Exhaustive Versus Demand Analysis

- Exhaustive analysis: **All** facts at **all** points
- Optimization: Concentrate on inner loops
- Program-understanding tools: Only some facts are of interest

What Are Slices Useful For?

- Understanding Programs
  - What is affected by what?
- Restructuring Programs
  - Isolation of separate "computational threads"
- Program Specialization and Reuse
  - Slices = specialized programs
  - Only reuse needed slices
- Program Differencing
  - Compare slices to identify changes
- Testing
  - What new test cases would improve coverage?
  - What regression tests must be rerun after a change?

Line-Character-Count Program

```
void line_char_count(FILE *f) {
     int lines = 0;
     int chars;
     BOOL eof_flag = FALSE;
     int n;
     extern void scan_line(FILE *f, BOOL *bptr, int *iptr);
     scan_line(f, &eof_flag, &n);
     chars = n;
     while(eof_flag == FALSE){
          lines = lines + 1;
          scan_line(f, &eof_flag, &n);
          chars = chars + n;
     }
     printf("lines = %d\n", lines);
     printf("chars = %d\n", chars);
}
```

Character-Count Program

```
void char_count(FILE *f) {
     int lines = 0;
     int chars;
     BOOL eof_flag = FALSE;
     int n;
     extern void scan_line(FILE *f, BOOL *bptr, int *iptr);
     scan_line(f, &eof_flag, &n);
     chars = n;
     while(eof_flag == FALSE){
          lines = lines + 1;
          scan_line(f, &eof_flag, &n);
          chars = chars + n;
     }
     printf("lines = %d\n", lines);
     printf("chars = %d\n", chars);
}
```

Line-Character-Count Program

```
void line_char_count(FILE *f) {
     int lines = 0;
     int chars;
     BOOL eof_flag = FALSE;
     int n;
     extern void scan_line(FILE *f, BOOL *bptr, int *iptr);
     scan_line(f, &eof_flag, &n);
     chars = n;
     while(eof_flag == FALSE){
          lines = lines + 1;
          scan_line(f, &eof_flag, &n);
          chars = chars + n;
     }
     printf("lines = %d\n", lines);
     printf("chars = %d\n", chars);
}
```

Line-Count Program

```
void line_count(FILE *f) {
     int lines = 0;
     int chars;
     BOOL eof_flag = FALSE;
     int n;
     extern void scan_line2(FILE *f, BOOL *bptr, int *iptr);
     scan_line2(f, &eof_flag, &n);
     chars = n;
     while(eof_flag == FALSE){
          lines = lines + 1;
          scan_line2(f, &eof_flag, &n);
          chars = chars + n;
     }
     printf("lines = %d\n", lines);
     printf("chars = %d\n", chars);
}
```

Exhaustive Versus Demand Analysis

- Demand analysis:
  - Does a **given** fact hold at a **given** point?
  - **Which** facts hold at a **given** point?
  - At **which** points does a **given** fact hold?
- Demand analysis via CFL-reachability
  - single-source/single-target CFL-reachability
  - single-source/multi-target CFL-reachability
  - multi-source/single-target CFL-reachability

CFL-Reachability: Benefits

- Algorithms
  - Exhaustive & demand
- Complexity
  - Linear-time and cubic-time algorithms
  - PTIME-completeness
  - Variants that are undecidable
- Complementary to
  - Equations
  - Set constraints
  - Types
  - . . .

Constraint Problems
in
Program Analysis

from the sublime to the ridiculous

Alex Aiken
Stanford University

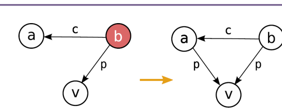
Two Styles of Analysis

- Whole program
  - Entire program needed for analysis of any piece
- Compositional
  - Can analyze partial or "open" programs (libraries)
- Intimately connected to solving complexity
  - Leads to very different engineering issues
  - This is poorly understood today

Algorithms and Engineering

- Algorithms: PTIME is good enough
- Engineering: linear space is essential
  - Must also be close to linear time
  - These algorithms are applied at large scales
  - Linux kernel 6.2MLOC

Constraints as Graphs

name	rule	ensures
copy		$pts(a) \subseteq pts(b)$

Solutions

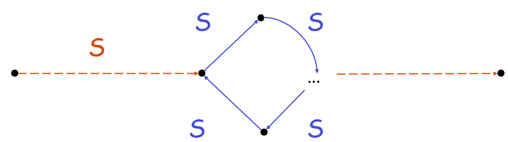
- Solution size is potentially $O(n^2)$
  - May be the complete graph
- Solution time is $O(n^3)$
  - Each of $O(n^2)$ edges may be added in $O(n)$ ways
- A major engineering issue
  - 1996: analyze 5 KLOC
  - 2002: analyze 6MLOC
    - now in production compilers

Optimization: Cycle Elimination

- Variables in a cycle are all equivalent

$$X_1 \mu X_2 \ldots \mu X_n \mu X_1$$

- Optimization: collapse them into one variable



Discussion

- Good techniques for cycle-elimination known
  - Does not change worst-case complexity
  - But makes 100X time difference
- Specific algorithmic/implementation techniques are critical to the success of decision procedures
  - Even “cheap” ones
  - Support for such research is important

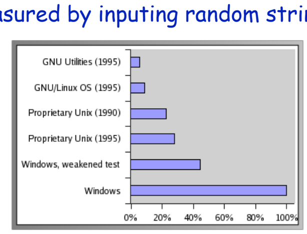
CSC 255/455
Software Analysis and Improvement

Introduction to Software Correctness

Instructor: Chen Ding

Correctness/Reliability/Security

- Windows 98 crashed on live TV
  - http://www.youtube.com/watch?v=z6b9ToIRbgM
- A complete failure of Navy’s “smart ship” in 1997
- ABI report 2004
  - approximately 30% of all automotive warranty issues today are software and silicon-related
- Failure rate measured by inputing random strings



Wheeler’s Report

- Why Open Source Software / Free Software (OSS/FS, FLOSS, or FOSS)? Look at the Numbers!
  - http://www.dwheeler.com/oss_fs_why.html
  - David A. Wheeler, 2007
- Reliability
  - under random inputs, frequency of reboots, time to fix hardware/system faults
  - code defects per KSLOC
    - memory leaks, null pointer dereference, out of bound array access, uninitialized variables
  - maintainability
  - modularity
    - cost of redesign

Wheeler's Report (cont'd)

- Security
  - cost of "hacker insurance" for Windows vs. Unix/Linux
  - number of installations compromised
  - time before compromised on Internet
    - 4 minute for unpatched Windows XP
  - vulnerability database
  - faster response to advisories
  - survey of developers
  - frequency of attacks, prevalence of viruses
  - expert recommendations
    - adoption by Homeland security e.g.
- Other factors
  - performance, scalability, total cost of ownership, legal, social, moral etc

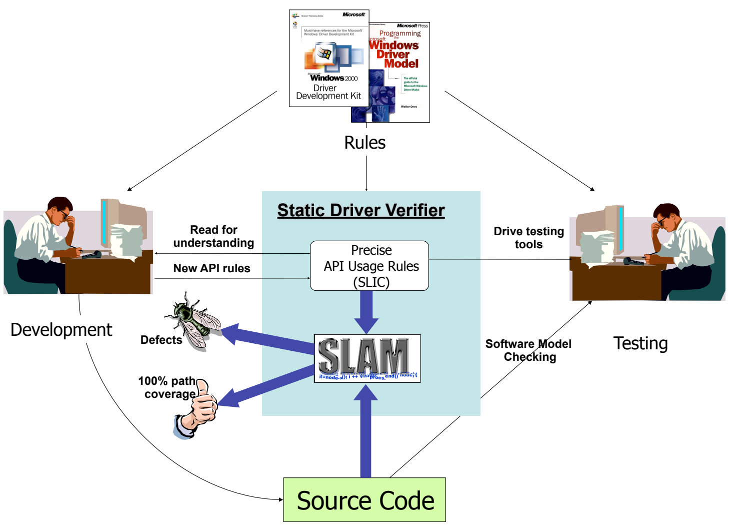
The Static Driver Verifier Research Platform

Thomas Ball[1], Ella Bounimova[1], Vladimir Levin[2], Rahul Kumar[2], and Jakob Lichtenberg[2]

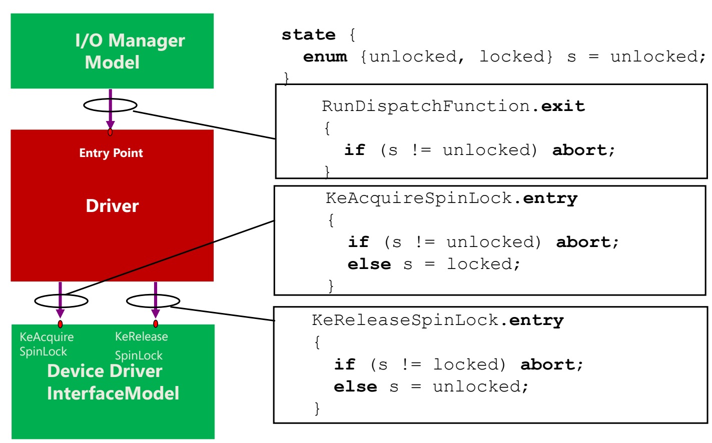
[1]Microsoft Research

[2]Microsoft Windows

http://research.microsoft.com/slam/



API SLIC Rule



SLAM2 Verification Engine

SLAM 2.0 released with SDV 2.0, part of Windows 7 WDK

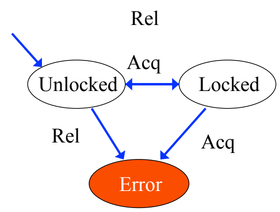
Parameter for WDM drivers	SDV 2.0 (SLAM2)	SDV 1.6 (SLAM1)
False defects	0.4% (2/512)	19.7% (31/157)
Give-up results	3.2% (187/5727)	6% (285/4692)

SLAM Status

- **2000-2001**
  - foundations, algorithms, prototyping
  - papers in CAV, PLDI, POPL, SPIN, TACAS
- **March 2002**
  - Bill Gates review
- **May 2002**
  - Windows committed to hire two people with model checking background to support Static Driver Verifier (SLAM+driver rules)
- **July 2002**
  - running SLAM on 100+ drivers, 20+ properties
- **September 3, 2002**
  - made initial release of SDV to Windows (friends and family)
- **April 1, 2003**
  - made wide release of SDV to Windows (any internal driver developer)

Type-State Analysis in MSR SLAM Tool

State Machine for Locking



Locking Rule in SLIC

```
state {
  enum {Locked,Unlocked}
    s = Unlocked;
}

KeAcquireSpinLock.entry {
  if (s==Locked) abort;
  else s = Locked;
}

KeReleaseSpinLock.entry {
  if (s==Unlocked) abort;
  else s = Unlocked;
}
```

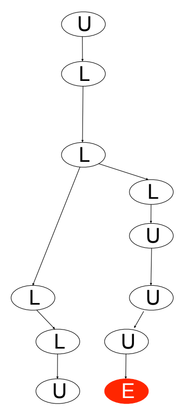
Example

Does this code obey the locking rule?

```
do {
  KeAcquireSpinLock();

  nPacketsOld = nPackets;

  if(request){
      request = request->Next;
      KeReleaseSpinLock();
      nPackets++;
  }
} while (nPackets != nPacketsOld);

KeReleaseSpinLock();
```

Example

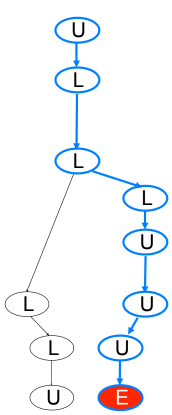
Model checking boolean program (bebop)




```
do {
  KeAcquireSpinLock();


  if(*){

      KeReleaseSpinLock();

  }
} while (*);

KeReleaseSpinLock();
```

Example

Is error path feasible in C program? (newton)




```
do {
  KeAcquireSpinLock();

  nPacketsOld = nPackets;

  if(request){
      request = request->Next;
      KeReleaseSpinLock();
      nPackets++;
  }
} while (nPackets != nPacketsOld);

KeReleaseSpinLock();
```

c2bp: Predicate Abstraction for C Programs

Given

- P : a C program
- F = {$e_1,\ldots,e_n$}
  - each e_i a pure boolean expression
  - each e_i represents set of states for which e_i is true

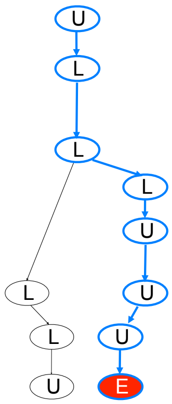
Produce a *boolean program* B(P,F)

- same control-flow structure as P
- boolean vars {$b_1,\ldots,b_n$} to match {$e_1,\ldots,e_n$}
- properties true of B(P,F) are true of P

Example

b : (nPacketsOld == nPackets)

Add new predicate to boolean program (c2bp)

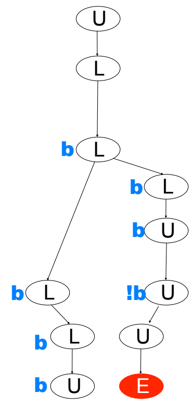


```
do {
  KeAcquireSpinLock();

  nPacketsOld = nPackets; b = true;

  if(request){
      request = request->Next;
      KeReleaseSpinLock();
      nPackets++; b = b ? false : *;
  }
} while (nPackets != nPacketsOld);   !b

KeReleaseSpinLock();
```

Example

b : (nPacketsOld == nPackets)

Model checking refined boolean program (bebop)



```
do {
  KeAcquireSpinLock();

  b = true;

  if(*){

      KeReleaseSpinLock();
      b = b ? false : *;
  }
} while ( !b );

KeReleaseSpinLock();
```

Bebop

- Model checker for boolean programs
- Based on CFL reachability
  - [Sharir-Pnueli 81] [Reps-Sagiv-Horwitz 95]
- Iterative addition of edges to graph
  - "path edges": <entry,d1> → <v,d2>
  - "summary edges": <call,d1> → <ret,d2>

CFL-Reachability

[Yannakakis 90]

- G: Graph (N nodes, E edges)
- L: A context-free language
- L-path from s to t iff $s \xrightarrow{\alpha}{}^{*} t, \alpha \in L$
- Running time: $O(N^3)$

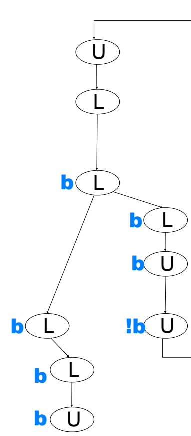
Symbolic CFL reachability

- Partition path edges by their "target"
  - PE(v) = { <d1,d2> | <entry,d1> → <v,d2> }
- What is <d1,d2> for boolean programs?
  - A bit-vector!
- What is PE(v)?
  - A set of bit-vectors
- Use a BDD (attached to v) to represent PE(v)

Example

b : (nPacketsOld == nPackets)

Model checking refined boolean program (bebop)



```
do {
  KeAcquireSpinLock();

  b = true;

  if(*){

      KeReleaseSpinLock();
      b = b ? false : *;
  }
} while ( !b );

KeReleaseSpinLock();
```

Observations about SLAM

- Automatic discovery of invariants
  - driven by property and a finite set of (false) execution paths
  - predicates are ***not*** invariants, but *observations*
  - abstraction + model checking computes inductive invariants (boolean combinations of observations)
- A hybrid dynamic/static analysis
  - newton executes path through C code symbolically
  - c2bp+bebop explore all paths through abstraction
- A new form of program slicing
  - program code and data not relevant to property are dropped
  - non-determinism allows slices to have more behaviors

Scaling SLAM

- Largest driver we have processed has ~60K lines of code
- Largest abstractions we have analyzed have several hundred boolean variables
- Routinely get results after 20-30 iterations
- Out of 672 runs we do daily, 607 terminate within 20 minutes

What is hard?

- Abstracting
  - from a language with pointers (C)
  - to one without pointers (boolean programs)
- All side effects need to be modeled by copying (as in dataflow)
- Open environment problem

What worked well?

- Specific domain problem
- Safety properties
- Shoulders & synergies
- Separation of concerns
- Summer interns & visitors
  - Sagar Chaki, Todd Millstein, Rupak Majumdar (2000)
  - Satyaki Das, Wes Weimer, Robby (2001)
  - Jakob Lichtenberg, Mayur Naik (2002)
  - Giorgio Delzanno, Andreas Podelski, Stefan Schwoon
- Windows Partners
  - Byron Cook, Vladimir Levin, Abdullah Ustuner

Future Work

- Concurrency
  - SLAM analyzes drivers one thread at a time
  - Work in progress to analyze interleavings between threads
- Rules and environment-models
  - Large scale development or rules and environment-models is a challenge
  - How can we simplify and manage development of rules?
- Modeling C semantics faithfully
- Theory:
  - Prove that SLAM will make progress on any property and any program
  - Identify classes of programs and properties on which SLAM will terminate