Automated Program Repair through the Evolution of Assembly Code

Eric Schulte

University of New Mexico

08 August 2010
We present a method of *automated program repair* through the evolution of *assembly code* compiled from *extant software*.

- previous work demonstrated the repair of C programs through evolution of C statements
- we extend previous work by operating at the level of assembly commands
Benefits

Benefits of the assembly code level of representation include:

**General** applicable to any language which compiles to Java byte code or x86 assembly code

**Expressive** the small scale of assembly instructions is capable of expressing repairs not possible at the C statement level

**Coverage** the small alphabet of assembly commands and large number of commands in assembly programs provides access to a larger subset of the space of possible programs

**Robust** the functionality of assembly programs is more robust to mutation
Comparative Performance

Expected Test Suite Runs per Repair

C

assembly

Test Suite Runs

ultrix

nullhttpd

indent

Repair at the Assembly level is roughly 17% slower than at the C level.
Technical Approach

Stages

preprocessing

fault localization

mutation

evolution
Technical Approach

Stages

- preprocessing
- fault localization
- mutation
- evolution

C Statement Tree

Buggy C Program

Assembly Line, Linear Genome, Buggy C, Java, etc... Program
Technical Approach

Stages

- preprocessing
- fault localization
- mutation
- evolution

C Statement Tree

1. Buggy C Program
2. CIL
3. CIL intermediate
Technical Approach

Stages

- preprocessing
- fault localization
- mutation
- evolution

C Statement Tree

- Buggy C Program
- CIL
- CIL intermediate
- walk path
- weighted C statement tree
Technical Approach

Stages

preprocessing

fault localization

mutation

evolution

C Statement Tree

Buggy C Program

CIL

CIL intermediate

walk path

weighted C statement tree

tree mut.

population
Technical Approach

Stages

- preprocessing
- fault localization
- mutation
- evolution

C Statement Tree

1. Buggy C Program
2. CIL
3. CIL intermediate
4. walk path
5. weighted C statement tree
6. tree mut.
7. population
8. mutate
9. evaluate
10. terminate?
Technical Approach

### Stages
- Preprocessing
- Fault Localization
- Mutation
- Evolution

### C Statement Tree
- Buggy C Program
  - CIL
    - CIL Intermediate
      - Walk Path
        - Weighted C Statement Tree
          - Tree Mut.
            - Population
              - Mutate
                - Evaluate
                  - Terminate?

### Assembly Linear Genome
- Buggy C, Java, etc... Program
Technical Approach

Stages
- preprocessing
- fault localization
- mutation
- evolution

C Statement Tree
- Buggy C Program
  - CIL
  - CIL intermediate
  - walk path
  - weighted C statement tree
  - tree mut.
  - population
  - mutate
  - evaluate
  - terminate?

Assembly Linear Genome
- Buggy C, Java, etc... Program
  - gcc -S
  - assembly code
Technical Approach

**Stages**
- preprocessing
- fault localization
- mutation
- evolution

**C Statement Tree**
- Buggy C Program → CIL → CIL intermediate → walk path (deterministic) → weighted C statement tree → tree mut. → population → evaluate → mutate → terminate?

**Assembly Linear Genome**
- Buggy C, Java, etc... Program → gcc -S → assembly code → sample pc (stochastic)
Technical Approach

Stages
- preprocessing
- fault localization
- mutation
- evolution

C Statement Tree
- Buggy C Program
  - CIL
  - CIL intermediate
  - walk path
  - weighted C statement tree
  - tree mut.
  - population
  - mutate
  - evaluate
  - terminate?

Assembly Linear Genome
- Buggy C, Java, etc... Program
  - gcc -S
  - assembly code
  - sample pc
  - weighted assembly genome
Technical Approach

Stages
- preprocessing
- fault localization
- mutation
- evolution

C Statement Tree
- Buggy C Program
  - CIL
  - CIL intermediate
    - walk path
    - weighted C statement tree
      - tree mut.
        - population
          - mutate
          - evaluate
            - terminate?

Assembly Linear Genome
- Buggy C, Java, etc... Program
  - gcc -S
  - assembly code
  - sample pc
  - weighted assembly genome
    - array mut.
      - population
Technical Approach

**Stages**

- preprocessing
- fault localization
- mutation
- evolution

**C Statement Tree**

1. Buggy C Program
2. CIL
3. CIL intermediate
4. walk path
5. weighted C statement tree
6. tree mut.
7. population
8. mutate
9. evaluate
10. terminate?

**Assembly Linear Genome**

1. Buggy C, Java, etc... Program
2. gcc -S
3. assembly code
4. sample pc
5. weighted assembly genome
6. array mut.
7. population
8. mutate
9. evaluate
10. terminate?
Representation and Genetic Operators

Linear Genome

Individuals are *linear genomes* of weighted assembly commands.

("main:" "pushl %ebp" "movl %esp" ... )

Genetic Operators

- **insert** selects an instruction, selects a location, copies the instruction and inserts it in the location
- **delete** selects an instruction and deletes it
- **swap** selects two instructions and swaps them
- **crossover** selects a crossover point for each of two individuals and exchanges all instructions after that point between the individuals
Fault Localization

Technique

CPU: program counter

oprofile → raw addresses

Dump of assembler code for function main:
0x08048414 main+0: push %ebp
0x08048415 main+1: mov %esp,%ebp
0x08048417 main+3: and $0xffffffff0,%esp

gdb → offset in method

main: pushl %ebp
movl %esp, %ebp
andl $-16, %esp

mem-mapping.clj → index in genome

("main:" "pushl %ebp" "movl %esp" ... )
Fault Localization

Results

Raw Sample Counts

Smoothed Weighted Paths

samples

assembly instruction index

weight

assembly instruction index
Fault Localization

Comparative Path Size

Weighted Path Sizes

Weighted Path Length

ultrixuniq  ultrixlook  look  units  ultrix-deroff  nullhttpd  indent

C

assembly
Generality to Multiple Languages

Input: Integer $a$
Input: Integer $b$
Output: $gcd(a, b)$ or $\bot$

1: if $a \equiv 0$ then
2: print $a$
3: end if
4: while $b \neq 0$ do
5: if $a > b$ then
6: $a \leftarrow a - b$
7: else
8: $b \leftarrow b - a$
9: end if
10: end while
11: print $a$

Figure: A Buggy version of Euclid's Algorithm

<table>
<thead>
<tr>
<th></th>
<th>C</th>
<th>Haskell</th>
<th>Java</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program Length</td>
<td>79</td>
<td>885</td>
<td>33</td>
</tr>
<tr>
<td>Total Solutions</td>
<td>2</td>
<td>15</td>
<td>13</td>
</tr>
<tr>
<td>Unique Solutions</td>
<td>2</td>
<td>10</td>
<td>1</td>
</tr>
</tbody>
</table>

Table: GCD Repair Results by Language
Mutational Robustness & Neutral Spaces

Concepts

Mutational Robustness robustness of phenotype to changes in genotype in the case of computer programs

phenotype behavior of the program
genotype representation of the program

Neutral Space contiguous region of genotype space with constant phenotype
Mutational Robustness & Neutral Spaces

Relevance

- Mutational Robustness directly influences evolvability
- A large neutral space allows for significant diversity in
  - representation
  - behavior
  - efficiency
  - size
  - etc.
- A *neutral neighborhood* defines which functionalities are reachable in a single genotypic step
Robustness under Genetic Operators

Robustness of Behavior under Genetic Operations

<table>
<thead>
<tr>
<th>Program</th>
<th>gcd</th>
<th>look</th>
<th>deroft</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insert</td>
<td>70%</td>
<td>50%</td>
<td>80%</td>
</tr>
<tr>
<td>Delete</td>
<td>30%</td>
<td>50%</td>
<td>20%</td>
</tr>
<tr>
<td>Swap</td>
<td>0%</td>
<td>0%</td>
<td>10%</td>
</tr>
</tbody>
</table>
Future Work

Investigate Mutational Robustness

- differences across languages, algorithms and representations
- properties of the neutral spaces of assembly programs

Improve GP Technique

- apply ongoing work on the C statement level
- homologous crossover, steady state mutation, etc...

Non-Repair Evolution

- machine specific optimization
- disruption of mono-culture (N-variant)
Discussion

- Questions
- Comments
- Suggestions