다중 쓰레드 프로그램의 경쟁상태오류 검출기법 분류

KAIST Provable SW 연구실
홍신, 김문주
Motivations (1/2)

• Many SW applications utilize multi-threaded programming techniques as multi-core hardware become widely spread.

• Writing correct multi-threaded programs is difficult.
  – Exponential number of execution scenarios
  – Detecting errors by assertion is not effective

  ➔ Bug detection techniques specialized for concurrency errors
Motivations (2/2)

• Many techniques have evolved
  – Deadlock: [K. M. Chandy et al., TOCS 1983],
    [R. Agarwal et al., IBM J. 2010]

• However, for race bugs, techniques have used their own
definitions and notations without any reliance on a common
ground or platform.

<table>
<thead>
<tr>
<th>Bug name</th>
<th>Specification</th>
<th>Target program</th>
<th>Bug checking method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kivati</td>
<td>Atomicity violation</td>
<td>User annotations</td>
<td>C program</td>
</tr>
<tr>
<td>LiteRace</td>
<td>Data race</td>
<td>N/A</td>
<td>x86 binary</td>
</tr>
<tr>
<td>Havelund et al.</td>
<td>High-level data race</td>
<td>Analyze code and infer specifications</td>
<td>Java</td>
</tr>
</tbody>
</table>

→ Develop classification which clarify the relationship between techniques and provide a clear top-down view of race detection techniques
Approach

• Provide a formal execution model and specify various bug conditions according to the model

- Decoupling **what to detect** and **how to detect**
  
  – Concurrent behavior analyses: generate potential executions of a program by information from static/dynamic analyses [F. Chen *et al.* ICSE 2008]

  – **Bug condition checking**: examine a given execution is acceptable/erroneous
Approach

- Classify bug conditions according to type of specifications
  - Operation block
  - Data association
Contents

• Execution model for multithreaded program

• Four classes of race detection techniques
  – For each class,
    • bug example
    • bug conditions
    • techniques for checking conditions

• Implications for better race detections
Execution Model

• An execution model of a target program $P$ used for technique $D$ is defined as

$$EM_P(D) = (\boxed{T, e, \triangleright}, \boxed{B_{op}, A_{data}})$$

Program behavior Requirements

• $T$: a finite set of threads
• $e$: an interleaved execution
  a finite sequence of operations $p_1, p_2, ..., p_n$ where $p_i \in Operation$
  - $thread(p) \in T$
  - $optr(p) \in Operator$
    $conflict(optr(p), optr(q))$ if the operators of $p$ and $q$ are commutable.
  - $oprd(p) \subseteq V_S$
An execution model of a target program $P$ used for technique $D$ is defined as

$$EM_P(D) = (T, e, \triangleright, B_{op}, A_{data})$$

- **$T$**: a finite set of threads
- **$e$**: an interleaved execution
  - a finite sequence of operations $p_1, p_2, \ldots, p_n$ where $p_i \in Operation$
  - $thread(p) \in T$
  - $optr(p) \in Operator$
  - $conflict(optr(p), optr(q))$ if the operators of $p$ and $q$ may not be commutable.
  - $oprd(p) \subseteq V_S$
An execution model of a target program $P$ used for technique $D$ is defined as

$$EM_P(D) = (T, e, \triangleright, B_{op}, A_{data})$$

- Operations in an execution are totally ordered by their start time.
- $\triangleright \subseteq Operation \times Operation$
  $$(p, q) \in \triangleright \text{ if } t_s(p) < t_s(q) \text{ and } t_e(p) < t_e(q)$$
An execution model of a target program $P$ used for technique $D$ is defined as

$$EM_P(D) = ( T, e, \triangleright, B_{op}, A_{data} )$$

$B_{op} = \{ b_1, b_2, \ldots, \}$ where $b_i : Operation \times Operation$

An execution of an atomic code region corresponds to a sequence of operation, operation blocks. $(p, q) \in b_i$ indicates that $p$ and $q$ are in the same operation block.

class BankAccount {
    int balance ;
    ...
    void withdraw(int amount) {
        if (getBalance() \geq amount) {
            lock(m) ;
            balance = balance - amount ;
            unlock(m) ;
        }
    }
}
An execution model of a target program $P$ used for technique $D$ is defined as

$$EM_P(D) = ( T, e, \triangleright, B_{op}, A_{data} )$$

$A_{data}$: $V_S \times V_S$ where $V_S$ is a set of shared variables

Frequently, variables in a composite data structure have dependencies and there exists relations/invariances on these variables.

class BankAccount {
    int balance;
    int debt;
    /* invariant: (debt == 0 $\land$ balance == 0) $\lor$
    (debt > 0 $\land$ balance == 0) $\lor$
    (balance > 0 $\land$ debt == 0) */

    (balance, debt) $\in$ $A_{data}$
    (debt, balance) $\in$ $A_{data}$
Contents

• Execution model for multithreaded program

• Four classes of race detection techniques
  – For each class,
    • bug example
    • bug conditions
    • techniques for checking conditions

• Implications for better race detections
In parallelization,

\[
\{ B_1 \} \rightarrow \{ B_1 \} \quad \{ B_2 \} \rightarrow \{ B_2 \}
\]

A sufficient condition for safe parallelization:

\[
\text{MemoryRead}(B_1) \cap \text{MemoryWrite}(B_2) = \emptyset \quad \text{and} \quad \text{MemoryWrite}(B_1) \cap \text{MemoryRead}(B_2) = \emptyset
\]

- In “What are race conditions?” [R. H. Netzer et al., LOPLAS 1992], a data race \( \langle a, b \rangle \) over an execution exists if and only if
  1. a data conflict exists in a program between \( a \) and \( b \),
  2. no temporal ordering between \( a \) and \( b \).
In [Savage et al., SOSP 1997], a *data race* occurs when there exists two operations:

1. executed by two concurrent threads,
2. access a shared variable
3. at least one access is write,
4. no explicit mechanism to coordinate their execution order
- Buggy program code

```java
class BankAccount_A {
    int balance;
    // balance should be non-negative
    void withdraw(int x) {
        if (balance >= x) {
            balance = balance - x;
        }
    }
}
```

- Error scenario

```
[ balance = 10 ]
--t1: withdraw(10)--
1: if(balance >= 10)
2: balance = 0 - 10;
--t2: withdraw(10)--
1: if(balance >= 10)
2: balance = 10 - 10;
```

- A target program $P$ has a race-1 bug if there is an execution $\sigma$ such that $\sigma$ has two operations $p$ and $q$ that satisfy the following conditions:

(A1) $\text{thread}(p) \neq \text{thread}(q)$
(A2) $\text{oprd}(p) \cap \text{oprd}(q) \neq \emptyset$
(A3) $\text{conFLICT}(\text{optr}(p), \text{optr}(q))$
(A4) $p \not\gg q \land q \not\gg p$

$p$ and $q$ are commutable?
## Race-1: Data-race

### Race-1 Detection Techniques

<table>
<thead>
<tr>
<th>Technique</th>
<th>(A1) ( \text{thread}(p) \neq \text{thread}(q) )</th>
<th>(A2) ( \text{oprd}(p) \cap \text{oprd}(q) \neq \emptyset )</th>
<th>(A3) ( \text{conflict}(\text{optr}(p), \text{optr}(q)) )</th>
<th>(A4) ( p \nrightarrow q )</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Choi et al.</strong></td>
<td>concrete thread id.</td>
<td>concrete mem. addr.</td>
<td>O</td>
<td>tracking lock acq/rel, thread fork/join</td>
</tr>
<tr>
<td><strong>Eraser</strong></td>
<td>check shared or non-shared</td>
<td>concrete mem. addr.</td>
<td>O</td>
<td>tracking lock acq/rel</td>
</tr>
<tr>
<td><strong>Hybrid data race detection</strong></td>
<td>concrete thread id.</td>
<td>concrete mem. addr.</td>
<td>O</td>
<td>tracking lock acq/rel, message send/receive</td>
</tr>
<tr>
<td><strong>Racer</strong></td>
<td>concrete thread id.</td>
<td>concrete mem. addr.</td>
<td>O</td>
<td>tracking lock acq/rel</td>
</tr>
<tr>
<td><strong>RaceTrack</strong></td>
<td>check shared or non-shared</td>
<td>concrete mem. addr.</td>
<td>O</td>
<td>tracking lock acq/rel, thread fork/join</td>
</tr>
<tr>
<td><strong>TRaDe</strong></td>
<td>check shared or non-shared</td>
<td>concrete mem. addr.</td>
<td>O</td>
<td>tracking lock acq/rel</td>
</tr>
<tr>
<td><strong>LiteRace</strong></td>
<td>concrete thread id.</td>
<td>concrete mem. addr.</td>
<td>O</td>
<td>tracking lock acq/rel, message passing, atomic instructions</td>
</tr>
<tr>
<td><strong>Chord</strong></td>
<td>static approx.</td>
<td>static alias analysis</td>
<td>O</td>
<td>tracking lock acq/rel</td>
</tr>
<tr>
<td><strong>RacerX</strong></td>
<td>static approx.</td>
<td>static alias analysis</td>
<td>O</td>
<td>tracking lock acq/rel</td>
</tr>
<tr>
<td><strong>RELAY</strong></td>
<td>static approx.</td>
<td>static alias analysis</td>
<td>O</td>
<td>tracking lock acq/rel</td>
</tr>
<tr>
<td><strong>RccJava</strong></td>
<td>static approx.</td>
<td>static alias analysis</td>
<td>O</td>
<td>tracking lock acq/rel</td>
</tr>
</tbody>
</table>
Race-1: Data-race (5/6)

- Limitations – false positive

Interferences by $q$ do not affect the correctness of further executions of $T_1$

Case 1: no further $p$ is dependent on $p$

Case 2: $p$ and $q$ are commutable due to semantics
Race-1: Data-race

- Limitations – false negative

Invariant/constraints over \(x_1\) and \(x_2\) might be broken
Race-2: Atomic Block Violation (1/5)

- Race-2 techniques check whether or not an *operation block* can interfere with another thread.

In [Savage et al., SOSP 1997],
- a *data race* occurs when there exists two operations
  1. executed by two concurrent threads,
  2. access a shared variable
  3. at least one access is write,
  4. no explicit mechanism to coordinate their execution order

From a previous slide for race-1
- a sequence of accesses (operation block)

Thread 1

Thread 2
Race-2: Atomic Block Violation (2/5)

- A target program $P$ has a race-2 bug if there is an execution $\sigma$ such that $\sigma$ has three operations $p$, $p'$, and $q$ that satisfy the following conditions:

(B1) $\text{thread}(p) \neq \text{thread}(q)$
(B2) $\text{oprd}(p) \cap \text{oprd}(q) \neq \emptyset$
(B3) $\text{conflict} (\text{optr}(p), \text{optr}(q))$
(B4) $\exists b_i \in B_{op} \cdot ((p, p') \in b_i)$
(B5) $\text{oprd}(p) \cap \text{oprd}(p') \neq \emptyset$
(B6) $\text{conflict} (\text{optr}(p), \text{optr}(p'))$
(B7) $\text{oprd}(p') \cap \text{oprd}(q) \neq \emptyset$
(B8) $\text{conflict} (\text{optr}(p'), \text{optr}(q))$
(B9) $q \not\equiv p \land p' \not\equiv q$
Race-2: Atomic Block Violation (3/5)

- Race-2 bug example:

- Buggy program code

```java
class BankAccount_B {
    Lock m;
    int balance;
    // balance should be non-negative
    // balance should be synchronized by m

    int getBalance() {
        int tmp;
        1: lock(m);
        2: tmp = balance;
        3: unlock(m);
        4: return tmp;
    }

    void withdraw(int x) {
        /*@atomic region begins*/
        11: if (getBalance() >= x) {
        12: lock(m);
        13: balance = balance - x;
        14: unlock(m);
        /*@atomic region ends*/
        ...
    }
}
```

- Race-2 error scenario

```
[ balance = 10 ]
 --t1: withdraw(10) --
 operation block b,
11: if(getBalance()>=10)
\> 1:lock(m);
  2:tmp = balance; p
  3:unlock(m);
  4:return tmp;
12: lock(m);
13: balance=10-10; q
14: unlock(m);

12: lock(m);
13: balance=0-10; p'
14: unlock(m);
```

The invariant is violated: balance becomes -10.
## Race-2: Atomic Block Violation (4/5)

### Race-2 Detection Techniques

<table>
<thead>
<tr>
<th></th>
<th>(B1) thread()</th>
<th>(B2, B4, B7) oprd()</th>
<th>(B6) conflict(optr(p), optr(q)) where thread(p) ≠ thread(q)</th>
<th>(B3, B8) conflict(optr(p), optr(q)) where thread(p) = thread(q)</th>
<th>(B9)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Atomic-Aid</td>
<td>concrete thread id.</td>
<td>concrete mem. addr.</td>
<td>O   O   Cond.   X</td>
<td>O   O   O   O   O</td>
<td>tracking lock/unlock</td>
</tr>
<tr>
<td>AtomRace</td>
<td>concrete thread id.</td>
<td>concrete mem. addr.</td>
<td>O   O   Cond.   X</td>
<td>O   O   O   O   O</td>
<td>tracking lock/unlock</td>
</tr>
<tr>
<td>AVIO</td>
<td>concrete thread id.</td>
<td>concrete mem. addr.</td>
<td>O   O   Cond.   X</td>
<td>O   O   O   O   O</td>
<td>total execution order</td>
</tr>
<tr>
<td>Block-based algorithm</td>
<td>concrete thread id.</td>
<td>concrete mem. addr.</td>
<td>O   O   Cond.   X</td>
<td>O   O   O   O   O</td>
<td>tracking lock/unlock, message passing</td>
</tr>
<tr>
<td>Commit-node</td>
<td>concrete thread id.</td>
<td>concrete mem. addr.</td>
<td>O   O   O   O   X</td>
<td>O   O   O   O   O</td>
<td>tracking lock/unlock, message passing</td>
</tr>
<tr>
<td>HAVE</td>
<td>concrete thread id.</td>
<td>concrete mem. addr.</td>
<td>O   O   O   O   X</td>
<td>O   O   O   O   O</td>
<td>tracking lock/unlock, message passing</td>
</tr>
<tr>
<td>Kivati</td>
<td>concrete thread id.</td>
<td>concrete mem. addr.</td>
<td>O   O   O   O   X</td>
<td>O   O   O   O   O</td>
<td>total execution order</td>
</tr>
<tr>
<td>SVD</td>
<td>concrete thread id.</td>
<td>concrete mem. addr.</td>
<td>O   O   O   O   O</td>
<td>X   X   O   O   O</td>
<td>total execution order</td>
</tr>
<tr>
<td>PENEOPE</td>
<td>concrete thread id.</td>
<td>concrete mem. addr.</td>
<td>O   O   O   O   X</td>
<td>O   O   O   O   O</td>
<td>tracking lock/unlock</td>
</tr>
<tr>
<td>Velodrome</td>
<td>concrete thread id.</td>
<td>concrete mem. addr.</td>
<td>O   O   O   O   X</td>
<td>O   O   O   O   O</td>
<td>tracking lock/unlock, message passing</td>
</tr>
<tr>
<td>Atomizer</td>
<td>static approx. alias analysis</td>
<td>alias analysis</td>
<td>O   O   O   O   O</td>
<td>O   O   O   O   O</td>
<td>tracking lock/unlock</td>
</tr>
</tbody>
</table>
Race-2: Atomic Block Violation (5/5)

- Limitation: false-negative

- Limitations – false negative

Invariants/constraints over $x_1$ and $x_2$ might be broken
Race-3: Data Assoc. Violation (1/4)

- A unit of data can be located in two or more distinct variables.
- Race-3 detection techniques look for **inconsistent updates of associated variables**. [K. Havelund VVEIS03, S. Lu SOSP07, F. Tip ICSE08]

In [Savage et al., SOSP 1997],

*a data race* occurs when there exists two operations

1. executed by **two concurrent threads**,  
2. access **a shared variable** 
3. at least one access is write,  
4. no explicit mechanism to coordinate their execution order

---

From a previous slide for Race-1

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>→</th>
<th>memory area, set of variables (associated variables)</th>
<th>←</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>←</td>
<td></td>
<td>←</td>
<td></td>
</tr>
</tbody>
</table>

2011-06-27 다중 쓰레드 프로그램의 경쟁상태오류 검출기법 분류

Hong, Shin @ PSWLAB
Race-3: Data Assoc. Violation (2/4)

- Race-3 bug condition:

\[(C1)\] \( \text{thread}(p) \neq \text{thread}(q) \)

\[(C2)\] \( \exists v_1, v_2 \in V_s. (v_1 \in \text{opr}(p) \land v_2 \in \text{opr}(q) \land (v_1, v_2) \in A_{data}) \)

\[(C3)\] \( \text{conflict}(\text{opr}(p), \text{opr}(q)) \)

\[(C4)\] \( p \triangleright q \land q \triangleright p \)
Race-3: Data Assoc. Violation  (3/4)

- Example

class BankAccount_C {
    int balance;
    int debt;
    /* Invariant:
        (balance == 0 ∧ debt == 0) ∨
        (debt > 0 → balance == 0) ∨
        (balance > 0 → debt == 0) */
        (balance, debt) ∈ A_{data} ∧
        (debt, balance) ∈ A_{data}

    Lock m_balance;
    Lock m_debt;

    [balance=0, debt=10]
    --t1: deposit(20)--
    --t2: withdraw(5)--
    21: if(getBalance()==0)...

    11: lock(m_debt);
    12: if(0 < 10 && 10 <=20)
    13: tmp = 20-10;
    14: dept=0;
    15: unlock(m_debt);
    16: lock(m_balance);
    17: balance = 10;  \( p \)
    22: lock(m_debt);
    23: debt = 0+5;  \( q \)

    The invariant is violated:
    balance is 10 and debt is 5
Race-3: Data Assoc. Violation

- Race-3 detection techniques

| (C1) thread() | (C2) $A_{data}$ | (C3) conflict() | (C4) $
\n| \text{Type} | \text{transitive} | \text{symmetric} | \text{Source} | \text{tracking lock/unlock} |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>MultiRace</td>
<td>Concrete thread id.</td>
<td>$[a_1, a_2, \ldots, a_n]$</td>
<td>O</td>
<td>O</td>
</tr>
<tr>
<td>MUVI-Eraser</td>
<td>Heuristics</td>
<td>$&lt;(a_1, t_1), (a_2, t_2)&gt;$ where $t_1, t_2 \in {rd, wr, rd&amp;wr}$</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Object data race detection</td>
<td>Static approx.</td>
<td>${a_1, a_2, \ldots, a_n}$</td>
<td>O</td>
<td>O</td>
</tr>
</tbody>
</table>
• Race-4 techniques utilize both operation block and data association together to reduce false positives and false negatives.

In [Savage et al., SOSP 1997], a data race occurs when there exists two operations
(1) executed by two concurrent threads,
(2) access a shared variable
(3) at least one access is write,
(4) no explicit mechanism to coordinate their execution order
Race-4

Race-4 bug conditions:

1. Race-B conditions
2. Race-C conditions

(D1) $\text{thread}(p) \neq \text{thread}(q)$

(D2) $\exists v_1, v_2 \in V_S. (v_1 \in \text{opr}(p) \land v_2 \in \text{opr}(q)) \land (v_1, v_2) \in A_{data}$

(D3) $\text{conflict}(\text{optr}(p), \text{optr}(q))$

(D4) $\exists v_3, v_4 \in V_S. (v_3 \in \text{opr}(p') \land v_4 \in \text{opr}(q)) \land (v_3, v_4) \in A_{data}$

(D5) $\text{conflict}(\text{optr}(p'), \text{optr}(q))$

(D6) $\exists b_i \in B_{op}. (p, p') \in b_i$

(D7) $\exists v_5, v_6 \in V_S. (v_5 \in \text{opr}(p') \land v_6 \in \text{opr}(q)) \land (v_5, v_6) \in A_{data}$

(D8) $\text{conflict}(\text{optr}(p), \text{optr}(p'))$

(D9) $q \not\in p \land p' \not\in q$
Race-4

- Buggy program

class BankAccount_D {

    Lock m;
    int balance, debt;
    /*(balance, debt) ∈ A_{data}
        (debt, balance) ∈ A_{data} */

    int getBalance(int x) {
        int tmp;
        1: lock(m);
        2: tmp = balance;
        3: unlock(m);
        4: return tmp;
    }

    int withdraw(int x) {
        11: if (getBalance() == 0) {
            b1
            12: lock(m);
            13: debt = debt + x;
            14: unlock(m);
        }
        21: lock(m);
        22: if (0 == 0)
        23: if (20 > 10);
        24: balance = 20-10;
        25: debt = 0;
        26: ...
        27: unlock(m);
    }

    int deposit(int x) {
        21: lock(m);
        22: if (balance == 0) {
            23: if (x > debt) {
            24: balance = x - debt;
            25: debt = 0;
            26: ...
            27: unlock(m);
        }
    }

- Error execution scenario

<table>
<thead>
<tr>
<th>[balance=0, debt=10 ]</th>
</tr>
</thead>
<tbody>
<tr>
<td>--t1: withdraw(5) --</td>
</tr>
<tr>
<td>11: if (getBalance() == 0)</td>
</tr>
<tr>
<td>1: lock(m);</td>
</tr>
<tr>
<td>2: tmp = 0;</td>
</tr>
<tr>
<td>3: unlock(m);</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>--t2: deposit(20) --</th>
</tr>
</thead>
<tbody>
<tr>
<td>11: if (getBalance() == 0)</td>
</tr>
<tr>
<td>1: lock(m);</td>
</tr>
<tr>
<td>2: tmp = 0;</td>
</tr>
<tr>
<td>3: unlock(m);</td>
</tr>
<tr>
<td>21: lock(m);</td>
</tr>
<tr>
<td>22: if (0 == 0)</td>
</tr>
<tr>
<td>23: if (20 &gt; 10);</td>
</tr>
<tr>
<td>24: balance = 20-10;</td>
</tr>
<tr>
<td>25: debt = 0;</td>
</tr>
</tbody>
</table>
| 26: ...
| 27: unlock(m); |

The invariant is violated: balance is 10 and debt is 15
Race-4 detection techniques

<table>
<thead>
<tr>
<th>Method</th>
<th>(D1) thread()</th>
<th>(D2, D4, D7) $A_{data}$</th>
<th>(D3, D5) conflict() where thread($p$) $\neq$ thread($q$)</th>
<th>(D6, D8) conflict() where thread($p$) $\neq$ thread($q$)</th>
<th>(D9) total execution order/lock/unlock</th>
</tr>
</thead>
<tbody>
<tr>
<td>Atomic-Set serializability</td>
<td>Concrete thread id.</td>
<td>$&lt; a_1, a_2&gt;$</td>
<td>O</td>
<td>O</td>
<td>W-R, R-W, W-W</td>
</tr>
<tr>
<td>ColorSafe</td>
<td>Concrete thread id.</td>
<td>$&lt; a_1, a_2&gt;$</td>
<td>O</td>
<td>O</td>
<td>W-R, R-W, W-W</td>
</tr>
<tr>
<td>Method-consistency</td>
<td>Static approx.</td>
<td>$(a_1, t_1), (a_2, t_2)$</td>
<td>O</td>
<td>O</td>
<td>W-R, R-W, W-W</td>
</tr>
<tr>
<td>MUVI-AVIO</td>
<td>Concrete thread id.</td>
<td>$(a_1, t_1), (a_2, t_2)$</td>
<td>X</td>
<td>X</td>
<td>W-R, R-W, W-W(cond), W-W, W-W, W-W, R-R</td>
</tr>
<tr>
<td>View-consistency</td>
<td>Concrete thread id.</td>
<td>$(a_1, t_1), (a_2, t_2)$</td>
<td>O</td>
<td>O</td>
<td>W-R, R-W, W-W</td>
</tr>
</tbody>
</table>
Implications to Better Race Detection (1/2)

- Relations in four class of race detections

\begin{figure}
\centering
\includegraphics[width=\textwidth]{race_detection_diagram.png}
\caption{Diagram illustrating the relations in four classes of race detections.}
\end{figure}
Implications to Better Race Detection (2/2)

- Static analyses can be applied for much precise race detections
  - Only few work use static analyses for inferring/checking
    operation block and data associations

\[ \text{Theorem 1. Let } G^{du} = (G, \Sigma, D, U) \text{ be a def/use graph with } G = (N, E), \text{ and let } u, v \in N. \text{ If } u \text{ is semantically dependent on } v \text{ and this semantic dependence is finitely demonstrated, then } u \text{ is syntactically dependent on } v^2. \]

Theorem 1 is significant because it shows that, given appropriate definitions of control and data dependence, syntactic dependence is a necessary condition for (finitely demonstrated) semantic dependence. Thus, the theorem provides justification for algorithms that use syntactic dependence to approximate semantic dependence. We refer to this desirable relationship between syntactic and semantic dependence as the “syntactic–semantic relationship”.