
CMSC 838Z

**Tools and Techniques for
Software Dependability**

Spring 2004

Instructor: Dr. Michael Hicks

Dependable Software

- The Holy Grail of computing?
- People make mistakes
 - Especially when writing big and complicated software
- Key idea: use software to check software
 - But need to work around the fact that all useful software properties are undecidable

This Course

- Focus on recently-developed tools
 - How do they make software more dependable?
- Understand the techniques behind them
 - Type checking, advanced language features, type inference, dataflow analysis, theorem proving, model checking, runtime instrumentation, ...
- Put them into practice
 - Use the tools, evaluate them, make them better

Prerequisites

- CMSC 838Y, CMSC 630, or CMSC 631
 - Or my permission
- Type systems
 - Type checking, type inference, typical usage
- Theorem Proving
 - First-order logic, Hoare triples $\{P\} s \{Q\}$
- Dataflow analysis
 - Transform functions
 - Example usage (e.g., constant propagation)
- Model Checking
 - Automata, Temporal logic
- Runtime instrumentation
 - Program transformation techniques

Readings

- Textbooks/tutorials, and research papers
 - Available on the web, or I will provide copies
 - Expect 1-2 papers per week
- Will go over these in class
 - Clarify challenging technical points
 - Evaluate
 - What will this tool do well? Where will it perform poorly?
How could it be improved?
 - Class participation expected
 - 15% of grade

Projects

- Five 1 to 2-week implementation projects that involve the tools we will cover
 - Use the tools to understand and evaluate them better
 - 20% of grade
- One 7-week project of greater depth
 - Larger-scale evaluation or other research project
 - Turn in write-up at the end of class (10-15 pages), and give presentation about what you did (20 minutes)
 - 40% of grade

Exam

- Final Exam
 - Take-home
 - Technical questions involving the techniques we have studied
 - Will have some written homeworks to help prepare
 - More open-ended questions on promising research directions
 - Based on your experience with the tools
 - 25% of grade

Other sources of information

- Software Chat
 - Meets Mondays at 11 am – 12 noon.
 - PL and Software Engineering Faculty and interested students meet to
 - Discuss research being done locally
 - Listen to practice research talks for conferences
 - Discuss research being done within the community (read papers, report on conference trips)

Other sources of information

- Types and PL Reading Group
 - Meets Wednesdays from 2 – 3 pm.
 - Students (and sometimes faculty) meet to learn more about formal PL topics
 - Currently, going through Pierce's Types and Programming Languages book.
 - Will expect deviations to other books, technical papers, etc.
 - Goal is to get more in-depth knowledge on these issues (supplements this course).

Administrivia

- I'll be gone Tuesday, Thursday, Tuesday!
- Reschedule class:
 - Move next Tuesday back to Monday
 - open slots are between 9-11 am, 12-4 pm, or 5-6:30 pm.
 - Move the following Tuesday back to Monday
 - Open slots are 9-11 am, 12-6:30 pm.
- Guest lecturer next Thursday

Tools and Techniques

Evaluation Criteria

- Three main axes
 - What does the tool buy me (purpose)?
 - How often can I use it profitably?
 - How hard it is to use?

I. Purpose (1): Proving Correctness

- Partial correctness
 - Assuming it terminates, does the program do what it's supposed to? Very hard!
- Safety properties
 - Correct API usage
 - files, sockets, etc
 - Memory safety
 - no dangling pointer dereferences or buffer overflows)
 - Concurrency safety
 - absence of deadlock, data races
 - Termination
 - Abstraction/information hiding

I. Purpose (1): Proving Correctness

- Examples
 - Automatic or semi-automatic theorem provers, model checkers, type checking and/or inference systems,...
 - Languages
 - SML, Java, Cyclone, Vault
 - Static (+Dynamic) Analysis Tools
 - CQual, CCured, RCCJava, Blast, SLAM, ESP
 - Theorem Provers
 - HOL, Coq, Twelf, Simplify
 - Related Tools
 - Daikon, ESC/Java

I. Purpose (2): Finding Bugs

- Patterns of improper usage
 - Largely syntactic
 - Makes no guarantees
 - Does not establish that a particular property holds uniformly (i.e., not complete)
 - Does not suppose that all reported errors are actually bugs (i.e., not sound)
- Example Tools
 - FindBugs (UMCP), MCC (Stanford), preFAST (Microsoft), lint (GNU)

I. Purpose (3): Improve Design

- For faster debugging and maintenance
 - Clarify programmer intent
 - Make program structure easier to understand
 - Ensure adherence to high-level design
- Examples
 - Languages with advanced type structure
 - Cyclone, Vault
 - Annotation-based systems
 - RCCJava, ESC/Java

II. Frequency (how fast)?

- Each compile/test/debug cycle (seconds)
- Whenever I unit test (minutes)
- Whenever I system test (hours)
 - E.g., overnight build process
- To validate the entire release (days)
 - E.g., before I ship

III. Burden (how hard?)

- “Additional” annotations
 - How frequently? How hard to write?
- Disallowed coding idioms
 - Forced to use idioms that
 - Feel unnatural or unfamiliar
 - Perform less well
 - E.g., GC rather than malloc/free, adherence to strong typing, functional vs. imperative programming, restrictions on aliases, ...
 - How hard to do? Very subjective.

III. Burden (how hard?)

- Poring over tool results to classify false positives from actual errors
 - Sometimes hard to tell from the site of the reported error without detailed understanding of the code
- Analysis assistance
 - Manual creation of abstraction or desired theorem
 - e.g., for model checking
 - Guiding a proof search
 - How hard to do? Error prone?

III. Burden (how hard?)

- How does continuing development affect these criteria?
 - reclassification of false positives
 - reworking of the model or proofs
 - the less modular/compositional the approach, the more of a pain this is

Tools to Study

Cyclone

- Safe, C-like programming language
- Features
 - Type- and memory-safety
 - Support low-level operations and representations
 - Flattened datastructures
 - Manual memory management
 - Ease of use
 - Tools for interfacing with C
 - Advanced features: exceptions, pattern matching, parametric polymorphism, extensible datatypes, ...

Vault

- Safe, C-like programming language
- Features
 - Supports “property checking” to ensure APIs are used correctly.
 - Relies on “linear” system of tracked types to ensure that resources are managed soundly.
 - Targeted at device drivers

CQual

- Static analysis tool for C programs
- Features
 - Set-constraint-style whole-program analysis, much like alias analysis, for analyzing uses of type qualifiers in programs.
 - Qualifiers can be flow-sensitive
 - Can be used to check proper uses of `const`, to find possible deadlocks, to prevent unsafe access to “tainted” data, ...

RCCjava

- Tool that ensures the absence of data races in Java programs
- Features
 - Defines extended type system; type correctness implies no data races
 - Supports polymorphism and thread-local data
 - Requires extra annotations (no inference)
 - Supports suppression of spurious warnings

ESC/Java

- Partial correctness checker for Java programs
- Features
 - Based on Hoare-style reasoning with automated theorem proving
 - Requires annotations on Java programs, which are checked
 - Neither sound nor complete

Blast

- Automatic software model checker for C programs
- Features
 - Widely used approach of automated abstraction with counterexample-driven refinement
 - Supports reasoning about concurrent programs
 - Reasonably fast

Simplify

- Automated Theorem Prover
- Features
 - Underlying prover for ESC/Java
 - Simplification of predicate logic formulas augmented with other theories (e.g., real number arithmetic, arrays, etc.)
 - Technique focuses on disproving the negation of the desired formula, to help understand where the proof failed

Twelf

- Permits encoding checkable formal proofs
- Features
 - Logical framework via dependent types
 - Permits one to encode the logic in which to perform the proof.
 - Type checking = proof checking
 - If your proof type-checks then it's consistent.
 - The proof language for proof-carrying code
 - Prove properties about software
 - Reduces the burden of trust on the code consumer

Daikon

- Dynamic invariant inference
- Features
 - Instruments programs to discover likely program invariants at runtime
 - Inferred variants are not sound per se, but often turn out to be so
 - Has front-ends for C, C++, and Java

Others?

- If there's a tool you know about that you'd like to learn more about, let me know.

Type-based Tools

Type Systems

- Designed to prevent certain kinds of *execution errors* during the run of a program.
- A typechecker checks that a program is *well-typed*, relative to the language's type system.
- Typechecking be described as a *judgment*
 - $G \vdash e : T$
 - Reads “program e has type T under assumptions G .”
- Logical *rules of inference* are used to determine when a judgment is *valid*. A proof that a judgment is valid is called a *derivation*.

Type Soundness

- That a well-typed program cannot have execution errors (of a certain kind) is formalized in a statement of *type soundness*:
- If $G \vdash e : T$, then either
 - e is a value (a legal final form), or else
 - $G \vdash e' : T$, where $e \rightarrow e'$
 - $e \rightarrow e'$ indicates that e *evaluates* to e' . The fact that e' is well-typed implies no execution errors

Type System Features

- Why use type systems? Why not informal comments on the code only? Or else stronger theorem provers, model checkers, etc.?
- Type systems are
 - *Decidably verifiable*. A (terminating) algorithm can check that the program is well-typed.
 - *Transparent*. A programmer can understand why typechecking succeeds and fails.
 - *Enforceable*. Typed declarations largely can be statically checked, with added dynamic checks if necessary.

Type System Properties

- What execution errors can type systems prevent? More than you'd think:
 - Memory errors
 - Buffer overflows, dangling pointer dereferences
 - Deadlocks and data races
 - Violations of security properties
 - information flow
 - Violations of ADT usage
 - ...

Formal Type Systems

- We'll look at simple type systems,
- Usage of *parametric polymorphism* (universal quantification) and *data hiding* (existential quantification).
- This leads into our discussion of **Cyclone**, for which you have a project due in two weeks ...