Lightweight Language-Independent Program Slicing

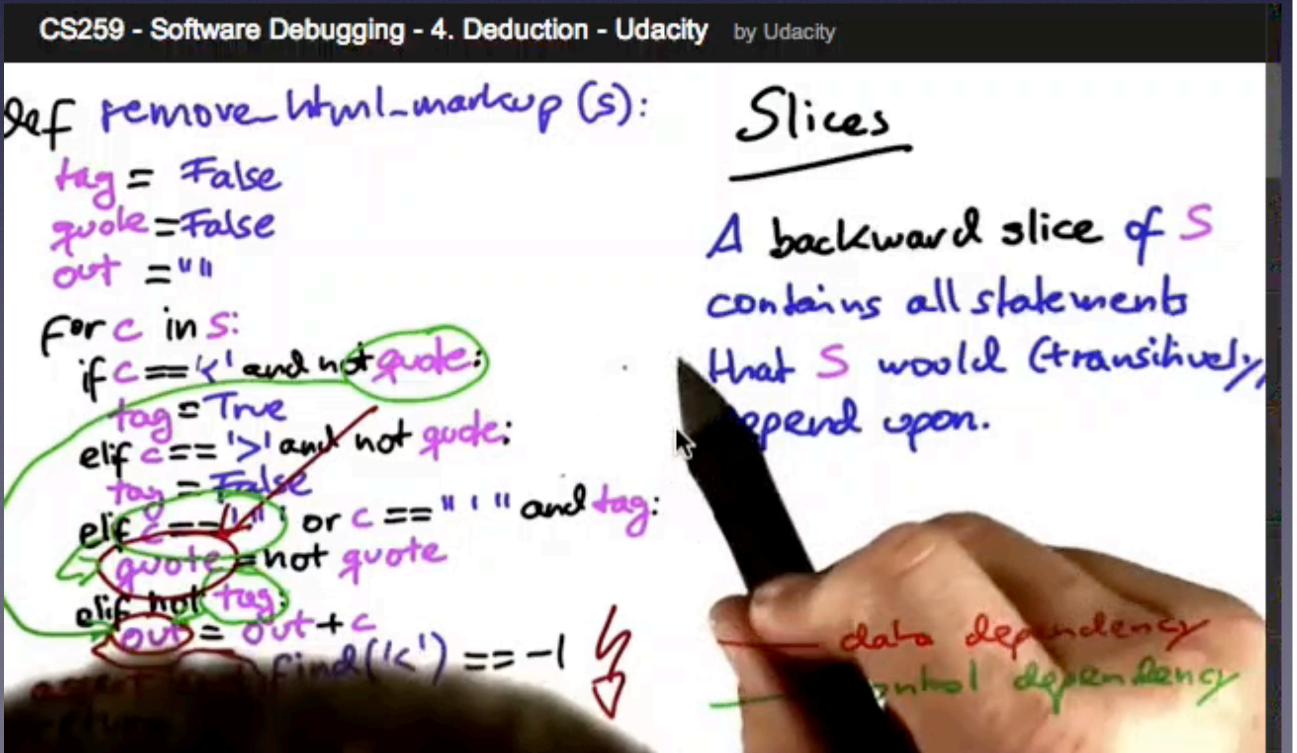
Jens Krinke
CREST, Department of Computer Science
University College London
David Binkley, Nicolas Gold, Mark Harman, Syed Islam, Shin Yoo

COW 29, 19/11/2013

What is Program Slicing?

A program slice contains all statements that a statement depends on.

Redundant code is deleted.



CS259 - Software Debugging - 4. Deduction - Udacity by Udacity

```
def remove_html_markup(s):
  tag = False
  quote = False
  out = ""
  for c in s:
    if c == '<' and not quote:
      tag = True
    elif c == '>' and not quote:
      tag = False
    elif c == '"' or c == "'" and tag:
      quote = not quote
    elif not tag:
      out = out + c
          find('<') == -1
```

Slices

A backward slice of S contains all statements that S would (transitively) depend upon.

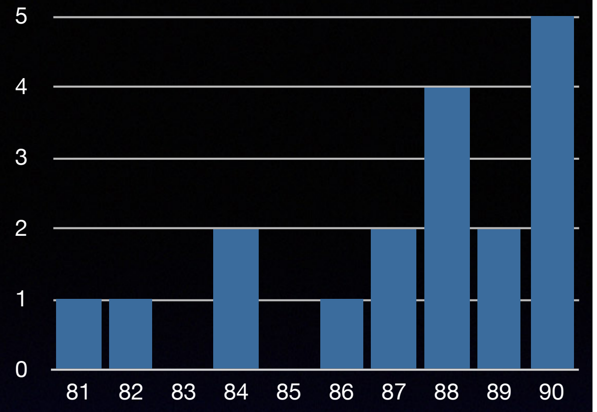
Where is Slicing used?

- Debugging:
  Which statements may have caused a fault?
- Comprehension:
  Which statements influence a statement?
- Evolution:
  What is a change’s impact?
- Testing:
  Which tests have to be rerun?

Program Slicing
100 Article Titles

dynamic
generation
static
mutants
brief
atpg-based
integration
hardware
extraction
maintenance
approach
high
assisted
presence
node
problems
explicitly
formalisation
technique
semantic
comprehension
bound
ming
restructuring
oriented
techniques
study
eng
algorithmic
vhdl
applications
functions
algorithm
application
ansi
assignment
wcet
testing
lightweight
information
relations
unifying
flow
location
composition
subsets
xml
fault
fermat
effective
data
reusable
evaluation
analysis
checking
detection
generalized
test
debugging
dependence-cache
interpretation
method
aspect-oriented
softw
ripple
languages
large
parametric
forward
effect
empirical
specc
new
automatic
transformations
preconditions
execution
support
combination
framework
hierarchical
using
understanding
parallel
theoretical
object-oriented
invariant
classification
via
simultaneous
metrics
amorphous
tool
relationship
dependence
provenance
improving
evolution
transformation
loop
abstract
points-to
behaviors
sigsoft
reverse
case
comments
simplify
role
model
localization
executable
pointer
code
identifying
specialization
aspect
graphs
software
evaluating
comparative
description
traditional
faster
based
towards
cost
component
source
semantics
regression
recovery
extracting
beyond
denotational
engineering
complexity
experimental
survey
overview
process
evaluate
system
source-level
analyze
descriptions
specification
bug
methods
function

First 10 years



79, 81, 82, 84 - Mark Weiser's articles

84 - Slicing in Dependence Graphs

86 - Dicing

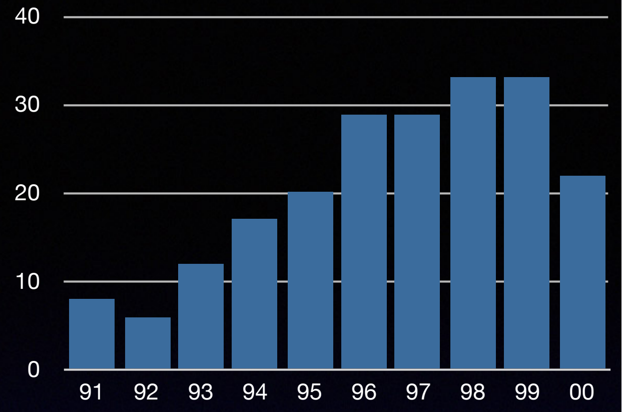
87 - Fault Localisation

88 - Dynamic Slicing

88 - Applications: Maintenance, Differencing

88 - Semantics

Busy 10 years



91 - Quasi-static slicing

92 - Testing

93 - Pointers

93 - Concurrency

93 - Specifications

93 - Functional Languages

93 - Function Extraction

94 - Chopping

94 - OOP

95 - Parametric Slicing

95 - Frank Tip's Survey

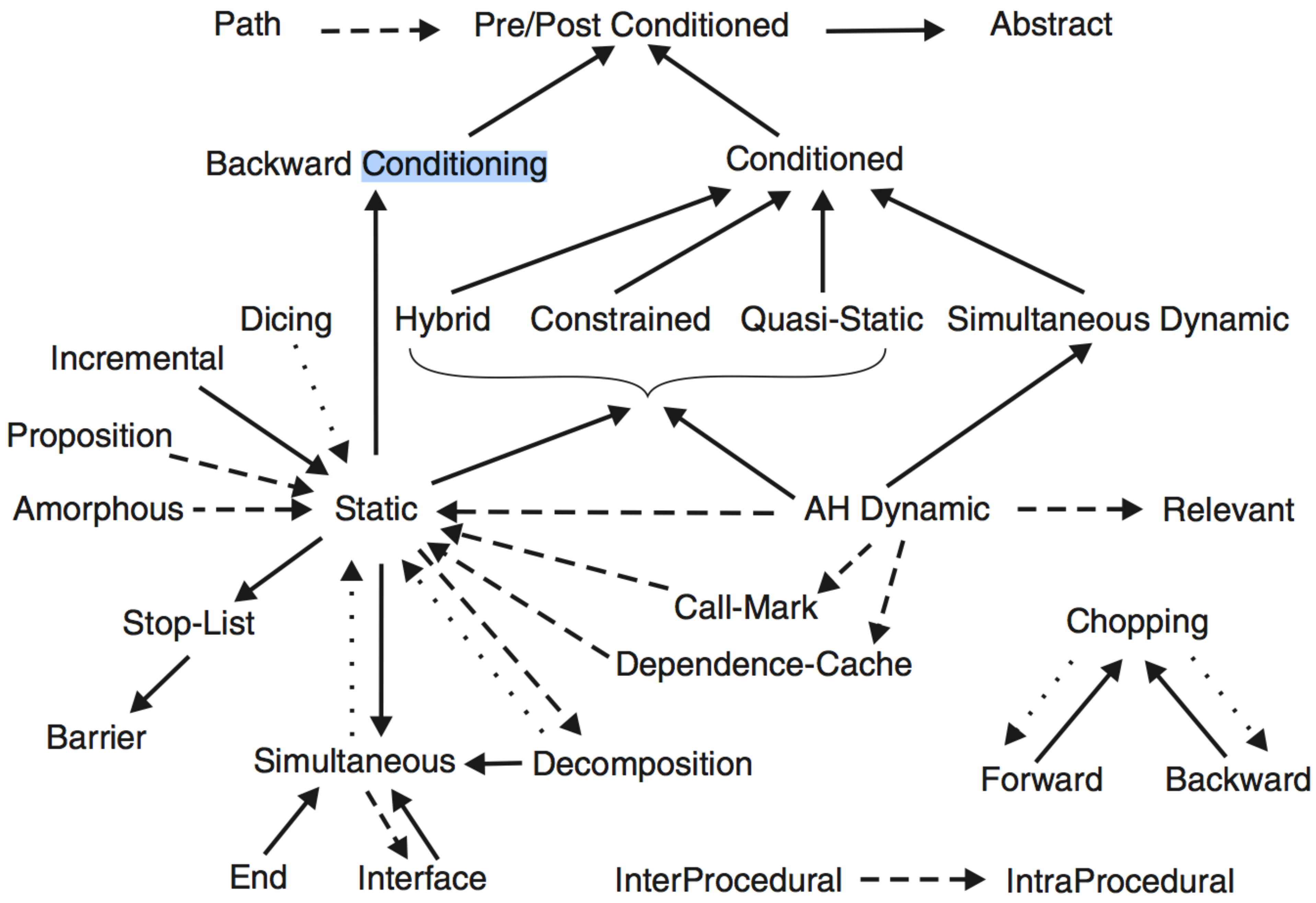
96 - Prolog

96 - VHDL

97 - Amorphous Slicing

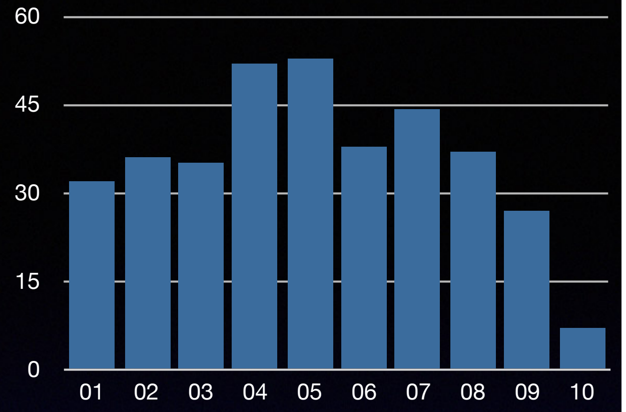
98 - Conditioned Slicing

98 - State Machines



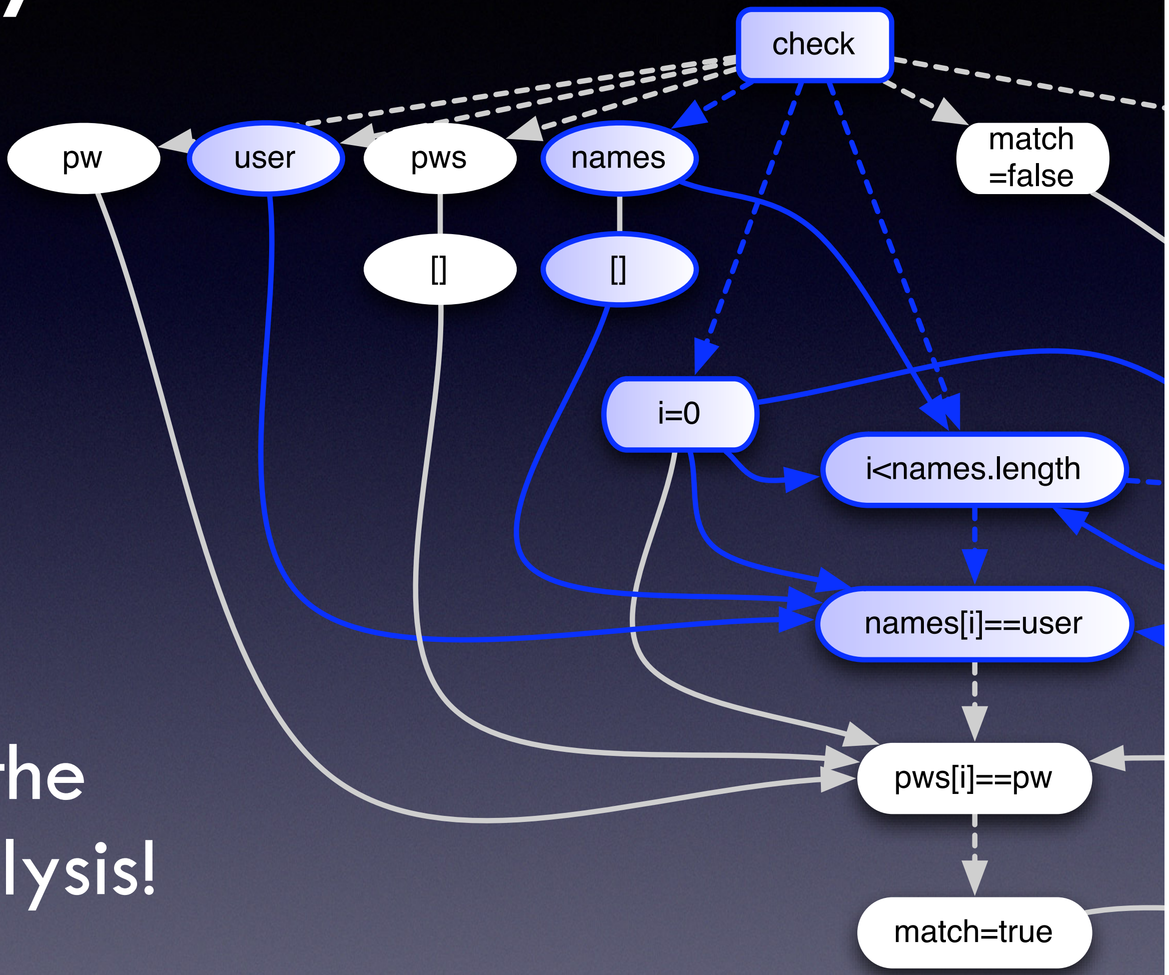
J. Silva. A vocabulary of program slicing-based techniques.
ACM Computing Surveys, 2011

Stable 10 years



- Improvements in precision, efficiency, applications, usability, applicability, ...
- Empirical studies
- Tool(s): CodeSurfer and some prototypes (Kaveri, JSlice, Sprite, Unravel, Frama-c, WET, WALA, LLVM, Joana, JavaSlicer,...)

Slicing is easy.

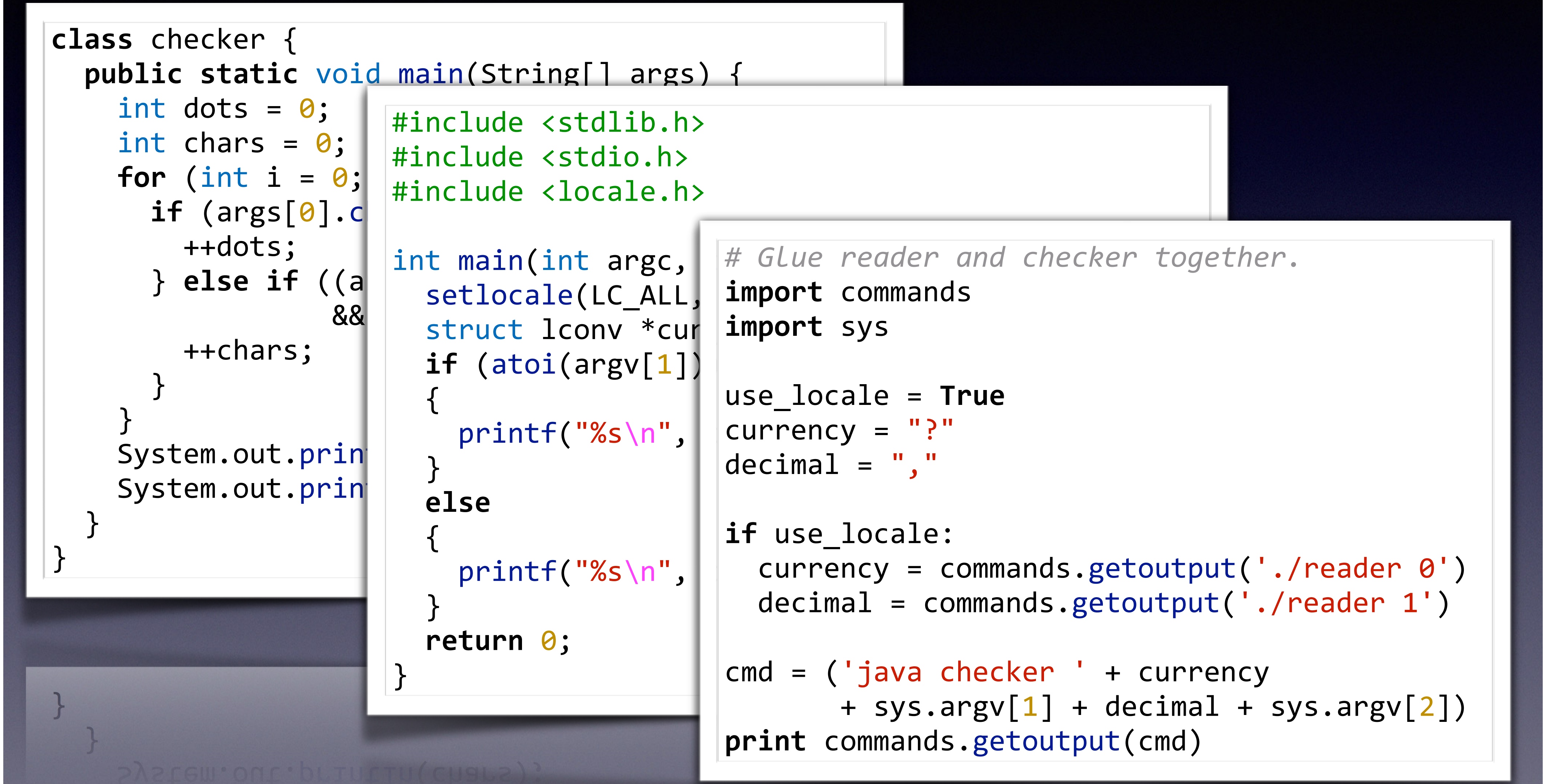


- Slicing is just a traversal of dependences.
- The hard part is the Dependence Analysis!
- Not to mention the Pointer Analysis...

Challenges

- Almost no advances in the past 10 years!
- Tools cannot handle real world software:
  - Exhaustive analyses are impossible, source code is not available or compilable.
  - Systems programmed in various languages, including scripting and configurations.

Who can slice this?



```
class checker {
  public static void main(String[] args) {
    int dots = 0;
    int chars = 0;
    for (int i = 0;
      if (args[0].c
        ++dots;
      } else if ((a
                 &&
        ++chars;
      }
    }
    System.out.prin
    System.out.prin
  }
}
```

```
#include <stdlib.h>
#include <stdio.h>
#include <locale.h>

int main(int argc,
  setlocale(LC_ALL,
  struct lconv *cur
  if (atoi(argv[1])
  {
    printf("%s\n",
  }
  else
  {
    printf("%s\n",
  }
  return 0;
}
```

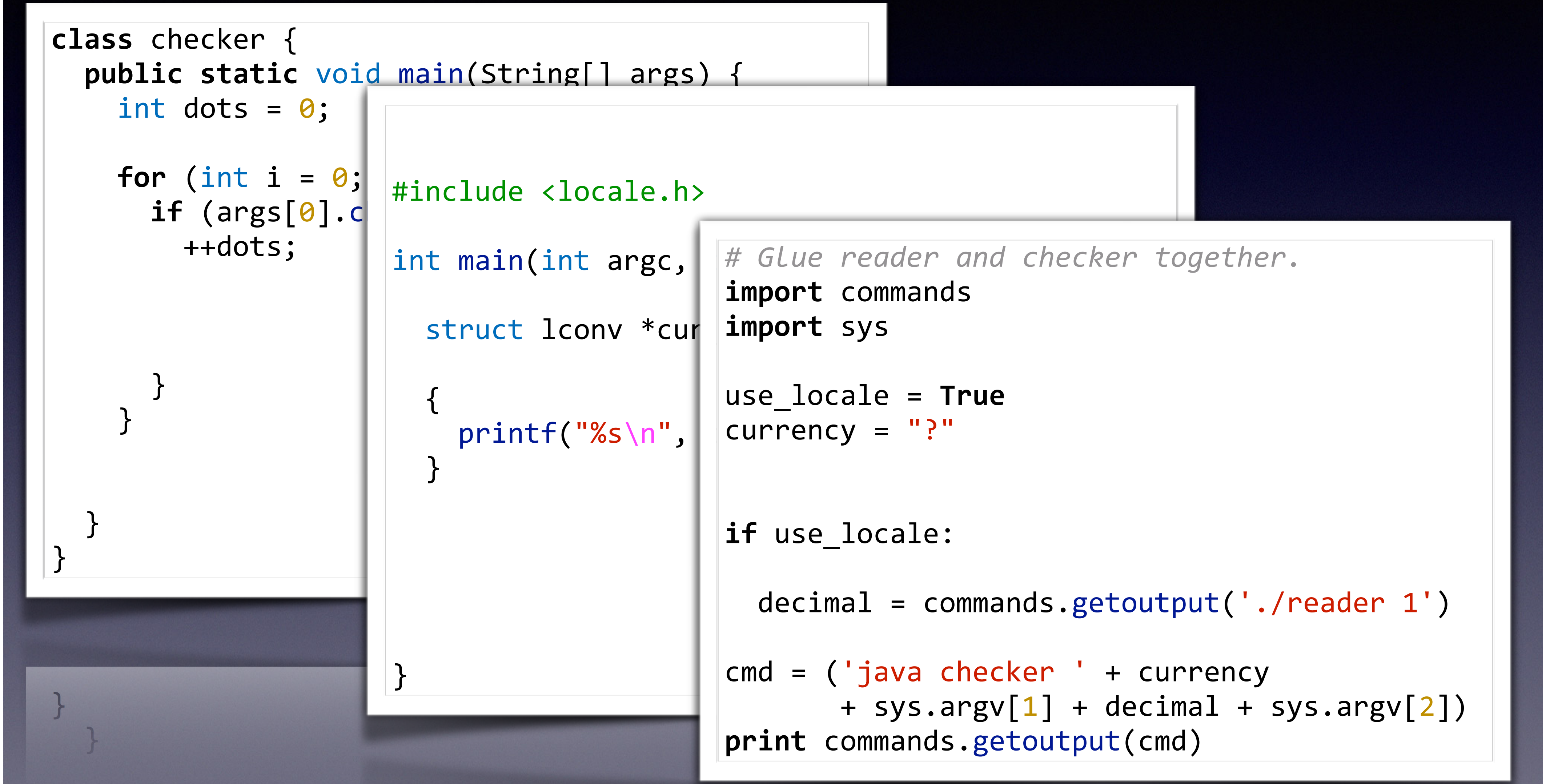
```
# Glue reader and checker together.
import commands
import sys

use_locale = True
currency = "?"
decimal = ","

if use_locale:
  currency = commands.getoutput('./reader 0')
  decimal = commands.getoutput('./reader 1')

cmd = ('java checker ' + currency
       + sys.argv[1] + decimal + sys.argv[2])
print commands.getoutput(cmd)
```

Yes, we can!



```
class checker {
  public static void main(String[] args) {
    int dots = 0;

    for (int i = 0;
      if (args[0].c
        ++dots;




      }
    }



  }
}
```

```
#include <locale.h>

int main(int argc,

  struct lconv *cur

  {
    printf("%s\n",
  }






}
```

```
# Glue reader and checker together.
import commands
import sys

use_locale = True
currency = "?"


if use_locale:

  decimal = commands.getoutput('./reader 1')

cmd = ('java checker ' + currency
       + sys.argv[1] + decimal + sys.argv[2])
print commands.getoutput(cmd)
```

Slicing

A slice *S* of program *P* on slicing criterion C
is any executable program with:

1. *S* can be obtained from *P*
   by deleting zero or more statements from *P*.

2. Whenever *P* halts on input *i*
   with state trajectory *T*,
   then *S* also halts on input *i* with state trajectory *T*′,
   and $\mathrm{PROJ}_C(T) = \mathrm{PROJ}_C(T')$, where PROJ_C is the
   projection function associated with criterion C.

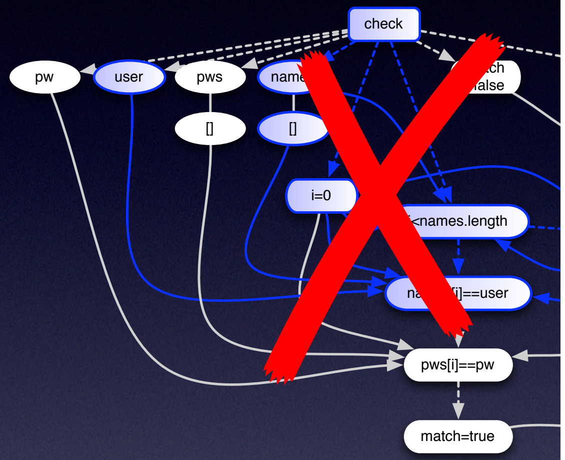
Dynamic Slicing

A dynamic slice *S* of program *P* on slicing criterion C **for inputs *I*** is any executable program with:

1. *S* can be obtained from *P*
   by deleting zero or more statements from *P*.

2. Whenever *P* halts on input *i* **from *I***
   with state trajectory *T*,
   then *S* also halts on input *i* with state trajectory *T′*,
   and $\mathrm{PROJ}_C(T) = \mathrm{PROJ}_C(T')$, where PROJ_C is the projection function associated with criterion C.

Our approach: Observation-based Slicing



- delete statements
- execute the candidate slice
- observe the behaviour for a given criterion
- accept deletion if behaviour is unchanged
- repeat until no statement can be deleted

ORBS

- is language independent
- manipulates files,
  builds and executes the system as usual
- comes in a plain iterative version
  and a delta debugging version
- creates correct and executable slices
  (by construction)

Similar approaches

- Critical Slicing (DeMillo et al, 1996):
  A critical slice contains all statements
  that cannot be independently deleted.
- STRIPE (Cleve and Zeller, 2000):
  Uses delta debugging to remove statements
  from an execution trace using a debugger.
- Both may produce invalid slices!

Example (a=10)

```
W O C D S E int main(int argc, char **argv) {
  O C   S E   int a;
    C     E   int z;
W O C   S E   int x;
W O C   S E   int j;
        S E   a = atoi(argv[1]);
      D S E   x = 0;
W O C D S E   j = 5;
    C   S E   a = a - 10;
  O C   S E   if (a > j) {
        S       x = x + 1;
        S     } else {
          E     z = 0;
  O C   S E   }
W O C D S E   x = x + j;
          E   printf("%d\n", x);
          E   return 0;
W O C D S E }
```

Empirical Evaluation (small programs)

- 13 test programs, 8 languages, 41 criteria
- ORBS is feasible
- delta debugging is more expensive than the plain iterative version
- different versions create different results
- critical slicing needs fewest executions, but produces invalid slices

Case Study: bash

- 1153 files
- 118,167 SLOC
- 8 different languages
- includes generated source code
- contains libraries

Criterion

- Variable ‘val’ at line 1393 in ‘expr.c’ (result of converting a string to an int)
- Test cases ‘arith.tests’ are used as inputs (executes the arithmetic functions)
- Criterion is executed 80,425 times (i.e. 80,425 elements in the trajectory)

Scenario 1

Files to be sliced:

- variables.c (variables are used in tests)
- parse.y (defines input format)

Results:

- 9,417 of 10,804 lines are deleted
- 42,793 compilations, 5,370 executions
- slice size: 13% (17% SLOC)
- only 88 lines of 849 grammar lines are left
  8 rules have been removed completely

Scenario 2

Only the first 100 elements of the trajectory are compared.

Small changes in the results:

- 510 more lines are deleted
- 7846 fewer compilations
- 1008 fewer executions

Scenario 3

A third file is to be sliced: ‘lib/glob/glob.c’

- part of a library, used as a binary component
- nothing in it is actually executed

Results

- Only 6 out of 1100 are left
- 1865 more compilations, 510 more executions

Scenario 4

A fourth file is to be sliced: ‘subst.c’,

- the largest single source file within bash
- 9392 lines

Results

- 665 out of 9392 lines are left
- 19,758 out of 21,296 are deleted
- 10 additional lines in parse.y are deleted
- 29,590 more compilations, 4137 more executions

External Factors

- Order of files
- Source code layout
- Environment
  - Operating systems
  - Tool set (gcc vs. llvm)
  - Build configuration (optimisation, profiling)

ORBS
(Observation-based Slicing)

- Uses deletion–execution–observation
- Generates correct and executable slices
- Slices systems built using multiple languages, including libraries and binary components
- Produces significantly smaller slices