Ερευνητικά Θέματα Ανάπτυξης Λογισμικού

Σύνοψη Μαθήματος (1)

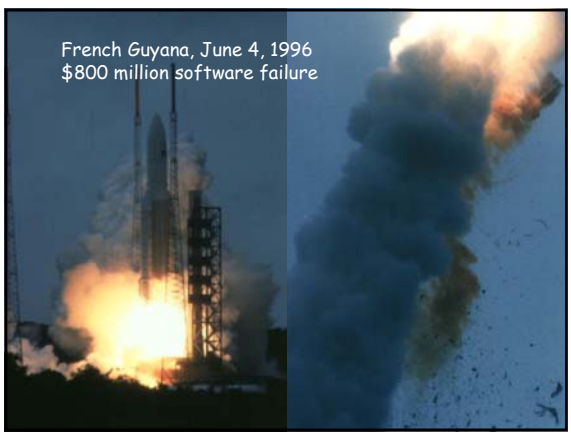
1. **Introduction & Course Overview**
   - How software is built & software defects
   - Dataflow analysis fundamentals
2. **Testing**
   - Testing practice
   - Coverage
   - Automatic test generation — Korat, JUnit
3. **Debugging**
   - Debuggers
   - Debugging without debuggers — Delta debugging
   - Simplifying failure inducing input
4. **Runtime Monitoring**
   - Detecting data races — Eraser
   - Virtual machine simulation
   - Fault isolation — Valgrind

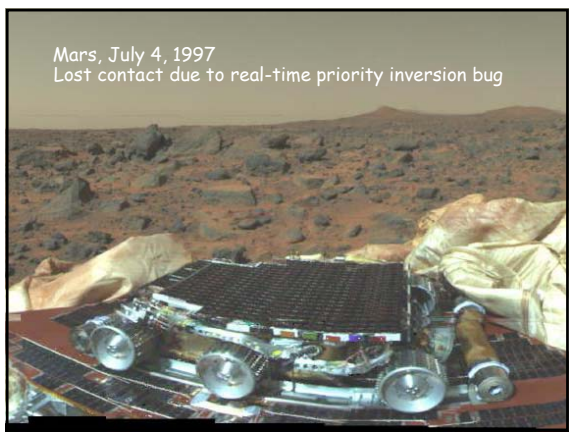
Σύνοψη Μαθήματος (2)

5. **Static Analysis**
   - Basic principles
   - Dataflow analysis
6. **Static Bug Detection I**
   - Heuristics-based methods
   - Annotations & dataflow analysis — Lint, LClint
7. **Static Bug Detection II**
   - Bug Patterns — FindBugs
   - Scalable analyses — Prefast, Prefix
8. **Static Bug Detection III**
   - Metacompilation — Metal
   - Statistical ranking

Σύνοψη Μαθήματος (3)

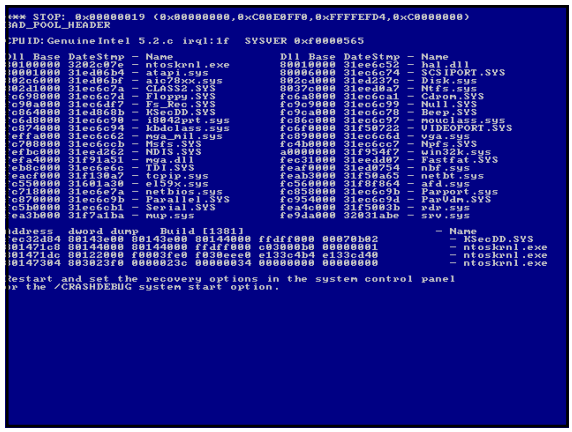
9. **Extended Static Checking**
   ESC/Java, Houdini, ESC/Java2, ESC/Haskell
10. **Model Checking**
    Slam, BLAST
11. **Memory Management & Safety**
    - Regions: an alternative to garbage collection
    - Cyclone
12. **Fixing C**
    - CCured
13. **Fixing Java**
    - FindBugs
    - **Type-based race detection for Java**
    - **Dynamic atomicity checker** — **Atomizer**





Mars Climate Orbiter

- The 125 million dollar Mars Climate Orbiter is assumed lost by officials at NASA. The failure responsible for loss of the orbiter is attributed to a failure of NASA's system engineer process. The process did not specify the system of measurement to be used on the project. As a result, one of the development teams used Imperial measurement while the other used the metric system of measurement. When parameters from one module were passed to another during orbit navigation correct, no conversion was performed, resulting in the loss of the craft.



The Blue Screen

More Blue Screen Embarrassments

CNN.com

PRINT THIS

Powered by clickability

Spread of buggy software raises new questions

NEW YORK (AP) –When his dishwasher acts up and won't stop beeping, Jeff Seigle turns it off and then on, just as he does when his computer crashes. Same with the exercise machines at his gym and his CD player.

"Now I think of resetting appliances, not just computers," says Seigle, a software developer in Vienna, Virginia.

Malfunctions caused by bizarre and frustrating glitches are becoming harder and harder to escape now that software controls everything from stoves to cell phones, trains, cars and power plants.

--A poorly programmed ground-based altitude warning system was partly responsible for the 1997 Korean Air crash in Guam that killed 228 people.

--Faulty software in anti-lock brakes forced the recall of 39,000 trucks and tractors and 6,000 school buses in 2000.

--The $165 million Mars Polar Lander probe was destroyed in its final descent to the planet in 1999, probably because its software shut the engines off 100 feet above the surface.

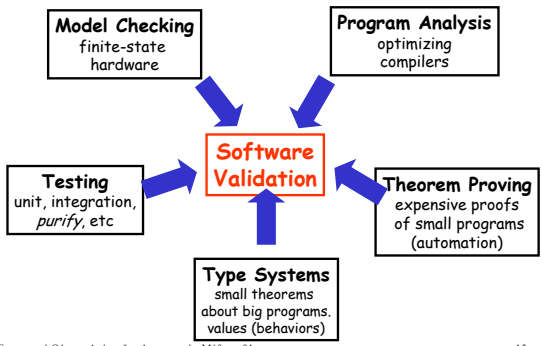
Economic Impact

- **NIST study**
  - On CNN.com - April 27, 2003

Last year, a study commissioned by the National Institute of Standards and Technology found that software errors cost the U.S. economy about $59.5 billion annually, or about 0.6 percent of the gross domestic product. More than half the costs are borne by software users, the rest by developers and vendors.

`http://www.nist.gov/director/prog-ofc/report02-3.pdf`

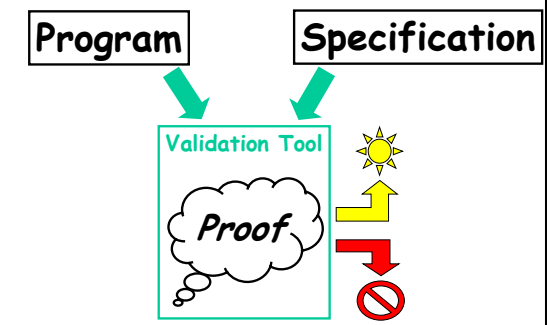
Course Overview



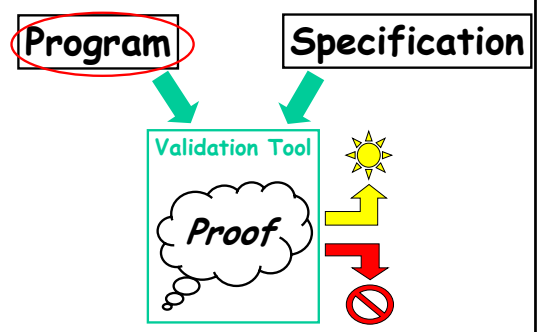
Analysis vs. Validation

- **Analysis**
  - figure out what properties a program satisfies
- **Validation**
  - does a program satisfy a particular property or "specification"

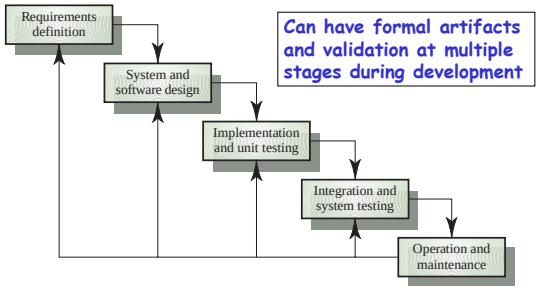
Software Validation



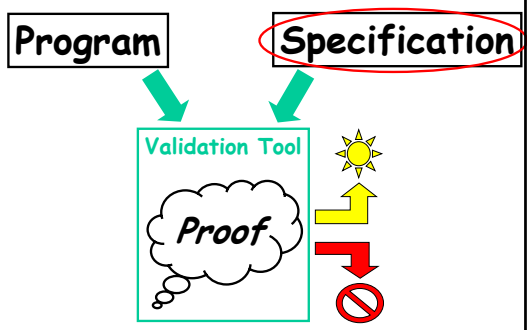
Software Validation



Waterfall Model



Software Validation



Specifications

- **Safety**
  - something "bad" will never happen
  - finds most bugs
- **Liveness**
  - something "good" will eventually happen
  - (we don't know when)

For Sequential Programs

- **Safety**
  - the program will never produce a wrong result
    - "partial correctness"
- **Liveness**
  - the program will produce a result
    - "termination"

Example Specifications

Basic specifications
- no null dereference
- no bounds errors
- no segmentation faults

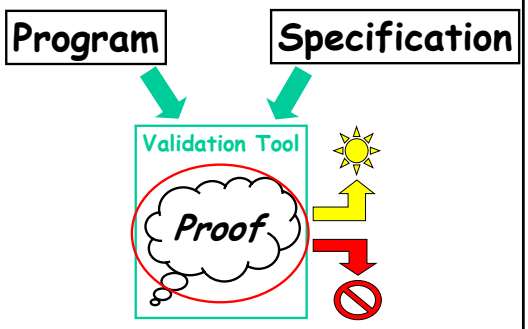
Resource management
- no memory leaks

Concurrency
- no race conditions
- no deadlock

Example Specifications

- **Data invariants**
  - shape properties (L is an acyclic list)
- **Security**
  - integrity, confidentiality
- **API Usage rules**
  - files
  - UNIX sockets

Software Validation



Undecidability

- **Does a program P satisfy a specification S ?**
  - everything interesting about infinite-state programs is undecidable
    - consequence of halting problem
  - no sound, complete, terminating algorithm

Avoiding Undecidability

- **Finite state systems**
  - model checking
  - automatic
  - effective for small systems
- **Miss errors (unsound)**
  - testing, bounded model checking
  - test coverage problem
- **False alarms (incomplete)**
  - program analysis, type systems
  - only consider certain proofs

Program Analysis Fundamentals

Uses of Program Analysis

- **Historically: Optimizing compilers**
- **More recently: Finding bugs**

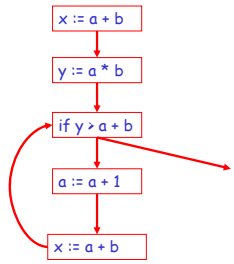
Culture

- **Emphasis on low-complexity techniques**
  - Because of emphasis on usage in tools
  - High-complexity techniques also studied, but often don't survive
- **Emphasis on complete automation**
- **Driven by language features**
  - Particular languages and features give rise to their own sub-disciplines

Introduction to Dataflow Analysis

Control-Flow Graphs

```
x := a + b;
y := a * b;
while y > a + b {
  a := a + 1;
  x := a + b
}
```



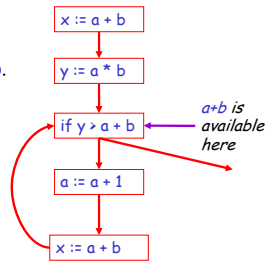
Notation

s is a statement
succ(s) = { *successor statements of s* }
pred(s) = { *predecessor statements of s* }
write(s) = { *variables written by s* }
read(s) = { *variables read by s* }

Note: *In literature write = kill and read = gen*

Available Expressions

- For each program point p, finds which expressions must have already been computed, and not later modified, on all paths to p.
- Optimization: Where available, expressions need not be recomputed.

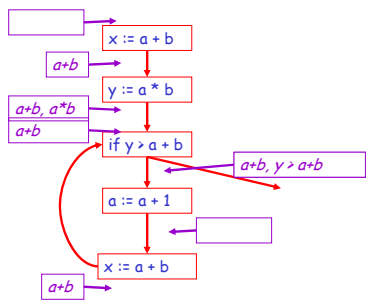


Dataflow Equations

$$A_{in}(s) = \begin{cases} \varnothing & \text{if } pred(s) = \varnothing \\ \bigcap_{s' \in pred(s)} A_{out}(s') & \text{otherwise} \end{cases}$$

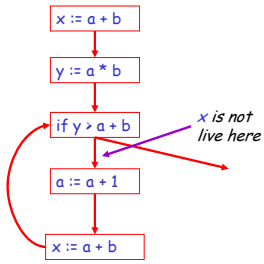
$$A_{out}(s) = (A_{in}(s) - \{a \in S \mid write(s) \cap V(a) \neq \varnothing\}) \cup \{s \mid \text{if } write(s) \cap read(s) = \varnothing\}$$

Example



Liveness Analysis

- For each program point p, finds which of the variables defined at that point are used on some execution path?
- Optimization: If a variable is not live, there is no need to keep it in a register.

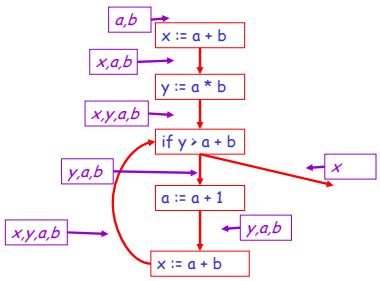


Dataflow Equations

$$L_{in}(s) = (L_{out}(s) - write(s)) \cup read(s)$$

$$L_{out}(s) = \begin{cases} \emptyset & \text{if } succ(s) = \emptyset \\ \bigcup_{s' \in succ(s)} L_{in}(s') & \text{otherwise} \end{cases}$$

Example



Available Expressions Again

$$A_{in}(s) = \begin{cases} \emptyset & \text{if } pred(s) = \emptyset \\ \bigcap_{s' \in pred(s)} A_{out}(s') & \text{otherwise} \end{cases}$$

$$A_{out}(s) = (A_{in}(s) - \{a \in S \mid write(s) \cap V(a) \neq \emptyset\}) \cup \{s \mid write(s) \cap read(s) = \emptyset\}$$

Available Expressions: Schematic

$$A_{in}(s) = \bigcap_{s' \in pred(s)} A_{out}(s')$$

Transfer function:

$$A_{out}(s) = A_{in}(s) - C_1 \cup C_2$$

Must analysis: property holds on all paths

Forwards analysis: from inputs to outputs

Live Variables Again

$$L_{in}(s) = (L_{out}(s) - write(s)) \cup read(s)$$

$$L_{out}(s) = \begin{cases} \emptyset & \text{if } succ(s) = \emptyset \\ \bigcup_{s' \in succ(s)} L_{in}(s') & \text{otherwise} \end{cases}$$

Live Variables: Schematic

Transfer function:

$$in(s) = out(s) - C_1 \cup C_2$$

$$out(s) = \bigcup_{s' \in succ(s)} in(s')$$

May analysis: property holds on some path

Backwards analysis: from outputs to inputs

Very Busy Expressions

- An expression *e* is very busy at a program point p if every path from p must evaluate e before any variable in e is redefined
- Optimization: hoisting expressions
- A must-analysis
- A backwards analysis

Reaching Definitions

- For a program point p, which assignments made on paths reaching p have not been overwritten
- Connects definitions with uses (use-def chains)
- A may-analysis
- A forwards analysis

One Cut at the Dataflow Design Space

	May	*Must*
Forwards	**Reaching definitions**	**Available expressions**
Backwards	**Live variables**	**Very busy expressions**

The Literature

- **Vast literature of dataflow analyses**
- **90+% can be described by**
  - Forwards or backwards
  - May or must
- **Some oddballs, but not many**
  - Bidirectional analyses

Flow Sensitivity

- **Flow sensitive analyses**
  - The order of statements matters
  - Need a control flow graph
    - Or transition system, ….
- **Flow insensitive analyses**
  - The order of statements doesn't matter
  - Analysis is the same regardless of statement order

Example Flow Insensitive Analysis

- **What variables does a program fragment modify?**

$$G(x := e) = \{x\}$$
$$G(s_1; s_2) = G(s_1) \cup G(s_2)$$

- Note $G(s_1;s_2) = G(s_2;s_1)$

The Advantage

- **Flow-sensitive analyses require a model of program state at each program point**
  - E.g., liveness analysis, reaching definitions, …
- **Flow-insensitive analyses require only a single global state**
  - E.g., for G, the set of all variables modified

Notes on Flow Sensitivity

- **Flow insensitive analyses seem weak, but:**
- **Flow sensitive analyses are hard to scale to very large programs**
  - Additional cost: state size X # of program points
- **Beyond 1000's of lines of code, only flow insensitive analyses have been shown to scale**

Context-Sensitive Analysis

- **What about analyzing across procedure boundaries?**

```
Def f(x){…}
Def g(y){…f(a)…}
Def h(z){…f(b)…}
```

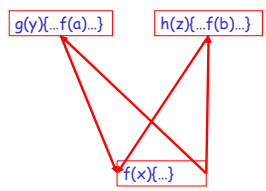
- **Goal:** Specialize analysis of f to take advantage of
  - f is called with a by g
  - f is called with b by h

Control-Flow Graphs Again

- **How do we extend control-flow graphs to procedures?**
- **Idea: Model procedure call f(a) by:**
  - Edge from point before call to entry of f
  - Edge from exit(s) of f to point after call

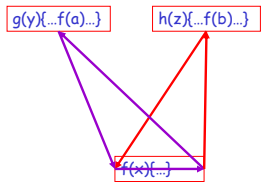
Example

- **Edges from**
  - before f(a) to entry of f
  - Exit of f to after f(a)
  - Before f(b) to entry of f
  - Exit of f to after f(b)



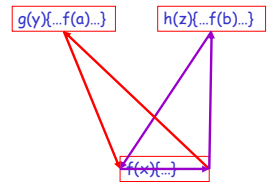
Example

- Edges from
  - before f(a) to entry of f
  - Exit of f to after f(a)
  - Before f(b) to entry of f
  - Exit of f to after f(b)
- Has the correct flows for g



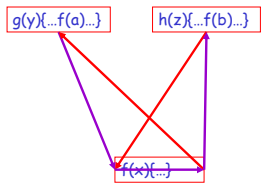
Example

- Edges from
  - before f(a) to entry of f
  - Exit of f to after f(a)
  - Before f(b) to entry of f
  - Exit of f to after f(b)
- Has the correct flows for h



Example

- But also has flows we don't want
  - One path captures a call to g returning at h!
- So-called "infeasible paths"
- Must distinguish calls to f in different contexts



Review of Terminology

- Must vs. May
- Forwards vs. Backwards
- Flow-sensitive vs. Flow-insensitive
- Context-sensitive vs. Context-insensitive

Where is Dataflow Analysis Useful?

- Best for flow-sensitive, context-insensitive problems on small pieces of code
  - E.g., the examples we've seen and many others
- Extremely efficient algorithms are known
  - Use different representation than control-flow graph, but not fundamentally different
  - More on this in a minute . . .

Where is Dataflow Analysis Weak?

- Lots of places

Data Structures

- **Not good at analyzing data structures**
- **Works well for atomic values**
  - Labels, constants, variable names
- **Not easily extended to arrays, lists, trees, etc.**
  - Work on shape analysis

The Heap

- **Good at analyzing flow of values in local variables**
- **No notion of the heap in traditional dataflow applications**
- **In general, very hard to model anonymous values accurately**
  - Aliasing
  - The "strong update" problem

Context Sensitivity

- Standard dataflow techniques for handling context sensitivity don't scale well
- Brittle under common program edits

Flow Sensitivity (Beyond Procedures)

- **Flow sensitive analyses are standard for analyzing single procedures**
- **Not used (or not aware of uses) for whole programs**
  - Too expensive

The Call Graph

- **Dataflow analysis requires a call graph**
  - Or something close
- **Inadequate for higher-order programs**
  - First class functions
  - Object-oriented languages with dynamic dispatch
- **Call-graph hinders algorithmic efficiency**
  - Desire to keep executable specification is limiting

Forwards vs. Backwards

- **Restriction to forwards/backwards reachability**
  - Very constraining
  - Many important problems not easy to fit into this mold