C09 – Concolic execution & Model checking

Program Verification

FMI · Denisa Diaconescu · Spring 2022

Overview

Concolic execution

What is Model Checking?

Concolic execution

Concolic execution

$$\boxed{\text{concolic} = \text{concrete} + \text{symbolic}}$$

- Combines both symbolic execution and concrete (normal) execution.
- The basic idea is to have the concrete execution drive the symbolic execution.
- The programs run as usual (it needs to be given some input), but in addition it also maintains the usual symbolic information.

Concolic execution - Example

```
int twice(int v) {
  return 2 * v;
}

void test(int x, int y) {
  z = twice(y);
  if (x == z) {
    if (x > y + 10)
      ERROR;
  }
}

int main() {
  x = read();
  y = read();
  test(x,y);
}
```

The read() functions read a value from the input. Suppose we read x = 22 and y = 7.

We will keep both the concrete store and the symbolic store and path constraint.

σ : x ↦ 22,
y ↦ 7

σ_s : x ↦ x0,
y ↦ y0

pct : true

source: Lecture Notes on "Program Analysis", ETH Zurich, Martin Vechev.

Concolic execution - Example

```
int twice(int v) {
  return 2 * v;
}

void test(int x, int y) {
  z = twice(y);
  if (x == z) {
    if (x > y + 10)
      ERROR;
  }
}

int main() {
  x = read();
  y = read();
  test(x,y);
}
```

$\sigma : x \mapsto 22, \; y \mapsto 7, \; z \mapsto 14$

$\sigma_s : x \mapsto x0, \; y \mapsto y0, \; z \mapsto 2*y0$

pct : true

The concrete execution will now take the ‘else’ branch of z == x.

source: Lecture Notes on "Program Analysis", ETH Zurich, Martin Vechev.

Concolic execution - Example

```
int twice(int v) {
  return 2 * v;
}

void test(int x, int y) {
  z = twice(y);
  if (x == z) {
    if (x > y + 10)
      ERROR;
  }
}

int main() {
  x = read();
  y = read();
  test(x,y);
}
```

Hence, we get:

σ : x ↦ 22,
y ↦ 7,
z ↦ 14

σ_s : x ↦ x0,
y ↦ y0
z ↦ 2*y0

pct : x0 ≠ 2*y0

source: Lecture Notes on "Program Analysis", ETH Zurich, Martin Vechev.

Concolic execution - Example

```
int twice(int v) {
  return 2 * v;
}

void test(int x, int y) {
  z = twice(y);
  if (x == z) {
    if (x > y + 10)
      ERROR;
  }
}

int main() {
  x = read();
  y = read();
  test(x,y);
}
```

At this point, concolic execution decides that it would like the explore the "true" branch of x == z and hence it needs to generate concrete inputs in order to explore it. Towards such inputs, it negates the pct constraint, obtaining:

pct : `x0 = 2*y0`

It then calls the SMT solver to find a satisfying assignment of that constraint. Let us suppose the SMT solver returns:

$$x0 \mapsto 2, \; y0 \mapsto 1$$

The concolic execution then runs the program with this input.

source: Lecture Notes on "Program Analysis", ETH Zurich, Martin Vechev.

Concolic execution - Example

```
int twice(int v) {
  return 2 * v;
}

void test(int x, int y) {
  z = twice(y);
  if (x == z) {
    if (x > y + 10)
      ERROR;
  }
}

int main() {
  x = read();
  y = read();
  test(x,y);
}
```

With the input $x \mapsto 2,\ y \mapsto 1$ we reach this program point with the following information:

$\sigma : x \mapsto 2,\ y \mapsto 1,\ z \mapsto 2$

$\sigma_s : x \mapsto x0,\ y \mapsto y0,\ z \mapsto 2*y0$

pct : $x0 = 2*y0$

Continuing further we get:

source: Lecture Notes on "Program Analysis", ETH Zurich, Martin Vechev.

Concolic execution - Example

```
int twice(int v) {
  return 2 * v;
}

void test(int x, int y) {
  z = twice(y);
  if (x == z) {
    if (x > y + 10)
      ERROR;
  }
}

int main() {
  x = read();
  y = read();
  test(x,y);
}
```

We reach the “else” branch of x > y + 10

σ : x $\mapsto$ 2,
y $\mapsto$ 1,
z $\mapsto$ 2

σ_s : x $\mapsto$ x0,
y $\mapsto$ y0
z $\mapsto$ 2*y0

pct : x0 = 2*y0
$\wedge$
x0 $\leq$ y0+10

Again, concolic execution may want to explore the ‘true’ branch of x > y + 10.

source: Lecture Notes on "Program Analysis", ETH Zurich, Martin Vechev.

Concolic execution - Example

```
int twice(int v) {
  return 2 * v;
}

void test(int x, int y) {
  z = twice(y);
  if (x == z) {
    if (x > y + 10)
      ERROR;
  }
}

int main() {
  x = read();
  y = read();
  test(x,y);
}
```

We reach the "else" branch of x > y + 10

$$\sigma : x \mapsto 2,\ y \mapsto 1,\ z \mapsto 2$$

$$\sigma_s : x \mapsto x0,\ y \mapsto y0,\ z \mapsto 2*y0$$

$$\text{pct} : x0 = 2*y0 \wedge x0 \leq y0+10$$

Concolic execution now negates the conjunct $x0 \leq y0+10$ obtaining:

$$x0 = 2*y0 \wedge x0 > y0+10$$

A satisfying assignment is: $x0 \mapsto 30,\ y0 \mapsto 15$

source: Lecture Notes on "Program Analysis", ETH Zurich, Martin Vechev.

Concolic execution - Example

```
int twice(int v) {
  return 2 * v;
}

void test(int x, int y) {
  z = twice(y);
  if (x == z) {
    if (x > y + 10)
      ERROR;
  }
}

int main() {
  x = read();
  y = read();
  test(x,y);
}
```

If we run the program with the input:

$$x0 \mapsto 30, \; y0 \mapsto 15$$

we will now reach the ERROR state.

As we can see from this example, by keeping the symbolic information, the concrete execution can use that information in order to obtain new inputs.

source: Lecture Notes on "Program Analysis", ETH Zurich, Martin Vechev.

Non-linear constraints - Example

```
int twice(int v) {
  return v * v;
}

void test(int x, int y) {
  z = twice(y);
  if (x == z) {
    if (x > y + 10)
      ERROR;
  }
}

int main() {
  x = read();
  y = read();
  test(x,y);
}
```

Let us again consider our example and see what concolic execution would do with non-linear constraints.

source: Lecture Notes on "Program Analysis", ETH Zurich, Martin Vechev.

Non-linear constraints - Example

```
int twice(int v) {
  return v * v;
}

void test(int x, int y) {
  z = twice(y);
  if (x == z) {
    if (x > y + 10)
      ERROR;
  }
}

int main() {
  x = read();
  y = read();
  test(x,y);
}
```

The read() functions read a value from the input. Suppose we read x = 22 and y =7.

σ : x $\mapsto$ 22,
y $\mapsto$ 7

σ_s : x $\mapsto$ x0,
y $\mapsto$ y0

pct : true

source: Lecture Notes on "Program Analysis", ETH Zurich, Martin Vechev.

Non-linear constraints - Example

```
int twice(int v) {
  return v * v;
}

void test(int x, int y) {
  z = twice(y);
  if (x == z) {
    if (x > y + 10)
      ERROR;
  }
}

int main() {
  x = read();
  y = read();
  test(x,y);
}
```

σ : x $\mapsto$ 22,
y $\mapsto$ 7,
z $\mapsto$ 49

σ_s : x $\mapsto$ x0,
y $\mapsto$ y0
z $\mapsto$ y0*y0

pct : true

The concrete execution will now take the ‘else’ branch of x == z.

source: Lecture Notes on "Program Analysis", ETH Zurich, Martin Vechev.

Non-linear constraints - Example

```
int twice(int v) {
  return v * v;
}

void test(int x, int y) {
  z = twice(y);
  if (x == z) {
    if (x > y + 10)
      ERROR;
  }
}

int main() {
  x = read();
  y = read();
  test(x,y);
}
```

Hence, we get:

σ : x $\mapsto$ 22,
y $\mapsto$ 7,
z $\mapsto$ 49

σ_s : x $\mapsto$ x0,
y $\mapsto$ y0
z $\mapsto$ y0*y0

pct : x0 $\neq$ y0*y0

source: Lecture Notes on "Program Analysis", ETH Zurich, Martin Vechev.

Non-linear constraints - Example

```
int twice(int v) {
  return v * v;
}

void test(int x, int y) {
  z = twice(y);
  if (x == z) {
    if (x > y + 10)
      ERROR;
  }
}

int main() {
  x = read();
  y = read();
  test(x,y);
}
```

However, here we have a non-linear constraint `x0 ≠ y0*y0` . If we would like to explore the true branch we negate the constraint, obtaining `x0 = y0*y0` but again we have a non-linear constraint !

In this case, concolic execution simplifies the constraint by plugging in the concrete values for `y0` in this case, 7, obtaining the simplified constraint:

$$x0 = 49$$

Hence, it now runs the program with the input

$$x \mapsto 49, \quad y \mapsto 7$$

source: Lecture Notes on "Program Analysis", ETH Zurich, Martin Vechev.

Non-linear constraints - Example

```
int twice(int v) {
  return v * v;
}

void test(int x, int y) {
  z = twice(y);
  if (x == z) {
    if (x > y + 10)
      ERROR;
  }
}

int main() {
  x = read();
  y = read();
  test(x,y);
}
```

Running with the input

$$x \mapsto 49, \quad y \mapsto 7$$

will reach the error state.

However, notice that with these inputs, if we try to simplify non-linear constraints by plugging in concrete values (as concolic execution does), then concolic execution we will never reach the else branch of the `if (x > y + 10)` statement.

source: Lecture Notes on "Program Analysis", ETH Zurich, Martin Vechev.

Symbolic execution

- Is a popular technique for analyzing programs.
    - Completely automated
    - Relies on SMT solvers
- To terminate, may need to bound loops.
    - Leads to under-approximation
- To handle non-linear constraints and external environment, mixes concrete and symbolic execution (concolic execution).

Quiz time!

`https://www.questionpro.com/t/AT4NiZr80r`

What is Model Checking?

The big picture

- Application domain: mostly concurrent, reactive systems
- Verify that a system satisfies a property
  1. Model the system in the model checker's description language.
     Call this model M.
  2. Express the property to be verified in the model checker's specification language. Call this formula ϕ.
  3. Run the model checker to show that M satisfies ϕ.
- Automatic for finite-state models

Example

- Two processes executed in parallel
- Each process undergoes transitions $n \to r \to c \to n \to \ldots$ where
  - n denotes "**n**ot in critical section"
  - r denotes "**r**equesting to enter critical section"
  - c denotes "**c**ritical section"

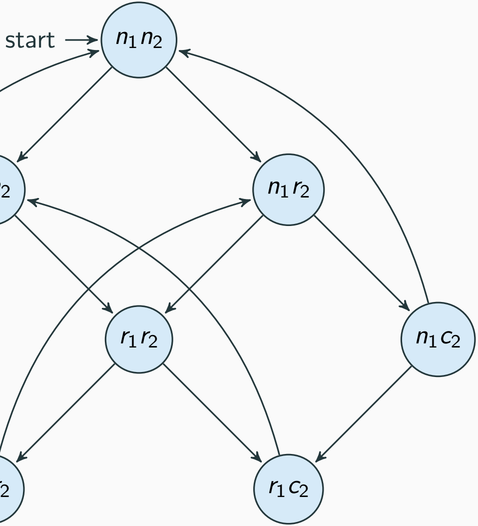
Example

- Two processes executed in parallel
- Each process undergoes transitions $n \to r \to c \to n \to \ldots$ where
  - n denotes "**n**ot in critical section"
  - r denotes "**r**equesting to enter critical section"
  - c denotes "**c**ritical section"
- Requirements
  - Safety: only one process may execute critical section code at any point
  - Liveness: whenever a process requests to enter its critical section, it will eventually be allowed to do so
  - Non-blocking: a process can always request to enter its critical section

Example (cont.)

- We write a program P to fulfill these requirements. But is it really doing its job?
- We construct a model M for P such that M captures the relevant behaviour of P.

Example (cont.)



Example (cont.)

- Based on a definition of when a model satisfies a property, we can determine whether M satisfies P's required properties.
- Example: M satisfies the safety requirement if no state reachable from the start state (including itself) is labeled $c_1 c_2$. Thus our M satisfies P's safety requirement.
- NOTE: the conclusion that P satisfies these requirements depends on the (unverified) assumption that M is a faithful representation of all the relevant aspects of P.

Uses of Model Checking

Verification of specific properties of

- Hardware circuits
- Communication protocols
- Control software
- Embedded systems
- Device drivers
- ...

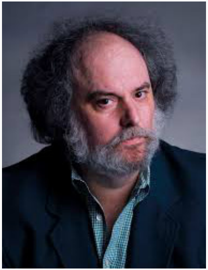
Uses of Model Checking

Model Checking has been used to

- Check Microsoft Windows device drivers for bugs
    - The Static Driver Verifier tool
    - SLAM project of Microsoft
- The SPIN tool
    - `http://spinroot.com`
    - `http://spinroot.com/spin/success.html`
    - Flood control barrier control software
    - Parts of *Mars Science Laboratory*, *Deep Space 1*, *Cassini*, *the Mars Exploration Rovers*, *Deep Impact*
- PEPA (Performance Evaluation Process Algebra)
    - `http://www.dcs.ed.ac.uk/pepa/`
    - Multiprocess systems
    - Biological systems
- ...

The ACM Turing Award in 2007

Edmund Clarke



Allen Emerson

Joseph Sifakis

"for their role in developing Model-Checking into a highly effective verification technology that is widely adopted in the hardware and software industries"

Model Checking - Models

A model of some system has

- A finite set of states
- A subset of states considered as the initial states
- A transition relation which, given a state, describes all the states that can be reached "in one time step"

Model Checking - Models

A model of some system has

- A finite set of states
- A subset of states considered as the initial states
- A transition relation which, given a state, describes all the states that can be reached "in one time step"

Refinements of this setup can handle:

- Infinite state spaces
- Continuous state spaces
- Probabilistic Transitions
- ...

Model Checking - Models

Models are always abstraction of reality.

- We must choose what to model and what not to model
- There are limitations forced by the formalism
  - e.g., here we are limited to finite state models
- There will be things we do not understand sufficiently to model
  - e.g., people

Source: *La trahison des images* by René Magritte. Licensed under Fair use via Wikipedia
`http://en.wikipedia.org/wiki/File:MagrittePipe.jpg#mediaviewer/File:MagrittePipe.jpg`

Model Checking - Specifications

We are interested in specifying behaviours of systems over time (use Temporal Logic).

Specifications are built from

- primitive properties of individual states
  - e.g., *is on*, *is off*, *is active*, *is reading*
- propositional connectives $\wedge, \vee, \neg, \rightarrow$
- temporal connectives
  - e.g., *At* **all times**, *the system is not simultaneously reading and writing.*
  - e.g., *If a request signal is asserted* **at some time**, *a corresponding grant signal will be asserted* **within 10 time units.**

Model Checking - Specifications

The exact set of temporal connectives differs across temporal logics.

Logics can differ in how they treat time:

- Continuous time vs. Discrete time
- Linear Time vs. Branching time
- ...

Linear vs. Branching Time

Linear Time

- Considers paths (sequences of states)
- Questions of the form
  - *For all paths, does some path property hold?*
  - *Does there exist a path such that some path property holds?*

Linear vs. Branching Time

Linear Time

- Considers paths (sequences of states)
- Questions of the form
  - *For all paths, does some path property hold?*
  - *Does there exist a path such that some path property holds?*

Branching Time

- Considers trees of possible future states from each initial state
- Questions can become more complex
  - *For all states reachable from an initial state, does there exist an onwards path to a state satisfying some property?*

References

- Lecture Notes on "Program Analysis", ETH Zurich, Martin Vechev.
- Lecture Notes on "Techniques for Program Analysis and Verification", Stanford, Clark Barrett.
- Lecture Notes on "Program verification", ETH Zurich, Alexander Summers.
- Lecture Notes on "Computer-Aided Reasoning for Software Engineering", University of Washington, Emina Torlak.