Program Synthesis

with Jennisys

Aleksandar Milicevic
Rustan Leino
Concolic Synthesis with Jennisys

Aleksandar Milicevic
Rustan Leino
Program Extrapolation with Jennisys

Aleksandar Milicevic
Rustan Leino
Specifications are good

- Formally give meaning to your programs

Typically used to check a separate program

- Program verification
- Proving the absence of safety/security violations
- Test case generation

Also convenient

- Elegantly and succinctly express complex properties/invariants

We would like to use specs even for writing programs
» Write programs **declaratively** (say *what* not *how*)

» “It would be very nice to input this description into some suitably programmed computer, and get the computer to translate it automatically into a subroutine”

  - *Tony Hoare* [“An overview of some formal methods for program design”, 1987]

» A solution: **British Museum algorithm**
  > Start with some set of axioms
  > Use them to generate at random all provable theorems
  > Wait until your program is generated

» “Under reasonable assumptions, the whole universe will reach a uniform temperature around four degrees Kelvin long before any interesting computation is complete”
» **Executable specifications**
  
  > Specification are executed directly at *runtime*
  > Typically a constraint solver is used to search for a model
  > The solution is *valid* for the *current program state* only
  > Preferably integrated within an existing programming language

» **Program synthesis**
  
  > *Statically* generate imperative code *equivalent* to given declarative spec
  > Covers all cases at once

---

<table>
<thead>
<tr>
<th></th>
<th>Executable Specifications</th>
<th>Program Synthesis</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>running time</strong></td>
<td>✅ Big</td>
<td>✗ Huge</td>
</tr>
<tr>
<td><strong>frequency</strong></td>
<td>✗ At every invocation</td>
<td>✅ once, statically</td>
</tr>
<tr>
<td><strong>power</strong></td>
<td>✅ NP-hard specs</td>
<td>✗ (mostly) linear algorithms</td>
</tr>
</tbody>
</table>
» Combine the green checkmarks of both?
  > Synthesis and executable specs are still quite orthogonal

» Instead: find a sweet spot of synthesis
  > Identify a category of programs that can be easily synthesized
  > The synthesis should be fully automatic
  > It shouldn’t be super slow: order of seconds, not hours
  > The only input from the user is the spec (declarative, first-order)
  > Implementation:
    → execute specifications and generalize from concrete instances
Public interface: high-level interface in terms of abstract fields

Data-model: data description, concrete fields, additional invariants

Code: implementation code for methods that could not be synthesized
interface SetNode {
    var elems: set[int]
}

constructor Init(x: int)
    ensures elems = {x}

constructor Double(a: int, b: int)
    ensures elems = {a b}

method Contains(p: int) returns (ret: bool)
    ensures ret = (p in elems)
}

datamodel SetNode {
    var data: int
    var left: SetNode
    var right: SetNode

    invariant
        elems = {data} + (left != null ? left.elems : {}) + (right != null ? right.elems : {})
        left != null ==> forall e :: e in left.elems ==> e < data
        right != null ==> forall e :: e in right.elems ==> e > data
}
» **Techniques**

- Solving for **concrete instances** that meet the spec
- Generalizing from **concrete heap instances**
- Inferring branching (flow) structure
- Delegating to method calls

» **Application**

- Synthesizing **Constructors**
- Synthesizing **Recursive Functional-Style Methods**
Synthesizing Constructors – Initial Idea

- Constructors only initialize the object fields enough to find assignments to all object fields
- Execute the constructor specification to find a concrete instance (a model that satisfies all constraints of the spec)
- Print out straight-line code that assigns values to fields according to the model
- Use Dafny program verifier to execute specifications

Executing Specs
Example (Executing Specification)

Jennisys

```
interface SetNode {
    invariant ...
}
```

Dafny

```
class SetNode {
    ghost var elems: set<int>;
    var data: int;
    var left: SetNode;
    var right: SetNode;

    function Valid(): bool {
        user-defined invariant &&
        left != null ==> left.Valid() &&
        right != null ==> right.Valid()
    }
}
```

```
class Set {
    constructor SingletonZero()
    ensures elems = {0}
}
```

```
class Set {
    ghost var elems: set<int>;
    var root: SetNode;

    function Valid(): bool {
        ...
    }

    method SingletonZero() modifies this;
    {
        // assume invariant and postcondition
        assume Valid();
        assume elems == {0};
        // assert false
        assert false;
    }
}
```

Counterexample encodes an instance for which all constraints hold
Example (Synthesized Code)

Jennisys

```
interface SetNode {
  invariant ...
}
```

Dafny

```
class SetNode {
  ghost var elems: set<int>;
  var data: int;
  var left: SetNode;
  var right: SetNode;

  function Valid(): bool { ... }
}
```

```
class Set {
  ghost var elems: set<int>;
  var root: SetNode;

  function Valid(): bool { ... }
}
```

```
method SingletonZero()
  modifies this;
  ensures Valid && elems == {0};
{
  var gensym74 := new SetNode;
  this.elems := {0};
  this.root := gensym74;
  gensym74.data := 0;
  gensym74.elems := {0};
  gensym74.left := null;
  gensym74.right := null;
}
```

interface Set {
  constructor SingletonZero()
  ensures elems = {0}
}
**Constructors with Parameters**

- Assigning concrete values obtained from the solver is no longer enough

```
interface Set {
    constructor SingletonSum(p: int, q: int)
    ensures elems = {p + q}
}
```

Spec

- Simply matching up values of unmodifiable fields (e.g. method input args) with values assigned to fields is not enough

- **Custom spec evaluation:**
  evaluate parts of the spec wrt the current instance
Custom Spec Evaluation

**datamodel** Set {
  invariant
  root = null ==> elems = {}
  root != null ==> elems = root.elems
  constructor SingletonSum(p: int, q: int)
  ensures elems = {p + q}
}

**datamodel** SetNode {
  invariant
  elems = {data} + (left != null ? left.elems : {})
  + (right != null ? right.elems : {})
  left != null ==> forall e :: e in left.elems ==> e < data
  right != null ==> forall e :: e in right.elems ==> e > data
}

> Evaluate the spec without resolving unmodifiable fields
> Then do the match-up
> Matching up can still be ambiguous

→ **better approach:** use **concolic spec evaluation** and **unification**

Generalizing
Concolic Spec Evaluation

**datamodel** Set {
  **invariant**
  root = null ==> elems = {}
  root != null ==> elems = root.elems

  **constructor** SingletonSum(p: int, q: int)
  **ensures** elems = \{p + q\}
}

datamodel SetNode {
  **invariant**
  elems = {data} + (left != null ? left.elems : {})
  + (right != null ? right.elems : {})
  left != null ==> forall e :: e in left.elems ==> e < data
  right != null ==> forall e :: e in right.elems ==> e > data
}

elems = \{p + q\}  \rightarrow  data = p + q  \rightarrow  elems = \{data\}

> Evaluate the spec against the instance without resolving anything
  - This gets us a **simpler spec** for the current instance
> Use **unification** to obtain **symbolic** values for fields
Inferring Branching (Flow) Structure

- Straight-line code is no longer enough

```java
interface Set {
    constructor Double(p: int, q: int)
    requires p != q
    ensures elems = {p q}
}
```

Spec

- A correct solution has to consider two cases
  1. $p > q$, and
  2. $p < q$

Approach:
- Find a concrete instance
- Generalize and try to verify
- If it doesn’t verify
  - Infer the needed guard using custom spec evaluation

Concrete Instance

p = 1
q = -2

Infering Flow
» **Inferring Guards**

| datamodel Set { |
| invariant |
| root = null ==> elems = {} |
| root != null ==> elems = root.elems |
| constructor Double(p: int, q: int) |
| ensures elems = {p q} |
| } |

| datamodel SetNode { |
| invariant |
| elems = {data} + (left != null ? left.elems : {}) |
| + (right != null ? right.elems : {}) |
| left != null ==> forall e :: e in left.elems ==> e < data |
| right != null ==> forall e :: e in right.elems ==> e > data |
| } |

> Evaluate the spec without resolving unmodifiable fields
> Find all true clauses and try to use them as if guards

→ *Concolic evaluation discovers clauses hidden behind the declarativness*
> If it verifies, negation the inferred guard and go all over again.
» **Delegating** to existing methods

> So far, all objects are initialized in the constructor for the root object
> → Breaks encapsulation
> > Instead, each object should be initialized in its own constructor
> > **Approach:**
> > → Find a solution as before
> > → For each child object **infer a spec needed for its initialization**
> > → Find an **existing** constructor that meets this spec, or create a new one

» Spec Inference for Child Objects

> Simply **use** the obtained **assignments** to all of its **public fields**

» Finding existing methods that meet a given spec

> Use **syntactic unification** with a few semantics rules
> > Limitation: in some cases valid candidate methods can be missed
Delegation Example

class Set {
    method Double(p: int, q: int)
        more_spec
        ensures elems == {p, q}
        {
            var sym80 := new SetNode;
            sym80.Double(p, q);
            this.elems := {q, p};
            this.root := sym80;
        }
}

class SetNode {
    method Double(p: int, q: int)
        more_spec
        ensures elems == {p, q}
        {
            if (b > a) {
                this.DoubleBase(b, a);
            } else {
                this.DoubleBase(a, b);
            }
        }
}

method DoubleBase(x: int, y: int)
    more_spec
    requires x < y;
    ensures elems == {x, y};
    {
        var sym88 := new SetNode;
        sym88.Init(x);
        this.data := y;
        this.elems := {y, x};
        this.left := null;
        this.right := sym88;
    }

Finding existing methods that meet a given spec

> Use syntactic unification with a few semantics rules
> Limitation: in some cases valid candidate methods can be missed
Synthesizing **Recursive Methods**

- **Goal**: synthesize simple functional-style methods:
  - assignments to fields are in the form of function compositions
    (as opposed to arbitrary statement sequences with mutable variables)

- **Idea**:
  - Again, generalize from concrete instances
  - Again, obtain a set of true clauses using concolic evaluation
  - *(new)* use an *inference engine* to *derive* additional logical *conclusion*
  - *(new)* use *unification* to match up clauses from the knowledge base with specs of the existing methods
Example (SetNode.Contains)

```java
interface SetNode {
    constructor Contains (p: int) returns (ret: bool)
    ensures ret = p in elems
}
```

```java
datamodel SetNode {
    invariant elems = {data} + (left != null ? left.elems : {})
    + (right != null ? right.elems : {})
    left != null ==> forall e :: e in left.elems ==> e < data
    right != null ==> forall e :: e in right.elems ==> e > data
}
```

1. \( p = 1 \)
   - Guard: \( \text{left} == \text{null} \&\& \text{right} == \text{null} \)
   - Assignments: \( \text{ret} = (p == \text{data}) \)

2. \( p = 4 \)
   - KB: \( \text{elems} = \{\text{data}\} + \text{left.elems} \)
     \( \text{left.elems} = \{\text{left.data}\} \)
     \( \text{left.data} < \text{data} \)
     \( \text{ret} = \text{p in elems} \)
     \( \text{ret} = \text{p in} \{\text{data}\} + \text{left.elems} \)
     \( \text{false} \)
     \( \text{ret} = \text{p in} \{\text{data}\} \| \text{p in left.elems} \)
     \( \text{ret} = \text{p in left.elems} \)

Recursive Methods
Example (SetNode.Contains)

```java
interface SetNode {
  constructor Contains (p: int) returns (ret: bool)
  ensures ret = p in elems
}
```

```java
datamodel SetNode {
  invariant
  elems = {data} + (left != null ? left.elems : {})
  + (right != null ? right.elems : {})
  left != null ==> forall e :: e in left.elems ==> e < data
  right != null ==> forall e :: e in right.elems ==> e > data
}
```

KB:

```
elems = {data} + left.elems
left.elems = {left.data}
left.data < data
ret = p in elems
ret = p in ({data} + left.elems)
ret = p in left.elems

$ret = p in this.elems (Contains($p))
```

Recruferent Methods
method Contains(n: int) returns (ret: bool)
  requires Valid();
  ensures Valid();
  ensures ret == (n in elems);
{
  if (left != null && right != null) {
    ret := n == data || left. Contains(n) || right. Contains(n);
  } else {
    if (left != null && right == null) {
      ret := n == data || left. Contains(n);
    } else {
      if (right != null && left == null) {
        ret := n == data || right. Contains(n);
      } else {
        ret := n == data;
      }
    }
  }
}
» Domain Specific Rules

\[ e \in (\text{set}_1 + \text{set}_2) \iff (e \in \text{set}_1) || (e \in \text{set}_2) \]

\[ |\text{seq}_1 + \text{seq}_2| \iff |\text{seq}_1| + |\text{seq}_2| \]

\[
(\text{seq}_1 + \text{seq}_2)[\text{idx}] \iff \\
\begin{cases} 
\text{seq}_1[\text{idx}], & \text{when } \text{idx} < |\text{seq}_1| \\
\text{seq}_2[\text{idx} - |\text{seq}_1|], & \text{when } \text{idx} \geq |\text{seq}_1| 
\end{cases}
\]

forall e :: e in seq_1 ⇒ P(e) ⇔
|seq_1| > 0 ⇒ (P(seq_1[0]) ∧ (foralle :: e in seq_1[1.. ] ⇒ P(e)))

Recursive Methods
» Expressiveness

> “Very declarative” specifications cannot be synthesized

```plaintext
constructor Sqrt(p: int) returns (ret: int)
requires p > 0
ensures ret * ret <= p && (ret+1)*(ret+1) > p
```

> Works mostly for specifications with assignments
> Takes advantage of recursively defined specifications

» Synthesized Methods

> No loops (synthesizing loop invariants is a problem); recursion instead
> Not necessarily the most efficient implementation
(e.g. like in Set.Contains()),
→ but still faster than executing the same specification every time
> (currently) Simple read-only queries
» **Sketch** – Armando Solar Lezama [2008]
  > **spec**: a correct (but presumably inefficient) implementation
  > **extras**: a sketch: outlining the control structure of a desired solution
  > **output**: equivalent low-level procedure

» **Storyboard Programming** – Rishabh Singh [2011]
  > **spec**: abstract graphical input/output examples
  > **extras**: a similar sketch of the final solution
  > **output**: low-level procedure that works for the given examples

» **KIDS (Kestrel Interactive Development System)** – Douglas R. Smith [1990]
  > **spec**: high-level logical specification
  > **extras**: much more verbose than pre/post conditions, semi-automated
  > **output**: efficient implementation

**Related Work**
» Finish up implementation for recursive methods

» Further explore the idea of concolic synthesis

» Try to generalize the idea of concolic synthesis to a broader range of (functional) programs

» Formalize the synthesis algorithm

» More examples

» Evaluation and comparison with other tools

Next Steps
» Finish up implementation for recursive methods

» Further explore the idea of concolic synthesis

» Try to generalize the idea of concolic synthesis to a broader range of (functional) programs

» Formalize the synthesis algorithm

» More examples

» Evaluation and comparison with other tools

THANK YOU!