Symbolic Execution and Verification Conditions

Meeting 13, CSCI 5535, Spring 2009

Survey from Homework 4

• Time spent:
  7.1 hrs (mean), 6 hrs (median)

Fun facts about your classmates:
- Born and raised in Indiana
- "Likes animals more than human beings"
- First programming language was Fortran
- Is a synesthete – visual perception of time

Review of Verification Conditions

Review by Class Participation

• Axiomatic semantics: The meaning of a program is what is true at every
• Hoare triples = Hoare assertion
  \[ E_{A3} \subseteq E_{B3} \]
• Weakest Precondition = "pre that allows part to be done"
  \[ wp(c, B) \subseteq E_{B3} \]
• Verification Condition
  "calc approx of wp"
  "not quite wp"

Not Quite Weakest Preconditions

• Recall what we are trying to do:
  \[
  \begin{array}{ccc}
  \text{false} & \implies & \text{true} \\
  \text{strong} & \uparrow & \text{weak} \\
  \end{array}
  \]
  \[
  \text{A} \quad \text{precondition: } WP(c, B) \\
  \text{verification condition: VC(c, B)}
  \]
• We shall construct a verification condition: VC(c, B)
  - The loops are annotated with loop invariants!
  - VC is guaranteed stronger than WP
  - But hopefully still weaker than A: \[ A \implies VC(c, B) \implies WP(c, B) \]
### VCGen for while

\[ \text{VC(while}_\text{do c, B)} = \text{Inv} \land (\forall x_1, x_n. \text{Inv} \Rightarrow ((b \Rightarrow \text{VC(c, Inv))} \land (\neg b \Rightarrow B))) \]

- **Inv** is the loop invariant (provided externally)
- \(x_1, \ldots, x_n\) are all the variables modified in \(c\)

### Forward VCGen

- Traditionally the VC is computed backwards
  - That's how we've been doing it in class
  - It works well for structured code
- But it can also be computed forward
  - Works even for unstructured languages (e.g., assembly language)
  - Uses **symbolic execution**, a technique that has broad applications in program analysis
    - e.g., the PREfix tool (Intrinsa, Microsoft) works this way

### Plan for Applying VCGen

- Symbolic Execution and Forward VCGen
- Handling **Exponential Blowup**
  - Invariants
  - Dropping Paths
- VCGen For Exceptions (double trouble)
- VCGen For Memory (McCarthyism)
- VCGen For Structures (have a field day)
- VCGen For "Semantic Checksum"

### Simple Assembly Language

- **Consider the language of instructions:**
  \[ I ::= x := e \mid f() \mid \text{if } e \text{ goto } L \mid \text{goto } L \mid L \mid \text{return} \mid \text{inv e} \]
- The "inv e" instruction is an annotation
  - Says that boolean expression \(e\) holds at that point
- Each function \(f()\) comes with \(\text{Pre}_f\) and \(\text{Post}_f\) annotations (\(\text{pre-}\) and \(\text{post-}\) conditions)
- New Notation (yay!): \(I_k\) is the instruction at address \(k\)

### Symbolic Execution Symbolic State

- We set up a symbolic execution state:
  \[ \Sigma : \text{Var} \rightarrow \text{SymbolicExpressions} \]
  \[ \Sigma(x) = \text{the symbolic value of } x \text{ in state } \Sigma \]
  \[ \Sigma[x := e] = \text{a new state where } x \text{'s value is } e \]
- We use states as substitutions:
  \[ \Sigma(e) = e \text{ where } x \text{ replaced by } \Sigma(x) \]
- So far, much like operational semantics
Symbolic Execution Invariant State

- The symbolic executor keeps track of the encountered invariants
- A new part of symex state: Inv \(\subseteq \{1...n\}\)
- If \(k \in \text{Inv}\) then \(I_k\) is an invariant instr. that we have already executed
- Basic idea: execute an \text{inv} instruction only twice:
  - The first time it is encountered
  - Once more time around an arbitrary iteration

Symbolic Execution Rules

- Define a VC function as an interpreter:
  \[ VC : \text{Address} \times \text{SymbolicState} \times \text{InvariantState} \rightarrow \text{Assertion} \]

\[ VC(k, \Sigma, \text{Inv}) = \begin{cases} \Sigma(e) \land V(a_1...a_n, \Sigma'(\text{Post}_1)) \Rightarrow VC(k+1, \Sigma', \text{Inv} \cup \{k\}) & \text{if } I_k = \text{if } e \text{ goto } L \text{ (like a function call)} \\ \Sigma(e) \land V(a_1...a_n, \Sigma'(\text{Post}_1)) \Rightarrow VC(k+1, \Sigma', \text{Inv} \cup \{k\}) & \text{if } I_k = f() \\ \Sigma(e) \land V(a_1...a_n, \Sigma'(\text{Post}_1)) \Rightarrow VC(k+1, \Sigma', \text{Inv} \cup \{k\}) & \text{if } I_k = \text{return} \\ \end{cases} \]

Symbolic Execution Invariants

Two cases when seeing an invariant instruction:

1. We see the invariant for the first time
   - \(I_k = \text{inv } e\)
   - \(k \notin \text{Inv}\) ("not in the set of invariants we've seen")
   - Let \(\{y_1, ... y_m\}\) be the variables that could be modified on a path from the invariant back to itself
   - Let \(a_1, ..., a_n\) be fresh new symbolic parameters
   \[ VC(k, \Sigma, \text{Inv}) = \Sigma(e) \land V(a_1...a_n, \Sigma(\text{Post}_1)) \Rightarrow VC(k+1, \Sigma', \text{Inv} \cup \{k\}) \]

2. We see the invariant for the second time
   - \(I_k = \text{inv } E\)
   - \(k \in \text{Inv}\)
   \[ VC(k, \Sigma, \text{Inv}) = \Sigma(e) \]

Some tools take a more simplistic approach
- Do not require invariants
- Iterate through the loop a fixed number of times
- PREfix, versions of ESC (DEC/Compaq/HP SRC)
- Sacrifice soundness for usability

Symbolic Execution Top-Level

- Let \(x_1, ..., x_n\) be the variables and \(a_1, ..., a_n\) fresh params
- Let \(\Sigma_0\) be the state \([x_1 := a_1, ..., x_n := a_n]\)
- Let \(\emptyset\) be the empty Inv set
- For all functions \(f\) in your program, prove:
  \[ \forall a_1...a_n. \Sigma_0(\text{Pre}_1) = VC(f_{entry}, \Sigma_0, \emptyset) \]
  - If you start the program by invoking any \(f\) in a state that satisfies \(\text{Pre}_1\), then the program will execute such that
    - At all \text{inv} 's the \(e\) holds, and
    - If the function returns then \(\text{Post}_1\) holds

- Can be proved w.r.t. a real interpreter (operational semantics)
- Or via a proof technique called co-induction (or, assume-guarantee)
Forward VCGen Example

Precondition: \( x \leq 0 \)

Loop: inv \( x \leq 6 \)
- if \( x > 5 \) goto End
- \( x := x + 1 \)
goto Loop

End: return

Postcondition: \( x = 6 \)

∀\( x \). \( x \leq 0 \) \( \implies \) \( x \leq 6 \lor \)
\( \forall x'. (x' \leq 6 \implies x' > 5 \implies x' = 6 ) \)

• VC contains both proof obligations and assumptions about the control flow

VCs Can Be Large

• Consider the sequence of conditionals
  
  (if \( x < 0 \) then \( x := -x \))
  (if \( x \leq 3 \) then \( x += 3 \))

  - With the postcondition \( P(x) \)
    
  • The VC is
    
    \[
    \begin{align*}
    x < 0 & \land -x \leq 3 \implies P(-x + 3) \\
    x < 0 & \land -x > 3 \implies P(-x) \\
    x \geq 0 & \land x \leq 3 \implies P(x + 3) \\
    x \geq 0 & \land x > 3 \implies P(x )
    \end{align*}
    \]

  • There is one conjunct for each path
  
  ⇒ exponential number of paths!

  - Conjects for infeasible paths have unsatisfiable guards!

  • Try with \( P(x) = x \geq 3 \)

VCs Can Be Exponential

• VCs are exponential in the size of the source because they attempt relative completeness:
  
  - Perhaps the correctness of the program must be argued independently for each path

  • Unlikely that the programmer wrote a program by considering an exponential number of cases
    
    - But possible. Any examples? Any solutions?

VCs Can Be Exponential. Solutions?

• VCs are exponential in the size of the source because they attempt relative completeness:
  
  - Perhaps the correctness of the program must be argued independently for each path

  • Unlikely that the programmer wrote a program by considering an exponential number of cases
    
    - But possible. Any examples? Any solutions?

  1) Ignore paths - arb. (unsoundness)
VCs Can Be Exponential. Solutions?

- VCs are exponential in the size of the source because they attempt relative completeness:
  - Perhaps the correctness of the program must be argued independently for each path

- Standard Solutions:
  - Allow invariants even in straight-line code
  - And thus do not consider all paths independently!

Invariants in Straight-Line Code

- Purpose: modularize the verification task
- Add the command "after c establish Inv"
  - Same semantics as c (Inv is only for VC purposes)
  
  \[ \text{VC}(\text{after } c \text{ establish } \text{Inv}, P) = \text{def} \]

  \[ \text{VC}(c, \text{Inv}) \land \forall x_i. \text{Inv} \Rightarrow P \]

Dropping Paths

- Without annotations, we can drop some paths
- VC(if E then c₁ else c₂, P) = choose one of
  - E \Rightarrow \text{VC}(c₁, P) \land \neg E \Rightarrow \text{VC}(c₂, P) (drop no paths)
  - E \Rightarrow \text{VC}(c₁, P) (drops "else" path!)
  - \neg E \Rightarrow \text{VC}(c₂, P) (drops "then" path!)

- We sacrifice soundness! (we are now unsound)
  - No more guarantees
  - Possibly still a good debugging aid

- Remarks:
  - A recent trend is to sacrifice soundness to increase usability (e.g., Metal, ESP, even ESC)
  - The PREfix tool considers only 50 non-cyclic paths through a function (almost at random)

VCGen for Exceptions

- Extend the language with exceptions without arguments (cf. HW2):
  - throw throws an exception
  - try c₁ catch c₂ executes c₂ if c₁ throws

- Problem:
  - We have non-local transfer of control
  - What is VC(throw, P) ?

- Solutions?
VCGen for Exceptions

- Extend the language with exceptions without arguments (cf. HW2):
  - throw throws an exception
  - try \( c_1 \) catch \( c_2 \) executes \( c_2 \) if \( c_1 \) throws

- Problem:
  - We have non-local transfer of control
  - What is \( VC(\text{throw}, P) \)?

- Standard Solution: use 2 postconditions
  - One for normal termination
  - One for exceptional termination

VCGen for Exceptions

- \( VC(c, P, Q) \) is a precondition that makes \( c \) either not terminate, or terminate normally with \( P \) or throw an exception with \( Q \)

- Rules:
  - \( VC(\text{skip}, P, Q) = P \)
  - \( VC(c_1; c_2, P, Q) = VC(c_1, VC(c_2, P, Q), Q) \)
  - \( VC(\text{try } c_1 \text{ catch } c_2, P, Q) = VC(c_1, P, VC(c_2, P, Q)) \)

VCGen Try-Finally

- Given these:
  - \( VC(c_1; c_2, P, Q) = VC(c_1, VC(c_2, P, Q), Q) \)
  - \( VC(\text{try } c_1 \text{ catch } c_2, P, Q) = VC(c_1, P, VC(c_2, P, Q)) \)

- Finally is somewhat like "if":
  - \( VC(\text{try } c_1 \text{ finally } c_2, P, Q) = VC(c_1, VC(c_2, P, Q), \text{true}) \land VC(c_1, \text{true}, VC(c_2, Q, Q)) \)

- Which reduces to:
  - \( VC(c_1, VC(c_2, P, Q), VC(c_2, Q, Q)) \)
Hoare Rules and the Heap

- When is the following Hoare triple valid?
  \[ \{ A \} \; *x := 5 \{ *x + *y = 10 \} \]

- The Hoare rule for assignment would give us:
  \[ [5/*x][*x + *y = 10] = 5 + *y = 10 = *y = 5 \] (we lost one case)

Why didn’t this work?

Handling The Heap

- We do not yet have a way to talk about memory (the heap, pointers) in assertions

- Model the state of memory as a symbolic mapping from addresses to values:
  - If \( A \) denotes an address and \( M \) is a memory state then:
    - sel\( (M, A) \) denotes the contents of the memory cell
    - upd\( (M, A, V) \) denotes a new memory state obtained from \( M \) by writing \( V \) at address \( A \)

More on Memory

- Allow variables to range over memory states
  - Can quantify over all possible memory states
- Use the special pseudo-variable \( \mu \) (mu) in assertions to refer to the current memory
- Example:
  \[ \forall i. \; i \geq 0 \land i < 5 \Rightarrow sel(\mu, A + i) > 0 \]
says that entries 0..4 in array \( A \) are positive

Hoare Rules: Side-Effects

- To model writes we use memory expressions
  - A memory write changes the value of memory
    \[ \{ B[upd(\mu, E_1, E_2)/\mu] \} *E_1 := E_2 \{ B \} \]

- Important technique: treat memory as a whole
- And reason later about memory expressions with inference rules such as (McCarthy, ~67):

  \[ sel(upd(M, A_1, V), A_2) = \begin{cases} V & \text{if } A_1 = A_2 \\ sel(M, A_2) & \text{if } A_1 = A_2 \end{cases} \]

Memory Aliasing

- Consider again:
  \[ \{ A \} \; *x := 5 \{ *x + *y = 10 \} \]
- We obtain:
  \[ A = [\text{upd}(\mu, x, 5)/\mu] \{ *x + *y = 10 \} = [\text{upd}(\mu, x, 5)/\mu] \{ sel(\mu, x) + sel(\mu, y) = 10 \} \]

  (1) \[ = sel(upd(\mu, x, 5), x) + sel(upd(\mu, x, 5), y) = 10 \]

  = 5 + sel(upd(\mu, x, 5), y) = 10

  = if \( x = y \) then 5 + 5 = 10 else 5 + sel(\mu, y) = 10

  (2) \[ = x = y \text{ or } *y = 5 \]

  Up to (1) is theorem generation
  - From (1) to (2) is theorem proving
Alternative Handling for Memory

• Reasoning about aliasing can be expensive
  – It is NP-hard (and/or undecidable)
• Sometimes completeness is sacrificed with the following (approximate) rule:
  \[
  \text{sel}(\text{upd}(M, A_1, V), A_2) = \\
  \begin{cases}
  V & \text{if } A_1 = (\text{obviously}) A_2 \\
  \text{sel}(M, E_3) & \text{if } A_1 = (\text{obviously}) A_2 \\
  p & \text{otherwise (if } p \text{ is a fresh new parameter)}
  \end{cases}
  \]
• The meaning of “obvious” varies:
  • The addresses of two distinct globals are ≠
  • The address of a global and one of a local are ≠
• “PREfix” and GCC use such schemes

VCGen Overarching Example

• Consider the program
  - Precondition: B : bool ∧ A : array(bool, L)
  1: I := 0
  2: R := B
  3: inv I ≥ 0 ∧ R : bool
     if I ≥ L goto 9
     assert saferd(A + I)
     T := *(A + I)
     I := I + 1
     R := T
     goto 3
  9: return R
  - Postcondition: R : bool

VCGen Overarching Example

VA, VB, VL, VP
B : bool ∧ A : array(bool, L) ⇒
  0 ≥ 0 ∧ B : bool ∧
  ∀I. ∀R.
   I ≥ 0 ∧ R : bool ⇒
   I ≥ L ⇒ R : bool ∧
   I < L ⇒ saferd(A + I) ∧
   I + 1 ≥ 0 ∧
   \text{sel}(\mu, A + I) : bool

• VC contains both proof obligations and assumptions about the control flow

Mutable Records - Two Models

• Let r : RECORD { f1 : T1; f2 : T2 } END
• For us, records are reference types
• Method 1: one “memory” for each record
  – One index constant for each field
  – r.f1 is sel(r,f1) and r.f1 := E is r := upd(r,f1,E)
• Method 2: one “memory” for each field
  – The record address is the index
  – r.f1 is sel(f1,r) and r.f1 := E is f1 := upd(f1,r,E)
• Only works in strongly-typed languages like Java
  – Fails in C where &r.f2 = &r + sizeof(T1)

VC as a “Semantic Checksum”

• Weakest preconditions are an expression of the program’s semantics:
  – Two equivalent programs have logically equivalent WPs
  – No matter how different their syntax is!
• VC are almost as powerful
VC as a “Semantic Checksum”

- Consider the “assembly language” program to the right:
  - $x := 4$
  - $x := (x == 5)$
  - `assert x : bool`
  - $x := \neg x$
  - `assert x`

- High-level type checking is not appropriate here
- The VC is: $((4 == 5) : \text{bool}) \land (\neg (4 == 5))$
- No confusion from reuse of $x$ with different types

Invariance of VC Across Optimizations

- VC is so good at abstracting syntactic details that it is syntactically preserved by many common optimizations
  - Register allocation, instruction scheduling
  - Common subexpression elimination, constant and copy propagation
  - Dead code elimination
- We have identical VCs whether or not an optimization has been performed
  - Preserves syntactic form, not just semantic meaning!
- This can be used to verify correctness of compiler optimizations (Translation Validation)

VC Characterize a Safe Interpreter

- Consider a fictitious “safe” interpreter
  - As it goes along it performs checks (e.g. “safe to read from this memory addr”, “this is a null-terminated string”, “I have not already acquired this lock”)
  - Some of these would actually be hard to implement
- The VC describes all of the checks to be performed
  - Along with their context (assumptions from conditionals)
  - Invariants and pre/postconditions are used to obtain a finite expression (through induction)
- VC is valid $\Rightarrow$ interpreter never fails
  - We enforce same level of “correctness”
  - But better (static + more powerful checks)

VC Big Picture

- Verification conditions
  - Capture the semantics of code + specifications
  - Language independent
  - Can be computed backward/forward on structured/unstructured code
  - Make Axiomatic Semantics practical

Invariants Are Not Easy

- Consider the following code from QuickSort:
  ```c
  int partition(int *a, int L, int H, int pivot) {
      int L = L, H = H;
      while(L < H) {
          while(a[L] < pivot) L ++;
          while(a[H] > pivot) H --;
          if(L < H) { swap a[L] and a[H] }
      }
      return L
  }
  ```
- Consider verifying only memory safety
- What is the loop invariant for the outer loop?

Questions?
Verification conditions make axiomatic semantics practical. We can compute verification conditions forward for use on unstructured code (= assembly language). This is sometimes called symbolic execution.

We can add extra invariants or drop paths (dropping is unsound) to help verification condition generation scale.

We can model exceptions, memory operations and data structures using verification condition generation.