Developing Verified Programs with Boogie and Boogaloo

Nadia Polikarpova

Software Verification

October 16, 2013
Overview

What is Boogie?

The Language: how to express your intention?
  Imperative constructs
  Specification constructs

The Tool: how to get it to verify?
  Debugging techniques
  Boogaloo to the rescue
Overview

What is Boogie?

The Language
- Imperative constructs
- Specification constructs

The Tool
- Debugging techniques
- Boogaloo to the rescue
“Auto-active” verification

all interaction at the program level

- Specification
- Program
-Annotations

Verifier

Logical Formula

Reasoning Engine

Chair of Software Engineering, ETH Zurich
Verifying imperative programs

- Language A
  - Verifier A
- Language B
  - Verifier B
- Language C
  - Verifier C

Control flow & state
...  
Control flow & state, built-in types, framing,...  
Control flow & state
...

Logical Formula
Reasoning Engine

Reuse
Intermediate Verification Language

- Language A
  - Verifier A
- Language B
  - Verifier B
- Language C
  - Verifier C
  - High-level constructs, built-in types and operations, framing, ...
- IVL Program
- IVL Verifier
  - Invariant inference, ...
- Logical Formula I
  - Reasoning Engine I
  - Logical Formula II
  - Reasoning Engine II
  - Logical Formula III
  - Reasoning Engine III
  - Control flow & state
The Boogie IVL

Simple yet expressive
procedures
first-order logic
integer arithmetic

Great for teaching verification!
skills transferable to other auto-active tools

Getting started with Boogie

boogie

Try online [rise4fun.com/Boogie]
Download [boogie.codeplex.com]
User manual [Leino: This is Boogie 2]

Hello, world?
Overview

What is Boogie?

The Language

- Imperative constructs
- Specification constructs

The Tool

- Debugging techniques
- Boogaloo to the rescue
Types

Booleans: `bool`
Mathematical integers: `int`
User-defined: `type` Name $t_1, \ldots, t_n$;

Maps: $[\text{dom}_1, \ldots, \text{dom}_n] \text{range}$

Synonyms: `type` Name $t_1, \ldots, t_n = type$;

Chair of Software Engineering, ETH Zurich
Imperative constructs

Regular procedural programming language

[Absolute Value & Fibonacci]

... and non-determinism

great to simplify and over-approximate behavior

```
havoc x; // assign an arbitrary value to x

if (*) { // choose one of the branches non-deterministically
  statements
} else {
  statements
}
```

```
while (*) { // loop some number of iterations
  statements
}
```
Specification statements: assert

**assert e**: executions in which **e** evaluates to **false** at this point are **bad**

expressions in Boogie are pure, no procedure calls

**Uses**

explaining semantics of other specification constructs
encoding requirements embedded in the source language

```
assert lo <= i && i < hi; // bounds check
result := array[i];
```

```
assert this != null; // 0-0 void target check
call M(this);
```

debugging verification (see later)

[**Absolute Value**]
Specification statements: assume

**assume** \( e \): executions in which \( e \) evaluates to **false** at this point are impossible

```
havoc x; assume x*x == 169; // assign such that
assume true; // skip
assume false; // this branch is dead
```

Uses

explaining semantics of other specification constructs
encoding properties guaranteed by the source language

```
havoc Heap; assume NoDangling(Heap); // managed language
```

debugging verification (see later)

Assumptions are dangerous! [Absolute Value]
Loop invariants

before_statements;
while (c)
    invariant inv;
{
    body;
}
after_statements;

= 

before_statements;
assert inv;

havoc all_vars;
assume inv && c;
body;
assert inv;

havoc all_vars;
assume inv && !c;
after_statements;

The only thing the verifier know about a loop
simple invariants can be inferred

[Fibonacci]
The only thing the verifier knows about a call this is called modular verification

[Abs and Fibonacci]
Enhancing specifications

How do we express more complex specifications?

   e.g. ComputeFib actually computes Fibonacci numbers

Uninterpreted functions

function fib(n: int): int;

Define their meaning using axioms

axiom fib(0) == 0 && fib(1) == 1;
axiom (forall n: int :: n >= 2 ==> fib(n) == fib(n-2) + fib(n-1));

[Fibonacci]
Overview

What is Boogie?

The Language
- Imperative constructs
- Specification constructs

The Tool
- Debugging techniques
- Boogaloo to the rescue
What went wrong?

Specification
Program
Annotations

Boogie
Debugging techniques

Proceed in small steps [Swap]
  use assert statements to figure out what Boogie knows

Divide and conquer the paths
  use assume statements to focus on a subset of executions

Prove a lemma [Non-negative Fibonacci]
  write ghost code to help Boogie reason

Look at a concrete failing test case [Array Max]
  Boogaloo to the rescue!
Getting started with Boogaloo

Try online [cloudstudio.ethz.ch/comcom/#Boogaloo]
Download [bitbucket.org/nadiapolikarpova/boogaloo]
Features

Print directives

\begin{verbatim}
assume { : print “hello, world”, x + y } true;
\end{verbatim}

[Array Max, print the loop counter]

Bound on loop iterations

\begin{verbatim}
--loop-max=N -l=N
\end{verbatim}

N = 1000 by default

[Array Max, comment out loop counter increment]
Conclusions

Boogie is an Intermediate Verification Language (IVL)
IVLs help develop verifiers

The Boogie language consists of:
- imperative constructs ≈ Pascal
- specification constructs ([assert](#), [assume](#), [requires](#), [ensures](#), [invariant](#))
- math-like part (functions + first-order axioms)

There are several techniques to debug a failed verification attempt

Boogaloo helps by generating concrete test cases
Backup slides
How it works: an Example

```java
procedure Test(a: [int]int, x: int)
    requires (forall i: int :: a[i] > i);
{
    if (x == 1000) {
        assert a[x] > 1001;
    }
}
```

Path constraints

- `forall i: int :: a[i] > i`
- `!(x == 1000)`
- `a[x] > 1001`

Valid executions

1: Test(a = [1000 -> 1001], x = 1000) failed
2: Test(a = [1000 -> 1002], x = 1000) passed
3: Test(a = [], x = 0) passed
# Evaluation

<table>
<thead>
<tr>
<th>Program (LOC)</th>
<th>Correct</th>
<th>Buggy</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>N</td>
<td>T (sec)</td>
</tr>
<tr>
<td><strong>Fast</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fibonacci (40)</td>
<td>20</td>
<td>6.4</td>
</tr>
<tr>
<td>TuringFactorial (37)</td>
<td>21</td>
<td>0.2</td>
</tr>
<tr>
<td>ArrayMax (33)</td>
<td>46</td>
<td>0.4</td>
</tr>
<tr>
<td>ArraySum (34)</td>
<td>46</td>
<td>0.3</td>
</tr>
<tr>
<td>BinarySearch (49)</td>
<td>46</td>
<td>0.0</td>
</tr>
<tr>
<td>DutchFlag (96)</td>
<td>20</td>
<td>3.8</td>
</tr>
<tr>
<td>Invert (37)</td>
<td>20</td>
<td>13.3</td>
</tr>
<tr>
<td>BubbleSort (74)</td>
<td>10</td>
<td>6.5</td>
</tr>
<tr>
<td>QuickSort (89)</td>
<td>10</td>
<td>2.0</td>
</tr>
<tr>
<td>QuickSortPartial (79)</td>
<td>10</td>
<td>16.7</td>
</tr>
<tr>
<td>ListTraversal (49)</td>
<td>20</td>
<td>2.5</td>
</tr>
<tr>
<td>ListInsert (52)</td>
<td>7</td>
<td>164.5</td>
</tr>
<tr>
<td><strong>Declarative</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SendMoreMoney (36)</td>
<td>-</td>
<td>0.3</td>
</tr>
<tr>
<td>Primes (31)</td>
<td>8</td>
<td>0.2</td>
</tr>
<tr>
<td>NQueens (37)</td>
<td>15</td>
<td>1.2</td>
</tr>
</tbody>
</table>

*Chair of Software Engineering, ETH Zurich*