introduction to program monitoring

CS 119 part II

beyond assert and print
Program Monitoring

Course material for part II of CS 119 - Reliable Software: Testing and Monitoring

Caltech, third term 2009, Monday/Wednesday 2:30-3:55, Jorgensen 287

Teacher
Klaus Havelund, Phone (cell): 650-814-0877, Email: havelund@gmail.com

Teaching Assistant
Mike Flotron, Phone: 626-395-3582, Email: mflotron@cs.caltech.edu

Part I
Part I of this course on testing is taught by Alex Groce

Acknowledgements

Overview

This course offers an introduction to the theory and practice of program monitoring, or runtime verification as we shall call it. Runtime verification is the study of how to design artifacts for monitoring and analyzing program executions. Such artifacts can be used for a variety of purposes, including debug/program understanding and fault isolation. Although programmers have written more or less ad-hoc monitors since the birth of the computer, only recently (last decade) has this area achieved a status as a field on its own. In this course we shall specifically focus on methods for specifying properties of Java programs, and frameworks for monitoring such. The course will initially address the issue of program instrumentation and monitoring using aspect oriented programming and Java. This will not only provide the student with a new useful technology but will also motivate tools presented later in the course. At the end of the course the student will have gained insight into the important problems in the field and will have encountered a core selection of libraries for monitoring programs. The course will enable the student to apply monitoring in software development as well as initiate research in this field. Although the course focuses on Java, ideas extend to other languages.

We will study 3 systems: AspectJ (lectures 2-3), JavaMOP (lectures 4-7), and Ruler (lecture 8).

Systems

- AspectJ (aspect oriented programming)
- MOP (monitoring oriented programming)
- Ruler (rule-based monitoring)

Lecture Plan

Lecture 1, Monday May 4 - Introduction
- resources (covers lectures 1-3):
  1. An Overview of AspectJ
  2. AspectIJ
standing order: sell when price drops more than 2% within 1 day.
**Definition**

*runtime verification* is the study of how to design artifacts for monitoring and analyzing system executions. Such artifacts can be used for a variety of purposes, including testing/program understanding and fault protection.
the wider version

runtime verification is the study of how to get as much out of your runs possible.
specification and programming
specification and programming
specification and programming
specification and programming

test input/schedule generation
specification and programming
specification and programming

specification

specification learning

program

input

output
specification and programming
purposes

• correctness checking
  – debugging and testing
  – fault protection/dependability
• security/intrusion detection
• aspect oriented programming
• program understanding
field still has many names

- runtime verification
- runtime monitoring
- runtime checking
- runtime result checking
- runtime reflection
- monitoring oriented programming
- design by contract
- runtime analysis
- dynamic analysis
- trace analysis
- fault protection
Runtime Verification

RV

Events 2001 - 2009

Background

The Runtime verification workshop series was initiated in 2001, and workshops have occurred each year since then. The objective of these workshops is to bring scientists from both academia and industry together to debate on how to monitor, analyze and guide the execution of programs. The ultimate longer term goal is to investigate the use of lightweight formal methods applied during the execution of programs from the following two points of view. On the one hand, whether runtime application of formal methods is a viable complement to the traditional methods proving programs correct before their execution, such as model checking and theorem proving. On the other hand, whether formality improves traditional ad-hoc monitoring techniques used in performance monitoring, distributed debugging, etc. Dynamic program monitoring and analysis can occur during testing or during operation.

Future Event

RV 2009
The 9th International Workshop on Runtime Verification.

Past Events

RV'01 First Workshop on Runtime Verification, Paris, France, July 2001
RV'02 Second Workshop on Runtime Verification, Copenhagen, Denmark, July 2002
RV'03 Third Workshop on Runtime Verification, Boulder, USA, July 2003
RV'04 Fourth Workshop on Runtime Verification, Barcelona, Spain, April 2004
RV'05 Fifth Workshop on Runtime Verification, Edinburgh, Scotland, July 2005
FATES/RV'06 First Joint Workshop on Formal Aspects of Testing and Runtime Verification, Seattle, USA, August 2006
RV'07 Seventh Workshop on Runtime Verification, Vancouver, Canada, March 2007
RV'08 Eighth Workshop on Runtime Verification, Budapest, Hungary, March 2008
Dagstuhl First Dagstuhl seminar on Runtime Verification, Schloss Dagstuhl, Saarland, Germany, January 3-6, 2007

Discussion Lists etc.
Extended deadline: March 25, 2009

Welcome to RV’09

The objective of RV’09 is to bring scientists from both academia and industry together to debate on how to monitor and analyze the execution of programs, for example by checking conformance with a formal specification written in temporal logic or some other form of history tracking logic. The purpose might be testing a piece of software before deployment, detecting errors after deployment in the field and potentially triggering subsequent fault protection actions, or the purpose can be to augment the software with new capabilities in an aspect oriented style. The longer term goal is to investigate whether the use of lightweight formal methods applied during the execution of programs is a viable complement to the current heavyweight methods proving programs correct always before their execution, such as model checking and theorem proving. This year’s workshop is organized as a satellite event of CAV.

The proceedings of RV’09 will be published in Lecture Notes in Computer Science.

Interests

The subject covers several technical fields as outlined below.

- Specification Languages and Logics. Formal methods scientists have investigated logics and developed technologies that are suitable for model checking and theorem proving, but monitoring can reveal new observation-based foundational logics.
Latest News

The submission deadline has been extended to Thursday, April 23. The notification deadline has been updated to Friday, May 8. The camera-ready copy deadline has been updated to Friday, May 15.

Call for Papers

Dynamic analysis techniques are increasingly used to complement more traditional static analysis. Approaches based on static analysis operate on a static representation of the program, considering all possible (and some infeasible) behaviors, and are thus complete, but often imprecise. Dynamic analysis techniques, conversely, reason over a set of program executions and analyze only observed behaviors. Dynamic analysis includes both offline techniques, which operate on some captured representation of the system's behavior (e.g., a trace), and run-time techniques, which analyze the system's behavior on the fly, while the system is executing. Although inherently incomplete, dynamic analyses can be more precise than their static counterparts and show promise in aiding the understanding, development, and maintenance of robust and reliable large-scale systems. Moreover, the data they provide enable statistical inferences to be made about program behavior. In recent years, both practitioners and researchers are realizing that the limitations of static analysis can be overcome by integrating static and dynamic analysis, and that the performance of dynamic analysis can in turn be improved by leveraging static analysis.

The overall goal of WODA 2009 is to bring together researchers and practitioners working in all areas of dynamic analysis to discuss new issues, share results and ongoing work, and foster collaborations. This workshop will focus on achieving a consensus among the participants as to the structure of the field, the important future research directions this field should take, inputs needed from other research areas, and outputs that could benefit other research domains.

Areas of interest include, but are not limited to:

- Development of dynamic analysis tools and frameworks
- Efficient instrumentation techniques
- Fault detection and debugging
- Performance analysis and optimization techniques
- Program evolution
- Remote analysis and measurement of software systems
- Runtime monitoring
- Software testing
- Statistical reasoning techniques
- Synergies between static and dynamic analysis techniques
- Visualization and classification of program behavior
- Analysis of program usage
- Relating user feedback to execution dynamics

The workshop will be a one-full-day workshop, structured to encourage discussion and develop research collaborations.

Accepted WODA papers will be included in the ISSTA proceedings and thus will be accessible to the workshop participants before the workshop to facilitate interaction and discussion. WODA papers will also be published in the ACM Digital Library.

Important Dates

- Submissions: Thursday, April 23, 2009 by 23:59:59 Aplia time
- Author notification: Friday, May 8, 2009
- Camera-ready copy: Wednesday, May 15, 2009
comparison of techniques

we are giving up on coverage to write better specifications and scale
combining
static and dynamic analysis

program + spec

static analyzer

proof obligations

monitor the proof obligations
components

- instrumentation language (I)
- property specification language (P)
- reaction language (R)
a trace view of execution

• a formal view of an execution is to consider it as a sequence \( \sigma \) of program states: \( \sigma = s_1 s_2 s_3 \ldots s_n \)

• during program execution we are at any point in time in the present moment now where the past is known and the future is unknown.
instrumentation

- instrumentation language (I)
- property specification language (P)
- reaction language (R)
while(true){
    if(x>0)lock(L);
    x = shared;
    shared = f(x);
    unlock(L);
}
while(more()){
  if(x>10){
    lock(L);
    logLock(t,L);
  }
  x = shared;
  shared = f(x);
  release(L);
  logRelease(t,L);
}

Instrument program.
For example using Aspect Oriented Programming.

execute

monitor

Trace:
lock(t2,L)
release(t2,L)
lock(t2,L)
release(t2,L)
lock(t2,L)
release(t2,L)
lock(t2,L)
release(t2,L)
instrumentation

• manual
  – assertions
  – pre/post conditions in design by contract solutions
• automated
  – instrumentation of source code:
    • CIL (C) http://manju.cs.berkeley.edu/cil/
  – instrumentation of byte/object code
    • BCEL (Java) http://jakarta.apache.org/bcel/
    • Valgrind (C) http://valgrind.org
  – high level bytecode APIs
    • Sofya http://sofya.unl.edu
  – aspect oriented programming: this course
    • AspectJ (Java) http://www.eclipse.org/aspectj
    • AspectC (C) http://research.msrg.utoronto.ca/ACC
    • AspectC++ (C++) http://www.aspectc.org
monitor

- instrumentation language (I)
- property specification language (P)
- reaction language (R)
property language

• programming language
• program (built-in algo. focused on specific problem)
  – data race detection
  – atomicity violation
  – deadlock detection
• logic (formal system)
  – design by contract (pre/post conditions), JML for example
  – state machines
  – regular expressions
  – grammars (context free languages)
  – temporal logic (past time, future time)
  – csp/ccs (process algebra)
  – “super languages” combining several of these
  – full fledged formal specification languages (Z, VDM, …)
  – graphical languages (UML, …)
design by contract (JML)

```java
//@ public invariant 0 <= size;
/*@ requires size < elems.length-1;
   @ assignable elems[size], size;
   @ ensures size == \old(size+1);
   @ ensures elems[size-1] == x;
   @ ensures_redundantly
   @   (\forall int i; 0 <= i && i < size-1;
   @      elems[i] == \old(elems[i]));
/*@ */

public void push(Object x) {
    elems[size] = x;
    size++;
}
```
what about the rest of the trace?

- pre/post conditions are not designed for total trace view, only “now” and “previous” (old) state.
- one can of course encode a total trace view using history variables.
- but that can be rather inconvenient.
future time properties

• If A happens now  
  B must happen

\(\Box (A \rightarrow \Diamond B)\)

past  now  future
Boolean logic allows us to formulate statements about a single world/situation: now

- \( p \land q \) “p and q”
- \( p \lor q \) “p or q”
- \( p \rightarrow q \) “p implies q”
- \( \neg p \) “not p”
future time temporal logic

\[ \square (A \rightarrow \diamond B) \]

\[ p \land q \quad \text{“p and q”} \]
\[ p \lor q \quad \text{“p or q”} \]
\[ p \rightarrow q \quad \text{“p implies q”} \]
\[ \neg p \quad \text{“not p”} \]

\[ \square p \quad \text{“always p”} \]
\[ \diamond p \quad \text{“eventually p”} \]
\[ p \mathbf{U} q \quad \text{“p until q”} \]
past time properties

- If A happens now
  B must have happened

\[ \Box (A \rightarrow \Diamond B) \]
past time temporal logic

□ (A → ◆B) ?

B

□(A → ◆B)
{◆B}

□(A → ◆B)
{◆B}

true ∧ □(A → ◆B)
{◆B}

□(A → ◆B)
{◆B}

□(A → ◆B)
{◆B}

…

p ∧ q  “p and q”
p ∨ q  “p or q”
p → q  “p implies q”
¬p    “not p”

■p    “sofar p”
◆p    “previously p”
p  S  q “p since q”
state machines

- A and B should alternate
regular expressions

• A and B should alternate

(A B)*
context free grammars

- $A^n B^n$

```
unary_expression
  : postfix_expression
  | INC_OP unary_expression
  | DEC_OP unary_expression
  | unary_operator cast_expression
  | SIZEOF unary_expression
  | SIZEOF '(' type_name ')
```

```
S → A S B | ε
```

A A A A B B B B

past now future
process algebra

- A, B randomly interleaved with C, D

\[
\begin{align*}
  p &= q \parallel r \\
  q &= A \rightarrow B \rightarrow q \\
  r &= C \rightarrow D \rightarrow r
\end{align*}
\]
expressiveness

lectures

Java

2,3

“super” logics

8

context free grammars

state machines

regular expressions

temporal logic

4,5

6
monitor integration

- **offline**
  - analyzing log file / trace dump

- **online**
  - **outline**
    - monitor runs in parallel with application.
    - communication for ex. over socket.
      - synchronous (appl. waits for response)
      - asynchronous (buffered communication)
  - **inline**
    - monitoring code is embedded into the application
violation versus validation

• violation: the normal case, checking that the system conforms to a property. A violation is reported when the property is violated.

• validation:
  – it can sometimes be easier to state property in negative form: what we do not want to happen. Hence we want reported when the bad property gets “validated”.
  – or: it is a good property and we just want to log whenever something good happens

most systems can only do one of the two forms
violation example

(green yellow red)*
violation $\phi \not\equiv$ validation $\neg \phi$

- some systems support both concepts
- one cannot achieve validation semantics just by negating a violation formula. example:
  - suppose our system can only check validation
  - violate $ab \not\equiv$ validate $\neg(ab)$.
  - reason: observe the first ‘a’ in the trace ‘ab’.
    - it validates $\neg(ab)$
    - it does not violate $ab$ yet (and will not in this case).
- some logics do not have negation (context free grammars for example)
total trace versus trace suffix semantics

- **total trace semantics**: property must be violated/validated on the whole trace

- **suffix trace semantics**:
  - **validation**: there exists a rightmost suffix for which it holds
  - **violation**: hard to make sense of
predictive analysis

hmm … looks pretty good to me!

good run

bad run
predictive analysis

shoot … some foot prints of a bug!

turn a hard to test property into an easy to test property
factual versus predictive

• factual
  – the standard case: violation/validation is what it is:
    \( \text{violation/validation}(\sigma) \Rightarrow \text{error } (\sigma) \)

• predictive
  – violation/validation is only suggestive, indicating the potential for an error in some other trace of the monitored system:
    \( \text{violation/validation}(\sigma) \rightarrow \exists \sigma' : \text{Traces } \bullet \text{error}(\sigma') \)
reaction

- instrumentation language (I)
- property specification language (P)
- reaction language (R)
interaction degree

1. error message
2. separate code execution (not interacting with monitored program, not same memory space)
3. exception in monitored program, and monitored program then deals with it
4. integrated code execution (access to memory). Here the monitor defines how to change the behavior of the monitored program
systems for Java

- Java’s assert statements, Java expressions:
  - [http://java.sun.com/j2se/1.5.0/docs/guide/language/assert.html](http://java.sun.com/j2se/1.5.0/docs/guide/language/assert.html)
- JML, pre/post conditions:
- jContractor, pre/post conditions:
- Temporal Rover (commercial), metric future time temporal logic/UML:
- Java MaC, past time temporal logic:
  - [http://rtg.cis.upenn.edu/mac/index.php3](http://rtg.cis.upenn.edu/mac/index.php3)
- JLO, future time temporal logic:
- Tracematches, regular expression suffix-validation:
  - [http://abc.comlab.ox.ac.uk/papers](http://abc.comlab.ox.ac.uk/papers)
- PQL, context free grammars, suffix-validation:
- PTQL, SQL, suffix-validation:
  - [http://techreports.lib.berkeley.edu/accessPages/CSD-04-1315](http://techreports.lib.berkeley.edu/accessPages/CSD-04-1315)
systems for Java

• LOLA:

• T-UPPAAL, timed automata:

• Jass, CSP (process algebra, pre/post conditions):
  – [http://csd.informatik.uni-oldenburg.de/~jass](http://csd.informatik.uni-oldenburg.de/~jass)

• Eagle/Hawk, recursion-based “super logic”:

• AspectJ, aspect oriented programming for Java:

• Java MOP, state machines, reg. exps, temp. logics, grammars:

• RuleR, rule-based system:
systems for C/C++

- AspectC, AspectJ-like AOP for C:
  - [http://research.msrg.utoronto.ca/ACC](http://research.msrg.utoronto.ca/ACC)
- Arachne, AOP for C - extended pointcut-language (subset of reg. exps.)
- Aspicere, AOP for C - extended pointcut-language (based on Prolog)

- AspectC++, AspectJ-like AOP for C++:
  - [http://www.aspectc.org](http://www.aspectc.org)
learning specs from runs

• DAIKON, learning JML:
  – http://groups.csail.mit.edu/pag/daikon

• Perecotta, learning temporal formulas:
  – http://www.cs.virginia.edu/perracotta
predictive systems

- Visual threads (the original Eraser data race detection algorithm)
  - http://h21007.www2.hp.com/portal/site/dspp/PAGE.template/page/document?ciid=e608bce06ee02110bce06ee02110275d6e10RCRD

- jPredictor (from the JavaMOP team)
program visualization

Thread main (polygon) executes method main in the Server Object (rectangle). Method main creates a Worker object and calls start on it.

After the call of start on the Worker object the Worker thread can run and eventually it does. The run can happen anytime after start.
Alphabet:
Given an alphabet (set) A of symbols

Language:
A language $L \subseteq A^*$ over A is a subset of $A^*$.

Property:
A property $P$ over A is a language:

$$P \subseteq A^*$$
### examples

<table>
<thead>
<tr>
<th>formula $\varphi$</th>
<th>language $L(\varphi)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Box(a \to \Diamond b)$</td>
<td>${\epsilon, cc, b, ab, aab, ...}$</td>
</tr>
<tr>
<td>$\Box(a \to \lozenge b)$</td>
<td>${\epsilon, c, ba, baa, baba, ...}$</td>
</tr>
<tr>
<td>$(ab)^*$</td>
<td>${\epsilon, ab, abab, ababab, ...}$</td>
</tr>
<tr>
<td>$S \to a \ S \ b \mid \epsilon$</td>
<td>${\epsilon, ab, aabb, aaabbb, ...}$</td>
</tr>
</tbody>
</table>

A formula in a logic compacts an infinite language.
safety properties

Safety property:
A property $P \subseteq A^*$ is a safety property if and only if $P$ is prefix closed:

$$\text{prefix}(P) = P$$

where $\text{prefix}(L) = \{ \sigma \mid \exists \sigma' \cdot \sigma \sqsubseteq \sigma' \in L \}$

intuitively: if a trace is in $P$ then all prefixes of the trace are in $P$. This is for example not true for $\square(a \rightarrow \Diamond b)$. Here ‘aab’ is in $P$, but ‘aa’ is not.
monitoring a language

A monitor: A monitor $M$ for a language $L \subseteq A^*$ is a "box" which as input takes a list of symbols $\alpha \in A$, one by one, and emits a value for the next symbol $\alpha_n$ as follows:

$$M(\omega) = \alpha_1, \alpha_2, \alpha_3, \ldots, \alpha_n$$

iff. $\omega \in L$

iff. $\not\exists \omega' \cdot \omega \land \omega' \in L$

otherwise
The monitoring problem:

Given a logic \( \mathcal{L} \).
Given a formula \( \varphi \in \text{Formulas}(\mathcal{L}) \).

Construct a monitor for the language of \( \varphi \):

\[ M_{\mathcal{L}}(\varphi) \]

For \( \mathcal{L} \in \{ \text{reg.exp, temp.logic, grammars, ...} \} \)
challenges

• instrumentation:
  – how to instrument code (languages for specifying relevant events, reactions)

• specification:
  – definition of specification languages
    • succinct
    • easy to adopt
    • expressive
  – creation of efficient monitors from specs
  – minimize impact on monitored system
  – integrate static and dynamic analysis
  – how to control application in case of violation/validation (fault protection)
overview of course

1. Introduction (today)
2. AspectJ for instrumentation (the language)
3. AspectJ for monitoring (logic : Java)
4. propositional regular expressions
5. parameterized regular expressions
6. temporal logic
7. context free grammars
8. rule-based systems

see course website for details
properties of Java library APIs

1: There should be no two calls to `next()` without a call to `hasNext()` in between, on the same iterator.
http://www.eclipse.org/aspectj

aspectj crosscutting objects for better modularity

aspectj is
- a seamless aspect-oriented extension to the Java™ programming language
- Java platform compatible
- easy to learn and use

aspectj enables
- clean modularization of crosscutting concerns, such as error checking and handling, synchronization, context-sensitive behavior, performance optimizations, monitoring and logging, debugging support, and multi-object protocols

Quick Links
- For Eclipse development: AJDT: The AspectJ Development Tools
- Popular AspectJ downloads: Latest development build | Latest stable release | More downloads...
- Popular AspectJ docs: AspectJ 5 Developer's Notebook | Programming Guide | More docs...
- Eclipse AspectJ: the book, by some of the leading AspectJ committers
- AOSD.net
- AspectJ PARC Page
public class Resource {
    void authenticate() {
    }
    void access() {
    }
}

aspect MonitorAspect {
    boolean authenticated = false;

    before() : call(void Resource.authenticate()) {
        authenticated = true;
    }

    before() : call(void Resource.access()) {
        if (!authenticated)
            error("access without authentication");
    }
}
http://fsl.cs.uiuc.edu/index.php/JavaMOP
end