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:

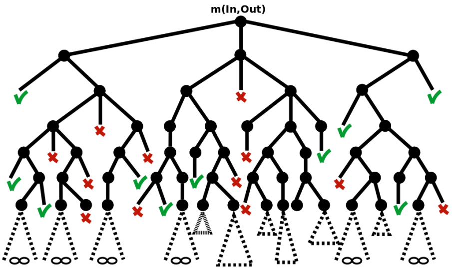
- Random vs. **non-random** ⇒ difficult to obtain high degree of code coverage in random unless consider huge number of inputs
- Black-box vs. **white-box** ⇒ test cases obtained from specifications vs. from program
- Dynamic vs. **static** ⇒ depending if input variables are instantiated

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

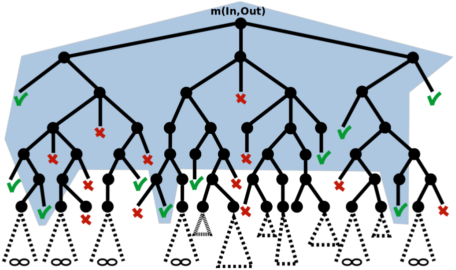


White-box Test Case Generation

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 explotion 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

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

Elements Involved in the Testing Process

Java Code

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

Test Cases

$\{ \langle X >= 0, Z = X \rangle,$
$\langle X < 0, Z = -X \rangle \}$

Elements Involved in the Testing Process

Java Code

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

Concrete Inputs

$\{ \langle X = 1, Z = 1\rangle, \langle X = -1, Z = 1\rangle \}$

Test Cases

$\{ \langle X >= 0, Z = X\rangle, \langle X < 0, Z = -X\rangle \}$

Elements Involved in the Testing Process

Java Code

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

Test Cases

$\{ \langle X >= 0, Z = X\rangle,$
$\langle X < 0, Z = -X\rangle \}$

Concrete Inputs

$\{ \langle X = 1, Z = 1\rangle,$
$\langle X = -1, Z = 1\rangle \}$

JUnit Code

```
void test_abs(){
  assertEquals(abs(1),1);
  assertEquals(abs(-1,1));
}
```

Termination Criteria

Java source code

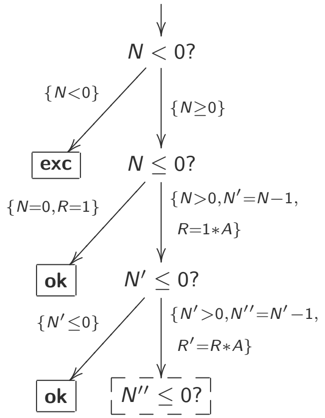
```
int exp(int a,int n) {
  if (n < 0)
    throw new Exception();
  else {
    int r = 1;
    while (n > 0) {
      r = r*a;
      n--;
    }
    return r;
  }
}
```

Termination Criteria

Java source code

```
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



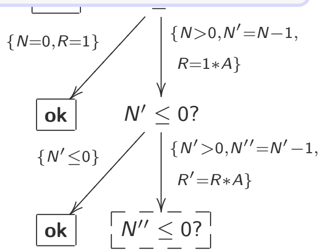
Termination Criteria

Test cases

#	Input	Output	Path condition
1	[A, N]	[exception]	{N<0}
2	[A, N]	1	{N=0}
3	[A, N]	R	{N>0,N'=N-1,N'<=0,R=1*A}

Jav

```
int
    while (n > 0) {
      r = r*a;
      n--;
    }
    return r;
  }
}
```



Termination Criteria

Test cases

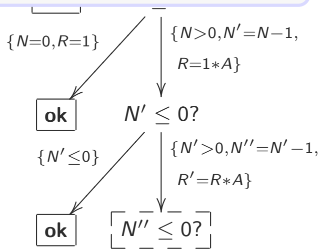
#	Input	Output	Path condition
1	[A, N]	[exception]	{N<0}
2	[A, N]	1	{N=0}
3	[A, N]	R	{N>0,N'=N-1,N'<=0,R=1*A}

Concrete inputs

#	Input	Output
1	[-10, -10]	[Exception]
2	[-10, 0]	1
3	[-10, 1]	-10

Jav

```
int
①
②
③
④
⑤
⑥
⑦
⑧
⑨
⑩
}
```



Termination Criteria

Jav

int
①
②
③
④
⑤
⑥
⑦
⑧
⑨
⑩
}

Test cases

#	Input	
1	[A, N]	[ex
2	[A, N]	
3	[A, N]	

Concrete input

#	Input
1	[-10, -10]
2	[-10, 0]
3	[-10, 1]

Unit tests (JUnit)

```
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;
  }
  fail("Fail");
}
public void test_2(){
  int input0 = -10, input1 = 0;
  int output = Test.intExp(input0,input1);
  int expected = 1;
  assertEquals("OK",expected,output);
}
public void test_3(){
  int input0 = -10, input1 = 1;
  int output = Test.intExp(input0,input1);
  int expected = -10;
  assertEquals("OK",expected,output);
}
```

1,

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

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

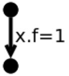
Handling Heap-manipulating Programs

- 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);
}
```

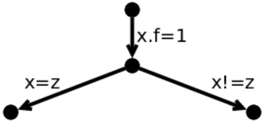
- Standard technique to handle aliasing. Used in state-of-the-art systems, e.g., PET (UCM&UPM) and SPF (NASA Ames)

```
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);
}
```



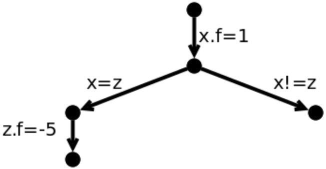
- Standard technique to handle aliasing. Used in state-of-the-art systems, e.g., PET (UCM&UPM) and SPF (NASA Ames)

```
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);
}
```



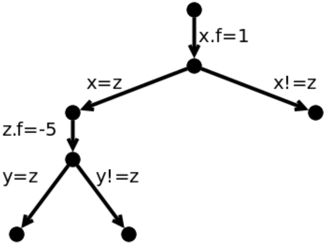
- Field accesses on unknown references trigger non-determinism: 1) Null 2) New reference 3) Each aliasing possibility

```
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);
}
```



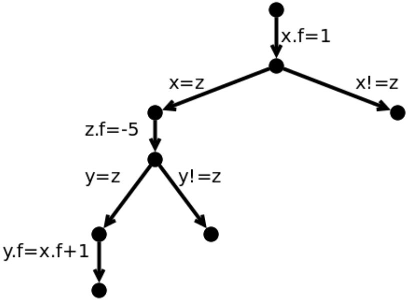
- Field accesses on unknown references trigger non-determinism: 1) Null 2) New reference 3) Each aliasing possibility

```
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);
}
```



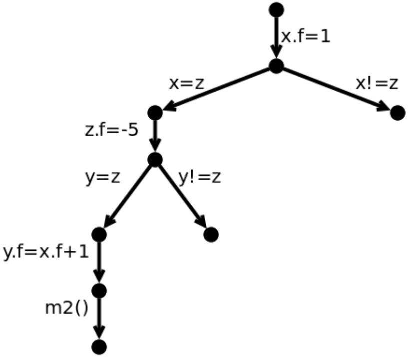
- Field accesses on unknown references trigger non-determinism: 1) Null   2) New reference   3) Each aliasing possibility

```
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);
}
```



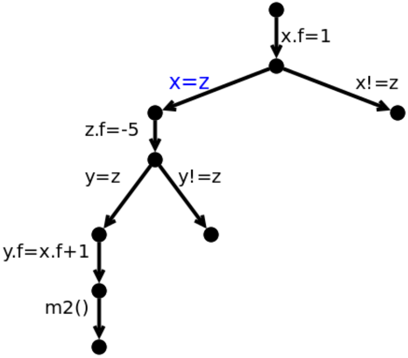
- Field accesses on unknown references trigger non-determinism: 1) Null 2) New reference 3) Each aliasing possibility

```
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);
}
```



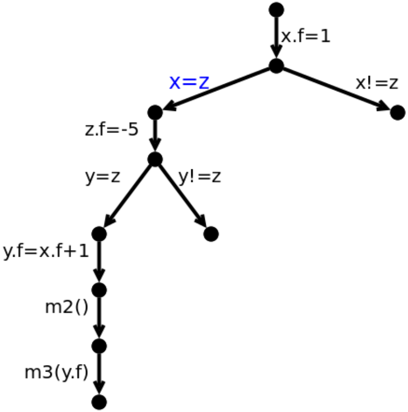
- Field accesses on unknown references trigger non-determinism: 1) Null 2) New reference 3) Each aliasing possibility

```
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);
}
```



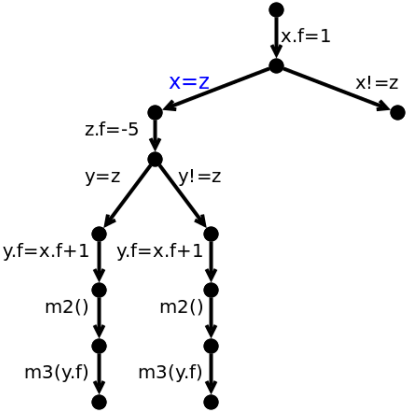
- Field accesses on unknown references trigger non-determinism: 1) Null 2) New reference 3) Each aliasing possibility

```
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);
}
```



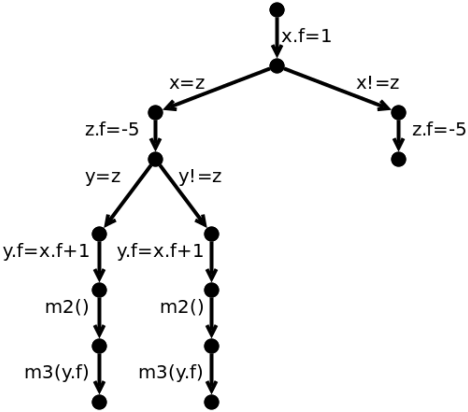
- Symbolic execution quickly becomes impractical
- Redundant exploration of large number of paths

```
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);
}
```



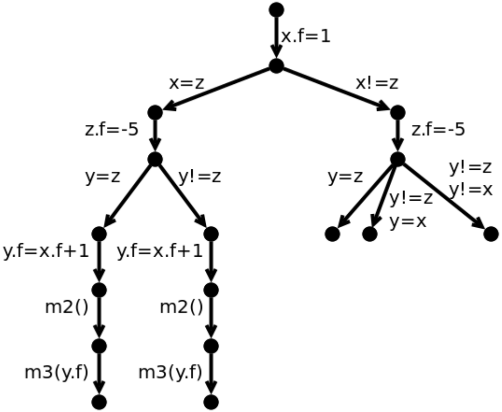
- Symbolic execution quickly becomes impractical
- Redundant exploration of large number of paths

```
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);
}
```



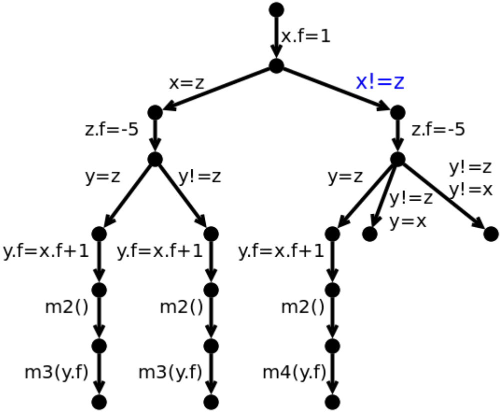
- Symbolic execution quickly becomes impractical
- Redundant exploration of large number of paths

```
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);
}
```



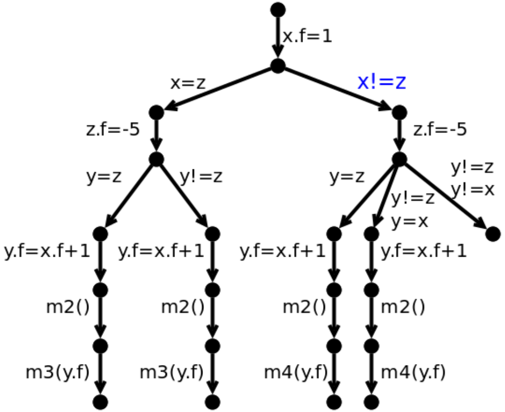
- Symbolic execution quickly becomes impractical
- Redundant exploration of large number of paths

```
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);
}
```



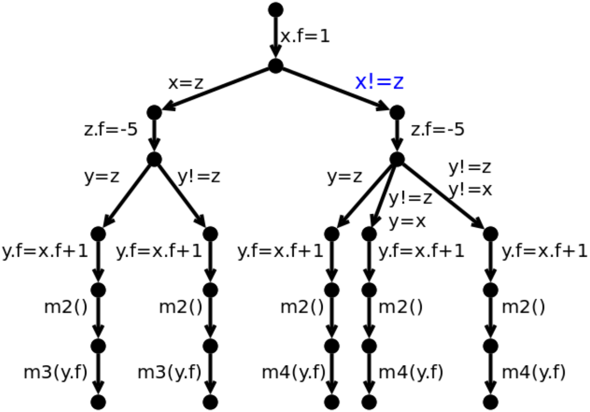
- Symbolic execution quickly becomes impractical
- Redundant exploration of large number of paths

```
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

```
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);
}
```

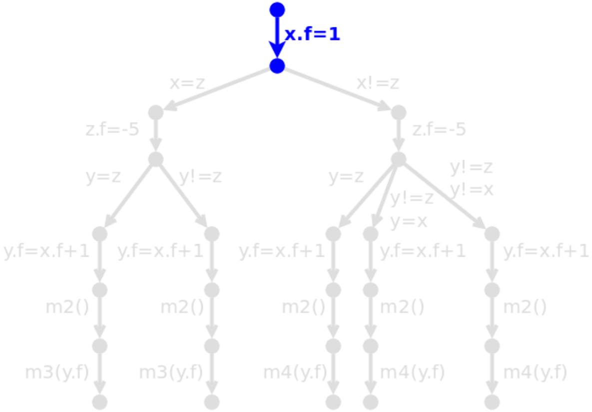


- Symbolic execution quickly becomes impractical
- Redundant exploration of large number of paths

```
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);
}
```

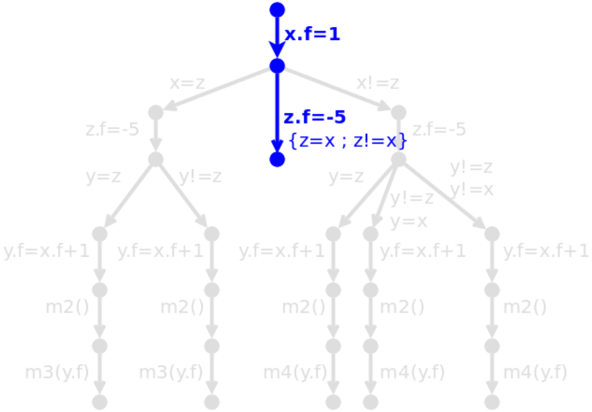

- A more scalable approach than lazy initialization

```
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);
}
```



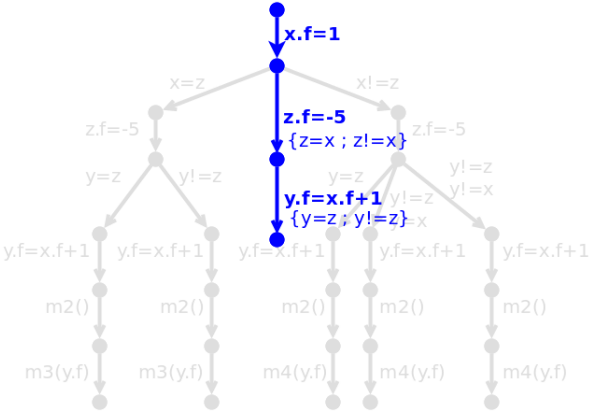
- A more scalable approach than lazy initialization

```
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);
}
```



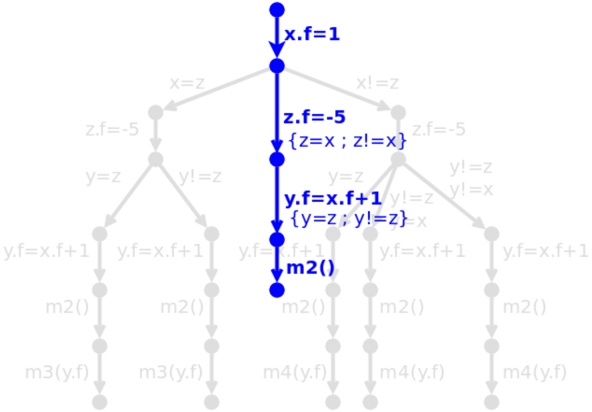
- Avoid non-determinism as much as possible

```
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);
}
```



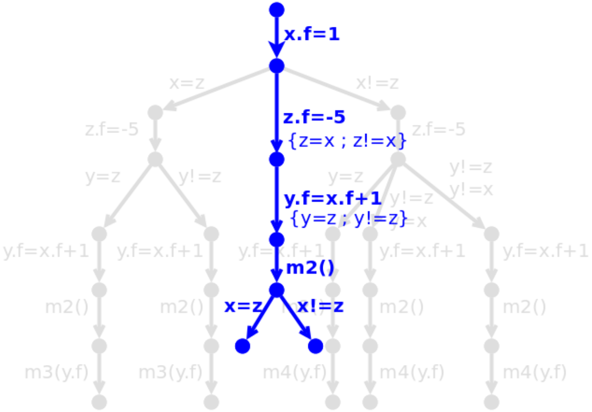
- Treatment of reference aliasing by means of disjunctions

```
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);
}
```



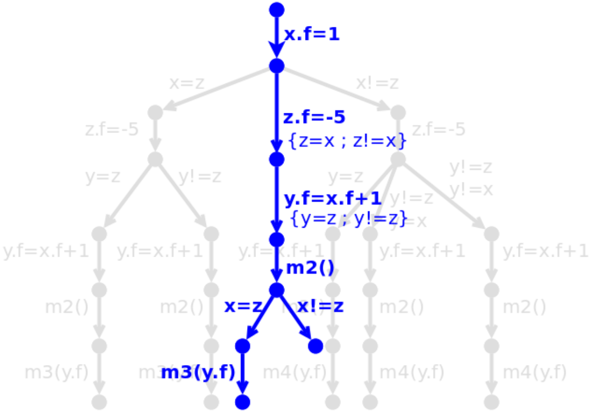
- Treatment of reference aliasing by means of disjunctions

```
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);
}
```



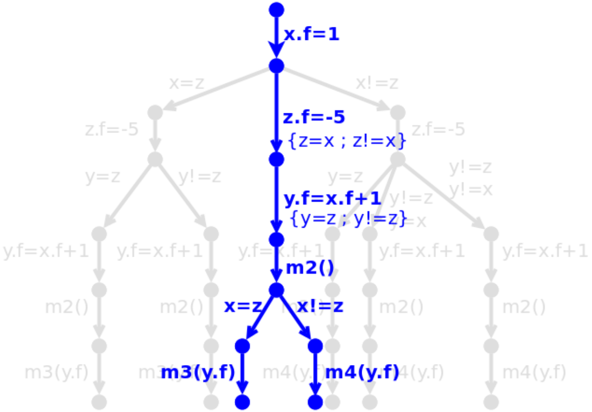
- Propagation of heap-related constraints

```
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)

```
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);
}
```

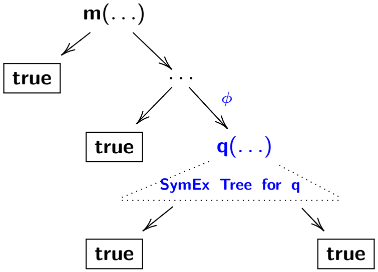


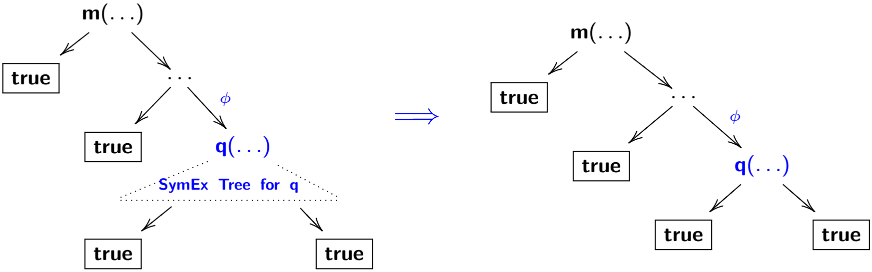
- Implemented in PET
- Applicable to other systems

Plan of the Lecture

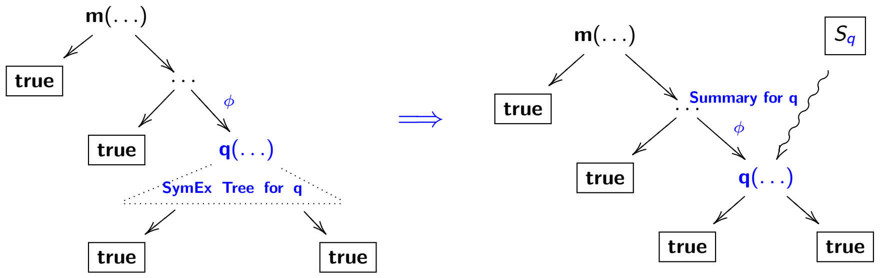
- Part 1: Symbolic execution and TCG
  - Introduction
  - Handling heap-manipulating programs
  - Compositionallity
- Part 2: CLP-based TCG
  - Introduction
  - Translation from imperative to CLP
  - Guided-TCG
  - Demo
- Part 3: TCG of Concurrent (Actor) Programs
  - The path explotion problem
  - Symbolic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

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

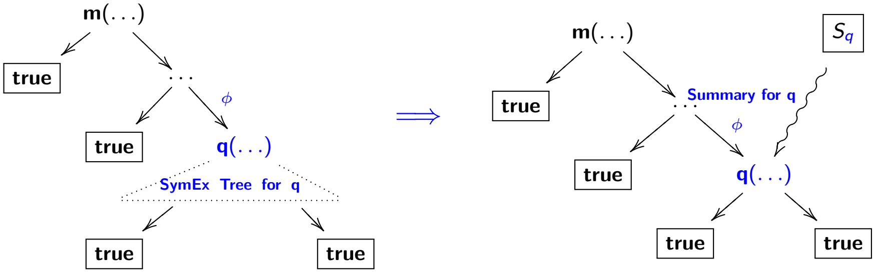

m(. . .)
true
. . .
true
ϕ
q(. . .)
SymEx Tree for q
true
true



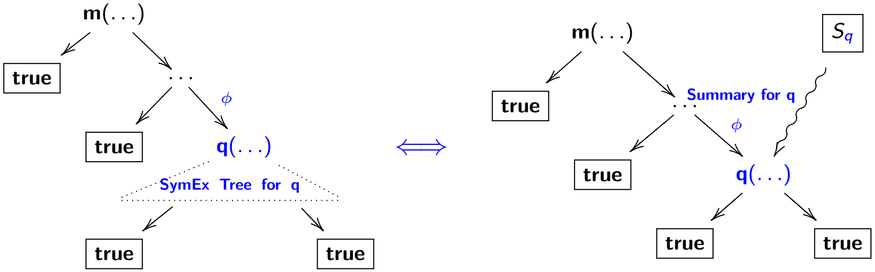
- Avoid inlining the symbolic execution tree of q



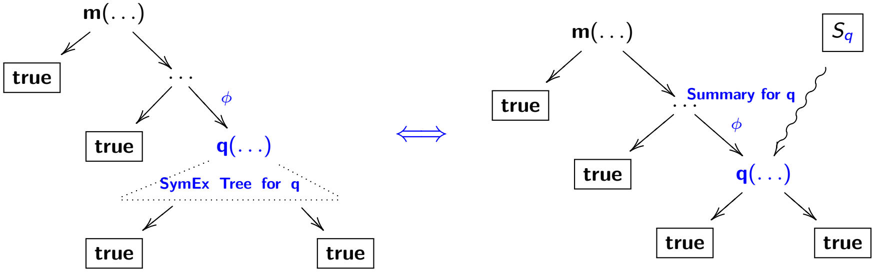
- Avoid inlining the symbolic execution tree of `q`
- Use method summary for `q`: Check compatibility with current state of `m` (Only compatible summary cases are composed)



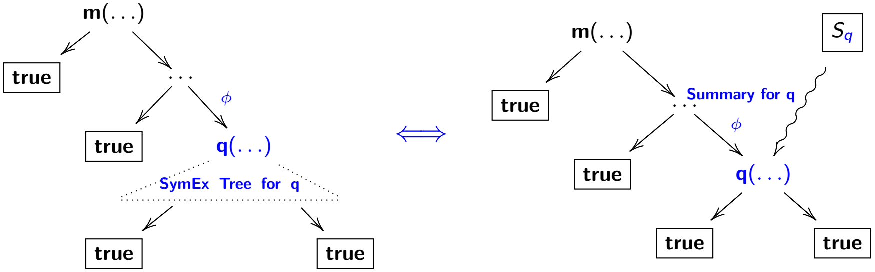
- Avoid inlining the symbolic execution tree of `q`
- Use method summary for `q`: Check compatibility with current state of `m` (Only compatible summary cases are composed)
- Incremental: summary for method `m` is created



- 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

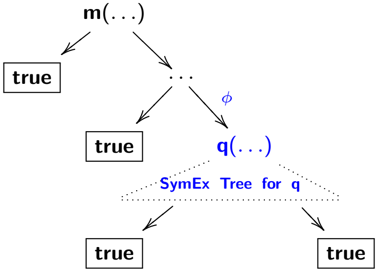


m(...)
true
...
true
ϕ
q(...)
SymEx Tree for q
true
true
⟺
m(...)
S_q
true
Summary for q
...
ϕ
true
q(...)
true
true

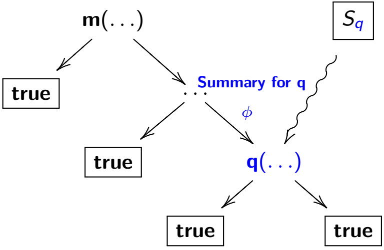


Context-sensitive

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



$\Longleftrightarrow$

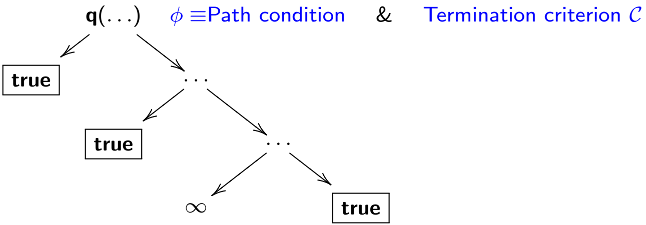


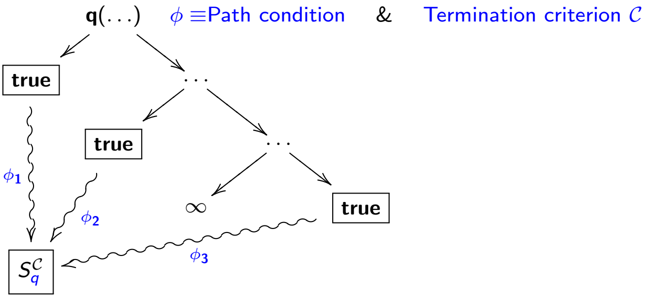
Context-sensitive

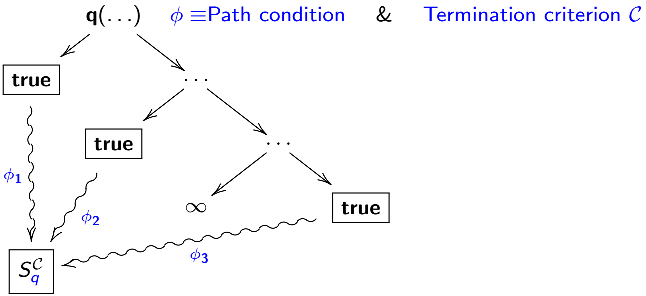
- Top-down traversal of call-graph
- Pro.: Only required information is computed
- Con.: Reusability of summaries is not always possible

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)


q(...)
ϕ ≡Path condition
&
Termination criterion $\mathcal{C}$
true
...
true
...
∞
true


q(...)
ϕ ≡Path condition & Termination criterion $\mathcal{C}$
true
...
true
...
∞
true
ϕ_1
ϕ_2
ϕ_3
$S_q^{\mathcal{C}}$

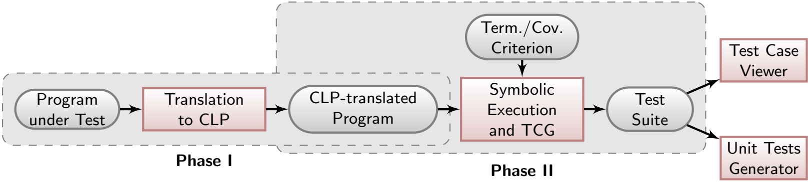


- A summary is a finite representation of the symbolic execution of a program for a given termination criterion, i.e., $S_q^{\mathcal{C}} \equiv \mathcal{T}_q^{\mathcal{C}}$
- Each element in a summary corresponds to a symbolic execution path (test case)
- Complete for a given coverage criterion, but partial in general

Plan of the Lecture

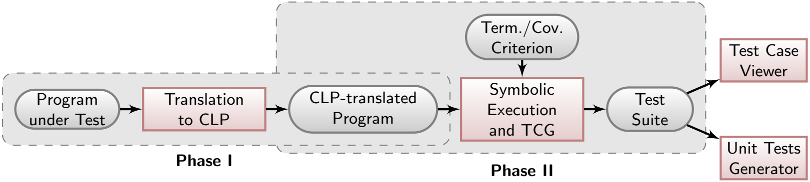
- Part 1: Symbolic execution and TCG
  - Introduction
  - Handling heap-manipulating programs
  - Compositionallity
- Part 2: CLP-based TCG
  - Introduction
  - Translation from imperative to CLP
  - Guided-TCG
  - Demo
- Part 3: TCG of Concurrent (Actor) Programs
  - The path explotion problem
  - Symbolic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

CLP-based Symbolic Execution and TCG



- Translation of the source language to CLP

CLP-based Symbolic Execution and TCG



- Translation of the source language to CLP

Java Code

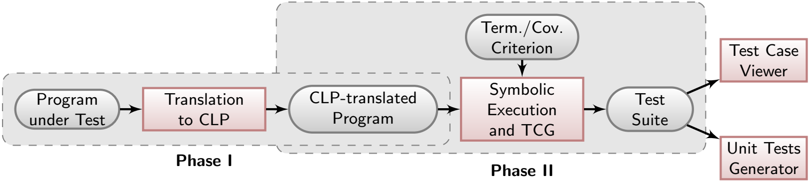
```
int abs(int x){
  if (x >= 0) return x;
  else return -x; }
```

CLP-translated

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

- Bounded symbolic execution of the CLP-translated program

CLP-based Symbolic Execution and TCG



- Translation of the source language to CLP

Java Code

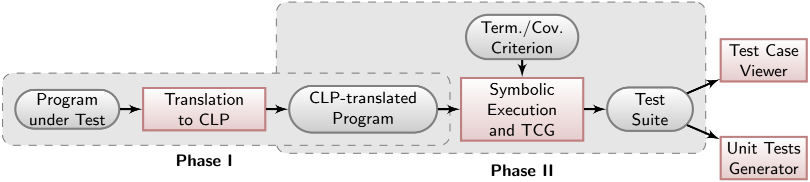
```
int abs(int x){
  if (x >= 0) return x;
  else return -x; }
```

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

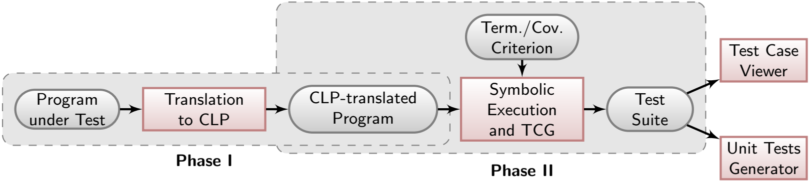
```
int abs(int x){
  if (x >= 0) return x;
  else return -x; }
```

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

CLP-based Symbolic Execution and TCG



- Translation of the source language to CLP

Java Code
```
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

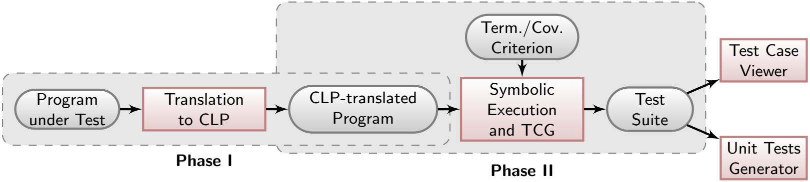
{ ⟨X >= 0,Z = X⟩,
⟨X < 0,Z = -X⟩ }

Concrete inputs

{ ⟨ X = 1, Z = 1⟩,
⟨ X = -1, Z = 1⟩ }

JUnit code
```
void test_abs(){
  assertEquals(abs(1),1);
  assertEquals(abs(-1),1);}
```

CLP-based Symbolic Execution and TCG



- Translation of the source language to CLP

Java Code
```
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
```
void test_abs(){
  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 $\mathcal{C}$ be a termination criterion.

Let M be a method, m be its corresponding predicate from its CLP-translated program P, and $\mathcal{C}$ 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 $\theta = \{\}$.

Let M be a method, m be its corresponding predicate from its CLP-translated program P, and $\mathcal{C}$ 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 $\theta = \{\}$.
- $\mathcal{T}_m^{\mathcal{C}}$ is the finite, possibly incomplete version of $\mathcal{T}_m$ bounded by $\mathcal{C}$.

Let M be a method, m be its corresponding predicate from its CLP-translated program P, and $\mathcal{C}$ 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 $\theta = \{\}$.
- $\mathcal{T}_m^{\mathcal{C}}$ is the finite, possibly incomplete version of $\mathcal{T}_m$ bounded by $\mathcal{C}$.
- A test case for m wrt $\mathcal{C}$ is $\langle\sigma(In), \sigma(Out), \sigma(H_{in}), \sigma(H_{out}), \sigma(EF), \theta'\rangle$ where σ and θ' are, resp., the substitution and the constraint store associated to a successful (terminating) path in $\mathcal{T}_m^{\mathcal{C}}$.

Let M be a method, m be its corresponding predicate from its CLP-translated program P, and $\mathcal{C}$ 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 $\theta = \{\}$.
- $\mathcal{T}_m^{\mathcal{C}}$ is the finite, possibly incomplete version of $\mathcal{T}_m$ bounded by $\mathcal{C}$.
- A test case for m wrt $\mathcal{C}$ is $\langle\sigma(In), \sigma(Out), \sigma(H_{in}), \sigma(H_{out}), \sigma(EF), \theta'\rangle$ where σ and θ' are, resp., the substitution and the constraint store associated to a successful (terminating) path in $\mathcal{T}_m^{\mathcal{C}}$.
- TCG is the process of generating the set of test cases for all successful (terminating) paths in $\mathcal{T}_m^{\mathcal{C}}$.

Java source code

```
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-based Symbolic Execution and TCG

Concrete example

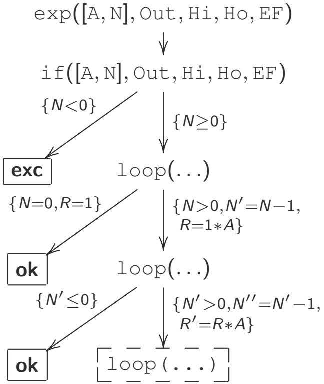
Java source code

```
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

```
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).
```

Symb. Ex. Tree



CLP-translated program

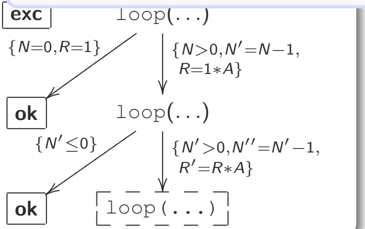
```
exp([A,N],Out,Hi,Ho,EF) :-
    if([A,N],Out,Hi,Ho,EF).
if([A,N],_Out,Hi,Ho,exc(X)):-
    N < 0,
    new_object(Hi,'Exc',X,Ho).
if([A,N],Out,H,H,ok) :-
    N >= 0,
    loop([A,N,1],Out).
loop([_A,N,R],R) :-
    N <= 0.
loop([A,N,R],Out) :-
    N > 0,
    R' = R*A,
    N' = N-1,
    loop(A,N',R',Out).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

CLP-based Symbolic Execution and TCG
Concrete example

Sy

Test cases

#	Input	Output	Path condition
1	[A, N]	[exception]	{N<0}
2	[A, N]	1	{N=0}
3	[A, N]	R	{N>0,N′=N−1,R=1∗A,N′<=0}




```
:-
   ).
    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).
```

Sy

Test cases

#	Input	Output	Path condition
1	[A, N]	[exception]	{N<0}
2	[A, N]	1	{N=0}
3	[A, N]	R	{N>0,N'=N-1,R=1*A,N'<=0}

:-

o).

ex

Concrete inputs

#	Input	Output
1	[-10, -10]	[Exception]
2	[-10, 0]	1
3	[-10, 1]	-10

```
    N >= 0,
    loop([A,N,1],Out).
oop([_A,N,R],R) :-
    N <= 0.
oop([A,N,R],Out) :-
    N > 0,
    R' = R*A,
    N' = N-1,
    loop(A,N',R',Out).
```

Sy

Test cases

#	Input
1	[A, N] [ex
2	[A, N]
3	[A, N]

ex

Concrete input

#	Input
1	[-10, -10]
2	[-10, 0]
3	[-10, 1]

Unit tests (JUnit)

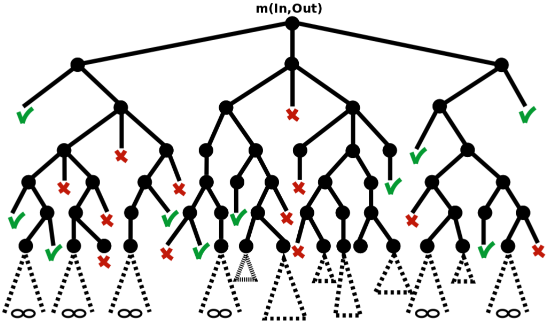
```
public void test_1(){
  int input0 = -10, input1 = -10;
  try{ int output = Test.intExp(input0,input1); }
  catch(Exception ex){
   assertEquals("exception","ArithmeticException",
                        ex.getClass().getName());
   return;
  }
  fail("Fail");
}
public void test_2(){
  int input0 = -10, input1 = 0;
  int output = Test.intExp(input0,input1);
  int expected = 1;
  assertEquals("OK",expected,output);
}
public void test_3(){
  int input0 = -10, input1 = 1;
  int output = Test.intExp(input0,input1);
  int expected = -10;
  assertEquals("OK",expected,output);
}
```

).

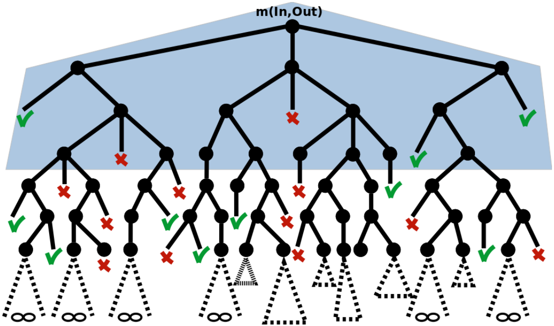
Plan of the Lecture

- 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 explotion problem
  - Symbolic execution and TCG for actors
  - Demo

TCG = Symbolic exec. +

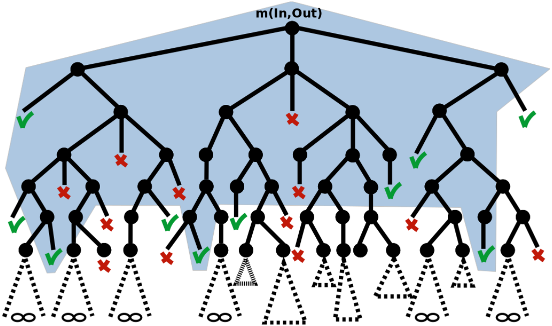


TCG = Symbolic exec. + termination criterion + constraint solving



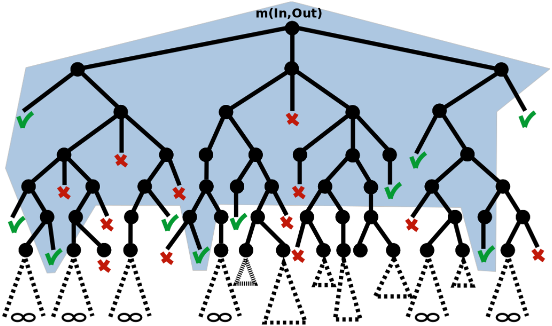
- Termination criteria: depth-k,

TCG = Symbolic exec. + termination criterion + constraint solving



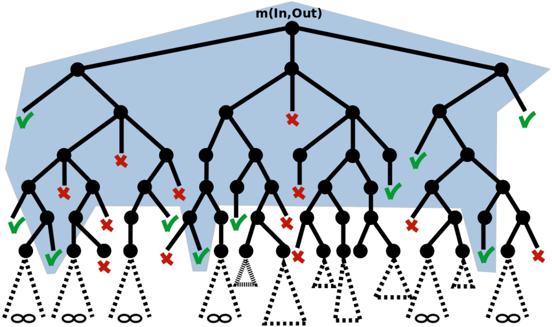
- Termination criteria: depth-k, loop-k

TCG = Symbolic exec. + termination criterion + constraint solving

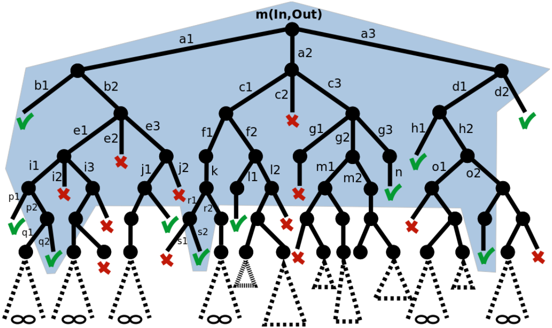


- Termination criteria: depth-k, loop-k
- Selection criteria: specific program point(s) (specific exception(s)), all local paths, worst memory consumption (within a loop-k limit), ...

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

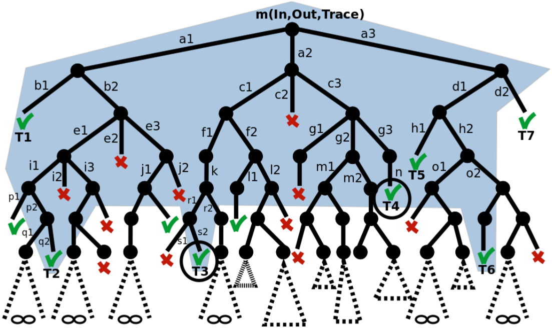


Selective TCG (naive) = TCG + filtering of test cases



- Paths in the symbolic execution tree can be labeled

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

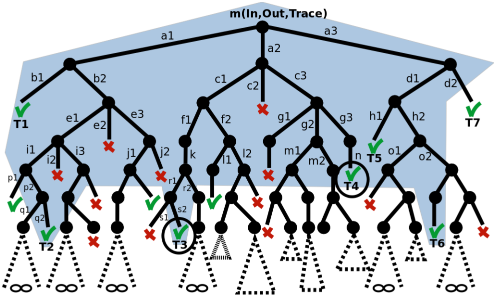


- Paths in the symbolic execution tree can be labeled
- Filtering is done by looking at the traces associated to the test cases

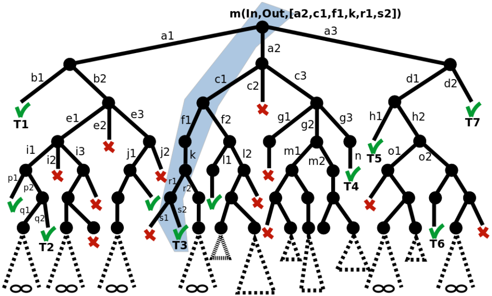
Guided Test Case Generation

Intuition

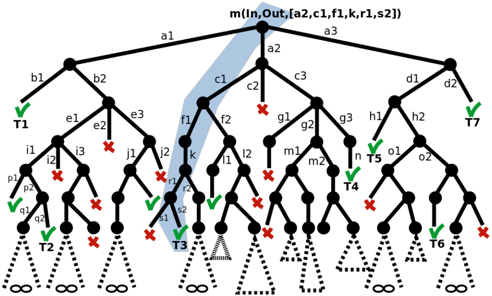
- Challenge: Avoid the generation of non-interesting paths
- Idea: Use the trace argument as an input to guide symbolic exec.



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



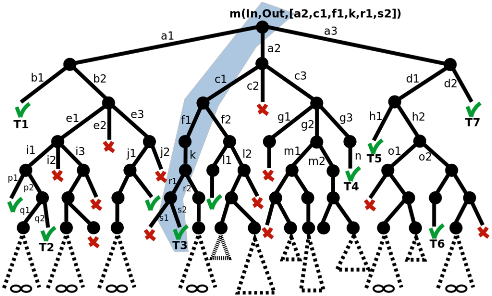
- Challenge: Avoid the generation of non-interesting paths
- Idea: Use the trace argument as an input to guide symbolic exec.



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

Guided Test Case Generation
Intuition

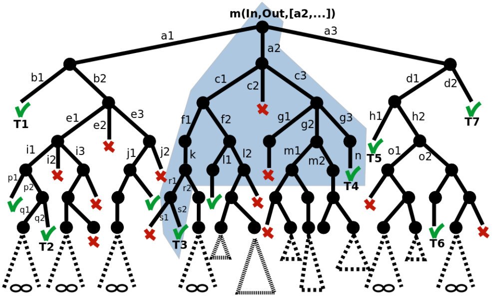
- Challenge: Avoid the generation of non-interesting paths
- Idea: Use the trace argument as an input to guide symbolic exec.



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

- Traces can be complete

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

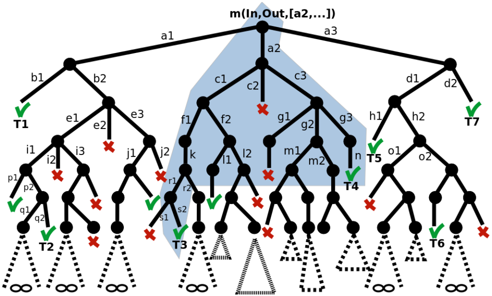


Guided TCG = Traces generator + guided symb. execs. + constr. solving

- 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*
TestCases = {}
while *TraceGen* has more traces and *TestCases* doesn't satisfy *SC*
  Ask *TraceGen* to generate a new trace in *Trace*
  TestCases = *TestCases* ∪ {first of guidedSymbExec(*M*,*TC*,*Trace*)}
Output: *TestCases*

Conclusions & References (Parts 1 and 2)

1. Symbolic execution consists in executing the program with symbolic (constraint) variables
2. Test cases are extracted from successful branches of the symbolic execution tree
3. The main challenges are related to scalability:
   - heap-manipulating programs [ICLP'10,ICLP'13]
   - compositionallity [LOPSTR'09]
4. CLP-based instance:
   - Symbolic execution almost for free [LOPSTR'08]
   - Language-independent approach (same TCG engine)
   - Guided TCG [LOPSTR'11,LOPSTR'12]
5. 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

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)

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 $\Rightarrow$ 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 $\Leftrightarrow$ concurrency unit
  2. No shared memory $\Rightarrow$ Information exchange by means of messages/asynchronous-method-calls
  3. Task scheduling is cooperative

The Actor Model

Syntax of the Language

$$M \ ::= \ \textbf{void}\ m(\bar{T}\ \bar{x})\{s;\}$$
$$s \ ::= \ s\ ;\ s \mid x = e \mid x = \textbf{this}.f \mid \textbf{this}.f = y \mid \textbf{if}\ b\ \textbf{then}\ s\ \textbf{else}\ s \mid$$
$$\textbf{while}\ b\ \textbf{do}\ s \mid x = \textbf{new}\ C \mid x\ !\ m(\bar{z}) \mid \textbf{return}$$

- 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

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

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

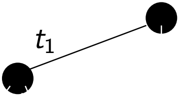
$$\{\overbrace{t_1, t_2}^{o_1}, \overbrace{\ t_3\ }^{o_2}\}$$

Partial Order Reduction

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

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

$$\{\overbrace{t_1, t_2}^{o_1}, \overbrace{t_3}^{o_2}\}$$

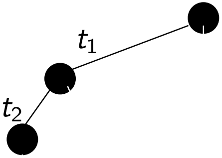


Partial Order Reduction

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

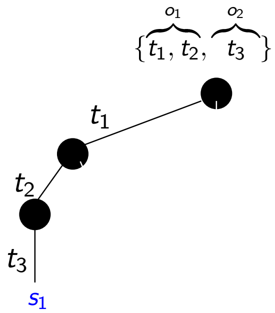
$h_{o_2} = \{this.g = 1\}$ $\quad$ $t_3 \mapsto this.g = 9$

$$\{\overbrace{t_1, t_2}^{o_1}, \overbrace{t_3}^{o_2}\}$$



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

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



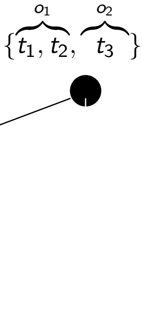
$this.g = 9$
$this.f = 7$

s_1

Partial Order Reduction

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

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



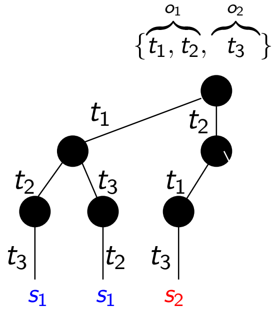
$this.g = 9$
$this.f = 7$

s_1

Partial Order Reduction

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

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



$this.g = 9$
$this.f = 7$

s_1

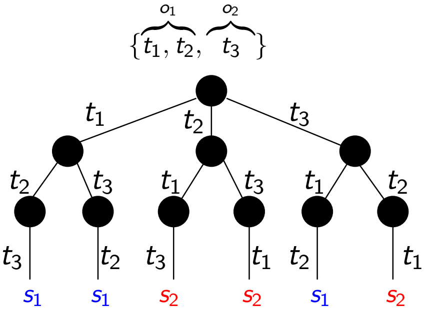
$this.g = 9$
$this.f = 5$

s_2

Partial Order Reduction

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

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



$this.g = 9$
$this.f = 7$

s_1

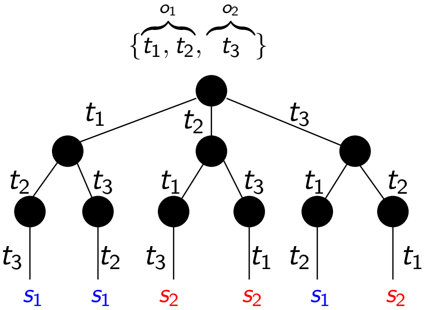
$this.g = 9$
$this.f = 5$

s_2

Partial Order Reduction

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

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



$this.g = 9$
$this.f = 7$

s_1

$this.g = 9$
$this.f = 5$

s_2

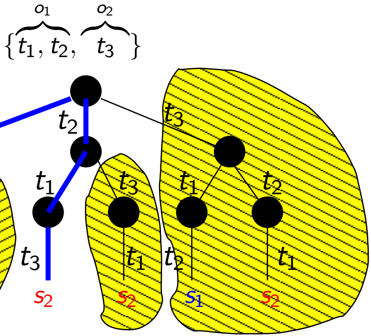
order in $o_1 : t_1 < t_2 \quad t_2 < t_1$

o_1, o_2 *are temporarily stable*

Partial Order Reduction

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

$h_{o_2} = \{this.g = 1\}$ $t_3 \mapsto this.g = 9$



$this.g = 9$
$this.f = 7$

s_1

$this.g = 9$
$this.f = 5$

s_2

order in $o_1 : t_1 < t_2 \quad t_2 < t_1$

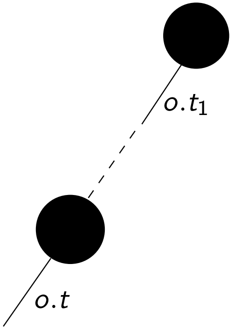
o_1, o_2 *are temporarily stable*

How to avoid redundant exploration in testing?

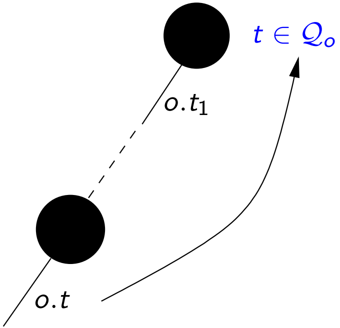
TransDPOR [Tasharofi et al FMOODS/FORTE 2012]

- 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 **backtraking** on $o'.t'$ in node n'. Otherwise repeat the process with $o'.t'$ upwards.

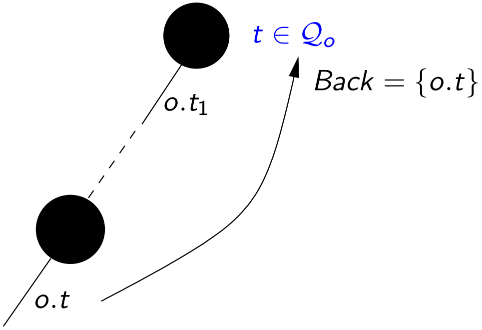
How to avoid redundant exploration in testing?



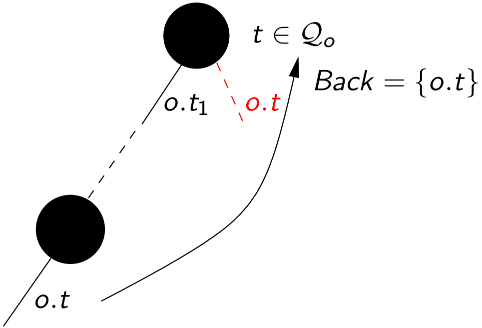
How to avoid redundant exploration in testing?



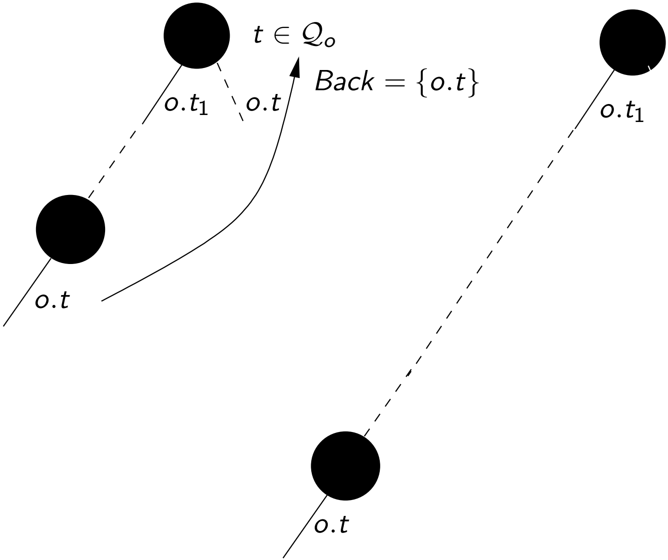
How to avoid redundant exploration in testing?



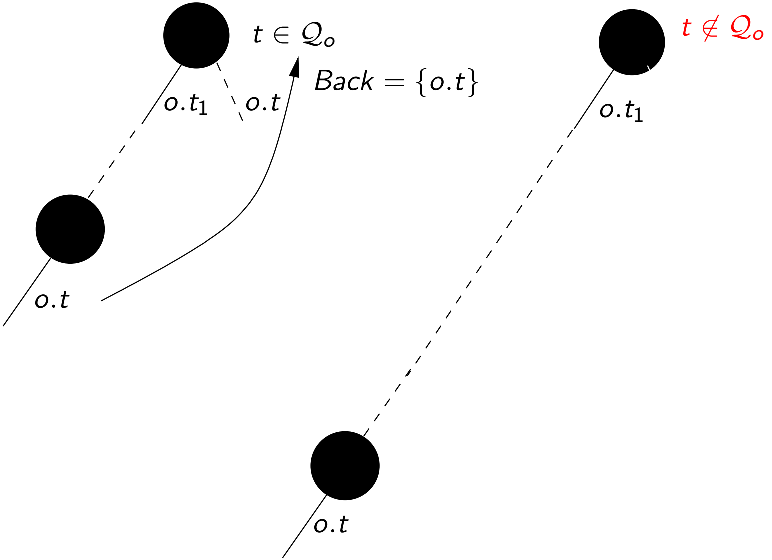
How to avoid redundant exploration in testing?



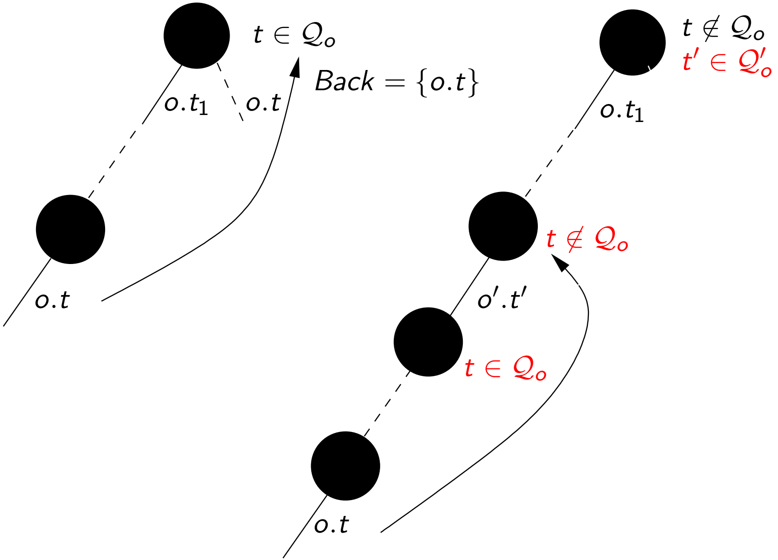
How to avoid redundant exploration in testing?



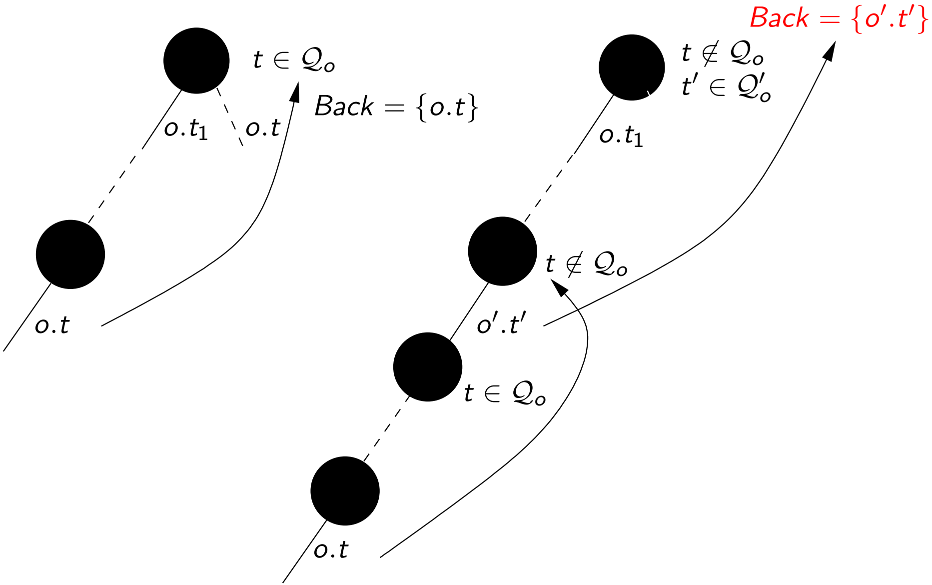
How to avoid redundant exploration in testing?



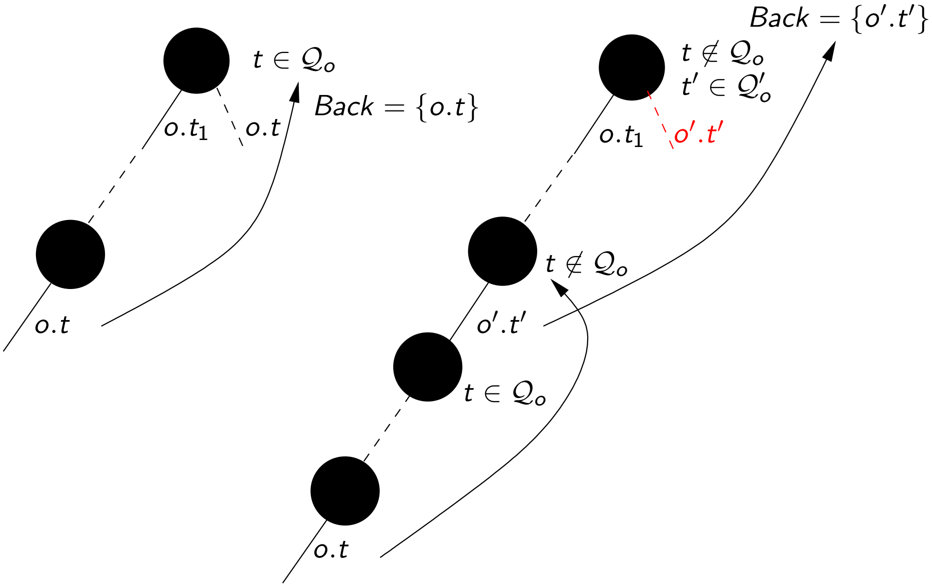
How to avoid redundant exploration in testing?



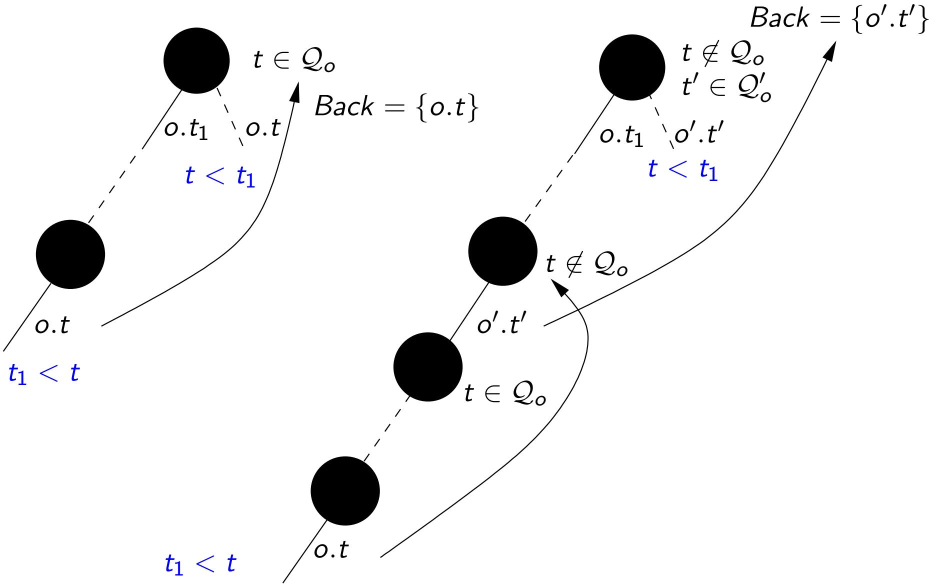
How to avoid redundant exploration in testing?



How to avoid redundant exploration in testing?

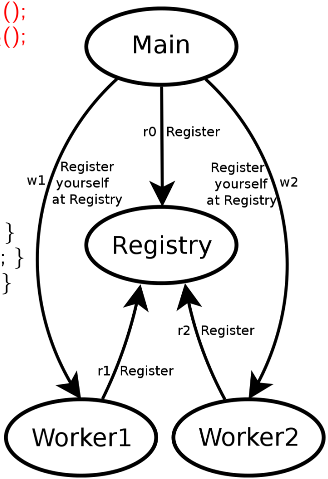


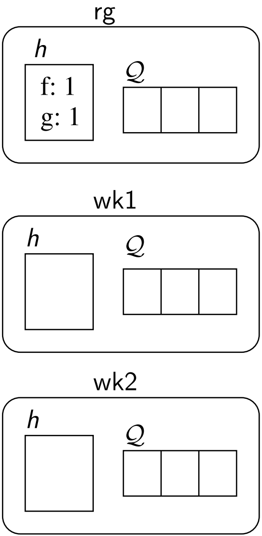
How to avoid redundant exploration in testing?



Working Example. Actors in Action

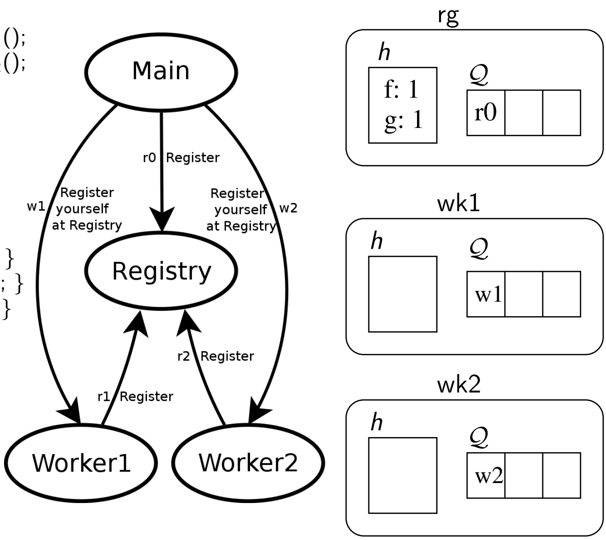
```
{
  Reg rg = new Reg();
  Worker1 wk1 = new Worker1();
  Worker2 wk2 = new Worker2();
  rg!r0();
  wk1!w1(rg);
  wk2!w2(rg);
}
class Reg {
  int f=1;  int g=1;

  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; }
}
```





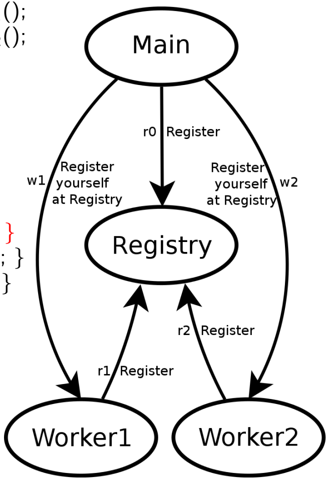
Working Example. Actors in Action

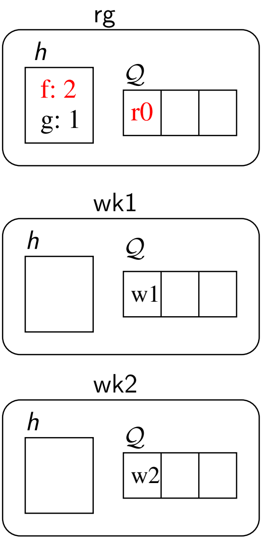
```
{
   Reg rg = new Reg();
   Worker1 wk1 = new Worker1();
   Worker2 wk2 = new Worker2();
   rg!r0();
   wk1!w1(rg);
   wk2!w2(rg);
}
class Reg {
   int f=1;  int g=1;
   void r0() { this.f++; return; }
   void r1() { this.g*=2; return; }
   void r2() { this.g++;return; }
}
class Worker1 {
   void w1(Reg rg) {
      rg!r1(); return; }
}
class Worker2 {
   void w2(Reg rg) {
      rg!r2(); return; }
}
```



Working Example. Actors in Action

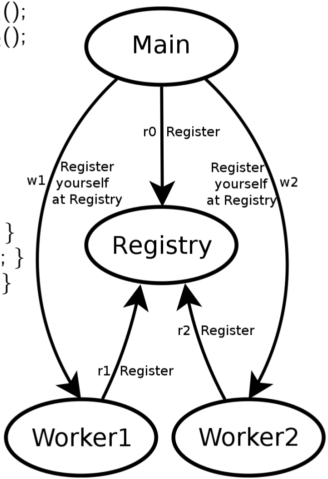
```
{
   Reg rg = new Reg();
   Worker1 wk1 = new Worker1();
   Worker2 wk2 = new Worker2();
   rg!r0();
   wk1!w1(rg);
   wk2!w2(rg);
}
class Reg {
   int f=1;  int g=1;
   void r0() { this.f++; return; }
   void r1() { this.g*=2; return; }
   void r2() { this.g++;return; }
 }
class Worker1 {
   void w1(Reg rg) {
      rg!r1(); return; }
}
class Worker2 {
   void w2(Reg rg) {
      rg!r2(); return; }
}
```

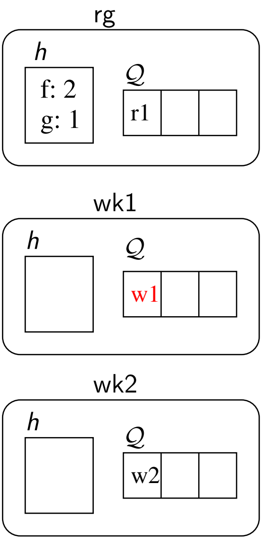




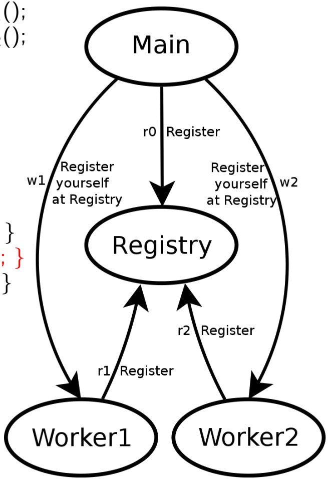
Working Example. Actors in Action

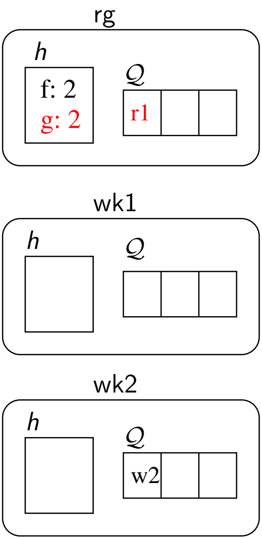
```
{
   Reg rg = new Reg();
   Worker1 wk1 = new Worker1();
   Worker2 wk2 = new Worker2();
   rg!r0();
   wk1!w1(rg);
   wk2!w2(rg);
}
class Reg {
   int f=1;  int g=1;

   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; }
}
```





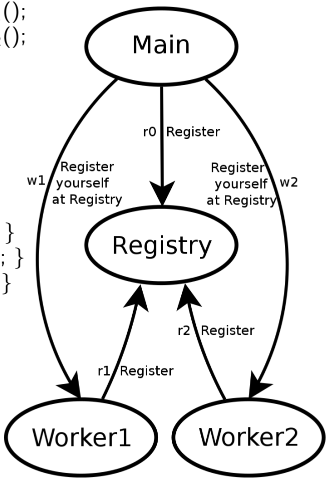
```
{
  Reg rg = new Reg();
  Worker1 wk1 = new Worker1();
  Worker2 wk2 = new Worker2();
  rg!r0();
  wk1!w1(rg);
  wk2!w2(rg);
}
class Reg {
  int f=1;  int g=1;
  void r0() { this.f++; return; }
  void r1() { this.g*=2; return; }
  void r2() { this.g++;return; }
}
class Worker1 {
  void w1(Reg rg) {
    rg!r1(); return; }
}
class Worker2 {
  void w2(Reg rg) {
    rg!r2(); return; }
}
```

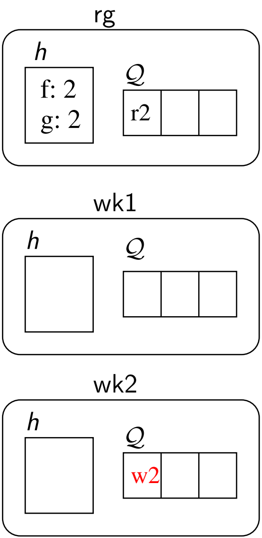




Working Example. Actors in Action

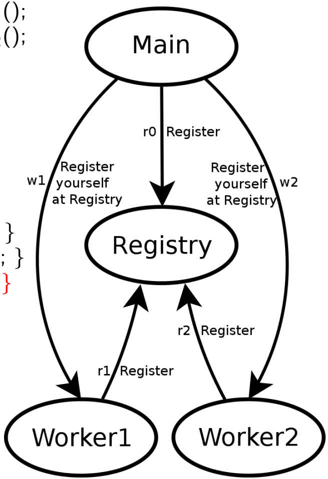
```
{
  Reg rg = new Reg();
  Worker₁ wk1 = new Worker₁();
  Worker₂ wk2 = new Worker₂();
  rg!r0();
  wk1!w1(rg);
  wk2!w2(rg);
}
class Reg {
  int f=1;  int g=1;

  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; }
}
```

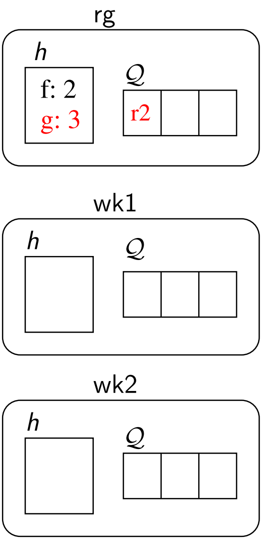




Working Example. Actors in Action

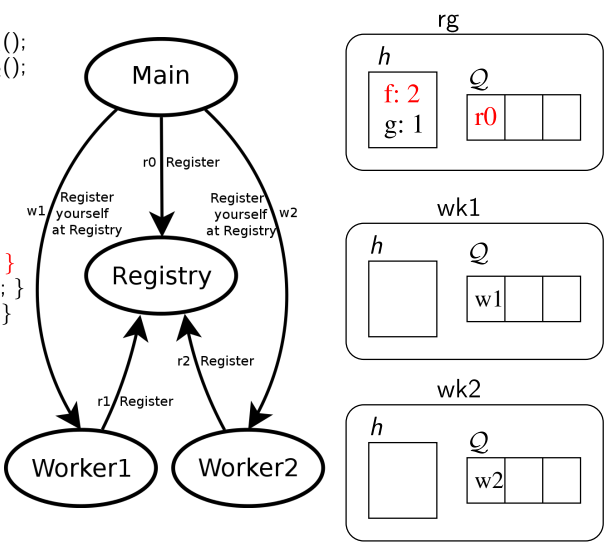
```
{
  Reg rg = new Reg();
  Worker1 wk1 = new Worker1();
  Worker2 wk2 = new Worker2();
  rg!r0();
  wk1!w1(rg);
  wk2!w2(rg);
}
class Reg {
  int f=1;  int g=1;
  void r0() { this.f++; return; }
  void r1() { this.g*=2; return; }
  void r2() { this.g++;return; }
}
class Worker1 {
  void w1(Reg rg) {
    rg!r1(); return; }
}
class Worker2 {
  void w2(Reg rg) {
    rg!r2(); return; }
}
```





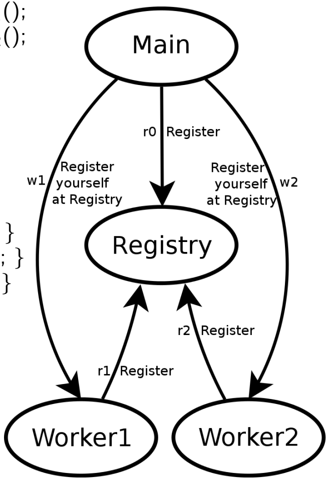
Working Example. Actors in Action

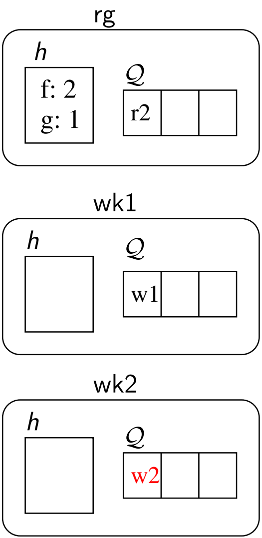
```
{
  Reg rg = new Reg();
  Worker1 wk1 = new Worker1();
  Worker2 wk2 = new Worker2();
  rg!r0();
  wk1!w1(rg);
  wk2!w2(rg);
}
class Reg {
  int f=1;  int g=1;
  void r0() { this.f++; return; }
  void r1() { this.g*=2; return; }
  void r2() { this.g++;return; }
 }
class Worker1 {
  void w1(Reg rg) {
    rg!r1(); return; }
}
class Worker2 {
  void w2(Reg rg) {
    rg!r2(); return; }
}
```



Working Example. Actors in Action

```
{
  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; }
}
```





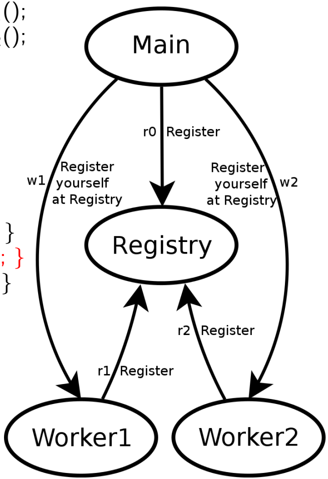
Working Example. Actors in Action

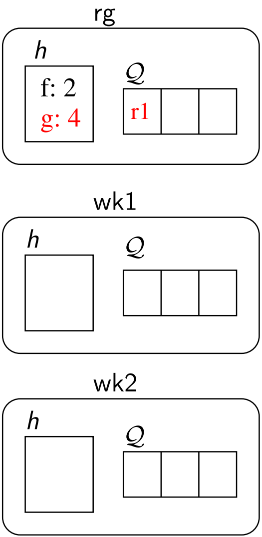
{
  Reg rg = **new** Reg();
  Worker_1 wk1 = **new** Worker_1();
  Worker_2 wk2 = **new** Worker_2();
  rg!r0();
  wk1!w1(rg);
  wk2!w2(rg);
}

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**; }
}





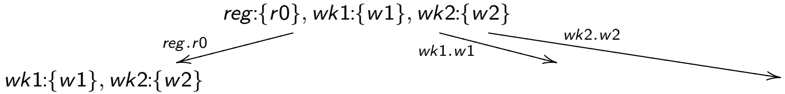
```
// main Block
Reg reg = new Reg;
Worker₁ wk1 = new Worker₁();
Worker₂ wk2 = new Worker₂();
reg!r0();
wk1!w1(reg); wk2!w2(reg);
```

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

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

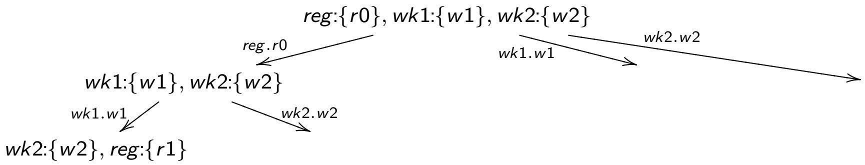
```
class Reg {
  int f=1;  int g=1;
  void r0() {this.f++; return;}
  void r1() {this.g*=2; return;}
  void r2() {this.g++; return;}
}
```

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



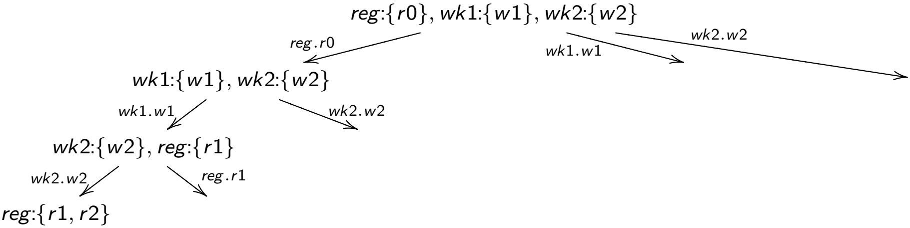
- **Macro-step semantics** $\Rightarrow$ Interleavings only at return points

```
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

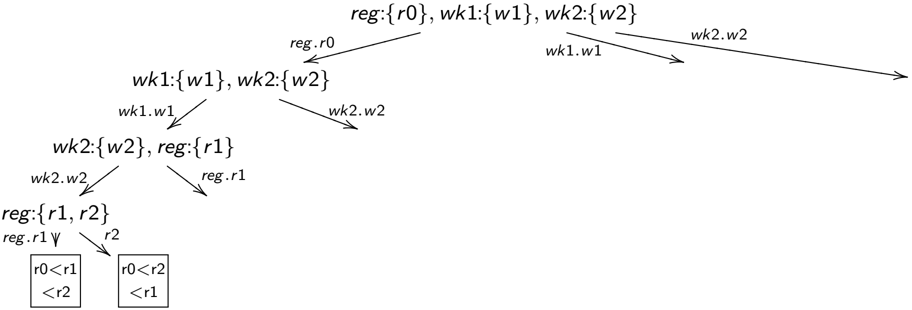
```
class Reg {
  int f=1;  int g=1;
  void r0() {this.f++; return;}
  void r1() {this.g*=2; return;}
  void r2() {this.g++; return;}
}

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

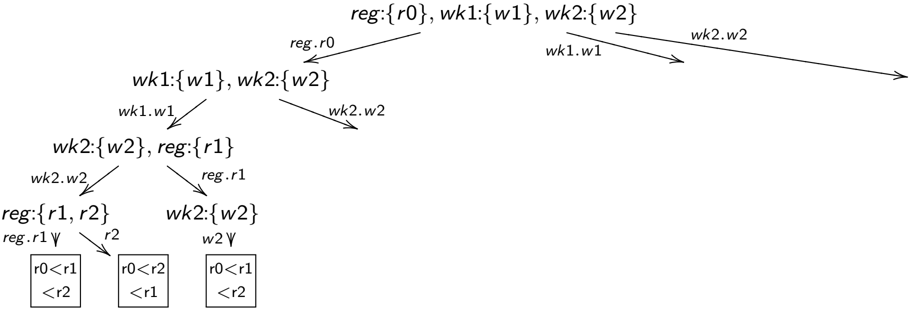
```
class Reg {
  int f=1;  int g=1;

  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

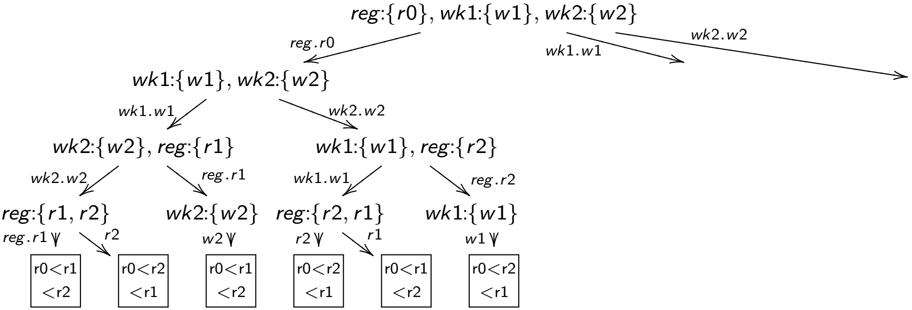
```
class Reg {
  int f=1;  int g=1;
  void r0() {this.f++; return;}
  void r1() {this.g*=2; return;}
  void r2() {this.g++; return;}
}

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

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

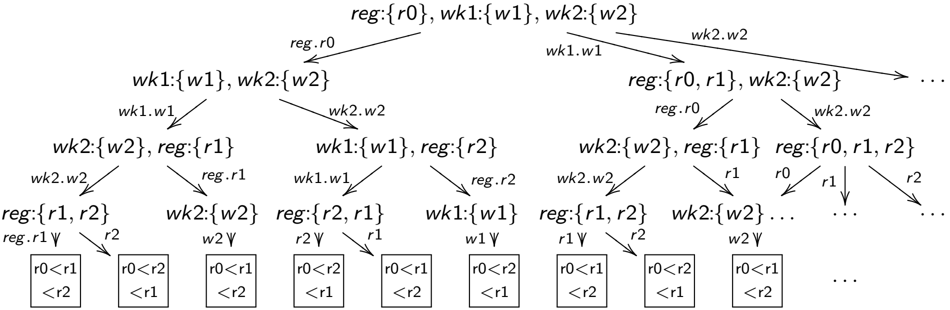
```
class Worker1 {
  void w1(Reg rg) {rg!r1(); return;}
}
class Worker2 {
  void w2(Reg rg) {rg!r2(); return;}
}
```



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

```
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
- 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;}
}
```

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

```
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;}
}
```

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

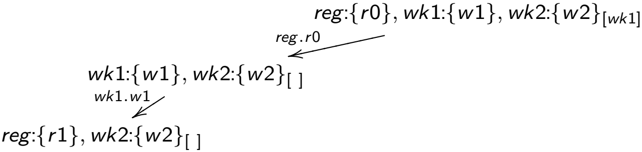
$\xrightarrow{reg.r0}$ $wk1{:}\{w1\}, wk2{:}\{w2\}_{[\,]}$

$\xrightarrow{wk1.w1}$ $reg{:}\{r1\}, wk2{:}\{w2\}_{[\,]}$

- 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;}
}
```

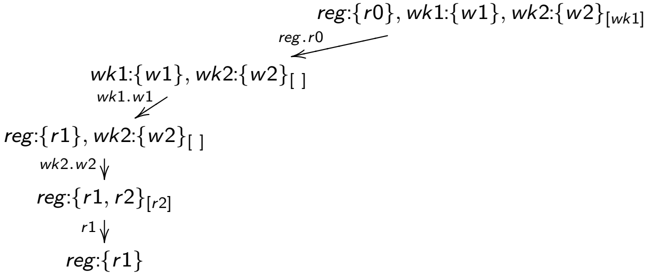
```
class Worker1 {
  void w1(Reg rg) {rg!r1(); return;}
}
class Worker2 {
  void w2(Reg rg) {rg!r2(); return;}
}
```



- Actor $wk1$ is added to the backtrack set of the root

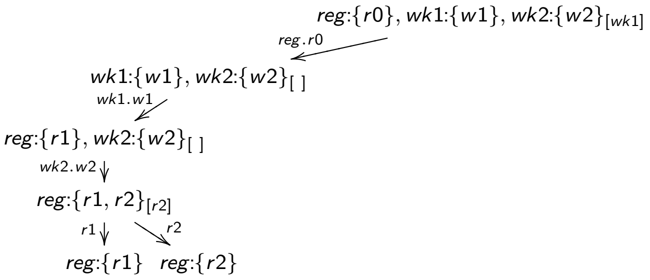
```
class Reg {
  int f=1;  int g=1;
  void r0() {this.f++; return;}
  void r1() {this.g*=2; return;}
  void r2() {this.g++; return;}
}
```

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



r0<r1<r2

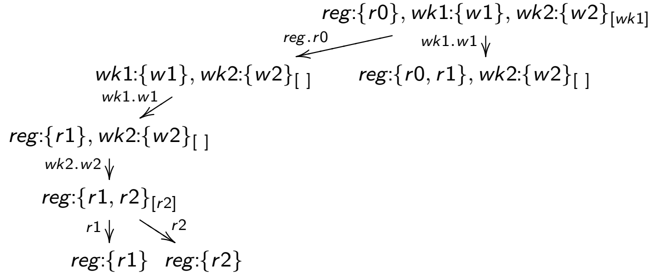
```
class Reg {
  int f=1;  int g=1;
  void r0() {this.f++; return;}
  void r1() {this.g*=2; return;}
  void r2() {this.g++; return;}
}

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



r0<r1<r2	r0<r2<r1

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

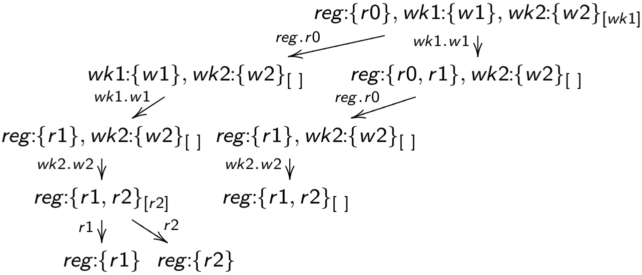
```
class Worker1 {
  void w1(Reg rg) {rg!r1(); return;}
}
class Worker2 {
  void w2(Reg rg) {rg!r2(); return;}
}
```



r0<r1<r2	r0<r2<r1

```
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;}
}
```

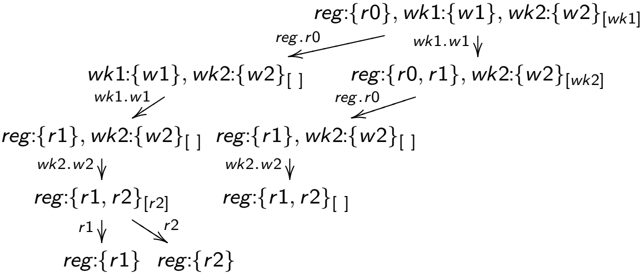


r0<r1<r2	r0<r2<r1

- To explore *r*2 before *r*0 actor *wk*2 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;}
}
```

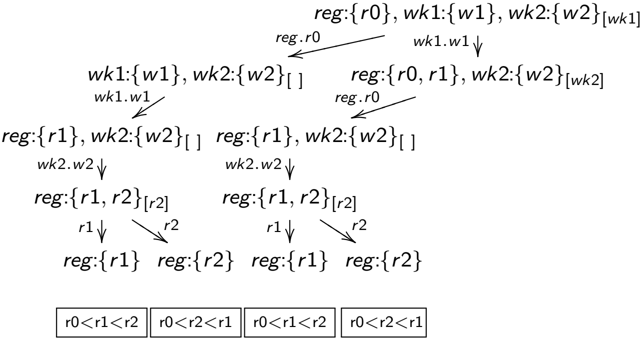


r0<r1<r2	r0<r2<r1

- Actor *wk2* is added to the backtrack set

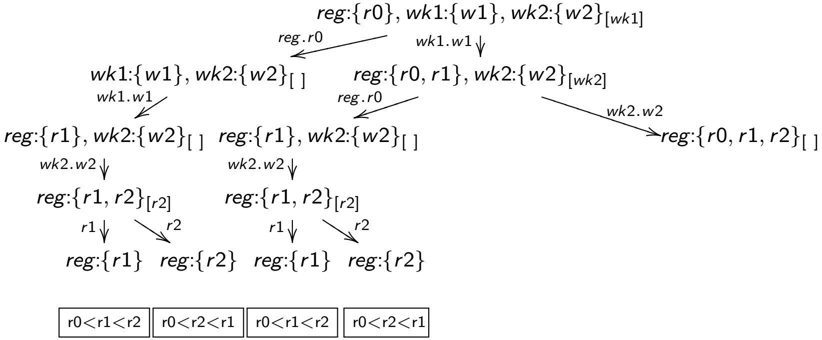
```
class Reg {
  int f=1;  int g=1;
  void r0() {this.f++; return;}
  void r1() {this.g*=2; return;}
  void r2() {this.g++; return;}
}
```

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

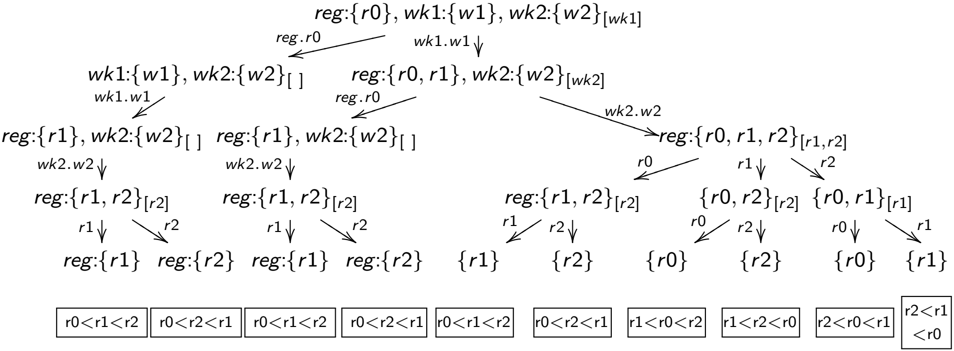


```
class Reg {
  int f=1; int g=1;
  void r0() {this.f++; return;}
  void r1() {this.g*=2; return;}
  void r2() {this.g++; return;}
}
```

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

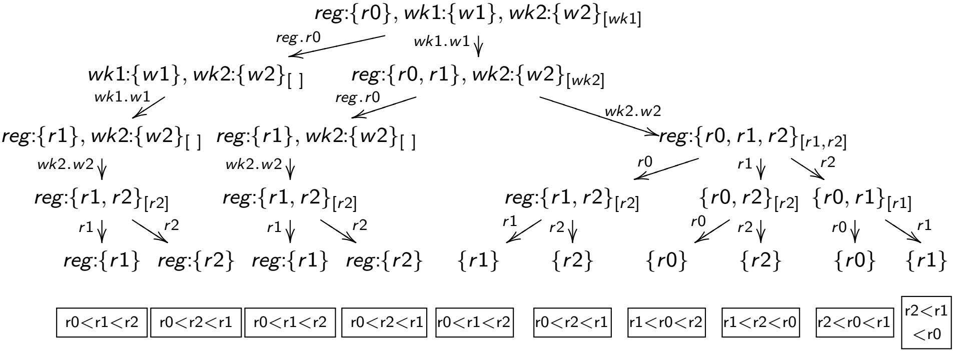


```
class Reg {
  int f=1;  int g=1;
  void r0() {this.f++; return;}
  void r1() {this.g*=2; return;}
  void r2() {this.g++; return;}
}

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

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



- TransDPOR reduces the exploration from 32 to 10 explorations

```
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

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)

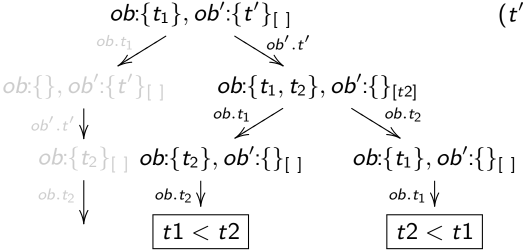
First Contribution: Actor Selection based on Stability Crit.

- 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:

$ob{:}\{t_1\}, ob'{:}\{t'\}_{[\,]}$ $\qquad$ (t' calls $ob.t_2$)

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:



(t' calls $ob.t_2$)

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

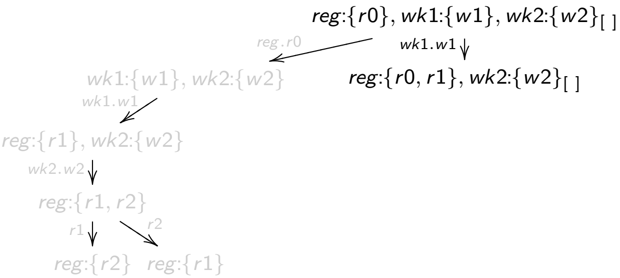
```
class Worker1 {
  void w1(Reg rg) {rg!r1(); return;}
}
class Worker2 {
  void w2(Reg rg) {rg!r2(); return;}
}
```

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

- Actor *reg* is not stable. *wk*1 and *wk*2 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;}
}
```

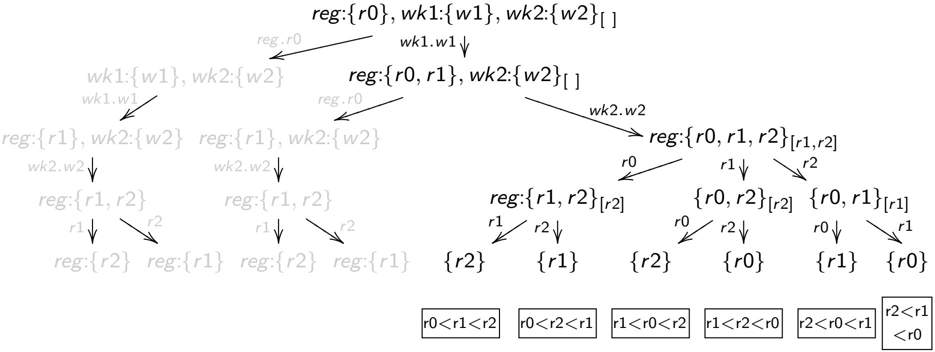
```
class Worker1 {
  void w1(Reg rg) {rg!r1(); return;}
}

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



- Actor *reg* is not stable. *wk2* is 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;}
}
```



- This reduces the exploration further, from 10 to 6 executions

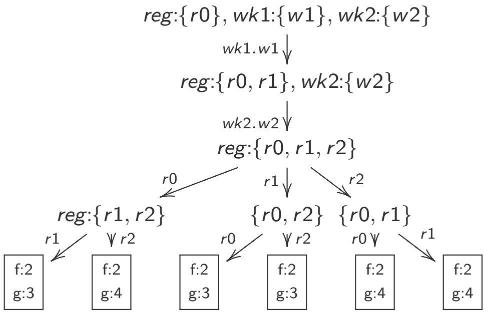
Experimental Results of Actor Selection

- 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

2nd Contrib.: Task Selection based on Dependency Info.

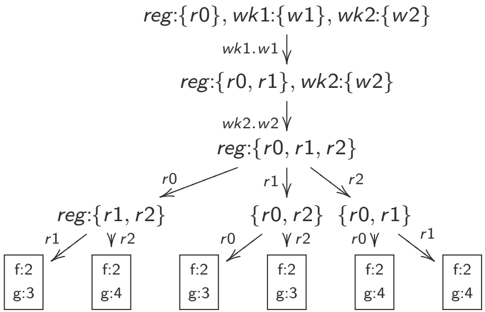
Motivation

- Observation: Execs. with different partial order lead to the same state




```
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




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

  void r0() {this.f++; return;}
  void r1() {this.g*=2; return;}
  void r2() {this.g++; return;}
}
```

- Execution of $r0$ is independent from that of $r1$ and $r2$

indep(t,t') ⇐ t does not write to fields that t' 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

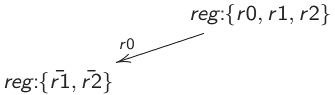
reg:$\{r0, r1, r2\}$

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

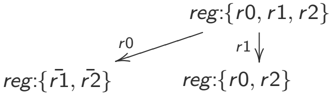


indep(r0,r1) and indep(r0,r2)

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

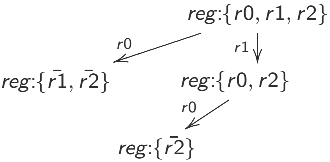


indep(r0,r1) and indep(r0,r2)

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

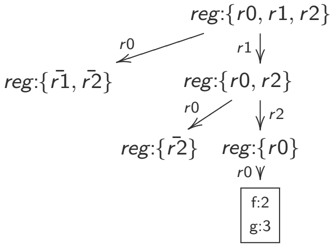


indep(r0,r1) and indep(r0,r2)

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

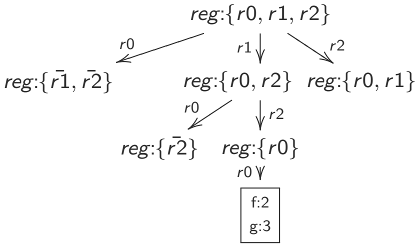
Algorithm in action



indep(r0,r1) and indep(r0,r2)

- 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

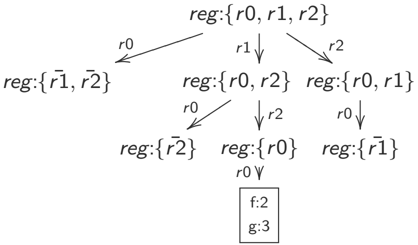


indep(r0,r1) and indep(r0,r2)

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

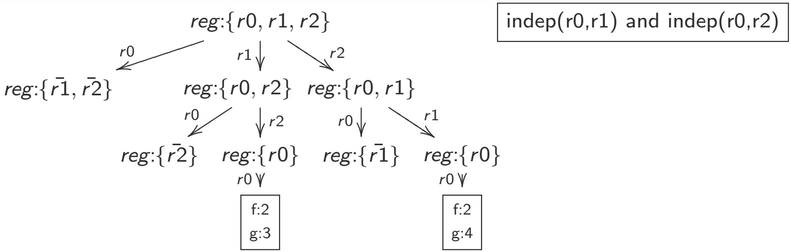


indep(r0,r1) and indep(r0,r2)

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

	No task sel. reduction				With task. sel. reduct.				Speedup	
Test name	Execs	Time	States	H	Execs	Time	States	H	Execs	Time
QSort.test1	4	2	18	3	4	2	18	3	1.0x	1.0x
QSort.test2	16	10	70	21	16	10	70	21	1.0x	1.0x
Fib.test1	4	3	18	3	4	3	18	3	1.0x	1.0x
Fib.test2	128	80	524	189	128	81	524	189	1.0x	1.0x
PSort.test1	288	69	1294	144	288	71	1294	144	1.0x	1.0x
PSort.test2	5760	1385	25829	2880	288	71	1304	144	20.0x	19.5x
RegSim.test1	10080	806	27415	0	720	136	3923	0	14.0x	5.9x
RegSim.test2	11520	864	31576	0	384	70	2132	0	30.0x	12.3x
DHT.test1	1152	137	3905	0	36	6	141	0	32.0x	22.8x
DHT.test2	480	97	2304	0	12	4	85	0	40.0x	24.2x
Mail.test1	2648	557	11377	0	460	120	2270	0	5.8x	4.6x
Mail.test2	1665500	>200s	5109783	0	27880	4064	94222	0	>60x	49.2x
BB.test1	155520	23907	475205	0	4320	681	13214	0	36.0x	35.1x
BB.test2	1099008	165114	3028298	0	45792	6945	126192	0	24.0x	23.8x

- Except for the first two benckmarks, 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

Symbolic Execution and TCG for Actor Models

Define a TCG framework for Actors:

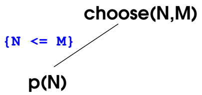
- Symbolic execution (previous part)
- Termination criteria
- Coverage criteria
- TCG with synchronization primitives (await and get)

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).

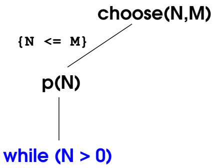
```
class A {
  int f = 1;
void choose(int n, int m) {
  if (n ≤ m) then this ! p(n);
  else this ! q(m);
}
void p(int n) {
  while (n > 0) {
    this.f = this.f * n;
    n = n - 1;
  }
}
...
```



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).

```
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;
  }
}
...
```

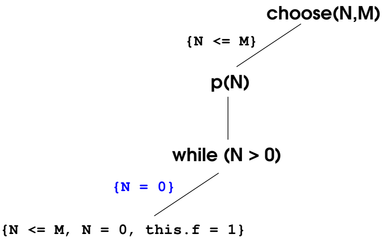


Symbolic Execution and TCG for Actor Models

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).

```
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;
  }
}
...
```

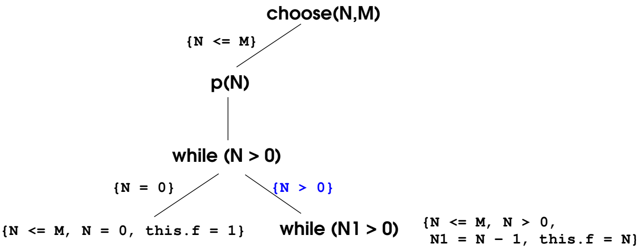


Symbolic Execution and TCG for Actor Models

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).

```
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;
  }
}
...
```

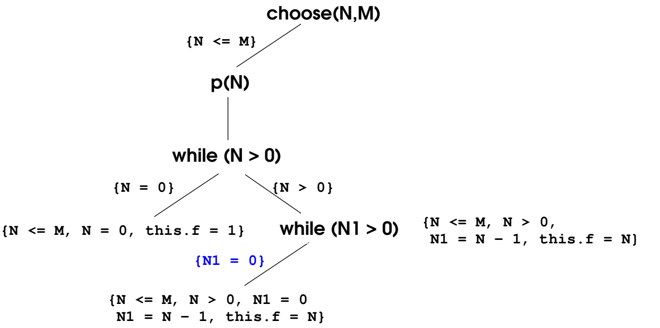


Symbolic Execution and TCG for Actor Models

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).

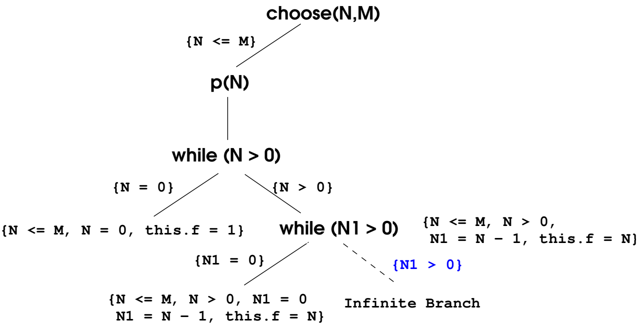
```
class A {
  int f = 1;
void choose(int n, int m) {
  if (n ≤ m) then this ! p(n);
  else this ! q(m);
}
void p(int n) {
  while (n > 0) {
    this.f = this.f * n;
    n = n - 1;
  }
}
...
```



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).

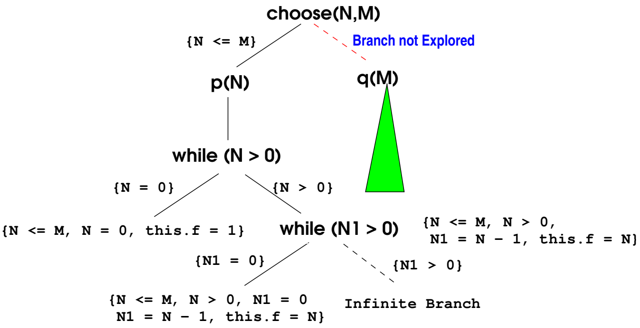
```
class A {
  int f = 1;
void choose(int n, int m) {
  if (n ≤ m) then this ! p(n);
  else this ! q(m);
}
void p(int n) {
  while (n > 0) {
    this.f = this.f * n;
    n = n - 1;
  }
}
...
```



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).

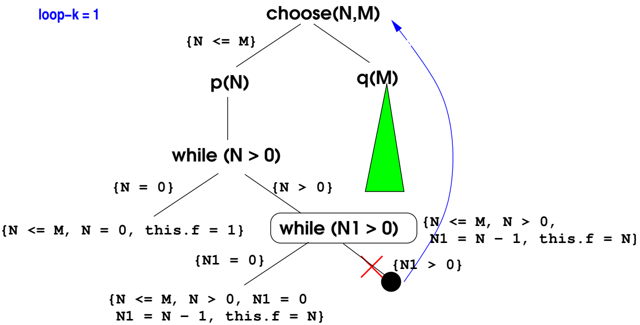
```
class A {
  int f = 1;
void choose(int n, int m) {
  if (n ≤ m) then this ! p(n);
  else this ! q(m);
}
void p(int n) {
  while (n > 0) {
    this.f = this.f * n;
    n = n - 1;
  }
}
...
```



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).

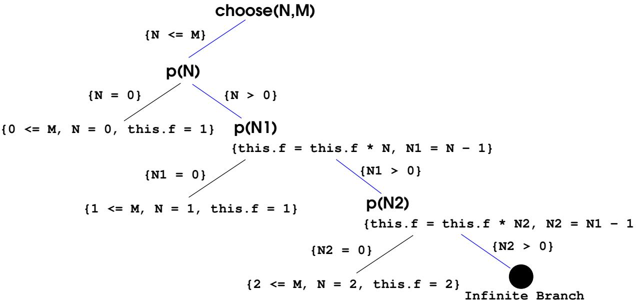
```
class A {
  int f = 1;
void choose(int n, int m) {
  if (n ≤ m) then this ! p(n);
  else this ! q(m);
}
void p(int n) {
  while (n > 0) {
    this.f = this.f * n;
    n = n - 1;
  }
}
...
```



Coverage and Termination Criteria for Concurrent Objects

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

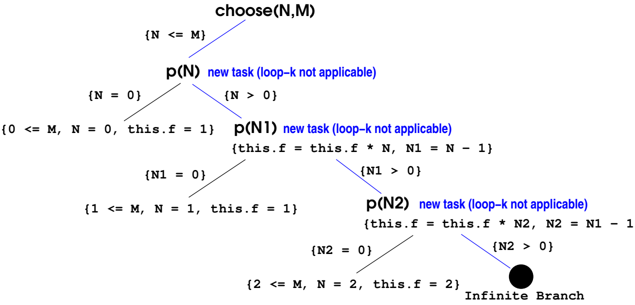
```
class A {
  int f = 1;
void choose(int n, int m) {
  if (n ≤ m) then this ! p(n);
  else this ! q(m);
}

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



Coverage and Termination Criteria for Concurrent Objects

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

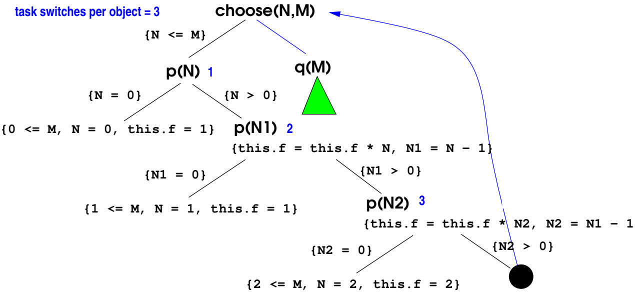
```
class A {
  int f = 1;
void choose(int n, int m) {
  if (n ≤ m) then this ! p(n);
  else this ! q(m);
}

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



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);
  }
}
...
```



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 ≤ 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);
  }
}

...
```

Coverage and Termination Criteria for Concurrent Objects

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

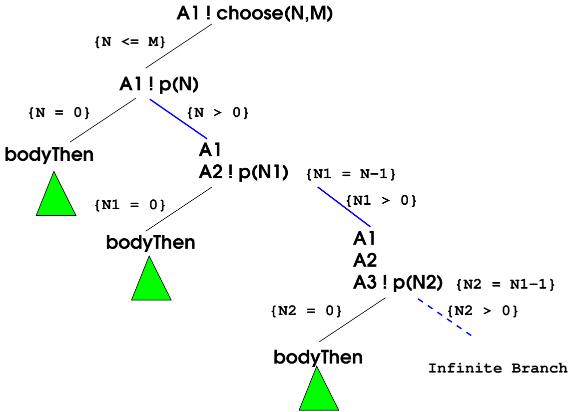
```
class A {
void choose(int n, int m) {
   if (n ≤ m) then this ! p(n);
   else this ! q(m);
}

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.

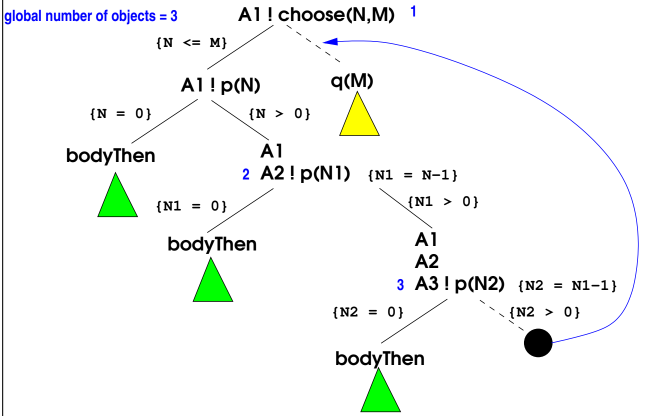
```
class A {
void choose(int n, int m) {
   if (n ≤ m) then this ! p(n);
   else this ! q(m);
}

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

...
```



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 ≤ 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);
  }
}

...
```



Synchronization Primitives for Concurrent Objects

await and **get** primitives

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.

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

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

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.

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.

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

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.

```
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.

```
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.

Local Trace

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

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 **related(m)**, which contains all methods whose interleaved execution with m can lead to a local execution not considered before.

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 **related**(**m**), which contains all methods whose interleaved execution with m can lead to a local execution not considered before.
- Initially **related**(**m**) will contains all methods of the class under test.
- Limit the size of the queue

Reducing the set **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

```
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); }
}
```

$\Rightarrow \mathbf{related}(\mathbf{p}) = \{\mathbf{setF}, \mathbf{setG}, \mathbf{set}\}$

Reducing the set **related(m)**

Pruning 2

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

Reducing the set **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); }
}
```

$\Rightarrow$ **related(p)** $=$ {**setF**, **setG**}

Reducing the set **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 **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); }
}
```

$\Rightarrow$ **related**(**p**) = {**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

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