ABSTRACTING PROGRAM DEPENDENCIES USING THE METHOD DEPENDENCE GRAPH

Haipeng Cai and Raul Santelices
Department of Computer Science and Engineering
University of Notre Dame
Dependence analysis underlies many tasks

Testing / Debugging
Evolution / Maintenance
Performance Optimization

Program Dependence Analysis

Program
Traditional dependence model is heavyweight

- Offer fine-grained results
- Suffer from low scalability

```c
void main() {
    int i = 1;  int sum = 0;
    while (i<11) {
        sum = add(sum, i);
        i = add(i, 1);
    }
    printf("sum = %d\n", sum);
    printf("i = %d\n", i);
}
static int add(int a, int b) {
    return(a+b);
}
```

An example program and its System Dependence Graph (SDG), both courtesy of GrammaTech Inc.
Fine granularity may not be necessary

- Impact analysis
  - Mostly based on program dependence analysis
  - Commonly adopted at method level (even coarser levels)

- Program comprehension
  - Largely reduced to understanding program dependencies
  - More practical to explore method-level artifacts
Problems with fine-grained model

- Excessive overhead
  - Building the model is expensive or impractical
- Low cost-effectiveness
  - Large overhead not well paid off
Problems with existing abstraction

- Static execute after (SEA) [J. Jasz et al., 2012]
- ICFG (Interprocedural Control Flow Graph)

```c
void main() {
    int i = 1; int sum = 0;
    while (i<11) {
        sum = add(sum, i);
        i = add(i, 1);
    }
    printf("sum = %d\n", sum);
    printf("i = %d\n", i);
}
static int add(int a, int b) {
    return(a+b);
}
```
Problems with existing abstraction

- **Static execute after (SEA)**
  - Further simplified as **ICCFG** *(Interprocedural Component CFG)*
    - SEA := CALL U RET U SEQ
    - SEA (A,B) => B depends on A

```c
void main() {
    int i = 1;  int sum = 0;
    while (i<11) {
        sum = add(sum, i);
        i = add(i, 1);
    }
    printf("sum = %d\n", sum);
    printf("i = %d\n", i);
}

static int add(int a, int b) {
    return(a+b);
}
```

- **Control flow does not imply dependency**
  - Imprecision
- **Fast but rough**
  - Low cost-effectiveness
Abstracting program dependencies

Solution

- Method-level dependence abstraction
  - Model complete dependencies among methods directly

Goals

- Improved precision
  - Compute dependencies explicitly
- Improved cost-effectiveness
  - Trade precision for efficiency

Approach

- METHOD DEPENDENCE GRAPH (MDG)
  - An abstraction of the SDG
Abstracting program dependencies

- **The MDG abstraction**
  - **Ports**
    - Statements at the *boundary* of a method
    - Endpoints of interprocedural dependence edges
  - **Classification of ports**
    - Incoming/outgoing ports (IP/OP)
    - Data-dependence (DD) / control-dependence (CD) ports

```java
static int add(int a, int b)
{
    return(a+b);
}
```

```
static int add(int a, int b)
{
    return(a+b);
}
```
Abstracting program dependencies

- The MDG abstraction
  - Interprocedural dependencies
    - Incoming/outgoing dependencies (ID/OD)
    - Data /control dependencies
      - Data dependencies: Parameter / Return / Heap
  - Intraprocedural dependencies
    - Abstract with summary dependencies
Abstracting program dependencies

data dependence
control dependence

\( p \): parameter
\( r \): return value
\( h \): heap variable

An example MDG (top) and the closeup of one node M2 (bottom)
Abstracting program dependencies

- **Construction of MDG for a program P**
  - Initialize MDG for P
  - For each method m in P
    - Find all CD ports on m
    - Find all DD ports on m
  - For each method m in P
    - Match OPs of m against IPs of all other methods
    - Build procedure dependence graph (PDG) of m [J. Ferrante et al., 1987]
    - Connect IPs to OPs in m based on the PDG of m
Abstracting program dependencies

- Data-Dependence (DD) port matching

<table>
<thead>
<tr>
<th>DD type</th>
<th>Outgoing Port (OP)</th>
<th>Incoming Port (IP)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameter</td>
<td>Actual parameter at call site</td>
<td>Formal parameter at callee’s entry</td>
</tr>
<tr>
<td>Return</td>
<td>Return value at callee</td>
<td>Use of return at caller site</td>
</tr>
<tr>
<td>Heap</td>
<td>Definition of heap variable</td>
<td>Use of heap variable</td>
</tr>
</tbody>
</table>
Abstracting program dependencies

- Control-Dependence (CD) port matching

<table>
<thead>
<tr>
<th>CD type</th>
<th>Outgoing Port (OP)</th>
<th>Incoming Port (IP)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal</td>
<td>Branch / polymorphic call site</td>
<td>Entry of callee</td>
</tr>
<tr>
<td>Exception-driven</td>
<td>Exception-throwing site</td>
<td>Entry of catch block that handles the exception</td>
</tr>
</tbody>
</table>
## Evaluating the MDG

### Subject programs

<table>
<thead>
<tr>
<th>Subject</th>
<th>KLOC</th>
<th>#Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>Schedule 1</td>
<td>0.3</td>
<td>24</td>
</tr>
<tr>
<td>NanoXml</td>
<td>3.5</td>
<td>282</td>
</tr>
<tr>
<td>Ant-v0</td>
<td>18.8</td>
<td>1,863</td>
</tr>
<tr>
<td>XML-security-v1</td>
<td>22.4</td>
<td>1,928</td>
</tr>
<tr>
<td>Jaba</td>
<td>37.9</td>
<td>3,332</td>
</tr>
</tbody>
</table>
Evaluating the MDG

- **Data**
  - Method-level forward dependence sets

- **Metrics**
  - Effectiveness: precision and recall
  - Costs: time costs of MDG construction and querying

- **Ground truth**
  - Statement-level forward static slicing
    - Uplifted to method level slices
MDG is significantly more accurate

Results: precision

Mean precision improvement: 46.9%

*Both techniques are sound (100% recall). The higher the bar, the better
MDG remains efficient

Results: costs

Abstraction time

<table>
<thead>
<tr>
<th>Subject</th>
<th>SEA</th>
<th>MDG</th>
</tr>
</thead>
<tbody>
<tr>
<td>Schedule1</td>
<td>3s</td>
<td>4s</td>
</tr>
<tr>
<td>NanoXml</td>
<td>4s</td>
<td>9s</td>
</tr>
<tr>
<td>Ant-v0</td>
<td>17s</td>
<td>130s</td>
</tr>
<tr>
<td>XML-security-v1</td>
<td>22s</td>
<td>77s</td>
</tr>
<tr>
<td>Jaba</td>
<td>28s</td>
<td>302s</td>
</tr>
<tr>
<td>Overall average</td>
<td>14.8s</td>
<td>104.4s</td>
</tr>
</tbody>
</table>
MDG remains efficient

- Results: costs
  - Mean querying time

<table>
<thead>
<tr>
<th>Subject</th>
<th>SEA</th>
<th>MDG</th>
<th>Static slicing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Schedule1</td>
<td>6ms</td>
<td>4ms</td>
<td>124ms</td>
</tr>
<tr>
<td>NanoXml</td>
<td>9ms</td>
<td>3ms</td>
<td>12,67ms</td>
</tr>
<tr>
<td>Ant-v0</td>
<td>64ms</td>
<td>45ms</td>
<td>34,896ms</td>
</tr>
<tr>
<td>XML-security-v1</td>
<td>50ms</td>
<td>43ms</td>
<td>24,092ms</td>
</tr>
<tr>
<td>JABA</td>
<td>213ms</td>
<td>121ms</td>
<td>444,188ms</td>
</tr>
<tr>
<td>Overall average</td>
<td>131.4ms</td>
<td>53.3ms</td>
<td>55737.9ms</td>
</tr>
</tbody>
</table>
Summing up

- **Contributions**
  - A new method-level program-dependence abstraction – the method dependence graph (MDG)
  - Empirical evidence showing the advantage of the MDG over the baseline abstraction approach (SEA)
  - Study contrasting traditional dependence model and method-level abstraction for forward dependence analysis

- **Future work**
  - Improve hybrid dynamic analysis using the MDG
  - Develop MDG-based program-comprehension tools
Acknowledgements

Abstracting Program Dependencies using the Method Dependence Graph
Haipeng Cai
http://cse.nd.edu/~hcai/
hcai@nd.edu
Problems with existing abstraction

- **Component dependence graph** [B. Li et al., 2005]
  - High-level coarse dependencies among components for component-based systems w/o traditional code-level analysis

- **Influence graph** [B. Breech et al., 2006]
  - Interface-level data dependencies among functions for procedural programs w/o intraprocedural dependencies

- **Program summary graph** [D. Callahan, 1988]
  - Interprocedural data dependencies w/o control dependencies

- **Linkage grammar** [S. Horwitz et al., 1990]
  - Statement-level dependencies (from in to out parameters)
MDG Construction Algorithm

Algorithm 1: BUILDMDG(program P, exception set unhandled)
1: $G := \text{empty graph} // \text{start with empty MDG of } P$
2: $IP := OP := \emptyset // \text{maps of methods to incoming/outgoing ports}$
   // Step 1: find ports
3: for each method $m$ of $P$ do
4:    FINDDDPORTS($m, IP, OP$)
5:    FINDCDPORTS($m, IP, OP$)
   // Step 2: connect ports
6: for each method $m$ of $P$ do
7:    for each DD port $z \in OP[m]$ do
8:       add $\{(z, z') \mid \exists m' \text{ s.t. } z' \in IP[m'] \land \text{data dep}(z, z')\}$ to $G$
9:    COMPUTEINTERCD(G, unhandled, m, IP, OP)
10:   pdg := GETPDG($m$)
11: for each port $z \in IP[m]$ do
12:    add $\{(z, z') \mid z' \in OP[m] \land \text{reaches}(z, z', \text{pdg})\}$ to node $G_m$
13: return $G$

Algorithm 2: FINDDDPORTS($m, IP, OP$)
1: for each call site $cs$ in $m$ do
2:    for each callee $m'$ of $cs$ do
3:       add $\{D(a, cs) \mid a \in \text{actual_params}(cs)\}$ to $OP[m']$
4:       add $\{U(f, m') \mid f \in \text{formal_params}(m')\}$ to $IP[m']$
5: if $\text{return_type}(m) \neq \text{void}$ then
6:   add $\{D(rs) \mid rs \in \text{return_sites}(m)\}$ to $OP[m']$
7: for each caller site $crs$ of $m$ do
8:   add $\{U(crs.rs, rs) \mid rs \in \text{return_sites}(m)\}$ to $IP[crs.m]$
9: for each heap variable definition $hd$ in $m$ do add $hd$ to $OP[m']$
10: for each heap variable use $hu$ in $m$ do add $hu$ to $OP[m']$

Algorithm 3: FINDCDPORTS($m, IP, OP$)
1: add entry of $m$ to $IP[m]$ // entry represents all CD targets for callers
2: for each edge $(h, t)$ in $\text{GETCD}(m)$ do
3:    if $t$ is a single-target call site then {add $h$ to $OP[m]$}
4:    if $t$ unconditionally throws unhandled exception in $m$ then
5:       add $h$ to $OP[m]$
6: for each multi-target call site $cs$ in $m$ do {add $cs.s$ to $OP[m]$}
7: for each statement $s$ in $m$ do
8:    if $s$ catches interprocedural exception then {add $s$ to $OP[m]$}
9:    if $s$ conditionally throws exception unhandled in $m$ then
10:       add $s$ to $OP[m]$
The *method dependence graph* offers a program abstraction of better cost-effectiveness tradeoff than both fine-grained model and existing alternative abstractions.