A machine checked soundness proof for an intermediate verification language

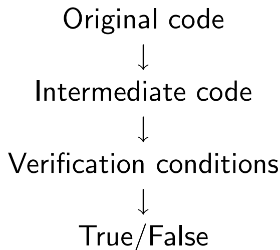
Bart Jacobs Frank Piessens Frédéric Vogels

Katholieke Universiteit Leuven

SOFSEM 2009

Introduction

- Boogie: static verification of program correctness
- Multiple phases
- Each phase must be correct



Contribution

- First part of the paper
  - Focus on VC generation
  - Full formalization
  - Soundness proof
  - Fully machine checked (Coq)
- Second part of the paper
  - Focus on translation to intermediate language
  - Formalization
  - Soundness proof

Mechanized theorem proving

Computer-assisted proving

- Software to assist development of formal proofs
- Automated/interactive theorem proving
- "Guaranteed" correctness

Examples

- HOL
- Isabelle
- Twelf
- Coq

Coq

- Used for our soundness proof
- Developed at INRIA
- Coq allows
  - to define functions or predicates
  - to state mathematical theorems and software specifications
  - to develop interactively formal proofs of these theorems
  - to check these proofs by a relatively small certification "kernel"
  - to extract certified programs
- See
  - Coq'Art (Bertot and Castéran)
  - POPLmark Challenge
  - Types and programming languages

1 Boogie

2 Using Boogie

What is Boogie?

- Goal: static verification
- Designed for imperative/OO languages
- Uses a simple intermediate language (BoogiePL)
- Modular
- Reusable
- Verification condition generation
- Provides extra assistance with property inference
- See Barnett et al., 2006

BoogiePL

- Intermediate language
- Simple, minimal
- More abstract view of the code to be verified
- Describes
  - possible program flows
  - conditions to be fulfilled along the way
  - assumptions that can be made along the way

Structure of BoogiePL

Logical part

- Classical logic
- Defines constants, functions, axioms
- Pre-equipped with integers, finite maps, booleans, ...

Example

const *object* : **name**;
function *superclass*(**name**) **returns** (**name**);
axiom ($\forall$ T : **name** $\bullet$ $T <: superclass(T)$);

const *string* : **name**;
axiom $superclass(string) = object$;

Structure of BoogiePL

Imperative part

- Global variables
- Series of procedure declarations
  - Type signature
  - Pre- and postconditions
- Zero or more implementations for each procedure
- Each implementation must satisfy contract

Example

procedure $abs(n : \mathbf{int}) : (r : \mathbf{int});$
  ensures $(n < 0 \Rightarrow r = -n) \wedge (n \geq 0 \Rightarrow r = n);$

implementation $abs(n : \mathbf{int}) : (result : \mathbf{int})\ \{\dots\}$

BoogiePL$^\flat$ commands

Subset of BoogiePL:

$$
\begin{array}{lcl}
\mathit{Command} & ::= & \textbf{assert } \mathit{Expr} \\
 & | & \textbf{assume } \mathit{Expr} \\
 & | & \textbf{havoc } \mathit{Id} \\
 & | & \mathit{Designator} := \mathit{Expr} \\
 & | & \mathit{Command}; \mathit{Command} \\
 & | & \mathit{Command} \ [] \ \mathit{Command} \\
 & | & \textbf{call } \mathit{Id} := \mathit{Id}(\mathit{Expr}^*) \\
 & | & \mathit{Block}
\end{array}
$$

Difference with BoogiePL: goto (see Barnett and Leino, 2005)

Example: original code to BoogiePL$^\flat$

Example (Original)

```
int abs(int x)
   ensures (x < 0  ==> result = -x)
   ensures (x >= 0 ==> result = x) {
   if ( x < 0 ) return -x;
   else return x;
}
```

Example (BoogiePL$^\flat$ translation)

assume $x < 0$; *result* $:= -x$ [] **assume** $\neg(x < 0)$; *result* $:= x$

Verification conditions

- Weakest precondition $\mathrm{wp}(c, P)$
- P is true after execution of c
- For each procedure: $\mathit{Pre} \Rightarrow \mathrm{wp}(\mathit{body}, \mathit{Post})$
- Our contribution: machine-checked soundness proof

$$
\begin{array}{rcl}
\mathrm{wp}(\textbf{assert } e, P) & = & e \wedge P \\
\mathrm{wp}(\textbf{assume } e, P) & = & e \Rightarrow P \\
\mathrm{wp}(c_1; c_2, P) & = & \mathrm{wp}(c_1, \mathrm{wp}(c_2, P)) \\
\mathrm{wp}(\textbf{havoc } x, P) & = & \forall\, x \bullet P \\
\mathrm{wp}(c_1 \; [] \; c_2, P) & = & \mathrm{wp}(c_1, P) \wedge \mathrm{wp}(c_2, P) \\
\mathrm{wp}(x := e, P) & = & P[e/x]
\end{array}
$$

Example: BoogiePL to verification conditions

Example (BoogiePL code)

assume $x < 0$; $result := -x$ [] **assume** $\neg(x < 0)$; $result := x$

Verification condition

$$\forall\, x \bullet \quad \begin{array}{l}(x < 0 \Rightarrow Post[-x/result]) \wedge \\ (\neg x < 0 \Rightarrow Post[x/result])\end{array}$$

Example: BoogiePL to verification conditions

Example (BoogiePL code)

assume $x < 0$; $result := -x$ [] **assume** $\neg(x < 0)$; $result := x$

Verification condition

$$\forall\, x \bullet \quad \begin{array}{l} (x < 0 \Rightarrow ((x < 0 \Rightarrow -x = -x) \wedge (x \geq 0 \Rightarrow -x = x)))\wedge \\ (\neg x < 0 \Rightarrow ((x < 0 \Rightarrow x = -x) \wedge (x \geq 0 \Rightarrow x = x))) \end{array}$$

Soundness proof

- Show weakest preconditions are correct
- Operational semantics
- Failure states
- If VCs fulfilled, then no failure

Operational semantics: simplification of commands

$$
\begin{array}{rcl}
\mathit{command} & ::= & \textbf{assert}\ \mathit{expression}; \mathit{command} \\
 & | & \textbf{assume}\ \mathit{expression}; \mathit{command} \\
 & | & \textbf{havoc}\ \mathit{identifier}; \mathit{command} \\
 & | & \mathit{identifier} := \mathit{expression}; \mathit{command} \\
 & | & \mathit{command}\ [\,]\ \mathit{command}; \mathit{command} \\
 & | & \textbf{nil}
\end{array}
$$

Differences

- Tree gets linearized
- Blocks gone
- Calls are syntactic sugar

Operational semantics: reduction rules

Assert

$$\frac{e[\mu] = \text{true}}{(\textbf{assert } e; c|\mu) \leadsto (c|\mu)} \qquad \frac{e[\mu] \neq \text{true}}{(\textbf{assert } e; c|\mu) \leadsto \textbf{fail}}$$

Assume

$$\frac{e[\mu] = \text{true}}{(\textbf{assume } e; c|\mu) \leadsto (c|\mu)}$$

Havoc

$$\frac{}{(\textbf{havoc } x; c|\mu) \leadsto (c|\mu, x \mapsto v)}$$

Operational semantics: reduction rules

Assignment

$$\overline{(x := e; c|\mu) \rightsquigarrow (c|\mu, x \mapsto e[\mu])}$$

Choice

$$\overline{(c_1 \; [] \; c_2; c|\mu) \rightsquigarrow (c_1 \oplus c|\mu)} \qquad \overline{(c_1 \; [] \; c_2; c|\mu) \rightsquigarrow (c_2 \oplus c|\mu)}$$

with $\oplus$ the command appending operator.

Soundness proof

Failing

$$\textbf{fails}(c|\mu) \equiv (c|\mu) \rightsquigarrow^* \textbf{fail}$$

Succeeding

$$\textbf{succeeds}(c|\mu) \equiv \neg\textbf{fails}(c|\mu)$$

Soundness

$$\forall\ c, \mu\ \bullet\ \mathrm{wp}(c, \mathrm{true})[\mu] \Rightarrow \textbf{succeeds}(c|\mu)$$

- Coq script available at

  `http://www.cs.kuleuven.be/~frederic/papers/boogie/boogie.v8`

- Technical report available at

  `http://www.cs.kuleuven.be/~frederic/papers/boogie/techreport.pdf`

Soundness proof: focus on choice

By induction on command size

$$c = c_1 [] c_2 ; c'$$

Two applicable reduction rules, one being

$$(c_1 \; [] \; c_2 ; c' | \mu) \rightsquigarrow (c_1 \oplus c' | \mu)$$

It can be proved that

$$|c_1 \oplus c'| < |c_1 \; [] \; c_2 ; c'|$$

Also,

$$\mathrm{wp}(c_1 \; [] \; c_2 ; c', P) \Rightarrow \mathrm{wp}(c_1 \oplus c', P)$$

Induction hypothesis gives

$$\textbf{succeeds}(c_1 \oplus c' | \mu)$$

1 Boogie

2 Using Boogie

Using Boogie

- Second part of paper
- Short example of Boogie usage
- How to translate OO-code into BoogiePL
- Catch typical errors at compile-time
- Prove translation correct

Toy language

Features

- Classes, objects
- No inheritance
- Methods with pre- and postconditions

Errors to catch

- Accessing fields/invoking methods on null
- Invoking methods with invalid arguments (preconditions)
- Exiting methods in an invalid state (postconditions)

Approach to soundness proof

- Define operational semantics $\longrightarrow$
- Preconditions are satisfied prior to non-failing method calls
- Postconditions are satisfied after non-failing method calls
- Define "modular" operational semantics $\curvearrowright$

$$(\mathrm{method\ call}, \mathrm{store}_1, \mathrm{heap}_1) \curvearrowright (\ldots, \mathrm{store}_2, \mathrm{heap}_2)$$

  where store_2 and heap_2 are random but satisfying postconditions.

- $\curvearrowright$ conservatively approximates $\longrightarrow$

Approach to soundness proof

- Consider each method separately
- Relate $\longrightarrow$ and $\curvearrowright$

$$\sigma \longrightarrow^* \textbf{Fail} \Rightarrow \sigma \curvearrowright^* \textbf{Fail}$$

- Relate $\curvearrowright$ with $\rightsquigarrow$

$$\sigma \curvearrowright^* \textbf{Fail} \Rightarrow \sigma \rightsquigarrow^* \textbf{Fail}$$

- Resulting in

$$\begin{aligned} \text{WP are true} &\Rightarrow \neg\,(\sigma \rightsquigarrow^* \textbf{Fail}) \\ &\Rightarrow \neg\,(\sigma \curvearrowright^* \textbf{Fail}) \\ &\Rightarrow \neg\,(\sigma \longrightarrow^* \textbf{Fail}) \end{aligned}$$

Related work

Boogie is used in

- Spec$^\sharp$ (Barnett et al., 2005)
- FreeBoogie
- HAVOC: Heap-Aware Verifier for C Programs
- VCC: A Verifier for Concurrent C (Cohen et al., 2008)
- Dafny: dynamic-frames specifications (Leino, 2008)
- Chalice (Leino, 2009)

Other related work

- Why: another verifier (Filliâtre, 2003)
- Caduceus: verification of C programs (Filliâtre, 2004)
- Krakatoa: verification of Java programs (March et al., 2004)

The End

Questions?