BOOGIE

A MODULAR REUSABLE VERIFIER FOR OBJECT-ORIENTED PROGRAMS

MICROSOFT RESEARCH

Presentation by Itsik Hefez
Boogie is an intermediate verification language, intended as a layer on which to build program verifiers for other languages.

Several program verifiers have been built in: Spec#, the HAVOC and VCC verifiers for C, the Dafny language and verifier, and the concurrent language Chalice.
Previous version BoogiePL, newer version Boogie 2

Boogie is also the name of a tool. The tool accepts Boogie language as input, generates VC’s that are passed to an SMT solver

The Boogie research project is being developed primarily in the RiSE group at Microsoft Research in Redmond
BoogiePL — a verification tool bus

Spec#
HAVOC and VCC
Dafny
...

BoogiePL

Z3
Simplify
Zap 2
...

Z3
Simplify
Zap 2
...

...?
Overview – The Boogie Pipeline

- Source Language
- Intermediate Language
- Inferred Properties
- Verification Conditions
- Theorem Prover
Spec#, VCC, Havoc, Dafny …
VCC transforms C with annotations to BoogiePL
Spec# programs are compiled into CIL
Boogie starts with an AST for this CIL which it gets directly from the compiler or reconstructs from a DLL or EXE
The latter allows batch processing
The former enables a clean integration with MS Visual Studio to provide design-time feedback
Boogie checks for error conditions defined by the VM (array bounds, type cast errors) and user supplied contracts.

Feedback shows up as red underlinings (known as “red squiggles”)

Boogie is the first program verifier to provide such interactive design-time feedback.
class string {
  public int Length { get; private set; } // property
  ...
  public string Substring(int startIndex, int length)
    requires startIndex + length <= this.Length
  {
    ...
  }
}
Overview – Intermediate Language

- Generation of VC’s from source code involves a great number of verifier design decisions.
- Boogie architecture separates the concerns of encoding source language features and their usage rules from the concerns of how to reason about control flow.
- `assume` and `assert` statements encode proof obligations stemming from the source program.
- Properties guaranteed by the source language are encoded as `assume` statements.
VC generation via an intermediate notation was introduced by ESC/Modula-3 and sharpened by ESC/Java.

In ESC properties guaranteed by the source language (background predicates) were produced separately.

BoogiePL innovates further by encoding these predicates as part of the intermediate language.

Other program verifiers can re-use Boogie’s VC generation (by encoding their proof obligations as BoogiePL).

As such, BoogiePL can also be viewed as a high-level front-end to a theorem prover.
BoogiePL programs are turned into first-order verification conditions, a process which requires loop invariants.

These can come from source programs but many can be “boring” or “obvious”.

Thus, manually supplying these as part of the source code provides more clutter than insight.

Sometimes, these are impossible to express in the source language.
Boogie includes a framework for abstract interpretation which can infer loop invariants.

These loop invariants are inserted as `assume` statements into the loop heads.

They are later assumed by the VC to hold at the start of each loop iteration.
Inferred Properties – cont’d

var PreLoopHeap: [ref,name]any;
Start:
  i:=0;
  PreLoopHeap:=Heap;
LoopHead:
goto LoopBody,AfterLoop:
LoopBody:
  assume i<n;
  Heap[e, Example.x] := i;
  …
  i:=i+1;
  goto LoopHead;
AfterLoop:
  …

- The frame condition loop invariant inferred:

\[(\forall o: \text{ref}, f: \text{name} \bullet \\
  (o \neq \text{this} \lor f \neq \text{Example.x}) \Rightarrow \text{Heap}[o, f] = \text{PreLoopHeap}[o, f])\]

- This invariant is not only “boring” to the user, but cannot even be specified at the source level.

- The inferred loop invariants are inserted as assume statements at the beginning of LoopHead
Boogie generates VC’s after generating loop invariants.

There are many equivalent ways of expressing the VC’s and the chosen one can have a dramatic impact on performance.

The VC is represented as a formula in first-order logic and arithmetic.
A VC is produced for every BoogiePL procedure

First, the procedure implementation is transformed into some loop-free code that over-approximates the original loops

Then single assignment transformation is performed, resulting in code without state changes

Finally, for every block A, a boolean variable $A_{ok}$ is defined to be true if every execution starting from A is correct

$Axioms \land BlockEqs \Rightarrow Start_{ok}$

Axioms — conjunction of axioms in the Boogie program
BlockEqs — conjunction of block equations
Verification Conditions – cont’d

- It is possible to reconstruct an error trace from a failed proof
- Typically, a failed proof indicates an error in the program or some missing condition in a contract
- Due to incompleteness in the theorem prover, there is the possibility of spurious error reports
Boogie can generate VC’s for Simplify, Zap, or Z3 (all developed by Microsoft Research).

Boogie initially worked with Simplify, but has shifted towards Z3.

The architecture makes it fairly easy to retarget the final step of the VC generation to a new theorem prover.
Theorem Prover – Z3

- Z3 is a Satisfiability Modulo Theories (SMT) solver.
  - It is an automated satisfiability checker for many typed first-order logic with built-in theories, including support for quantifiers
- The currently supported theories are:
  - equality over free (aka uninterpreted) function and predicate symbols
  - real and integer arithmetic
  - bit-vectors and arrays
  - tuple/records/enumeration types and algebraic (recursive) data-types.
- Z3 checks whether a set of formulas is satisfiable in the built-in theories
- When, the set of formulas is existential then Z3 is a decision procedure: it is always guaranteed to return a correct answer.
- If a set of formulas $F$ is satisfiable, Z3 can produce a model for $F$.
- If a set of formulas contains universal quantifiers, then the model produced by Z3 should be viewed as a potential model, since Z3 is incomplete in this case.
Example: Spec# verifier architecture

Spec# compiler
MSIL ("bytecode")
translator
BoogiePL
V.C. generator
Inference engine
SMT solver
verification condition
“correct” or list of errors
BoogiePL

- An effective intermediate language for VC generation
- Lacks the complexities of a full-featured OO programming language
- As a result, VC generation complexity is distributed over two well-defined phases
BoogiePL – cont’d

- BoogiePL retains (compared with Spec#)
  - Procedures (but not methods)
  - Mutable variables
  - Pre and Post conditions

- Lacks the following complications:
  - Expressions with side effects
  - Heap with objects
  - Classes and interfaces
  - Call-by-reference parameter passing
  - Structured control-flow
Additional features for modeling:
- Constants
- Function symbols
- Axioms
- Non-deterministic control-flow
- Notion of “going wrong”
Translating CIL to BoogiePL

- BoogiePL has no notion of a heap, objects, fields
  - The heap is global two-dimensional array named $Heap$
  - $o.f$ becomes $Heap[o, 0.f]$
- The implicit receiver object of the method is the explicit parameter $this$
- BoogiePL has types and enforces type safety
- Each method is generated to a BoogiePL procedure
  - Boogie differentiates between static and dynamic method calls
BoogiePL declarations

- type
- const
- function
- axiom
- var
- procedure
- implementation
BoogiePL statements

- $x := E$
- $a[i] := E$
- havoc $x$
- assert $E$
- assume $E$
- ;
- call P()
Example: source program

class C : object {
    int x;
    C() { ... }
    virtual int M(int n) { ... }
    static void Main() {
        C c = new C();
        c.x = 12;
        int y = c.M(5);
    }
}
// class types
const unique System.Object: name;
const unique C: name;

axiom C <: System.Object;

function typeof(o: ref) returns (t: name);

// fields
// type field;
const unique C.x: field;
const unique allocated: field;

// the heap
var Heap: [ref, field] int;

class C
    : object {
        int x;
    }
Example: BoogiePL translation (1)

// method declarations
procedure C..ctor(this: ref);
    requires this != null &&
        typeof(this) <: C;
    modifies Heap;

procedure C.M(this: ref, n: int)
    returns (result: int);
    requires this != null &&
        typeof(this) <: C;
    modifies Heap;

procedure C.Main();
    modifies Heap;
Example: BoogiePL translation (2)

```java
// method implementations
implementation C.Main()
{
    var c: ref, y: int;
    C c = new C();

    havoc c;
    assume c != null;
    assume Heap[c, allocated] == 0;
    assume typeof(c) == C;
    Heap[c, allocated] := 1;
    call C..ctor(c);

    assume statements stemming from the source language

    int y = c.M(5);
    call y := C.M(c, 5);
}
```

Objects returned from `new` are guaranteed to not be null
Summary

BoogiePL is an intermediate verification language

- Separates concerns
- Enables sharing in the verification community
  - front ends for multiple languages
  - abstract interpretation
  - VC generation
  - multiple theorem provers
  - predicate abstraction
  - ...
Resources

- **Boogie Website**
  

- **Spec# Website**
  

- **Boogie online version**
  
  http://rise4fun.com/Boogie/McCarthy-91