Advanced Verification & Validation Tools for Java

Johann Schumann

7/31/2009

Abstract

- As more and more Java software finds it way into areas where quality, reliability, and software safety plays an important role, the verification and validation (V&V) of such code is paramount. Unfortunately, traditional approaches to software testing are extremely expensive and labor intensive, when a low software error rate is expected. Therefore, approaches and tools, which help to find errors and bugs as early as possible, which help to find hard-to-test errors, and which support the automatic generation of test suites, are extremely helpful and important.
- In this seminar, we will focus on advanced tools for V&V of Java code, some of which have been developed at NASA. All these tools are open source, so the threshold for use by students is low.

Abstract

- We will cover the following topics:
  - The role of V&V in modern Software Engineering
  - Early detection of bugs and defects (e.g., Findbugs)
  - Model Checking of Java Programs to find concurrency bugs in multithreaded programs with Java PathFinder
  - Using Model Checking to validate a graphical user interface (e.g., with Java Swing)
  - Automatic generation of Java testcases with SPF (Symbolic Path Finder)
  - Discussion of integration of V&V tools into a software process.
- This course will present a short theoretical background and will discuss practical examples and technqiues. All tools presented are open-source; Java PathFinder and SPF have been developed at NASA Ames for the V&V of safety-critical code.

Prerequisites

"This is your last chance..."

- knowledge of Java, including threads and Swing/awt.
- knowledge of eclipse (most tools run within eclipse)
- basics of software engineering

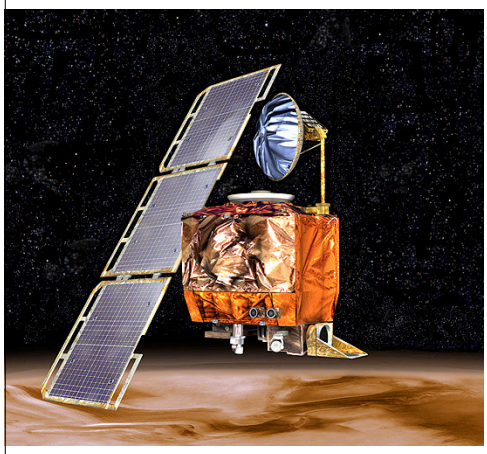
Books

- J. Schumann, *Automated Theorem Proving in Software Engineering,* Springer, 2001
- M. Mansouri-Samani, P. Mehlitz, C. Pasareanu, J. Penix, G. Brat, L. Markosian, O. O'Malley, Th. Pressburger, and W. Visser, Program Model Checking: a Practitioner's Guide, NASA/TM-2008-214577, January 2008 (available: http://sarpresults.ivv.nasa.gov/; search for "program model checking")
- B. Berard, M. Bidoit, et.al., Systems and Software Verification: Model-Checking Techniques and Tools, Springer, 2001
- U. Schoening, Logic for Computer Scientists, Birkhaeuser, 1994

more links etc. throughout the seminar

Safety-critical Software

Mars Climate Orbiter '99



- course deviated from calculated trajectory
- MCO burned up in atmosphere or flew beyond Mars (unknown)
- What happened:
  - NASA: lbs,ft,in,...
  - contractor: m,kg,..

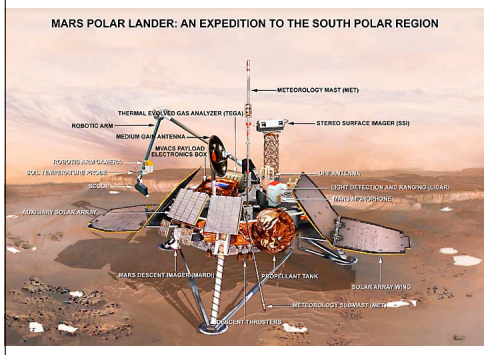
  in analysis and software
- They didn't talk to each other.

Did they learn?

- For Orion, NASA has chosen imperial units
- According to an international treaty, all units on moon are metric.

- Dual-scale units or switch?
- When will they throw the switch?

Mars Polar Lander



- MPL descended through atmosphere, deployed parachute, jettisoned parachute at low altitude, continued descend on rockets, and crashed
- What happened?

Mars Polar Lander II

- landing legs are deployed at ~200m
- shut off rocket as soon as landed
  - contact switch in leg
- Reality (what people think what happened):
  - landing legs deployed at ~200m (measured with radar altimeter)
  - landing legs unfold; vibration triggers landing switch
  - software says: switch activated, we have landed: turn off rocket engine
  - no rocket engine at 200m: bad, bad...

Software did not take other available sources into account

Mars Polar Lander III

- The software

actual:

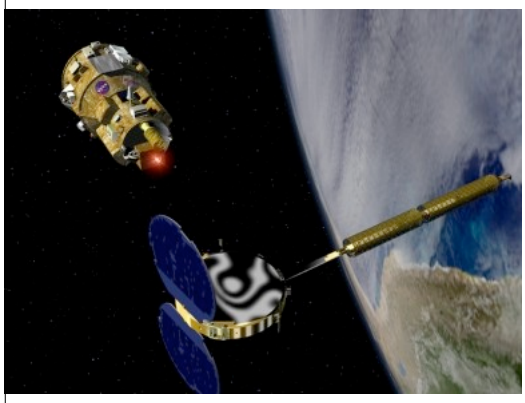
```
...
if (l_switch){
   turn_off_engine();
   }
```

correct:

```
...
if (l_switch){
   if (radar_alt < 5){
      turn_off_engine();
      }
   else {
       errors++;  // ignore spurious
        }
   }
```

Obviously no one tested the situation:
l_switch==1 AND radar_alt >100

DART-I

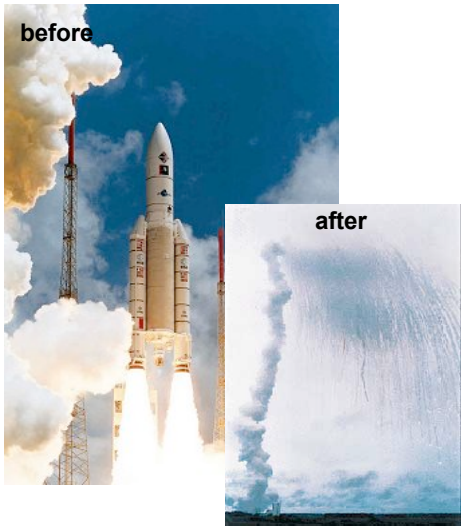


- DART spacecraft should autonomously maneuver to and dock at an “old” satellite
- What happened: DART zig-zagged around, burned its fuel, bumped into the satellite and turned itself off.

DART-I

- the Navigation software was buggy
- a new (and bad) GPS system was put on the spacecraft only few weeks prior to launch
- the GPS system caused to spacecraft to repeatedly reset its navigation to “wrong” default values
- SC zig-zagged around and missed its approach target

Ariane V



- Control software component was re-used from Ariane-IV
- Larger rocket=different parameters, leading to value overflow
- Exception handling was disabled
- Rocket went unstable and had to self-destruct

Harrier Autolander

- autolander receives altitude above ground from Radar altimeter
- Software had to deal with data drop-outs
  - if (reading.good){
    - alt = get_radar();
  - else {
    - alt = last_good_radar_value
  - }

In an experiment, the altimeter failed during a landing approach (at 10m) and almost crashed the AC into the tarmac. What happened?

B777

FAA Emergency Directive AD 2005-18-51 (Aug '05):

... we received a recent report of a significant nose-up pitch event on a Boeing Model 777-200 series air plane while climbing through 36,000 feet altitude. The flight crew disconnected the autopilot and stabilized the air plane, during which time the air plane climbed above 41,000 feet, decelerated to a minimum speed of 158 knots, and activated the stick shaker.

B777

... These errors were caused by the OPS using data from faulted (failed) sensors. OPS using data from faulted sensors, if not corrected, could result in anomalies of the fly-by-wire primary flight control, autopilot, auto-throttle, pilot display, and auto-brake systems, which could result in high pilot workload, deviation from the intended flight path, and possible loss of control of the air plane.

- Action Item: "Install previous Software Version" bugs and bug workarounds (Display of flight data) are known in this version
- install old version on all B777's
- software problem: Byzantine problem in handling sensor data

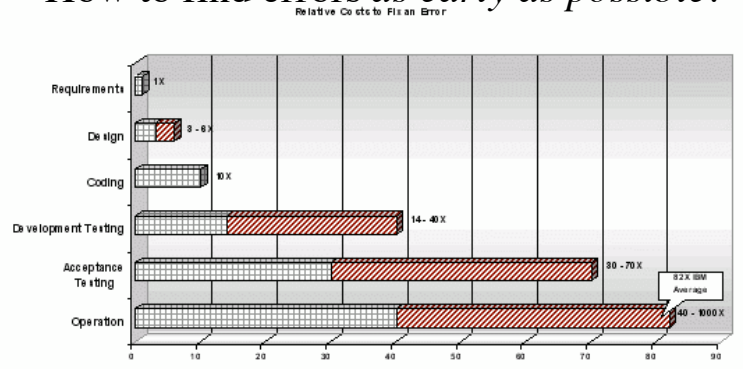
Other Failed Missions, Problems,...

- Many problems and mishaps can be attributed to faulty computers and/or software
- In safety-critical applications, such bugs can put human life at risk and can catastrophes
- Bugs and errors can be extremely costly to fix

- http://www.rvs.uni-bielefeld.de/publications/compendium/index.html computer-related incidents with Commercial Aircraft
- Peter Neumann, Computer-related Risks, Addison Wesley, 1995
- related URL + forum: http://www.csl.sri.com/users/neumann/#3

Big Questions I

- How to find errors *as early as possible?*



Bug-removal after deployment can cost up to 100x more than for finding the bug early in the SW development

Big Question II

- How to find errors/bugs as cheap and quick as possible?
  - can't spend years on testing
  - can't hire 10,000s of testers
  - *automatic* tools for
    - finding bugs
    - generating and executing tests
    - ...

Big Question III

- How to find as many and very complex errors as possible?
  - complex (logic-based) tools necessary to find difficult bugs
  - combination of different techniques

Big Question IV

- Which technology/tool to use?
  - code review
  - testing
  - model checking
  - static analysis
  - formal verification
  - ...

Big Question V

- How to do it in a “good” way?
  - Software processes and V&V processes exist (many of them)
  - Processes need to be adapted to specific needs
  - Processes need to be executed

Big Question VI

Do all approaches/tools work for all languages?

- Java
- C
- C++
- Ada
- other languages?

Summary

- Software is buggy
- Buggy software can risk lives and costs $$$
- Techniques and tools exist to help find bugs early
  - modern tools
  - expensive commercial tools (Klocwork, T-VEC, LDRA, PolySpace, Design verifier,...) and opensource tools
  - these tools are not push-button, so power users must understand underlying principles and limits
  - these tools are based upon mathematical logic, theorem proving, model checking, etc.
- These seminar covers the underlying principles and discusses various powerful tools and their applications

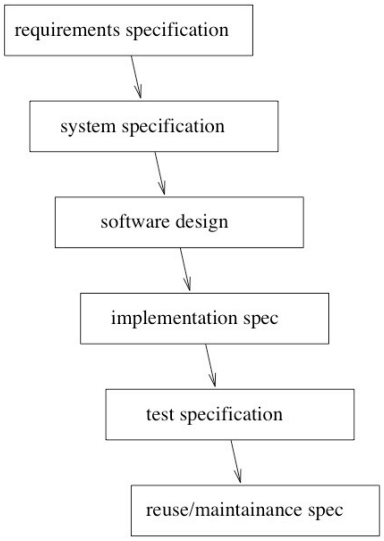
OVERVIEW

- The Software Life Cycle
- Verification and Validation
- Tools/approaches:
  - code review/static analysis/findbugs
  - Model Checking: Java Pathfinder
  - Testcase generation

Software Engineering

- "Art of Doing Software"
  - was a "dark art" for a long time rather an "engineering discipline"
- large field; here will focus on
  - elements of the software life cycle
  - verification and validation
  - levels of criticality
  - software processes and V&V processes

The Elements of SW Lifecycle



- Requirements
- Design
- Implementation
- Test/Validation
- Deployment
- Maintenance

Many refinements of the SW life-cycle elements exist.

Requirements

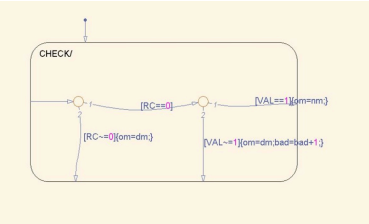
- Captures ``*What has to be done?*''
- Uses the word "Shall"
- Often just a word/power-point document
- can come out of meeting with customers
- more formal representations exist
  - high-level specification (logic-based)
  - tables (e.g. SCR)
  - UML (mostly use cases)

Design

- Captures "*How the program will look like?*"
- Typically, graphical representations and high-level language constructs (e.g., class diagrams) are used
- much more detailed than requirements
- Focuses on the "*how*" questions and on architecture
- Here come the "it's object-oriented" in

Design

- Important elements
  - definition of data structures, often on higher level: list_of_entries, sets, ...
  - static breakdown into components
  - dynamic breakdown into processes or threads
  - major decision logic
    - if-then-else cascades
    - automata
    - state charts



Implementation

- Actual coding phase
- more refinement of the architecture
  - how are the data structures implemented
  - low-level structuring
  - OS interface

But: the more *design decisions* are made during the coding phase, the harder the program is to test later on.
See Harrier example: the programmer ("junior duck") implemented the error-handling routine on his own.

Testing

- Run the program with ``test cases''
- various levels of testing
  - unit testing: test small components
  - system testing: test larger subsystems
  - integration testing: test together with environment

Testing – What you test for?

- functional test:
  - does the system do what it is supposed to do?
    - does it print the correct information,...
  - nominal cases / error cases
    - what happens if you give it a "wrong" file?
- code coverage:
  - has the entire code (line by line) been executed by at least one test case
  - different coverage metrics (->later)

Deployment

- packaging
  - how to pack and deliver software
    - upload/download Which mechanisms?
    - CD/DVD
    - licensing and checking of licenses
- installation of software
  - HW/SW requirements
  - SW/Version prerequisites
- documentation / training

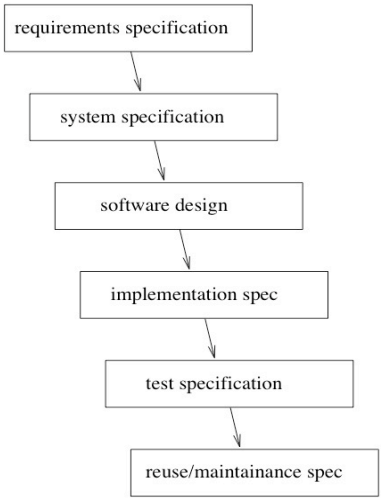
Maintenance and Reuse

- 1-800- / Helpdesk, etc.
- version control
- compatibility with older versions
- software patches (how to distribute and install)
- re-testing
- new features can cause problems (see our Boeing 777 example)

SW Development Elements

- The SW development elements (requirements, design, coding, testing, deployment, maintenance) are necessary to develop a piece of software.
- However, there are many different ways, in *which order* these steps are executed, and *how often* they are executed.

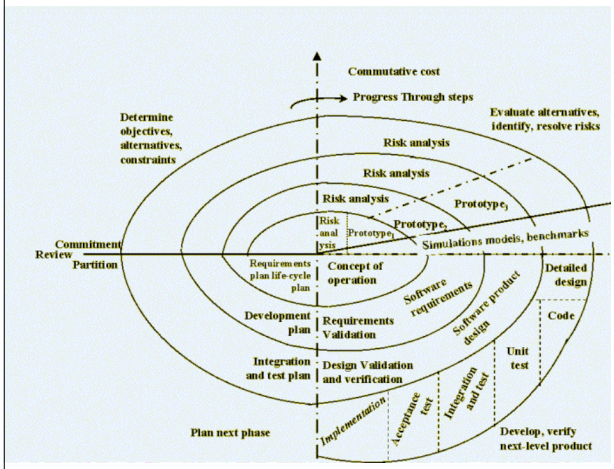
Software Life Cycle: Waterfall model



- Each stage is executed
- Previous stage needs to be finished (almost) before next stage begins

- traditional model
- often used for safety-critical software
- inflexible with respect to changes

Lifecycle: Spiral model



- each phase is run through multiple times
- early iterations with limited functionality
- prototypes assessed carefully for correctness and alignment with requirements
- in safety-critical SW: at each iteration, a risk assignment is performed

Lifecycle

- Regardless of the choice of the lifecycle model, it is always important to make sure that the software "*works as expected*". There are three major activities involved with this notion
  - verification
  - validation
  - certification

Verification, Validation, Certification

- *Verification*: "Did you implement the thing right?"
  - Typical: formal analysis, proofs, …
- *Validation*: "Did you implement the right thing?"
  - Typical: testing, testing, …
- *Certification*: "Did you do all the right steps (with respect to a *certification standard)* and demonstrated that nothing can go wrong?"
  - Typical: paperwork (plans, reports), meetings, …

Verification

- "Do you implement the thing right?"
  - "Does the design meet the requirements?"
  - "Is the code matching up with the design?"
- Verification is usually considered to be the "hardest part" of V&V
- Many different questions need to be answered for verification as
  - the "grand goal", namely a formal proof of equivalence, cannot be reached in reality (unless programs of <100 lines are reality)
  - multiple "properties" can/cannot be analyzed. Which ones to pick?

Verification

- Verification can be done on various levels of formality and rigor
  - Verification by documentation: e.g., Word-document
    - "Code lines 346-537 implement req 4.2.1.0 and req 7.2.4.1, because the variable "altitude" has a positive value.
  - Code inspection: peer reviewers go over the code line-by-line
  - automatic tools to show that important properties hold, e.g., no array-out-of-bounds, no deadlock, ...
  - Formal (logical) proof: Code -> Requirements
    - "forall L:=<date,amount>*. exists <D,A>. <D,A> \in L. A forall <D1,A1> \in L: A>= A1"

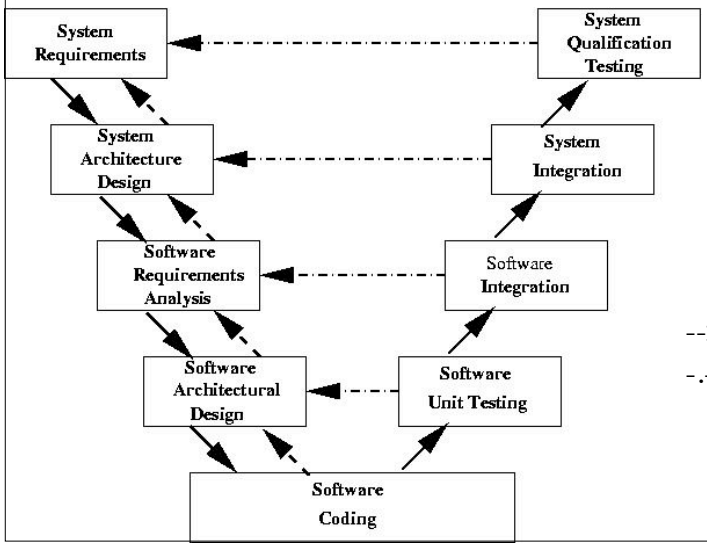
our topics

Effort GROWS

Validation

- Did you implement the right thing?
- Traditional technique: testing, i.e., exercising the code with specific inputs and check if the outputs are the expected ones
- Most software processes prescribe rigorous testing according to a metric
- ***Testing is one of the major cost driver in SW development***

The V-Shape



Tools for V&V

- In the rest of this seminar, we will look at various techniques to find and remove errors in various artifacts during the software development.
- There are many such techniques/tools out there, so we will have to focus

Tools for V&V

- we focus on
  - **Java**: although rarely used for embedded systems (yet), Java is a very popular language and finds its way more and more into safety-/mission-critical applications.
  - **open source**: commercial tools can be extremely expensive, but everybody should produce reliable software.
  - **automatic**: some V&V techniques (e.g., classical proof) requires heavy user interaction, others are fairly automatic (like a compiler). High degree of automation can lower threshold to use tools.

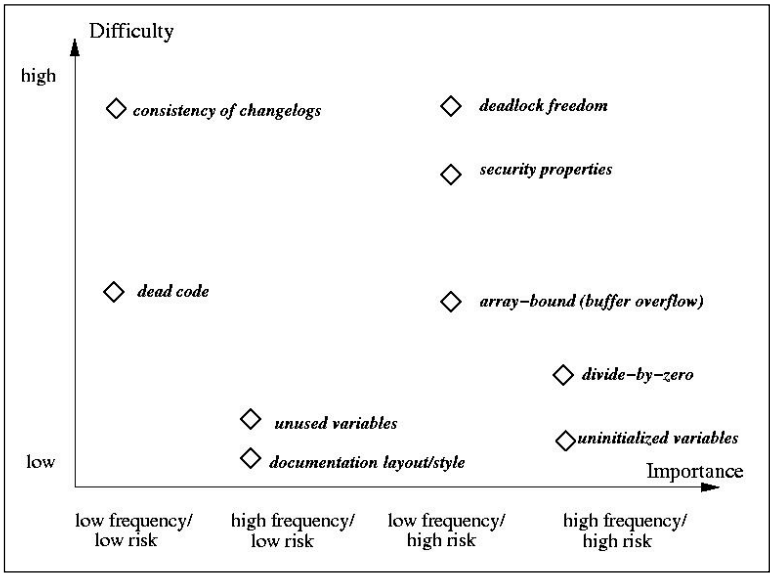
Overview

- *"Take a good look":* Analyze the code without executing it – code review, static analysis, Findbugs
- *"Execute all possibilities":* Model checking of models and code – JPF
- *"Test the code":* Testing and automatic testcase generation – SPF

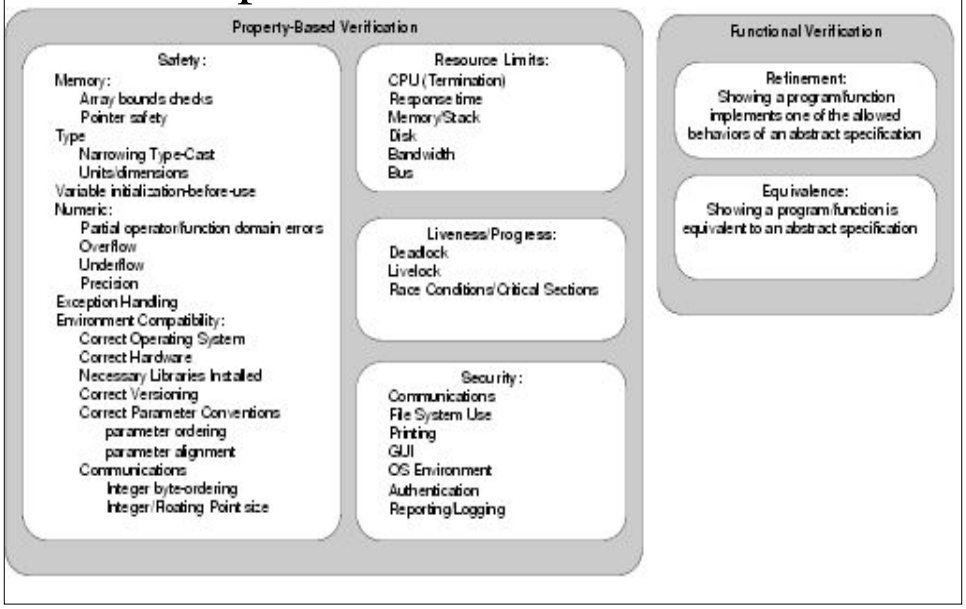
Code Review

- In most SW processes, parts (or all) of the implemented code has to undergo some kind of formal review
- A *review team* looks at the code (e.g., printout) and "tries to see if it is OK"
- There are multiple approaches to that and many "check-lists" exist
- Tools can support some aspects

Metrics for Code Review



Properties for Code Review



Coding Standards

- most programming languages are very rich and expressive
- some language constructs are hard to test and a especially error prone
  - multiple inheritance in C++
  - pointers and casting
  - dynamic memory, etc.
- There are several “coding standards” out there, which prescribe the subset of language constructs that can be used by the programmer.
- Such a list can be easily adapted and can be very helpful for all V&V purposes

Code Review and Coding Standards

- some pointers to code-review checklists and coding guidelines/standards
  - http://www.mathworks.com/industries/auto/maab.html: Standard for Simulink/Stateflow (modeling standard)
  - http://www.homeport.org/%7Eadam/review.html: Guideline for Security Code Review
  - http://www2.umassd.edu/SWPI/TechnicalReview/inspect.pdf: Question catalog for code inspections
  - http://www.chris-lott.org/resources/cstyle/Baldwin-inspect.pdf: An abbreviated C++ Code Inspection Checklist
  - http://hissa.nist.gov/publications/nistir4909/: NISTIR 4909 - SW quality assurance: documentation and review
  - http://www.jetcafe.org/jim/c-style.html: Standards and Style for Coding in ANSI C
  - http://www.alma.nrao.edu/development/computing/docs/joint/0009/2001-02-28.pdf: C coding standard for the Atacama Large Millimeter Array
  - http://www.possibility.com/Cpp/CppCodingStandard.htm: C++ Coding Standard
  - http://www.chris-lott.org/resources/cstyle/indhill-cstyle.pdf: somewhat older C coding standard
  - http://www.gnu.org/prep/standards/: GNU coding standards
  - https://www.securecoding.cert.org/confluence/display/seccode/CERT+Secure+Coding+Standards: CERT secure coding standars webpage pts. to books, wiki's etc.
  - http://membres.lycos.fr/pierret/cpp2.htm#cpp: Ellemtel C++ guide (middle of page) + other links
  - http://gee.cs.oswego.edu/dl/html/javaCodingStd.html: Draft Java Standard
  - http://java.sun.com/docs/codeconv/: Java Code conventions
  - http://geosoft.no/development/javastyle.html: Java progamming Style Guidelines

Static Analysis Tools

- tools for debugging and V&V support
- the code is never executed, but the code structure is analyzed in detail
- some tools are tailored more toward finding bugs (not complete), others guarantee the absence of certain property violations
- Lot of tools - for a list see e.g.,
  - http://en.wikipedia.org/wiki/List_of_tools_for_static_code_analysis

FindBugs

- FindBugs is an open source static analysis tool for Java
  - http://findbugs.sourceforge.net/
- There is an Eclipse Plugin
- the tool works with a set of filters
- highly flexible and extensible (user-defined filters)

Bug Categories

- **Correctness bug** Probable bug - an apparent coding mistake resulting in code that was probably not what the developer intended. We strive for a low false positive rate.
- **Bad Practice** Violations of recommended and essential coding practice. Examples include hash code and equals problems, cloneable idiom, dropped exceptions, serializable problems, and misuse of finalize. We strive to make this analysis accurate, although some groups may not care about some of the bad practices.
- **Dodgy** Code that is confusing, anomalous, or written in a way that leads itself to errors. Examples include dead local stores, switch fall through, unconfirmed casts, and redundant null check of value known to be null. More false positives accepted. In previous versions of FindBugs, this category was known as Style.

Current list of bug patterns

AM: Creates an empty jar file entry
AM: Creates an empty zip file entry
BC: Equals method should not assume anything about the type of its argumen
BC: Random object created and used only once
BIT: Check for sign of bitwise operation
CN: Class implements Cloneable but does not define or use clone method
CN: clone method does not call super.clone()
CN: Class defines clone() but doesn't implement Cloneable
Co: Abstract class defines covariant compareTo() method
Co: Covariant compareTo() method defined
DE: Method might drop exception
DE: Method might ignore exception
DMI: Don't use removeAll to clear a collection
DP: Classloaders should only be created inside doPrivileged block
DP: Method invoked that should be only be invoked inside a doPrivileged blo
Dm: Method invokes System.exit(...)
Dm: Method invokes dangerous method runFinalizersOnExit
ES: Comparison of String parameter using == or !=
ES: Comparison of String objects using == or !=
Eq: Abstract class defines covariant equals() method
Eq: Equals checks for noncompatible operand
Eq: Class defines compareTo(...) and uses Object.equals()
Eq: equals method fails for subtypes
Eq: Covariant equals() method defined

...

DEMO

- select java project (findbugs calculator/GUIConverter)
- CTRL-click
  - clear bug marks/ run bugmarks

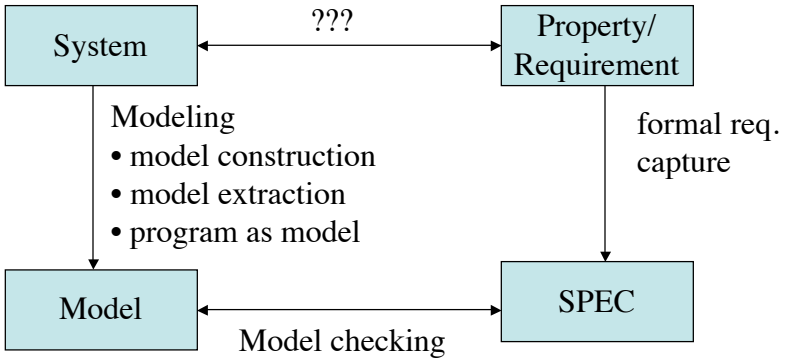
Summary: findbugs/static analysis

- tools run fully automatic
- find many weird things and possible bugs, but
  - many false alarms: tools says “could be a bug”, but code is correct waste of time
  - code is NOT executed, so many bugs cannot be found
- SA tools are useful for debugging and QA of daily builds. Can be problematic if too many false alarms occur

Model Checking

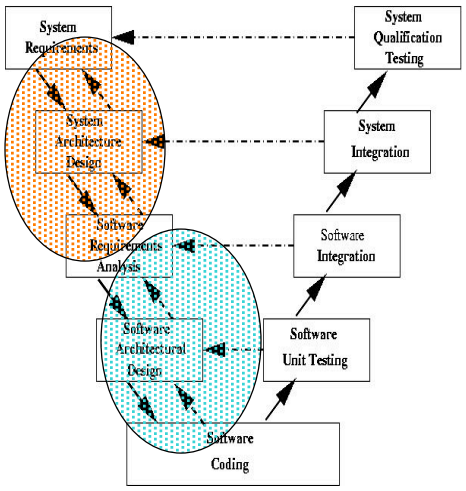
- Model Checking (in a nutshell)
  - given a (simplified) model of a system, test automatically whether this model meets a given specification or not
  - system: software or digital hardware
  - specification: formal safety requirement that should always hold or that should never occur

Models and Model Checking



- Model checking in a commutative diagram

Model Checking



- model checking can be used at different stages of the SW process and its artifacts
  - statemachines, statecharts, etc. on design level
  - actual code on the coding level

Program Model Checking

- Program model checking is concerned with checking properties, where the underlying system is a *program* (or a model thereof)
  - comes later in the SW process
  - programs usually contain much more details (good and bad for analysis)

Program Model Checking

- Use the actual program as the model
  - pros:
    - no translation into MC syntax/semantics necessary
    - easier to use
    - counter example already in the "program" world
  - cons:
    - How can we check a program?
      - explicit model checking
    - Programs contain too many details and are too large
      - abstraction
  - How to formulate properties?

Propositional and other logics

- Many specifications can be formulated in this propositional logic. What can't be done?
  - modeling with time: ***temporal logic(s)***
    - "first click A then click B";
    - "eventually the program will crash"
  - modeling with unknown parameters: *predicate logic*
    - "for all x: $x^2 > -1$",
    - "for all humans H: mortal(H)" (Socrates)
  - higher order logics, typed logics...

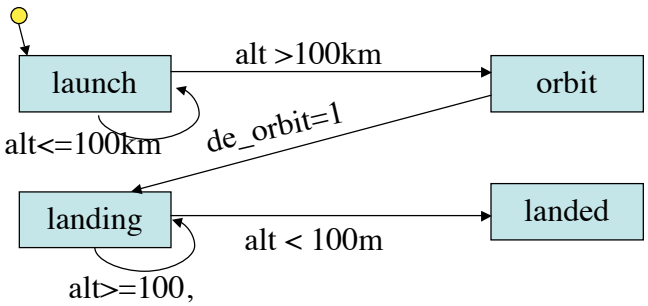
Most model checkers specialize on temporal logic

Why temporal logic?

- Most programs contain some kind of “state”, which is updated by operations
  - “file modified” is updated by “delete-char” and “save”
  - context sensitive menus
  - modes in GN&C, e.g., mode={launch,orbit,landing,landed}
    - if (mode==launch && alt >100km) mode=orbit
    - if (mode==landing && alt < 100m) mode=landed
  - network protocols: mode=init, connecting, connected, error

Why temporal logic?

- Often finite automata, state-machines, statecharts (UML), etc. are used



Why temporal logic?

- Simple programs are *deterministic* and *sequential*
- Debugging a simple program is hard
- Debugging a non-deterministic, parallel/multi-threaded can be a night-mare
  - Some errors cannot be reproduced
  - errors can show up sporadically
  - you cannot test for all cases

Example

- two different threads (e.g., in Java)

```
global int x=5;
thread 1:          thread 2:
   x = x + 2;          x = 2*x;
```

Which value of x do you get?

x=14, x=11, x=7, x=10, x=12 ?

Example cont'd

- One can get all five different values
- Why?
  - the different threads can run with different speed. So x=x+2 can be executed before or after x=2*x;
  - the assignments x=x+2 are not atomic (uninterruptable), so if after loading "x" (=5), the other process gets the CPU and sets x=10. After that, thread 1 gets the CPU again and calculates 5+2 and assigns it to x
- Bad thing: you cannot reproduce the runs; if you add "printf"s for debugging the behavior will be different again...

Obviously, the "synchronized(...)" was forgotten. Can we detect such situations?

Things to check in multithreaded code

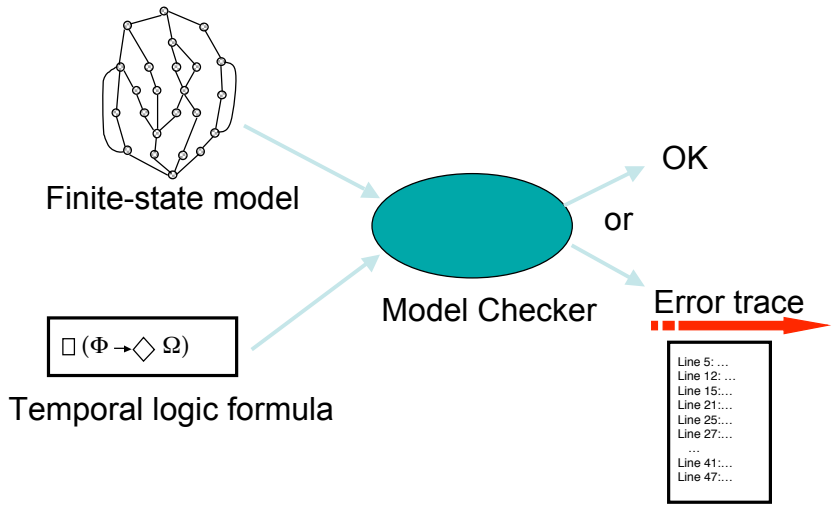
- ***deadlock***: if two (or more) threads mutually block each other, waiting for a resource that the other thread has. Execution grinds to a halt.
- ***race condition***: undesirable effects occur by having multiple threads trying to access the same resource

Temporal logic can be used to formalize these kinds of properties and check them. They actually occur: left-out protected region in Deep Space I

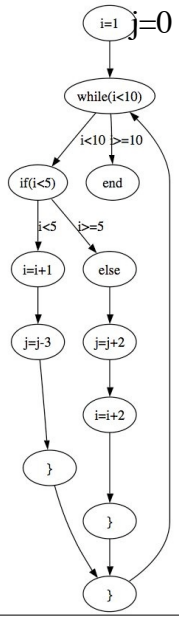
Properties

- In general, we have two kinds of properties
  - *safety properties:* Show that something (bad) cannot happen
  - *liveness properties*: Show that something (good) will happen eventually
- Violations can be shown by counter-examples. Note: for liveness, the counterexample might be of infinite length/size

Model Checking



Example



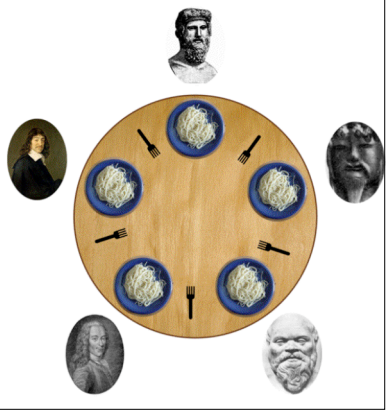
state:

<PC,i,j>

```
1  i=1;j=0;
2  while(i<10){
3     if (i < 5){
4        i=i+1;
5        j=j-3;
6       }
7   else {
8        j=j+2;
9        i=i+2;
10      }
11  }
```

Deadlocks and JPF

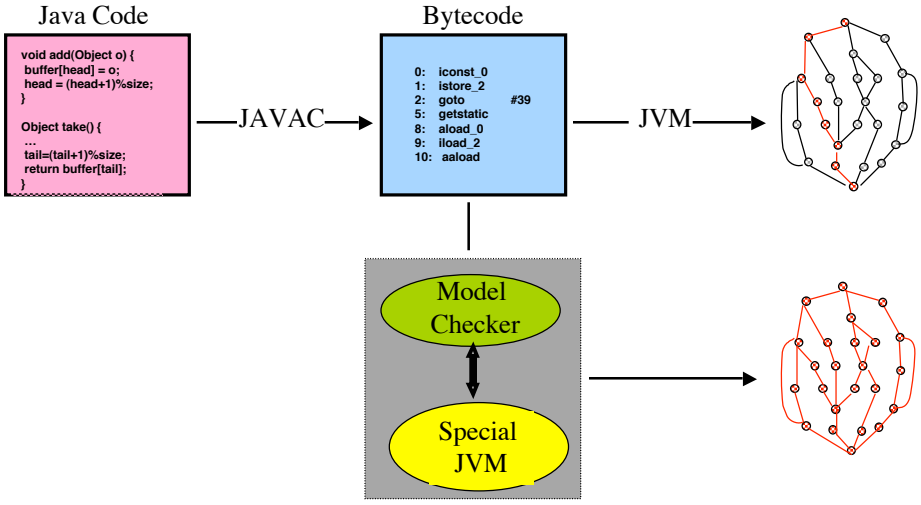
- The Dining Philosopher Example
  - philosophers do: eat, think
  - need 2 forks (left, right)
- What can happen?
  - deadlock
  - livelock



Java Pathfinder

- Java Pathfinder is an *explicit* model checker for Java, implemented in Java
- The tool was/is developed at NASA Ames
- Opensource: http://javapathfinder.sourceforge.net/
- first NASA-developed SW in open source

Java PathFinder



DEMO

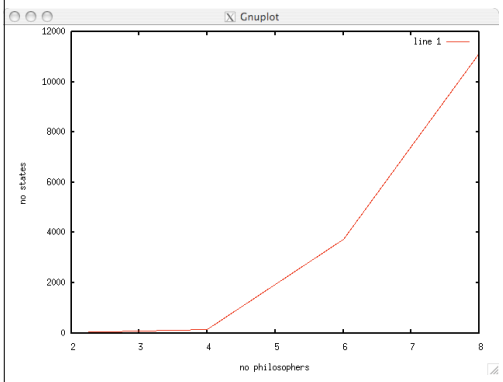
- Eclipse:
  - javapathfinder-trunk/examples/DiningPhilosophers
  - show code
  - runConfiguration: DiningPhilosophers
    - _JPF
  - fix with waiter
  - increase NUM_PHILS = 10 (DiningPhilosophers.java)
    - run with JPF

Running JPF: What does the output mean?

```
...
====================================================== results
no errors detected

====================================================== statistics
...
```

- the “no errors detected” is a *proof* that this piece of code does not violate this property.
- “error #1...” shows a trace, on how a violation of the property occurred.
- In contrast to testing, *all* possible paths through the code and process interleavings are checked.

Growing of the State Space



- Figure shows that number of states grows *exponentially* with the size of the problem.
- In general, large and complex programs get model checkers into problems (time, memory) *quickly*

JPF: Underlying Principles

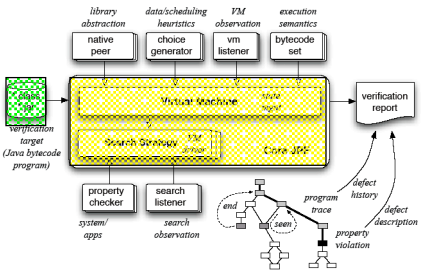
- An explicit program model checker consists of:
  - searching the state-transition space of the program
  - checking for properties as algorithm goes along
  - storing states for
    - recognizing states "already visited"
    - for backtracking
  - reading in
    - program
    - properties

JPF: Basic Idea

- implement a special-purpose JVM
  - add techniques for state compression
  - add search algorithm and search control
  - add property checker
  - and you are ready to go

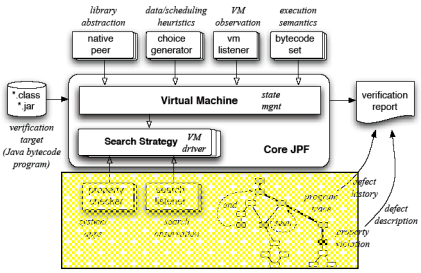
Well, it's not that easy...

JPF Architecture



- .class: input to JVM
- configuration file (not shown)
- Virtual machine: custom-made VM which includes
  - lots of interfaces
  - state management for storing and comparing states ("seen"/"visited")
- Search Strategy: do all kinds of search

JPF Architecture



- Property checkers: check properties when moving along
- contains default property checkers: deadlock, race condition, uncaught exception,...
- search listener: provide information about the search
- everything is implemented in an OO fashion with “listeners”, “factories”, ...

ChoiceGenerator

- Our example was extremely primitive, as all the variables were set to a constant value.
- What can be done if variable values are set from the outside (e.g., input to program, environment)?
- The ChoiceGenerator is an OO interface to provide enumeration of values for a variable during model checking
  - import gov.nasa.jpf.jvm.Verify;
  - boolean b = Verify.getBoolean();
  - int arg = Verify.getInt(-10,10);
- These “nondeterministic variable settings” are used to automatically generate the appropriate values during the MC search

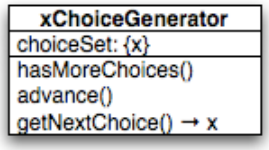
Example with ChoiceGenerator

```
import gov.nasa.jpf.jvm.Verify;

public class example2 {
example2(){}

int dodiv(int x, int y){
    int z;
    z=x/y;
    return z;
    }

public static void main(String[] args) {
    int res;
    example2 ex = new example2();
    int arg2 = Verify.getInt(-10,10);
    res = ex.dodiv(10,arg2);
    System.out.println("result: "+res);
    }
}
```

- This set-up causes JPF to go through all values of arg2 from -10 to 10.
- Should hit a div-by-0 exception somewhere...

Configurable Heuristic Choice Models

- take choice generator
  - double alt = Verify.getDouble("altitude");
- configure, how selected values for the variable "alt" are to be generated.
- This de-couples the value generation and the search (heuristics)
- user can easily extend this mechanism



Properties

- There are different kinds of properties
  - safety properties (something bad cannot happen)
  - liveness properties (something good will eventually happen)
  - fairness properties (something good happens infinitely often)
  - temporal properties

Properties

- generic properties
  - deadlock, no race condition
- language-specific
  - no array-out-of-bounds
  - no null-pointer dereference
- application specific
  - pitch ration should be between -1 and 1
  - The parachute can only jettisoned after ...

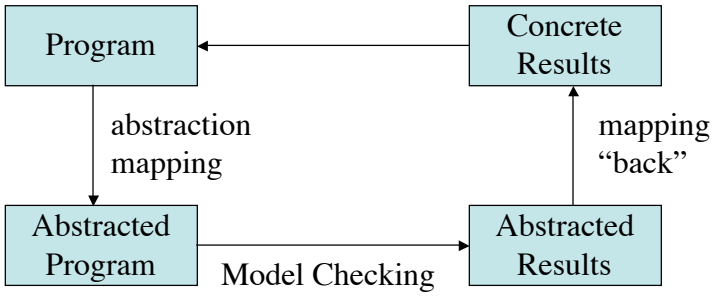
How to specify properties

- default properties (in JPF): deadlock, assertion violation, uncaught exception, race condition
- instrumentation (adding property oracles): insert code that says whether a property holds at a given state during model checking
  - this fits neatly with traditional "assert" programming and programming by contract (i.e., preconditions, postconditions, invariants).

Abstraction

- In Model Checking things usually break if
  - there are an infinite (or large) set of choices
    - there is no choice-generator for "double"
  - the program contains many variables and details
- The resulting search space is infinite (or at least too large)
- With *Abstraction*, we throw away some information about the program without (hopefully!) not changing the validity of properties
- e.g.: double altitude
  - can be abstracted into "altitudes larger than 30,000ft" and "smaller".

Abstraction - the Process



- commutative diagram

Abstraction

- control abstraction
  - remove program parts, which are not of interest for the current analysis
  - Example:
    - ...{ var1=5; var2=var2+1;} ==> act_1
  - "program slicing"
- data abstraction
  - replace large domains with small ones
  - see previous slide

Property Preservation

- An abstraction is *weakly preserving* if a set of properties, which are true in the abstracted system, is also true in the concrete system
- An abstraction is *strongly preserving* if a set of properties has the same truth value in the abstracted system and the control system
- An abstraction is *error preserving* if a set of properties, which are false in the abstracted system is also false in the conrete system

Property Preservation

- W*eakly preserving* properties are nice, because they can substantially reduce the space. However, violations of properties found in the abstracted systems need not correspond to property violations in the concrete system
  - this is called "false" alarms
- An *error preserving* abstraction is useful for disproving properties. Property violations in the abstracted system correspond to property violations in the concrete system.
  - this comes in handy for *debugging*
  - bugs (as violations of properties) can be found that way
  - however, such an abstraction does not guarantee that a property holds

Example

Program:

```
if (-100 < x && x < 100){
     y=1/x; // (*)
      }
elseif (x <= -100)
     action_b;
else
     action_c
```

- x={0,1000} is error-preserving wrt. division-by-zero
  - bad statement (*) in abstracted code is also bad in original code
- x={-1000,1000} is weakly preserving wrt. dead code
  - statement (*) cannot be reached in abstraction (violation of dead-code free property). However, orginal code is dead-code free

Abstract Interpretation

- Abstract interpretation is a framework for data abstractions, which are weakly preserving with respect to safety properties
- An abstract interpretation consists of
  - finite domain of abstract values
  - abstraction function
  - collection of primitive operators

Example: Sign abstraction

- A popular abstraction is the sign abstraction: instead of using the actual value of a variable, we only consider its sign.
- Usually, ZERO, NEG, POS are used
- we need to define
  - abstraction function
  - primitive operators: e.g., “+” and “<“

Sign Abstraction

“+”	ZERO	POS	NEG
ZERO	ZERO	POS	NEG
POS	POS	POS	{ZERO,NEG,POS}
NEG	NEG	{ZERO,NEG,POS}	NEG

Sign Abstraction: “<“

“<”	ZERO	POS	NEG
ZERO	FALSE	TRUE	FALSE
POS	FALSE	{FALSE, TRUE}	FALSE
NEG	TRUE	TRUE	{FALSE, TRUE}

Abstraction of Program

```
foo(){
int a = -10;
int b = 0;
while (a < 0){
  b = b + a;
  a = a + 1;
  }
assert(a==0);
```

```
foo(){
int a = Sign.map(-10);
int b = Sign.ZERO;
while (Sign.lt(a, ZERO)){
  b = Sign.add(b,a);
  a = Sign.add(a,Sign.map(1));
  }
assert(Sign.isequal(a,ZERO));
```

- program is actually *modified*

Useful abstractions

- range abstraction: [lt_l, l, l+1,...,u-1,i,gt_u]
- sign abstraction
- even/odd
- modulo abstraction: merge all values with the same remainder into one value. even-odd is a modulo-2 abstraction
- point abstraction: abstract all values into "unknown"

JPF as Framework

- The JPF architecture is very flexible and thus can be (and has been) used for multiple purposes/extensions
  - checking numerical properties (overflow, roundoff, ...): extensions/numeric
  - symbolic testcase generation: extensions/symbc
  - swing GUI model-checking: extensions/ui
  - translation of temporal logic into automata for property checking (this is LTL (linear time temporal logic)): extensions/LTL2Buchi
  - checking of statecharts: extensions/statecharts
  - compositional verification: extensions/cv
  - ...

Controlling JPF

- There are many knobs to turn
- Abstraction
- ChoiceGenerator
- Search: DFS, BFS, heuristic search
- ignoring parts of the program
- compressing transitions

Summary: JPF

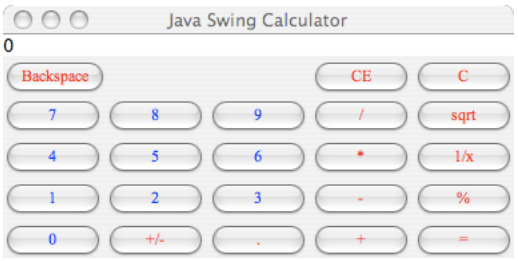
- JPF is a powerful search/checking framework for Java programs
- Easy to use for checking built-in properties, special-purpose ones require programming
- Many opensource extensions to JPF
- large programs/large variable domains require abstraction – this is where creativity comes in

Model Checking GUIs

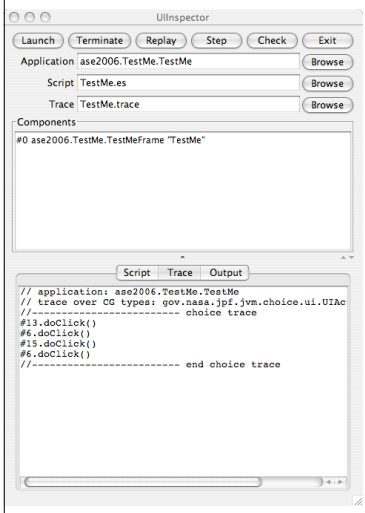
- Graphical user interfaces are used in many applications
- Often implemented in Java (e.g., Swing, awt)
- Often they hide a fairly complex control structure
- GUIs are traditionally very hard to test
  - requires “point and click actions”
  - lots of possibilities and sometime longer sequences are necessary to trigger a failure

Example

- I found an example of a "Calculator" in a Java tutorial on the Web
  - simple GUI, no background libraries
  - still enough buttons + operations



UIInspector: an extension to JPF



- UI to run, step the GUI under analysis and to run JPF
- Browser to identify individual components of the GUI
- Script to define (nondeterministic) sequences of GUI actions
- traces can be saved and loaded
- Stepwise execution or replay of saved traces
- UIInspector part of JPF distribution

DEMO: GUI-Model Checking

- JPF_GUI_Calculator
- run with configuration “Calculator”
- run with conf “Calculator_JPF”
  - show UI elements and numbers
  - show script file
  - run
  - demonstrate trace

GUIInspector script files

- simple language to specify non-deterministic sequence of GUI actions

```
REPEAT 2 {
  ANY { #13.doClick, #14.doClick, #15.doClick }
  #6.doClick
}
```

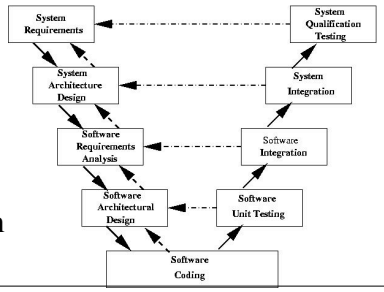
•we repeat 2 time a sequence where we click any of the 3 buttons and the click “Cycle”
•The ID numbers (#13, etc) can be obtained using the content browser.
•Events can be added to script using the “add target” (RHS)

Limits of the GUI MCing

- if external libraries are used, they should be “abstracted”, i.e., they should be ignored during search (unless you want to find a bug in the library)
- generic input (e.g., editor fields) must be handled separately (e.g., with ChoiceGenerator) to produce reasonable multiple values
- graphical elements (e.g., drag, draw,...) would need some abstractions – one does usually not want to enumerate all (x,y) positions of the cursor

Testing

- Testing is one of the most important activity during the software lifecycle.
- With testing, it is assured that the implemented software performs according to the specification
- The V-shape



--> Verification
-.-.-> Validation

The Testing Hierarchy

- **unit test**: small SW component: your desktop
- **SW system test**: on a big workstation; environment and hardware are simulated
- Test on target platform "**baby bed**": flight computer ("black box") is attached to workstation. SW runs on black box, workstation simulates environment
- **HILS** (Hardware in the loop): flight computer runs SW, is attached to aircraft. An external workstation simulates the environment and produces simulated inputs; outputs directly drive the aircraft.

The Testing Hierarchy

- The costs for testing and the testing time grows dramatically with increasing level of fidelity (HILS: >$10k per hour)
- Time & effort to fix bugs and retest grows dramatically with increasing level of fidelity.

Thus it is essential that to do as many tests and to find as many bugs as early as possible.

Our testing with Model Checking is for unit testing

Black-box vs. White-box

- In *black-box* testing, the piece of SW under test is exercised using its inputs and outputs only. No aspects of its internal working is considered. No knowledge about the implementation is / should be available to the tester.
- In *white-box* testing, the SW is opened up and details of the SW implementation and behavior can be looked at and analyzed.

Functional vs. structural testing

- Functional testing: “Do I get the right output for the given input?”; “Is this thing working?”
- Structural testing: “Did I test every statement of the code?”
  - The aim of structural testing is to provide a given *coverage* of the code
  - This of course requires that the code is visible and accessible

safety-critical code is often required to be tested according to specific code coverage metrics

Code Coverage Metrics

- *Statement coverage*: Each statement of the source code needs to be executed
- *Decision coverage* (Branch coverage): Each conditional statement (if, switch, for, while) must be executed in true and false
- *Condition coverage*: each boolean subexpression must be evaluated to true and false
- *Path coverage*: Every possible route through a piece of code must be executed
- *MC/DC* (Modified Condition/Decision Coverage): similar to Path cov, but reduces number of test-cases. Important for Aerospace software

Example

```
if( A )
     do_1; x=1;
else
     do_2;x=2;
...
if (B)
     do_3;x=x-1;
else
     do_4;
...
if (C)
     do_5;
else
     do_6; y=5/x;
...
```

- 100% statement coverage requires 2 test cases:
  - A=B=C=true
  - A=B=C=false
- The branch coverage requires 6 test cases
  - A=true/false;B=C=true
  - B=true/false;A=C=true
  - C=true/false;A=B=true
- Path coverage requires all 8 test cases

Coverage and practice

- The number of required test cases increases tremendously with the metric.
- Condition covagerage and Path Coverage are the worst: they require an exponential number of test cases
- In practice, no one uses (can use) full condition or path coverage.

MC/DC

- MC/DC (Modified Condition/Decision Coverage)
- Every entry/exit point has been invoked
- Every condition in a decision has taken all possible outcomes
- Each condition has been shown to affect that decision outcome independently: just vary one decision and keep the others fixed

Why is MC/DC important?

- MC/DC is a powerful coverage metric, but generates much less test cases than condition or path coverage
- The standard DO-178B requires all that all level-A critical software in commercial Aircraft been tested according to DO-178B
- A lot of other safety-critical standards have adopted MC/DC.

Automatic Testcase Generation with JPF

- A model checker can generate testcases
- Put an “assert(false)” in the code and the MC will tell you how to get there (trace)
- This method does not produce input values for the tests

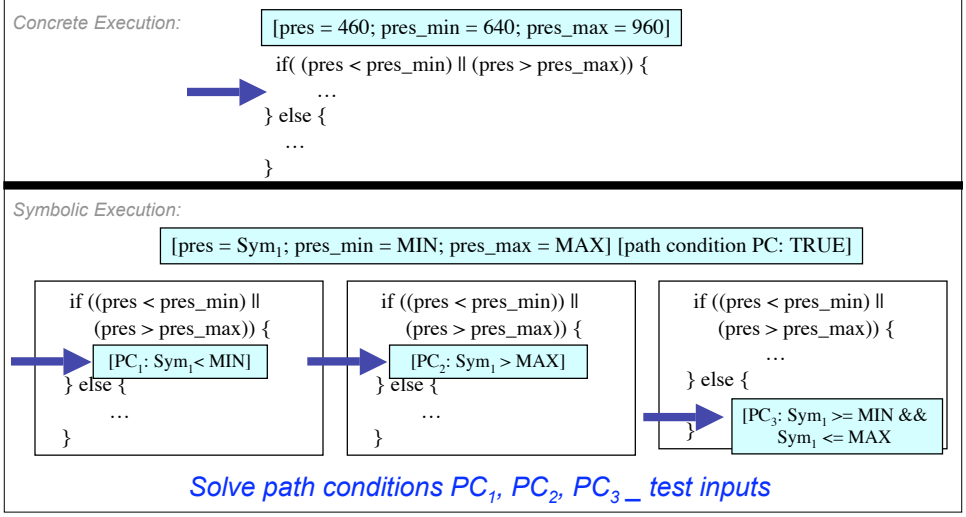
SPF: Symbolic PathFinder

- adds symbolic execution (i.e., calculation with symbolic variables to the JPF model-checking mechanism
- adds algorithms for solving symbolic equations/in-equalities
- the combined system can automatically generate test cases for a multitude of coverage metrics

Symbolic Execution

Systematic Path Exploration

Generation and Solving of Numeric Constraints



Solve path conditions PC_1, PC_2, PC_3 _ test inputs

DEMO: Testcase Generation

```
import gov.nasa.jpf.symbc.Debug;


public class MyClass1 {

public int myMethod(int x, int y) {
   int z = x + y;

   if ((z > 0) && (x >0 || y < 0)) {
      z = 1;
   } else {
      z = z - x;
   }
   z = x * z;
   return z;
}

// The test driver
public static void main(String[] args) {
MyClass1 mc = new MyClass1();
int x = mc.myMethod(1, 2);
Debug.printPC("\nMyClass1.myMethod Path Condition: ");
}
}
```

extension/symbc/examples/MyClass1.java

configuration: myClass1_symbolic

- the test cases are
  - –TC1: y=1 x=0
  - –TC2: y=0 x=1
  - –TC3: y=-10000 x=-10000

Test drivers, etc

- A test driver (test harness) is a program that wraps around the program under test (P) and executes the test cases/sequences.
- The output for each test is compared to the desired output "oracle" and set to PASS/FAIL
  - careful with floats: 1 != 0.9999997
    - so use fabs(oracle-out) < 1e-5
  - might require customized comparisons
- Instrumentation of code to measure coverage
- Generation of Test Report
  - % fail/pass, coverage, version/ history ...
- Connection to Data Base

Test Drivers/Harnesses

- There is a lot of tools ($$$ and open source, e.g. JUnit)
- usually one has to pay for nice (HTML) output
- MC/DC coverage analysis is usually $$$
- careful set-up of test plan is important
  - e.g., functional tests don't vary with code version, but code coverage tests do.

Limits of JPF testcase generation

- limited support for arithmetic functions: constraint solving is very similar to equation solving. Some equations cannot be solved symbolically or it is extremely hard. So no decision procedures exist (yet).
- currently limited support for other data structures

Summary

- There are V&V opportunities throughout the entire SW lifecycle
- V&V: “Do it early and do it often...” AND “Do it”
  - the use of V&V tools must be smoothly integrated into the SW process
  - a V&V plan should be set up before the software is tested
  - Don’t separate “developers” (the gurus) and “testers” (low-pay entry-level people) (but: there is IV&V)

Summary

- There are many open source V&V tools for Java, so use them
  - most are not “push-button”
  - often require only one-time setup
  - education/training is important
  - can be integrated into daily builds etc.
- The (ongoing) software crisis cannot be solved with these tools, but they can help to make software more reliable – not just safety-critical software

Summary