ABSTRACTING PROGRAM DEPENDENCIES USING THE METHOD DEPENDENCE GRAPH

Haipeng Cai and Raul Santelices

Department of Computer Science and Engineering

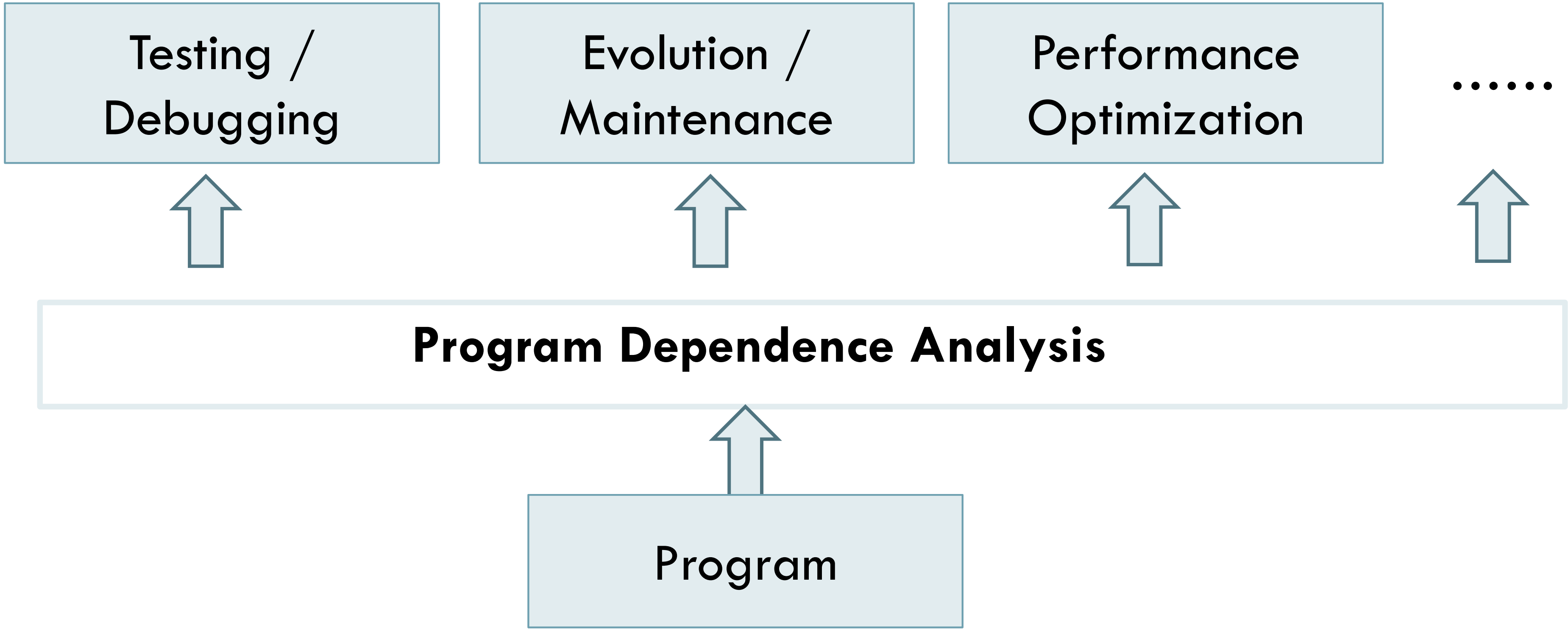
University of Notre Dame

UNIVERSITY OF NOTRE DAME
College of Engineering

QRS 2015

Supported by ONR Award N000141410037

Dependence analysis underlies many tasks



Testing / Debugging

Evolution / Maintenance

Performance Optimization

……

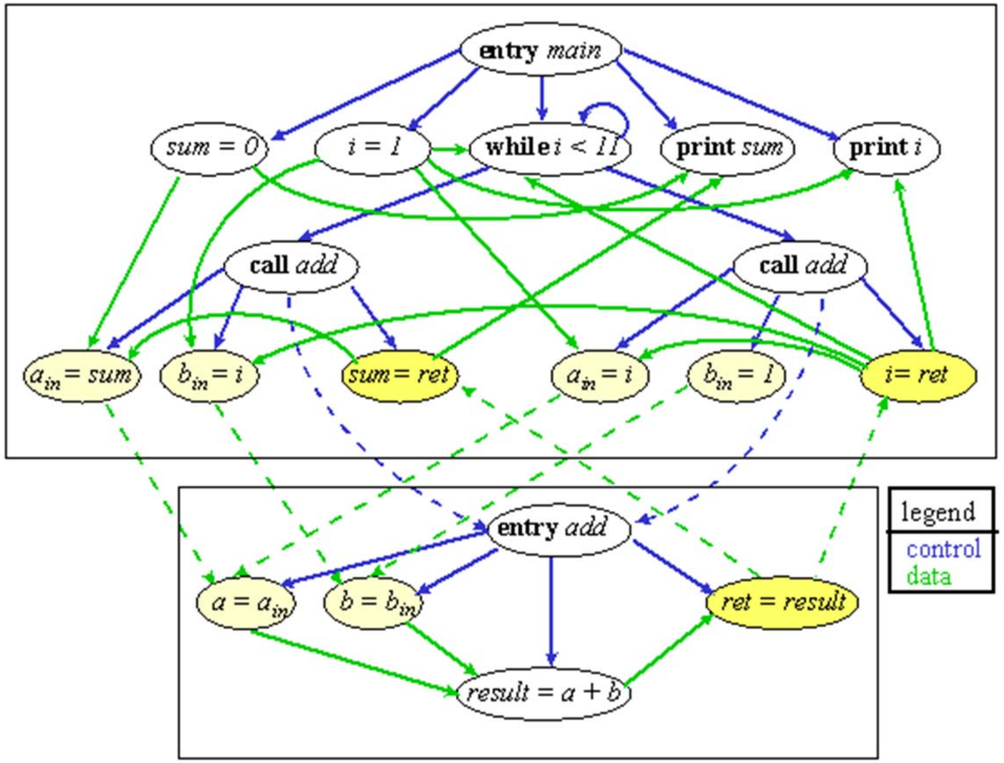
Program Dependence Analysis

Program

Traditional dependence model is heavyweight

- ☐ Offer fine-grained results
- ☐ Suffer from low scalability

```
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 *<u>System Dependence Graph</u>* (***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

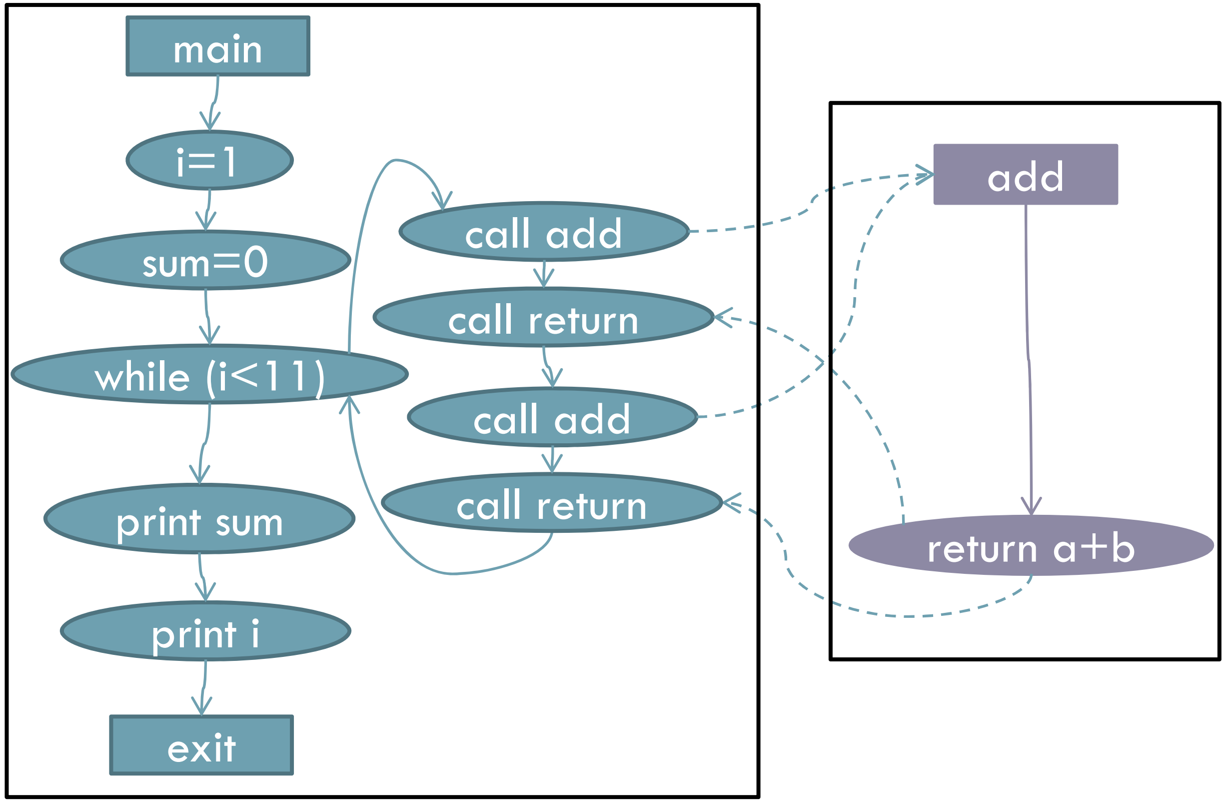
Problems with fine-grained model

- 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*)

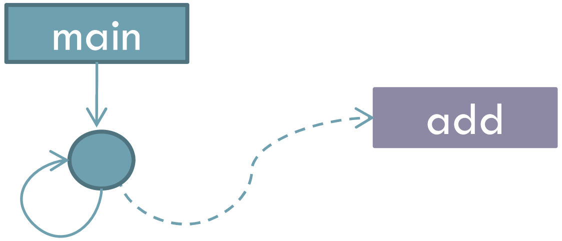
```
void main() {
    int i = 1;     int sum = 0;
    while (i<11) {
        sum = add(sum, i);
        i = add(i, 1);
    }
    printf("sum = %d\n", sum);
    printf("i = %d\n", i);
}
static int add(int a, int b)
{
    return(a+b);
}
```

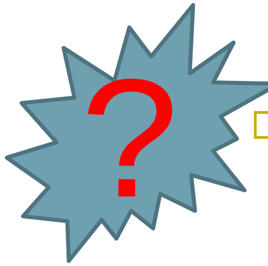


Problems with existing abstraction

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

```
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

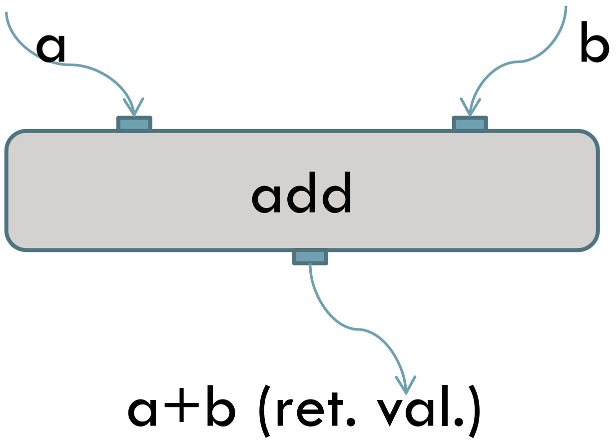
Abstracting program dependencies

- 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

```
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

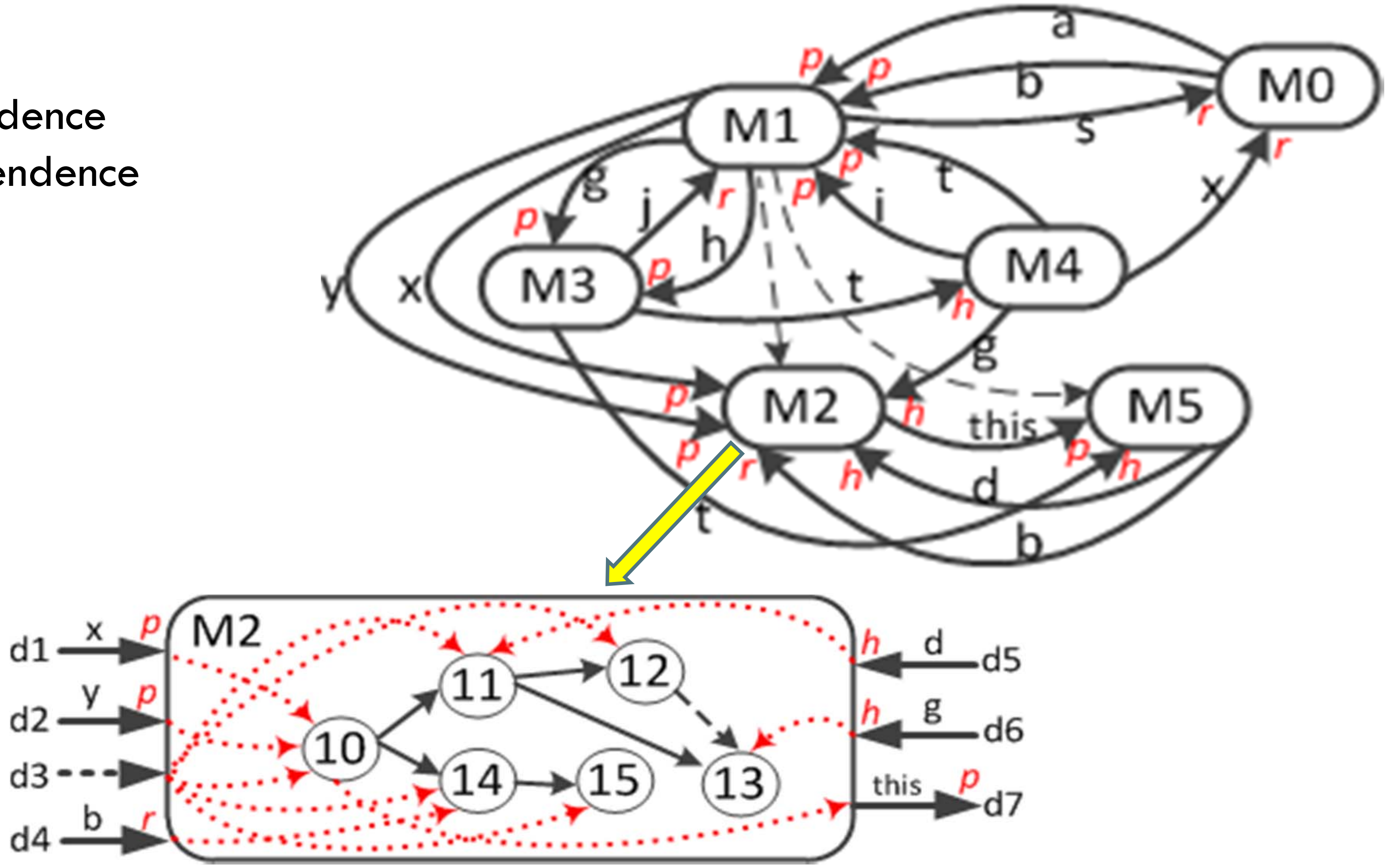
→ data dependence

- - → 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

DD type	**Outgoing Port (OP)**	**Incoming Port (IP)**
Parameter	Actual parameter at call site	Formal parameter at callee's entry
Return	Return value at callee	Use of return at caller site
Heap	Definition of heap variable	Use of heap variable

Abstracting program dependencies

- Control-Dependence (CD) port matching

CD type	Outgoing Port (OP)	Incoming Port (IP)
Normal	Branch / polymorphic call site	Entry of callee
Exception-driven	Exception-throwing site	Entry of catch block that handles the exception

Evaluating the MDG

- Subject programs

Subject	**KLOC**	**#Methods**
Schedule1	0.3	24
NanoXml	3.5	282
Ant-v0	18.8	1,863
XML-security-v1	22.4	1,928
Jaba	37.9	3,332

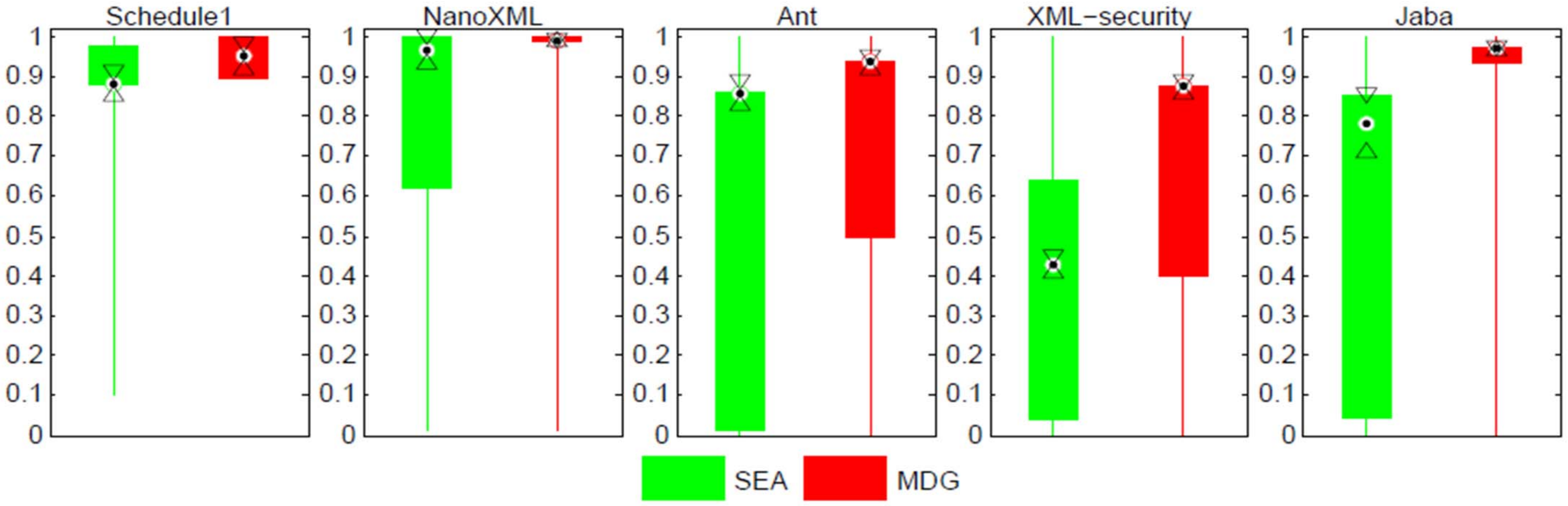
Evaluating the MDG

- 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

Subject	**SEA**	**MDG**
Schedule1	3s	4s
NanoXml	4s	9s
Ant-v0	17s	130s
XML-security-v1	22s	77s
Jaba	28s	302s
Overall average	14.8s	104.4s

MDG remains efficient

- Results: costs
  - Mean querying time

Subject	SEA	MDG	Static slicing
Schedule1	6ms	4ms	124ms
NanoXml	9ms	3ms	12,67ms
Ant-v0	64ms	45ms	34,896ms
XML-security-v1	50ms	43ms	24,092ms
JABA	213ms	121ms	444,188ms
Overall average	131.4ms	53.3ms	55737.9ms

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

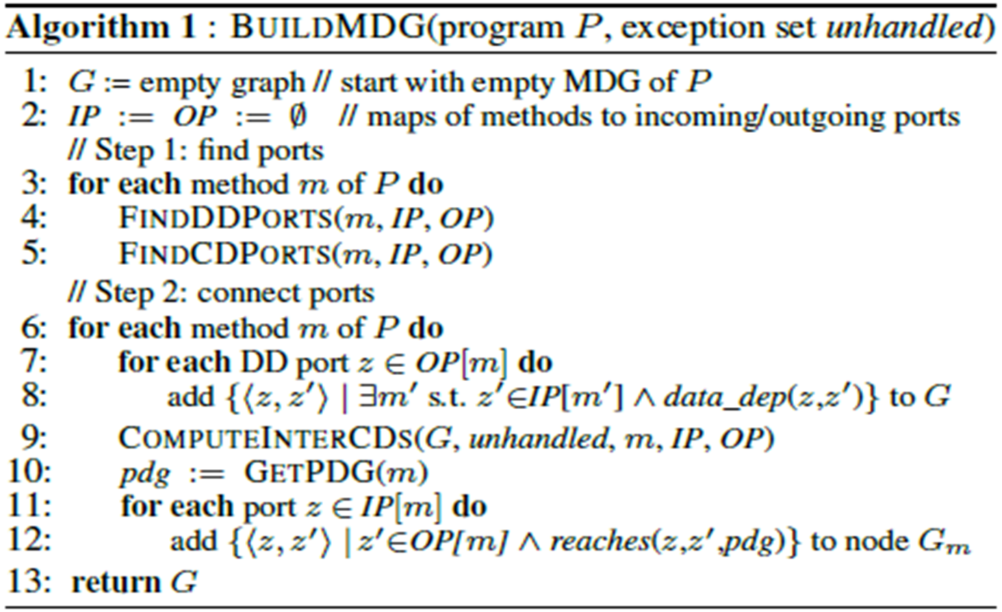
http://cse.nd.edu/~hcai/

hcai@nd.edu

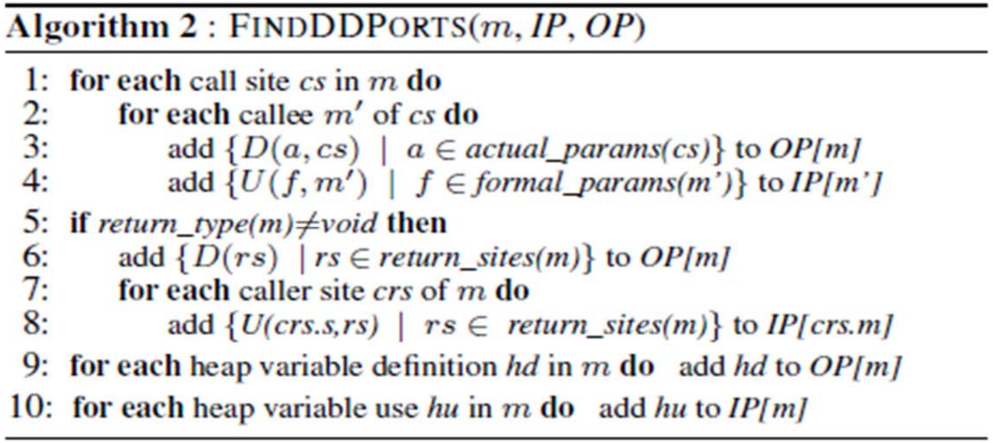
Problems with existing abstraction

- 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)

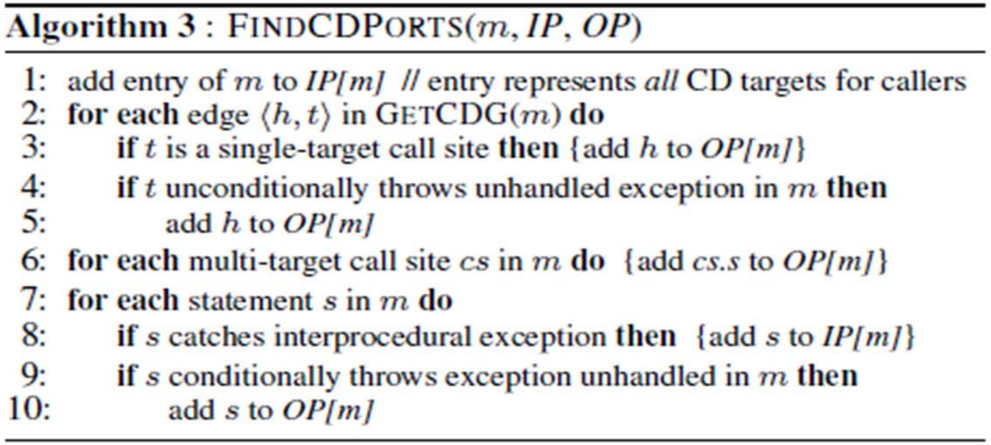
MDG Construction Algorithm



```
Algorithm 1 : BUILDMDG(program $P$, exception set $unhandled$)
$G$ := empty graph // start with empty MDG of $P$
$IP$ := $OP$ := $\emptyset$   // maps of methods to incoming/outgoing ports
// Step 1: find ports
for each method $m$ of $P$ do
    FINDDDPORTS($m$, $IP$, $OP$)
    FINDCDPORTS($m$, $IP$, $OP$)
// Step 2: connect ports
for each method $m$ of $P$ do
    for each DD port $z \in OP[m]$ do
        add $\{\langle z, z' \rangle \mid \exists m'$ s.t. $z' \in IP[m'] \wedge data\_dep(z,z')\}$ to $G$
    COMPUTEINTERCDS($G$, $unhandled$, $m$, $IP$, $OP$)
    $pdg$ := GETPDG($m$)
    for each port $z \in IP[m]$ do
        add $\{\langle z, z' \rangle \mid z' \in OP[m] \wedge reaches(z,z',pdg)\}$ to node $G_m$
return $G$
```



```
Algorithm 2 : FINDDDPORTS($m$, $IP$, $OP$)
for each call site $cs$ in $m$ do
    for each callee $m'$ of $cs$ do
        add $\{D(a, cs) \mid a \in actual\_params(cs)\}$ to $OP[m]$
        add $\{U(f, m') \mid f \in formal\_params(m')\}$ to $IP[m']$
if $return\_type(m) \neq void$ then
    add $\{D(rs) \mid rs \in return\_sites(m)\}$ to $OP[m]$
    for each caller site $crs$ of $m$ do
        add $\{U(crs.s, rs) \mid rs \in return\_sites(m)\}$ to $IP[crs.m]$
for each heap variable definition $hd$ in $m$ do   add $hd$ to $OP[m]$
for each heap variable use $hu$ in $m$ do   add $hu$ to $IP[m]$
```



```
Algorithm 3 : FINDCDPORTS($m$, $IP$, $OP$)
add entry of $m$ to $IP[m]$  // entry represents all CD targets for callers
for each edge $\langle h, t \rangle$ in GETCDG($m$) do
    if $t$ is a single-target call site then {add $h$ to $OP[m]$}
    if $t$ unconditionally throws unhandled exception in $m$ then
        add $h$ to $OP[m]$
for each multi-target call site $cs$ in $m$ do  {add $cs.s$ to $OP[m]$}
for each statement $s$ in $m$ do
    if $s$ catches interprocedural exception then  {add $s$ to $IP[m]$}
    if $s$ conditionally throws exception unhandled in $m$ then
        add $s$ to $OP[m]$
```

Q&A

The *method dependence graph* offers a program abstraction of better cost-effectiveness tradeoff than both fine-grained model and existing alternative abstractions.