CSE401: Miscellaneous

David Notkin
Autumn 2000

Software engineering tools

- A compiler is just one tool that helps in writing software
  - It is, indeed, absolutely necessary in practice
- There are lots of other tools, however, that can help programmers write software
- And many of these are based on techniques similar to or inspired by those found inside compilers

Software tools?

- Editors
  - Any compiler-like knowledge?
- Debuggers
  - Any compiler-like knowledge?
- List some others?

Program slicing

- Mark Weiser developed an idea called program slicing
- The idea is that you could select a program point, and a program slicer (a tool), would compute the subset of the program needed to compute the values in use at that point
- It originally conceived of as a support primarily for debugging, since it would allow you to focus on pertinent parts of the program and to ignore the others

Examples

from Reps/Horwitz

```
P := 3.14;                      P := 3.14;
rad := 3;                       rad := 3;
if DEBUG then rad := 4 fi       if DEBUG then rad := 4 fi
area := P*(rad*rad)             area := P*(rad*rad)
circ := 2*P*rad                 circ := 2*P*rad
output(area);                   output(area);
output(circ);                   output(circ);
```

Backward slice

Forward slice

Computing slices

- Weiser originally defined an approach to computing slices based on iterative data flow
  - Very similar to what we saw last week for determining the liveness of variables
- This approach was later shown to be slightly flawed
- Ottenstein and Ottenstein then formulated slicing as a graph traversal problem over program dependence graphs (PDGs)
  - In essence, a PDG combines control and data flow information
  - The control flow information is represented as control dependences
    - A → B if executing A means that you'll always execute B

PDG example: on white board

Complications

- Slicing across procedures is more complicated
  - If phrased as graph traversal over the "obvious" interprocedural PDG, the slices get very, very big
  - This is because a call enters a procedure's PDG, but the return is handled by traversing all possible callers
- PDGs for large programs are expensive to build, and take lots of space — slicing can be very expensive
- In languages like C, macro preprocessing is a complete pain to handle
- There isn't much data, but there is little evidence that for large program slices are small enough to give real benefits
- There are very few commercial slicers

Testing tools

- Software testing is difficult and costly
- One form of testing is white box testing
  - The actual source code is available
- There are a number of coverage measures
  - Did we execute all the statements?
  - Did we execute all the control flow edges?
  - Did we execute all the control flow paths?
  - Did we exercise all def-use chains?

Program transformation

- Compilers translate from source to target code
- Some tools do a source-to-source translation
- One approach that uses this is for program restructuring
- We are interested in restructuring, since program structure tends to degrade over time
- But people don't often restructure in practice
  - Doesn't make money now, introduces new bugs, decreases understanding, political pressures, who wants to do it, hard to predict lifetime costs & benefits

Griswold's approach

- Griswold developed an approach to meaning-preserving restructuring
- Make a local change
  - The tool finds global, compensating changes that ensure that the meaning of the program is preserved
    - What does it mean for two programs to have the same meaning?
  - If it cannot find these, it aborts the local change

Simple example

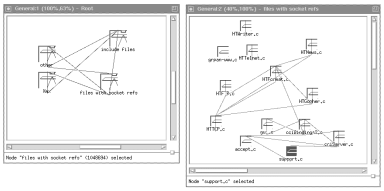
```
procedure push(s, v)
  insert(v, s.head)
  return s
end
 .
 .
 .
push(myStack,1)
 .
 .
 .
push(myStack,h(myStack))
```

- It's not a local change nor a syntactic change
- It requires semantic knowledge about the programming language
- Griswold uses a variant of the sequence-congruence theorem [Yang] for equivalence
  - Based on PDGs
- It's an O(1) tool

Limited power

- The actual tool and approach has limited power
- Indeed, too limited to be useful in practice
  - PDGs are limiting
    - Big and expensive to manipulate
    - Difficult to handle in the face of multiple files, etc.
- May encourage systematic restructuring in some cases

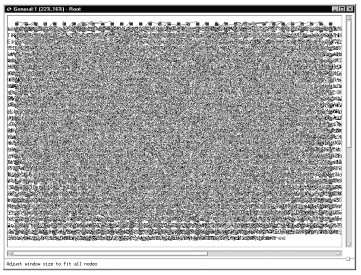
Reverse engineering



- Trying to find good designs in code whose structure has degraded
  - e.g., Rigi, various clustering algorithms (Rigi is used above)

Clustering

- The basic idea is to take one or more source models (e.g., dependence relations) of the code and find appropriate clusters that might indicate "good" modules
- Coupling and cohesion, of various definitions, are at the heart of most clustering approaches
- Many different algorithms

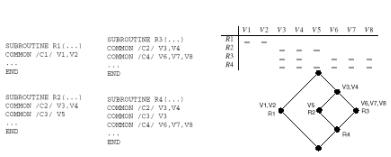


Rigi's approach

- Extract dependence relations (they call them resource relations)
- Build edge-weighted resource flow graphs
  - Discrete sets on the edges, representing the resources that flow from source to sink
- Compose these to represent subsystems
  - Looking for strong cohesion, weak coupling
- The systems uses definitions of interconnection strength and similarity measures (with tunable thresholds)

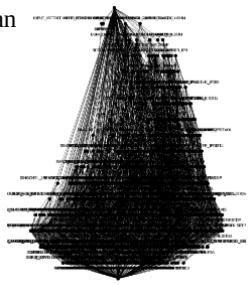
Math. concept analysis

- Define relationships between (for instance) functions and global variables [Snelting et al.]
- Compute a concept lattice capturing the structure
  - "Clean" lattices = nice structure
  - "ugly" ones = bad structure



An aerodynamics program

- 106KLOC Fortran
- 20 years old
- 317 subroutines
- 492 global variables
- 46 COMMON blocks



Recap

- The point of today's lecture is to show that many compiler-like analyses and representations can be used for other kinds of software engineering tools
  - Slicing, testing, reverse/re-engineering, etc.