Emina Torlak
emina@cs.washington.edu

Computer-Aided Reasoning for Software
Angellic Execution
Today
Today

Last lecture

• Verifying compiler optimizations with SMT solvers
Today

Last lecture
  • Verifying compiler optimizations with SMT solvers

Today
  • Beyond verification: solvers as interpreters
Today

Last lecture
• Verifying compiler optimizations with SMT solvers

Today
• Beyond verification: solvers as interpreters

Announcements
• Project presentation logistics:
  • 8 min talk (problem statement, demo, results)
  • An electronic poster (single slide)
So far, we have used solvers as demonic oracles

Program P

Spec S

Verifier

P ∧ ¬S

Solver

An input i on which P violates S
But solvers can also act as angelic oracles

```plaintext
P() {
    y = choose();
    ...
    assert S;
}
```

A trace of that $P$ satisfies $S$
But solvers can also act as angelic oracles

```plaintext
P() {
  y = choose();
  ...
  assert S;
}
```

1. Definitions
2. Implementations
3. Applications
Angelic non-determinism, two ways

Angelic choice:
\[ \text{choose}(T) \]

Specification statement:
\[ x_1, \ldots, x_n \leftarrow [\text{pre}, \text{post}] \]

Robert Floyd, 1966

Carroll Morgan, 1988
Angelic non-determinism, two ways

**Angelic choice:**
choose(T)

**Specification statement:**
\[ x_1, \ldots, x_n \leftarrow [\text{pre}, \text{post}] \]

Non-deterministically chooses a value of (finite) type T so that the rest of the program terminates successfully.

Designed to abstract away the details of backtracking search.

Robert Floyd, 1966

Carroll Morgan, 1988

A programming abstraction
Angelic non-determinism, two ways

**Angelic choice:**
choose(T)

**Specification statement:**
\[ x_1, \ldots, x_n \leftarrow [\text{pre}, \text{post}] \]

Non-deterministically modifies the values of frame variables \( x_1, \ldots, x_n \) so that \( \text{post} \) holds in the next state if \( \text{pre} \) holds in the current state.

Designed to enable derivation of programs from specifications via step-wise refinement.

Robert Floyd, 1966

Carroll Morgan, 1988

A programming abstraction

A refinement abstraction
Angelico non-determinismo, due modi: un esempio

**Angelico scelta:**
choose(T)

**Specifica affermazione:**
\[ x_1, \ldots, x_n \leftarrow [\text{pre}, \text{post}] \]

\[
\begin{align*}
s & = 16 \\
r & = \text{choose}(\text{int}) \\
\text{if } (r \geq 0) & \quad \text{assert } r \cdot r \leq s < (r+1) \cdot (r+1) \\
\text{else} & \quad \text{assert } r \cdot r \leq s < (r-1) \cdot (r-1)
\end{align*}
\]

\[
\begin{align*}
s & = 16 \\
r & \leftarrow [(r \geq 0 \land r \cdot r \leq s < (r+1) \cdot (r+1)) \lor (r < 0 \land r \cdot r \leq s < (r-1) \cdot (r-1))]
\end{align*}
\]
Angelic non-determinism, two ways: an example

**Angelic choice:**
\[
\text{choose}(T)
\]

**Specification statement:**
\[
x_1, \ldots, x_n \leftarrow [\text{pre}, \text{post}]
\]

Interleaves imperative and angelic execution. As a result, implementation requires global constraint solving.

Alternates between angelic and imperative execution. As a result, implementation requires only local constraint solving.
Angelic non-determinism, two ways: an example

**Angelic choice:**
choose(T)

**Specification statement:**
\[ x_1, \ldots, x_n \leftarrow \text{[pre, post]} \]

\[
\begin{align*}
\text{s} &= 16 \\
\text{r} &= \text{choose(int)} \\
\text{if (r} &\geq 0) \\
\quad &\text{assert } r*r \leq s < (r+1)*(r+1) \\
\text{else} \\
\quad &\text{assert } r*r \leq s < (r-1)*(r-1)
\end{align*}
\]

“Angelic Interpretation”

\[
\begin{align*}
\text{s} &= 16 \\
\text{r} &\leftarrow [(r \geq 0 \land \\
\quad r*r \leq s < (r+1)*(r+1)) \lor \\
\quad (r < 0 \land \\
\quad r*r \leq s < (r-1)*(r-1))] \\
\text{else} \\
\quad &\text{assert } r*r \leq s < (r-1)*(r-1)
\end{align*}
\]

“Mixed Interpretation”
Mixed interpretation with a model finder (1/4)

Java program with Alloy specification statements

Squander

PBnJ
Mixed interpretation with a model finder (1/4)

Java program with Alloy specification statements

Squander
@Requires("z.key !in this.nodes.key")
@Ensures("this.nodes = @old(this.nodes) + z")
@Modifies("this.root,
    this.nodes.left | _<1> = null,
    this.nodes.right | _<1> = null")

public void insert(Node z) {
    Squander.exe(this, z); }

Mixed interpretation with a model finder (2/4)

@Requires("z.key !in this.nodes.key")
@Ensures("this.nodes = @old(this.nodes) + z")
@Modifies("this.root,
    this.nodes.left | _<1> = null,
    this.nodes.right | _<1> = null")

public void insert(Node z) {
    Squander.exe(this, z); }

Specification statements describing insertion of a new node z into a binary search tree.
Mixed interpretation with a model finder (2/4)

@Requires("z.key !in this.nodes.key")
@Ensures("this.nodes = @old(this.nodes) + z")
@Modifies("this.root,
    this.nodes.left | _<1> = null,
    this.nodes.right | _<1> = null")

public void insert(Node z) {
    Squander.exe(this, z);
}

Specification statements describing insertion of a new node z into a binary search tree.

Call to the Squander mixed interpreter ensures that the state of this tree and the node z is mutated so that the insertion specification is satisfied when the insert method returns.
Mixed interpretation with a model finder (2/4)

@Requires("z.key !in this.nodes.key")
@Ensures("this.nodes = @old(this.nodes) + z")
@Modifies("this.root,
    this.nodes.left | _<1> = null,
    this.nodes.right | _<1> = null")

public void insert(Node z) {
    Squander.exe(this, z);
}

Specification statements describing insertion of a new node z into a binary search tree.

Execution steps:

- Serialize the relevant part of the heap to a universe and bounds
- Use Kodkod to solve the specs against the resulting universe / bounds
- Deserialize the solution (if any) and update the heap accordingly
Mixed interpretation with a model finder (3/4)

@Requires("z.key !in this.nodes.key")
@Ensures("this.nodes = @old(this.nodes) + z")
@Modifies("this.root,
    this.nodes.left | _<1> = null,
    this.nodes.right | _<1> = null")

public void insert(Node z) {
    Squander.exe(this, z); }

```
    t1
    root
    n1
    key: 5
    left
   /   \\ \
  n2  n3
  key: 0 key: 6
right
n4
key: 1
```
Mixed interpretation with a model finder (3/4)

@Requires("z.key !in this.nodes.key")
@Ensures("this.nodes = @old(this.nodes) + z")
@Modifies("this.root, this.nodes.left | _<1> = null, this.nodes.right | _<1> = null")

public void insert(Node z) {
    Squander.exe(this, z); }

reachable objects

T = \{⟨t_1⟩\}
N = \{⟨n_1⟩, ..., ⟨n_4⟩\}
null = \{⟨null⟩\}
this = \{⟨t_1⟩\}
z = \{⟨n_4⟩\}
ints = \{⟨0⟩, ⟨1⟩, ⟨5⟩, ⟨6⟩ \}
Mixed interpretation with a model finder (3/4)

@Requires("z.key !in this.nodes.key")
@Ensures("this.nodes = @old(this.nodes) + z")
@Modifies("this.root,
    this.nodes.left | _<1> = null,
    this.nodes.right | _<1> = null")

public void insert(Node z) {
    Squander.exe(this, z); }

---

pre-state

key_old = {⟨n₁, 5⟩, ..., ⟨n₄, 1⟩}
root_old = {⟨t₁, n₁⟩}
left_old = {⟨n₁, n₂⟩, ..., ⟨n₄, null⟩}
right_old = {⟨n₁, n₃⟩, ..., ⟨n₄, null⟩}

reachable objects

T = {⟨t₁⟩}
N = {⟨n₁⟩, ..., ⟨n₄⟩}
null = {⟨null⟩}
this = {⟨t₁⟩}
z = {⟨n₄⟩}
ints = {⟨0⟩, ⟨1⟩, ⟨5⟩, ⟨6⟩}
Mixed interpretation with a model finder (3/4)

@Requires("z.key !in this.nodes.key")
@Ensures("this.nodes = @old(this.nodes) + z")
@Modifies("this.root, this.nodes.left | _<1> = null, this.nodes.right | _<1> = null")

public void insert(Node z) {
    Squander.exe(this, z);
}

<table>
<thead>
<tr>
<th>pre-state</th>
<th>post-state</th>
</tr>
</thead>
<tbody>
<tr>
<td>key_{old} = {⟨n₁, 5⟩, ..., ⟨n₄, 1⟩}</td>
<td>{⟨n₁, 5⟩, ..., ⟨n₄, 1⟩}</td>
</tr>
<tr>
<td>root_{old} = {⟨t₁, n₁⟩}</td>
<td>{⟨n₁, 5⟩, ..., ⟨n₄, 1⟩}</td>
</tr>
<tr>
<td>left_{old} = {⟨n₁, n₂⟩, ..., ⟨n₄, null⟩}</td>
<td>{⟨t₁⟩} × {n₁, ..., n₄, null}</td>
</tr>
<tr>
<td>right_{old} = {⟨n₁, n₃⟩, ..., ⟨n₄, null⟩}</td>
<td>{n₁, n₂, n₃, n₄} × {n₁, ..., n₄, null}</td>
</tr>
</tbody>
</table>

reachable objects:

T = \{⟨t₁⟩\}
N = \{⟨n₁⟩, ..., ⟨n₄⟩\}
null = \{⟨null⟩\}
this = \{⟨t₁⟩\}
z = \{⟨n₄⟩\}
ints = \{⟨0⟩, ⟨1⟩, ⟨5⟩, ⟨6⟩\}
Mixed interpretation with a model finder (3/4)

```java
@Requires("z.key !in this.nodes.key")
@Ensures("this.nodes = @old(this.nodes) + z")
@Modifies("this.root,
            this.nodes.left | _<1> = null,
            this.nodes.right | _<1> = null")

public void insert(Node z) {
    Squander.exe(this, z); }
```

```
   n4
   |   \
   v   \\
  n2    n3
  |    |
  v    v
n1    n1
   |
   v
   n2
```

key: 0
key: 5
key: 6
key: 1

root
left
right

 tl
Mixed interpretation with a model finder (4/4)

@Requires("z.key !in this.nodes.key")
@Ensures("this.nodes = @old(this.nodes) + z")
@Modifies("this.root,
    this.nodes.left | _<1> = null,
    this.nodes.right | _<1> = null")

public void insert(Node z) {
    Squander.exe(this, z); }

Many more features (e.g., support for obtaining all solutions, support for data abstraction, etc.).
See Unifying Execution of Declarative and Imperative Code for details.
Mixed interpretation with a model finder (4/4)

@Requires("z.key !in this.nodes.key")
@Ensures("this.nodes = @old(this.nodes) + z")
@Modifies("this.root, this.nodes.left | _<1> = null, this.nodes.right | _<1> = null")

public void insert(Node z) {
    Squander.exe(this, z); }

Many more features (e.g., support for obtaining all solutions, support for data abstraction, etc.).
See Unifying Execution of Declarative and Imperative Code for details.

Incompleteness due to finitization: Squander bounds the number of new instances of a given type that Kodkod can create to satisfy the specification.
Mixed interpretation with an SMT solver (1/3)

Scala program with PureScala specifications

Kaplan

Leon

Z3
Mixed interpretation with an SMT solver (1/3)

Scala program with PureScala specifications

Kaplan

Leon

Z3

PureScala is a pure, Turing complete subset of Scala that supports unbounded datatypes and arbitrary recursive functions.
Mixed interpretation with an SMT solver (2/3)

```scala
@spec def noneDivides(from: Int, j: Int) : Boolean {
  from == j ||
  (j % from != 0 && noneDivides(from+1, j))
}

@spec def isPrime(i: Int) : Boolean {
  i >= 2 && noneDivides(2, i)
}

val primes =
((isPrime(_Int)) minimizing
 ((x:Int) => x)).findAll

> primes.take(10).toList
List(2, 3, 4, 5, 11, 17, 19, 23, 29)
```
Mixed interpretation with an SMT solver (2/3)

@spec def noneDivides(from: Int, j: Int) : Boolean {
    from == j ||
    (j % from != 0 && noneDivides(from+1, j))
}

@spec def isPrime(i: Int) : Boolean {
    i >= 2 && noneDivides(2, i)
}

val primes =
((isPrime(_Int)) minimizing
  ((x:Int) => x)).findAll

> primes.take(10).toList
List(2, 3, 4, 5, 11, 17, 19, 23, 29)

Recursive specification functions. Mutual recursion also allowed.
Mixed interpretation with an SMT solver (2/3)

@spec def noneDivides(from: Int, j: Int) : Boolean {
  from == j ||
  (j % from != 0 && noneDivides(from+1, j))
}

@spec def isPrime(i: Int) : Boolean {
  i >= 2 && noneDivides(2, i)
}

val primes = ((isPrime(_Int)) minimizing ((x: Int) => x)).findAll

> primes.take(10).toList
List(2, 3, 4, 5, 11, 17, 19, 23, 29)

Recursive specification functions. Mutual recursion also allowed.

Call the Kaplan mixed interpreter to obtain the first 10 primes.
Mixed interpretation with an SMT solver (2/3)

@spec def noneDivides(from: Int, j: Int) : Boolean {
    from == j ||
    (j % from != 0 && noneDivides(from+1, j))
}

@spec def isPrime(i: Int) : Boolean {
    i >= 2 && noneDivides(2, i)
}

val primes =
((isPrime(_Int)) minimizing
 ((x:Int) => x)).findAll

> primes.take(10).toList
List(2, 3, 4, 5, 11, 17, 19, 23, 29)

Recursive specification functions. Mutual recursion also allowed.

Two execution modes:
- Eager: uses Leon to find a satisfying assignment for a given specification.
- Lazy: accumulates specifications, checking their feasibility, until the programmer asks for the value of a logical variable. The variable is then frozen (permanently bound) to the returned value.

Call the Kaplan mixed interpreter to obtain the first 10 primes.
Mixed interpretation with an SMT solver (3/3)

@spec def noneDivides(from: Int, j: Int) : Boolean {
  from == j ||
  (j % from != 0 && noneDivides(from+1, j))
}

@spec def isPrime(i: Int) : Boolean {
  i >= 2 && noneDivides(2, i)
}

val primes =
  ((isPrime(_Int)) minimizing
   ((x:Int) => x)).findAll

> primes.take(10).toList
List(2, 3, 4, 5, 11, 17, 19, 23, 29)

Many more features (e.g., support for optimization).
See Constraints as Control for details.
Mixed interpretation with an SMT solver (3/3)

@spec def noneDivides(from: Int, j: Int) : Boolean {
  from == j ||
  (j % from != 0 && noneDivides(from+1, j))
}

@spec def isPrime(i: Int) : Boolean {
  i >= 2 && noneDivides(2, i)
}

val primes = ((isPrime(_Int)) minimizing ((x:Int) => x)).findAll

> primes.take(10).toList
List(2, 3, 4, 5, 11, 17, 19, 23, 29)

Incompleteness due to undecidability of PureScala.

Many more features (e.g., support for optimization).
See Constraints as Control for details.
s = 16
r = choose(int)
if (r ≥ 0)
    assert r*r ≤ s < (r+1)*(r+1)
else
    assert r*r ≤ s < (r-1)*(r-1)
Angelica interpretation with a solver

\[ s = 16 \]
\[ r = \text{choose}(\text{int}) \]
\begin{cases} 
\text{if } (r \geq 0) \\
\quad \text{assert } r \cdot r \leq s < (r+1) \cdot (r+1) \\
\text{else} \\
\quad \text{assert } r \cdot r \leq s < (r-1) \cdot (r-1) 
\end{cases}

Execution steps:

- Translate to the entire program to constraints using either BMC or SE.
- Query the solver for one or all solutions that satisfy the constraints.
- Convert each solution to a valid program trace (represented, e.g., as a sequence of choices made by the oracle in a given execution).
Applications of angelic execution
Applications of angelic execution

Declarative mocking [Samimi et al., ISSTA’13]
Applications of angelic execution

Declarative mocking [Samimi et al., ISSTA’13]

Angelic debugging [Chandra et al., ICSE’11]
Applications of angelic execution

Declarative mocking [Samimi et al., ISSTA’13]

Angelic debugging [Chandra et al., ICSE’11]

Imperative/declarative programming [Milicevic et al., ICSE’11]
Applications of angelic execution

Declarative mocking [Samimi et al., ISSTA’13]

Angelic debugging [Chandra et al., ICSE’11]

Imperative/declarative programming [Milicevic et al., ICSE’11]

Algorithm development [Bodik et al., POPL’10]
Applications of angelic execution

**Declarative mocking** [Samimi et al., ISSTA’13]

**Angelic debugging** [Chandra et al., ICSE’11]

**Imperative/declarative programming** [Milicevic et al., ICSE’11]

**Algorithm development** [Bodik et al., POPL’10]

**Dynamic program repair** [Samimi et al., ECOOP’10]
Applications of angelic execution

Declarative mocking [Samimi et al., ISSTA’13]

Angelic debugging [Chandra et al., ICSE’11]

Imperative/declarative programming [Milicevic et al., ICSE’11]

Algorithm development [Bodik et al., POPL’10]

Dynamic program repair [Samimi et al., ECOOP’10]

Test case generation [Khurshid et al., ASE’01]

...
Summary

Today

• Angelic nondeterminism with specifications statements and angelic choice

• Angelic execution with model finders and SMT solvers

• Applications of angelic execution

Next lecture

• Program synthesis