Test Case Generation by Symbolic Execution: Basic Concepts, a CLP-based Instance, and Actor-based Concurrency

Elvira Albert
Complutense University of Madrid
elvira@sip.ucm.es

SFM-14: ESM
Bertinoro, 16-20 June, 2014

http://www.envisage-project.eu
Introduction: Test Case Generation

- Testing: vital part of the software development process
- Three recent factors have made it take more central role:
  1. introduction of testing environments (e.g., JUnit)
  2. increasingly complex systems are being built
  3. there is a growing tendency to prove software correctness
Introduction: Test Case Generation

- Testing: vital part of the software development process
- Three recent factors have made it take more central role:
  1. introduction of testing environments (e.g., JUnit)
  2. increasingly complex systems are being built
  3. there is a growing tendency to prove software correctness
- TCG: automatic generation of a collection of test-cases to be applied to a system under test.
- Ensure certain coverage criterion: heuristics to estimate how well the program is exercised by a test suite.
  - statement coverage: each line of the code is executed,
  - path coverage: every possible trace is executed,
  - loop-\(k\): limit to a threshold \(k\) the number of times we iterate on loops
White-box Test Case Generation

Several classifications of testing techniques:

- Random vs. non-random ⇒ difficult to obtain high degree of code coverage in random unless consider huge number of inputs
- Black-box vs. white-box ⇒ test cases obtained from specifications vs. from program
- Dynamic vs. static ⇒ depending if input variables are instantiated
Several classifications of testing techniques:

- **Random vs. non-random** ⇒ difficult to obtain high degree of code coverage in random unless consider huge number of inputs
- **Black-box vs. white-box** ⇒ test cases obtained from specifications vs. from program
- **Dynamic vs. static** ⇒ depending if input variables are instantiated

- **Static white-box TCG**
  - **Symbolic Execution**
  - Execution with symbolic values ⇒ constrained variables
  - Non-determinism due to branching instructions involving unknown data
  - Termination criterion ⇒ **loop-k**
  - Path coverage
  - Result: Path conditions or equivalence classes of inputs
White-box Test Case Generation

Several classifications of testing techniques:

- Random vs. non-random ⇒ difficult to obtain high degree of code coverage in random unless consider huge number of inputs
- Black-box vs. white-box ⇒ test cases obtained from specifications vs. from program
- Dynamic vs. static ⇒ depending if input variables are instantiated

Static white-box TCG

- **Symbolic Execution**
  - Execution with symbolic values ⇒ constrained variables
  - Non-determinism due to branching instructions involving unknown data
  - Termination criterion ⇒ **loop-k**
- Path coverage
- Result: **Path conditions or equivalence classes of inputs**
Plan of the Lecture

- **Part 1: Symbolic execution and TCG**
  - Introduction
  - Handling heap-manipulating programs
  - Compositionallity

- **Part 2: CLP-based TCG**
  - Introduction
  - Translation from imperative to CLP
  - Guided-TCG
  - Demo

- **Part 3: TCG of Concurrent (Actor) Programs**
  - The path exploitation problem
  - Symbolic execution and TCG for actors
  - Demo
Plan of the Lecture

- Part 1: Symbolic execution and TCG
  - Introduction
  - Handling heap-manipulating programs
  - Compositionallity

- Part 2: CLP-based TCG
  - Introduction
  - Translation from imperative to CLP
  - Guided-TCG
  - Demo

- Part 3: TCG of Concurrent (Actor) Programs
  - The path explotion problem
  - Symbolic execution and TCG for actors
  - Demo
Symbolic Execution

- King [Comm. ACM 1976], Clarke [IEEE TSE 1976]
- Analysis of programs with unspecified inputs
- Symbolic states represent sets of concrete states
  - Variables carry symbolic expressions instead of concrete values
- For each path, build path condition
  - Condition on inputs, for the execution to follow that path
  - Check path condition satisfiability, explore only feasible paths
- Renewed interest in recent years
- Applications: test-case generation, error detection,...
- Tools: CUTE and jCUTE (UIUC), EXE and KLEE (Stanford),
  CREST and BitBlaze (UC Berkeley), Pex, SAGE, YOGI and PREfix
  (Microsoft), PET (UCM-UPM), SPF (Symbolic Pathfinder, NASA
  Ames),...
Elements Involved in the Testing Process

Java Code

```java
int abs(int x) {
    if (x >= 0) return x;
    else return -x;
}
```

Test Cases

\{ ⟨X \geq 0, Z = X⟩, ⟨X < 0, Z = -X⟩\}

Concrete Inputs

\{ ⟨X = 1, Z = 1⟩, ⟨X = -1, Z = 1⟩\}

JUnit Code

```java
void test_abs(){
    assertEquals(abs(1),1);
    assertEquals(abs(-1),1);
}
```
## Elements Involved in the Testing Process

### Java Code

```java
int abs(int x){
    if (x >= 0) return x;
    else return -x;
}
```

### Test Cases

{ ⟨X >= 0,Z = X⟩,
  ⟨X < 0,Z = -X⟩ }

---

Elvira Albert

Test Case Generation by Symbolic Execution

16-20 June 2014
## Elements Involved in the Testing Process

### Java Code

```java
int abs(int x) {
    if (x >= 0) return x;
    else return -x;
}
```

### Test Cases

```plaintext
\{ \langle X \geq 0, Z = x \rangle, \\
    \langle X < 0, Z = -x \rangle \}
```

### Concrete Inputs

```plaintext
\{ \langle X = 1, Z = 1 \rangle, \\
    \langle X = -1, Z = 1 \rangle \}
```
Elements Involved in the Testing Process

Java Code

```java
int abs(int x){
    if (x >= 0) return x;
    else return -x;
}
```

Test Cases

```java
{ ⟨X >= 0,Z = X⟩,
  ⟨X < 0,Z = -X⟩ }
```

Concrete Inputs

```java
{ ⟨X = 1, Z = 1⟩,
  ⟨X = -1, Z = 1⟩ }
```

JUnit Code

```java
void test_abs(){
    assertEquals(abs(1),1);
    assertEquals(abs(-1,1));
}
```
Java source code

```java
int exp(int a, int n) {
  if (n < 0) {
    throw new Exception();
  } else {
    int r = 1;
    while (n > 0) {
      r = r*a;
      n--;
    }
    return r;
  }
}
```
Java source code

```java
int exp(int a, int n) {
    if (n < 0) {
        throw new Exception();
    } else {
        int r = 1;
        while (n > 0) {
            r = r * a;
            n--;
        }
        return r;
    }
}
```

Symbolic Execution Tree
Java source code

```java
int exp(int a, int n) {
    if (n < 0)
        throw new Exception();
    else {
        int r = 1;
        while (n > 0) {
            r = r * a;
            n--;
        }
        return r;
    }
}
```

**Test cases**

<table>
<thead>
<tr>
<th>#</th>
<th>Input</th>
<th>Output</th>
<th>Path condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>[A, N]</td>
<td>[exception]</td>
<td>{N &lt; 0}</td>
</tr>
<tr>
<td>2</td>
<td>[A, N]</td>
<td>1</td>
<td>{N = 0}</td>
</tr>
<tr>
<td>3</td>
<td>[A, N]</td>
<td>R</td>
<td>{N &gt; 0, N' = N - 1, N' &lt;= 0, R = 1 * A}</td>
</tr>
</tbody>
</table>

**Symbolic Execution Tree**

```
while (n > 0) {
    r = r * a;
    n--;
}
return r;
```
### Java source code

```java
int exp(int a, int n) {
    if (n < 0)
        throw new Exception();
    else {
        int r = 1;
        while (n > 0) {
            r = r * a;
            n--;
        }
        return r;
    }
}
```

### Test cases

<table>
<thead>
<tr>
<th>#</th>
<th>Input</th>
<th>Output</th>
<th>Path condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>[A, N]</td>
<td>exception</td>
<td>{N&lt;0}</td>
</tr>
<tr>
<td>2</td>
<td>[A, N]</td>
<td>1</td>
<td>{N=0}</td>
</tr>
<tr>
<td>3</td>
<td>[A, N]</td>
<td>R</td>
<td>{N&gt;0, N'=N-1, N'&lt;=0, R=1*A}</td>
</tr>
</tbody>
</table>

### Concrete inputs

<table>
<thead>
<tr>
<th>#</th>
<th>Input</th>
<th>Output</th>
<th>Path condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>[-10, -10]</td>
<td>Exception</td>
<td>{N=0, R=1}</td>
</tr>
<tr>
<td>2</td>
<td>[-10, 0]</td>
<td>1</td>
<td>{N'=N-1, R'=R*A}</td>
</tr>
<tr>
<td>3</td>
<td>[-10, 1]</td>
<td>-10</td>
<td>{N'\leq 0}</td>
</tr>
</tbody>
</table>

### Symbolic Execution Tree

- \text{N<0?} \\
  - \text{N<0} \\
  - \text{exception} \\
- \text{N\geq 0} \\
  - \text{exc} \\
  - \text{N'=0, R=1} \\
  - \text{N'>0, N'=N-1, N'<=0, R=1*A} \\

- \text{ok} \\
  - \text{N' \leq 0?} \\
  - \text{N'\leq 0} \\
  - \text{R'=R*A} \\
  - \text{N'' \leq 0?} \\

- \text{ok}
Termination Criteria

Java source code

```java
int exp(int a, int n) {
    if (n < 0)
        throw new Exception();
    else {
        int r = 1;
        while (n > 0) {
            r = r * a;
            n--;
        }
        return r;
    }
}
```

Unit tests (JUnit)

```java
public void test_1(){
    int input0 = -10, input1 = -10;
    try {
        int output = Test.intExp(input0,input1);
    } catch (Exception ex) {
        assertEquals("exception","ArithmeticException",
                    ex.getClass().getName());
    } return;
}
```

```java
public void test_2(){
    int input0 = -10, input1 = 0;
    int output = Test.intExp(input0,input1);
    int expected = 1;
    assertEquals("OK",expected,output);
}
```

```java
public void test_3(){
    int input0 = -10, input1 = 1;
    int output = Test.intExp(input0,input1);
    int expected = -10;
    assertEquals("OK",expected,output);
}
```

Test cases

<table>
<thead>
<tr>
<th>#</th>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>[-10, -10]</td>
<td>[exception]</td>
</tr>
<tr>
<td>2</td>
<td>[-10, 0]</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>[-10, 1]</td>
<td>-10</td>
</tr>
</tbody>
</table>

Concrete input

<table>
<thead>
<tr>
<th>#</th>
<th>Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>[-10, -10]</td>
</tr>
<tr>
<td>2</td>
<td>[-10, 0]</td>
</tr>
<tr>
<td>3</td>
<td>[-10, 1]</td>
</tr>
</tbody>
</table>
Plan of the Lecture

- Part 1: Symbolic execution and TCG
  - Introduction
  - Handling heap-manipulating programs
  - Compositionallity
- Part 2: CLP-based TCG
  - Introduction
  - Translation from imperative to CLP
  - Guided-TCG
  - Demo
- Part 3: TCG of Concurrent (Actor) Programs
  - The path explotion problem
  - Symbolic execution and TCG for actors
  - Demo
Handling Heap-manipulating Programs

Challenge: Efficiently handling heap-manipulating programs
Handling Heap-manipulating Programs

- Challenge: Efficiently handling heap-manipulating programs
  - Complex dynamic data structures
Handling Heap-manipulating Programs

- Challenge: Efficiently handling heap-manipulating programs
  - Complex dynamic data structures
  - Aliasing of references
Handling Heap-manipulating Programs

- Challenge: Efficiently handling heap-manipulating programs
  - Complex dynamic data structures
  - Aliasing of references
  - Explore all possible heap shapes
Challenge: Efficiently handling heap-manipulating programs

- Complex dynamic data structures
- Aliasing of references
- Explore all possible heap shapes
- Path explosion problem
handling heap-manipulating programs

- Challenge: Efficiently handling heap-manipulating programs
  - Complex dynamic data structures
  - Aliasing of references
  - Explore all possible heap shapes
  - Path explosion problem
  - Outperform Lazy Initialization
m(C x, C y, C z)
{
    x.f = 1;
    z.f = -5;
    y.f = x.f + 1;
    m2();
    if (x == z)
        m3(y.f);
    else
        m4(y.f);
}

- Standard technique to handle aliasing. Used in state-of-the-art systems, e.g., PET (UCM&UPM) and SPF (NASA Ames)
m(C x, C y, C z)
{
    x.f = 1;
    z.f = -5;
    y.f = x.f+1;
    m2();
    if (x==z)
        m3(y.f);
    else
        m4(y.f);
}

▶ Standard technique to handle aliasing. Used in state-of-the-art systems, e.g., PET (UCM&UPM) and SPF (NASA Ames)
m(C x, C y, C z)
{
    x.f = 1;
    z.f = -5;
    y.f = x.f+1;
    m2();
    if (x==z)
        m3(y.f);
    else
        m4(y.f);
}

▶ Field accesses on unknown references trigger non-determinism: 1) Null 2) New reference 3) Each aliasing possibility
m(C x, C y, C z)
{
    x.f = 1;
    z.f = -5;
    y.f = x.f+1;
    m2();
    if (x==z)
        m3(y.f);
    else
        m4(y.f);
}

- Field accesses on unknown references trigger non-determinism:
  1) Null
  2) New reference
  3) Each aliasing possibility
Test Case Generation by Symbolic Execution

Lazy Initialization vs Heap Solver

```c
m(C x, C y, C z)
{
    x.f = 1;
    z.f = -5;
    y.f = x.f + 1;
    m2();
    if (x == z)
        m3(y.f);
    else
        m4(y.f);
}
```

- Field accesses on unknown references trigger non-determinism: 1) Null 2) New reference 3) Each aliasing possibility
\( m(C \ x, C \ y, C \ z) \)
\[
\begin{align*}
x.f &= 1; \\
z.f &= -5; \\
y.f &= x.f+1; \\
m2(); \\
\text{if} \ (x==z) \\
&\quad m3(y.f); \\
\text{else} \\
&\quad m4(y.f);
\end{align*}
\]

- Field accesses on unknown references trigger non-determinism: 1) Null 2) New reference 3) Each aliasing possibility
m(C x, C y, C z)
{
    x.f = 1;
    z.f = -5;
    y.f = x.f+1;
    m2();
    if (x==z)
        m3(y.f);
    else
        m4(y.f);
}

► Field accesses on unknown references trigger non-determinism: 1) Null 2) New reference 3) Each aliasing possibility
Test Case Generation by Symbolic Execution

Lazy Initialization vs Heap Solver

```c
m(C x, C y, C z)
{
    x.f = 1;
    z.f = -5;
    y.f = x.f + 1;
    m2();
    if (x == z)
        m3(y.f);
    else
        m4(y.f);
}
```

- Field accesses on unknown references trigger non-determinism: 1) Null 2) New reference 3) Each aliasing possibility
m(C x, C y, C z)
{
  x.f = 1;
  z.f = -5;
  y.f = x.f + 1;
  m2();
  if (x == z)
    m3(y.f);
  else
    m4(y.f);
}

- Symbolic execution quickly becomes impractical
- Redundant exploration of large number of paths
Test Case Generation by Symbolic Execution

Lazy Initialization vs Heap Solver

\[ m(C \ x, C \ y, C \ z) \]
\[
\begin{align*}
    x.f &= 1; \\
    z.f &= -5; \\
    y.f &= x.f + 1; \\
    m2(); \\
    \text{if} \ (x == z) \\
    &\quad m3(y.f); \\
    \text{else} \\
    &\quad m4(y.f);
\end{align*}
\]

- Symbolic execution quickly becomes impractical
- Redundant exploration of large number of paths
Test Case Generation by Symbolic Execution

Lazy Initialization vs Heap Solver

m(C x, C y, C z)
{
    x.f = 1;
    z.f = -5;
    y.f = x.f+1;
    m2();
    if (x==z)
        m3(y.f);
    else
        m4(y.f);
}

▶ Symbolic execution quickly becomes impractical
▶ Redundant exploration of large number of paths
Symbolic execution quickly becomes impractical
- Redundant exploration of large number of paths
m(C x, C y, C z) {
    x.f = 1;
    z.f = -5;
    y.f = x.f + 1;
    m2();
    if (x == z)
        m3(y.f);
    else
        m4(y.f);
}

> Symbolic execution quickly becomes impractical
> Redundant exploration of large number of paths
Test Case Generation by Symbolic Execution

Lazy Initialization vs Heap Solver

\[ m(C \ x, C \ y, C \ z) \]
\[
\begin{align*}
\text{x.f} &= 1; \\
\text{z.f} &= -5; \\
\text{y.f} &= \text{x.f}+1; \\
\text{m2}(); \\
\text{if} \ (\text{x}==\text{z}) \\
\phantom{\text{if} \ (\text{x}==\text{z})} & \text{m3(y.f);} \\
\text{else} \\
\phantom{\text{else}} & \text{m4(y.f);} \\
\end{align*}
\]

- Symbolic execution quickly becomes impractical
- Redundant exploration of large number of paths
Test Case Generation by Symbolic Execution

Lazy Initialization vs Heap Solver

\[
m(C \ x, C \ y, C \ z)
\]
\[
\{ \\
  \text{x.f = 1;} \\
  \text{z.f = -5;} \\
  \text{y.f = x.f + 1;} \\
  \text{m2();} \\
  \text{if (x == z)} \\
  \quad \text{m3(y.f);} \\
  \text{else} \\
  \quad \text{m4(y.f);} \\
\}
\]

- Symbolic execution quickly becomes impractical
- Redundant exploration of large number of paths
m(C x, C y, C z)
{
    x.f = 1;
    z.f = -5;
    y.f = x.f + 1;
    m2();
    if (x == z)
        m3(y.f);
    else
        m4(y.f);
}"
m(C x, C y, C z)
{
    x.f = 1;
    z.f = -5;
    y.f = x.f + 1;
    m2();
    if (x==z)
        m3(y.f);
    else
        m4(y.f);
}

- A more scalable approach than lazy initialization
m(C x, C y, C z)
{
  x.f = 1;
  z.f = -5;
  y.f = x.f + 1;
  m2();
  if (x==z)
    m3(y.f);
  else
    m4(y.f);
}

- Avoid non-determinism as much as possible
Test Case Generation by Symbolic Execution

Lazy Initialization vs Heap Solver

\[
m(C \; x, C \; y, C \; z) \\
\{ \\
\quad x.f = 1; \\
\quad z.f = -5; \\
\quad y.f = x.f+1; \\
\quad m2(); \\
\quad \text{if} \ (x==z) \\
\quad \quad m3(y.f); \\
\quad \text{else} \\
\quad \quad m4(y.f); \\
\}
\]

▶ Treatment of reference aliasing by means of disjunctions
\[ m(C \ x, C \ y, C \ z) \]
\[
\begin{align*}
  x.f &= 1; \\
  z.f &= -5; \\
  y.f &= x.f + 1; \\
  &m2(); \\
  \text{if } (x==z) \\
  &m3(y.f); \\
  \text{else} \\
  &m4(y.f); 
\end{align*}
\]

- Treatment of reference aliasing by means of disjunctions
\[ m(C \ x, C \ y, C \ z) \]
\[
\begin{align*}
x.f & = 1; \\
z.f & = -5; \\
y.f & = x.f + 1; \\
m2(); \\
\text{if (}x==z\text{)} & \\
\quad m3(y.f); \\
\text{else} & \\
\quad m4(y.f); \\
\end{align*}
\]

- Propagation of heap-related constraints
m(C x, C y, C z)
{
    x.f = 1;
    z.f = -5;
    y.f = x.f+1;
    m2();
    if (x==z)
        m3(y.f);
    else
        m4(y.f);
}

- Support for heap assumptions to avoid certain aliasing configurations. E.g., acyclic(x), non-aliasing(x,z)
Test Case Generation by Symbolic Execution

Lazy Initialization vs Heap Solver

\[ m(C \ x, C \ y, C \ z) \]
\[
\begin{align*}
  &x.f = 1; \\
  &z.f = -5; \\
  &y.f = x.f + 1; \\
  &m2(); \\
  &\text{if } (x == z) \\
  &\quad m3(y.f); \\
  &\text{else} \\
  &\quad m4(y.f); \\
\end{align*}
\]

- Implemented in PET
- Applicable to other systems
Plan of the Lecture

- **Part 1: Symbolic execution and TCG**
  - Introduction
  - Handling heap-manipulating programs
  - Compositionallity

- **Part 2: CLP-based TCG**
  - Introduction
  - Translation from imperative to CLP
  - Guided-TCG
  - Demo

- **Part 3: TCG of Concurrent (Actor) Programs**
  - The path explotion problem
  - Symbolic execution and TCG for actors
  - Demo
Compositional Test Case Generation

Motivation

- Compositional reasoning to tackle inter-procedural path explosion
- Generation and re-utilization of method summaries
- Handling native code and libraries
Compositional Test Case Generation

Challenge

\[ m(\ldots) \]

\[ \ldots \]

\[ \top \]

\[ q(\ldots) \]

SymEx Tree for \( q \)

\[ \top \]

\[ \top \]

\[ \top \]
Avoid inlining the symbolic execution tree of q
Avoid inlining the symbolic execution tree of q

Use method summary for q: Check compatibility with current state of m
(Only compatible summary cases are composed)
Avoid inlining the symbolic execution tree of $q$

Use method summary for $q$: Check compatibility with current state of $m$ (Only compatible summary cases are composed)

Incremental: summary for method $m$ is created
Avoid inlining the symbolic execution tree of \( q \)

- Use method summary for \( q \): Check compatibility with current state of \( m \) (Only compatible summary cases are composed)

- Incremental: summary for method \( m \) is created

- Compositional TCG must compute the same results as Standard TCG
Compositional Test Case Generation
Composition Strategies

\[ m(\ldots) \] 
\[ \text{true} \] 
\[ \phi \] 
\[ \text{true} \] 
\[ q(\ldots) \] 
\[ \text{SymEx Tree for } q \] 
\[ \text{true} \] 
\[ \text{true} \]

\[ m(\ldots) \] 
\[ \text{true} \] 
\[ \phi \] 
\[ \text{true} \] 
\[ q(\ldots) \] 
\[ \text{Summary for } q \] 
\[ S_q \] 
\[ \text{true} \] 
\[ \text{true} \] 
\[ \text{true} \] 
\[ \text{true} \] 

Context-sensitive
- Top-down traversal of call-graph
  - Pro.: Only required information is computed
  - Con.: Reusability of summaries is not always possible

Context-insensitive
- Bottom-up traversal of call-graph
  - Pro.: Composition can always be performed
  - Con.: Summaries can contain more test cases than necessary (more expensive)
Compositional Test Case Generation
Composition Strategies

**Context-sensitive**

- **Top-down** traversal of call-graph
- **Pro.**: Only required information is computed
- **Con.**: Reusability of summaries is not always possible
Compositional Test Case Generation
Composition Strategies

Context-sensitive
- **Top-down** traversal of call-graph
- **Pro.:** Only required information is computed
- **Con.:** Reusability of summaries is not always possible

Context-insensitive
- **Bottom-up** traversal of call-graph
- **Pro.:** Composition can always be performed
- **Con.:** Summaries can contain more test cases than necessary (more expensive)
A summary is a finite representation of the symbolic execution of a program for a given termination criterion, i.e.,

$$S_C^q \equiv T_C^q$$

Each element in a summary corresponds to a symbolic execution path (test case)

Complete for a given coverage criterion, but partial in general
A summary is a finite representation of the symbolic execution of a program for a given termination criterion, i.e., $S^C_q \equiv \Phi \equiv \text{Path condition} \land \text{Termination criterion } C$.

Each element in a summary corresponds to a symbolic execution path (test case) complete for a given coverage criterion, but partial in general.
Compositional Test Case Generation
Generating Symbolic Execution Summaries

A summary is a finite representation of the symbolic execution of a program for a given termination criterion, i.e., $S_q^C \equiv \mathcal{T}_q^C$

- Each element in a summary corresponds to a symbolic execution path (test case)
- **Complete** for a given coverage criterion, but **partial** in general
Plan of the Lecture

- Part 1: Symbolic execution and TCG
  - Introduction
  - Handling heap-manipulating programs
  - Compositionality
- Part 2: CLP-based TCG
  - Introduction
  - Translation from imperative to CLP
  - Guided-TCG
  - Demo
- Part 3: TCG of Concurrent (Actor) Programs
  - The path explotion problem
  - Symbolic execution and TCG for actors
  - Demo
Translation of the source language to CLP
Translation of the source language to CLP

Java Code
```java
int abs(int x) {
    if (x >= 0) return x;
    else return -x;
}
```

CLP-translated
```
abs(X,X) :- X #>= 0.
abs(X,Z) :- X #< 0, Z #= -X.
```

Bounded symbolic execution of the CLP-translated program
Translation of the source language to CLP

Java Code
```java
int abs(int x) {
    if (x >= 0) return x;
    else return -x; }
```

CLP-translated
```
abs(X,X) :- X #>= 0.
abs(X,Z) :- X #< 0, Z #= -X.
```

Bounded symbolic execution of the CLP-translated program

Symbolic execution comes (almost) for free in CLP
CLP-based Symbolic Execution and TCG

Translation of the source language to CLP

Java Code

```java
int abs(int x) {
    if (x >= 0) return x;
    else return -x;
}
```

CLP-translated

```
abs(X,X) :- X #>= 0.
abs(X,Z) :- X #< 0, Z#= -X.
```

Bounded symbolic execution of the CLP-translated program
Symbolic execution comes (almost) for free in CLP
Backtracking and constraint manipulation/solving
Translation of the source language to CLP

Java Code
```java
int abs(int x) {
    if (x >= 0) return x;
    else return -x;
}
```

CLP-translated
```
abs(X,X) :- X \#>= 0.
abs(X,Z) :- X \#< 0,
          Z \#= -X.
```

Bounded symbolic execution of the CLP-translated program

Symbolic execution comes (almost) for free in CLP

Backtracking and constraint manipulation/solving

Test cases
```
{ ⟨X >= 0,Z = X⟩,
  ⟨X < 0,Z = -X⟩ }
```

Concrete inputs
```
{ ⟨ X = 1, Z = 1⟩,
  ⟨ X = -1, Z = 1⟩ }
```

JUnit code
```java
void test_abs() {
    assertEquals(abs(1),1);
    assertEquals(abs(-1),1);
}
```
Translation of the source language to CLP

Java Code
```java
int abs(int x) {
    if (x >= 0) return x;
    else return -x;
}
```

CLP-translated
```prolog
abs(X,X) :- X #>= 0.
abs(X,Z) :- X #< 0, Z #= -X.
```

Bounded symbolic execution of the CLP-translated program

Symbolic execution comes (almost) for free in CLP

Backtracking and constraint manipulation/solving

<table>
<thead>
<tr>
<th>Test cases</th>
<th>Concrete inputs</th>
<th>JUnit code</th>
</tr>
</thead>
</table>
| `{ ⟨X >= 0, Z = X⟩,  | `{ ⟨X = 1, Z = 1⟩,      | void test_abs(){
| ⟨X < 0, Z = -X⟩ }    | ⟨X = -1, Z = 1⟩ }        |    assertEquals(abs(1),1);
|                     |                          |    assertEquals(abs(-1),1); }                  |

The PET System (costa.ls.fi.upm.es/pet)
Let $M$ be a method, $m$ be its corresponding predicate from its CLP-translated program $P$, and $C$ be a termination criterion.
Let $M$ be a method, $m$ be its corresponding predicate from its CLP-translated program $P$, and $C$ be a termination criterion.

- The symbolic execution of $m$ is the possibly infinite CLP derivation tree of $P$, denoted as $\mathcal{T}_m$, with root $m(In, Out, H_{in}, H_{out}, EF)$ and initial constraint store $\theta = \{\}$. 

- $T_{C}m$ is the finite, possibly incomplete version of $T_m$ bounded by $C$.

- A test case for $m$ wrt $C$ is $\langle \sigma(In), \sigma(Out), \sigma(H_{in}), \sigma(H_{out}), \sigma(EF), \theta' \rangle$ where $\sigma$ and $\theta'$ are, resp., the substitution and the constraint store associated to a successful (terminating) path in $T_{C}m$.

- TCG is the process of generating the set of test cases for all successful (terminating) paths in $T_{C}m$. 
Let $M$ be a method, $m$ be its corresponding predicate from its CLP-translated program $P$, and $C$ be a termination criterion.

- The symbolic execution of $m$ is the possibly infinite CLP derivation tree of $P$, denoted as $\mathcal{T}_m$, with root $m(In, Out, H_{in}, H_{out}, EF)$ and initial constraint store $\theta = \{\}$.  
- $\mathcal{T}_m^C$ is the finite, possibly incomplete version of $\mathcal{T}_m$ bounded by $C$.  

Let $M$ be a method, $m$ be its corresponding predicate from its CLP-translated program $P$, and $C$ be a termination criterion.

- The symbolic execution of $m$ is the possibly infinite CLP derivation tree of $P$, denoted as $T_m$, with root $m(In, Out, H_{in}, H_{out}, EF)$ and initial constraint store $\theta = \{\}$.  
- $T_m^C$ is the finite, possibly incomplete version of $T_m$ bounded by $C$.  
- A test case for $m$ wrt $C$ is $\langle \sigma(In), \sigma(Out), \sigma(H_{in}), \sigma(H_{out}), \sigma(EF), \theta' \rangle$ where $\sigma$ and $\theta'$ are, resp., the substitution and the constraint store associated to a successful (terminating) path in $T_m^C$.  

Elvira Albert  
Test Case Generation by Symbolic Execution  
16-20 June 2014
CLP-based Symbolic Execution and TCG
Symbolic Execution and Test Case Generation

Let $M$ be a method, $m$ be its corresponding predicate from its CLP-translated program $P$, and $C$ be a termination criterion.

- The symbolic execution of $m$ is the possibly infinite CLP derivation tree of $P$, denoted as $\mathcal{T}_m$, with root $m(In, Out, H_{in}, H_{out}, EF)$ and initial constraint store $\theta = \{\}$.  
- $\mathcal{T}^C_m$ is the finite, possibly incomplete version of $\mathcal{T}_m$ bounded by $C$.  
- A test case for $m$ wrt $C$ is $\langle \sigma(In), \sigma(Out), \sigma(H_{in}), \sigma(H_{out}), \sigma(EF), \theta' \rangle$ where $\sigma$ and $\theta'$ are, resp., the substitution and the constraint store associated to a successful (terminating) path in $\mathcal{T}^C_m$.  
- TCG is the process of generating the set of test cases for all successful (terminating) paths in $\mathcal{T}^C_m$.  

Elvira Albert  Test Case Generation by Symbolic Execution  16-20 June 2014  18 / 62
Java source code

```java
int exp(int a, int n) {
    if (n < 0)
        throw new Exception();
    else {
        int r = 1;
        while (n > 0) {
            r = r*a;
            n--;
        }
        return r;
    }
}
```

CLP-translated program

```prolog
exp([A,N],Out,Hi,Ho,EF) :-
    if ([A,N],Out,Hi,Ho,EF).
    if ([A,N],_Out,Hi,Ho,exc(X)) :-
        N < 0,
        new_object(Hi,'Exc',X,Ho).
    if ([A,N],Out,H,H,ok) :-
        N >= 0,
        loop([A,N,1],Out).
loop([_A,N,R],R) :-
    N <= 0.
loop([A,N,R],Out) :-
    N > 0,
    R' = R*A,
    N' = N-1,
    loop(A,N',R',Out).
```
CLP-based Symbolic Execution and TCG

Concrete example

Java source code

```java
int exp(int a, int n) {
    if (n < 0)
        throw new Exception();
    else {
        int r = 1;
        while (n > 0) {
            r = r*a;
            n--;
        }
        return r;
    }
}
```

CLP-translated program

```prolog
exp([A,N],Out,H₁,H₀,EF) :-
    if([A,N],Out,H₁,H₀,EF).
if([A,N],_Out,H₁,H₀,exc(X)):-
    N < 0,
    new_object(H₁,'Exc',X,H₀).
if([A,N],Out,H₁,H₀,ok) :-
    N >= 0,
    loop([A,N,1],Out).
loop([_A,N,R],R) :-
    N <= 0.
loop([A,N,R],Out) :-
    N > 0,
    R' = R*A,
    N' = N-1,
    loop(A,N',R',Out).
```
CLP-based Symbolic Execution and TCG
Concrete example

Symb. Ex. Tree

```
exp([A,N],Out,Hi,Ho,EF)
   ↓
if([A,N],Out,Hi,Ho,EF)
   {N<0}↓
exc
   {N=0,R=1}↓
ok
   {N'>=0,N'=N-1,R'=R*A}↓
loop(...)
ok
   {N'<=0,N''=N'-1,R'=R*A}↓
loop(...)
```

CLP-translated program

```
exp([A,N],Out,Hi,Ho,EF) :-
    if([A,N],Out,Hi,Ho,EF).
if([A,N],_Out,Hi,Ho,exc(X)) :-
    N < 0,
    new_object(Hi,'Exc',X,Ho).
if([A,N],Out,H,H,ok) :-
    N >= 0,
    loop([A,N,1],Out).
loop([_A,N,R],R) :-
    N <= 0.
loop([A,N,R],Out) :-
    N > 0,
    R' = R*A,
    N' = N-1,
    loop(A,N',R',Out).
```
CLP-based Symbolic Execution and TCG
Concrete example

Test cases

<table>
<thead>
<tr>
<th>#</th>
<th>Input</th>
<th>Output</th>
<th>Path condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>[A, N]</td>
<td>exception</td>
<td>{N&lt;0}</td>
</tr>
<tr>
<td>2</td>
<td>[A, N]</td>
<td>1</td>
<td>{N=0}</td>
</tr>
<tr>
<td>3</td>
<td>[A, N]</td>
<td>R</td>
<td>{N&gt;0, N'=N-1, R=1*A, N'&lt;=0}</td>
</tr>
</tbody>
</table>

CLP-translated program

exp([A, N], Out, Hi, Ho, EF) :-

if ([A, N], Out, Hi, Ho, EF).

if ([A, N], _Out, Hi, Ho, exc(X)) :- N < 0, new_object(Hi,'Exc',X,Ho).

if ([A, N], Out, Hi, Ho, ok) :- N >= 0, loop([A, N, 1], Out).

loop([_A, N, R], R) :- N <= 0.

loop([A, N, R], Out) :- N > 0, R' = R*A, N' = N-1, N' <= 0, loop(A, N', R', Out).
CLP-based Symbolic Execution and TCG
Concrete example

**Test cases**

<table>
<thead>
<tr>
<th>#</th>
<th>Input</th>
<th>Output</th>
<th>Path condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>[A, N]</td>
<td>exception</td>
<td>{N&lt;0}</td>
</tr>
<tr>
<td>2</td>
<td>[A, N]</td>
<td>1</td>
<td>{N=0}</td>
</tr>
<tr>
<td>3</td>
<td>[A, N]</td>
<td>R</td>
<td>{N&gt;0,N'=N-1,R=1*A,N'&lt;=0}</td>
</tr>
</tbody>
</table>

**Concrete inputs**

<table>
<thead>
<tr>
<th>#</th>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>[-10, -10]</td>
<td>Exception</td>
</tr>
<tr>
<td>2</td>
<td>[-10, 0]</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>[-10, 1]</td>
<td>-10</td>
</tr>
</tbody>
</table>
CLP-based Symbolic Execution and TCG

Concrete example

Test cases

<table>
<thead>
<tr>
<th>#</th>
<th>Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>[A, N] [exception]</td>
</tr>
<tr>
<td>2</td>
<td>[A, N]</td>
</tr>
<tr>
<td>3</td>
<td>[A, N]</td>
</tr>
</tbody>
</table>

Concrete inputs

<table>
<thead>
<tr>
<th>#</th>
<th>Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>[-10, -10]</td>
</tr>
<tr>
<td>2</td>
<td>[-10, 0]</td>
</tr>
<tr>
<td>3</td>
<td>[-10, 1]</td>
</tr>
</tbody>
</table>

Unit tests (JUnit)

```java
public void test_1(){
    int input0 = -10, input1 = -10;
    try{
        int output = Test.intExp(input0,input1);
    } catch(Exception ex){
        assuresEquals("exception","ArithmeticException", ex.getClass().getName());
        return;
    } fail("Fail");
}

public void test_2(){
    int input0 = -10, input1 = 0;
    int output = Test.intExp(input0,input1);
    int expected = 1;
    assuresEquals("OK",expected,output);
}

public void test_3(){
    int input0 = -10, input1 = 1;
    int output = Test.intExp(input0,input1);
    int expected = -10;
    assuresEquals("OK",expected,output);
}
```
Plan of the Lecture

- **Part 1: Symbolic execution and TCG**
  - Introduction
  - Handling heap-manipulating programs
  - Compositionallity

- **Part 2: CLP-based TCG**
  - Introduction
  - Translation from imperative to CLP
  - Guided-TCG
  - Demo

- **Part 3: TCG of Concurrent (Actor) Programs**
  - The path exploition problem
  - Symbolic execution and TCG for actors
  - Demo
**Motivation and Selective Coverage Criteria**

TCG = Symbolic exec. +

- Termination criteria: depth-k
- Selection criteria: specific program point(s) (specific exception(s)), all local paths, worst memory consumption (within a loop-k limit), ...
Guided Test Case Generation
Motivation and Selective Coverage Criteria

TCG = Symbolic exec. + termination criterion + constraint solving

Termination criteria: depth-k,
TCG = Symbolic exec. + termination criterion + constraint solving

- Termination criteria: depth-k, loop-k
TCG = Symbolic exec. + termination criterion + constraint solving

- Termination criteria: depth-k, loop-k
- Selection criteria: specific program point(s) (specific exception(s)), all local paths, worst memory consumption (within a loop-k limit), ...
Selective TCG (naive) = TCG + filtering of test cases
Guided Test Case Generation
Naive Approach to Selective TCG

Selective TCG (naive) = TCG + filtering of test cases

Paths in the symbolic execution tree can be labeled
Selective TCG (naive) = TCG + filtering of test cases

- Paths in the symbolic execution tree can be labeled
- Filtering is done by looking at the traces associated to the test cases
Guided Test Case Generation

Intuition

- Challenge: Avoid the generation of non-interesting paths
- Idea: Use the trace argument as an input to guide symbolic exec.
Guided Test Case Generation

Intuition

- Challenge: Avoid the generation of non-interesting paths
- Idea: Use the trace argument as an input to guide symbolic exec.

Guided TCG = Traces generator + guided symb. execs. + constr. solving

- Traces can be complete
- The different symbolic executions are independent of each other
  • Can be performed in parallel and simplifies constraint solving
Guided Test Case Generation

Intuition

- Challenge: Avoid the generation of non-interesting paths
- Idea: Use the trace argument as an input to guide symbolic exec.

Guided TCG = Traces generator + guided symb. execs. + constr. solving
Guided Test Case Generation

Intuition

- **Challenge**: Avoid the generation of non-interesting paths
- **Idea**: Use the trace argument as an input to guide symbolic exec.

Guided TCG = Traces generator + guided symb. execs. + constr. solving

- Traces can be complete
Guided Test Case Generation

Intuition

- **Challenge:** Avoid the generation of non-interesting paths
- **Idea:** Use the trace argument as an input to guide symbolic exec.

Guided TCG = Traces generator + guided symb. execs. + constr. solving

- Traces can be complete or partial
Guided Test Case Generation

Intuition

- Challenge: Avoid the generation of non-interesting paths
- Idea: Use the trace argument as an input to guide symbolic exec.

Guided TCG = Traces generator + guided symb. execs. + constr. solving

- Traces can be complete or partial
- The different symbolic executions are independent of each other
  - Can be performed in parallel and simplifies constraint solving
Input: \( M, \langle TC, SC \rangle \) and TraceGen
\[
\text{TestCases} = \{\}
\]
\textbf{while} TraceGen has more traces and TestCases doesn't satisfy SC
   
   Ask TraceGen to generate a new trace in Trace
   
   TestCases = TestCases \cup \{ \text{first of guidedSymbExec}(M, TC, Trace) \}
\textbf{Output:} TestCases
Symbolic execution consists in executing the program with symbolic (constraint) variables.

Test cases are extracted from successful branches of the symbolic execution tree.

The main challenges are related to scalability:
- heap-manipulating programs [ICLP’10, ICLP’13]
- compositionallity [LOPSTR’09]

CLP-based instance:
- Symbolic execution almost for free [LOPSTR’08]
- Language-independent approach (same TCG engine)
- Guided TCG [LOPSTR’11, LOPSTR’12]

PET: implementation of this approach [PEPM’10]
Plan of the Lecture

- **Part 1: Symbolic execution and TCG**
  - Introduction
  - Handling heap-manipulating programs
  - Compositionallity

- **Part 2: CLP-based TCG**
  - Introduction
  - Translation from imperative to CLP
  - Guided-TCG
  - Demo

- **Part 3: TCG of Concurrent (Actor) Programs**
  - The path explotion problem
  - Symbolic execution and TCG for actors
  - Demo
Concurrency in programming is gaining importance

Additional hazards in concurrent programs: data races, deadlocks, etc.

Software validation techniques urge especially in this context

Path explosion problem - non-deterministic interleavings of processes
  - An exhaustive exploration is often computationally intractable
  - Challenge: Avoid redundant state exploration
  - Partial Order Reduction techniques (POR)

Thread-based concurrency tends to be error-prone, very difficult to debug and analyze and not scalable

Alternative ⇒ the Actors-based concurrency model (e.g. Erlang, Scala, ABS, Java libraries for actors, ...)

Actors concurrency model in OO style (Concurrent Objects):
  1. Actor/Object ⇔ concurrency unit
  2. No shared memory ⇒ Information exchange by means of messages/asynchronous-method-calls
  3. Task scheduling is cooperative
Introduction

- Concurrency in programming is gaining importance
- Additional hazards in concurrent programs: data races, deadlocks, etc.
- Software validation techniques urge especially in this context
- Path explosion problem - non-deterministic interleavings of processes
  - An exhaustive exploration is often computationally intractable
  - Challenge: Avoid redundant state exploration
  - Partial Order Reduction techniques (POR)
- Thread-based concurrency tends to be error-prone, very difficult to debug and analyze and not scalable
- Alternative ⇒ the Actors-based concurrency model (e.g. Erlang, Scala, ABS, Java libraries for actors, ...)
- Actors concurrency model in OO style (Concurrent Objects):
  1. Actor/Object ⇔ concurrency unit
  2. No shared memory ⇒ Information exchange by means of messages/asynchronous-method-calls
  3. Task scheduling is cooperative
The Actor Model

### Syntax of the Language

\[
M ::= \text{void } m(\bar{T} \bar{x})\{s;\}
\]

\[
s ::= s ; \quad s | \quad x = e \quad | \quad x = \text{this} . f \quad | \quad \text{this} . f = y \quad | \quad \text{if } b \text{ then } s \text{ else } s \quad | \quad \text{while } b \text{ do } s \quad | \quad x = \text{new } C \quad | \quad x ! m(\bar{z}) \quad | \quad \text{return}
\]

- A **program** is a set of classes. A class contains a set of **fields** \( f \) and **methods** \( M \).
- Actors are created dynamically using the instruction **new**.
- Each actor has its own local state and thread control and communicate by exchanging messages asynchronously.
- An actor sends a message to another actor \( x \) by means of an **asynchronous method call** \( x ! m(\bar{z}) \).
- An actor configuration consists: **local state** and **pending tasks**.
Partial Order Reduction

- At each execution step, firstly an actor and secondly a process of its pending tasks are scheduled.
- There are two levels of non-determinism:
  - **Actor-selection**: The selection of which actor executes;
  - **Task-selection**: The selection of the task within the selected actor.
Partial Order Reduction

▶ At each execution step, firstly an actor and secondly a process of its pending tasks are scheduled.
▶ There are two levels of non-determinism:
  - **Actor-selection**: The selection of which actor executes;
  - **Task-selection**: The selection of the task within the selected actor.

State Explosion Problem

▶ As actors do not share their states, in testing we assume that evaluation of all statements of a task is serial until processor released
▶ A naïve exploration of the search space to reach all possible system configurations does not scale.
▶ **Partial-order reduction** (POR) helps mitigate this problem by exploring the subset of all possible interleavings which lead to a different configuration.
Partial Order Reduction

\[ h_{o_1} = \{ this.f = 2 \} \quad t_1 \mapsto this.f = 5 \quad t_2 \mapsto this.f = 7 \]

\[ h_{o_2} = \{ this.g = 1 \} \quad t_3 \mapsto this.g = 9 \]

\[ \{ t_1, t_2, t_3 \} \]
Partial Order Reduction

\[ h_{o_1} = \{ \text{this}.f = 2 \} \quad t_1 \mapsto \text{this}.f = 5 \quad t_2 \mapsto \text{this}.f = 7 \]

\[ h_{o_2} = \{ \text{this}.g = 1 \} \quad t_3 \mapsto \text{this}.g = 9 \]

\[ \{ t_1, t_2, t_3 \} \]

This.

<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>g</td>
</tr>
<tr>
<td>f</td>
</tr>
</tbody>
</table>

\[ h_{o_1} = \{ \text{this}.g = 1 \} \quad t_1 \mapsto \text{this}.g = 1 \quad t_2 \mapsto \text{this}.g = 9 \]

\[ h_{o_2} = \{ \text{this}.f = 2 \} \quad t_3 \mapsto \text{this}.f = 5 \]

\[ \{ t_1, t_2, t_3 \} \]

This.

<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>g</td>
</tr>
<tr>
<td>f</td>
</tr>
</tbody>
</table>

\[ h_{o_1} = \{ \text{this}.g = 1 \} \quad t_1 \mapsto \text{this}.g = 1 \quad t_2 \mapsto \text{this}.g = 9 \]

\[ h_{o_2} = \{ \text{this}.f = 2 \} \quad t_3 \mapsto \text{this}.f = 5 \]

\[ \{ t_1, t_2, t_3 \} \]
Partial Order Reduction

\[ h_{o_1} = \{ \text{this.f = 2} \} \quad t_1 \mapsto \text{this.f = 5} \quad t_2 \mapsto \text{this.f = 7} \]
\[ h_{o_2} = \{ \text{this.g = 1} \} \quad t_3 \mapsto \text{this.g = 9} \]

\[ o_1 \quad o_2 \]
\[ \{ t_1, t_2, t_3 \} \]
Partial Order Reduction

\[ h_{o_1} = \{ this.f = 2 \} \quad t_1 \mapsto this.f = 5 \quad t_2 \mapsto this.f = 7 \]

\[ h_{o_2} = \{ this.g = 1 \} \quad t_3 \mapsto this.g = 9 \]
Partial Order Reduction

\( h_{o_1} = \{ this.f = 2 \} \quad t_1 \mapsto this.f = 5 \quad t_2 \mapsto this.f = 7 \)
\( h_{o_2} = \{ this.g = 1 \} \quad t_3 \mapsto this.g = 9 \)

\[ t_3 \mapsto this.g = 9 \]
\[ t_2 \mapsto this.f = 7 \]
\[ s_1 \]
\[ s_1 \]
Partial Order Reduction

$h_{o_1} = \{ this.f = 2 \} \quad t_1 \mapsto this.f = 5 \quad t_2 \mapsto this.f = 7$

$h_{o_2} = \{ this.g = 1 \} \quad t_3 \mapsto this.g = 9$
Partial Order Reduction

\[ h_{o_1} = \{ \text{this.f} = 2 \} \quad t_1 \leftrightarrow \text{this.f} = 5 \quad t_2 \leftrightarrow \text{this.f} = 7 \]

\[ h_{o_2} = \{ \text{this.g} = 1 \} \quad t_3 \leftrightarrow \text{this.g} = 9 \]
Partial Order Reduction

\[ h_{o_1} = \{ \text{this.f} = 2 \} \quad t_1 \mapsto \text{this.f} = 5 \quad t_2 \mapsto \text{this.f} = 7 \]

\[ h_{o_2} = \{ \text{this.g} = 1 \} \quad t_3 \mapsto \text{this.g} = 9 \]

\[ \{ t_1, t_2, t_3 \} \]

\[ s_1 \]

\[ s_2 \]

\[ \text{this.g} = 9 \]
\[ \text{this.f} = 7 \]

\[ \text{order in } o_1 : t_1 < t_2 \quad t_2 < t_1 \]

\[ o_1, o_2 \text{ are temporarily stable} \]
Partial Order Reduction

\[ h_{o_1} = \{ this.f = 2 \} \quad t_1 \mapsto this.f = 5 \quad t_2 \mapsto this.f = 7 \]

\[ h_{o_2} = \{ this.g = 1 \} \quad t_3 \mapsto this.g = 9 \]

\[ \{ t_1, t_2, t_3 \} \]

\[ this.g = 9 \]
\[ this.f = 7 \]  

\[ this.g = 9 \]
\[ this.f = 5 \]  

\[ o_1, o_2 \text{ are temporarily stable} \]

order in \( o_1 : t_1 < t_2 \quad t_2 < t_1 \)
How to avoid redundant exploration in testing?

TransDPOR [Tasharofi et al FMOODS/FORTE 2012]

- Intuition: for each configuration, use a **backtrack set**, which is updated during the execution of the program when it realises that a non-deterministic choice must be tried

- **Select Object** and **Select Task** (non-deterministically) from a node $n$: $o.t$

- Execute $o.t$ in node $n$;

- If $o$ has been previously selected, look for the first node $n'$ from the root, selecting object $o$.
  - If $t$ was in $n'$, then mark **backtracking** on $n'$ with $o.t$;
  - Otherwise, look from $n$ upwards, the object $o'$ which introduced $t$ by executing $o'.t'$. If $o'.t'$ is in $n'$, add **backtracking** on $o'.t'$ in node $n'$. Otherwise repeat the process with $o'.t'$ upwards.
How to avoid redundant exploration in testing?

\[ o.t \in Q \quad o.t_1 \in Q \quad o.t \not\in Q \quad o.t' \in Q' \]

Back = \{ o.t' \}

Elvira Albert
Test Case Generation by Symbolic Execution
16-20 June 2014 39 / 62
How to avoid redundant exploration in testing?

$t \in Q_o$

$o.t_1$

$o.t$

Back = o.

$t \in Q_o$

$o.t \in Q_o$

Back = \{ o.t \}$

Elvira Albert

Test Case Generation by Symbolic Execution

16-20 June 2014

39 / 62
How to avoid redundant exploration in testing?

\[ t \in Q \]

\[ o.t_1 \]

\[ o.t \]

\[ t \in Q_o \]

\[ Back = \{ o.t \} \]
How to avoid redundant exploration in testing?

$t \in Q_o$

$Back = \{ o.t \}$
How to avoid redundant exploration in testing?

$t \in Q_o$

Back = \{o.t\}

Elvira Albert
Test Case Generation by Symbolic Execution
16-20 June 2014
39 / 62
How to avoid redundant exploration in testing?

$t \in Q_o$

$Back = \{ o.t \}$

$t \notin Q_o$

$Back = \{ o.t' \}$
How to avoid redundant exploration in testing?

$t \in Q_o$

Back = \{o.t\}

$t \notin Q_o$

$t' \in Q'_o$

$t \notin Q_o$

$t' \in Q'_o$

Back = \{o'.t'\}

$t \in Q_o$

$t \notin Q_o$
How to avoid redundant exploration in testing?

$t \in Q_o$

$Back = \{ o.t \}$

$t \notin Q_o$

$Back = \{ o'.t' \}$

$t' \in Q'_o$
How to avoid redundant exploration in testing?

\[ t \in Q_o \]

\[ \text{Back} = \{ o.t \} \]

\[ t \notin Q_o \]

\[ o.t \]

\[ o.t_1 \]

\[ o.t \]

\[ o.t \]

\[ t \in Q_o \]

\[ t' \in Q'_o \]

\[ o.t_1 \]

\[ o'.t' \]

\[ o'.t' \]

\[ \text{Back} = \{ o'.t' \} \]
How to avoid redundant exploration in testing?

$\exists t, t \in Q_o$

$\exists t_1, o.t_1, o.t < t_1$

$\exists t_1, t < t_1$

$\exists Back = \{ o.t \}$

$\exists t_1, t \notin Q_o$

$\exists t', o.t_1, o'.t' < t_1$

$\exists t', t < t_1$

$\exists Back = \{ o'.t' \}$
Working Example. Actors in Action

```java
{  
    Reg rg = new Reg();
    Worker1 wk1 = new Worker1();
    Worker2 wk2 = new Worker2();
    rg!r0();
    wk1!w1(rg);
    wk2!w2(rg);
}

class Reg {
    int f = 1; int g = 1;
    void r0() { this.f++; return; }
    void r1() { this.g* = 2; return; }
    void r2() { this.g++; return; }
}

class Worker1 {
    void w1(Reg rg) {
        rg!r1(); return;
    }
}

class Worker2 {
    void w2(Reg rg) {
        rg!r2(); return;
    }
}
```
```java
{  
    Reg rg = new Reg();
    Worker\_1 wk1 = new Worker\_1();
    Worker\_2 wk2 = new Worker\_2();
    rg!r0();
    wk1!w1(rg);
    wk2!w2(rg);
}

class Reg {
    int f=1; int g=1;
    void r0() { this.f++; return; }
    void r1() { this.g*=2; return; }
    void r2() { this.g++;
    return; }
}

class Worker\_1 {
    void w1(Reg rg) {
        rg!r1(); return;
    }
}

class Worker\_2 {
    void w2(Reg rg) {
        rg!r2(); return;
    }
}
```
Working Example. Actors in Action

```java
{  
    Reg rg = new Reg();  
    Worker1 wk1 = new Worker1();  
    Worker2 wk2 = new Worker2();  
    rg!r0();  
    wk1!w1(rg);  
    wk2!w2(rg);  
}

class Reg {
    int f=1; int g=1;
    void r0() { this.f++; return; }
    void r1() { this.g*=2; return; }
    void r2() { this.g++;return; }
}

class Worker1 {
    void w1(Reg rg) {
        rg!r1(); return; }
}

class Worker2 {
    void w2(Reg rg) {
        rg!r2(); return; }
}  
```
```java
{  
    Reg rg = new Reg();
    Worker₁ wk1 = new Worker₁();
    Worker₂ wk2 = new Worker₂();
    rg!r0();
    wk1!w1(rg);
    wk2!w2(rg);
}

class Reg {
    int f=1; int g=1;
    void r0() { this.f++; return; }
    void r1() { this.g*=2; return; }
    void r2() { this.g++; return; }
}

class Worker₁ {
    void w1(Reg rg) {
        rg!r1(); return;
    }
}

class Worker₂ {
    void w2(Reg rg) {
        rg!r2(); return;
    }
}
```
Working Example. Actors in Action

```java
{  
    Reg rg = new Reg();
    Worker1 wk1 = new Worker1();
    Worker2 wk2 = new Worker2();
    rg!r1();
    wk1!w1(rg);
    wk2!w2(rg);
}

class Reg {
    int f=1; int g=1;
    void r0() { this.f++; return; }
    void r1() { this.g*=2; return; }
    void r2() { this.g++; return; }
}

class Worker1 {
    void w1(Reg rg) {
        rg!r1(); return;
    }
}

class Worker2 {
    void w2(Reg rg) {
        rg!r2(); return;
    }
}
```
Working Example. Actors in Action

```java
{  
    Reg rg = new Reg();
    Worker₁ wk1 = new Worker₁();
    Worker₂ wk2 = new Worker₂();
    rg!r0();
    wk1!w1(rg);
    wk2!w2(rg);
}

class Reg {
    int f=1; int g=1;
    void r0() { this.f++; return; }
    void r1() { this.g*=2; return; }
    void r2() { this.g++;return; }
}

class Worker₁ {
    void w1(Reg rg) {
        rg!r1(); return;
    }
}

class Worker₂ {
    void w2(Reg rg) {
        rg!r2(); return;
    }
}
```
Working Example. Actors in Action

```java
{  
    Reg rg = new Reg();
    Worker1 wk1 = new Worker1();
    Worker2 wk2 = new Worker2();
    rg!r0();
    wk1!w1(rg);
    wk2!w2(rg);
}

class Reg {
    int f=1; int g=1;
    void r0() { this.f++; return; }
    void r1() { this.g*=2; return; }
    void r2() { this.g++;return; }
}

class Worker1 {
    void w1(Reg rg) {
        rg!r1(); return;
    }
}

class Worker2 {
    void w2(Reg rg) {
        rg!r2(); return;
    }
}
```
Working Example. Actors in Action

```java
{  
    Reg rg = new Reg();  
    Worker1 wk1 = new Worker1();  
    Worker2 wk2 = new Worker2();  
    rg!r0();  
    wk1!w1(rg);  
    wk2!w2(rg);  
}

class Reg {
    int f=1; int g=1;
    void r0() { this.f++; return; }
    void r1() { this.g*=2; return; }
    void r2() { this.g++;return; }
}

class Worker1 {
    void w1(Reg rg) {  
        rg!r1(); return;  
    }
}

class Worker2 {
    void w2(Reg rg) {  
        rg!r2(); return;  
    }
}
```
Working Example. Actors in Action

```java
{  
    Reg rg = new Reg();  
    Worker1 wk1 = new Worker1();  
    Worker2 wk2 = new Worker2();  
    rg!r0();  
    wk1!w1(rg);  
    wk2!w2(rg);  
}

class Reg {
    int f=1; int g=1;
    void r0() { this.f++; return; }
    void r1() { this.g*=2; return; }
    void r2() { this.g++;return; }
}

class Worker1 {
    void w1(Reg rg) {  
        rg!r1(); return;  
    }
}

class Worker2 {
    void w2(Reg rg) {  
        rg!r2(); return;  
    }
}
```

```
rg

rg

wk1

wk2

Main

Registry

Worker1

Worker2

r0 Register

r1 Register

r2 Register

w1 Register yourself at Registry

w2 Register yourself at Registry

h

g: 1

f: 2

Q

rg

Q

r2

Q

w1

wk1

Q

w2

wk2
Working Example. Actors in Action

```java
{  
    Reg rg = new Reg();
    Worker1 wk1 = new Worker1();
    Worker2 wk2 = new Worker2();
    rg!r0();
    wk1!w1(rg);
    wk2!w2(rg);
}

class Reg {
    int f=1; int g=1;
    void r0() { this.f++; return; }
    void r1() { this.g*=2; return; }
    void r2() { this.g++;return; }
}

class Worker1 {
    void w1(Reg rg) {
        rg!r1(); return;
    }
}

class Worker2 {
    void w2(Reg rg) {
        rg!r2(); return;
    }
}
```
// main Block
Reg reg = new Reg;
Worker1 wk1 = new Worker1();
Worker2 wk2 = new Worker2();
reg!r0();
wk1!w1(reg); wk2!w2(reg);

class Worker1 {
    void w1(Reg rg) { rg!r1(); return; }
}
class Worker2 {
    void w2(Reg rg) { rg!r2(); return; }
}

reg:{r0}, wk1:{w1}, wk2:{w2}
class Reg {
    int f=1; int g=1;
    void r0() {this.f++; return;}
    void r1() {this.g*=2; return;}
    void r2() {this.g++; return;}
}

class Worker1 {
    void w1(Reg rg) {rg.r1(); return;}
}

class Worker2 {
    void w2(Reg rg) {rg.r2(); return;}
}

Macro-step semantics ⇒ Interleavings only at return points
class Reg {
    int f=1; int g=1;
    void r0() {this.f++; return;}
    void r1() {this.g*=2; return;}
    void r2() {this.g++; return;}
}

class Worker1 {
    void w1(Reg rg) {rg.r1(); return;}
}

class Worker2 {
    void w2(Reg rg) {rg.r2(); return;}
}

▶ Macro-step semantics ⇒ Interleavings only at return points
```java
class Reg {
    int f = 1; int g = 1;
    void r0() {this.f++; return;}
    void r1() {this.g *= 2; return;}
    void r2() {this.g++; return;}
}
```

```java
class Worker1 {
    void w1(Reg rg) {rg.r1(); return;}
}
class Worker2 {
    void w2(Reg rg) {rg.r2(); return;}
}
```

Macro-step semantics ⇒ Interleavings only at return points
```java
class Reg {
    int f=1; int g=1;
    void r0() {this.f++; return;}
    void r1() {this.g*=2; return;}
    void r2() {this.g++; return;}
}

class Worker1 {
    void w1(Reg rg) {rg.r1(); return;}
}

class Worker2 {
    void w2(Reg rg) {rg.r2(); return;}
}
```

**Macro-step semantics** ⇒ Interleavings only at return points
class Reg {
    int f=1; int g=1;
    void r0() {this.f++; return;}
    void r1() {this.g*=2; return;}
    void r2() {this.g++; return;}
}

class Worker1 {
    void w1(Reg rg) {rg.r1(); return;}
}

class Worker2 {
    void w2(Reg rg) {rg.r2(); return;}
}

**Partial Order Reduction**: Executions with the same partial order are redundant
**Partial Order Reduction**: Executions with the same partial order are redundant.


- **Partial Order Reduction**: Executions with the same partial order are redundant
- 32 paths are explored. 26 of them redundant!
class Reg {
    int f=1; int g=1;
    void r0() {this.f++; return;}
    void r1() {this.g*=2; return;}
    void r2() {this.g++; return;}
}

class Worker1 {
    void w1(Reg rg) {rg.r1(); return;}
}

class Worker2 {
    void w2(Reg rg) {rg.r2(); return;}
}

reg:{r0}, wk1:{w1}, wk2:{w2}[]
To explore \( r1 \) before \( r0 \) actor \( wk1 \) must be selected in the root.
class Reg {
    int f=1; int g=1;
    void r0() {this.f++; return;}
    void r1() {this.g*=2; return;}
    void r2() {this.g++; return;}
}

class Worker1 {
    void w1(Reg rg) {rg!r1(); return;}
}

class Worker2 {
    void w2(Reg rg) {rg!r2(); return;}
}

reg:{r0}, wk1:{w1}, wk2:{w2}[wk1]

reg:{r1}, wk2:{w2}[ ]

wk1:{w1}, wk2:{w2}[ ]

wk1:{w1}, wk2:{w2}[ ]

reg:{r1}, wk2:{w2}[ ]

reg:{r0}, wk1:{w1}, wk2:{w2}[wk1]

wk1:{w1}, wk2:{w2}[ ]

reg:{r1}, wk2:{w2}[ ]

Actor wk1 is added to the backtrack set of the root
class Reg {
    int f=1; int g=1;
    void r0() {this.f++; return;}
    void r1() {this.g*=2; return;}
    void r2() {this.g++; return;}
}

class Worker1 {
    void w1(Reg rg) {rg.r1(); return;}
}

class Worker2 {
    void w2(Reg rg) {rg.r2(); return;}
}

reg:{r0}, wk1:{w1}, wk2:{w2}[wk1]

reg:{r1}, wk2:{w2}[ ]

wk1:{w1}, wk2:{w2}[ ]

wk1.w1

reg:{r1}, wk2:{w2}[ ]

wk2.w2↓

reg:{r1, r2}[r2]

r1↓

reg:{r1}

r0<r1<r2
class Reg {
    int f=1; int g=1;
    void r0() {this.f++; return;}
    void r1() {this.g*=2; return;}
    void r2() {this.g++; return;}
}

class Worker1 {
    void w1(Reg rg) {rg.r1(); return;}
}

class Worker2 {
    void w2(Reg rg) {rg.r2(); return;}
}
class Reg {
    int f=1; int g=1;
    void r0() {this.f++; return;}
    void r1() {this.g*=2; return;}
    void r2() {this.g++; return;}
}

class Worker1 {
    void w1(Reg rg) {rg.r1(); return;}
}

class Worker2 {
    void w2(Reg rg) {rg.r2(); return;}
}
class Reg {
  int f=1; int g=1;
  void r0() {this.f++; return;}
  void r1() {this.g*=2; return;}
  void r2() {this.g++; return;}
}

class Worker1 {
  void w1(Reg rg) {rg.r1(); return;}
}

class Worker2 {
  void w2(Reg rg) {rg.r2(); return;}
}

To explore r2 before r0 actor wk2 must be selected
class Reg {
    int f=1; int g=1;
    void r0() {this.f++; return;}
    void r1() {this.g*=2; return;}
    void r2() {this.g++; return;}
}

class Worker1 {
    void w1(Reg rg) {rg.r1(); return;}
}

class Worker2 {
    void w2(Reg rg) {rg.r2(); return;}
}

reg: {r0}, wk1: {w1}, wk2: {w2}[wk1]

wk1: {w1}, wk2: {w2}[ ]

reg: {r0}, r1}, wk2: {w2}[wk2]

reg: {r0}, wk1: {w1}, wk2: {w2}[wk1]

reg: {r0}, wk1: {w1}, wk2: {w2}[wk1]

reg: {r0}, r1}, wk2: {w2}[wk2]

reg: {r0}, wk1: {w1}, wk2: {w2}[wk1]

reg: {r0}, r1}, wk2: {w2}[wk2]

reg: {r0}, wk1: {w1}, wk2: {w2}[wk1]

reg: {r0}, r1}, wk2: {w2}[wk2]

reg: {r0}, wk1: {w1}, wk2: {w2}[wk1]

reg: {r0}, r1}, wk2: {w2}[wk2]

reg: {r0}, wk1: {w1}, wk2: {w2}[wk1]

reg: {r0}, r1}, wk2: {w2}[wk2]

reg: {r0}, wk1: {w1}, wk2: {w2}[wk1]

reg: {r0}, r1}, wk2: {w2}[wk2]

reg: {r0}, wk1: {w1}, wk2: {w2}[wk1]

reg: {r0}, r1}, wk2: {w2}[wk2]

reg: {r0}, wk1: {w1}, wk2: {w2}[wk1]

reg: {r0}, r1}, wk2: {w2}[wk2]

reg: {r0}, wk1: {w1}, wk2: {w2}[wk1]

reg: {r0}, r1}, wk2: {w2}[wk2]

reg: {r0}, wk1: {w1}, wk2: {w2}[wk1]

reg: {r0}, r1}, wk2: {w2}[wk2]

reg: {r0}, wk1: {w1}, wk2: {w2}[wk1]

reg: {r0}, r1}, wk2: {w2}[wk2]

reg: {r0}, wk1: {w1}, wk2: {w2}[wk1]

reg: {r0}, r1}, wk2: {w2}[wk2]

reg: {r0}, wk1: {w1}, wk2: {w2}[wk1]

reg: {r0}, r1}, wk2: {w2}[wk2]
class Reg {
    int f = 1; int g = 1;
    void r0() {this.f++; return;}
    void r1() {this.g *= 2; return;}
    void r2() {this.g++; return;}
}

class Worker1 {
    void w1(Reg rg) {rg.r1(); return;}
}

class Worker2 {
    void w2(Reg rg) {rg.r2(); return;}
}

reg: {r0}, wk1: {w1}, wk2: {w2}

reg: {r0}, wk1: {w1}, wk2: {w2}

reg: {r1}, wk2: {w2}

reg: {r1, r2}

reg: {r1, r2}

reg: {r1} reg: {r2} reg: {r1} reg: {r2}

r0 < r1 < r2 r0 < r2 < r1 r0 < r1 < r2 r0 < r2 < r1
```java
class Reg {
    int f=1; int g=1;
    void r0() {this.f++; return;}
    void r1() {this.g*=2; return;}
    void r2() {this.g++; return;}
}

class Worker1 {
    void w1(Reg rg) {rg.r1(); return;}
}

class Worker2 {
    void w2(Reg rg) {rg.r2(); return;}
}
```

Reg: {r0}, wk1: {w1}, wk2: {w2}[

wk1: {w1}, wk2: {w2}[

reg: {r1}, wk2: {w2}[

wk2.w2[

reg: {r1}, r2}[

wk2.w2[

reg: {r0, r1}, wk2: {w2}[

reg: {r0, r1, r2}[

r0 < r1 < r2 r0 < r2 < r1 r0 < r1 < r2 r0 < r2 < r1

reg: {r0} reg: {r2} reg: {r1} reg: {r2}

reg: {r1} reg: {r2}

Elvira Albert
Test Case Generation by Symbolic Execution
16-20 June 2014 42 / 62
class Reg {
    int f=1; int g=1;
    void r0() {this.f++; return;}
    void r1() {this.g*=2; return;}
    void r2() {this.g++; return;}
}

class Worker1 {
    void w1(Reg rg) {rg.r1(); return;}
}

class Worker2 {
    void w2(Reg rg) {rg.r2(); return;}
}

reg: {r0}, wk1: {w1}, wk2: {w2}

reg: {r0}, wk1: {w1}, wk2: {w2}

reg: {r0, r1}, wk2: {w2}

reg: {r0, r1, r2}

reg: {r1}, reg: {r2}

reg: {r1}, reg: {r2}

r0 < r1 < r2

r0 < r1 < r2

r0 < r1 < r2

r0 < r1 < r2

r0 < r1 < r2

r1 < r0 < r2

r1 < r0 < r2

r1 < r0 < r2

r2 < r0 < r1

r2 < r1 < r0

TransDPOR reduces the exploration from 32 to 10 explorations
```java
class Reg {
    int f = 1; int g = 1;
    void r0() { this.f++; return; }
    void r1() { this.g *= 2; return; }
    void r2() { this.g++; return; }
}

class Worker1 {
    void w1(Reg rg) { rg.r1(); return; }
}

class Worker2 {
    void w2(Reg rg) { rg.r2(); return; }
}
```

▶ TransDPOR reduces the exploration from 32 to 10 explorations
▶ But this can be improved further
Effectiveness of (Trans)DPOR highly depends on selection ordering
  • E.g., if $wk_1$ and $wk_2$ are selected before $reg$ no redundant execs are produced

Idea: Select first stable actors
  • An actor is stable if no other actor different from it introduces tasks in its queue
  • If we select a stable actor its backtrack set will remain empty
  • We provide an analysis which computes sufficient cond. for temporal object stability (wrt the actors in that state)
First Contribution: Actor Selection based on Stability Crit.

- Effectiveness of (Trans)DPOR highly depends on selection ordering
  - E.g., if \( wk_1 \) and \( wk_2 \) are selected before \( reg \) no redundant execs are produced
- Idea: Select first stable actors
  - An actor is stable if no other actor different from it introduces tasks in its queue
  - If we select a stable actor its backtrack set will remain empty
  - We provide an analysis which computes sufficient cond. for temporal object stability (wrt the actors in that state)
- Intuition:

\[
\begin{align*}
  ob: \{ t_1 \}, & \quad ob': \{ t' \} \quad (t' \text{ calls } ob.t_2) \\
  ob: \{ t_2 \}, & \quad ob': \{ \} \\
  ob: \{ t_1, t_2 \}, & \quad ob': \{ \} \\
  ob: \{ t_2 \}, & \quad ob': \{ \} \\
  ob: \{ t_1 \}, & \quad ob': \{ \} 
\end{align*}
\]
First Contribution: Actor Selection based on Stability Crit.

- Effectiveness of (Trans)DPOR highly depends on selection ordering
  - E.g., if wk1 and wk2 are selected before reg no redundant execs are produced

- Idea: Select first stable actors
  - An actor is stable if no other actor different from it introduces tasks in its queue
  - If we select a stable actor its backtrack set will remain empty
  - We provide an analysis which computes sufficient cond. for temporal object stability (wrt the actors in that state)

- Intuition:

\[
\begin{align*}
\text{ob:}\{t_1\}, \text{ob':}\{t'\}[] & \quad \text{ob:}\{t_1, t_2\}, \text{ob':}\{}[t_2] \\
\text{ob'} . t' & \quad \text{ob}. t_1 & \quad \text{ob}. t_2 \\
\text{ob:}\{t_2\}[] & \quad \text{ob:}\{t_2\}, \text{ob':}\{}[ ] & \quad \text{ob:}\{t_1\}, \text{ob':}\{}[ ] \\
\text{ob}. t_2 & \quad \text{ob}. t_2 & \quad \text{ob}. t_1 \\
\end{align*}
\]

\(t'\) calls \(ob . t_2\)
```java
class Reg {
    int f=1;  int g=1;
    void r0() {this.f++; return;}
    void r1() {this.g*=2; return;}
    void r2() {this.g++; return;}
}

class Worker1 {
    void w1(Reg rg) {rg.r1(); return;}
}

class Worker2 {
    void w2(Reg rg) {rg.r2(); return;}
}

reg: {r0}, wk1: {w1}, wk2: {w2}

▷ Actor reg is not stable. wk1 and wk2 are stable
```
class Reg {
    int f=1; int g=1;
    void r0() {this.f++; return;}
    void r1() {this.g*=2; return;}
    void r2() {this.g++; return;}
}

class Worker1 {
    void w1(Reg rg) {rg.r1(); return;}
}

class Worker2 {
    void w2(Reg rg) {rg.r2(); return;}
}

reg: {r0}, wk1: {w1}, wk2: {w2}

wk1: {w1}, wk2: {w2}
wk1.w1

reg: {r0, r1}, wk2: {w2}

reg: {r0, r1}, wk2: {w2}
wk1.w1

reg: {r0}, wk1: {w1}, wk2: {w2}

reg: {r0}, wk1: {w1}, wk2: {w2}

reg: {r0, r1}, wk2: {w2}[ ]

reg: {r0}, wk1: {w1}, wk2: {w2}[ ]

Actor reg is not stable. wk2 is stable
This reduces the exploration further, from 10 to 6 executions.
Not always possible finding a stable actor
  • Either because our analysis loses precision or because there is not
  • We propose **Heuristics** based on stability

Experimental evaluation with 10 benchmarks:
  • In 9 of them no backtracking due to actor selection is performed
    • In 99% of the states (thousands, even millions!) a stable actor is found
    • In the remaining 1% the heuristics selects a stable actor
  • In the other benchmark more intelligent heuristics would be required

Our actor selection is very effective in practice and has no significant overhead
Observation: Execs. with different partial order lead to the same state

\[
\begin{align*}
\text{reg:} & \{r0\}, \text{wk1:} \{w1\}, \text{wk2:} \{w2\} \\
\text{wk1.w1} & \downarrow \\
\text{reg:} & \{r0, r1\}, \text{wk2:} \{w2\} \\
\text{wk2.w2} & \downarrow \\
\text{reg:} & \{r0, r1, r2\} \\
\text{r0} & \downarrow \\
\text{r1} & \downarrow \text{r2} \\
\{r0, r2\} & \uparrow \{r0, r1\} \\
\end{align*}
\]

```
class Reg {
    int f=1; int g=1;

    void r0() {this.f++; return;}
    void r1() {this.g*=2; return;}
    void r2() {this.g++; return;}
}
```
Observation: Execs. with different partial order lead to the same state

```
reg:{r0}, wk1:{w1}, wk2:{w2}
wk1.w1 ↓
reg:{r0, r1}, wk2:{w2}
wk2.w2 ↓
reg:{r0, r1, r2}
```

Execution of `r0` is independent from that of `r1` and `r2`

\[
\text{indep}(t,t') \iff t \text{ does not write to fields that } t' \text{ accesses and viceversa}
\]

In the example we have: `indep(r0,r1)` and `indep(r0,r2)`
A Task Selection Algorithm based on Indep. Info.

- Intuition of algorithm:
  - Tasks have an associated mark, and can be marked or unmarked during the execution
  - A marked task cannot be selected
  - When selecting a task, independent tasks after it in the queue are marked, and the rest are unmarked
A Task Selection Algorithm based on Indep. Info.

- **Intuition of algorithm:**
  - Tasks have an associated mark, and can be marked or unmarked during the execution
  - A marked task cannot be selected
  - When selecting a task, independent tasks after it in the queue are marked, and the rest are unmarked

### Algorithm in action

\[
\text{reg:}\{r_0, r_1, r_2\} \quad \text{indep}(r_0, r_1) \text{ and indep}(r_0, r_2)
\]
A Task Selection Algorithm based on Indep. Info.

Intuition of algorithm:

- Tasks have an associated mark, and can be marked or unmarked during the execution
- A marked task cannot be selected
- When selecting a task, independent tasks after it in the queue are marked, and the rest are unmarked

Algorithm in action

\[
\text{reg:}\{r_0, r_1, r_2\} \\
\text{reg:}\{\bar{r}_1, \bar{r}_2\} \\
\text{indep}(r_0, r_1) \text{ and indep}(r_0, r_2)
\]
A Task Selection Algorithm based on Indep. Info.

- **Intuition of algorithm:**
  - Tasks have an associated mark, and can be marked or unmarked during the execution
  - A marked task cannot be selected
  - When selecting a task, independent tasks after it in the queue are marked, and the rest are unmarked

**Algorithm in action**

\[
\begin{align*}
\text{reg:}\{r_0, r_1, r_2\} & \quad \text{indep}(r_0, r_1) \text{ and indep}(r_0, r_2) \\
\text{reg:}\{\bar{r}_1, \bar{r}_2\} & \quad \text{reg:}\{r_0, r_2\} \\
\end{align*}
\]
Intuition of algorithm:

- Tasks have an associated mark, and can be marked or unmarked during the execution
- A marked task cannot be selected
- When selecting a task, independent tasks after it in the queue are marked, and the rest are unmarked

Algorithm in action
A Task Selection Algorithm based on Indep. Info.

- Intuition of algorithm:
  - Tasks have an associated mark, and can be marked or unmarked during the execution
  - A marked task cannot be selected
  - When selecting a task, independent tasks after it in the queue are marked, and the rest are unmarked

Algorithm in action

```
reg:{r0, r1, r2}

reg:{r1, r2}

reg:{r2}

reg:{r0}

f:2

reg:{r0}

reg:{r1}

reg:{r2}

indep(r0,r1) and indep(r0,r2)
```
A Task Selection Algorithm based on Indep. Info.

- **Intuition of algorithm:**
  - Tasks have an associated mark, and can be marked or unmarked during the execution
  - A marked task cannot be selected
  - When selecting a task, independent tasks after it in the queue are marked, and the rest are unmarked

**Algorithm in action**

```
reg: {r0, r1, r2}

reg: {r1, r2}
reg: {r0, r1}
reg: {r0}
reg: {r0}
reg: {r1}
reg: {r2}
```

\[ \text{indep}(r0, r1) \text{ and indep}(r0, r2) \]
A Task Selection Algorithm based on Indep. Info.

- Intuition of algorithm:
  - Tasks have an associated mark, and can be marked or unmarked during the execution
  - A marked task cannot be selected
  - When selecting a task, independent tasks after it in the queue are marked, and the rest are unmarked

Algorithm in action

```
reg: {r0, r1, r2}

reg: {r1, r2}
reg: {r0, r2}
reg: {r0, r1}

reg: {r2}
reg: {r0}
reg: {r1}

r0
r1
r2

indep(r0,r1) and indep(r0,r2)
```
A Task Selection Algorithm based on Indep. Info.

- Intuition of algorithm:
  - Tasks have an associated mark, and can be marked or unmarked during the execution
  - A marked task cannot be selected
  - When selecting a task, independent tasks after it in the queue are marked, and the rest are unmarked

Algorithm in action

Independent tasks are selected consecutively just in a single order
Experimental Results

<table>
<thead>
<tr>
<th>Test name</th>
<th>Execs</th>
<th>Time</th>
<th>States</th>
<th>H</th>
<th>Execs</th>
<th>Time</th>
<th>States</th>
<th>H</th>
<th>Execs</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>QSort.test1</td>
<td>4</td>
<td>2</td>
<td>18</td>
<td>3</td>
<td>4</td>
<td>2</td>
<td>18</td>
<td>3</td>
<td>1.0x</td>
<td>1.0x</td>
</tr>
<tr>
<td>QSort.test2</td>
<td>16</td>
<td>10</td>
<td>70</td>
<td>21</td>
<td>16</td>
<td>10</td>
<td>70</td>
<td>21</td>
<td>1.0x</td>
<td>1.0x</td>
</tr>
<tr>
<td>Fib.test1</td>
<td>4</td>
<td>3</td>
<td>18</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>18</td>
<td>3</td>
<td>1.0x</td>
<td>1.0x</td>
</tr>
<tr>
<td>Fib.test2</td>
<td>128</td>
<td>80</td>
<td>524</td>
<td>189</td>
<td>128</td>
<td>81</td>
<td>524</td>
<td>189</td>
<td>1.0x</td>
<td>1.0x</td>
</tr>
<tr>
<td>PSort.test1</td>
<td>288</td>
<td>69</td>
<td>1294</td>
<td>144</td>
<td>288</td>
<td>71</td>
<td>1294</td>
<td>144</td>
<td>1.0x</td>
<td>1.0x</td>
</tr>
<tr>
<td>PSort.test2</td>
<td>5760</td>
<td>1385</td>
<td>25829</td>
<td>2880</td>
<td>288</td>
<td>71</td>
<td>1304</td>
<td>144</td>
<td>20.0x</td>
<td>19.5x</td>
</tr>
<tr>
<td>RegSim.test1</td>
<td>10080</td>
<td>806</td>
<td>27415</td>
<td>0</td>
<td>720</td>
<td>136</td>
<td>3923</td>
<td>0</td>
<td>14.0x</td>
<td>5.9x</td>
</tr>
<tr>
<td>RegSim.test2</td>
<td>11520</td>
<td>864</td>
<td>31576</td>
<td>0</td>
<td>384</td>
<td>70</td>
<td>2132</td>
<td>0</td>
<td>30.0x</td>
<td>12.3x</td>
</tr>
<tr>
<td>DHT.test1</td>
<td>1152</td>
<td>137</td>
<td>3905</td>
<td>0</td>
<td>36</td>
<td>6</td>
<td>141</td>
<td>0</td>
<td>32.0x</td>
<td>22.8x</td>
</tr>
<tr>
<td>DHT.test2</td>
<td>480</td>
<td>97</td>
<td>2304</td>
<td>0</td>
<td>12</td>
<td>4</td>
<td>85</td>
<td>0</td>
<td>40.0x</td>
<td>24.2x</td>
</tr>
<tr>
<td>Mail.test1</td>
<td>2648</td>
<td>557</td>
<td>11377</td>
<td>0</td>
<td>460</td>
<td>120</td>
<td>2270</td>
<td>0</td>
<td>5.8x</td>
<td>4.6x</td>
</tr>
<tr>
<td>Mail.test2</td>
<td>1665500</td>
<td>&gt;200s</td>
<td>5109783</td>
<td>0</td>
<td>27880</td>
<td>4064</td>
<td>94222</td>
<td>0</td>
<td>&gt;60x</td>
<td>49.2x</td>
</tr>
<tr>
<td>BB.test1</td>
<td>155520</td>
<td>23907</td>
<td>475205</td>
<td>0</td>
<td>4320</td>
<td>681</td>
<td>13214</td>
<td>0</td>
<td>36.0x</td>
<td>35.1x</td>
</tr>
<tr>
<td>BB.test2</td>
<td>1099008</td>
<td>165114</td>
<td>3028298</td>
<td>0</td>
<td>45792</td>
<td>6945</td>
<td>126192</td>
<td>0</td>
<td>24.0x</td>
<td>23.8x</td>
</tr>
</tbody>
</table>

- Except for the first two benchmarks, the pruning is huge, the speedup ranging from one to two orders of magnitude
Plan of the Lecture

- **Part 1: Symbolic execution and TCG**
  - Introduction
  - Handling heap-manipulating programs
  - Compositionallity

- **Part 2: CLP-based TCG**
  - Introduction
  - Translation from imperative to CLP
  - Guided-TCG
  - Demo

- **Part 3: TCG of Concurrent (Actor) Programs**
  - The path explotion problem
  - Symbolic execution and TCG for actors
  - Demo
Define a TCG framework for Actors:

- Symbolic execution (previous part)
- Termination criteria
- Coverage criteria
- TCG with synchronization primitives (await and get)
Coverage and Termination Criteria for Concurrent Objects

- *loop-k* coverage criteria: limits the number of times we iterate on loops for a task (similar to the sequential setting).

```java
class A {
    int f = 1;
    void choose(int n, int m) {
        if (n <= m) then this ! p(n);
        else this ! q(m);
    }
    void p(int n) {
        while (n > 0) {
            this.f = this.f * n;
            n = n - 1;
        }
    }
    ...
}
```

```
choose(N,M)
\{N <= M\}

p(N)
```

Branch not Explored
Infinite Branch

```
\{N <= M, N > 0, N1 = N - 1, this.f = N\}
\{N <= M, N = 0, this.f = 1\}
```

Elvira Albert
Test Case Generation by Symbolic Execution
16-20 June 2014 51 / 62
Coverage and Termination Criteria for Concurrent Objects

- **loop-k coverage criteria**: limits the number of times we iterate on loops for a task (similar to the sequential setting).

```java
class A {
    int f = 1;
    void choose(int n, int m) {
        if (n <= m) then this ! p(n);
        else this ! q(m);
    }
    void p(int n) {
        while (n > 0) {
            this.f = this.f * n;
            n = n - 1;
        }
    }
    ...
}
```

```
choose(N,M)

{N <= M}

p(N)

while (N > 0)
```
Coverage and Termination Criteria for Concurrent Objects

- **loop-*k* coverage criteria**: limits the number of times we iterate on loops for a task (similar to the sequential setting).

```java
class A {
    int f = 1;
    void choose(int n, int m) {
        if (n ≤ m) then this ! p(n);
        else this ! q(m);
    }
    void p(int n) {
        while (n > 0) {
            this.f = this.f * n;
            n = n - 1;
        }
    }
    . . .
}
```
Coverage and Termination Criteria for Concurrent Objects

- **loop-k coverage criteria**: limits the number of times we iterate on loops for a task (similar to the sequential setting).

```java
class A {
    int f = 1;

    void choose(int n, int m) {
        if (n <= m) then this! p(n);
        else this! q(m);
    }

    void p(int n) {
        while (n > 0) {
            this.f = this.f * n;
            n = n - 1;
        }
    }
}
```

Elvira Albert
Test Case Generation by Symbolic Execution 16-20 June 2014 51 / 62
Coverage and Termination Criteria for Concurrent Objects

- \textit{loop-}k coverage criteria: limits the number of times we iterate on loops for a task (similar to the sequential setting).

```java
class A {
    int f = 1;

    void choose(int n, int m) {
        if (n \leq m) then this ! p(n);
        else this ! q(m);
    }

    void p(int n) {
        while (n > 0) {
            this.f = this.f * n;
            n = n - 1;
        }
    }
}

choose(N,M)
    {N <= M}
    p(N)
    while (N > 0)
        {N = 0}
        {N > 0}
        {N <= M, N = 0, this.f = 1}
        while (N1 > 0)
            {N1 = 0}
            {N <= M, N > 0, N1 = N - 1, this.f = N}
            {N <= M, N > 0, N1 = 0}
            {N1 = N - 1, this.f = N}
```
Coverage and Termination Criteria for Concurrent Objects

- loop-\(k\) coverage criteria: limits the number of times we iterate on loops for a task (similar to the sequential setting).

```
class A {
    int f = 1;
    void choose(int n, int m) {
        if (n ≤ m) then this ! p(n);
        else this ! q(m);
    }
    void p(int n) {
        while (n > 0) {
            this.f = this.f * n;
            n = n - 1;
        }
    }
    ...  
choose(N,M)

```

```
{N <= M}
    p(N)
        while (N > 0)
            {N = 0}
            {N > 0}
        {N <= M, N = 0, this.f = 1}
        {N1 = 0}
        while (N1 > 0)
            {N <= M, N > 0, N1 = N - 1, this.f = N}
            {N1 = 0}
            {N1 > 0}
            {N <= M, N > 0, N1 = 0, N1 = N - 1, this.f = N} Infinite Branch
```
Symbolic Execution and TCG for Actor Models

Coverage and Termination Criteria for Concurrent Objects

- **loop-k coverage criteria**: limits the number of times we iterate on loops for a task (similar to the sequential setting).

```java
class A {
    int f = 1;
    void choose(int n, int m) {
        if (n ≤ m) then this ! p(n); else this ! q(m);
    }
    void p(int n) {
        while (n > 0) {
            this.f = this.f * n;
            n = n - 1;
        }
    }
    ... 
}
```

![Diagram showing the flow of control for the `choose` method and its submethods `p` and `q`. The diagram includes state transitions and branch labels to illustrate the coverage criteria.](image)
Coverage and Termination Criteria for Concurrent Objects

- **loop-k coverage criteria**: limits the number of times we iterate on loops for a task (similar to the sequential setting).

```java
class A {
    int f = 1;
    void choose(int n, int m) {
        if (n <= m) then this ! p(n);
        else this ! q(m);
    }
    void p(int n) {
        while (n > 0) {
            this.f = this.f * n;
            n = n - 1;
        }
    }
    ...
}

loop-k = 1

choose(N,M)
while (N > 0)
{N <= M}
while (N1 > 0)
{N >= M, N > 0, N1 = N - 1, this.f = N}
{N1 = 0}
{N <= M, N = 0, this.f = 1}
{N > 0}

p(N)
q(M)

while (N1 > 0)
{N1 > 0}
Infinite Branch
{N <= M, N > 0,
 N1 = N - 1, this.f = N}

Elvira Albert
Test Case Generation by Symbolic Execution 16-20 June 2014
51 / 62
Coverage and Termination Criteria for Concurrent Objects

*Task-switching* coverage criteria: limit the number of task switches per object

```java
class A {
    int f = 1;
    void choose(int n, int m) {
        if (n ≤ m) then this ! p(n);
        else this ! q(m);
    }
    void p(int n) {
        if (n > 0) then {
            this.f = this.f * n;
            this ! p(n-1);
        }
    }
    ...
}
```

Diagram:
```
choose(N,M)
{N ≤ M}
    p(N)
      {N = 0}
      {N > 0}
        p(N1)
          {0 ≤ M, N = 0, this.f = 1}
          {this.f = this.f * N, N1 = N - 1}
            p(N2)
              {1 ≤ M, N = 1, this.f = 1}
              {this.f = this.f * N2, N2 = N1 - 1}
                {N2 = 0}
                {N2 > 0}
                  {2 ≤ M, N = 2, this.f = 2}
Infinite Branch
```
Coverage and Termination Criteria for Concurrent Objects

Task-switching coverage criteria: limit the number of task switches per object

```java
class A {
    int f = 1;
    void choose(int n, int m) {
        if (n ≤ m) then this ! p(n);
        else this ! q(m);
    }
    void p(int n) {
        if (n > 0) then {
            this.f = this.f * n;
            this ! p(n-1);
        }
    }
    ...
}
```

choose(N,M)
---
\[
\begin{align*}
&\text{N} \leq M \\
&\text{p(N)} \text{ new task (loop-k not applicable)} \\
&\quad \begin{cases}
&\quad \text{N} = 0 \\
&\quad \text{N > 0}
&\end{cases}
\end{align*}
\]

\[
\begin{align*}
&\text{0} \leq M, \text{N} = 0, \text{this.f} = 1 \\
&\text{p(N1)} \text{ new task (loop-k not applicable)} \\
&\quad \begin{cases}
&\quad \text{this.f} = \text{this.f} \ast N, \text{N1} = \text{N} - 1 \\
&\quad \text{N1} = 0 \\
&\quad \text{N1 > 0}
&\end{cases}
\end{align*}
\]

\[
\begin{align*}
&\text{1} \leq M, \text{N} = 1, \text{this.f} = 1 \\
&\text{p(N2)} \text{ new task (loop-k not applicable)} \\
&\quad \begin{cases}
&\quad \text{this.f} = \text{this.f} \ast N2, \text{N2} = \text{N1} - 1 \\
&\quad \text{N2} = 0 \\
&\quad \text{N2 > 0}
&\end{cases}
\end{align*}
\]

\[
\begin{align*}
&\text{2} \leq M, \text{N} = 2, \text{this.f} = 2 \\
&\quad \text{Infinite Branch}
\end{align*}
\]
Symbolic Execution and TCG for Actor Models

Coverage and Termination Criteria for Concurrent Objects

Task-switching coverage criteria: limit the number of task switches per object

class A {
    int f = 1;
    void choose(int n, int m) {
        if (n ≤ m) then this!p(n);
        else this!q(m);
    }
    void p(int n) {
        if (n > 0) then {
            this.f = this.f * n;
            this!p(n-1);
        }
    }
    ...

    choose(N,M) {N ≤ M}
p(N) {N = 0}
    q(M) {N > 0}
p(N1) {0 ≤ M, N = 0, this.f = 1}
    p(N2) {1 ≤ M, N = 1, this.f = 1}
    p(N3) {2 ≤ M, N = 2, this.f = 2}
}
Coverage and Termination Criteria for Concurrent Objects

*Number of objects* coverage criteria: limits the total number of created objects during the execution.

```java
class A {
    void choose(int n, int m) {
        if (n <= m) then this ! p(n);
        else this ! q(m);
    }

    void p(int n) {
        if (n==0) then bodyThen
        else {
            A a = new A(. . .);
            a ! p(n-1);
        }
    }
}
...
```

$\{N \leq M\}$
$q(M)$
$\{N = 0\}$
$\{N1 = 0\}$
$\{N > 0\}$
$\{N1 > 0\}$
bodyThen
$A1 ! p(N)$
$A1 ! choose(N, M)$
$A1$
$A2 ! p(N1)$
$\{N1 = N−1\}$
$\{N2 = 0\}$
$\{N2 > 0\}$
bodyThen
$\text{Branch not Explored}$
$\{N2 = N1−1\}$
$\text{global number of objects = 3}$
Coverage and Termination Criteria for Concurrent Objects

*Number of objects* coverage criteria: limits the total number of created objects during the execution.

```java
class A {
    void choose(int n, int m) {
        if (n ≤ m) then this ! p(n);
        else this ! q(m);
    }

    void p(int n) {
        if (n==0) then bodyThen
        else {
            A a = new A(...);
            a ! p(n-1);
        }
    }

    ...
}
```
Symbolic Execution and TCG for Actor Models

Coverage and Termination Criteria for Concurrent Objects

*Number of objects* coverage criteria: limits the total number of created objects during the execution.

```java
class A {
  void choose(int n, int m) {
    if (n <= m) then this ! p(n); 
    else this ! q(m);
  }

  void p(int n) {
    if (n==0) then bodyThen
    else { 
      A a = new A(...);
      a ! p(n-1);
    }
  }

  ... 
}
```

Global number of objects = 3

```
A1 ! choose(N,M)

{N <= M}

A1 ! p(N)

{N = 0} {N > 0}

bodyThen

A1

A2 ! p(N1) {N1 = N−1} {N1 > 0}

bodyThen

A1

A1

A2 ! p(N1) {N1 = N−1} {N1 > 0}

A3 ! p(N2) {N2 = N1−1}

{N2 = 0} {N2 > 0}

bodyThen

Infinite Branch
```
Coverage and Termination Criteria for Concurrent Objects

*Number of objects* coverage criteria: limits the total number of created objects during the execution.

```
class A {
  void choose(int n, int m) {
    if (n \leq m) then this ! p(n);
    else this ! q(m);
  }
  void p(int n) {
    if (n==0) then bodyThen
    else {
      A a = new A(...);
      a ! p(n-1);
    }
  }
  ...
}
```

```
{N <= M}
A1 ! choose(N,M)

{N = 0}
A1 ! p(N)

{N > 0}
A1

{N1 = 0}
2 A2 ! p(N1)

A1

{N1 > 0}
A1

{N2 = N1-1}
3 A3 ! p(N2)

A1

{N2 > 0}
A1

global number of objects = 3
Branch not Explored
{N2 = N1−1}
bodyThen
```

Infinite Branch
**await** and **get** primitives

- **await** `x?:` If the value of `x` is ready, then the execution proceeds. Otherwise, the execution from **await** `x?` on is stored in the queue of tasks of the current object, and a new task is selected to be executed.

- **get** `y = x`.

- **get** `y = o ! q(n);
  **await** y?;
  z = y`.

Elvira Albert

Test Case Generation by Symbolic Execution

16-20 June 2014

54 / 62
**await and get primitives**

- **await x?**: If the value of x is ready, then the execution proceeds. Otherwise, the execution from **await x?** on is stored in the queue of tasks of the current object, and a new task is selected to be executed.
Synchronization Primitives for Concurrent Objects

**await** and **get** primitives

- **await** `x?`: If the value of `x` is ready, then the execution proceeds. Otherwise, the execution from `await x?` on is stored in the queue of tasks of the current object, and a new task is selected to be executed.

- `y = x.get`: If the value of `x` is ready then the execution proceeds. Otherwise the execution in the current object is blocked until the value of `x` be ready. Another task is selected to be executed.
**await** and **get** primitives

- **await** x?: If the value of x is ready, then the execution proceeds. Otherwise, the execution from **await** x? on is stored in the queue of tasks of the current object, and a new task is selected to be executed.

- y = x.get: If the value of x is ready then the execution proceeds. Otherwise, the execution in the current object is blocked until the value of x be ready. Another task is selected to be executed.

```plaintext
y = o ! q(n);
await y?;
z = y.get;
```
Task Interleavings

- When a task $t$ suspends, there could be other tasks on the same object whose execution at this point could interleave with $t$ and modify the information stored in the heap.
Synchronization Primitives for Concurrent Objects

Task Interleavings

- When a task *t* suspends, there could be other tasks on the same object whose execution at this point could interleave with *t* and modify the information stored in the heap.

```java
class A {
    int n;
    int p(...) {
        n=0;
        await ...;
        if (n ≥ 0) ...; else ...;
    }
}
```

The symbolic execution of *p* will consider just the path that goes through the *if* branch; there can be another task (suspended in the queue of the object) which executes when *p* suspends and writes a negative value on *n*. This would exercise the *else* branch when *p* resumes.
Task Interleavings

- When a task \( t \) suspends, there could be other tasks on the same object whose execution at this point could interleave with \( t \) and modify the information stored in the heap.

    ```java
    class A {
        int n;
        int p(...) {
            n=0;
            await ...;
            if (n ≥ 0) ...; else ...;
        }
    }
    ```

- The symbolic execution of \( p \) will consider just the path that goes through the if branch;
Synchronization Primitives for Concurrent Objects

Task Interleavings

- When a task $t$ suspends, there could be other tasks on the same object whose execution at this point could interleave with $t$ and modify the information stored in the heap.

```java
class A {
    int n;
    int p(...) {
        n=0;
        await ...;
        if (n $\geq$ 0) ...; else ...;
    }
}
```

- The symbolic execution of $p$ will consider just the path that goes through the $if$ branch;
- There can be another task (suspended in the queue of the object) which executes when $p$ suspends and writes a negative value on $n$. This would exercise the $else$ branch when $p$ resumes.
Local Trace

Given a method \( m \), the local trace associated with an execution of \( m \) is the sequence of instructions that belong to \( m \).
Local Trace

Given a method $m$, the **local trace** associated with an execution of $m$ is the sequence of instructions that belong to $m$.

- We look at the local trace rather than at the global trace since, when testing $m$, our aim is to ensure proper coverage of the instructions in method $m$. 

Task Interleavings

Local Trace

Given a method \( m \), the **local trace** associated with an execution of \( m \) is the sequence of instructions that belong to \( m \).

- We look at the local trace rather than at the global trace since, when testing \( m \), our aim is to ensure proper coverage of the instructions in method \( m \).
- The objective is to overapproximate, for each method \( m \), the set \( \text{related}(m) \), which contains all methods whose interleaved execution with \( m \) can lead to a local execution not considered before.
Local Trace

Given a method \( m \), the local trace associated with an execution of \( m \) is the sequence of instructions that belong to \( m \).

- We look at the local trace rather than at the global trace since, when testing \( m \), our aim is to ensure proper coverage of the instructions in method \( m \).
- The objective is to overapproximate, for each method \( m \), the set \( \text{related}(m) \), which contains all methods whose interleaved execution with \( m \) can lead to a local execution not considered before.
- Initially \( \text{related}(m) \) will contain all methods of the class under test.
- Limit the size of the queue.
Reducing the set $\text{related}(m)$

Pruning 1

Discard those methods which do not modify the heap
Reducing the set related(m)

Pruning 1
Discard those methods which do not modify the heap

```java
class A {
    int f;
    int g;
    int p(B o, int n) {
        this.f = this.f + 1;
        y = o ! q(n);
        await y?;
        z = y.get;
        return z + this.f;
    }
    void setF(int v) { this.f = v; }
    void setG(int v) { this.G = v; }
    void set(int v1, int v2) { this.setF(v1); this.setG(v2); }
}
⇒ related(p) = {setF, setG, set}
```
Reducing the set $\text{related}(m)$

Pruning 2

Pruning 1 but discarding also those methods which modify the heap transitively (not directly)
Reducing the set $\text{related}(m)$

Pruning 2

Pruning 1 but discarding also those methods which modify the heap transitively (not directly)

class A {
    int f;
    int g;
    int p(B o, int n) {
        this.f = this.f + 1;
        y = o ! q(n);
        await y?;
        z = y.get;
        return z + this.f;
    }
    void setF(int v) { this.f = v; }
    void setG(int v) { this.G = v; }
    void set(int v1, int v2) { this.setF(v1); this.setG(v2); }
}

$\Rightarrow \text{related}(p) = \{\text{setF}, \text{setG}\}$
Reducing the set $\text{related}(m)$

Pruning 3

Consider only interleavings with those methods that write directly on fields which are used before an `await` and used after the `await`
Reducing the set $\text{related}(m)$

Pruning 3

Consider only interleavings with those methods that write directly on fields which are used before an `await` and used after the `await`

class A {
    int f;
    int g;
    int p(B o, int n) {
        this.f = this.f + 1;
        y = o ! q(n);
        await y?;
        z = y.get;
        return z + this.f;
    }
    void setF(int v) { this.f = v; }
    void setG(int v) { this.G = v; }
    void set(int v1, int v2) { this.setF(v1); this.setG(v2); }
}

$\Rightarrow \text{related}(p) = \{\text{setF}\}$
Plan of the Lecture

▶ Part 1: Symbolic execution and TCG
  • Introduction
  • Handling heap-manipulating programs
  • Compositionallity

▶ Part 2: CLP-based TCG
  • Introduction
  • Translation from imperative to CLP
  • Guided-TCG
  • Demo

▶ Part 3: TCG of Concurrent (Actor) Programs
  • The path explotion problem
  • Symbolic execution and TCG for actors
  • Demo
Conclusions & References (Part 3)

Conclusions

- Symbolic execution of actor systems [PADL’12]
- We have proposed termination and coverage criteria for actors
- We have proposed different prunings to consider task interleavings in TCG [ICLP’12]
- An implementation of the technique [ACM/FSE’13]
- We have proposed two improvements to the state-of-the-art algorithm for testing actor systems [FORTE’14]
  1. Actor selection strategy based on actors stability
  2. Task selection based on task independence

Ongoing/Future Work

- Experiment with more intelligent heuristics
- Improve sufficient condition for task independence
Conclusions & References (Part 3)

Conclusions

- Symbolic execution of actor systems [PADL’12]
- We have proposed termination and coverage criteria for actors
- We have proposed different prunings to consider task interleavings in TCG [ICLP’12]
- An implementation of the technique [ACM/FSE’13]
- We have proposed two improvements to the state-of-the-art algorithm for testing actor systems [FORTE’14]
  1. Actor selection strategy based on actors stability
  2. Task selection based on task independence

Ongoing/Future Work

- Experiment with more intelligent heuristics
- Improve sufficient condition for task independence
Conclusions

(CLP-based) TCG based on Symbolic Execution:
- Symbolic execution is the standard approach to generating glass-box test cases statically
- The main challenges in TCG based on symbolic execution are related to the scalability of the approach
- We have presented a (scalable) approach to TCG of heap-manipulating programs
- We have studied compositionallity in TCG
- Guided TCG

CLP-based TCG for Actor Systems:
- Novel termination and coverage criteria
- Elimination of redundant exploration
- Consider tasks interleavings