Software Verification (Autumn 2015)
Lecture 5: Auto-Active Verification

Chris Poskitt

(based on material by Nadia Polikarpova)
This time last week

\[- \{ x > 0 \} \ x := x+1; \ \text{skip} \ \{ x > 1 \} \]

\[
\frac{\text{[ass]}}{\vdash \{ x + 1 > 1 \} \ x := x + 1 \ \{ x > 1 \}}
\]

\[
\frac{\text{[cons]}}{\vdash \{ x > 0 \} \ x := x + 1 \ \{ x > 1 \}}
\]

\[
\frac{\text{[skip]}}{\vdash \{ x > 1 \} \ \text{skip} \ \{ x > 1 \}}
\]

\[
\vdash \{ x > 0 \} \ x := x + 1; \ \text{skip} \ \{ x > 1 \}
\]
Can we reason about \{pre\}P\{post\} mechanically?

Verification problem **undecidable in general**

How far can we go? What challenges do we face?

- Determining loop invariants
- Weak or missing assertions
- Undecidable assertion logics

... 

**Idea:** automate as much as possible, with users indirectly providing guidance through program-level annotations
“Auto-active” verification

all interaction at the program level

Specification
Program
Annotations

Verifier

Logical Formula

Reasoning Engine
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

**IVL Program**

- High-level constructs, built-in types and operations, framing, ...

**IVL Verifier**

- Invariant inference, ...

- Control flow & state

**Logical Formula I**

- Reasoning Engine I

**Logical Formula II**

- Reasoning Engine II

**Logical Formula III**

- Reasoning Engine III

Chair of Software Engineering, ETH Zurich
The Boogie IVL

Simple yet expressive
procedures
first-order logic
integer arithmetic

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

Viper [http://www.pm.inf.ethz.ch/]
Overview

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

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

The AutoProof Verifier
Overview

The Boogie Language
- Imperative constructs
- Specification constructs

The Boogie Tool
- Debugging techniques
- Boogaloo to the rescue

The AutoProof Verifier
Getting started with Boogie

boogie

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

Hello, world?
Types

Basic types: \texttt{bool, int, real}

User-defined: \texttt{type Name t_1, \ldots, t_n;}
\begin{itemize}
  \item \texttt{type ref; // references}
  \item \texttt{type Person;}
  \item \texttt{type Field t; // fields with values of type t}
\end{itemize}

Maps: \langle t_1, \ldots, t_n \rangle [\text{dom}_1, \ldots, \text{dom}_n] \text{range}
\begin{itemize}
  \item \texttt{[int]int // array of int}
  \item \texttt{[Person]bool // set of persons}
  \item \texttt{[ref][ref // “next” field of a linked list}
  \item \texttt{<t>[ref, Field t]t // generic heap}
\end{itemize}

Synonyms: \texttt{type Name t_1, \ldots, t_n = type;}
\begin{itemize}
  \item \texttt{type Array t = [int]t;}
  \item \texttt{type HeapType = <t>[ref, Field t]t;}
\end{itemize}
Imperative constructs

Regular procedural programming language

[Absolute Value & Fibonacci]

... and non-determinism
great to simplify and over-approximate behavior

```plaintext
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

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

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

debbuging verification (see later)

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

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

```plaintext
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

```plaintext
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

The Boogie Language
  Imperative constructs
  Specification constructs

The Boogie Tool
  Debugging techniques
  Boogaloo to the rescue

The AutoProof Verifier
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
Overview

The Boogie Language
  Imperative constructs
  Specification constructs

The Boogie Tool
  Debugging techniques
  Boogaloo to the rescue

The AutoProof Verifier
AutoProof: a Boogie-based verifier for Eiffel

Translates contract-annotated Eiffel programs to Boogie

Try online [via Comcom]
Manual, tutorial, examples [AutoProof webpage]
How the translation works [Slides]
```plaintext
-- Simple bank account class.
-- Try to fix it and make the verification go through.

class
  ACCOUNT

feature -- Access

  balance: INTEGER
  -- Balance of account.

feature -- Element change

deposit (amount: INTEGER)
  -- Deposit 'amount' on account.
  require
    amount_not_negative: amount >= 0
  do
    balance := balance + amount
  ensure

<table>
<thead>
<tr>
<th>Feature</th>
<th>Line</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACCOUNT (invariant admissibility)</td>
<td></td>
<td>Successfully verified.</td>
</tr>
<tr>
<td>ANY.default_create (creator, inherited by ACCOUNT)</td>
<td></td>
<td>Successfully verified.</td>
</tr>
<tr>
<td>ACCOUNT.deposit</td>
<td></td>
<td>Successfully verified.</td>
</tr>
<tr>
<td>ACCOUNT.withdraw</td>
<td>31</td>
<td>Postcondition balance_decreased may be violated.</td>
</tr>
<tr>
<td>ACCOUNT.transfer</td>
<td>49</td>
<td>Postcondition balance_decreased may be violated.</td>
</tr>
</tbody>
</table>
```
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

AutoProof is one of several auto-active verifiers, based on translating annotated programs to Boogie