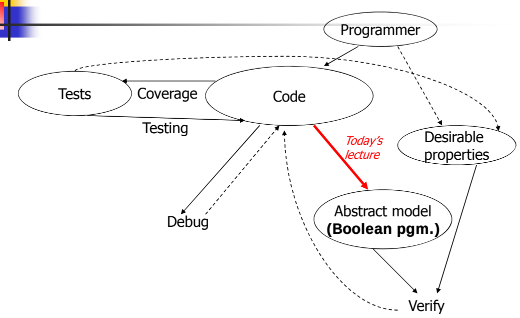
Software Abstractions

Abhik Roychoudhury
CS 5219
National University of Singapore

No model may be available



Recap on Model Checking

- Inputs:
  - A finite state transition system M
  - A "temporal" property φ
- Check M $|= \varphi$
- Output
  - True if M $|= \varphi$
  - Counter-example evidence, otherwise

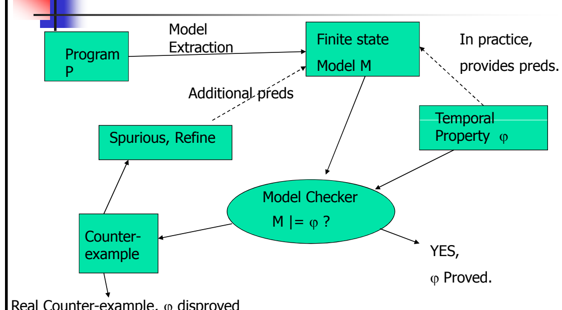
Model Checking for SW Verif.

- The steps:
  - Generate transition system-like models from code
    - Typically involves at least data abstractions
  - Exhaustive search through the model
    - For time/space efficiency, the model may not be explicitly represented and searched.
  - Explaining counter-examples

More on the big picture

- Explaining counter-example
  - Counter-example points to an actual violation of property φ in program.
    - How to locate the bug from the counter-example – SW Engineering activity
  - It was introduced owing to the abstractions
    - Refine the abstraction and run model checking on the model derived by refined abstraction
    - Abstract → Model Check →Refine loop.

Abst -> MC -> Refine



The approach (1)

- Reasoning techniques over finite-state models well-understood.
  - Search based procedures (Model Checking)
- Need to generate models from code
  - Typically finitely many control locations
  - Infinitely many data states (memory store)
- How to abstract the memory store ?
  - This can give a finite state model

The approach (2)

- Boolean abstraction used on memory store
  - State of memory captured by finitely many boolean variables which answer queries about its contents
- Check all possible behaviors of a program
  - Translate program to a finite state model and employ model checking (this lecture)
  - OR Modify the state space search algorithm in model checking to directly verify programs
    - e.g. Verisoft checker from Bell Labs (not covered in this course)

Model Generation Projects

- Source Language → Modeling Language
- E.g. C → PROMELA (FeaVer tool)
- C → Boolean Pgm (SLAM toolkit)
- Various choices in Bandera toolkit
- In this lecture, we consider a
  - source language with sequential programs
  - Properties are locational invariants
    - AG((pc = 34) ⇒ (v = 0))

Predicate Abstraction

- Input
  - Source Program P
  - S_P, Set of Predicates about variables in P
- Output
  - Abstracted program P1
  - Data states in P1 correspond to valuations of predicates in S_P

Predicate Abs. (once more)

- Input :
  - A C program P1
  - A set of predicates containing vars of P1
- Output
  - A boolean program P2
    - Only data type of P2 is "boolean"
    - P2 contains more execution paths than P1 i.e.
      - All paths of P1 are captured in P2, not vice-versa
      - P2 is being used for invariant verification of P1.

The Language of Predicates

- Boolean expressions containing program variables,
  - No function calls
  - Pointer referencing is allowed
    - P→val > Var
  - Of course Bool. Exp contains
    - B = B ∧ B | B ∨ B | ¬ B | A Relop A
    - A = A + A | A – A | A*A | A/A | Var | Int
    - Relop = < | > | ≤ | ≥ | ≠ | =

Simple Examples

- Source Code
- Var := 0

- Var := Var1

- Abstracted Code
- [Var = 0] := true
- [Var = 1] := false

- [Var = 0] := unknown
- *(no preds. about Var1)*
- *OR-*
- [Var= 0] := [Var1= 0]
- *(Var1=0 is another pred)*

Control constructs

- Abstraction scheme will be developed for
  - Within a procedure
    - Assignments
    - Branches
    - All other constructs can be represented by these
  - Across procedures
    - Formal and actual parameters
    - Local variables
    - Return variables

Assignments to predicates

- We are converting a C program to a "boolean" program where the only type is boolean.
  - The boolean program will not be executed.
- Assignment to our predicate variables can assign
  - true / false / unknown
  - If "unknown" is assigned, both possibilities should be explored during model checking

Assignments

- Predicate abstraction of pgm. P w.r.t. $\{ b_1,\ldots,b_k \}$
- Effect of X := e on $b_1,\ldots,b_k$
- Variable b_i denotes expression φ_i
- If $\varphi_i[x \to e]$ holds before X := e then set
  - b_i := true
- If $\neg\varphi_i[x \to e]$ holds before X := e then set
  - b_i := false

Simple Ex. of Assignments

- b1 ≡ X > 2 b2 ≡ Y > 2
- Assignment X := Y
- Transform it to
  - b1 := b2

- b1 ≡ X > 2 b2 ≡ Y > 2 b3 ≡ X < 3 b4 ≡ Y < 3
- Transform X := Y to the parallel assignment
  - b1, b3 := b2, b4

Assignments – (2)

- But $\varphi_i[x \to e]$ may not be representable as a boolean formula over $b_1,\ldots,b_k$
- Examples:
  - Predicates: X < 5, X = 2
  - Assignment stmt: X := X + 1
  - X < 5 [X →X+1] equivalent to X +1 < 5 equivalent to X < 4
  - X = 2 [X →X+1] equivalent to X + 1 = 2 equivalent to X = 1

Assignments – (3)

- Define predicate b1 as X < 5
- b2 as X = 2
- What is the weakest formula over b1 and b2 which implies X < 4 ?
- If this formula is true, we can conclude
  - X < 4 before X := X +1 is executed
  - X < 5 after X := X + 1 is executed
  - b1 = true after X := X + 1 is executed

Assignments - Summary

- Predicates: $\{b_1,\ldots,b_k\}$
- Predicate b_i represents expression φ_i
- X := e is an assignment statement in the pgm. being abstracted.
- We can conclude b_i = true after X := e iff φ_i[X →e] before X :=e is executed.

Assignments - Summary

- Find the weakest formula over b1,…,bk which implies φ_i[X →e] and check whether it is true before X := e
- If yes, set b_i = true as an effect of X := e in the abstracted program
- Set b_i = false in the abstracted pgm if the weakest formula over b1,…,bk which implies $\neg\varphi_i$[X →e] holds
- If none of this is possible, b_i = unknown

Assignments - Example

- Predicates: b1 is X < 5, b2 is X =2
- Assignment: X := X + 1
- Weakest pre-condition for b1 to hold, denoted as WP(X:= X+1, b1)
  - X < 4
- Weakest formula over {b1, b2} to imply WP(X:= X+1, b1), denoted as F(WP(X := X +1), b1))
  - X = 2, that is, the formula b2

Assignments Example

- Predicates: b1 is X < 5, b2 is X =2
- WP(X:= X+1, ¬b1) equivalent to X + 1 ≥ 5 equivalent to X ≥ 4
- F(WP(X:= X+1, ¬b1)) = F(X ≥ 4) is
  - X ≥ 5, that is, the formula ¬b1 itself
- Computation of the F function is in general exponential, Why ??

Computation of F(φ)

- Consider all minterms of b1,…,bk
  - ¬b1 ∧ ¬b2
  - ¬b1 ∧ b2
  - b1 ∧ ¬b2
  - b1 ∧ b2
- Which of them imply φ ?
- Take the disjunction of all such minterms and simplify. Improvements to this algo. possible.

Exercise

- b1 ≡ X < 5 , b2 ≡ X = 2
- Assignment in the program
  - X := X + 1
- What will it be substituted with in our "boolean" program ?
  - Let us do it now

Aliasing via pointers

- To compute the effect of X := 3 on b1
  - We compute F(WP(X := 3, b1))
  - Suppose b1 is *p > 5, p is a pointer
- Effect of X := 3 depends on whether
  - X and p are aliases
  - Use a "points-to" analysis to determine this.
    - Typically flow insensitive
  - Aliasing analysis sharpens information about program states and hence the abstraction.

Effect of aliasing

- WP(X := 3, *p > 5) is
  - $(\&x = p \wedge 3 > 5) \vee (\&x \neq p \wedge *p > 5)$
- Thus, WP(X := e, φ(Y)) is
  - $(\&X = \&Y \wedge \varphi[Y \rightarrow e]) \vee (\&x \neq \&Y \wedge \varphi(Y)$
  - If X and Y are aliases replace Y by e in φ
  - Otherwise, the assignment has no effect
- If φ refers to several locations, each of them may/may not alias to X.

Another exponential blowup

- If φ refers to k locations
  - Each may/not alias to X
  - 2^k possibilities
  - WP is a disjunction of 2^k minterms
- In practice, accurate static not-points-to analysis is feasible
  - Removes conjuncts corresponding to confirmed non-aliases (in any control loc.)

Control constructs

- Abstraction scheme will be developed for
  - Within a procedure
    - Assignments
    - Branches
    - All other constructs can be represented by these
  - Across procedures
    - Formal and actual parameters
    - Local variables
    - Return variables

Control branches

- So far, considered straight-line code.
- Consider the effect of conditional branch instructions as in if-then-else statements.
- Loops are conditional branch instructions with one branch executing a goto.
- Sufficient to consider
  - Abstract(If (c) {S1} else {S2})

Control Branches

- If (c) {S1} else {S2}
- ⇑⇓
- If (*) { assume (c) ; S1 } else
- { assume (¬c); S2 }
- (*) denotes non-deterministic choice
- assume(φ) terminates exec. if φ is false
  - Otherwise, the statement has no effect.

Different from the *assert* statement

Abstracting Branches

- Abstract(If (c) {S1} else {S2}) is
  - If (*) { assume G(c); Abstract(S1) }
  - else { assume G(¬c); Abstract(S2)}
- Predicates: $b_1,\ldots,b_k$
- G(c) is the strongest formula over $b_1,\ldots,b_k$ which is implied by c
  - Formal definition in next slide.

Abstracting Branches

- G(c) = ¬ F (¬ c)
  - Dual of the F operator studied earlier
- CAUTION: G and F operators of this lecture different from temporal ops
- Exercise: Why choose the G operator for abstracting branches, why not F ?

Questions

- Abstract(if (c) {S1} else {S2})
- ⇑⇓
- If G(c) { Abstract(S1)} else {Abstract(S2)}
- Was the assume statement necessary Does the assume statement introduce new paths ?

Abstracting Branches-Example

- If (*p <= x) {*p := x} else {*p := *p + x}
- Predicates
  - b1 is *p <= 0
  - b2 is x = 0
- G(*p <= x) = ¬ F (*p > x)
- To compute F (*p > x) consider all minterms of b1 and b2

Abstracting Branches-Example

- Minterms of b1, b2
  - ¬b1 ∧ ¬b2 is *p > 0 ∧ x ≠ 0
  - b1 ∧ ¬b2 is *p <= 0 ∧ x ≠ 0
  - ¬b1 ∧ b2 is *p >0 ∧ x = 0
  - b1 ∧ b2 is *p <= 0 ∧ x = 0
- F(*p > x) = ¬b1 ∧ b2
  - &x and p are considered to be non-aliases

Abstracting Branches- Example

- G(*p <= x) = ¬ F(*p > x)= ¬(b2∧¬b1)
  = ¬b2 ⋁ b1 = b2 ⇒ b1
  = (x = 0) ⇒ (*p <= 0)
- Similarly compute G(¬(*p <= x))
- Abstracted template
  - If (*) { assume (x = 0 ⇒ (*p <= 0)) ; … }
  - else { assume (x=0 ⇒ ¬(*p <=0)); … }

Control constructs

- Abstraction scheme will be developed for
  - Within a procedure
    - Assignments
    - Branches
    - All other constructs can be represented by these
  - Across procedures
    - Formal parameter, Local variables, Return variables
    - Procedure calls and returns

Inter-procedural Abstraction

- One-to-one mapping of procedure
  - Each proc. to an abstract one
  - No inlining introduced by abstraction.
- Given predicates: b1,…,bk
  - Each pred. is marked global (refers to global vars.) or local to a specific procedure.
  - Does not allow capturing relationships of variables across procedures. Will Revisit this!

Abstracted procedures ?

- Given
  - A concrete procedure R
  - A set E_R of predicates b1,…,bj specific to R
  - E_R can refer to parameters of R
- Need to define an abstract procedure R1
  - Formal Parameters of R1
  - Return Vars. of R1

Example

```
int  procedure(int* q, int  y)
{
int l1, l2;
…..
…..
return l1;
}
```

Predicates:

b1 is y >= 0
b2 is *q <= y
b3 is y = l1
b4 is y > l2

Parameters, Local Vars

- Formal parameters of R1
  - All predicates in E_R which do not refer to local variables of R
- All other preds. in E_R are local vars. of R1.
- Natural notion of *input context* for R1.
- Example:
  - Concrete Parameters: q, y
  - Abstract Parameters: y>=0, *q <= y

Return Variables

- Natural notion of *output context* for R1. Pass information to callers about
  - Return value of R
  - Global Vars
  - Call-by-reference parameters …
- Info. about return value captured by those preds in E_R which refer to return var. of R, but no other local variable (return var. can be a local var.)

Return Variables

- Info about global var/reference parameters
  - Preds. in E_R which were computed to be formal parameters of R1, AND
  - Refer to global variables, dereferences
- E_R = { y>=0, *q <=y, y = l1, y > l2 }
  - Concrete ret. Var. : l1
  - Concrete Parameters: q, y
  - Abst. Ret. Vars: *y =l1, *q <= y*

Control constructs

- Abstraction scheme will be developed for
  - Within a procedure
    - Assignments
    - Branches
    - All other constructs can be represented by these
  - Across procedures
    - Formal parameter, Local variables, Return variables
    - Procedure calls and returns

Procedure Calls

- So far, abstraction of a single procedure
  - Assignments (with aliasing)
  - Branches (if-then-else, loops)
  - Formal Parameters
  - Local and global variables
  - Return variables
- Use input/output contexts in procedure call/return in inter-procedural abstraction.

Passing Parameters

- Take any formal parameter predicate b of R1

```
Void main()
{
…
r = procedure(p, x);
}
```

```
int procedure(int *q, int y){
int l1, l2;
…
return l1;
}
```

All predicates of "procedure" :
- y >= 0
-*q <= y
-y= l1
-y > l2

Formal parameter preds. of procedure
-y >= 0
-*q < = y

Passing Parameters

- Replace formals by actuals in b.
  - y >= 0 is a formal parameter pred.
  - After replacement, it becomes x >= 0
- If F(b[formals →actuals)) holds during procedure invocation of the boolean pgm, then pass *true* to the parameter b
- If F(¬b[formals →actuals)) holds, then pass *false* to parameter b
- Otherwise, pass *unknown*.

Exercise

- Work out the **boolean expressions** passed to the two parameters of *procedure* in our example shown before
- Use the definition of the F operator given earlier and the abst. predicates given.

Procedure Returns

- If procedure S calls procedure R, and
  - S1/R1 are abstractions of S/R
  - b1,…,bj are abstract ret. Vars of R1
- Then S1 has j corresponding local boolean vars. which will be updated by call to R1.
- Do the local preds. in S need to be updated ? **YES**

Procedure returns

- These local preds. of S can refer to
  - Concrete Return var. for R
  - Global Vars (along with other local vars)
- For each such pred b, again compute F(b) and F(¬b) to decide the value of b.
- The function F is computed w.r.t
  - Set of abstraction preds (under the carpet ☺

Procedure returns

- To compute the effect of return from R into S (calling procedure), compute F w.r.t.
  - Return predicates of R
    - (Capture effect on global vars/return vars/ref.)
  - Predicates of S which do not need to be updated.
- An implicit partitioning of the preds of S !!
- **Self Study: This portion in the reading.**

Reading(s)

- *Automatic Predicate Abstraction of C Programs*
  - Ball, Majumdar, Millstein, Rajamani
  - PLDI 2001.
- Also useful: *Polymorphic Predicate Abstraction*
  - MSR Tech Rep. by same set of authors.

Reading Exercise

- Currently, the predicates used for abstraction can only contain program variables. Is this a restriction ?
  - What about values returned by procedures and/or passed by parameters ?
  - Can we track such values by introducing new names ? We can have preds like
    - Ret_value_of_v = Passed_value_of_v + 1