“Phoenix-Based Clone Detection Using Suffix Trees”
and
“Program Behavior Modeling with Event Grammars”

Robert Tairas
Phoenix-Based Clone Detection Using Suffix Trees
Code Clones

- A sequence of statements that are duplicated in multiple locations in a program

Source Code

Cloned Code
Types of Clones

Original code

```c
int main() {
    int x = 1;
    int y = x + 5;
    return y;
}
```

- `int func1() {` (Exact match)
  ```c
  int x = 1;
  int y = x + 5;
  return y;
  }
  ```

- `int func2() {` (Exact match, with only the variable names differing)
  ```c
  int p = 1;
  int q = p + 5;
  return q;
  }
  ```

- `int func3() {` (Near exact match)
  ```c
  int s = 1;
  int t = s + 5;
  s++;
  return t;
  }
  ```

As defined in an experiment comparing existing clone detection techniques at the 1st International Workshop on Detection of Software Clones (02)
Clones in Source Code

- Copy-and-paste parts of code from one location to another
  - The copied code already works correctly
  - No time to be efficient

- Research shows that 5-10% of large scale computer programs are clones (Baxter, 98)
Dominant decomposition: A block of statements that performs a function/concern dominates another block
- The two concerns crosscut each other
- One concern will have to yield to the other
- Related to Aspect Oriented Programming (AOP)
Clones in Source Code

- logging in org.apache.tomcat
  - red shows lines of code that handle logging
  - not in just one place
  - not even in a small number of places
Clone Dilemma

- Maintenance
  - To update code that is cloned will require all clones to be updated
- Restructure/refactor
- Separate into aspects

But first we need to find the clones
Contribution: Automated Clone Detection

- Searches for exact matching function level clones in the Microsoft Phoenix framework utilizing suffix tree structures
What is Phoenix?

- Next-Generation Framework for
  - building Compilers
  - building Software Analysis Tools
- Basis for Microsoft compilers for 10+ years
- Framework for Academic Research & Teaching

More information:
http://research.microsoft.com/phoenix

Note: Contents of this slide courtesy of John Lefor at Microsoft Research
Note: This slide courtesy of John Lefor at Microsoft Research
A suffix tree for a string is a tree where each suffix of the string is represented by a path from the root to a leaf.

In bioinformatics it is used to search for patterns in DNA or protein sequences.

Example: suffix tree for \texttt{abgf$}
Another Suffix Tree Example

Suffix tree for \textit{abcebcf}$\$

12345678

Leaf numbers:
The number indicates the starting position of the suffix from the left of the string.
Another Suffix Tree Example

Suffix tree for \textit{abgf}\$\textit{abgf}#$

Two identical strings (\textit{abgf}) separated by unique terminating characters

Leaf numbers:
The first number indicates the string.
The second number indicates the starting position of the suffix in that string.
Abstract Syntax Tree Nodes

int func1() {
    return x;
}

int func2() {
    return y;
}

Note: Node names are Phoenix-defined.
Remember This?

Suffix tree for \texttt{abgf$abgf#}

For exact function matching, we're looking for suffix tree nodes or edges, where the edges include all the AST nodes of a function.

Leaf numbers:
The first number indicates the function.
The second number indicates the starting position of the suffix in that function.
Phoenix Phases

- Processes are divided into “phases”
- Custom phases can be inserted to perform tasks such as software analysis
- Phases are inserted through “plug-ins” in the form of a library (DLL) module
Clone Detector in Phoenix

- example.c
- C/C++ Front-end
- example.ast
- C#
- csclones.cs
- C#
- csclones.dll
- Phoenix Back-end
- csclones.dll
- Report
Demo
Limitations and Future Work

- Looks only for exact matches
  - Currently working on a process called hybrid dynamic programming, which includes the use of suffix trees (*k*-difference inexact matching)
- Looks only at the function level
  - Enable multiple levels clone detection
  - Higher: statement level; Lower: program level
- Recognizes only C nodes
  - Coverage for other languages, such as C++ and C#
  - Another approach: language independent
Program Behavior Modeling with Event Grammars
UniFrame Quality
Framework - Event Grammar Approach

Mikhail Auguston
Naval Postgraduate School
Program Behavior Models

- Program monitoring activities can be specified in a uniform way using program behavior models based on the event notion.
- An event corresponds to any detectable action, e.g. subroutine call, message passing, etc. An event corresponds to a time interval.
- Two partial order binary relations are defined for events: precedence and inclusion.
- An event has attributes: type, duration, program state at beginning or end of the event, value, …
## Event Types

Event types and type specific attributes for the Unicorn language

<table>
<thead>
<tr>
<th>Event Type</th>
<th>Description</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>prog_ex</td>
<td>whole program execution</td>
<td></td>
</tr>
<tr>
<td>expr_eval</td>
<td>expression evaluation</td>
<td>value, operator, type, failure_p</td>
</tr>
<tr>
<td>func_call</td>
<td>function call</td>
<td>func_name, paramlist</td>
</tr>
<tr>
<td>input, output</td>
<td>I/O</td>
<td>file</td>
</tr>
<tr>
<td>variable</td>
<td>variable reference</td>
<td></td>
</tr>
<tr>
<td>literal</td>
<td>reference to a constant</td>
<td></td>
</tr>
<tr>
<td>lhp</td>
<td>lefthand part, assignment</td>
<td>address</td>
</tr>
<tr>
<td>rhp</td>
<td>righthand part, assignment</td>
<td></td>
</tr>
<tr>
<td>clause</td>
<td>then-, else-, or case execution</td>
<td></td>
</tr>
<tr>
<td>test</td>
<td>test evaluation</td>
<td></td>
</tr>
<tr>
<td>iteration</td>
<td>loop iteration</td>
<td></td>
</tr>
</tbody>
</table>
Program Behavior Models

- **Event grammar** specifies the constraints on configurations of events generated at the run time (axioms, or “lightweight semantics” of the target language)

- Some axioms are generic, e.g. transitivity

  \[
  A \text{ PRECEDES } B \quad \text{and} \quad B \text{ PRECEDES } C \quad \Rightarrow \quad A \text{ PRECEDES } C
  \]

  \[
  A \text{ IN } B \quad \text{and} \quad B \text{ IN } C \quad \Rightarrow \quad A \text{ IN } C
  \]
Example of Event Grammar

```
ex_prog::  ex_stmt *
ex_stmt::  ex_assignmt | ex_read_stmt | ...

ex_assignmt:: eval_expr  destination
```

Example of an event trace

```
IN
PRECEDES
```
Program Monitoring

- **Monitoring activities**: assertion checking, profiles, performance measurements, dynamic QoS metrics, visualization, debugging queries, intrusion detection

- Program monitoring can be specified in terms of **computations over event traces**

- We introduce a specific language **FORMAN** to specify computations over event traces (based on event patterns and aggregate operations over events)
Examples

1) Profile
   SAY( "Number of function A calls is "
       CARD[ x: func_call & x.name == "A"
           FROM ex_prog ]
   )

2) Generic debugging rule (typical error description)
   FOREACH e: eval_expr :: (v: variable)
       FROM ex_prog
       EXISTS d: destination FROM e.PREV_PATH
       v.source_code = d.source_code
       ONFAIL SAY("Uninitialized variable 
                   v.source_code "is used in expression 
                   e)
Process

Target Language

Event Grammar

FORMAN

Program

Event Trace

Computation (Query)

Results
Examples

3)  Debugging query

SAY("The history of variable x ")
[d: destination & d.source_code == "x"
   FROM ex_prog APPLY d.value ]

4)  Traditional debugging print statements

FOREACH f: func_call & f.name == "A"
    FROM ex_prog
    f.value_at_begin(
        printf("variable x is %d\n", x) )
The novelty claims of our approach

- **Uniform framework** for program monitoring based on precise behavior models and event trace computations
- Computations on the event traces can be implemented in a **nondestructive** way via automatic instrumentation of the source code or even of the executables (Dyninst approach)
- Can specify **generic trace computations**: typical bug detection, dynamic QoS metrics, profiles, visualization, …
- An alternative approach to **aspect-oriented** paradigm
Accomplished projects and work in progress

- Assertion checker for Pascal subset (via interpreter)
- Assertion checker for the C language (via source code instrumentation)
- Assertion checker and visualization tool for the Unicon language (via Virtual Machine monitors)
- UniFrame dynamic QoS metrics, UniFrame project (via glue and wrapper instrumentation), funded by ONR
- Intrusion detection and countermeasures (via Linux kernel library instrumentation), funded by the Department of Justice Homeland Security Program