Program verification using Verification-Condition generation

Shuvendu Lahiri
Software Reliability Research
Microsoft Research, Redmond
Outline

- From Programs to SMT formulas
  - Straight line programs
- Annotations
  - Loops and Procedures
- Modeling low-level C programs
- HAVOC toolkit
- Challenges for scalable and automated verification
Program Correctness: Hoare Triple

- **Hoare triple**
  \[ \{ P \} \text{ } S \{ Q \} \]

  - \( P, Q \) : predicates/property
  - \( S \) : a program

- From a state satisfying \( P \), if \( S \) executes, then either:
  - Either \( S \) does not terminate, or
  - \( S \) terminates in a state satisfying \( Q \)
Program verification → Formula

\{ b.f = 5 \} a.f = 5 \{ a.f + b.f = 10 \}

is valid

iff

Select(f1,b) = 5 \land f2 = \text{Store}(f1,a,5) \implies \text{Select}(f2,a) + \text{Select}(f2,b) = 10

is valid

theory of equality: f, =

theory of arithmetic: 5, 10, +

theory of arrays: Select, Store

• [Nelson & Oppen ’79]
Satisfiability-Modulo-Theory (SMT)

- **Boolean satisfiability solving + theory reasoning**
- Ground theories
  - Equality, arithmetic, Select/Store
- NP-complete logics

- Phenomenal progress in the past few years
  - Yices [Dutretre&deMoura’06],
  - Z3 [deMoura&Bjorner’07]
Simple prog. language (BoogiePL)

- **Commands**
  - `x := E`
    - evaluate E and change x to that value
  - **havoc x**
    - change x to an arbitrary value
  - **assert E**
    - if E holds, terminate; otherwise, go wrong
  - **assume E**
    - if E holds, terminate; otherwise, block
  - `S ; T`
    - execute S, then T
  - **goto A or B;**
    - change point of control to block A or block B, choosing blindly

[Leino et al. ‘05]
Variables and Assignments

- Two types of variables
  - Scalar (e.g. int, bool,..)
  - Maps (mutable arrays, n-dimensional)

```plaintext
var y: int;
var a: [int] int;  // array variable

start:
  y := 10;          // assignment to scalar
  a[y] := 20;       // sugar for a := update(a, y, 20) :
  y := a[10];       //read from a map
```
Example translation: if-then-else

```
[[ if P then S else T end ]]
```

IR

Start:  goto Then or Else

Then:   assume P ;
        [[ S ]] ;
        goto After

Else:   assume \( \neg P \);
        [[ T ]] ;
        goto After

After:   ...

Programs $\rightarrow$ Formulas

- Assume no loops (back-edge gotos)

$\{P\} S \{Q\}$

**Task:** Transform a program with `assume/assert` into a formula

- Known as *Verification Condition* (VC) generation
Verification-condition generation

1. passive features:  assert, assume, ;
2. state changes:  :=, havoc
3. control flow:  goto (no loops)
Weakest (liberal) precondition

- The *weakest liberal precondition* of a statement $S$ with respect to a predicate $Q$ on the post-state of $S$, denoted $\text{wp}(S,Q)$, is the set of pre-states from which execution:
  - does not go wrong (by failing an $\text{assert}$), and
  - if it terminates, terminates in $Q$
VC generation: passive features

- \( \text{wp( assert } E, Q) = E \land Q \)
- \( \text{wp( assume } E, Q) = E \Rightarrow Q \)
- \( \text{wp( } S; T, Q) = \text{wp( } S, \text{wp( } T, Q) \) \)
Eliminate :=, havoc

- Dynamic single assignment (DSA) form
  - There is at most one definition for each variable on each path
  - Replace defs/uses with new incarnations
    - $x := x + 1$ with $x_{n+1} := x_n + 1$
  - At join points unify variable incarnations

- Replace **havoc** $x$ with new incarnations
  $x_{n+1}$

- Eliminate assignments by replacing
  $x := E \Rightarrow \text{assume } x = E$
Example

```plaintext
assume x = 1;
x := x + 1;

if (x = 0) {
    x := x + 2;
} else {
    x := x + 3;
}

assert x = 5;
```

```plaintext
assume x₀ = 1;
assume x₁ = x₀ + 1;

if (x₁ = 0) {
    assume x₂ = x₁ + 2;
    assume x₄ = x₂;
} else {
    assume x₃ = x₁ + 3;
    assume x₄ = x₃;
}

assert x₄ = 5;
```
Example

```
assume x₀ = 1;
assume x₁ = x₀ + 1;

if (x₁ = 0) {
    assume x₂ = x₁ + 2;
    assume x₄ = x₂;
} else {
    assume x₃ = x₁ + 3;
    assume x₄ = x₃;
}
assert x₄ = 5;
```

```
start: assume x₀ = 1; goto l₁;

l₁: assume x₁ = x₀ + 1; goto l₂, l₃;

l₂: assume x₁ = 0;
    assume x₂ = x₁ + 2;
    assume x₄ = x₂; goto l₄;

l₃: assume x₁ ≠ 0;
    assume x₃ = x₁ + 3;
    assume x₄ = x₃; goto l₄;

l₄: assert x₄ = 5
```
Control flow

- Finally, program is a collection of blocks
- Each block is
  - $l$: $A_1, \ldots, A_m$; goto $l_1, \ldots, l_n$
  - $l$ is block label
  - $A$ is either assume $E$ or assert $E$
- Distinguished "start" label
VC Generation for Unstructured Control Flow

For each block $A \equiv l: S; \text{goto } l_{1,..,l_n}$

1. Introduce a boolean variable $A_{ok}$
   - $A_{ok}$ holds iff all executions starting at $A$ do not end in a failed assertion

2. Introduce a Block Equation for each block $A$:

   \[
   \text{BE}_A \equiv A_{ok} \iff \text{VC}(S, \bigwedge_{B \in \text{Succ}(A)} B_{ok})
   \]

VC of entire program:

\[
(\bigwedge_A \text{BE}_A) \Rightarrow \text{Start}_{ok}
\]
VC Generation

A \quad \text{start: assume } x_0 = 1; \text{goto } l_1;

B \quad \begin{aligned} l_1 & : \text{assume } x_1 = x_0 + 1; \text{goto } l_2, l_3; \\ l_2 & : \text{assume } x_1 = 0; \\ & \quad \text{assume } x_2 = x_1 + 2; \\ & \quad \text{assume } x_4 = x_2; \text{goto } l_4; \end{aligned}

C \quad \begin{aligned} l_2 & : \text{assume } x_1 = 0; \\ & \quad \text{assume } x_2 = x_1 + 2; \\ & \quad \text{assume } x_4 = x_2; \text{goto } l_4; \end{aligned}

D \quad \begin{aligned} l_3 & : \text{assume } x_1 \neq 0; \\ & \quad \text{assume } x_3 = x_1 + 3; \\ & \quad \text{assume } x_4 = x_3; \text{goto } l_4; \end{aligned}

E \quad \begin{aligned} l_4 & : \text{assert } x_4 = 5 \\ \Rightarrow \ A_{\text{ok}} \end{aligned}

A_{\text{ok}} \iff (x_0 = 1 \Rightarrow B_{\text{ok}})

B_{\text{ok}} \iff (x_0 = 1 \Rightarrow C_{\text{ok}} \land D_{\text{ok}})

C_{\text{ok}} \iff (x_1 = 0 \Rightarrow 
(x_2 = x_1 + 2 \Rightarrow 
(x_4 = x_2 \Rightarrow E_{\text{ok}})))

D_{\text{ok}} \iff (x_1 \neq 0 \Rightarrow 
(x_2 = x_1 + 2 \Rightarrow 
(x_4 = x_3 \Rightarrow E_{\text{ok}})))

E_{\text{ok}} \iff (x_4 = 5 \land \text{true})

\Rightarrow A_{\text{ok}}
VC Generation

Formula over Arithmetic, Equality, and Boolean connectives

Can be solved by a SMT solver

\[ A_{ok} \iff (x_0 = 1 \Rightarrow B_{ok}) \land \]
\[ B_{ok} \iff (x_0 = 1 \Rightarrow C_{ok} \land D_{ok}) \land \]
\[ C_{ok} \iff (x_1 = 0 \Rightarrow (x_2 = x_1 + 2 \Rightarrow (x_4 = x_2 \Rightarrow E_{ok}))) \land \]
\[ D_{ok} \iff (x_1 \neq 0 \Rightarrow (x_2 = x_1 + 2 \Rightarrow (x_4 = x_3 \Rightarrow E_{ok}))) \land \]
\[ E_{ok} \iff (x_4 = 5 \land \text{true}) \land \]
\[ \Rightarrow A_{ok} \]
Summary: VC Generation

- From programs $\rightarrow$ logical formulas
- Efficient VC generation
  - A formula that is *linear* in the size of the (loop-free and call-free) program
- Language of assertion determines the logical theories for the VC
Loops and procedure calls

- Transform loops to loop-free programs
  - With *loop-invariants*

- Transform procedure calls into asserts and assumes
  - With *preconditions* and *postconditions*
Example translation: loop

[[ while { invariant J } B do S end ]]

LoopHead: assert J ;
goto LoopBody or AfterLoop

LoopBody: assume B ;
[[ S ]] ;
goto LoopHead

AfterLoop: assume ¬B ;
...
Transforming loops

loop head:

assert LoopInv( x ) ;

loop body:

assume Guard( x ) ;

x := ...

after loop:

assume \neg Guard( x ) ;
Transforming loops

assert P

loop

head:
assert LoopInv( x )
assume LoopInv( x )

body:
assume Guard( x )
x := ...

after
loop:
assume ¬Guard( x )
Transforming loops

loop head:

assert LoopInv(x);
assume LoopInv(x);

loop body:

assume Guard(x);
x := ...
assert LoopInv(x);

after loop:
assume ¬Guard(x);
Transforming loops

assert LoopInv( x )

loop head:

assume LoopInv( x )

havoc x ;

assume Guard( x )

loop body:

x := ...

assert LoopInv( x )

after loop:

assume ¬Guard( x )

loop target
Transforming loops

assert LoopInv( x )

havoc x ;
assume LoopInv( x )

assume Guard( x ) ;
x := ...  
assert LoopInv( x )
assume false ;

assume Guard( x ) ;

loop body:

loop head:

after loop:
assume ¬Guard( x ) ;
What about procedure calls?

- Each procedure verified separately
- Procedure calls replaced with their specifications

```
procedure Foo();
requires pre;
ensures post;
modifies M;
```

call Foo();

(assert pre; 
havoc M; 
assume post; 

precondition
postcondition
set of locations possibly modified in Foo
Example of loop invariant + procedure contract

```c
int x, y;  //globals

void main(){
    x := y := 0;
    Foo(4);
    assert (x = 5); // should fail
    assert (x = 4); //should pass
}

void Foo(int n){
    x := 0; y := n;
    invariant ( y >= 0 ∧ x + y = n)
    while(y > 0) {
        x++; y—;
    }
}

requires n >= 0;
ensures x = n;
modifies x;
```

assert (x = 5); // should fail
assert (x = 4); //should pass
Summary

- **VC generation**
  - Generates a SMT formula from loop/call free code

- **Loop invariants**
  - Transform loops to loop-free code

- **Procedure contracts**
  - Transform procedure calls to call-free code
Modeling C

- Convert C to BoogiePL
  - A memory model to model C’s operations
  - Using the HAVOC toolkit

- Generate formulas using Boogie VC generator
C language

C types
- Scalars (int, long, char, short)
- Pointers (int*, struct T*, ..)
- Nested structs and unions
- Array (struct T a[10];)
- Function pointers
- Void *

Difficult to establish type safety in presence of pointer arithmetic, casts
C language (cont.)

Operations
- Arithmetic and Relational (+, -, ≤, =..)
- Pointer arithmetic ((T*) x ++)
- Address-of (&) operation
  - Allows taking address of fields and nested structures
- Allocate and free
- Casts
Memory Model in HAVOC

- Every pointer (address or value) is an integer
- Pointer dereference modeled as lookups(updates) to a map $\text{Mem}$
- Accounts for pointer arithmetic, internal pointers, address-of operations, arrays, casts, …
Memory model

- Maps (in BoogiePL) for memory model

  // Mutable
  **Mem**: int → int
  **Alloc**: int → \{UNALLOCATED, ALLOCATED, FREED\}

  // Immutable
  **Base**: int → int // base address of each pointer
  **Type**: int → type // type of each pointer
C Variable \(\rightarrow\) Boogie Variable

- Variables are *usually* allocated in the heap/Mem
  - To account for the address of a variable (e.g. \(y = \&x\))
  - To deal with structure/union variables
    - \(\text{struct T \{ int f; int g;\} x;}\)
  - All globals are allocated on the heap (make the translation modular)

- Only scalar locals and parameters whose addresses haven’t been taken are mapped to BoogiePL variables
typedef struct {
    int g[10];  // int f;
} DATA;

DATA *create() {
    int a;

    DATA *d = (DATA*) malloc(sizeof(DATA));
    init(d->g, 10, &a);
    d->f = a;
    d->g[1] = 2;
    return d;
}

function f_DATA: int -> int;
for all u: int:: f_DATA(u) = u + 40;

procedure create() returns d:int{
    var @a: int;
    @a := malloc(4);
    d := call malloc(44);
    call init(g_DATA(d), 10, @a);
    Mem[f_DATA(d)] := Mem[@a];
    Mem[g_DATA(d) + 1*4] := 2;
    free(@a);
    return;
}
HAVOC

Assertion checker for low-level (C) systems programs (kernel, device drivers, etc.)

1. **Precise** modeling of the heap
   - Internal pointers, pointer arithmetic, casts, lists, arrays, ..

2. **Expressive** annotation language for C
   1. Deal with linked lists, arrays, types

3. **Automated and efficient** reasoning using SMT solvers + decision procedures for heap
HAVOC Flow

Annotated C program

C → BoogiePL

Memory model

BoogiePL program

Boogie VCGen

Verification condition

SMT Solver (Z3)

Verified

Warning
Challenges

- Expressive yet efficient logic to specify properties
  - Properties about lists, arrays,
  - Yet amenable to automated SMT solvers

- Annotation inference
  - Infer procedure contracts and loop invariants
typedef struct _DATA {
    int data;
    struct _DATA *next;
} DATA, *PDATA;

void InitializeList(PDATA head) {
    PDATA iter;
    iter = head->next;
    while (iter != head) {
        iter->data = 5;
        iter = iter->next;
    }
}

Possible null dereference !!

Need to understand lists
typedef struct _DATA {
    int data;
    struct _DATA *next;
} DATA, *PDATA;

Btwn(next, x, y)
Example (lists)

typedef struct _DATA {
    int data;
    struct _DATA *next;
} DATA, *PDATA;

void InitializeList(PDATA head) {
    PDATA iter;
    iter = head->next;

    invariant(iter ∈ Btwn(next, head->next, head))
    while (iter != head) {
        iter->data = 5;
        iter = iter->next;
    }

    requires (head ∈ Btwn(next, head->next, head))
}
Annotation inference

- Currently semi-automatic
  - Can infer simple annotations
  - Needs user to provide complex annotations

- Houdini algorithm (from ESC/JAVA) [Flanagan&Leino ‘01]
  - User “guesses” a candidate annotation for all the procedures in a module
    - Parameters non-null
    - Locks are unheld on entry and exit
  - The tool prunes away the annotations that may be violated using an SMT solver
    - More examples on HAVOC webpage
Applications of HAVOC

- Functional correctness of medium sized programs with lists and arrays (e.g. custom allocators)
  - POPL’08

- Complex synchronization properties of a core Windows component
  - ~300KLOC, ~1500 procedure
  - 40+ bugs found with only 600L of manual annotations
  - Annotations allow documentation and incremental checking

- Type checking of low-level C code
  - Device drivers ~1KLOC,
HAVOC is available

- Download:
  - [http://research.microsoft.com/projects/HAVOC](http://research.microsoft.com/projects/HAVOC)

- Has more details
  - How to check properties of linked lists and arrays
  - Type checking for C using SMT solvers
  - Case studies
Summary

- **Pros**
  - **Scalable**: Modular checking (one procedure at a time)
  - **Precision**: Abstract only at procedure/loop boundaries
  - **Expressive**: Easy to encode new properties

- **Cons**
  - Requires annotations

- [Research] Need to automate more for a given class of properties
  - E.g. SALINFER [Hacket et al.06] tool for inferring buffer annotations for buffer overrun checking at Microsoft
Next parts: automated techniques

- Loop invariant generation
  - Abstract interpretation
  - Predicate-abstraction with refinement

- Bounded model checking
  - Sacrifice soundness for automation
  - Does not require loop invariants/procedure contracts
Questions?