CSE584: Software Engineering

Lecture 9: Tools & Analysis (B)

David Notkin
Computer Science & Engineering
University of Washington
http://www.cs.washington.edu/education/courses/584/

Lackwit (O'Callahan & Jackson)

- **Code-oriented tool that exploits type inference**
- **Answers queries about C programs**
  - **e.g., "locate all potential assignments to this field"**
  - **Accounts for aliasing, calls through function pointers, type casts**
- **Efficient**
  - **e.g., answers queries about a Linux kernel (157KLOC) in under 10 minutes on a PC**

Placement

- **Lexical tools are very general, but are often imprecise because they have no knowledge of the underlying programming language**
- **Syntactic tools have some knowledge of the language, are harder to implement, but can give more precise answers**
- **Semantic tools have deeper knowledge of the language, but generally don't scale, don't work on real languages and are hard to implement**

Lackwit

- **Semantic**
- **Scalable**
- **Real language (C)**
- **Static**
- **Can work on incomplete programs**
  - **Make assumptions about missing code, or supply stubs**

•Sample queries

- **–Which integer variables contain file handles?**
- **–Can pointer `foo` in function `bar` be passed to `free()`? If so, what paths in the call graph are involved?**
- **–Field `f` of variable `v` has an incorrect value; where in the source might it have changed?**
- **–Which functions modify the `cur_veh` field of `map_manager_global`?**

Lackwit analysis

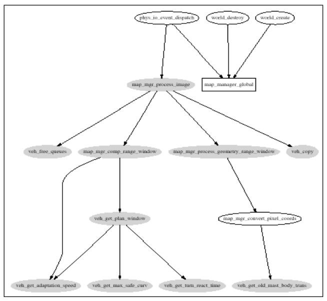
- **Approximate (may return false positives)**
- **Conservative (may not return false negatives) under some conditions**
  - **C's type system has holes**
  - **Lackwit makes assumptions similar to those made by programmers (e.g., "no out-of-bounds memory accesses")**
  - **Lackwit is unsound only for programs that don't satisfy these assumptions**

Query commonalities

- **There are a huge number of names for storage locations**
  - **local and global variables; procedure parameters; for records, etc., the sub-components**
- **Values flow from location to location, which can be associated with many different names**
- ***Archetypal query*: Which other names identify locations to which a value could flow to or from a location with this given name?**
  - **Answers can be given textually or graphically**

An example

- Query about the `cur_veh` field of `map_manager_global`
- Shaded ovals are functions extracting fields from the global
- Unshaded ovals pass pointers to the structure but don't manipulate it
- Edges between ovals are calls
- Rectangles are globals
- Edges to rectangles are variable accesses



Claim

- This graph shows which functions would have to be checked when changing the invariants of the current vehicle object
  - Requires semantics, since many of the relationships are induced by aliasing over pointers

Underlying technique

- Use type inference, allowing type information to be exploited to reduce information about values flowing to locations (and thus names)
- But what to do in programming languages without rich type systems?

Trivial example

- `DollarAmt getSalary(EmployeeNum e)`
- Relatively standard declaration
- Allows us to determine that there is no way for the value of `e` to flow to the result of the function
  - Because they have different types

- `int getSalary(int e)`
- Another, perhaps more common, way to declare the same function
- This doesn't allow the direct inference that `e`'s value doesn't flow to the function return
  - Because they have the same type
- Demands type inference mechanism for precision

Lackwit's type system

- Lackwit ignores the C type declarations
- Computes new types in a richer type system

```
char* strcpy(char* dest,char* source)
```

$(\mathrm{num}^{\alpha}\ \mathrm{ref}^{\beta}, \mathrm{num}^{\alpha}\ \mathrm{ref}^{\gamma}) \rightarrow^{\phi} \mathrm{num}^{\alpha}\ \mathrm{ref}^{\beta}$

Implies

- Result may be aliased with `dest` (flow between pointers)
- Values may flow between the characters of the parameters
- No flow between `source` and `dest` arguments (no aliasing)

Incomplete type information

```
void* return1st(void* x,void* y)
  {
      return x; }
```

$(a\ \mathrm{ref}^{\beta}, b) \rightarrow^{\phi} a\ \mathrm{ref}^{\beta}$

- The type variable a indicates that the type of the contents of the pointer `x` is unconstrained
  - But it must be the same as the type of the contents of pointer `y`
- Increases the set of queries that Lackwit can answer with precision

Polymorphism

- ```
  char* ptr1;
  struct timeval* ptr2;
  char** ptr3;
  …
  return1st(ptr1,ptr2); return1st(ptr2,ptr3)
  ```
- Both calls match the previous function declaration
- This is solved (basically) by giving `return1st` a richer type and instantiating it at every call site
  - $(c \; \mathrm{ref}^{\beta}, d) \rightarrow^{\delta} c \; \mathrm{ref}^{\beta}$
  - $(e \; \mathrm{ref}^{\alpha}, f) \rightarrow^{\chi} e \; \mathrm{ref}^{\alpha}$

Type stuff

- Modified form of Hindley-Milner algorithm "W"
- Efforts made to handle
  - Mutable types
  - Recursive types
  - Null pointers
  - Uninitialized data
  - Type casts
  - Declaration order

```
void copy(char * from, char * to) {
    *to = *from;
}
void copy5(char * fromarray, char * toarray) {
    int i;
    for (i = 0; i < 5; i++) {
        copy(from + i, to + i);
    }
}
void main(void) {
    char from1[5] = { 'h', 'e', 'l', 'l', 'o' };
    char to1[5];
    char from2[5] = { 'k', 'i', 't', 't', 'y' };
    char to2[5];
    copy5(from1, to1);
    copy5(from2, to2);
```

copy	$\forall\alpha.\forall\beta.\forall\phi.(\mathrm{num}^{\alpha}\ \mathrm{ref}^{\beta}, \mathrm{num}^{\alpha}\ \mathrm{ref}^{\chi}) \rightarrow^{\phi} ()$
copy5	$\forall\delta.\forall\varphi.\forall\sigma.(\mathrm{num}^{\delta}\ \mathrm{ref}^{\varphi}, \mathrm{num}^{\delta}\ \mathrm{ref}^{\kappa}) \rightarrow^{\sigma} ()$
main:from1	$\mathrm{num}^{\theta}\ \mathrm{ref}^{\rho}$
main:to1	$\mathrm{num}^{\theta}\ \mathrm{ref}^{\eta}$
main:from2	$\mathrm{num}^{\mu}\ \mathrm{ref}^{\zeta}$
main:to2	$\mathrm{num}^{\mu}\ \mathrm{ref}^{\gamma}$

`*from1` is not compatible with either `*from2` or `*to2`

But it is with `copy:*from`, `copy:*to`, `copy5:*from` + `copy5:*to`

Morphin case study

- Robot control program of about 17KLOC
- Vehicle object contains two queue objects
  - Client was investigating combining these two queues into one
- Queried each queue object to discover operations performed and their contexts
- The two graphs each contained 171 nodes
  - But each graph had only five nodes highlighted as "accessor" nodes

Example

- These five matches helped identify code to be changed
- grep would have returned false matches and missed matches when parameters were passed to functions
- Context-sensitivity needed to distinguish the two queue onjects
  - Because both are passed as arguments to the same queue functions

Recap

- Helps find relationships among variables in a C program
  - Exploits type inference to understand values flowing to locations and thus names
- Approximate, although safe under many (most?) conditions
- Reasonably efficient
  - Although I didn't show the numbers, they are now better than reported in the ICSE paper
```

Program invariants

- One way to try to manage the complexity of software systems is to use program invariants
- Invariants can aid in the development of correct programs
  - The invariants are defined explicitly as part of the construction of the program [Dijkstra][Hoare][Gries][…]

Invariants and evolution

- Invariants can aid in the evolution of software as well
- In particular, programmers can easily make changes that violate unstated invariants
  - The violated invariants are often far from the site of the change
  - These changes can cause errors
  - The presence of invariants can reduce the number of or cost of finding these violations

Other uses for invariants

- Documenting code
- Checking assumptions: convert to `assert`
- Locating unusual conditions
- Providing hints for higher-level profile-directed compilation [Calder]
- Bootstrapping proofs [Wegbreit][Bensalem]
- …

However…

Despite all the potential benefits of having invariants in programs, there are relatively few invariants found in actual programs

There is no obvious reason to believe that this will change due to more research results and more education

Today's focus

- *An approach to make invariants more prevalent and more practical*
- Underlying assumption:
  - The presence of invariants will reduce the difficulty and cost of evolution
- Goal: recover invariants from programs
- Technique: run the program, examine values
- Artifact: Daikon

Goal: Recover invariants

- Detect invariants such as those found in assert statements or specifications
  - *x > abs(y)*
  - *x = 16*y + 4*z + 3*
  - *array a contains no duplicates*
  - *for each node n, n = n.child.parent*
  - *graph g is acyclic*
  - *…*

Experiment 1 [Gries 81]: *Recover formal specifications*

```
// Sum array b of length n into
// variable s
i := 0; s := 0;
while i ≠ n do
  { s := s+b[i];  i := i+1 }
```

Precondition: $n \geq 0$
Postcondition: $S = \Sigma_{0 \leq j < n} b[j]$
Loop invariant:
$0 \leq i \leq n$ and $S = \Sigma_{0 \leq j < i} b[j]$

Test suite

- **100 randomly-generated arrays**
  - length uniformly distributed from 7 to 13
  - elements uniformly distributed from –100 to 100
- **First guess for a test suite**
  - Turned out to work well
  - More on test suites later on

Inferred invariants

♦ *Formal specification (goal)*

```
ENTRY:
   N = size(B)
   N in [7..13] ♦
   B: All elements in [-100..100]
EXIT:
   N = I = orig(N) = size(B)
   B = orig(B)
   S = sum(B) ♦
   N in [7..13]
   B: All elements in [-100..100]
```

Inferred loop invariants

```
LOOP:
   N = size(B)
   S = sum(B[0..I-1]) ♦
   N in [7..13]
   I in [0..13] ♦
   I <= N ♦
   B: All elements in [-100..100]
   B[0..I-1]: All elements in [-100..100]
```

Experiment 2: *Code without explicit invariants*

- **563-line C program: regular expression search & replace [Hutchins][Rothermel]**
- **Task: modify to add Kleene +**
- **Complementary use of both detected invariants and traditional tools (such as grep)**

Programmer use of invariants

- **Helped explain use of data structures**
  - regexp compiled form (a string)
- **Contradicted some maintainer expectations**
  - anticipated *lj* < *j* in `makepat`
  - queried for counterexample
  - avoided introducing a bug
- **Revealed a bug**
  - when *lastj* = **j* in `stclose`, array bounds error

More invariant uses

- Showed procedures used in limited ways
  - `makepat`
    start = 0 and *delim = '\0'*
- Demonstrated test suite inadequacy
  - *#calls(in_set_2) = #calls(stclose)*
- Changes in invariants validated program changes
  - `stclose`: **j = orig(*j)+1*
  - `plclose`: **j ≥ orig(*j)+2*

Experiment 2 conclusions

- Invariants
  - effectively summarize value data
  - support programmer's own inferences
  - lead programmers to think in terms of invariants
  - provide serendipitous information
- Additional useful components of Daikon
  - trace database (supports queries)
  - invariant differencer

Other experiments

Students
- UW CSE 142 (C, small)
- MIT 6.170 (Java, ≤ 5000 lines)

Testing research
- Hoffman (Java, 2000 lines)
- Siemens (C, ~500 lines)

Program checkers
- Xi (Java, small)
- ESC (Java, 500 lines)

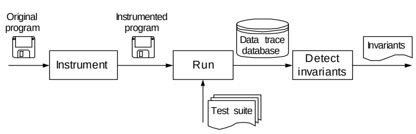
Textbooks
- Gries (Lisp, tiny)
- Weiss (Java, small)
- Java in a Nutshell (Java, ≤ 300 lines)

Medic planner (Lisp, 13,000 lines)

Ways to obtain invariants

- Programmer-supplied
- Static analysis: examine the program text [Cousot][Gannod]
  - properties are guaranteed to be true
  - pointers are intractable in practice
- Dynamic analysis: run the program
  - complementary to static techniques

Dynamic invariant detection



- Look for patterns in values the program computes
  - Instrument the program to write data trace files
  - Run the program on a test suite
  - Invariant engine reads data traces, generates potential invariants, and checks them
- Roughly, machine learning over program traces

Daikon implementation

- 20,000 lines of Java
  - Plus front ends, utilities
  - Daikon 1: 6,500 lines of Python
- Front ends: C/C++, Java, Lisp
- Builds on others' programs and libraries
  - Lackwit, Ajax, EDG, Jikes
  - (JOIE, Bobby, JavaClass, javac, JavaCC)
- Prototype available from Ernst

Running the program

- Requires a test suite
  - Standard test suites are adequate
  - Relatively insensitive to test suite (if large enough)
- No guarantee of completeness or soundness
  - Useful nonetheless (cf. Purify, ESC, PREfix)
  - Complementary to other techniques and tools

Sample invariants

- *x,y,z* are variables; *a,b,c* are constants
- Invariants over numbers
  - unary: $x = a$, $a \leq x \leq b$, $x \equiv a (\text{mod } b)$, …
  - n-ary: $x \leq y$, $x = ay + bz + c$, $x = max(y, z)$, …
- Invariants over sequences
  - unary: sorted, invariants over all elements
  - with sequence: subsequence, ordering
  - with scalar: membership
- Why these invariants?

Checking invariants

- For each potential invariant:
  - Instantiate
    - That is, determine constants like *a* and *b* in $y = ax + b$
  - Check for each set of variable values
  - Stop checking when falsified
- This is inexpensive
  - Many invariants, but each cheap to check
  - Falsification usually happens very early

Performance: runtime growth

- Cubic in number of variables at a program point
  - Linear in number of invariants checked/discovered
- Linear in number of samples (test suite size)
- Linear in number of instrumented program points

Absolute runtime

- A few minutes per “average” procedure
  - 10,000 calls
  - 70 variables
  - Instrument entry and exit
- Unoptimized prototype

Relevance

- Our first concern in this research was whether we could find *any* invariants of interest
- When we found we could, we found a different problem
  - We found many invariants of interest
  - But *most* invariants we found were not relevant

Improved invariant relevance

- Add desired invariants
  - Implicit values
  - Unused polymorphism
- Eliminate undesired invariants (and improve performance)
  - Unjustified properties
  - Redundant invariants
  - Incomparable variables

1. Implicit values

Find relationships over non-variables

- array: *length, sum, min, max*
- array and scalar: element at index, subarray
- number of calls to a procedure
- …

Derived variables

- Successfully produces desired invariants
- Adds many new variables
  - slowdown
  - irrelevant invariants
- Staged derivation and invariant inference
  - avoid deriving meaningless values
  - avoid computing tautological invariants

2. Unused polymorphism

- Variables declared with general type, used with more specific type
  - Ex: given a generic list that contains only integers, report that the contents are sorted
- Also applicable to subtype polymorphism

Unused polymorphism example

```
class MyInteger { int value; … }
class Link { Object element; Link next; … }
class List { Link header; … }
List myList = new List();
for (int i=0; i<10; i++)
  myList.add(new MyInteger(i));
```

Desired invariant in class `List`

```
header.closure(next).element.value:
  sorted by ≤
```

Polymorphism elimination

- Pass 1: front end outputs object ID, runtime type, and all known fields
- Pass 2: given refined type, front end outputs more fields

- Effective for programs tested so far
- Sound for deterministic programs

3. Unjustified properties

Given three samples for `x`:

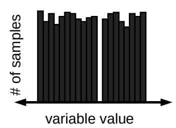
```
x = 7
x = -42
x = 22
```

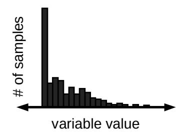
Potential invariants:

```
x ≠ 0
x ≤ 22
x ≥ -42
```

Statistical checks: *check hypothesized distribution*

- **Probability of no zeroes (to show x ≠ 0) for v values of x in range of size r** $\left(1-\frac{1}{r}\right)^{v}$
- **Range limits (e.g., x ≤ 22)**
  - **same number of samples as neighbors (uniform)**
  - **more samples than neighbors (clipped)**





Duplicate values

- **Array sum program:**

```
i := 0; s := 0;
while i ≠ n do
  { s := s+b[i];  i := i+1 }
```

- ***b* is unchanged inside loop**
- **Problem: at loop head**
  –88 ≤ b[n – 1] ≤ 99
  –556 ≤ sum(b) ≤ 539
- **Reason: more samples inside loop**

Disregard duplicate values

- **Idea: count a value only if its var was just modified**
- **Front end outputs modification bit per value**
  - **compared techniques for eliminating duplicates**
- **Result: eliminates undesired invariants**

4. Redundant invariants

- **Given**
  0 ≤ i ≤ j
- **Redundant**
  a[i] ∈ a[0..j]
  max(a[0..i]) ≤ max(a[0..j])
- **Redundant invariants are logically implied**
- **Implementation contains many such tests**

Suppress redundancies

- **Avoid deriving variables: suppress 25-50%**
  - **equal to another variable**
  - **nonsensical**
- **Avoid checking invariants:**
  - **false invariants: trivial improvement**
  - **true invariants: suppress 90%**
- **Avoid reporting trivial invariants: suppress 25%**

5. Unrelated variables

```
bool p;
int *p;
```

b < p

```
int myweight, mybirthyear;
```

myweight < mybirthyear

Limit comparisons

- **Check relations only over comparable variables**
  - **declared program types**
  - **Lackwit [O'Callahan]**

Comparability results

- **Comparisons:**
  - **declared types: 60% as many comparisons**
  - **Lackwit: 5% as many comparisons; scales well**
- **Runtime: 40-70% improvement**
- **Few differences in reported invariants**

Richer types of invariant

Object/class invariants

```
node.left.value < node.right.value
string.data[string.length] = '\0'
```

Pointers (recursive data structures)

```
tree is sorted
```

Conditionals

```
if  proc.priority < 0 then
      proc.status = active
ptr = null  or  *ptr > i
```

Pointer experiment

- **Data structures from Weiss's *Data Structures and Algorithm Analysis in Java***
- **Identified goal invariants by reading book**
- **Added linearization and data splitting to Daikon**
- **Results**
  - **90-100% of goal invariants**
  - **few extraneous invariants**

Object invariant

```
class LinkedList { Link header; … }
class Link { int element; Link next; …
  }
```

Object invariant:

```
header ≠ null
header.element = 0
size(header.closure(next)) ≥ 1
```

Conditional pointer invariant

At exit of
`LinkedList.insert(Object x, LinkedListItr p)`

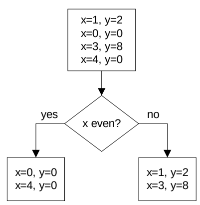
```
if (p ≠ null and p.current ≠ null) then
  size(header.closure(next)) =
    size(orig(header.closure(next))) + 1
else
  header.closure(next)) =
    orig(header.closure(next))
```

Linearize data structures

- Traverse pointer-directed data structures
- Present to invariant engine as sequence
  - cyclicity determined by front end

Conditionals: mechanism

1. Split the data into parts
2. Compute invariants over each subset of data
3. Compare results, produce implications




```
if even(x)then
  y = 0
else
  y = 2x
```

Data splitting criteria

- Static analysis ♦
- Distinguished values: zero, source literals, mode, outliers, extrema
- Exceptions to detected invariants
- User-selected
- Exhaustive over random sample

Scaling

- Technology
  - many program points
  - large data structures
  - solution: next slide
- Utility
  - many program points
  - different invariants
  - different uses
  - solution: experiments, case studies

Incremental inference

- Online algorithm improves
  - response time
  - space
  - front end computation
  - back end computation
- Process each variable value once, then discard
- Stop checking invariants after falsification
- To do: selectively disable instrumentation

Summary

- Dynamic invariant detection is feasible
  - Conceived and developed the idea
  - Prototype implementation
- Dynamic invariant detection is accurate & useful
  - Techniques to improve basic approach
  - Experiments provide preliminary support
- Dynamic invariant detection is a challenging and promising area for research and practice
  - Practical? Not yet clear!

Path Profiling: Ball and Larus

```
#include <stdio.h>
main(t,_,a)
char *a;
{
return!0<t?t<3?main(-79,-13,a+main(-87,1-_,main(-86,0,a+1)+a)):
1,t<_?main(t+1,_,a):3,main(-94,-27+t,a)&&t==2?_<13?
main(2,_+1,"%s %d %d\n"):9:16:t<0?t<-72?main(_,t,
"@n'+,#'/*{}w+/w#cdnr/+,{}r/*de}+,/*{*+,/w{%+,/w#q#n+,/#{l+,/n{n+,/+#n+,/#\
;#q#n+,/+k#;*+,/'r :'d*'3,}{w+K w'K:'+}e#';dq#'l \
q#'+d'K#!/+k#;q#'r}eKK#}w'r}eKK{nl]'/#;#q#n'){)#}w'){){nl]'/+#n';d}rw' i;#\
){nl]!/n{n#'; r{#w'r nc{nl]'/#{l,+'K {rw' iK{;[{nl]'/w#q#n'wk nw' \
iwk{KK{nl]!/w{%'l##w#' i; :{nl]'/*{q#'ld;r'}{nlwb!/*de}'c \
;;{nl'-{}rw]'/+,}##'*}#nc,',#nw]'/+kd'+e}+;#'rdq#w! nr'/ ') }+}{rl#'{n' ')# \
}'+}##(!!/")
:t<-50?_==*a?putchar(31[a]):main(-65,_,a+1):main((*a=='/')+t,_,a+1)
:0<t?main(2,2,"%s"):*a=='/'||main(0,main(-61,*a,
"!ek;dc i@bK'(q)-[w]*%n+r3#l,{}:\nuwloca-O;m .vpbks,fxntdCeghiry"),a+1);
}
```

What does it do? Run it!

```
On the first day of Christmas my true love gave to me
   a partridge in a pear tree.
On the second day of Christmas my true love gave to me
   two turtle doves
   and a partridge in a pear tree.
On the third day of Christmas my true love gave to me
   three french hens, two turtle doves
   and a partridge in a pear tree.
   ...
```

- But why?
  - http://www.research.microsoft.com/~tball/papers/XmasGift/
  - Reverse engineering the Twelve Days of Christmas

Counting arguments

- The poem takes O(N*N) time to read and O(N*N) space to write
  - N is the number of gifts
- We can derive an exact count of the number of times gifts
- A gift with ordinal value t is mentioned 13-t times in the poem
  - For example, "five gold rings" occurs 13-5=8 times
- Summing over all gifts yields 1+2+...11+12 = 13*6 = 78 total gift mentions
  - 66 mentions of non-partridge gifts

Continuing like this…key numbers are

- 12 days of Christmas (also 11, to catch "off-by-one" cases)
- 26 unique strings
- 66 occurrences of non-partridge-in-a-pear-tree presents
- 114 strings printed
- 2358 characters printed

Pretty printing the program...

```
/* pretty-printed version of twelve days of christmas program */
#include <stdio.h>
main(t,_,a)
char *a;
{
  return
     ((!0) < t )
       ? ((t < 3
            ? main(-79,-13,a+main(-87,1-_,main(-86,0,a+1)+a))
            : 1),

           (t < _
            ? main(t+1,_,a)
            : 3),

           (main(-94,-27+t,a)
            && (t==2
                ? ( _ < 13
                    ? main(2,_+1,"%s %d %d\n")
                    : 9)
                : 16)))

         : (t < 0
            ? (t < -72
                ?
```

```
main(_,t,

    "@n'+,#'/*{}w+/w#cdnr/+,{}r/*de}+,/*{*+,/w{%+,/w#q#n+,/#{l+,/n{n+,/+#n+,
    /#\
;#q#n+,/+k#;*+,/'r :'d*'3,}{w+K w'K:'+}e#';dq#'l \
q#'+d'K#!/+k#;q#'r}eKK#}w'r}eKK{nl]'/#;#q#n'){)#}w'){){nl]'/+#n';d}rw' i;#
    \
){nl]!/n{n#'; r{#w'r nc{nl]'/#{l,+'K {rw' iK{;[{nl]'/w#q#n'wk nw' \
iwk{KK{nl]!/w{%'l##w#' i; :{nl]'/*{q#'ld;r'}{nlwb!/*de}'c \
;;{nl'-{}rw]'/+,}##'*}#nc,',#nw]'/+kd'+e}+;#'rdq#w! nr'/ ') }+}{rl#'{n' ')#
    \
}'+}##(!!/")

              : (t < -50
                 ? (_ == *a
                    ? putchar(31[a])
                    : main(-65,_,a+1))
                 : main((*a=='/')+t,_,a+1)))
           : (0 < t
              ? main (2,2,"%s" )
              : * a=='/'
              || main(0,main(-61,*a,
                             "!ek;dc i@bK'(q)-[w]*%n+r3#l,{}:\nuwloca-O;m
    .vpbks,fxntdCeghiry"),a+1)
              )
           );
}
```

Structure of the program

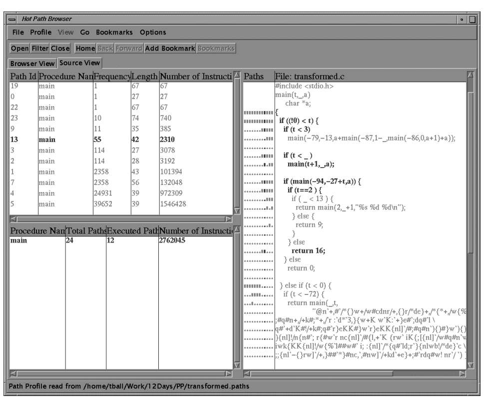
- **After some pretty easy work, the program consists of just `main`**
  - **Calls itself repeatedly**
    - **No loops, only recursion**
  - **No assignments to any variables**
  - **Two large strings appear to encode the text of the poem**

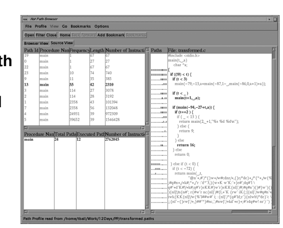
main: three arguments

- **The first argument t is count of the number of arguments on the command line (including the name of the program itself)**
- **The selection of different legs of the function seem to be driven by the parameter t**

Use profiling to extract counts

- **Apply the Hot Path Browser (HPB) tool (Ball, Larus and Rosay)**
  - **Instruments programs to record and display Ball/Larus path profiles**
  - **A Ball/Larus path profile counts how many times each acyclic intraprocedural path executes**





- **The upper left pane shows the statistics about each executed path**
- **12 out of a total of 24 possible paths executed**
- **The paths listed in ascending order of frequency**
- **The path with id 13 has been selected (red line) and highlighted in the source code view**

Path clusters by frequency:
manually identify computational signature

- Path 0 initializes the recursion with the call main(2,2,...)
- Paths 19, 22, and 23 control the printing of the 12 verses
  - Path 19 represents the first verse
  - Path 23 the middle 10 verses
  - Path 22 the last verse
  - The sum of these paths' frequencies is 12
  - The browser can help show that each of the paths covers a different set of recursive calls to main
- Paths 9 and 13 control the printing of the non-partridge-gifts within a verse
  - The frequencies of the two paths sum to 66

More

- Paths 2 and 3 print out a string
  - Each path has frequency 114, the exact number of strings predicted by our model
- Paths 1 and 7 print out the characters in a string
  - Each path executes 2358 times
- Paths 4 and 5 with the large and unusual frequencies of 24931 and 39652?
  - Path 4 skips over n sub-strings in the large string
    - Every time a sub-string is printed, a linear search through the text string is done to find the string
  - Path 5 linearly scans — for each character to be printed — the string that encodes the character translation to find the character that matches the current character to be printed

Jinsight:
http://www.research.ibm.com/jinsight/

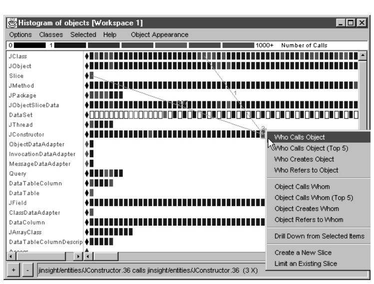
- Tools for analyzing the dynamic behavior of Java programs
  - Visualization
  - Pattern extraction
  - Database query
  - Multidimensional analysis
- Applied to
  - performance analysis
  - memory leak diagnosis
  - debugging
  - program understanding
- A special focus on the analysis of large, complex, data-intensive, and web-based systems

De Pauw, Sevitsky, et al.

Tasks

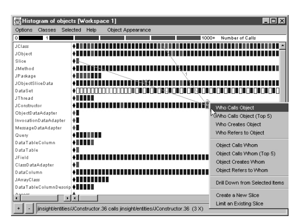
- Visualizations of object usage, garbage collection and the sequence of activity in each thread
- Pattern visualizations extract structure in repetitive calling sequences and complex data structures
  - Analyze large amounts of information in a concise form
- Information exploration
  - Specify filtering criteria
  - Drill down from one view to another to explore details
  - Create units that match features of study
- Measurement
  - Execution activity or memory summarized at any level of detail, along call paths, and along two dimensions simultaneously

Object histogram view:
instances grouped by class, indicating level of activity

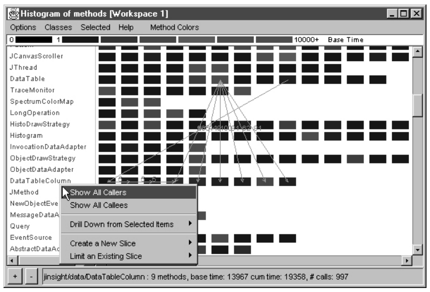


Object histogram view

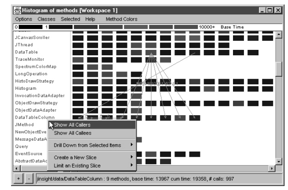
- Class names along the left edge
- Each rectangle denotes an instance of that class or the amount of memory consumed by instances of the class
- A diamond shape denotes the class object for a given class
- A rectangle's color will vary according to a black-to-blue-to-red color spectrum
- Garbage collected objects appear as rectangular outlines



Method histogram view: methods grouped by class

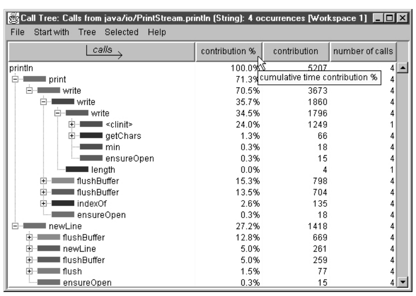


- Class names along the left edge
- Rectangles represent method of the class to its left



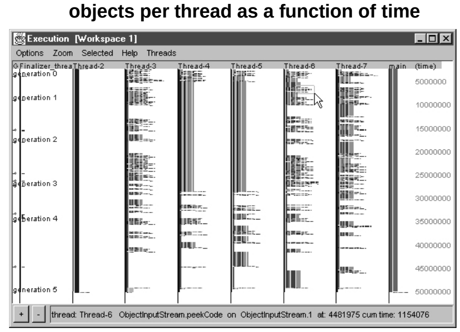
Call tree view:

Summarize call paths from or to a given set of method invocations

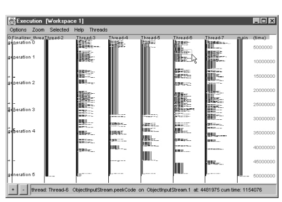


Execution view:

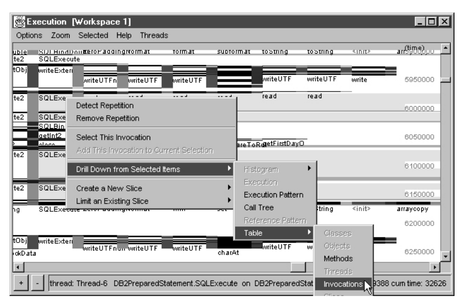
overview and details of communication among objects per thread as a function of time



- Object represented by vertical stripe colored according to the object's class
- Time progresses downward and time units on right
- A stripe's top edge is the time of method call
  - The height reflects total time spent executing the method
- Stripes cascade to the right as methods sends messages
- Stripes grouped in columns by thread
- Leftmost column reserved for garbage collection information

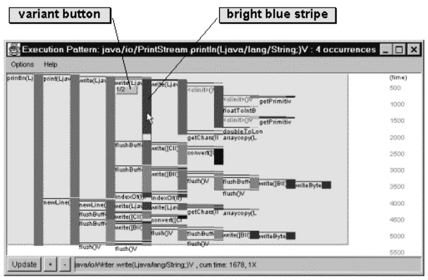


Zoomed in for detail



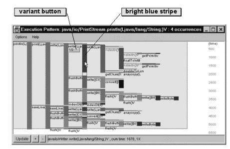
Execution pattern view:

summarizes all invocations of a selected method highlighting the differences

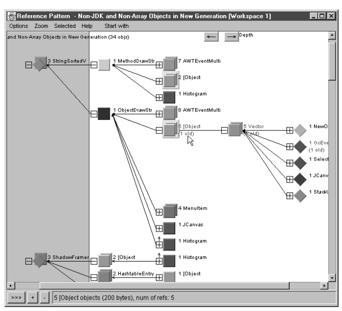


A summary of all the `println` occurrences in the trace

- **Reveals that all `println` messages produce the same pattern of execution except for *one* area of divergence**
- **Mouse the bright blue stripe to identify it as a call to `java/io/Writer.write`.**
  - **–"1X" indicates that this particular call pattern occurred just once**
- **½ in beveled frame indicates there are two variant execution patterns at this point and that pattern 1 is shown**

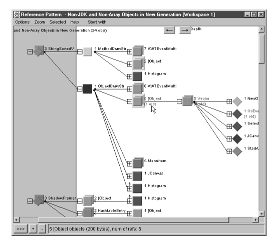


Reference pattern view



Shows patterns of references to or from a set of objects

- **Squares represent objects, each colored uniquely by class**
- **A diamond represents a class object**
- **Single squares denotes a single instance**
- **Twin squares represent multiple instances**
- **Arrows between nodes denote one or more references between instances**
- **An arrow points to the object(s) being referenced**



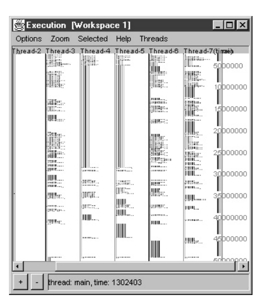
Slices

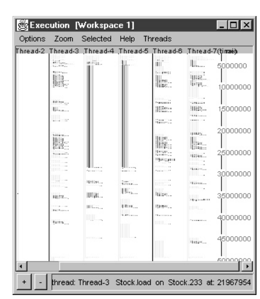
(not Weiser slices)

- **A slice is a subset of the trace information corresponding to a user-selected feature in a program**
  - **Applies to any view**
- **Slices intended to filter out extraneous information, focusing analysis on one area**
- **Slices give you an extra dimension for measuring program execution**
  - **Can compute any measurement about a program relative to any defined slice**
    - **Ex: define slices to represent functional areas of your program; then measure execution time in each thread, method, method invocation, etc. spent in each functional area**

Workspaces:

collections of filterings





Tools

- Static vs. dynamic
  - Complementary
- Finding bugs vs. improving performance
- Program representations
  - Affect precision and performance
- Partial specifications
  - You get some benefit for small cost
- Inference to reduce programmer effort
  - Type, dynamic