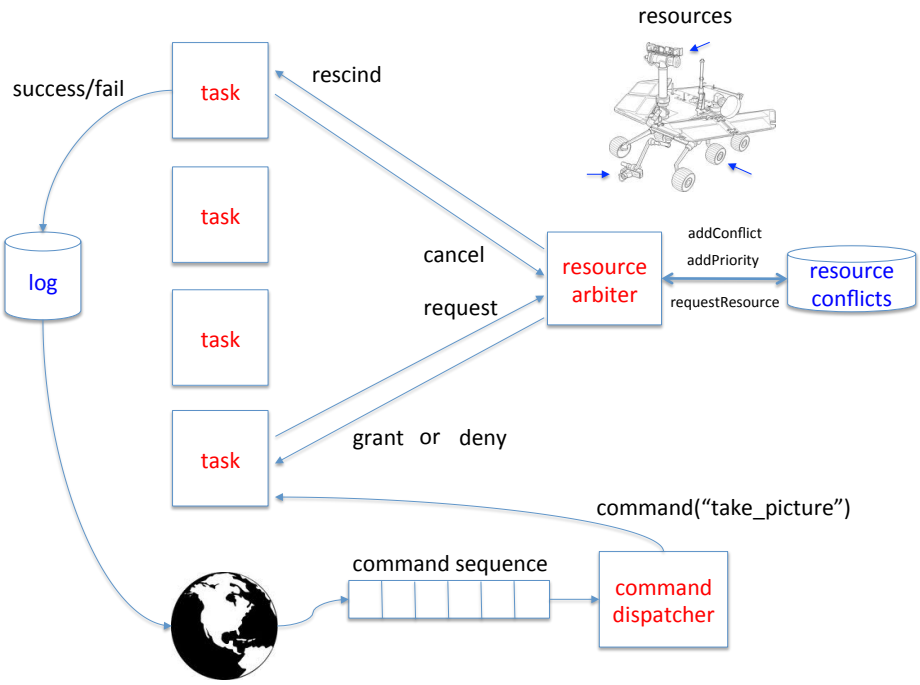
- We will demonstrate how program monitors can be written in JAVA using `AspectJ` for code instrumentation

- 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`

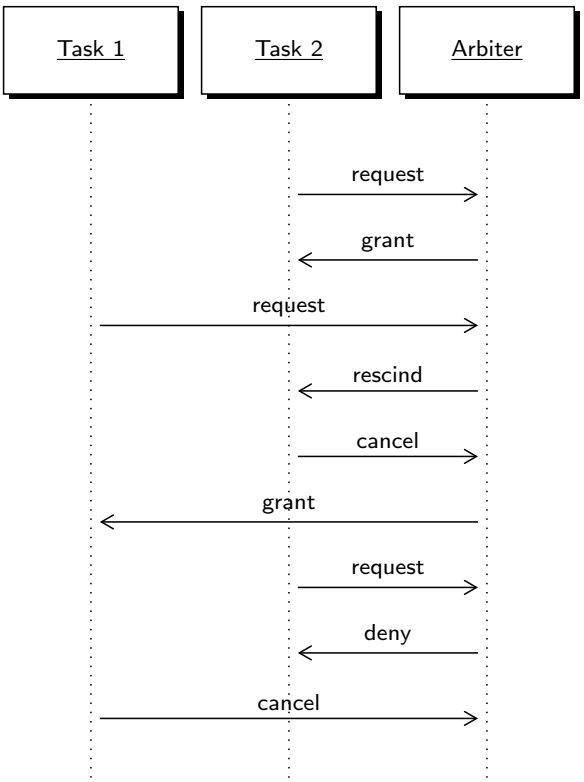
Rover example

- 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

Systems architecture



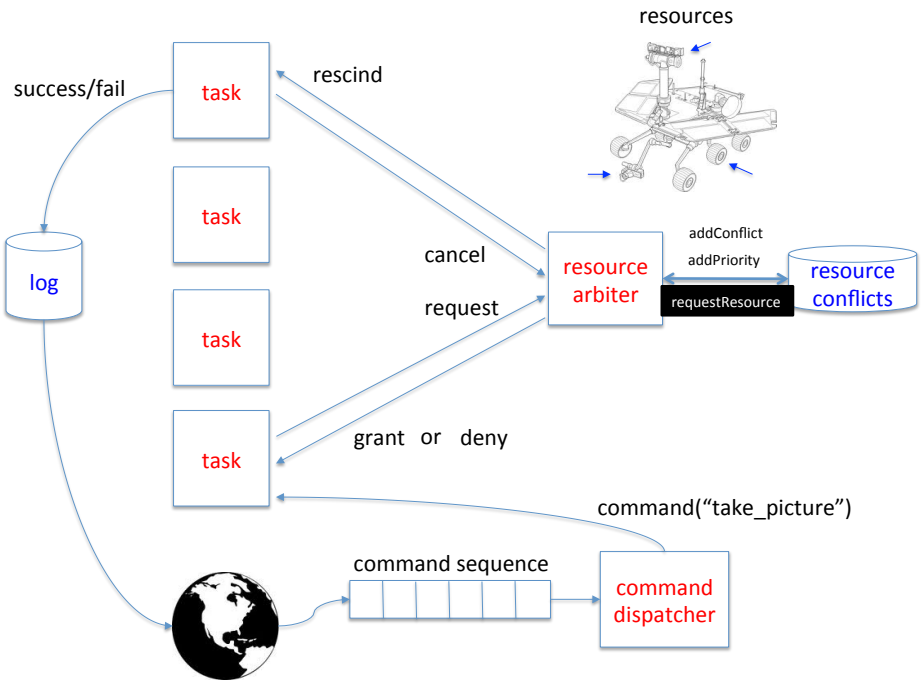
Example of resource acquisition sequence



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.

```
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

```
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

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

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:*
  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)

```
/*@ 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  }
```

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.

```
1  Response requestResource(Actor actor, String name) {
2    assert resources.containsKey(name);
3    Resource resource = resources.get(name);
4    boolean oldIsAvailable = isAvailable(resource);
5    List<Resource> oldInConflict = getActiveConflicts(resource);
6    Response result = null;
7    ...
8    assert post_requestResource(actor, name, oldIsAvailable, oldInConflict, result);
9    return result;
10 }
```

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)

```
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

	remoteness	
temporal	internal to code	external to code
assertions	**assert**	`AspectJ` + **assert**
design by contract	JML	`AspectJ` + **assert**
temporal logic	JML + TL	`AspectJ` + **assert** + history

- 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

```
1  public class Hello {
2    public static void main(String[] args) {
3      System.out.println("this is");
4      System.out.println("hello");
5    }
6  }
```

```
1  public aspect World {
2    after(String s) :
3      call(void java.io.PrintStream.println(String))
4      && args(s)
5      && if (s.equals("hello"))
6    {
7      System.out.println("world");
8    }
9  }
```

this is
hello
world

Hello world AspectJ example

```
1  public class Hello {
2    public static void main(String[] args) {
3      System.out.println("this is");
4      System.out.println("hello");
5    }
6  }
```

```
1  public aspect World {
2    after(String s) : // advice
3      call(void java.io.PrintStream.println(String)) // pointcut
4      && args(s) // pick up argument
5      && if (s.equals("hello")) // condition
6    {
7      System.out.println("world"); // do this after such a call
8    }
9  }
```

```
this is
hello
world
```

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.

Pointcuts

- **call**(⟨*methodPat*⟩) : call of a method
- **execution**(⟨*methodPat*⟩) : body of method
- **get**(⟨*fieldPat*⟩) : reading from a field
- **set**(⟨*fieldPat*⟩) : writing to a field
- **handler**(⟨*typePat*⟩) : exception handler
- **adviceexecution**() : within any advice
- **within**(⟨*typePat*⟩) : within class
- **withincode**(⟨*methodPat*⟩) : within method
- **this**(⟨*type*⟩ | ⟨*var*⟩) : current object
- **target**(⟨*type*⟩ | ⟨*var*⟩) : object on which method is called
- **args**(⟨*type*⟩ | ⟨*var*⟩ . . .) : arguments of method call
- **if**(⟨*expression*⟩) : conditional
- **cflow**(⟨*pointcut*⟩) : any join point in the control flow of pointcut
- ...

Recall pre- and post-condition for method requestResource

```
/*@ 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

```
1  privileged public aspect PrePostRequestResource {
2     ... // declaration of data fields
3
4     pointcut requestResource(ResourceTable table, String name) :
5       call(Response ResourceTable.requestResource(Actor,String))
6       && args(*, name) && target(table);
7
8     before(ResourceTable table, String name) :
9       requestResource(table, name) {
10          ... // check pre-condition
11    }
12
13    after(ResourceTable table, String name)
14    returning (Response result) :
15      requestResource(table, name) {
16          ... // check post-condition
17    }
18 }
```

Declaring data in aspect

```
1  privileged public aspect PrePostRequestResource {
2     Resource resource;
3     boolean oldIsAvailable;
4     List<Resource> oldInConflict;
5
6     pointcut requestResource(ResourceTable table, String name) :
7       call(Response ResourceTable.requestResource(Actor,String))
8       && args(*, name) && target(table);
9
10    before(ResourceTable table, String name) :
11      requestResource(table, name) {
12          ... // check pre-condition
13    }
14
15    after(ResourceTable table, String name)
16    returning (Response result) :
17      requestResource(table, name) {
18          ... // check post-condition
19    }
20 }
```

The pre-condition, and preparing for post-condition

```
1  privileged public aspect PrePostRequestResource {
2    Resource resource;
3    boolean oldIsAvailable;
4    List<Resource> oldInConflict;
5
6    pointcut requestResource(ResourceTable table, String name) :
7      call(Response ResourceTable.requestResource(Actor,String))
8      && args(*, name) && target(table);
9
10   before(ResourceTable table, String name) :
11     requestResource(table, name) {
12     assert table.resources.containsKey(name);
13     resource = table.resources.get(name);
14     oldIsAvailable = table.isAvailable(resource);
15     oldInConflict = table.getActiveConflicts(resource);
16   }
17
18   after(ResourceTable table, String name)  ... { ... }
19 }
```

The post-condition

```
1  privileged public aspect PrePostRequestResource {
2    Resource resource;
3    boolean oldIsAvailable;
4    List<Resource> oldInConflict;
5
6    pointcut requestResource(ResourceTable table, String name) :
7      call(Response ResourceTable.requestResource(Actor,String))
8      && args(*, name) && target(table);
9
10   before(ResourceTable table, String name) ... { ... }
11
12   after(ResourceTable table, String name)
13   returning (Response result) :
14     requestResource(table, name) {
15     if (result.getResource() != null && result.getRescinds() == null) {
16       assert oldIsAvailable && oldInConflict.isEmpty();
17     }
18   }
19 }
```

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.

```
1  public class Arbiter extends Actor {
2    ResourceTable table = new ResourceTable();
3
4    public void run() {
5      while (true) {
6        Message message = receive();
7        if (message instanceof Request) {
8          Response response = table.requestResource(requester,resourceName);
9          ...
10         if (resource == null && rescinds == null) {
11           requester.sendDeny();
12         } else if (rescinds == null) {
13           requester.sendGrant(resource);
14         } else ...
15       } else else if (message instanceof Cancel) { ... }
16     }
17   }
18 }
```

The instrumentation aspect

```
1  privileged aspect Instrument {
2    // List of all monitors to be notified on each new event
3    List<Monitor> monitors = new ArrayList<Monitor>();
4
5    // Creating the list of monitors in aspect constructor
6    public Aspect() {
7      monitors.add(new monitors.GrantCancel(),...);
8    }
9
10   // Advice for the sendGrant event
11   after(Resource resource, Actor receiver) :
12     call(void missioncontrol.Task+.sendGrant(Resource))
13     && args(resource) && target(receiver)
14   {
15     // activate event on each monitor (not efficient)
16     for (Monitor monitor : monitors)
17       monitor.sendGrant(resource, receiver);
18   }
19   ...
20   // Advice for other events
21 }
```

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.

```
1  public class Monitor extends MonitorUtils {
2    void sendRequest(Actor sender, String resource) {};
3    void sendRequest(Actor sender, Resource resource) {};
4    void sendCancel(Actor sender, Resource resource) {}
5    void sendGrant(Resource resource, Actor receiver) {}
6    void sendRescind(Resource resource, Actor receiver) {}
7    void sendDeny(Actor receiver) {}
8    void addConflict(Resource resource1, Resource resource2) {}
9    void addPriority(Resource resource1, Resource resource2) {}
10   void requestResource(Actor actor, String name,
11           Response response) {}
12   void cancelResource(Actor actor, Resource resource) {}
13   void end() {}
14 }
```

The class MonitorUtils

- Auxiliary methods used for writing monitors.

```
1  public class MonitorUtils {
2    protected void verify(boolean condition, String message) {
3      if (!condition) error(message);
4    }
5
6    protected void error(String message) {
7      System.out.println("*** error: " + message);
8    }
9
10   protected void fail(String message) {
11     try {
12       throw new RuntimeException(message);
13     } catch (Exception e) {
14       e.printStackTrace();
15     }
16     System.exit(0);
17   }
18 }
```

JavaDoc documentation of methods representing events

Class Monitor

```
java.lang.Object
  └ monitor.MonitorUtils
      └ monitor.Monitor
```

Direct Known Subclasses:
GrantCancel, OnlyRescindGranted, RespectConflicts, RespectPriorities, WellformedResponse

---

```
public class Monitor
extends MonitorUtils
```

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()

Method Summary	
void	addConflict(Resource resource1, Resource resource2) Represents the addition of a conflict between two resources.
void	addPriority(Resource resource1, Resource resource2) Represents the addition of a priority between two resources.
void	cancelResource(Actor actor, Resource resource) Represents a call of the method void missioncontrol.ResourceTable.cancel(Actor, Resource).
void	end() Signals the end of the program.
void	requestResource(Actor actor, java.lang.String name, Response response) Represents a call of the method: Response ResourceTable.reqestResource(Actor actor, String name).
void	sendCancel(Actor sender, Resource resource) Represents a task canceling a resource.
void	sendDeny(Actor receiver) Represents the arbiter telling a task that the resource it last asked for has been denied.
void	sendGrant(Resource resource, Actor receiver) Represents a grant of a resource to a task.
void	sendRequest(Actor sender, Resource resource) Represents a task requesting a resource.
void	sendRequest(Actor sender, java.lang.String resource) Represents a task requesting a resource.
void	sendRescind(Resource resource, Actor receiver) Represents a rescind demand sent from arbiter to a task.

JavaDoc details of methods representing events

---

sendRequest

```
public void sendRequest(Actor sender,
                        Resource resource)
```

Represents a task requesting a resource. Represents the same event as `sendRequest(Actor, String)` but with the name replaced with the actual resource.

Parameters:
`sender` - the task requesting the resource.
`resource` - the resource requested.

---

sendCancel

```
public void sendCancel(Actor sender,
                       Resource resource)
```

Represents a task canceling a resource.

Parameters:
`sender` - the task canceling the resource.
`resource` - the resource being canceled.

---

sendGrant

```
public void sendGrant(Resource resource,
                      Actor receiver)
```

Represents a grant of a resource to a task.

Parameters:
`resource` - the resource granted.
`receiver` - the task that is granted the resource.

---

sendRescind

```
public void sendRescind(Resource resource,
                        Actor receiver)
```

Represents a rescind demand sent from arbiter to a task.

Let us specify some properties over these events

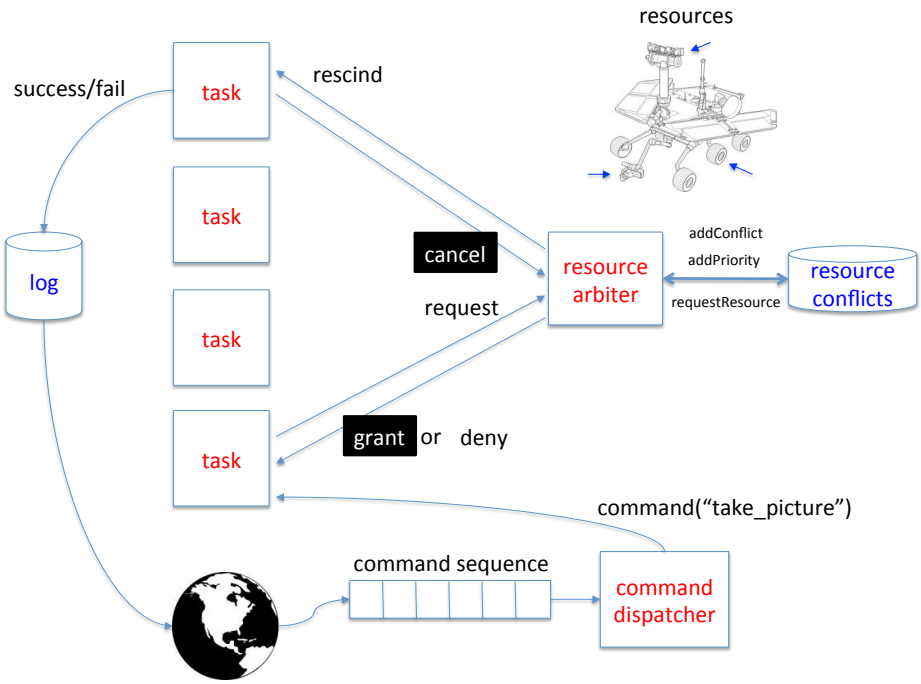
- GrantCancel: For a given resource, grants and cancellations should alternate, starting with a grant. Furthermore: a cancellation should be performed by the same task that was last granted the resource.
- OnlyRescindGranted: Only ask a task to rescind the resource if it is currently owned by the task. 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 $allocated(r) = undefined$.
    - $allocated(r) := a$.
  - When a cancels r:
    - Verify that $allocated(r) = a$.
    - $allocated(r) := undefined$

The GrantCancel Monitor

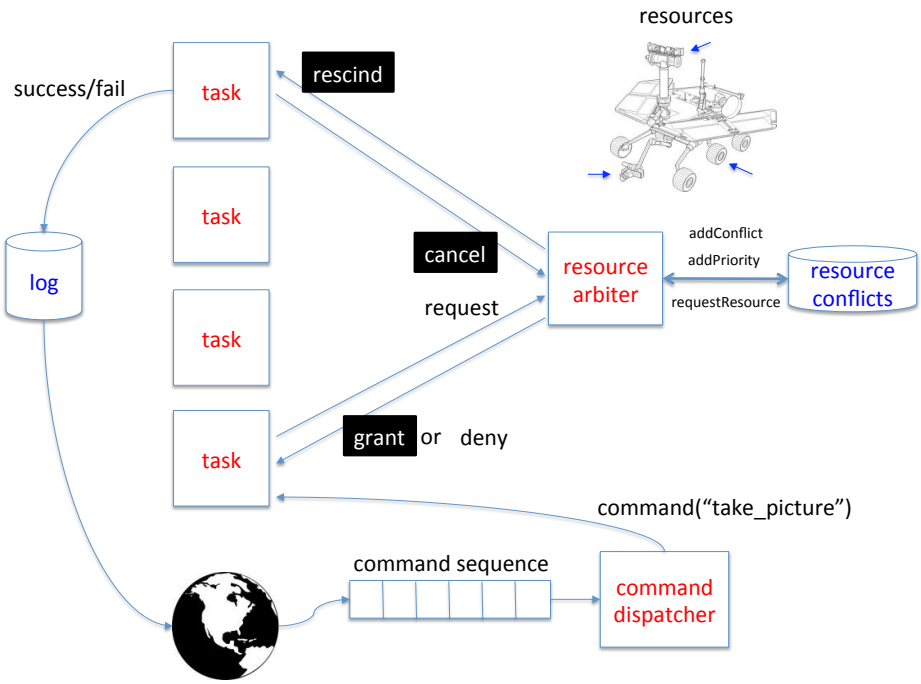
```
1  public class GrantCancel extends Monitor {
2    HashMap<Resource, Actor> allocated =
3      new HashMap<Resource, Actor>();
4
5    @Override
6    void sendGrant(Resource resource, Actor receiver) {
7      verify(!allocated.containsKey(resource));
8      allocated.put(resource, receiver);
9    }
10
11   @Override
12   void sendCancel(Actor sender, Resource resource) {
13     verify(allocated.get(resource) == sender);
14     allocated.remove(resource);
15   }
16 }
```

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:
    - $allocated(r) := a$.
  - When a cancels r:
    - $allocated(r) := undefined$.
  - When a rescind message is sent to an actor a to cancel r:
    - Verify that $allocated(r) = a$.

The OnlyRescindGranted Monitor

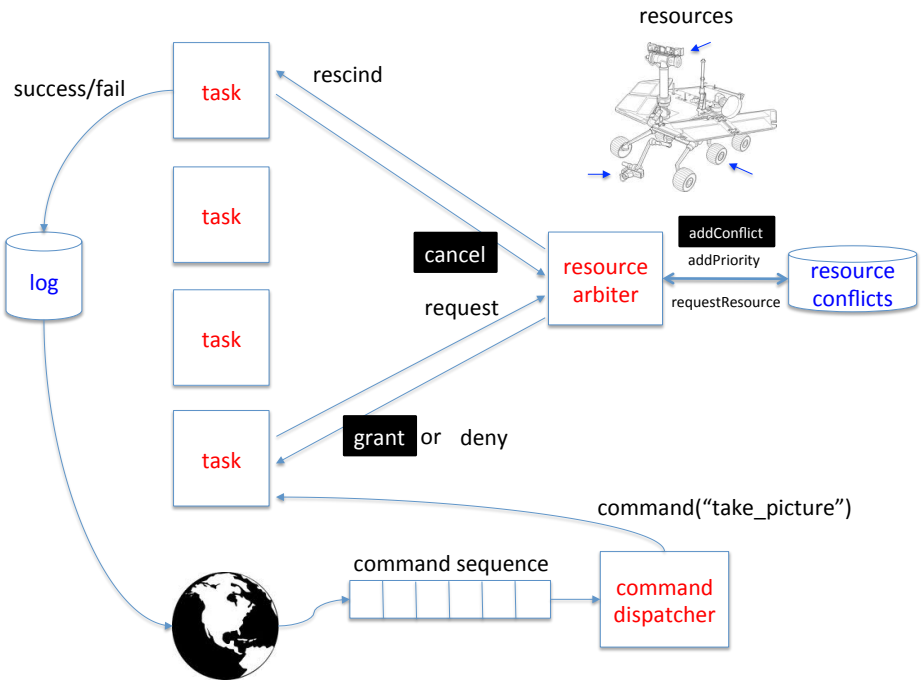
```
1  public class OnlyRescindGranted extends Monitor {
2    HashMap<Resource, Actor> allocated =
3      new HashMap<Resource, Actor>();
4
5    @Override
6    void sendGrant(Resource resource, Actor receiver) {
7      allocated.put(resource, receiver);
8    }
9
10   @Override
11   void cancelResource(Actor actor, Resource resource) {
12     allocated.remove(resource);
13   }
14
15   @Override
16   void sendRescind(Resource resource, Actor receiver) {
17     verify(allocated.get(resource) == receiver);
18   }
19 }
```

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

```
1  public class RespectConflicts extends Monitor {
2    MapToSet<Resource, Resource> conflicts =
3      new MapToSet<Resource, Resource>();
4    Map<Resource, Actor> allocated =
5      new HashMap<Resource, Actor>();
6
7    @Override
8    void addConflict(Resource resource1, Resource resource2) {
9      conflicts.getSet(resource1).add(resource2);
10     conflicts.getSet(resource2).add(resource1);
11   }
12   ...
13 }
```

The RespectConflicts Monitor 2

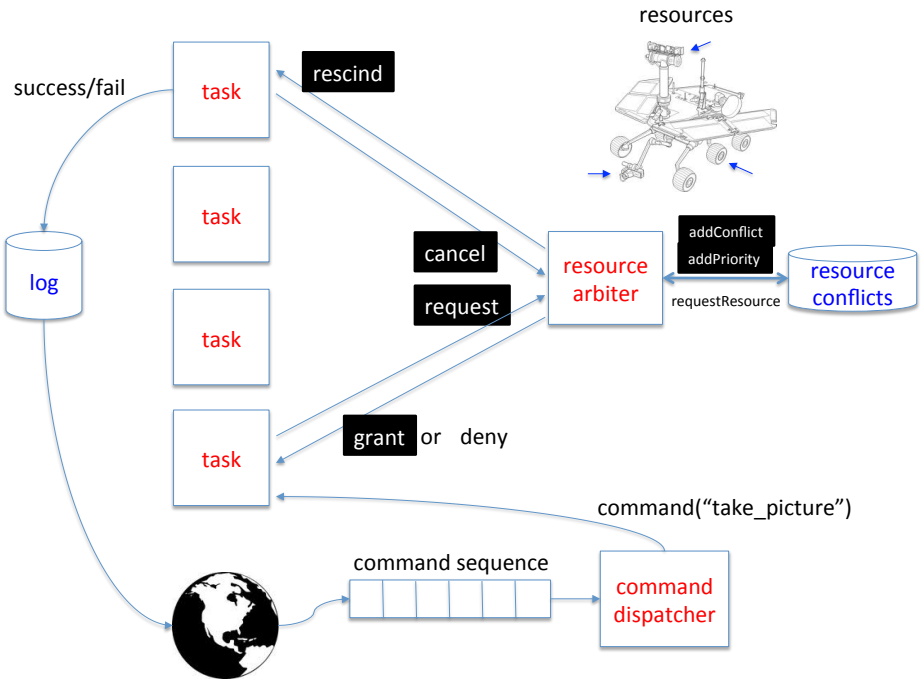
```
1  public class RespectConflicts extends Monitor {
2    MapToSet<Resource, Resource> conflicts = ...
3    Map<Resource, Actor> allocated = ...
4
5    ...
6
7    @Override
8    void sendGrant(Resource resource, Actor receiver) {
9      Set<Resource> intersection =
10       intersect(conflicts.getSet(resource),allocated.keySet());
11     verify(intersection.isEmpty());
12     allocated.put(resource, receiver);
13   }
14
15   @Override
16   void sendCancel(Actor actor, Resource resource) {
17     verify(allocated.get(resource) == actor);
18     allocated.remove(resource);
19   }
20 }
```

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:
  - *conflicts*: map from resource to set of resources it is in conflict with.
  - *priorities*: map from resource to set of resources with lower priority.
  - *allocated*: map from resource to actor if allocated by the actor.
  - *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 $\mathit{conflicts}(r_1)$ and vice versa.
  - When r_1 is declared as having higher priority than r_2:
    - ★ Add r_2 to the set $\mathit{priorities}(r_1)$.
  - When r is granted to an actor a:
    - ★ Verify that $\mathit{toRescind}(r)$ is empty.
    - ★ $\mathit{allocated}(r) := a$.
  - When a cancels r:
    - ★ $\mathit{allocated}(r) := \mathit{undefined}$.
  - When a requests r:
    - ★ If winning conflict, add loosing conflicts to *toRescind*.
  - When an actor a is asked to rescind r:
    - ★ Verify that the pair (a, r) is mapped to by some resource in *toRescind*.
    - ★ Remove the pair (a, r) from *toRescind*.

The RespectPriorities Monitor 1

```
1  public class RespectPriorities extends Monitor {
2    MapToSet<Resource, Resource> conflicts = ...;
3    MapToSet<Resource, Resource> priorities = ...;
4    HashMap<Resource, Actor> allocated = ...;
5    MapToSet<Resource, Pair<Actor, Resource>> toRescind = ...;
6
7    @Override
8    void addConflict(Resource resource1, Resource resource2) {
9      conflicts.getSet(resource1).add(resource2);
10     conflicts.getSet(resource2).add(resource1);
11   }
12
13   @Override
14   void addPriority(Resource resource1, Resource resource2) {
15     priorities.getSet(resource1).add(resource2);
16   }
17   ...
18 }
```

The RespectPriorities Monitor 2

```
1  public class RespectPriorities extends Monitor {
2    MapToSet<Resource, Resource> conflicts = ...;
3    MapToSet<Resource, Resource> priorities = ...;
4    HashMap<Resource, Actor> allocated = ...;
5    MapToSet<Resource, Pair<Actor, Resource>> toRescind = ...;
6    ...
7    @Override
8    void sendGrant(Resource resource, Actor actor) {
9      Set<Pair<Actor, Resource>> rescinds =
10       toRescind.getSet(resource);
11     verify(rescinds.isEmpty());
12     allocated.put(resource, actor);
13   }
14
15   @Override
16   void cancelResource(Actor actor, Resource resource) {
17     allocated.remove(actor);
18   }
19   ...
20 }
```

The RespectPriorities Monitor 3

```
1  @Override void sendRequest(Actor sender, Resource resource) {
2    Set<Resource> conflicting =
3      intersect(conflicts.getSet(resource), allocated.keySet());
4    if (!conflicting.isEmpty()) {
5      boolean winningConflict = false;
6      for (Resource conflict : conflicting) {
7        if (priorities.get(resource).contains(conflict))
8          winningConflict = true;
9          if (priorities.get(conflict).contains(resource)) {
10           winningConflict = false; break;
11         }
12       } // a conflict is winning if none have higher priority and this has higher priority than at least one
13       if (winningConflict) {
14         Set<Pair<Actor, Resource>> rescinds =
15           new HashSet<Pair<Actor, Resource>>();
16         for (Resource conflict : conflicting)
17           rescinds.add(new Pair<Actor, Resource>
18             (allocated.get(conflict), conflict));
19         toRescind.put(resource, rescinds);
20       }
21   }
22 }
```

The RespectPriorities Monitor 4

```
1  public class RespectPriorities extends Monitor {
2    MapToSet<Resource, Resource> conflicts = ...;
3    MapToSet<Resource, Resource> priorities = ...;
4    HashMap<Resource, Actor> allocated = ...;
5    MapToSet<Resource, Pair<Actor, Resource>> toRescind = ...;
6    ...
7    @Override
8    void sendRescind(Resource resource, Actor receiver) {
9      for (Resource waiting : toRescind.keySet()) {
10       Set<Pair<Actor, Resource>> rescinds =
11         toRescind.get(resource);
12       for (Pair<Actor, Resource> pair : rescinds) {
13         if (pair.second == resource) {
14           verify(pair.first == receiver);
15           rescinds.remove(pair);
16           break;
17         }
18       }
19     }
20   }
21 }
```

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


Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm and William G. Griswold : *An Overview of AspectJ*. In ECOOP'01, LNCS volume 2072.


Xerox Corporation, Palo Alto Research Center : *The AspectJTM Programming Guide*. http://www.eclipse.org/aspectj/doc/released/progguide.

Part III
TraceMatches

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`.

Contents

In this section we will:

- Capture the principles behind Parametric Runtime Verification.
- Introduce the TraceMatches tool including:
  - The syntax of TraceMatches
  - How to use the tool
  - An illustrative example
  - The semantics of TraceMatches
  - An overview of implementation and efficiency

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 Σ, the next state is computed using a transition function $\delta \in (\Sigma \times S) \to S$.
- In Parametric Runtime Verification we consider events carrying data values drawn from a set of objects $\mathcal{O}$.
- 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

$$\delta \in (\Sigma \times 2^{\mathcal{O}} \times S) \to S$$

- 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 `priority`(wheels, camera) and `priority`(camera, 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 $Bind = Var \rightharpoonup \mathcal{O}$ be the set of all bindings (partial maps), given some set of variables Var.
- Let $\texttt{name} : (\Sigma \times \mathcal{O}^*) \rightarrow Bind$ be a function that creates bindings from a parametric event, how this is implemented is not important here.
- For example, a possible implementation of `name` might give

$$\begin{array}{lcl} \texttt{name(priority(wheels,camera))} & = & [r1 \mapsto \text{wheels}, r2 \mapsto \text{camera}] \\ \texttt{name(priority(camera,wheels))} & = & [r1 \mapsto \text{camera}, r2 \mapsto \text{wheels}] \end{array}$$

- It is now possible to differentiate between events `priority`(wheels, camera) and `priority`(camera, wheels).
- The parametric transition function should therefore be:

$$\delta \in (\Sigma \times Bind \times S) \rightarrow S$$

Different approaches

- For efficiency reasons, it is desirable to organise the computation of δ, and there are three different approaches to this:
- Object-based

$$\delta \in Bind \rightarrow (S \times \Sigma) \rightarrow S$$

- State-based

$$\delta \in S \rightarrow (\Sigma \times Bind) \rightarrow S$$

- Event-based

$$\delta \in \Sigma \rightarrow (S \times Bind) \rightarrow S$$

- 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

⟨TRACEMATCH⟩ ::=
  [**perthread**] **tracematch** (⟨VARIABLE DECLARATIONS⟩)
  {
    ⟨TOKEN DECLARATION⟩+
    ⟨REGEX⟩
    ⟨METHOD BODY⟩
  }

⟨TOKEN DECLARATION⟩ ::=
  sym ⟨NAME⟩ ⟨KIND⟩: ⟨POINTCUT⟩;

⟨KIND⟩ ::=
  before
| **after**
| **after returning** [(⟨VARIABLE⟩)]
| **after throwing** [(⟨VARIABLE⟩)]
| ⟨TYPE⟩ **around** [(⟨VARIABLES⟩)]

⟨REGEX⟩ ::=
  ⟨NAME⟩
| ⟨REGEX⟩ ⟨REGEX⟩ — *AB — A followed by B*
| ⟨REGEX⟩ | ⟨REGEX⟩ — *A|B — A or B*
| ⟨REGEX⟩ * — *A* — 0 or more As*
| ⟨REGEX⟩ + — *A+ — 1 or more As*
| ⟨REGEX⟩ [⟨CONSTANT⟩] — *A[n] — exactly n As*
| (⟨REGEX⟩) — *(A) — grouping*

- 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/`.

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  -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.

```
1  public class Util {
2   protected static void fail(String message) {
3     try {
4       throw new RuntimeException(message);
5     } catch (Exception e) {
6       e.printStackTrace();
7     }
8     System.exit(0);
9   }
10 }
```

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*`)`

```
1  pointcut grant(Actor requester, Resource resource) :
2   call(void missioncontrol.Task+.sendGrant(Resource))
3   && args(resource) && target(requester);
4
5  pointcut cancel(Actor owner,Resource resource) :
6   call(void missioncontrol.Arbiter.sendCancel(Task,Resource))
7   && args(owner,resource);
```

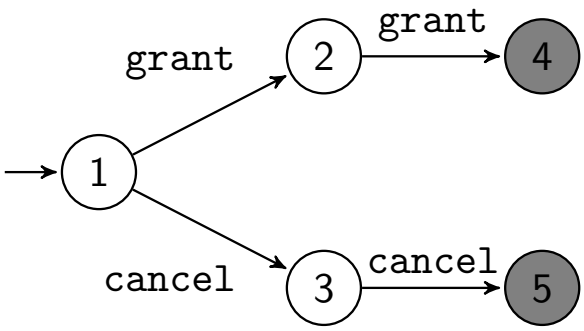
Grant and cancel should alternate

- 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*.

```
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␣grant␣and␣cancel␣on␣resource␣"+r
9                                    +"␣do␣not␣alternate");
10   }
11 }
```

Detecting a match

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



Detecting a match on a propositional trace

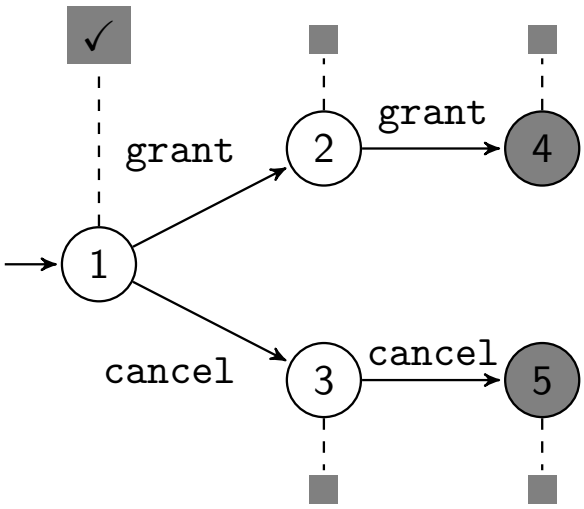
- Let us consider the trace:

`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.

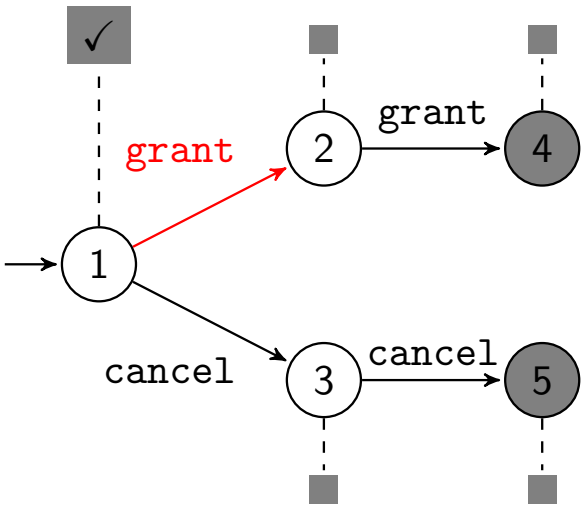
Detecting a match on a propositional trace

ϵ



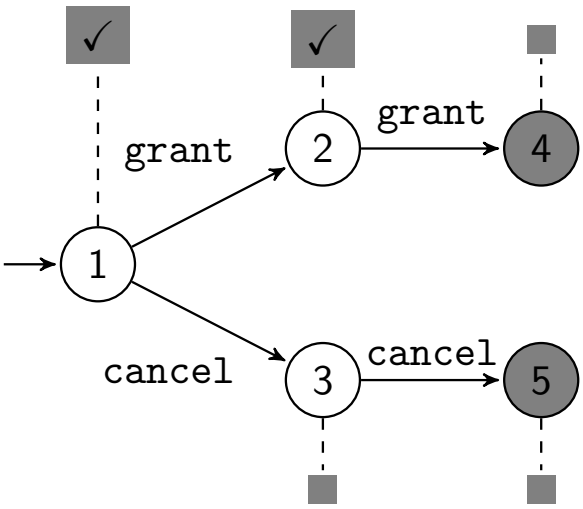
Detecting a match on a propositional trace

ϵ.grant



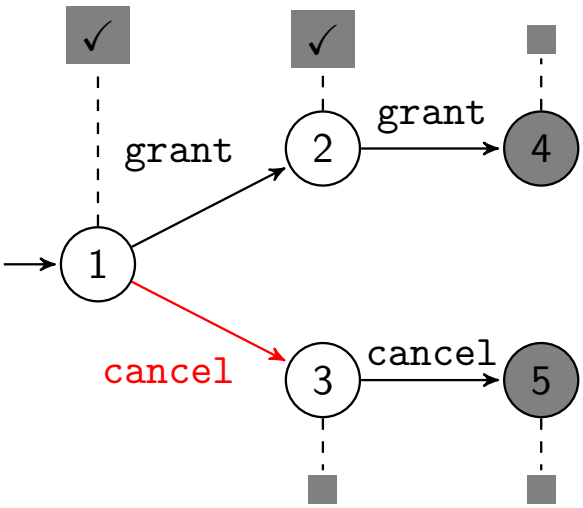
Detecting a match on a propositional trace

ϵ.grant



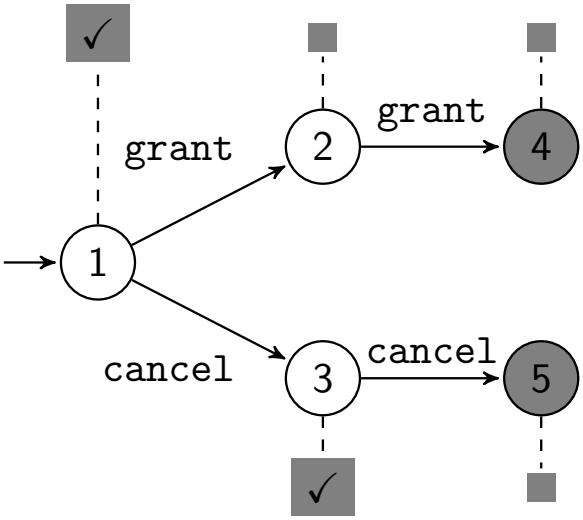
Detecting a match on a propositional trace

ϵ.grant.cancel



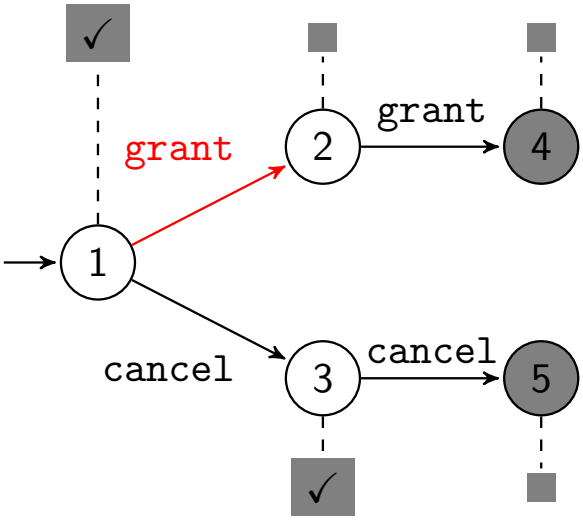
Detecting a match on a propositional trace

ϵ.grant.cancel



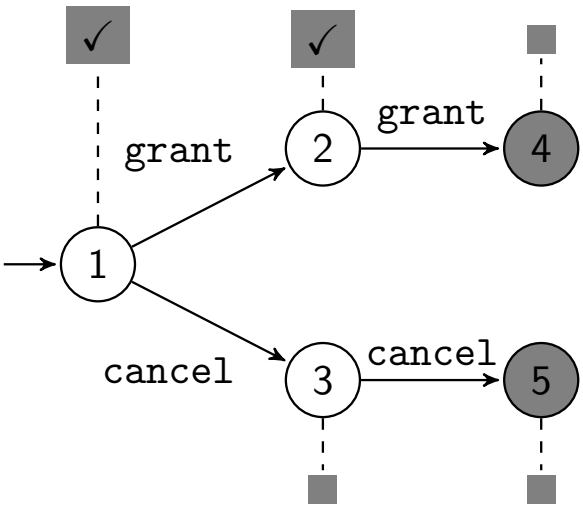
Detecting a match on a propositional trace

ϵ.grant.cancel.grant



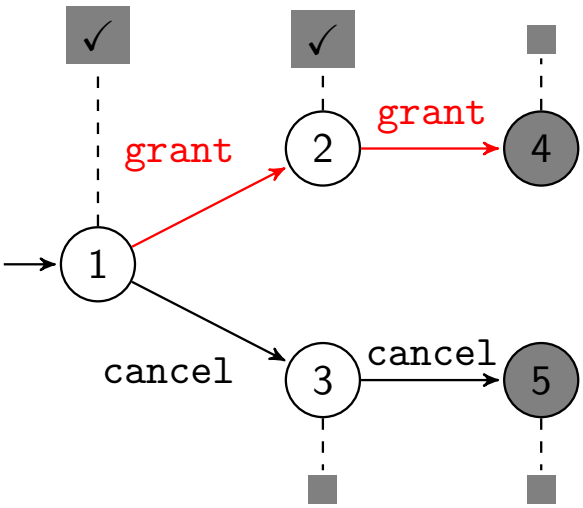
Detecting a match on a propositional trace

ϵ.grant.cancel.grant



Detecting a match on a propositional trace

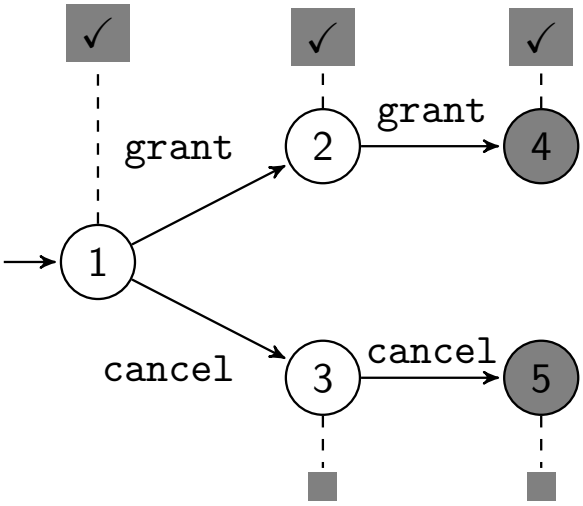
ϵ.grant.cancel.grant.grant



Detecting a match on a propositional trace

- We have a match

ϵ.grant.cancel.grant.grant

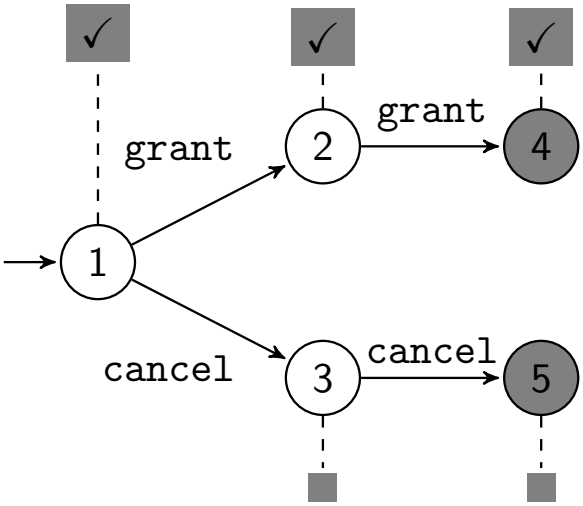


Detecting a match on a propositional trace

- We have a match

ϵ.grant.cancel.grant.grant

- Because this suffix matched the regular expression



Detecting a match on a parametric trace

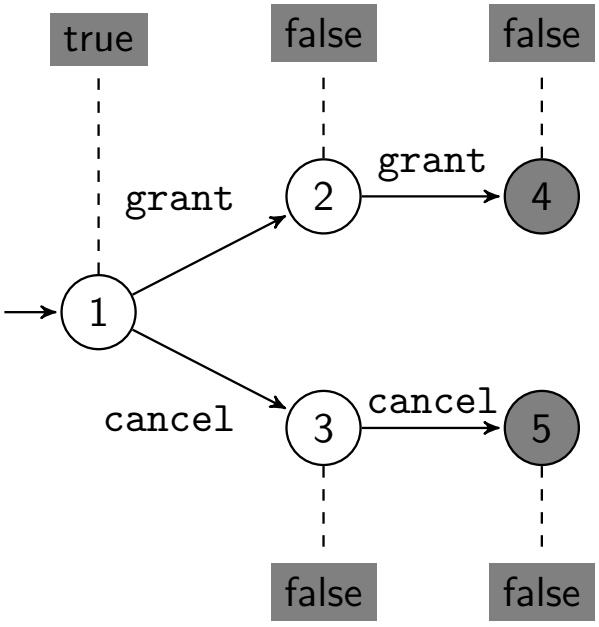
- Works fine when there is no data - but data is useful!
- Consider the observations:

$$\texttt{grant}(\text{driving\_task}, \text{wheels})$$
$$\texttt{grant}(\text{driving\_task}, \text{antenna})$$
$$\texttt{cancel}(\text{driving\_task}, \text{wheels})$$
$$\texttt{grant}(\text{camera\_task}, \text{antenna})$$

- 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

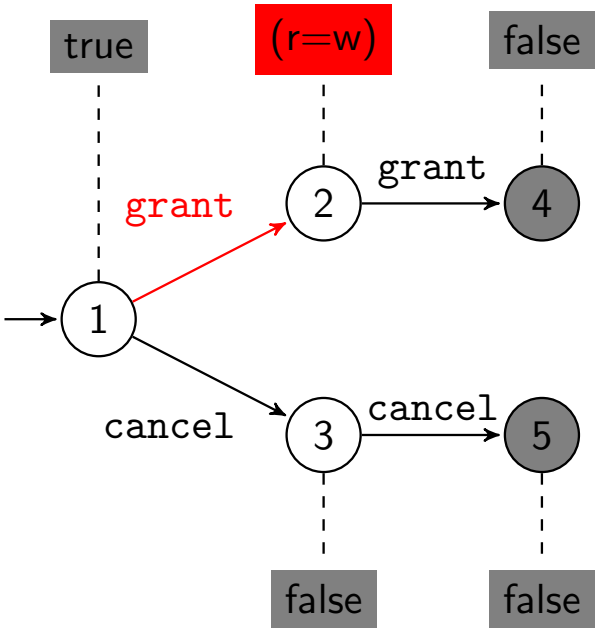
- Label the initial state as true and all other states as false.



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.

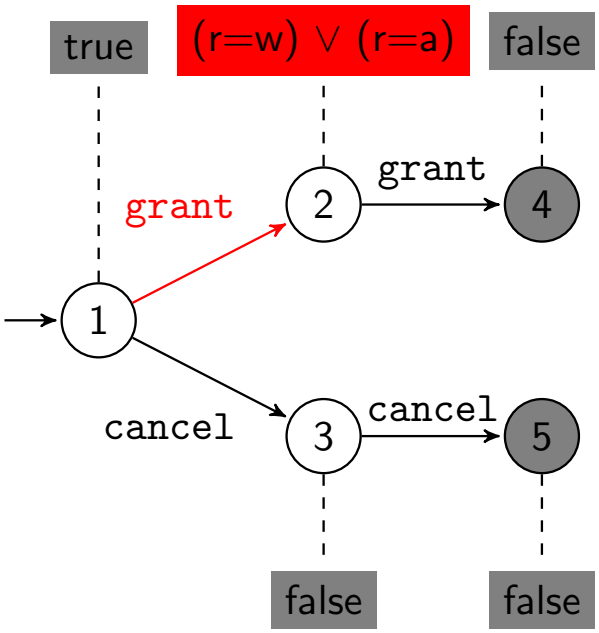
grant(driving_task,wheels)



Detecting a match on a parametric trace

- Add constraint (r=a) to state 2 (in disjunction).

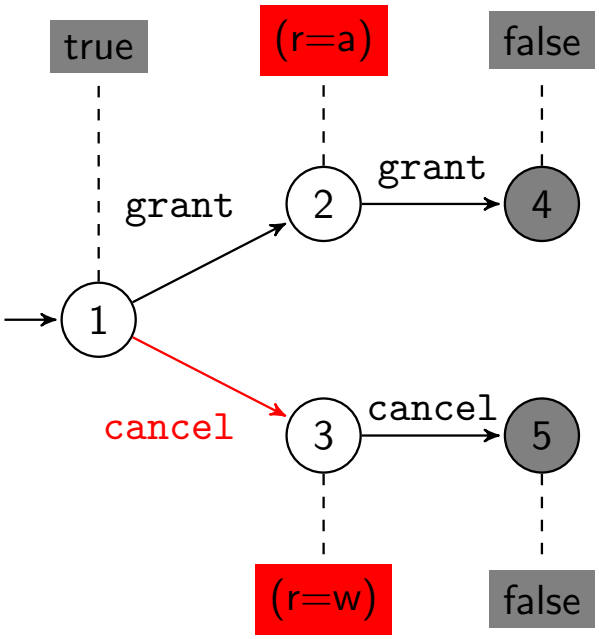
grant(driving_task,wheels)
grant(driving_task,antenna)



Detecting a match on a parametric trace

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

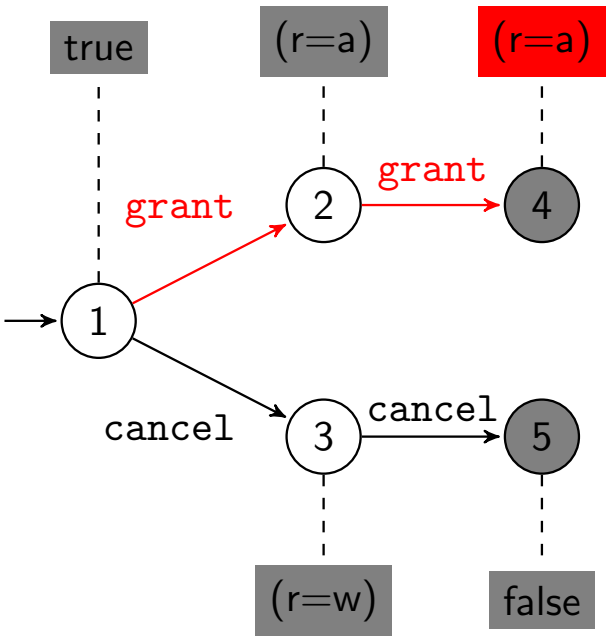
grant(driving_task,wheels)
grant(driving_task,antenna)
cancel(driving_task,wheels)



Detecting a match on a parametric trace

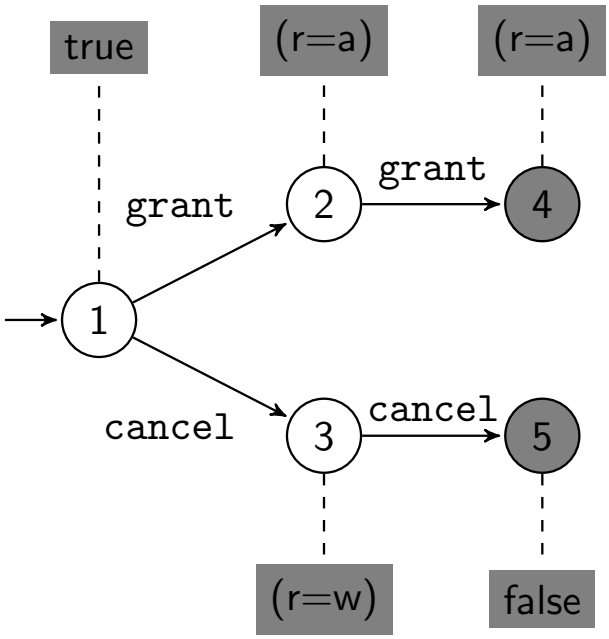
- Add constraint (r=a) to state 4.

grant(driving_task,wheels)
grant(driving_task,antenna)
cancel(driving_task,wheels)
grant(camera_task,antenna)



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 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 $\mathit{Event} \to \mathit{Constraint}$.
- The constraint captures a binding for the variables in the symbol.
- For example:

$$\begin{aligned} \underbrace{\texttt{grant}}_{\text{symbol}}(\underbrace{\texttt{grant}(\text{driving\_task,wheels})}_{\text{event}}) &= \underbrace{(r = \mathit{wheels})}_{\text{constraint}} \\ \underbrace{\texttt{grant}}_{\text{symbol}}(\underbrace{\texttt{cancel}(\text{driving\_task,wheels})}_{\text{event}}) &= \underbrace{\mathit{false}}_{\text{constraint}} \end{aligned}$$

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

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

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.,

$$\begin{aligned} [r \mapsto w]\,((\text{r=w}) \vee (\text{r=a})) &= (\text{w=w}) \vee (\text{w=a}) \\ &= \mathit{true} \vee \mathit{false} \\ &= \mathit{true} \end{aligned}$$

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

$$\theta(a) = \lambda e.\theta(a(e)) \in \mathit{Event} \to \mathbb{B}$$

here we assume that bindings bind all free variables

- Let $P(\theta)$ be the regular expression constructed by applying θ 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 τ.
- We start by defining which events are relevant to a binding θ:

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

- We can filter irrelevant events out of a trace:

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

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

$$\mathit{satisfy}(\tau, \theta) = \bigvee_{\sigma \in \mathcal{L}(P(\theta))} \mathit{match}(\sigma, \tau \downarrow_\theta) \wedge \mathit{last}(\tau) \in \mathit{relevant}(\theta)$$

- The bindings to execute the code for given trace τ are:

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

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

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

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

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

Defining **skip**

- The **skip** symbol has two functions:
  - Match any event not in A
  - Ensure that we do not skip relevant events
- So what constraint should **skip** produce?
- An event is relevant (where C is the current constraint) iff

$$\exists a \in A : (a(e) \wedge C) \neq \mathit{false}$$

- Therefore, let **skip** symbol be defined as:

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

- For example:

$$\begin{aligned} \mathbf{skip}(\texttt{grant(d\_task,w)}) &= \neg \texttt{grant}(\texttt{grant(d\_task,w)}) \wedge \\ &\qquad \neg \texttt{cancel}(\texttt{grant(d\_task,w)}) \\ &= \neg(r = w) \wedge \neg \mathit{false} \\ &= (r \neq w) \end{aligned}$$

Defining **skip**

- The **skip** symbol has two functions:
  - Match any event not in A
  - Ensure that we do not skip relevant events
- So what constraint should **skip** produce?
- An event is relevant (where C is the current constraint) iff

$$\exists a \in A : (a(e) \wedge C) \neq \mathit{false}$$

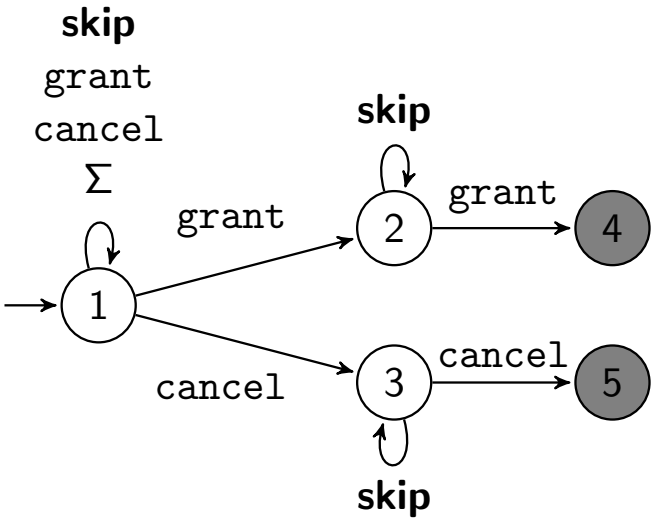
- Therefore, let **skip** symbol be defined as:

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

- This satisfies the two functions:
  - If e is not in A then this will be *true*
  - If e is relevant to C then this will contradict C - therefore not allowing us to match with **skip**

From semantics to implementation

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



From semantics to implementation

- Construct an automaton for $Pat = \Sigma^*(P \parallel \mathbf{skip}^*) \cap (\Sigma^* A)$
- We missed out some transitions earlier.
- Associate a label (of constraints) with each state.
- Update this label for state i as follows:

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

- 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 repsect 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.

Starts with a grant

For a given resource, grants and cancellations should start with a grant.

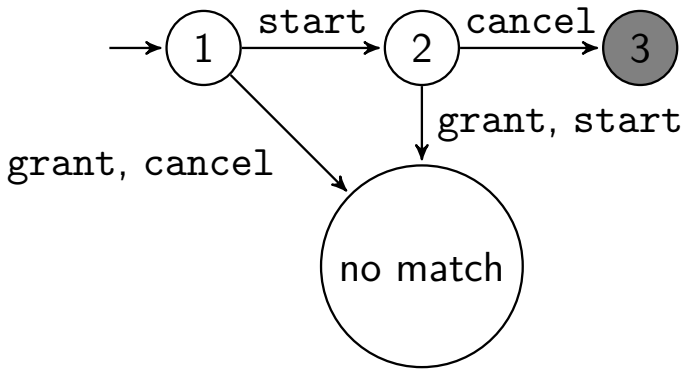
- We need to detect the start of the trace.
- It is important that the `grant` symbol is present.

```
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 "+r+" was cancelled before"
11                                  +" it was granted");
12   }
```

Starts with a grant

For a given resource, grants and cancellations should [start] with a grant.

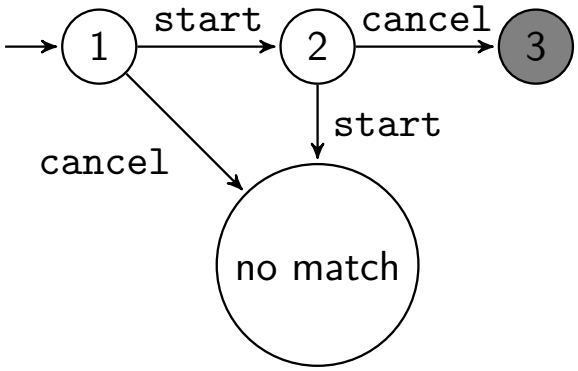
- We need to detect the start of the trace.
- It is important that the `grant` symbol is present.
- With it in the alphabet we don't match on `start.grant.cancel`:



Starts with a grant

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.
- With it in the alphabet we do not match on `start.grant.cancel`.
- Otherwise, `grant` would be filtered out of the trace and it would match:



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.

```
1  tracematch(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 "+r+" cancelled by "
11                 +other+" when held by "+owner);
12     }
13 }
```

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*)

```
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.Arbiter.sendCancel(Task,
11   Resource)) && args(owner,resource);
```

RespectConflicts : defining the property

- We define our tracematch using these events.
- A trace matches if there is a conflict between resource r1 and resource r2 and they are (at some point) granted at the same time - note that `cancel_r2` is not defined here so the ordering matters.

```
1  tracematch(Resource r1, Resource r2)
2  {
3    sym conflict after: (conflict(r1,r2)|| conflict(r2,r1));
4    sym grant_r1 after: grant(r1);
5    sym cancel_r1 after: cancel(r1);
6    sym grant_r2 after: grant(r2);
7
8    conflict+ (grant_r1 | cancel_r1 | grant_r2)*
9                                      grant_r1 grant_r2
10   {
11     if(r1!=r2)
12       Util.fail("Conflicting resources "+r1 + " and "+r2+
13                 " granted at the same time");
14   }
15 }
```

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.

References


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


Making Trace Monitors Feasible: P. Avguistinov, J. Tibble and O. doe Moor. In OOPSLA 2007

Part IV
JavaMOP

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

Contents

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

⟨*Specification*⟩ ::= {⟨*Instance Modifier*⟩} ⟨*Id*⟩ ⟨*Instance Parameters*⟩ "{"
{⟨*Instance Declaration*⟩}
{⟨*Event*⟩}
{⟨*Property*⟩
{⟨*Property Handler*⟩}
}
"}"
⟨*Event*⟩ ::= ["*creation*"]"*event*" ⟨*Id*⟩ ⟨*Instance Event Definition*⟩ "{" ⟨*Instance Action*⟩ "}"
⟨*Property*⟩ ::= ⟨*Logic Name*⟩ " : " ⟨*Logic Syntax*⟩
⟨*Property Handler*⟩ ::= "@" ⟨*Logic State*⟩ ⟨*Instance Handler*⟩

- 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

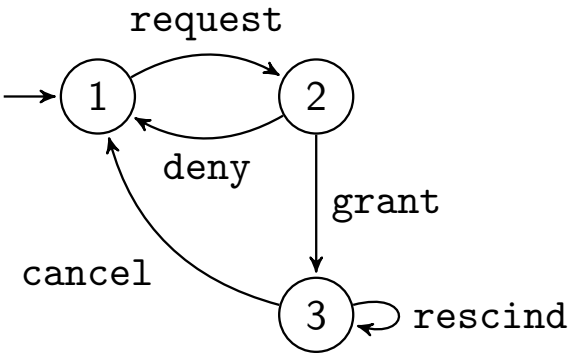
- The MOP website is `http://fsl.cs.uiuc.edu/index.php/MOP`.
- Here you can
  - See all publications related to JavaMOP
  - Download the tool.
  - View examples
  - Interact with the tool

Running JavaMOP

- Download JavaMOP installer from `http://fsl.cs.uiuc.edu/index.php/MOP`.
- 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.



- 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.

```
1  ResourceLifeCycle(Resource r) {
```

Defining the events

- Events are defined in terms of `AspectJ` pointcuts

```
2    event request after(Resource r) :
3      call(void *.Task+.sendRequest(Resource))&& args(r){}
4
5    event grant after(Resource r) :
6      call(void *.Task+.sendGrant(Resource))&& args(r){}
7
8    event rescind(Resource r) :
9      call(void *.Task+.sendRescind(Resource)) && args(r){}
10
11   event cancel after(Resource r) :
12     call(void *.Arbiter.sendCancel(Task,Resource))
13     && args(*,r){}
14
15   event deny after(Resource r) :
16     call(void *.Arbiter.sendDeny(Task,Resource))
17     && args(*,r){}
```

Describing the property

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

```
18      fsm :
19        start [
20           request -> requested
21        ]
22        requested [
23           deny    -> start
24           grant   -> owned
25        ]
26        owned [
27           rescind -> owned
28           cancel  -> start
29        ]
```

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.

```
31        @owned{
32             log.print("Resource "+r+" granted");
33        }
```

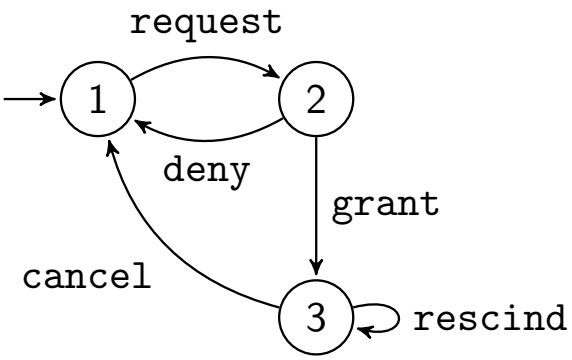
```
34        @fail{
35             Util.fail("Resource "+r+
36                "was used incorrectly");
37        }
```

Matching

- Let us consider how to match against the trace:

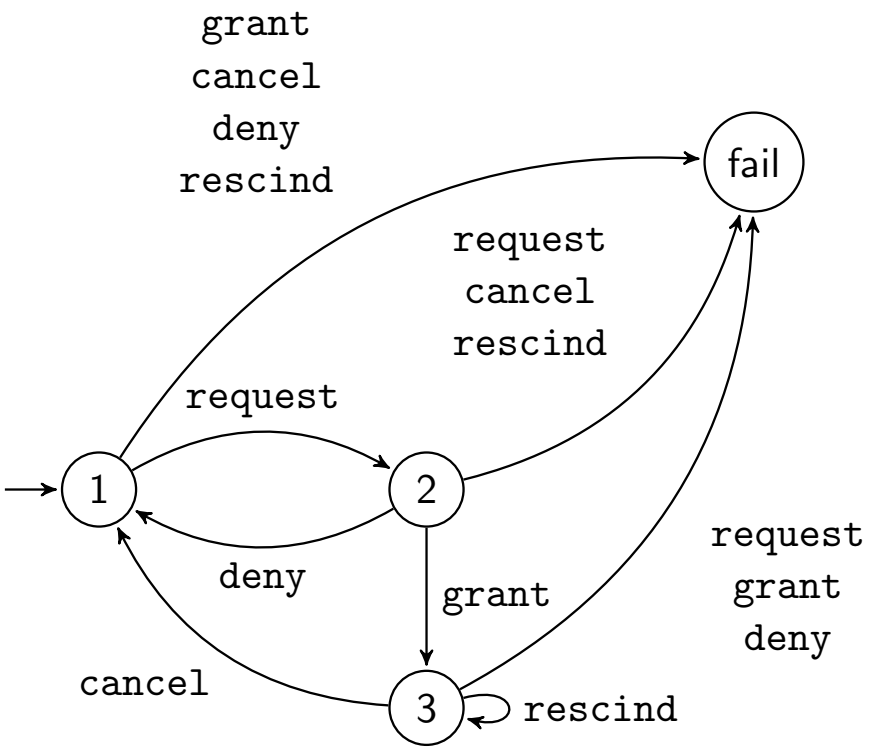
```
request(wheels)
request(antenna)
grant(antenna)
deny(wheels)
cancel(antenna)
request(wheels)
rescind(antenna)
```

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



Capturing full behaviour

- We can add in the implicit fail state:



Processing the trace

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

$$Bind \rightharpoonup State$$

- We start with an empty binding and initial state.

r	state
-	1

Processing the trace

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

$$Bind \rightharpoonup State$$

- We start with an empty binding and initial state.

`request`(wheels)

r	state	
-	1	
wheels	2	(requested)

Processing the trace

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

$$Bind \rightharpoonup State$$

- We start with an empty binding and initial state.

```
request(wheels)
request(antenna)
```

r	state	
-	1	
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

$$Bind \rightharpoonup State$$

- We start with an empty binding and initial state.

```
request(wheels)
request(antenna)
grant(antenna)
```

r	state	
-	1	
wheels	2	(requested)
antenna	3	(owned)

Processing the trace

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

$$Bind \rightharpoonup State$$

- We start with an empty binding and initial state.

```
request(wheels)
request(antenna)
grant(antenna)
deny(wheels)
```

r	state	
-	1	
wheels	1	
antenna	3	(owned)

Processing the trace

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

$$Bind \rightharpoonup State$$

- We start with an empty binding and initial state.

```
request(wheels)
request(antenna)
grant(antenna)
deny(wheels)
cancel(antenna)
```

r	state
-	1
wheels	1
antenna	1

Processing the trace

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

$$Bind \rightharpoonup State$$

- We start with an empty binding and initial state.

```
request(wheels)
request(antenna)
grant(antenna)
deny(wheels)
cancel(antenna)
request(wheels)
```

r	state	
-	1	
wheels	2	(requested)
antenna	1	

Processing the trace

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

$$Bind \rightharpoonup State$$

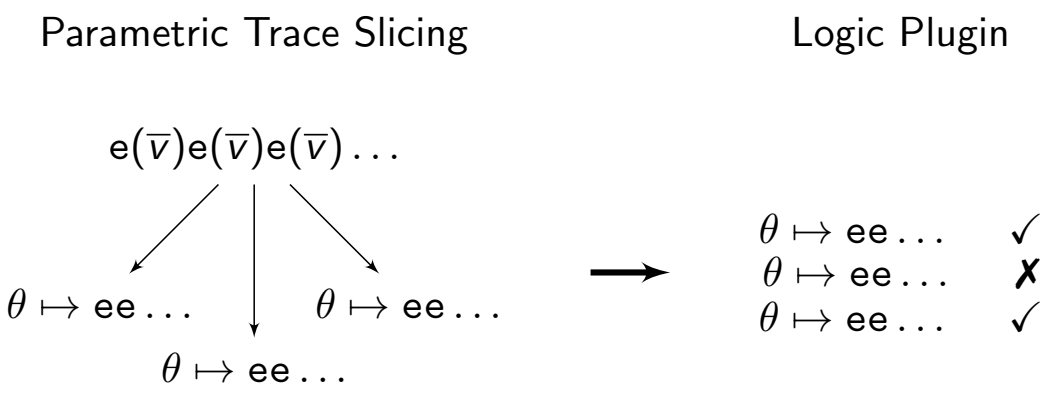
- We start with an empty binding and initial state.

```
request(wheels)
request(antenna)
grant(antenna)
deny(wheels)
cancel(antenna)
request(wheels)
rescind(antenna)
```

r	state	
-	1	
wheels	2	(requested)
antenna	fail	

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 θ if it only includes things mentioned in θ - θ' is a submap of θ.
- Therefore, Parametric Trace Slicing is defined as

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

- Note the similarity with filtering in TraceMatches.

Slicing our trace

- Let us call this trace τ.

  `request`(wheels)
  `request`(antenna)
  `grant`(antenna)
  `deny`(wheels)
  `cancel`(antenna)
  `request`(wheels)
  `rescind`(antenna)

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

$\tau \downarrow_{[r \mapsto w]} =$ `request`(w).`deny`(w)`request`(w)
$\tau \downarrow_{[r \mapsto a]} =$ `request`(a).`grant`(a).`cancel`(a).`rescind`(a)

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_r1` and `grant_r2` 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

$$PropositionalTrace \rightarrow Verdict$$

- 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.

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

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.

```
1  GrantCancel(Resource r) {
2        ...
3        ltl : [](grant => () cancel)
4
5        @violation{
6            ...
7        }
8  }
```

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.

```
1  GrantCancel(Resource r) {
2        ...
3        pltl : cancel => (*) grant
4
5        @violation{
6            ...
7        }
8  }
```

A cancel must be preceded by a grant.

Context free grammars

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

```
1  GrantCancel(Resource r) {
2        ...
3        cfg : S -> W | G | C
4              W -> start cancel
5              G -> grant grant
6              C -> canel cancel
7
8        @match{
9            ...
10       }
11 }
```

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:

$$\texttt{pts} \in \mathit{ParametricTrace} \times \mathit{Bind} \to \mathit{PropositionalTrace}$$

- And a Logic Plugin as a function:

$$\texttt{plugin} \in \mathit{PropositionalTrace} \to \mathit{Verdict}$$

- So the verdict for a trace given a binding is:

$$\texttt{check} \in \mathit{ParametricTrace} \times \mathit{Bind} \to \mathit{Verdict} = \texttt{plugin} \circ \texttt{pts}$$

- We can model an action as a predicate that checks the verdict and a function that takes a binding and returns code:

$$\texttt{Action} = (\mathit{Verdict} \to \mathbb{B}) \times (\mathit{Bind} \to \mathit{Code})$$

- Therefore, the code to execute on observing parametric trace τ is:

$$\{act.snd(\theta) \mid act \in \texttt{Actions} \wedge \theta \in \mathit{Bind} \wedge act.fst(\texttt{check}(\tau, \theta))\}$$

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 θ can only cause an action to fire if there does not exist a binding θ' such that $\theta \sqsubset \theta'$ and $\mathtt{pts}(\theta', \tau) \neq \epsilon$.
  - 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.

keySet

`Set<K> keySet()`

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.remove`, `Set.remove`, `removeAll`, `retainAll`, and `clear` operations. It does not support the `add` or `addAll` operations.

Returns:

a set view of the keys contained in this map

values

`Collection<V> values()`

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.remove`, `Collection.remove`, `removeAll`, `retainAll` and `clear` operations. It does not support the `add` or `addAll` operations.

Returns:

a collection view of the values contained in this map

The JavaMOP specification

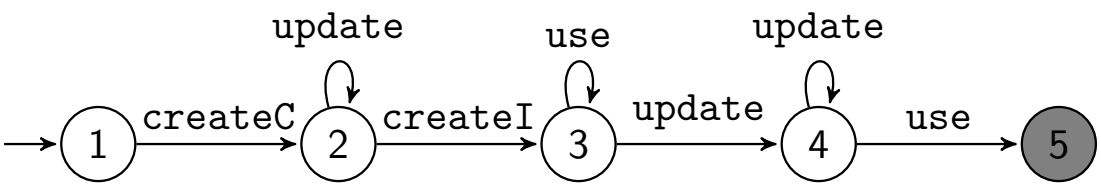
```
1  import java.util.*;
2
3  full-binding UnsafeMapIterator(Map m, Collection c, Iterator i){
4
5    event createC after(Map m) returning(Collection c) :
6      (call(* Map.values()) || call(* Map.keySet()))
7      && target(m) {}
8
9    event createI after(Collection c) returning(Iterator i) :
10     call(* Collection.iterator()) && target(c) {}
11
12   event use before(Iterator i) :
13     call(* Iterator.next()) && target(i) {}
14
15   event update after(Map m) :
16     (call(* Map.put*(..))    || call(* Map.putAll*(..))
17     || call(* Map.clear())   || call(* Map.remove*(..)))
18     && target(m) {}
19
20   ere : createC update* createI use* update update* use
21
22   @match{ System.out.println("unsafe iterator usage!");}
23 }
```

An automaton

- We can construct an automaton for the expression

  $$\texttt{createC update}^* \texttt{ createI use}^* \texttt{ update update}^* \texttt{ use}$$

- A match is detected if we reach state 5.



An example trace

- Let us consider the trace:

$$\begin{array}{l}\texttt{createC}(M_1,C_1)\\ \texttt{createC}(M_1,C_2)\\ \texttt{createI}(C_1,I_1)\\ \texttt{update}(C_1)\\ \texttt{createI}(C_2,I_2)\\ \texttt{use}(I_1)\end{array}$$

- 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

$$\texttt{createC createI update use}$$

matches the expression

A basic algorithm

Input: a parametric trace τ
Output: a map from bindings to propositional traces

$\Delta : [Bind \rightharpoonup PropositionalTrace]$;
$\Theta : Bind$;
$\Delta \leftarrow [\bot \rightarrow \epsilon]$;
foreach $e(\theta) \in \tau$ *in order* **do**
  $\Theta \leftarrow \underline{dom}(\Delta)$;
  foreach $\theta' \in \Theta$ **do**
    if θ *is consistent with* θ' **then**
      $\theta_{max} \leftarrow [\,]$;
      foreach $\theta_{alt} \in \Theta$ **do**
        if $\theta_{max} \sqsubseteq \theta_{alt} \sqsubseteq \theta \dagger \theta'$ **then** $\theta_{max} = \theta_{alt}$
      $\Delta(\theta \dagger \theta') \leftarrow \Delta(\theta_{max})e$
return Δ

What it does

```
1  Δ ← [⊥ → ϵ] ;
2  foreach e(θ) ∈ τ in order do
3      Θ ← dom(Δ);
4      foreach θ′ ∈ Θ do
5          if θ is consistent with θ′ then
6              θ_max ← θ′;
7              foreach θ_alt ∈ Θ do
8                  if θ_max ⊑ θ_alt ⊑ θ ⨾ θ′
                   then θ_max = θ_alt
9              Δ(θ ⨾ θ′) ← Δ(θ_max)e
10 return Δ
```

- Initialise Δ
- For each event in the trace
- Save the domain of Δ in Θ
- For each binding θ' in Θ
- If the event is relevant to θ'
- Find the existing binding θ_{max} that is the largest binding bigger than θ'
- Append the event name to the trace for θ_{max} and set this as the trace for $\theta \fatsemi \theta'$
- Return the resulting map Δ

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 δ be the initial state and transition function.
- Therefore, we compute the `check` function.

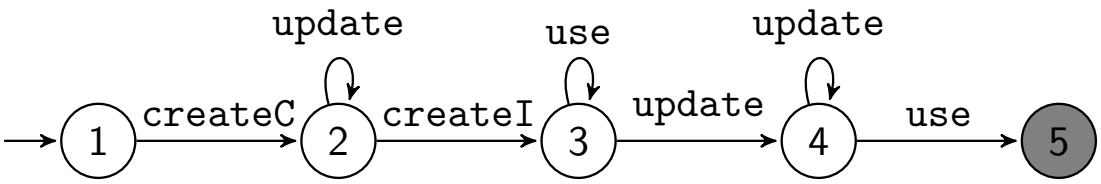
Input: a parametric trace τ
Output: a map from bindings to monitors

```
1  CHECK(τ){
2  Δ : [Bind ⇀ State]; Θ : Bind;
3  Δ ← [⊥ → q0] ;
4  foreach e(θ) ∈ τ in order do
5      Θ ← dom(Δ);
6      foreach θ′ ∈ Θ do
7          if θ is consistent with θ′ then
8              θ_max ← θ′;
9              foreach θ_alt ∈ Θ do
10                 if θ_max ⊑ θ_alt ⊑ θ ⨾ θ′ then
                   θ_max = θ_alt
11             Δ(θ ⨾ θ′) ← δ(Δ(θ_max), e)
12 return Δ return θ ∈ dom(Δ) where Δ(θ)
   is final
13 }
```

How it works

- Bindings can be represented in a lattice using the submap relation $\sqsubseteq$.
- 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).

How it works



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

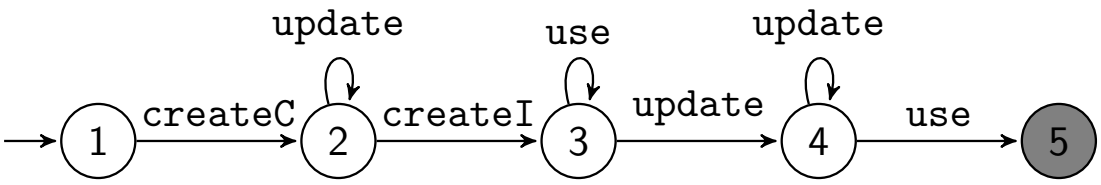
Trace

`createC`(M_1,C_1)

(M_1,C_1,-):2

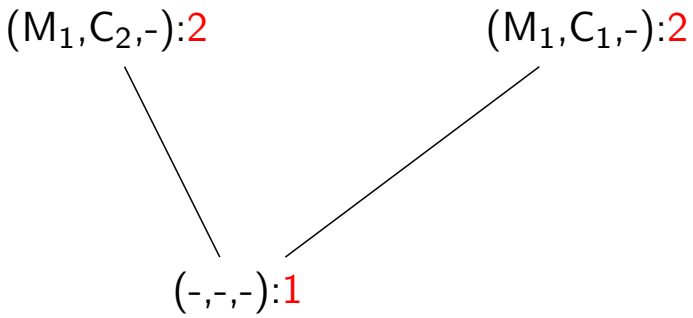
(-,-,-):1

How it works

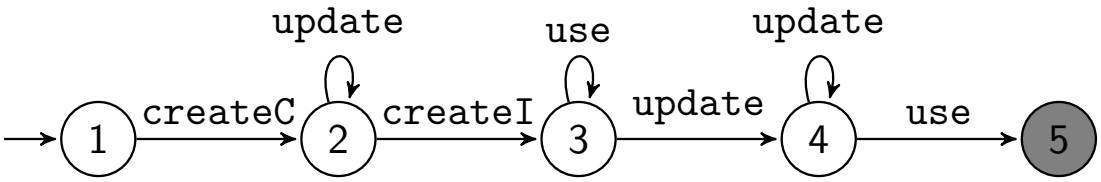


- We initialise new bindings with the state from the maximal binding (here (-,-,-)) and then apply the event

Trace
createC(M_1,C_1)
createC(M_1,C_2)

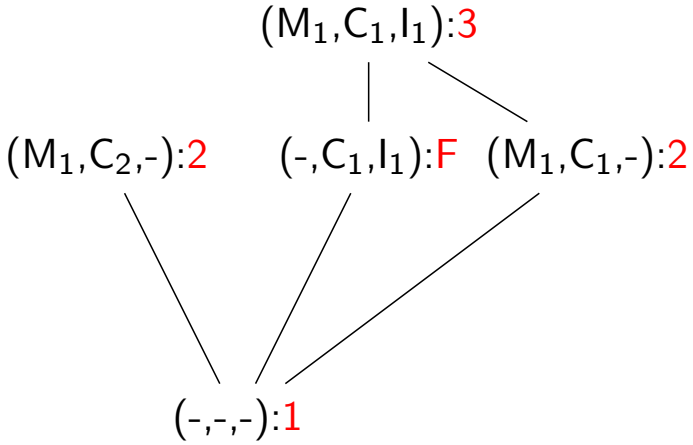


How it works

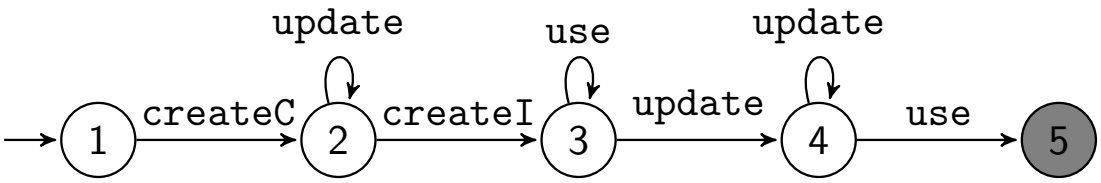


- We extend the existing (M_1,C_1,-) to get (M_1,C_1,I_1)
- As (M_1,C_1,-) is maximal, we initialise this binding with state 2 and make the transition to state 3
- There is no createI transition from state 1 so (-,C_1,I_1) fails

Trace
createC(M_1,C_1)
createC(M_1,C_2)
createI(C_1,I_1)

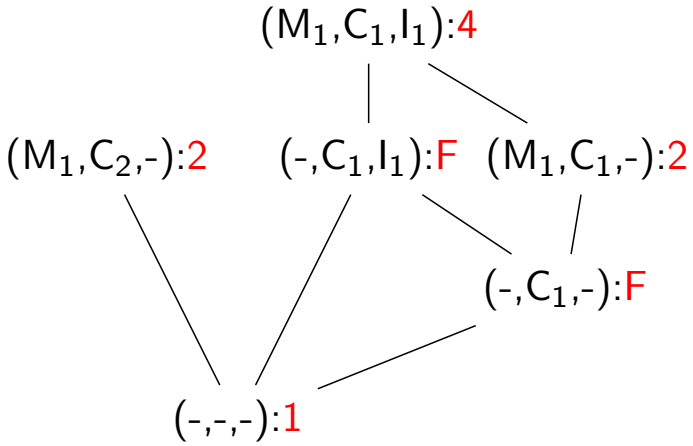


How it works

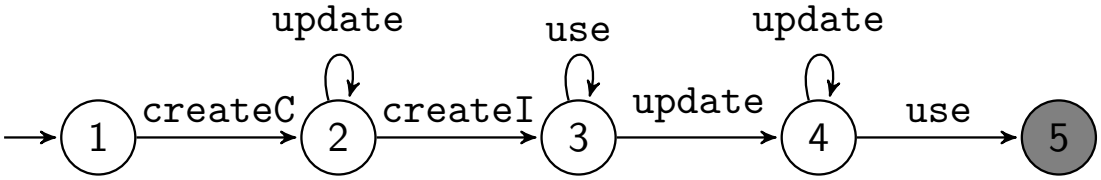


- update(C_1) is relevant to (-,C_1,-), (-,C_1,I_1), (M_1,C_1,-) and (M_1,C_1,I_1)
- (M_1,C_1,-) does a self transition and (M_1,C_1,I_1) moves to state 4

Trace
createC(M_1,C_1)
createC(M_1,C_2)
createI(C_1,I_1)
update(C_1)

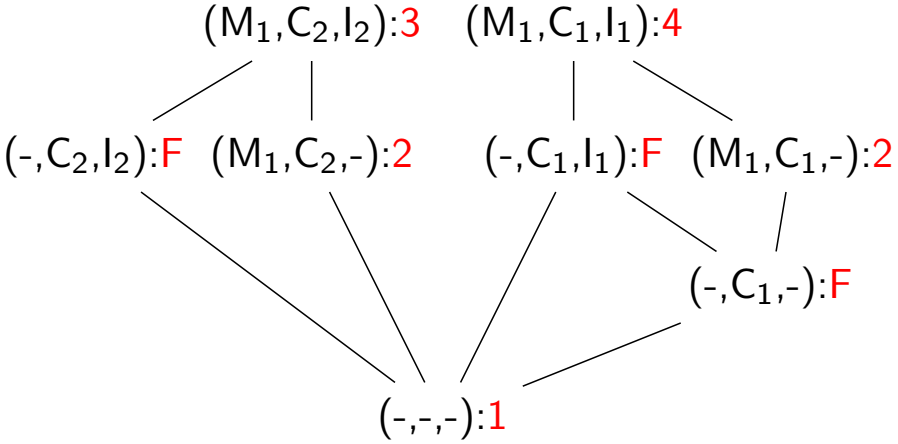


How it works

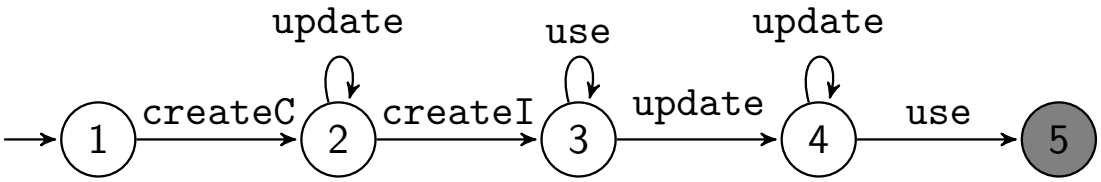


- (M_1,C_2,I_2) extends (M_1,C_2,-) and (-,C_2,I_2) extends (-,-,-)

Trace
createC(M_1,C_1)
createC(M_1,C_2)
createI(C_1,I_1)
update(C_1)
createI(C_2,I_2)

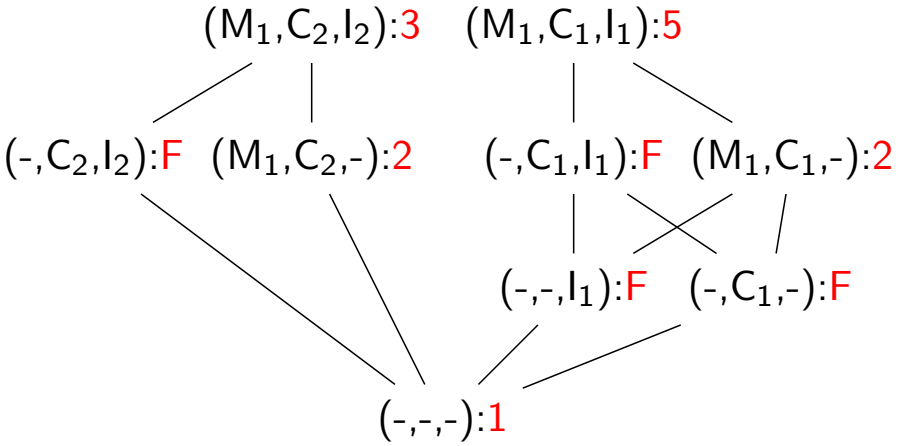


How it works



- applying use for binding (M_1,C_1,I_1) takes it to state 5

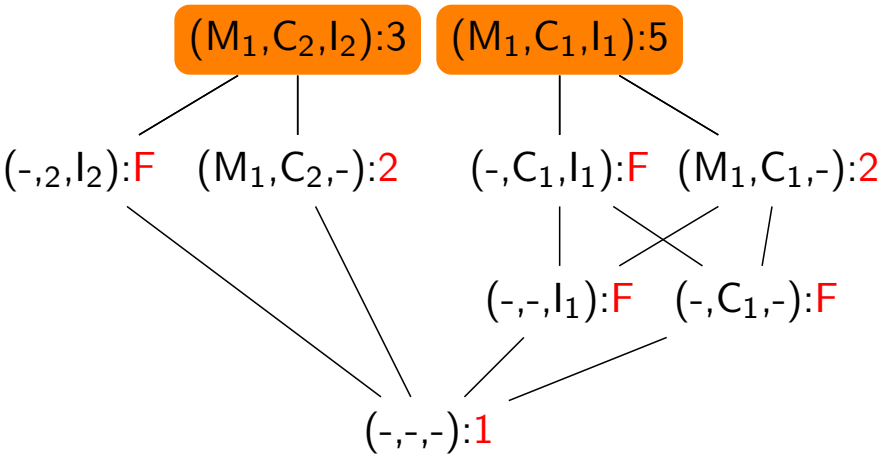
Trace
createC(M_1,C_1)
createC(M_1,C_2)
createI(C_1,I_1)
update(C_1)
createI(C_2,I_2)
use(I_1)



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
createC(M_1,C_1)
createC(M_1,C_2)
createI(C_1,I_1)
update(C_1)
createI(C_2,I_2)
use(I_1)



Second inefficiency

- Let n be the number of bindings in $\underline{dom}(\Delta)$ at a particular step.
- We access Δ 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 $\mathbb{U}$ be?

- Let $\mathbb{U} : \mathit{Bind} \to 2^{\mathit{Bind}}$ be such a map
- We want $\mathbb{U}$ to help us update Δ
- Δ should be 'union-closed' - if two compatible bindings are in Δ, their union should also be in Δ:

$$\forall \theta, \theta' \in \underline{dom}(\Delta) : \mathit{compatible}(\theta, \theta') \Rightarrow \theta \sqcup \theta' \in \underline{dom}(\Delta)$$

- $\mathbb{U}$ should be 'submap-closed' - every submap of a binding in Δ should be in $\mathbb{U}$:

$$\forall \theta \in \underline{dom}(\Delta), \forall \theta' \in \mathit{Bind} : \theta' \sqsubseteq \theta \Rightarrow \theta' \in \underline{dom}(\mathbb{U})$$

- $\mathbb{U}$ should be 'relevance-closed' - every entry in $\mathbb{U}$ should point to the relevant bindings in Δ:

$$\forall \theta, \theta' \in \underline{dom}(\Delta) : \theta \sqsubseteq \theta' \Rightarrow \theta' \in \mathtt{U}(\theta)$$

A refined algorithm

```
1  Δ : [Bind ⇀ State]; 𝕌 : Bind → 2^Bind
2  Δ ← {⊥ → q0}; 𝕌 ← ∅ for any θ ∈ Bind
3  foreach e(θ) ∈ τ in order do
4      if θ ∉ dom(Δ) then
5          foreach θm ⊏ θ (big to small) do
6              if θm ∈ dom(Δ) then break
7          defTo(θ, θm)
8          foreach θm ⊏ θ (big to small) do
9              foreach θ' ∈ 𝕌(θm)
               compatible with θ do
10                 if (θ' ⊔ θ) ∉ dom(Δ)
                   then defTo(θ' ⊔ θ, θ')
11     foreach θ' ∈ {θ} ∪ 𝕌(θ) do
12         Δ(θ') ← σ(Δ(θ'), e)
13 return Δ
```

- Initialisation
- If θ not in $\mathbb{U}$ add it and ensure closure properties

  We will look at how this is done next
- Update states for relevant bindings

Closing $\mathbb{U}$

```
1  if θ ∉ dom(Δ) then
2      foreach θm ⊏ θ (big to small) do
3          if θm ∈ dom(Δ) then break
4      defTo(θ, θm)
5      foreach θm ⊏ θ (big to small) do
6          foreach θ' ∈ 𝕌(θm)
           compatible with θ do
7              if (θ' ⊔ θ) ∉ dom(Δ) then
               defTo(θ' ⊔ θ, θ')
8
9  ...
10
11 defTo(θ, θ'):
12   Δ(θ) ← Δ(θ')
13   foreach θ'' ⊏ θ do
       𝕌(θ'') ← 𝕌(θ'') ∪ {θ}
```

- We only need to update $\mathbb{U}$ if θ is not in $\mathbb{U}$
- We first find the maximal binding in Δ (might be $\bot$)
- Use it to add θ
- Ensures closure properties
- Consier all submaps
- Attempt to create all unions
- `defTo` uses the state from the maximal binding to initialise θ
- Relevance-closes $\mathbb{U}$ for θ i.e. adds it to the $\mathbb{U}$-entry for all smaller existing bindings

Why is this better?

```
1  foreach e(θ) ∈ τ in order do
2      if θ ∉ dom(Δ) then
3          foreach θ_m ⊏ θ (big to small) do
4              if θ_m ∈ dom(Δ) then break
5          defTo(θ, θ_m)
6          foreach θ_m ⊏ θ (big to small) do
7              foreach θ' ∈ 𝕌(θ_m)
               compatible with θ do
8                  if (θ' ⊔ θ) ∉ dom(Δ)
                   then defTo(θ' ⊔ θ, θ')
9      foreach θ' ∈ {θ} ∪ 𝕌(θ) do
10         Δ(θ') ← σ(Δ(θ'), e)
11 return Δ
```

- We only update $\mathbb{U}$ if we haven't seen the event's objects before.

Optimise Common Case

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

```
1  defTo(θ, θ'):
2    Δ(θ) ← Δ(θ')
3    foreach θ'' ⊏ θ do
       𝕌(θ'') ← 𝕌(θ'') ∪ {θ}
```

How it works

- We begin with Δ containing the empty binding and initial state, and $\mathbb{U}$ empty

Trace	Δ		$\mathbb{U}$
	(-,-,-)	1	

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

Trace	Δ		𝕌	
	(-,-,-)	1	(-,-,-)	(M_1,C_1,-)
createC(M_1,C_1)	(M_1,C_1,-)	2		
			(M_1,-,-)	(M_1,C_1,-)
			(-,C_1,-)	(M_1,C_1,-)

How it works

- (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

Trace	Δ		𝕌	
	(-,-,-)	1	(-,-,-)	(M_1,C_1,-)(M_1,C_2,-)
createC(M_1,C_1)	(M_1,C_1,-)	2		
createC(M_1,C_2)	(M_1,C_2,-)	2		
			(M_1,-,-)	(M_1,C_1,-)(M_1,C_2,-)
			(-,C_1,-)	(M_1,C_1,-)
			(-,C_2,-)	(M_1,C_2,-)

How it works

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

Trace	Δ		$\mathbb{U}$	
	(-,-,-)	1	(-,-,-)	(M_1,C_1,-)(M_1,C_2,-)
createC(M_1,C_1)	(M_1,C_1,-)	2		(-,C_1,I_1)(M_1,C_1,I_1)
createC(M_1,C_2)	(M_1,C_2,-)	2		
createI(C_1,I_1)	(-,C_1,I_1)	F		
	(M_1,C_1,I_1)	3	(M_1,-,-)	(M_1,C_1,-)(M_1,C_2,-)
				(M_1,C_1,I_1)
			(-,C_1,-)	(M_1,C_1,-)(-,C_1,I_1)
				(M_1,C_1,I_1)
			(-,C_2,-)	(M_1,C_2,-)(-,C_2,I_2)
			(-,-,I_1)	(-,C_1,I_1)(M_1,C_1,I_1)
			(M_1,C_1,-)	(M_1,C_1,I_1)
			(-,C_1,I_1)	(M_1,C_1,I_1)
			(M_1,-,I_1)	(M_1,C_1,I_1)

How it works

- θ_m is (-,-,-) therefore defTo((-,C_1,-),(-,-,-)) sets (-,C_1,-) to state 1 which is updated to F by σ
- As expected $\mathbb{U}$((-,C_1,-)) = { (M_1,C_1,-),(-,C_1,I_1),(M_1,C_1,I_1) }

Trace	Δ		$\mathbb{U}$	
	(-,-,-)	1	(-,-,-)	(M_1,C_1,-)(M_1,C_2,-)
createC(M_1,C_1)	(M_1,C_1,-)	2		(-,C_1,I_1)(M_1,C_1,I_1)
createC(M_1,C_2)	(M_1,C_2,-)	2		(-,C_1,-)
createI(C_1,I_1)	(-,C_1,I_1)	F		
update(C_1)	(M_1,C_1,I_1)	4	(M_1,-,-)	(M_1,C_1,-)(M_1,C_2,-)
	(-,C_1,-)	F		(M_1,C_1,I_1)
			(-,C_1,-)	(M_1,C_1,-)(-,C_1,I_1)
				(M_1,C_1,I_1)
			(-,C_2,-)	(M_1,C_2,-)
			(-,-,I_1)	(-,C_1,I_1)(M_1,C_1,I_1)
			(M_1,C_1,-)	(M_1,C_1,I_1)
			(-,C_1,I_1)	(M_1,C_1,I_1)
			(M_1,-,I_1)	(M_1,C_1,I_1)

How it works

- θ_m is (-,-,-) so defTo((-,C_2,I_2),(-,-,-)) adds this to Δ with state 1 and applying σ updates this to F
- We consider (-,C_2,-) $\sqsubset$ (-,C_2,I_2) and use $\mathbb{U}$((-,C_2,-)) to add (M_1,C_2,I_2)

Trace	Δ		$\mathbb{U}$	
	(-,-,-)	1	(-,-,-)	(M_1,C_1,-)(M_1,C_2,-)
createC(M_1,C_1)	(M_1,C_1,-)	2		(-,C_1,I_1)(M_1,C_1,I_1)
createC(M_1,C_2)	(M_1,C_2,-)	2		(-,C_1,-)(-,C_2,I_2)
createI(C_1,I_1)	(-,C_1,I_1)	F		(M_1,C_2,I_2)
update(C_1)	(M_1,C_1,I_1)	4	(M_1,-,-)	(M_1,C_1,-)(M_1,C_2,-)
createI(C_2,I_2)	(-,C_1,-)	F		(M_1,C_1,I_1)(M_1,C_2,I_2)
	(-,C_2,I_2)	F	(-,C_1,-)	(M_1,C_1,-)(-,C_1,I_1)
	(M_1,C_2,I_2)	3		(M_1,C_1,I_1)
			(-,C_2,-)	(M_1,C_2,-)(-,C_2,I_2)
				(M_1,C_2,I_2)
			...	...
			(-,-,I_2)	(-,C_2,I_2)(M_1,C_2,I_2)
			(M_1,C_2,-)	(M_1,C_2,I_2)
			(-,C_2,I_2)	(M_1,C_2,I_2)
			(M_1,-,I_2)	(M_1,C_2,I_2)

How it works

- We can use the (-,-,I_1) entry in $\mathbb{U}$ to find the two relevant bindings
- Previously we would have had to compare (-,-,I_1) with every binding in Δ

Trace	Δ		$\mathbb{U}$	
	(-,-,-)	1	(-,-,-)	(M_1,C_1,-)(M_1,C_2,-)
createC(M_1,C_1)	(M_1,C_1,-)	2		(-,C_1,I_1)(M_1,C_1,I_1)
createC(M_1,C_2)	(M_1,C_2,-)	2		(-,C_1,-)1-¿(-,C_2,I_2)
createI(C_1,I_1)	(-,C_1,I_1)	F		(M_1,C_2,I_2)(-,-,I_1)
update(C_1)	(M_1,C_1,I_1)	5	(M_1,-,-)	(M_1,C_1,-)(M_1,C_2,-)
createI(C_2,I_2)	(-,C_1,-)	F		(M_1,C_1,I_1)(M_1,C_2,I_2)
use(I_1)	(-,C_2,I_2)	F	(-,C_1,-)	(M_1,C_1,-)(-,C_1,I_1)
	(M_1,C_2,I_2)	3		(M_1,C_1,I_1)
			(-,C_2,-)	(M_1,C_2,-)(-,C_2,I_2)
				(M_1,C_2,I_2)
			(-,-,I_1)	(-,C_1,I_1)(M_1,C_1,I_1)
			(M_1,C_1,-)	(M_1,C_1,I_1)
			...	...

How it works

Trace	Δ		$\mathbb{U}$	
	(-,-,-)	1	(-,-,-)	$(M_1,C_1,-)(M_1,C_2,-)$
createC(M_1,C_1)	(M_1,C_1,-)	F		(-,C_1,I_1)(M_1,C_1,I_1)
createC(M_1,C_2)	(M_1,C_2,-)	2		(-,C_1,-)(-,C_2,I_2)
createI(C_1,I_1)	(-,C_1,I_1)	F		(M_1,C_2,I_2)(-,-,I_1)
update(C_1)	(M_1,C_1,I_1)	5	(M_1,-,-)	(M_1,C_1,-)(M_1,C_2,-)
createI(C_2,I_2)	(-,C_1,-)	F		(M_1,C_1,I_1)(M_1,C_2,I_2)
use(I_1)	(-,C_2,I_2)	F	(-,C_1,-)	(M_1,C_1,-)(-,C_1,I_1)
	(M_1,C_2,I_2)	3		(M_1,C_1,I_1)
			(-,C_2,-)	(M_1,C_2,-)(-,C_2,I_2)
				(M_1,C_2,I_2)
			(-,-,I_1)	(-,C_1,I_1)(M_1,C_1,I_1)
			(M_1,C_1,-)	(M_1,C_1,I_1)
			(-,C_1,I_1)	(M_1,C_1,I_1)
			(M_1,-,I_1)	(M_1,C_1,I_1)
			(-,-,I_2)	(-,C_2,I_2)(M_1,C_2,I_2)
			(M_1,C_2,-)	(M_1,C_2,I_2)
			(-,C_2,I_2)	(M_1,C_2,I_2)
			(M_1,-,I_2)	(M_1,C_2,I_2)

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


Parametric Trace Slicing and Monitoring: F. Chen and G. Roşu. In TACAS 2009.


An Overview of the MOP Runtime Verification Framework: P. Meredith, D. Jin, D. Griffith, F. Chen and G. Roşu. In the International Journal on Software Techniques for Technology Transfer.

Part V
RuleR

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.

Syntax : high level

```
1
2  ruler <name>{
3     observes : <obs-def> list
4
5     <mod><rule-name>(<var-def> list){
6        condition -> obligation1
7                   "|" obligation2
8        ...
9     }
10     ...
11
12    initials  : <rule-name> list
13    [forbidden : <rule-name> list]
14    [succeed : <rule-name> list]
15    [assert  : <rule-name> list]
16 }
```

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 $\rightarrow$ obligations" parts

Syntax : low level

- This is only part of the full RuleR syntax.

```
<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>
<literal>    ::= <atom> | !<atom>
<p-literal>  ::= <p-atom> | !<p-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

    new `Ruler`(String,String,boolean)
    i.e.
    new `Ruler`(`"spec.ruler"`, `"out-file.txt"`, `false`)
    spec name, out file, use timing
  - a dispatch method of the form

    `dispatch`(String,Object[])
    i.e.
    `dispatch`(`"grant"`, **new Object[]**{ wheels })
    name, parameters
  - A verdict enumeration for returning results

    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

```
1   RuleR monitor = new RuleR(
2       "specs/RespectPriorities.ruler",
3       "out/RespectPriorities.out",
4       false);
```

Recording events

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

```
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  after(Resource resource1, Resource resource2) :
38    call(void missioncontrol.ResourceTable.addConflict
39    (Resource,Resource)) && args(resource1,resource2){
40     handle(monitor.dispatch("conflict",
41                     new Object[]{resource1,resource2}));
42  }
43  ...
```

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

```
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

```
49  after() :
50    execution(void missioncontrol.Main.main(String[])){
51     handle(monitor.dispatch("shutdown",new Object[]{}));
52  }
```

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.

```
1  private static handle(Signal verdict){
2   switch(verdict){
3     case TRUE          :
4     case STILL_TRUE    : break;
5     case STILL_FALSE   : log.println("Waiting␣for␣a␣Rescind");
6                          break;
7     case FALSE         : log.println("Failed␣:␣"+
8                              "outstanding␣Rescind␣Request");
9   }
10 }
```

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.

```
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.

```
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

```
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.

Trace	Rule Activations
	Start()

Building rule activations

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

Trace	Rule Activations
`conflict(w,a)`	Start() C(w,a) C(a,w)

```
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.

Trace	Rule Activations
conflict(w,a)	Start()
priority(a,w)	C(w,a)
	C(a,w)
	P(a,w)

```
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 priority(x:obj,y:obj).

Building rule activations

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

Trace	Rule Activations
conflict(w,a)	Start()
priority(a,w)	C(w,a)
request(w)	C(a,w)
	P(a,w)

- Nothing matches.

Building rule activations

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

Trace	Rule Activations
conflict(w,a)	Start()
priority(a,w)	C(w,a)
request(w)	C(a,w)
grant(w)	P(a,w)
	G(w)

```
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 grant(x:obj).

Building rule activations

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

Trace	Rule Activations
conflict(w,a)	Start()
priority(a,w)	C(w,a)
request(w)	C(a,w)
grant(w)	P(a,w)
request(a)	G(w)
	Res(w)

```
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 with request(y:oby).
- C(w,a) and P(a,w) exist.
- Cannot fint 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.

Trace	Rule Activations
conflict(w,a)	Start()
priority(a,w)	C(w,a)
request(w)	C(a,w)
grant(w)	P(a,w)
request(a)	G(w)
rescind(w)	~~Res(w)~~

```
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.

Trace	Rule Activations
conflict(w,a)	Start()
priority(a,w)	C(w,a)
request(w)	C(a,w)
grant(w)	P(a,w)
request(a)	~~G(w)~~
rescind(w)	
cancel(w)	

```
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.

Trace	Rule Activations
conflict(w,a)	Start()
priority(a,w)	C(w,a)
request(w)	C(a,w)
grant(w)	P(a,w)
request(a)	
rescind(w)	
cancel(w)	G(a)
grant(a)	

```
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 grant(x:obj).

Building rule activations

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

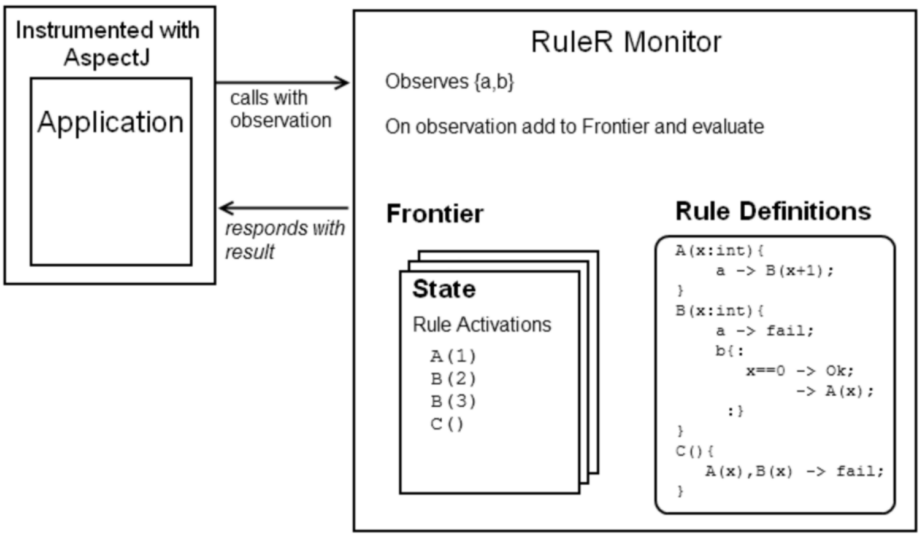
Trace	Rule Activations
conflict(w,a)	Start()
priority(a,w)	C(w,a)
request(w)	C(a,w)
grant(w)	P(a,w)
request(a)	
rescind(w)	
cancel(w)	G(a)
grant(a)	
shutdown	

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

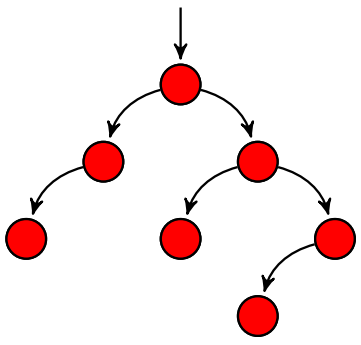
- A Rule Definition
  - Has a name.
  - Has a modifier.
  - Is parameterised by (typed) variables.
  - Associates conditions with obligations i.e., $\underbrace{\texttt{cancel(x)}}_{\text{condition}} \rightarrow \underbrace{\texttt{!G(x)}}_{\text{obligation}}$.
- A Rule Activation
  - Is associated with 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.

```
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);
  :}
}
```

```
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  PROCESS(obs):
2  newF = ∅;
3  foreach s ∈ frontier do
4      S = { { ra ∈ s |
5            RS(ra.name).mod ≠ step } };
6      foreach ra ∈ s do
7          rule = RS(ra.name);
8          foreach (c → O) ∈ rule.body do
9              foreach b ∈ unify(ra,c,s) do
10                 S = { s'∪ b(o) |
11                       s' ∈ S, o ∈ O};
12                 if r.mod=state then
13                     S = {s'-ra | s' ∈S}
14     newF ∪ = { s' ∈ S | s'∩ assert ≠ ∅};
15 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  CHECK(frontier):
2  collapse frontier;
3  if frontier= ∅ then
4      return False
5  if ∃s ∈frontier s = ∅ or s ∩ Success ≠ ∅
   then
6      return True
7  if ∃s ∈frontier. s ∩ Forbidden == ∅
   then
8      return Still_True;
9  else
10     if ∀s ∈frontier. s ∩ Forbidden ≠ ∅
       then
11         return Still_False;
12     else
13         return Unkown;
```

- 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. `command`(name,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:
  - `command`(name,id)
  - `fail`(name,id)
  - `succeed`(name,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

```
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 Run-time Monitoring: from EAGLE to RuleR: H. Barringer, D. Rydeheard and K. Havelund. In the Journal of Logic Computation, 2010.


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

Part VI

TraceContract
An Internal DSL for Trace Analysis

Recap

- We have seen three DSLs (Domain Specific Languages) for RV:
  - TraceMatches
  - JavaMOP
  - RuleR
- TraceMatches and JavaMOP focus on effiency 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"*.

TraceContract

- 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.
  - $\Box\ p = p \wedge \bigcirc(\Box\ p)$
  - $\Diamond\ p = p \vee \bigcirc(\Diamond\ p)$
  - $p\ \mathcal{U}\ q = q \vee (p \wedge \bigcirc(p\ \mathcal{U}\ q))$

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.

TraceContract
by Example

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

```
1 abstract class Event
2
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).

```
1  monitor CommandMustSucceed {
2    always {
3      Command(n,x) => RequireSuccess(n,x)
4    }
5
6    hot RequireSuccess(name,number) {
7      Fail (name,number) => error
8      Success(name,number) => ok
9    }
10 }
```

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.

```
1  class CommandMustSucceed extends Monitor[Event] {
2    require {
3      case Command(n, x) => RequireSuccess(n, x)
4    }
5
6    def RequireSuccess(name: String, number: Int) =
7      hot {
8        case Fail (‘name‘, ‘number‘) => error
9        case Success(‘name‘, ‘number‘) => ok
10     }
11 }
```

Inlining the call of *RequireSuccess(n,x)*

- Since *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

- TraceContract also offers future time linear temporal logic (LTL).
- allowing to write events as formulas, negations, propositional formulas, and temporal.
- ϕ until ψ means: ψ must eventually hold, and until then ϕ 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.

```
1 class First10CommandsMustSucceed extends Monitor[Event] {
2   var count = 0
3   require {
4     case Command(n, x) if count < 10 =>
5       count = count + 1
6       not(Fail(n, x)) until (Success(n, x))
7   }
8 }
```

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.

```
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

```
1  class AlternatingCommandSuccess extends Monitor[Event] {
2    property(s1)
3
4    def s1: Formula =
5      state {
6        case Command(n, x) => s2(n, x)
7        case _ => error
8      }
9
10   def s2(name: String, number: Int) =
11     state {
12       case Success(‘name‘, ‘number‘) => s1
13       case _ => error
14     }
15 }
```

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.

```
1  class SuccessHasAReason extends Monitor[Event] {
2    case class Commanded(name: String, nr: Int) extends Fact
3
4    require {
5      case Command(n, x) => Commanded(n, x) +
6      case Success(n, x) =>
7        if (Commanded(n, x) ?)
8          Commanded(n, x) -
9        else
10         error
11   }
12 }
```

The ?- abbreviation

- We can we 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*.

```
1 class SuccessHasAReason extends Monitor[Event] {
2   case class Commanded(name: String, nr: Int) extends Fact
3
4   require {
5     case Command(n, x) => Commanded(n, x) +
6     case Success(n, x) => Commanded(n, x) ?-
7   }
8 }
```

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.

```
1 class CommandRequirements extends Monitor[Event] {
2   monitor(
3     new CommandMustSucceed,
4     new MaxOneSuccess,
5     new SuccessHasAReason)
6 }
```

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.

```
1  object TraceAnalysis extends Application {
2    val trace: List[Event] =
3      List(
4        Command("STOP_DRIVING", 1),
5        Command("TAKE_PICTURE", 2),
6        Fail("STOP_DRIVING", 1),
7        Success("TAKE_PICTURE", 2),
8        Success("SEND_TELEMETRY", 42))
9
10   val monitor = new CommandRequirements
11   monitor.verify(trace)
12 }
```

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.

```
1  object TraceAnalysis extends Application {
2    val monitor = new CommandRequirements
3    monitor.verify(Command("STOP_DRIVING", 1))
4    monitor.verify(Command("TAKE_PICTURE", 2))
5    monitor.verify(Fail("STOP_DRIVING", 1))
6    monitor.verify(Success("TAKE_PICTURE", 2))
7    monitor.verify(Success("SEND_TELEMETRY", 42))
8    monitor.end()
9  }
```

Result

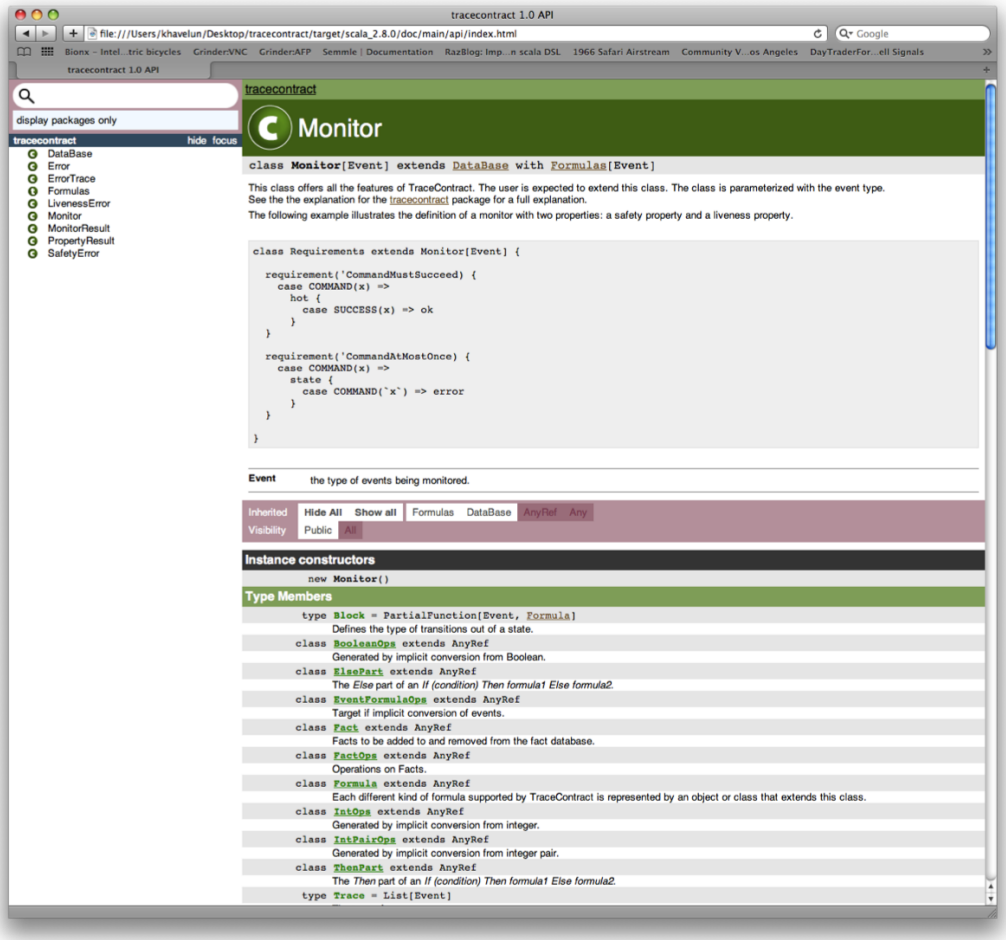
```
CommandMustSucceed property violated
Violating event number 3: Fail(STOP_DRIVING,1)
Error trace:
  1=Command(STOP_DRIVING,1)
  3=Fail(STOP_DRIVING,1)


SuccessHasAReason property violated
Violating event number 5: Success(SEND_TELEMETRY,42)
Error trace:
  5=Success(SEND_TELEMETRY,42)
```

TraceContract

an overview of functions

ScalaDoc documentation of API



ScalaDoc documentation of API

```
def eventuallyGt(n: Int)(formula: Formula): Formula
```
Eventually true after *n* steps.

```
def eventuallyLe(n: Int)(formula: Formula): Formula
```
Eventually true in maximally *n* steps.

```
def eventuallyLt(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.

```
def getMonitorResult: MonitorResult[Event]
```
Returns the result of a trace analysis for this monitor.

```
def getMonitors: List[Monitor[Event]]
```
Returns the sub-monitors of a monitor.

```
def globally(formula: Formula): Formula
```
Globally true (an LTL formula).

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

```
def hot(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 *e* matches the *block*, that is, until *block.isDefinedAt(e) == true*, in which case the state formula evaluates to *block(e)*.

At the end of the trace a *hot state* formula evaluates to False.

As an example, consider the following monitor, which checks the property: *"a command x eventually should be followed by a success"*:

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

block partial function representing the transitions leading out of the state.

returns the *hot state* formula.

definition classes: Formulas

```
def informal(name: Symbol)(explanation: String): Unit
```
Used to enter explanations of properties in informal language.

```
def informal(explanation: String): Unit
```
Used to enter explanations of properties in informal language.

```
def matches(predicate: PartialFunction[Event, Boolean]): Formula
```
Matches current event against a predicate.

```
def monitor(monitors: Monitor[Event]*): Unit
```
Adds monitors as sub-monitors to the current monitor.

```
def never(formula: Formula): Formula
```
Never true (an LTL-inspired formula).

Features by category: state functions

```
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

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

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

Temporal logic

```
1    ...
2    // propositional logic:
3    def matchEvent (predicate: Event =?=> Boolean): Formula
4    object True extends Formula
5    object False extends Formula
6    def not (formula: Formula): Formula
7
8    // temporal operators:
9    def globally (formula: Formula): Formula
10   def eventually (formula: Formula): Formula
11   def eventuallyLe (n: Int)(formula: Formula): Formula
12   def weaknext (formula: Formula): Formula
13   def strongnext (formula: Formula): Formula
14   ...
```

Infix operators defined in class Formula

```
1    ...
2    class Formula {
3      // propositional logic:
4      def and (that: Formula): Formula
5      def or (that: Formula): Formula
6      def implies (that: Formula): Formula
7
8      // temporal operators:
9      def unless (that: Formula): Formula
10     def until (that: Formula): Formula
11     def upto (block: Block): Formula
12   }
13   ...
```

Rule-based programming

```
1   class Fact { ... }
2
3   def matchFact (pred: Fact =?=> Boolean): Boolean
4
5   implicit def convFact2FactOps (fact: Fact): FactOps
6
7   class FactOps {
8     def + : Unit
9     def − : Unit
10    def ? : Boolean
11    def ˜ : Boolean
12    def ?− : Boolean
13    def ˜+ : Boolean
14  }
```

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).

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

- We will see the implementation of these implicit functions later.

Declaring and verifying properties

```
1    ...
2    // declaring properties inside a monitor
3    def informal (explanation: String): Unit
4    def property (formula: Formula): Unit
5    def property (name: Symbol)(formula: Formula): Unit
6    def require (block: Block): Unit
7    def requirement (name: Symbol)(block: Block): Unit
8    def monitor (monitors: Monitor[Event]*): Unit
9
10   // verification :
11   def select ( filter : Event =?=> Boolean): Unit
12   def verify (trace: Trace): MonitorResult[Event]
13   def verify (event: Event): Unit
14   def end (): Unit
15   def getMonitorResult : MonitorResult[Event]
16    ...
```

Towards a TraceContract calculus

$$
\begin{aligned}
\langle TC \rangle \quad ::= \; & \textbf{matchEvent} \text{ `\{'} \langle EventPred \rangle \text{ `\}'} \\
| \; & \textbf{macthFact} \text{ `\{'} \langle FactPred \rangle \text{ `\}'} \\
| \; & \neg \langle TC \rangle \\
| \; & \langle TC \rangle \vee \langle TC \rangle \\
| \; & \bigcirc \langle TC \rangle \\
| \; & \langle TC \rangle \, \mathcal{U} \, \langle TC \rangle \\
| \; & \langle StateKind \rangle \text{ `\{'} \langle TransitionBlock \rangle \text{ `\}'} \\
| \; & \langle BoolExp \rangle \\
| \; & \langle Stmt \rangle \\
| \; & \langle Fact \rangle \; (\text{`+'} \mid \text{`-'})
\end{aligned}
$$

$$\langle StateKind \rangle ::= \textbf{state} \mid \textbf{hot} \mid \textbf{always} \mid \textbf{weak} \mid \textbf{strong} \mid \textbf{step}$$

$$\langle EventPred \rangle ::= \text{partial function of type: } \langle Event \rangle \stackrel{\sim}{\Rightarrow} \{true, false\}$$

$$\langle FactPred \rangle ::= \text{partial function of type: } \langle Fact \rangle \stackrel{\sim}{\Rightarrow} \{true, false\}$$

$$\langle TransitionBlock \rangle ::= \text{partial function of type: } \langle Event \rangle \stackrel{\sim}{\Rightarrow} \langle TC \rangle$$

TraceContract
implementation

The following slides represent a complete implementation of a mini-TraceContract, focusing only on conceptual ideas. It is sufficient for the examples provided on previous slides.

Rule system

```
1  trait RuleSystem {
2    trait Fact
3    var facts: Set[Fact] = Set()
4    var toRecord: Set[Fact] = Set()
5    var toRemove: Set[Fact] = Set()
6
7    implicit def convE(fact: Fact) = new {
8      def + : Unit = { toRecord += fact }
9      def - : Unit = { toRemove += fact }
10     def ? : Boolean = facts contains fact
11     def ~ : Boolean = !(facts contains fact)
12
13     def ?- : Boolean = {toRemove += fact; facts contains fact}
14     def ~+ : Boolean = {toRecord += fact; !(facts contains fact)}
15   }
16
17   def matchFact(pred: Fact =?=> Boolean): Boolean = {
18     facts exists (pred orElse { case _ => false })
19   }
20
21   def updateFacts() {
22     toRemove foreach (facts -= _)
23     toRecord foreach (facts += _)
24     toRecord = Set()
25     toRemove = Set()
26   }
27 }
```

Class Monitor

```
1  trait Monitor[Event] extends RuleSystem {
2    var current: Formula = True
3    var monitors: List[Monitor[Event]] = List()
4
5    type Block = Event =?=> Formula
6
7    trait Formula {
8      def apply(e: Event): Formula
9      def reduce: Formula = this
10
11     def and(that: Formula) = And(this, that) reduce
12     def or(that: Formula) = Or(this, that) reduce
13
14     def until(f: Formula) = Until(this, f)
15     def upto(b: Block) = Upto(this, b)
16   }
17
18   ... // what follows on the next slides goes here
19 }
```

True and false

```
1    case object True extends Formula {
2      def apply(e: Event) = this
3    }
4
5    case object False extends Formula {
6      def apply(e: Event) = this
7    }
8
9
10
11   def error = False
12   def ok = True
```

Propositional logic

```
1   implicit def convEvent2Formula(e: Event): Formula = Now(e)
2   implicit def convBoolean(b: Boolean): Formula = if (b) True else False
3   implicit def convUnit(u: Unit): Formula = True
4
5   case class Now(expect: Event) extends Formula {
6     def apply(e: Event): Formula =
7       if (expect == e) True else False
8   }
9
10  case class matchEvent(p: Event =?=> Boolean) extends Formula {
11    def apply(event: Event): Formula = {
12      if (p.isDefinedAt(event))
13        p(event)
14      else
15        false
16    }
17  }
```

Propositional operators

```
1   case class not(f: Formula) extends Formula {
2     def apply(e: Event): Formula = not(f(e)).reduce()
3
4     override def reduce(): Formula = {
5       f match {
6         case True => False
7         case False => True
8         case _ => this
9       }
10    }
11  }
12
13  case class And(f1: Formula, f2: Formula) extends Formula {
14    def apply(e: Event) = And(f1(e), f2(e)) reduce
15
16    override def reduce: Formula =
17      (f1, f2) match {
18        case (False, _) | (_, False) => False
19        case (True, _) => f2
20        case (_, True) => f1
21        case _ => this
22      }
23  }
24
25  case class Or(f1: Formula, f2: Formula) extends Formula {
26    ... // same idea
27  }
```

Temporal Logic

```
1   case class Until(f1: Formula, f2: Formula) extends Formula {
2     def apply(e: Event): Formula =
3       Or(f2(e), And(this, f1(e)).reduce).reduce
4   }
5
6   case class strongnext(f: Formula) extends Formula {
7     def apply(e: Event): Formula = f(e)
8   }
9
10  case class globally(f: Formula) extends Formula {
11    def apply(e: Event): Formula = {
12      val formula_ = f(e)
13      if (formula_ == f)
14        this
15      else
16        And(this, formula_).reduce
17    }
18  }
19
20  case class eventually(f: Formula) extends Formula {
21    def apply(e: Event): Formula =
22      Or(this, f(e)).reduce
23  }
```

State logic

```
1   case class state(b: Block) extends Formula {
2     def apply(e: Event): Formula =
3       if (b.isDefinedAt(e)) b(e) else this
4   }
5
6   case class hot(b: Block) extends Formula {
7     def apply(e: Event) =
8       if (b.isDefinedAt(e)) b(e) else this
9   }
10
11  case class strong(b: Block) extends Formula {
12    def apply(e: Event) =
13      if (b.isDefinedAt(e)) b(e) else False
14  }
15
16  case class always(b: Block) extends Formula {
17    def apply(e: Event) =
18      if (b.isDefinedAt(e)) And(b(e), this) reduce else this
19  }
```

Upto

```
1  case class Upto(f: Formula, b: Block) extends Formula {
2    override def apply(e: Event): Formula = {
3      if (b.isDefinedAt(e)) {
4        isFinal(f) and b(e)
5      } else {
6        val formula_ = f(e).reduce
7        formula_ match {
8          case False => False
9          case True => True
10         case `f` => this
11         case _ => Upto(formula_, b)
12       }
13     }
14   }
15 }
```

Property declaration and verification

```
1  def monitor(monitorList: Monitor[Event]*) {
2    monitors ++= monitorList.toList
3  }
4
5  def property(f: Formula) {
6    current = f
7  }
8
9  def require(b: Block) = property(always(b))
10
11 def verify(e: Event) {
12   val current_ = current(e)
13   if (current_ == False && current != False)
14     println("*** safety violation " + this.getClass().getSimpleName)
15   current = current_
16   for (monitor <- monitors) monitor.verify(e)
17   updateFacts()
18 }
19
20 def verify(trace: List[Event]) {
21   for (e <- trace) {
22     println("--- " + e)
23     verify(e)
24   }
25   end()
26 }
```

The end of the trace

```
1   def end() {
2     if (! isFinal (current)) println ("*** liveness violation " + this.getClass().getSimpleName + " on " + e)
3     monitors foreach (_.end())
4   }
5
6   def isFinal (f: Formula): Boolean = {
7     f match {
8       case hot(_)
9         | strong(_)
10        | Now(_)
11        | matchEvent(_)
12        | Until(_, _)
13        | strongnext(_)
14        | eventually (_) => false
15      case Upto(f, _) => isFinal(f)
16      case not(f) => !isFinal(f)
17      case And(f1, f2) => isFinal(f1) && isFinal(f2)
18      case Or(f1, f2) => isFinal(f1) || isFinal (f2)
19      case True
20        | False
21        | state(_)
22        | always(_)
23        | globally (_) => true
24    }
25  }
```

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 if 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.

```
1  object Util {
2    type Actor = Any
3    type Resource = Any
4    type Response = Any
5  }
6
7  import Util._
```

Declaring events

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

```
1  abstract class Event
2
3  case class SendRequest(a: Actor, r: Resource) extends Event
4  case class SendCancel(a: Actor, r: Resource) extends Event
5  case class SendGrant(r: Resource, a: Actor) extends Event
6  case class SendRescind(r: Resource, a: Actor) extends Event
7  case class SendDeny(a: Actor) extends Event
8  case class AddConflict(r1: Resource, r2: Resource) extends Event
9  case class AddPriority(r1: Resource, r2: Resource) extends Event
10 case class CancelResource(a: Actor, r: Resource) extends Event
11 case object End extends Event
```

Analyzing a trace, assuming monitor: Requirements

```
1  object TraceAnalysis extends Application {
2    val trace: List[Event] =
3      List(
4        AddConflict('antenna, 'wheels),
5        AddConflict('camera, 'wheels),
6        AddConflict('drill, 'wheels),
7        AddPriority('antenna, 'wheels),
8        SendGrant('antenna, 'commandTask),
9        SendGrant('wheels, 'drivingTask),
10       CancelResource('commandTask, 'antenna),
11       CancelResource('cameraTask,'antenna)
12
13   val monitor = new ResourceRequirements
14   monitor.verify(trace)
15 }
```

The requirements

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

```
1  class ResourceRequirements extends Monitor[Event] {
2    monitor(
3      new GrantCancel,
4      new OnlyRescindGranted,
5      new RespectConflicts,
6      new RespectPriorities)
7  }
```

- 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:

  $\Box(\mathit{SendGrant}(r, a) \Rightarrow$
  $\quad \oplus(\neg \mathit{SendGrant}(r, \_)\ \mathcal{W}\ \mathit{CancelResource}(a, r))$

- 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:

  $\Box(\mathit{CancelResource}(a, r) \Rightarrow$
  $\quad \ominus(\neg \mathit{CancelResource}(a, r)\ \mathcal{S}\ \mathit{SendGrant}(r, a))$

Future time with states

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

```
1  class GrantCancel extends Monitor[Event] {
2    require {
3      case SendGrant(resource, receiver) =>
4        state {
5          case SendGrant(`resource`, _) => error
6          case CancelResource(`receiver`, `resource`) => ok
7        }
8    }
9  }
```

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`

```
1  class GrantCancel extends Monitor[Event] {
2    require {
3      case SendGrant(resource, receiver) =>
4        not(matches { case SendGrant(`resource`, _) => true }) unless
5          CancelResource(receiver, resource)
6    }
7  }
```

LTL and a predicate

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

```
1  class GrantCancel extends Monitor[Event] {
2    def grant(resource: Resource): Formula =
3      matches { case SendGrant(`resource`, _) => true }
4
5    require {
6      case SendGrant(resource, receiver) =>
7        not(grant(resource)) unless CancelResource(receiver, resource)
8    }
9  }
```

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.

```
1  class GrantCancel extends Monitor[Event] {
2    case class Granted(resource: Resource, receiver: Actor) extends Fact
3
4    require {
5      case SendGrant(resource, receiver) =>
6        Granted(resource, receiver) +
7      case CancelResource(sender, resource) =>
8        Granted(resource, sender) ?-
9    }
10 }
```

Past and future mixing states and rules

```
1  class GrantCancel extends Monitor[Event] {
2    case class Granted(resource: Resource, receiver: Actor) extends Fact
3
4    require {
5      case SendGrant(resource, receiver) =>
6        Granted(resource, receiver) +;
7        state {
8          case SendGrant(‘resource‘, _) => error
9          case CancelResource(‘receiver‘, ‘resource‘) => ok
10       }
11     case CancelResource(sender, resource) =>
12       Granted(resource, sender) ?−
13   }
14 }
```

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.

```
1  class GrantCancel extends Monitor[Event] {
2    case class Granted(resource: Resource, receiver: Actor) extends Fact
3
4    require {
5      case SendGrant(resource, receiver) =>
6        Granted(resource, receiver) ˜+
7
8      case CancelResource(sender, resource) =>
9        Granted(resource, sender) ?−
10   }
11 }
```

Resource Management: only rescind granted

Requirement OnlyRescindGranted

Only ask a task to rescind the resource if if 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

```
1  class OnlyRescindGranted extends Monitor[Event] {
2    case class Granted(resource: Resource, receiver: Actor) extends Fact
3
4    require {
5      case SendGrant(resource, receiver) =>
6        Granted(resource, receiver) +
7      case CancelResource(sender, resource) =>
8        Granted(resource, sender) -
9      case SendRescind(resource, receiver) =>
10       Granted(resource, receiver) ?
11   }
12 }
```

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

```
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

```
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

```
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

```
1  class RespectPriorities extends Monitor[Event] {
2    require {
3      case AddConflict(r1, r2) => state {
4          case AddPriority(hi, lo) if (hi, lo) =?= (r1, r2) =>
5            always {
6              case SendGrant('lo', actorLo) => state {
7                  case CancelResource('actorLo', 'lo') => ok
8                  case SendRequest(actorHi, 'hi') => state {
9                      case SendGrant('hi', 'actorHi') => error
10                     case SendRescind('lo', 'actorLo') => ok
11                   }
12             }
13           }
14       }
15   }
16 }
```

Curious about how =?= is defined?

```
1    implicit def conv[A](pair1: (A, A)) = new {
2      def =?=(pair2: (A, A)) = {
3        val (a1, a2) = pair1
4        val (b1, b2) = pair2
5        ((a1 == b1) && (a2 == b2)) || ((a1 == b2) && (a2 == b1))
6      }
7    }
```

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


Howard Barringer and Klaus Havelund: *TraceContract: A Scala DSL for Trace Analysis*. 17th International Symposium on Formal Methods (FM'11), Limerick, Ireland, 2011. LNCS Volume 6664.


Howard Barringer, Klaus Havelund, Elif Kurklu and Robert Morris: *Checking Flight Rules with TraceContract: Application of a Scala DSL for Trace Analysis*. Scala Days 2011, Stanford University, USA, June, 2011.


Klaus Havelund: *Closing the Gap Between Specification and Programming: VDM^{++} and Scala*. HOWARD-60: Higher Order Workshop on Automated Runtime verification and Debugging. Manchester, UK, December, 2011. EasyChair Proceedings Volume 1.