Automated Testing & Verification

Interprocedural Dataflow Analysis

Galeotti/Gorla/Rau
Saarland University
How to handle many methods?

- How do we know “divByX” does not fail?

```java
int divByX(int x) {
    [result := 10/x]1;
}

void caller1() {
    [x := 5]1;
    [y := divByX(x)]2;
    [y := divByX(5)]3;
    [y := divByX(1)]4;
}
```

- How do we know “area” does not fail?

```java
float area(Square c) {
    [result := c.l*c.l]1;
}

void caller1(Square c2) {
    [Square c = new Square()]1;
    [float a1 = area(c)]2;
    [return area(c2)+a1]3;
}
```
Interprocedural Dataflow Analysis

- Analyze a program with many methods
- Strategies:
  - Build an interprocedural CFG
  - Assume/Guarantee
  - Context sensitivity
    - Inlining
    - Call string
    - Compute “summaries”
- Extend CFG for many procedures
  - Add an edge from the **caller** to the **callee**'s entry node
  - Add an edge from the return node to the call's next node

```c
int double(int x) {
    [result := 2*x]_1;
}
int divByX(int x) {
    [result := 10/x]_1;
}
void caller1() {
    [x := 5]_1;
    [y := double(x)]_2;
    [z := divByX(y)]_3;
}
```
caller 1 Analysis

<table>
<thead>
<tr>
<th>Pos</th>
<th>caller1</th>
<th>double</th>
<th>divByX</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>x</td>
<td>y</td>
<td>z</td>
</tr>
</tbody>
</table>

1: \( x := 5 \)

2: \( y := \text{double}(x) \)

3: \( z := \text{divByX}(y) \)

1: \( \text{result} := 10 / x \)

\( X = X_{\text{caller1}} \)

\( x = x_{\text{caller1}} \)

\( y = \text{result} \)

\( z = \text{result} \)
Another example

int double(int x) {
    [result := 2*x]_1;
}
void caller1() {
    [x := 5]_1;
    [y := double(x)]_2;
    [z := 10/y]_3
    [x := 0]_4;
    [y := double(x)]_5;
}

Any problem with this?
Another example

Warning: div by zero!
- False positive!
The interprocedural CFG loses precision.
It can not distinguish different context calls.
In particular, information may flow through infeasible paths.

1: \( \text{x} := 5 \)
2: \( \text{y} := \text{double}(\text{x}) \)
3: \( \text{z} := 10/\text{y} \)
4: \( \text{x} := 0 \)
5: \( \text{y} := \text{double}(\text{x}) \)

1: \( \text{result} := 2 \times \text{x} \)
We have to distinguish among several context calls

Many techniques for this:
- Inlining/Cloning
- Call chains
- Assume/Guarantee
- Method Summaries
**Inlining/Cloning**

- **Idea:** Use dataflow, but rule out incorrect program paths

- **How?**
  - **Inlining:** Include the callee’s code within the caller
    - The program has an unique method now
    - Instantiate input/output parameters
  
  - **Cloning:** Create a copy of the method for each invocation.
    - Each copy represents a different context
### Inlining/Cloning

- **Cloning**
  ```c
  int divByX(int x) {
      result := 10/x;
  }
  void caller1() {
      [x := 5];
      [y := divByX(x)];
      [y := divByX(5)];
      [y := divByX(0)];
  }
  ```

- **Inlining**
  ```c
  int divByX(int x) {
      result := 10/x;
  }
  void caller1() {
      [x := 5];
      [y := divByX(x)];
      [y := divByX(5)];
      [y := divByX(0)];
  }
  ```

- **Advantages:**
  - More precision
  - A version of the method for each context

- **Disadvantages:**
  - More computational cost
  - Code explosion
- Interface usage

```java
Interface I
{
    int getValue();
}
Class A implements I...
Class B implements I...

void process() {
    I i = null;
    if(...) 
        i = new A();
    else
        i = new B();
    i.getValue();
}
```

- Recursion

```java
void process(int v, int x) {
    if(v==0) return x;
    else return process(v-1,x*v)
}
```
We have to distinguish among several context calls

- Many techniques for this:
  - Inlining/Cloning
  - Call chains
  - Assume/Guarantee
  - Method Summaries
During runtime, different method invocations are distinguished through the call stack
- In particular, we might distinguish them through control labels (example: m1.4+m2.1+....)

Disadvantage
- The stack might be unbounded
  - Recursion

Idea:
- Use only the last $k$ calls for distinguishing contexts
- We call this “$k$-limit”
What will happen if we double calls to another method?

- int double(int x) { return g(x); }
- int g(int x) { return (2*x); }
Recursion & call chains

Is it possible to find a “k” that allows us to precisely handle this problem?

- m_1.g_1F.double
- m_1. g_1T.g_1F.double
- m_1. g_1T. g_1T.g_1F.double
- ...

Call chains require approximation in case of recursive calls
Recap:

- We introduce to the abstract value the notion of context using a chain that models the last invocations
  - It is known as \textit{k-limiting}

- For non-recursive programs it could be precise enough
  - Although it is not as precise as possible! Why?
    - (loops!)

- For recursive programs contexts we have to approximate
  - Example: \texttt{m_1. (g_1T)*.g_1F.double}

- Although it seems a rather obvious solution, it is not widely used
  - Cost: multiplies the number of states given the number of paths (exponential!)
  - Depends on having a good call-graph approximation
Context Sensibility

- We have to distinguish among several context calls
- Many techniques for this:
  - Inlining/Cloning
  - Call chains
  - Assume/Verify
  - Method Summaries
Idea: annotate each method with information about what it assumes and what it guarantees

- **Precondition**: Initial values for all parameters
- **Postcondition**: A return value (result)
- Based on the programmer knowledge
  - Individual knowledge for each method.
- Or by “default”
  - example: every “equals” implementation

**Verification**

- In the annotated method
  - Assume all values for parameters
  - Verify in the annotated method that the result $\subseteq$ assumed result
- In the caller method
  - Verify $\text{arg} \subseteq \text{assumed}_{\text{arg}}$
    - Actual parameters satisfy assumptions on formal parameters
  - Use the annotated valued as result.
Example: Zero analysis

- Default: MZ (top) for all arguments and results
  - Meaning: The method can receive any value and return any value
  - Benefit: All parameters and result satisfy this assumption
  - Cost: Too conservative wrt. the actual input/output
    - Many false positives
We assume \( x \) must be \( \text{MZ} \) and result is \( \text{MZ} \)

```c
int divByX(int x) {
    [result := 10/x]_1;
}

void caller1() {
    [x := 5]_1;
    [y := divByX(x)]_2;
}
```

**divByX analysis**

<table>
<thead>
<tr>
<th>pos</th>
<th>( x )</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>MZ</td>
<td>⊥</td>
</tr>
<tr>
<td>1</td>
<td>MZ</td>
<td>NZ</td>
</tr>
</tbody>
</table>

- It holds \( \sigma(\text{result}) \subseteq \text{MZ} \)
- Warning: div by zero at \#1

**Caller1 Analysis**

<table>
<thead>
<tr>
<th>pos</th>
<th>( x )</th>
<th>( y )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>⊥</td>
<td>⊥</td>
</tr>
<tr>
<td>1</td>
<td>NZ</td>
<td>⊥</td>
</tr>
</tbody>
</table>

- It holds \( \sigma(\text{x}) \subseteq \text{MZ} \)
- Problem: div by zero is not possible!
We assume $x$ must be **NZ** and result must be **NZ**

```c
int divByX(int x) {
    [result := 10/x]_1;
}
void caller1() {
    [x := 5]_1;
    [y := divByX(x)]_2;
}
```

### divByX analysis

<table>
<thead>
<tr>
<th>pos</th>
<th>x</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NZ</td>
<td>⊥</td>
</tr>
<tr>
<td>1</td>
<td>NZ</td>
<td>NZ</td>
</tr>
</tbody>
</table>

- No warning $s$
- It holds $\sigma(result) \subseteq \text{NZ}$

### Caller1 analysis

<table>
<thead>
<tr>
<th>pos</th>
<th>x</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>⊥</td>
<td>⊥</td>
</tr>
<tr>
<td>1</td>
<td>NZ</td>
<td>⊥</td>
</tr>
</tbody>
</table>

- It holds $\sigma(x) \subseteq \text{NZ}$
Optimistic assumptions

- We assume x must be NZ and result is NZ

```c
int double(int x) {
    [result := 2*x]_1;
}

void caller1() {
    [x := 0]_1;
    [y := double(x)]_2;
}
```

- Double analysis

<table>
<thead>
<tr>
<th>pos</th>
<th>x</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NZ</td>
<td>⊥</td>
</tr>
<tr>
<td>1</td>
<td>NZ</td>
<td>NZ</td>
</tr>
</tbody>
</table>

- It holds \( \sigma(\text{result}) \subseteq \text{NZ} \)

- Caller1 analysis

<table>
<thead>
<tr>
<th>pos</th>
<th>x</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>⊥</td>
<td>⊥</td>
</tr>
<tr>
<td>1</td>
<td>Z</td>
<td>⊥</td>
</tr>
</tbody>
</table>

- \( \sigma(x) \subseteq \text{NZ} \) fails!
- It does not satisfy the assumption
- It is a false positive, although the program is OK.
Instead of defaults, we use method-level annotations
  - The programmer provides these annotations

Annotation
  - **Precondition**: Initial abstract values for all parameters
  - **Postcondition**: A return value (result)

Verification
  - In the annotated method
    - Assume all values for all parameters
    - Verify that result $\subseteq$ assumed_result
  - In the caller:
    - Verify that args $\subseteq$ assumed_args
      - Actual parameters satisfy the assumption on formal parameters
      - Use annotated value as result.
Example user-defined annotations

@NZ int divByX(@NZ int x)
{
    [result := 10/x]_1;
}

void caller1() {
    [x := 5]_1;
    [y := divByX(x)]_2;
}

- **divByX Analysis**

<table>
<thead>
<tr>
<th>pos</th>
<th>x</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NZ</td>
<td>⊥</td>
</tr>
<tr>
<td>1</td>
<td>NZ</td>
<td>NZ</td>
</tr>
</tbody>
</table>

- It holds $\sigma(\text{result}) \subseteq \text{NZ}$

- **Caller1 analysis**

<table>
<thead>
<tr>
<th>pos</th>
<th>x</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>⊥</td>
<td>⊥</td>
</tr>
<tr>
<td>1</td>
<td>NZ</td>
<td>⊥</td>
</tr>
</tbody>
</table>

- It verifies that $\sigma(x) \subseteq \text{NZ}$
Example

```c
@NZ int double(@NZ int x)
{
    [result := 2*x]_1;
}
void caller1() {
    [x := o]_1;
    [y := double(x)]_2;
}
```

- **double analysis**

<table>
<thead>
<tr>
<th>pos</th>
<th>x</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NZ</td>
<td>⊥</td>
</tr>
<tr>
<td>1</td>
<td>NZ</td>
<td>NZ</td>
</tr>
</tbody>
</table>

- It holds $\sigma(\text{result}) \subseteq \text{NZ}$

- **Caller1 analysis**

<table>
<thead>
<tr>
<th>pos</th>
<th>x</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>⊥</td>
<td>⊥</td>
</tr>
<tr>
<td>1</td>
<td>Z</td>
<td>⊥</td>
</tr>
</tbody>
</table>

- $\sigma(x) \subseteq \text{NZ}$ fails!
- The double precondition is not met
- It is a false positive, although the program is OK
Example

[@MZ int double(@MZ int x)
{
    [result := 2*x];
}

void caller1() {
    [x := 5];
    [y := double(x)];
    [z := 10/y];
}

- **Double Analysis**

<table>
<thead>
<tr>
<th>pos</th>
<th>x</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>MZ</td>
<td>⊥</td>
</tr>
<tr>
<td>1</td>
<td>MZ</td>
<td>MZ</td>
</tr>
</tbody>
</table>

- **Caller1 analysis**

<table>
<thead>
<tr>
<th>pos</th>
<th>x</th>
<th>y</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>⊥</td>
<td>⊥</td>
<td>⊥</td>
</tr>
<tr>
<td>1</td>
<td>NZ</td>
<td>⊥</td>
<td>⊥</td>
</tr>
</tbody>
</table>

- It holds $\sigma(\text{result}) \subseteq MZ$

- $\sigma(x) \subseteq MZ$ is verified
- **Warning: div by zero!**
- It is also a false positive!
We have to distinguish among several context calls

Many techniques for this:
- Inlining/Cloning
- Call chains
- Assume/Guarantee
- Method Summaries
Idea:
  - Compute a **unique summary for each method**
  - Map dataflow input information to dataflow output information

Context sensitive
  - Given different input, they output **different** results
  - Instantiate the summary on each method invocation
Abstract Summaries

- Represent symbolically the effect of the function over the elements of the lattice
### Caller1 analysis

<table>
<thead>
<tr>
<th>pos</th>
<th>x</th>
<th>Y</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>⊥</td>
<td>⊥</td>
<td>⊥</td>
</tr>
<tr>
<td>1</td>
<td>NZ</td>
<td>⊥</td>
<td>⊥</td>
</tr>
<tr>
<td>3</td>
<td>NZ</td>
<td>NZ</td>
<td>NZ</td>
</tr>
<tr>
<td>4</td>
<td>Z</td>
<td>NZ</td>
<td>NZ</td>
</tr>
</tbody>
</table>

### double: case x:NZ → result: NZ

<table>
<thead>
<tr>
<th>pos</th>
<th>x</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NZ</td>
<td>⊥</td>
</tr>
<tr>
<td>1</td>
<td>NZ</td>
<td>NZ</td>
</tr>
</tbody>
</table>

### double: case x:Z → result: Z

<table>
<thead>
<tr>
<th>pos</th>
<th>x</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Z</td>
<td>⊥</td>
</tr>
<tr>
<td>1</td>
<td>Z</td>
<td>Z</td>
</tr>
</tbody>
</table>
@Case("x:NZ -> result:NZ")
@Case("x:Z -> result:Z")

int double(int x) {
    [result := 2*x]_1;
}

void caller1() {
    [x := 5]_1;
    [y := double(x)]_2;
    [z := 10/y]_3
    [x := 0]_4;
    [y := double(x)]_5;
}

Verify:

<table>
<thead>
<tr>
<th>pos</th>
<th>x</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NZ</td>
<td>⊥</td>
</tr>
<tr>
<td>1</td>
<td>NZ</td>
<td>NZ</td>
</tr>
</tbody>
</table>

Verify for caller 1
Case x:NZ

<table>
<thead>
<tr>
<th>Pos</th>
<th>x</th>
<th>y</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>NZ</td>
<td>⊥</td>
<td>⊥</td>
</tr>
</tbody>
</table>

verify:

<table>
<thead>
<tr>
<th>pos</th>
<th>x</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Z</td>
<td>⊥</td>
</tr>
<tr>
<td>1</td>
<td>Z</td>
<td>Z</td>
</tr>
</tbody>
</table>

Verify for caller 1
Case x:Z
Summary:
Case x: A → result: A

```c
int double(int x) {
    [result := 2*x]_1;
}
void caller1() {
    [x := 5]_1;
    [y := double(x)]_2;
    [z := 10/y]_3
    [x := 0]_4;
    [y := double(x)]_5;
}
```

**Caller1 Analysis**

<table>
<thead>
<tr>
<th>pos</th>
<th>x</th>
<th>y</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>⊥</td>
<td>⊥</td>
<td>⊥</td>
</tr>
<tr>
<td>1</td>
<td>NZ</td>
<td>⊥</td>
<td>⊥</td>
</tr>
<tr>
<td>2</td>
<td>NZ</td>
<td>NZ</td>
<td>NZ</td>
</tr>
<tr>
<td>3</td>
<td>NZ</td>
<td>NZ</td>
<td>NZ</td>
</tr>
<tr>
<td>4</td>
<td>Z</td>
<td>NZ</td>
<td>NZ</td>
</tr>
</tbody>
</table>

**double:**

<table>
<thead>
<tr>
<th>pos</th>
<th>x</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>A</td>
<td>⊥</td>
</tr>
<tr>
<td>1</td>
<td>A</td>
<td>A</td>
</tr>
</tbody>
</table>

Summary:
Case x: A → result: A
The call-graph is traversed in a Bottom-Up walk:
- Starting by leafs (no further method invokations)
  - Perform intraprocedural dataflow analysis
  - Store summary
- When a method makes a call
  - Look for summary (bottom-up traverse ensures it exists)
  - Instantiate it with actual parameters (top-down)

For recursive methods (or cycles) requires another fix-point
- On the recursive subcomponent
  - Cycle in the Call Graph
A “map” to know which methods to analyze
  - Paramount in object oriente programs

```java
static void main(){
    B b1 = new B();
    A a1 = new A();
    f(b1);
    g(b1);
}
static void f(A a2){
    a2.foo();
}
static void g(B b2){
    B b3 = b2;
    b3 = new C();
    b3.foo();
}
```

```java
class A {
    foo(){..}
}
class B extends A{
    foo() {...}
}
class C extends B{
    foo() {...}
}
class D extends B{
    foo(){...}
}
```
A “map” to know which methods to analyze

- Paramount in object-oriented programs

```java
static void main()
{
    B b1 = new B();
    A a1 = new A();
    f(b1);
    g(b1);
}
static void f(A a2)
{
    a2.foo();
}
static void g(B b2)
{
    B b3 = b2;
    b3 = new C();
    b3.foo();
}
class A {
    foo() {..}
}
class B extends A{
    foo() {...}
}
class C extends B{
    foo() {...}
}
class D extends B{
    foo() {...}
}
```

Computed using Class Hierarchy Analysis (CHA)
A “map” to know which methods to analyze
- Paramount in object oriented programs

```java
static void main(){
    B b1 = new B();
    A a1 = new A();
    f(b1);
    g(b1);
}
static void f(A a2){
    a2.foo();
}
static void g(B b2){
    B b3 = b2;
    b3 = new C();
    b3.foo();
}

class A {
    foo(){..}
}
class B extends A{
    foo() {...}
}
class C extends B{
    foo() {...}
}
class D extends B{
    foo() {...}
}
```

Computed using Rapid Type Analysis (RTA)
A “map” to know which methods to analyze
- Paramount in object oriented programs

```java
static void main(){
    B b1 = new B();
    A a1 = new A();
    f(b1);
    g(b1);
}
static void f(A a2){
    a2.foo();
}
static void g(B b2){
    B b3 = b2;
    b3 = new C();
    b3.foo();
}
class A {
    foo(){..}
}
class B extends A{
    foo() {...}
}
class C extends B{
    foo() {...}
}
class D extends B{
    foo(){...}
}
```

Computed using XTA
Final remarks

- Assumptions
  - Simple & efficient
  - Imprecise (too general)

- Annotations
  - Requires effort
  - More precise than assumptions
  - More efficient than interprocedural analysis

- A whole-program analysis is not required!

- CFG Interprocedural
  - Easy to implement
  - Imprecise
  - Can be too expensive
  - As precise as simple

- Summaries
  - Very precise
  - Vert expensive if no abstraction is done

- They require whole-program analysis
Bibliography

- **Compilers: Principles, Techniques & Tools 2nd Edition**: Aho, Lam, Sethi, Ullman
- **Principles of Program Analysis**: Flemming Nielson, Hanne Riis Nielson, Chris Hankin.
- **Modern compiler implementation in Java**: Andrew Appel. 2nd Edition.