TECHNISCHE UNIVERSITÄT DRESDEN

Faculty of Computer Science Institute for System Architecture, Operating Systems Group

Bugs and what can be done about them...

Björn Döbel

Dresden, 2009-02-03

Outline

- What are bugs?
- Where do they come from?
- What are the special challenges related to systems software?
- Tour of the developer's armory

What are bugs? (IEEE 729)

- **Error:** some (missing) action in a program's code that makes the program misbehave
- **Fault:** corrupt program state because of an error
- **Failure:** User-visible misbehavior of the program because of a fault
- **Bug:** colloquial, most often means fault

Bug Classification

- **Memory/Resource leak** – forget to free a resource after use
- **Dangling pointers** – use pointer after free
- **Buffer overrun** – overwriting a statically allocated buffer
- **Race condition** – multiple threads compete for access to the same resource
- **Deadlock** – applications compete for multiple resources in different order
- **Timing expectations** that don't hold (e.g., because of multithreaded / SMP systems)
- **Transient errors** - errors that may go away without program intervention (e.g., hard disk is full)
- ...

Bug Classification – Another try

- **Bohrbugs:** bugs that are easy to reproduce
- **Heisenbugs:** bugs that go away when debugging
- **Mandelbugs:** the resulting fault seems chaotic and non-deterministic
- **Schrödingbugs:** bugs with a cause so complex that the developer doesn't fully understand it
- **Aging-bugs:** bugs that manifest only after very long execution times

Where do Bugs Come From?

- Operator errors
  - Largest error cause in large-scale systems
  - OS level: expect users to misuse system call
- Hardware failure
  - Especially important in systems SW
  - Device drivers
- Software failure
  - Average programmers write average software!

One Problem: Code Complexity

- Software complexity approaching human brain's capacity of understanding
- Complexity measures:
  - **S**ource **L**ines **o**f **C**ode
  - **Halstead Complexity**
    - Operands (variables, constants)
    - Operators (keywords, operators)
    - Relate to total number of used operators and operands

Code Complexity Measures

- **Cyclomatic Complexity** (McCabe)
  - Based on application's control flow graph
  - *M := number of branches in CFG + 1*
    - minimum of possible control flow paths
    - maximum of necessary test cases to cover all paths at least once
- **Function points**
  - Assign *"function point value"* to each function and datastructure of system
  - Based on experience
- Loads of others

Special Problems With Systems Software

- IDE / debugger integration:
  - No simple compile – run – breakpoint cycle
    - But: Linux does have KGDB
  - Can't just run an OS in a debugger
  - But: HW debugging facilities
    - single-stepping of (machine) instructions
    - HW performance counters
  - Stack traces, core dumps
  - printf() debugging

- *OS developers lack understanding of underlying HW. HW developers lack understanding of OS requirements.*

Breakpoint - What can we do?

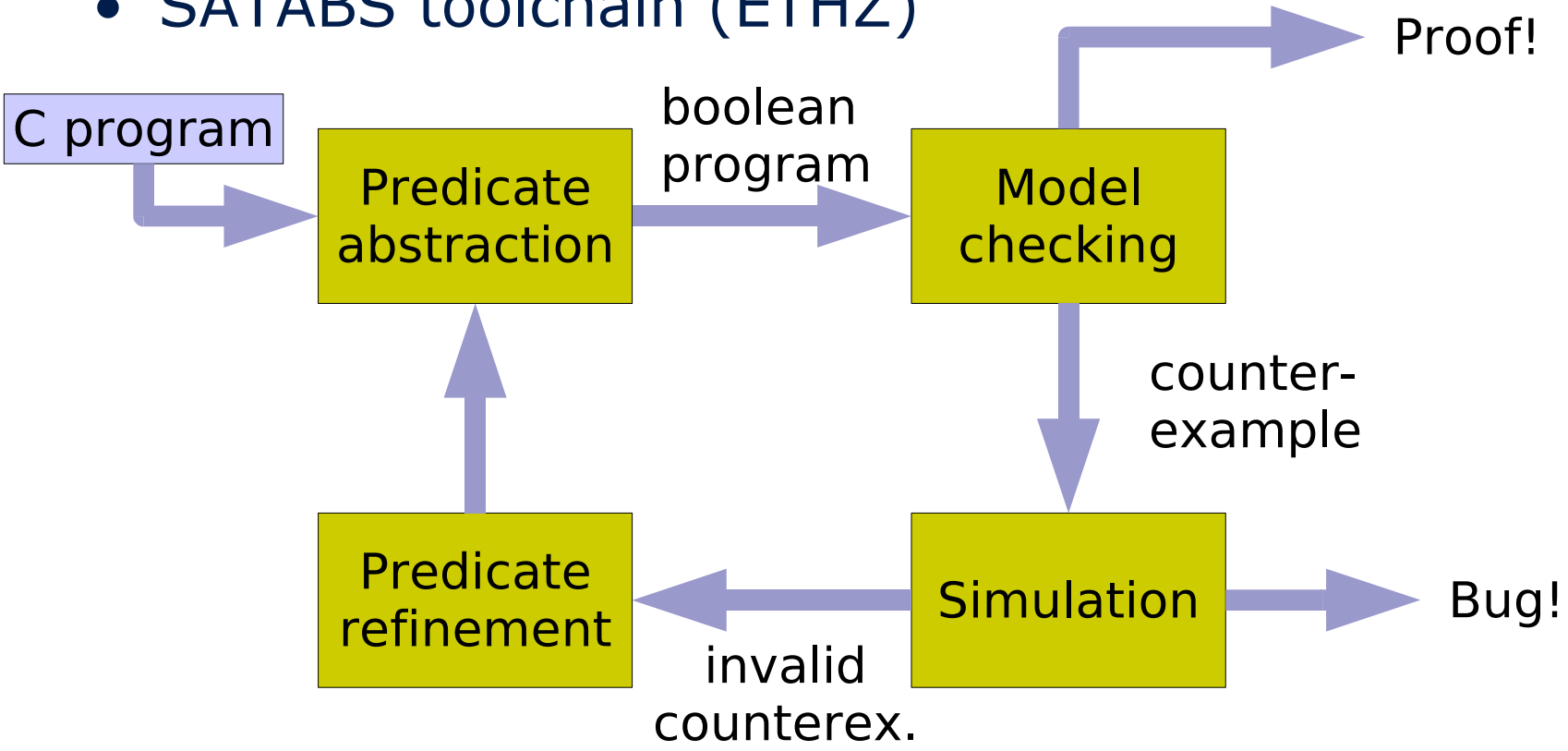
- Verification
- Static analysis
- Dynamic analysis
- Testing
- Use of
  - Careful programming
  - Language and runtime environments
  - Simulation / emulation / virtualization

Verification

- Goal: provide a mathematical proof that a program adheres a certain property φ.
- Model-based approach
  - Generate application model, e.g. state machine
  - Prove that φ is an invariant upon the program.
  - Works well for verifying protocols
- Typical techniques
  - interactive theorem proving
  - **Model checking**
  - **static / dynamic program analysis**

Model Checking: CEGAR / SATABS

- **C**ounter**e**xample **G**uided **A**bstraction **R**efinement
- SATABS toolchain (ETHZ)



Once upon a time... - a war story

- L4Linux CLI implementation with tamer thread
- After some hours of `wget` L4Linux got blocked
  - Linux kernel was waiting for message from tamer
  - tamer was ready to receive
- Manually debugging did not lead to success.
- Manually implemented system model in Promela
  - language for the SPIN model checker
  - 2 days for translating C implementation
  - more time for correctly specifying the bug's criteria
  - model checking found the bug

Once upon a time... - a war story (2)

- Modified Promela model
  - tested solution ideas
    - 2 of them were soon shown to be erroneous, too
  - finally found a working solution (checked a tree of depth ~200,000)
- Conclusion
  - 4 OS staff members at least partially involved
  - needed to learn new language, new tool
  - Time-consuming translation phase finally paid off!
  - Additional outcome: runtime checker for bug criteria

Model Checking

- The good:
  - Active area of research, many tools.
- The bad:
  - Often need to generate model manually
  - Need to search complete state space for counterexample <-> state space explosion
- The ugly:
  - Does model describe program in sufficient detail?
  - Correctness of program-to-model transformation?

Static Analysis

- Model Checking / Theorem proving do not (yet?) scale to large-scale systems.
- Many errors can be found faster using automated code-parsing tools and heuristics.
  - generate model during parsing
  - check adherence to external templates
- Simple forms:
  - compiler warnings
  - compile-time type checking of C++ data types

Static Analysis (2)

- Trade completeness / soundness of formal methods for scalability and performance.
  - Can lead to
    - **false positives** – find a bug where there is none
    - **false negatives** – miss an existing bug
- Many commercial and open source tools
  - wide and varying range of features

Lint

- 1979
- Mother of quite some static checking tools
  - xmllint
  - htmllint
  - jlint
  - SPLint
  - ...
- Flag use of unsafe constructs in C code
  - e.g.: not checking return value of a function

Flawfinder and Rats

- Check C programs for use of well-known insecure functions
  - sprintf() instead of snprintf()
  - strcpy() instead of strncpy()
  - …
- List potential errors by severity
- Provide advice to correct code
- Basically regular expression matching

Two Important Concepts

- **List errors by severity**
- **Source code annotations**
  - Specially formatted comments inside code for giving hints to static checkers
    - `/* @notnull@ */ int *foo` -> "I really know that this pointer is never going to be NULL, so shut the **** up complaining about me not checking it!"
  - Problem
    - Someone needs to force programmers to write annotations.
    - Formal methods people don't know what to prove.

SPLint

- **S**ecure **P**rogramming **Lint**
- Powerful annotation language
- Checks
  - NULL pointer dereferences
  - Buffer overruns
  - Use-before-check errors
  - Use-after-free errors
  - Returning stack references
  - ...
- **Demo**

Other Use Cases

- Support for program comprehension
  - Doxygen, JavaDoc
  - LXR
  - CScope/KScope
- Data flow analysis
  - Does potentially malicious (tainted) input end up in (untainted) memory locations that trusted code depends on?

Dynamic Analysis

- Static analysis cannot know about environmental conditions at runtime
  - need to make conservative assumptions
  - may lead to false positives
- Dynamic analysis approach:
  - Monitor application at runtime
  - Only inspects execution paths that are really used.
- Problems
  - Instrumentation overhead
  - When are you done?

Dynamic Analysis (2)

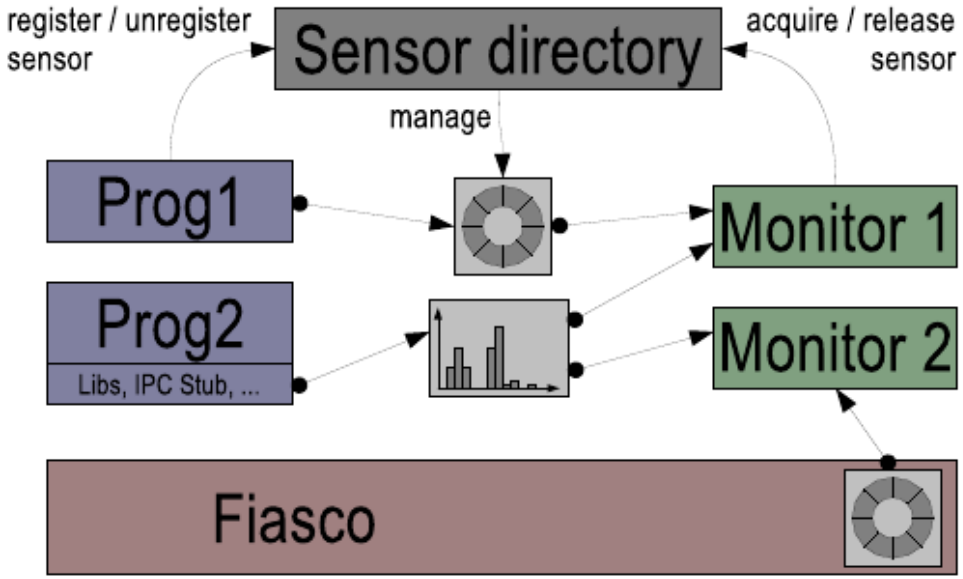
- Can also check timeliness constraints
  - But: take results with care – instrumentation overhead
- How do we instrument applications?
  - Manually
    - L4/Ferret
  - Runtime mechanisms
    - DTrace, Linux Kernel Markers
    - Linux kProbes
  - Binary translation
    - Valgrind

Manual Instrumentation - L4/Ferret

- *Aim:* Runtime monitoring framework for real-time systems with low instrumentation overhead
- Shared-memory ring buffer for events
  - Instrumented program posts events (low overhead)
  - Low-priority monitor collects events without interfering with application execution
- Sensor types
  - Scalar – simple counters
  - Histogram – distributions
  - List – arbitrary events

L4/Ferret (2)

- Manual instrumentation
  - Dice extension for instrumenting L4 IPC code
  - Use of Aspect-Oriented Programming
  - Can be coupled with other mechanisms, e.g., kProbes



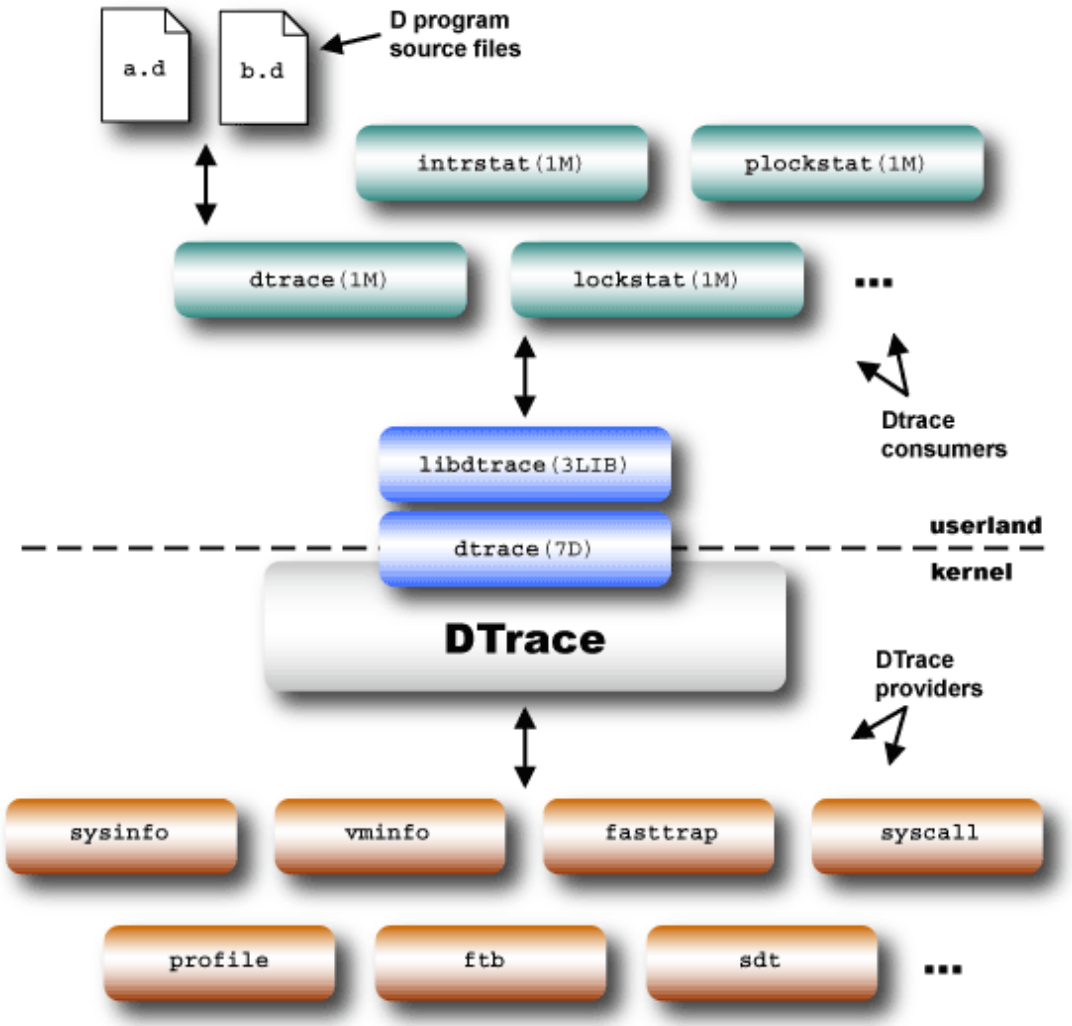
Runtime instrumentations with Trapping

- Linux kProbes
  - Linux kernel modules
  - Patch instructions with trap instruction (INT3)
  - When hit, debug interrupt occurs
    - Inspect (and store) system state before instruction
    - Use single-stepping to execute instruction
    - Inspect (and restore) system state after instruction
- SystemTap
  - Write probes in a scripting language
  - Automatically generate kProbe module

Runtime Instrumentation Without Traps

- Trapping has disadvantages:
  - Using traps leads to overhead
  - x86 is evil: varying opcode lengths
  - Cannot insert arbitrary instrumentation
- DTrace, Linux kernel markers
  - Identify interesting locations in the kernel
  - Insert bunch of NOOP statements (instrumentation markers) at compile time
  - Write kernel modules to overwrite NOOPs with instrumentation code at runtime

DTrace Architecture



Instrumentation Problems

- Problems:
  - Lack of source code access for manual instrumentation
  - Lack of knowledge about system internals
  - Markers: need to know interesting instrumentation locations beforehand
- Solutions:
  - Libraries for common instrumentation tasks (SystemTap)
  - Dynamic binary instrumentation (DBI) frameworks

Dynamic Binary Instrumentation

- Annotated binary code (DynamoRIO, Pin)
- Binary-to-binary translation (Valgrind)
  - binary -> intermediate language
    -> instrumented binary

```
0x24F27C:  addl %ebx,%eax
 4: ------ IMark(0x24F27C, 2) ------
 5: PUT(60) = 0x24F27C:I32          # put %eip
 6: t3 = GET:I32(0)                 # get %eax
 7: t2 = GET:I32(12)                # get %ebx
 8: t1 = Add32(t3,t2)               # addl
 9: PUT(32) = 0x3:I32               # put eflags val1
10: PUT(36) = t3                    # put eflags val2
11: PUT(40) = t2                    # put eflags val3
12: PUT(44) = 0x0:I32               # put eflags val4
13: PUT(0) = t1                     # put %eax
```

Valgrind

- Core
  - Application loader (get rid of dynamic linker)
  - JIT for basic blocks
  - Dedicated signal handling
  - System call wrappers to issue events upon kernel accesses to user memory, registers, ...
- Tool plugins
  - Perform instrumentation on intermediate language
  - Replace/wrap certain functions with own implementation

Valgrind Tools

- Valgrind core ~ 170.000 LOC
- Memcheck
  - Memory leak checker, ~10.000 LOC
- Cachegrind
  - Cache profiler, ~2.400 LOC
- Massif
  - Heap profiler, ~1.800 LOC

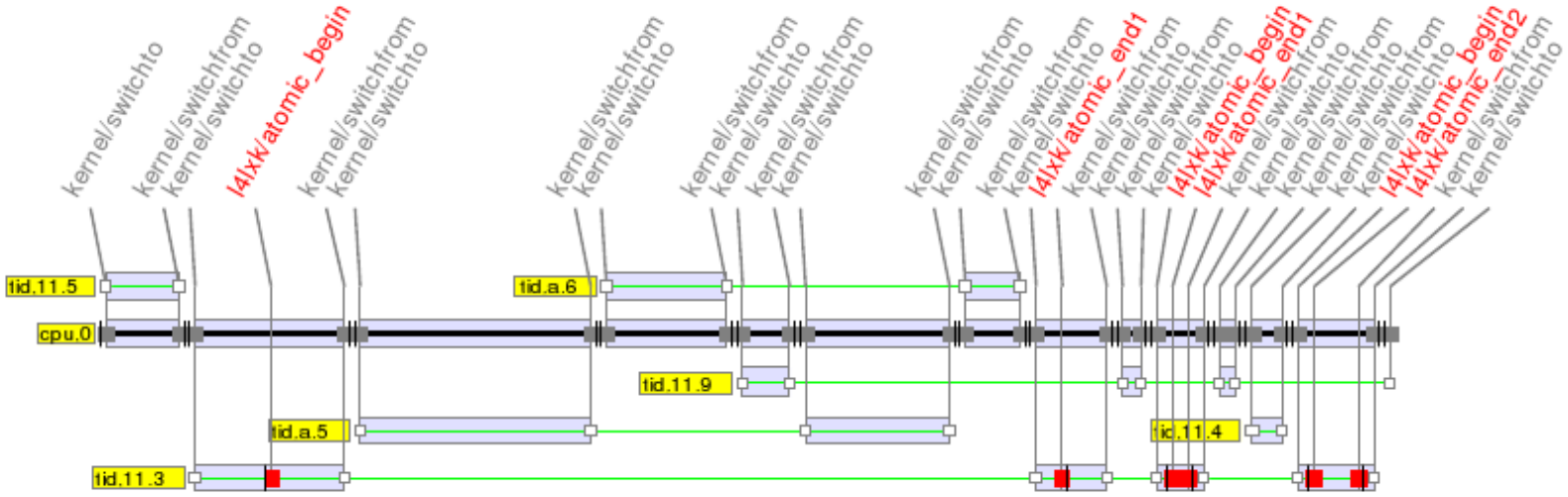
- **Demo**

Evaluation

- Dynamic instrumentation is cool, but someone needs to handle the results:
  - Online evaluation: Perform runtime monitoring to check that the system behaves correctly
    - tools need to be fast / low overhead
  - Offline evaluation: Perform instrumentation to understand system behavior
    - can use more heavyweight analysis tools
    - Magpie

Magpie

- Visualization for obtained events
  - header used for basic visualization, resource accounting
  - additional data for performing more thorough analysis



Testing

- Bugfixing cost becomes more expensive, the later the bug is discovered.
- Don't misunderstand the waterfall model!
  - Testing phase there means integration/usability testing.
- Proper testing right from the start can help to discover bugs early.
- Only finds bugs, no proof of correctness!

Test Types – Unit tests

- Unit tests
  - test software units (==functions) one at a time
  - external dependencies replaced by stubs (mockups)
- Blackbox testing
  - test for behavior
- Whitebox testing
  - test control flow paths
  - achieve certain code coverage
    - function-, statement-, condition-, path-, exit-coverage

Input Generation

- Good/bad input
- Boundary values
- Random data
- Zero / NULL
- Automation?
  - at least generate test skeletons automatically
  - static analysis can generate test cases
    - special values exist for certain types
    - annotations to define ranges of good input

Unit Test Frameworks

- xUnit
  - Kent Beck for Smalltalk
  - now available for most major programming languages
- **Test fixture** := predefined state for tests
- **Test suite** := set of tests running in the same fixture
- **assertions** to verify input, output, return values, ...
- available for many programming languages
  - CUnit also available for L4

Design by Contract

- Trivia: Is checking return values defensive programming?
- **Design by contract** – functions have
  - Preconditions -> guaranteed by caller
  - Postconditions -> guaranteed by callee
  - Invariants -> guaranteed by both
- Use assertions to check pre- and postconditions
  - overhead?
  - can serve as kind of annotation for static analysis tools

What else can we do?

- Virtual machines (QEmu, VMWare, Vbox, ...)
  - simulate HW which otherwise isn't available
  - but: be aware that HW behavior doesn't necessarily match...
- Safe programming languages (Java, C#, ...)
  - builtin garbage collection
  - runtime / compile time type checking
  - not necessarily a bad idea for systems programming:
    - Singularity mostly written in a C# dialect
    - Melange (network stacks in OCaml)

Where to go from here?

- Prof. Härtig / OS Chair – build real systems software
  - Microkernel Construction (summer)
  - Real-Time Systems (winter)
  - Distributed Operating Systems (summer)
  - Systems Paper Reading Group (weekly)
  - “Beleg” / Diploma thesis @ OS (always)

Where to go from here?

- Prof. Fetzer
  - Systems Engineering 1 & 2
  - Software fault tolerance
  - Principles of Dependable Systems
- Prof. Aßmann
  - Software Engineering, QA, and tools
- Prof. Baier
  - Model Checking

Resources

- Grottke, Trivedi: “Fighting bugs: remove, retry, replicate and rejuvenate”, IEEE Computer, Feb. 2007
- Engler, Musuvathi: *“Static analysis vs. software model checking for bug finding”*, LNCS Volume 2973/2003
- Engler, Chen, Hallem, Chou, Shelf: *“Bugs as deviant behavior – a general approach to inferring errors in system code”*, SOSP 2001
- Nethercote, Seward: *“Valgrind: A framework for heavyweight dynamic binary analysis”*, PDLI 2007
- Pohlack, Doebel, Lackorzynski: *“Towards runtime monitoring in real-time systems”*, RTLWS 2006
- Pohlack: *“Ein praktischer Erfahrungsbericht über Model Checking in L4Linux”*, OS group internal report, 2006

Resources (2)

- Madhavapedi, Ho, Deegan: *"Melange: Creating a functional internet",* EuroSys 2007
- http://www.valgrind.org
- http://sourceware.org/systemtap
- http://www.splint.org
- http://sourceforge.net/projects/cppunit
- http://sourceforge.net/projects/code2test