Recap on Model Checking
- Inputs:
  - A finite state transition system $M$
  - A “temporal” property $\varphi$
- Check $M \models \varphi$
- Output
  - True if $M \models \varphi$
  - Counter-example evidence, otherwise

More on the big picture
- Explaining counter-example
  - Counter-example points to an actual violation of property $\varphi$ 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 $\rightarrow$ Model Check $\rightarrow$Refine loop.

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

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
    - Always (pc = 34) ⇒ (v = 0)

What kind of model?

- Modeling languages typically do not support
  - Dynamic heap allocation/ de-allocation
  - Call Stack of Procedure Activation Records
- Restriction relaxed in SLAM toolkit
  - Allows for models with procedures
  - Invariant checking of such models by adapting existing inter-procedural dataflow analysis algorithms [Sharir & Pnueli 1981]

Predicate Abstraction

Input
- Source Program P
- \( S_p \), Set of Predicates about variables in P

Output
- Abstracted program \( P_1 \)
- Data states in \( P_1 \) correspond to valuations of predicates in \( S_p \)

The Language of Predicates

- Boolean expressions containing program variables,
  - No function calls
  - Pointer referencing is allowed
    - \( P\rightarrow\text{val} > \text{Var} \)
  - Of course Bool. Exp contains
    - \( B = B \land B \lor B \neg B \) | A Relop A
    - \( A = A + A \land A - A \land A \ast A \land A / A \land \text{Var} \land \text{Int} \)
    - Relop = \( < \lor > \lor > \lor \neq \lor = \)
**Simple Examples**

- **Source Code**
  - `Var := 0`
  - `Var := Var1`

- **Abstracted Code**
  - `[Var = 0] := true`
  - `[Var = 1] := false`
  - `[Var = 0] := unknown` (no preds. about `Var1`)
  - `[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. `{b1, ..., bk}`
- Effect of `X := e` on `b1, ..., bk`
- Variable `bi` denotes expression `ϕi`
- If `ϕi[X → e]` holds before `X := e` then set
  - `bi := true`
- If `¬ϕi[X → e]` holds before `X := e` then set
  - `bi := 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 `ϕ[X → e]` may not be representable as a boolean formula over `b1, ..., bk`
- 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 \( b_1 \) as \( X < 5 \)

\[ b_2 \text{ as } X = 2 \]

What is the weakest formula over \( b_1 \) and \( b_2 \) 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
- \( b_1 \) = true after \( X := X + 1 \) is executed

Assignments - Summary

Predicates: \( \{b_1, \ldots, b_k\} \)

Predicate \( b_i \) represents expression \( \phi_i \)

\( X := e \) is an assignment statement in the pgm. being abstracted.

We can conclude \( b_i \) = true after \( X := e \) iff \( \phi_i[X \rightarrow e] \) before \( X := e \) is executed.

Assignments - Example

Predicates: \( b_1 \) is \( X < 5 \), \( b_2 \) is \( X = 2 \)

Assignment: \( X := X + 1 \)

Weakest pre-condition for \( b_1 \) to hold, denoted as \( \text{WP}(X := X + 1, b_1) \)

- \( X < 5 \)

Weakest formula over \( \{b_1, b_2\} \) to imply \( \text{WP}(X := X + 1, b_1) \), denoted as \( F(\text{WP}(X := X + 1, b_1)) \)

- \( X = 2 \), that is, the formula \( b_2 \)

Assignments Example

Predicates: \( b_1 \) is \( X < 5 \), \( b_2 \) is \( X = 2 \)

\( \text{WP}(X := X + 1, \neg b_1) \) equivalent to \( X + 1 \geq 5 \) equivalent to \( X \geq 4 \)

\( F(\text{WP}(X := X + 1, \neg b_1)) = F(X \geq 4) \) is

- \( X \geq 5 \), that is, the formula \( \neg b_1 \) itself

Computation of the \( F \) function is in general exponential, Why ??

Computation of \( F(\phi) \)

Consider all minterms of \( b_1, \ldots, b_k \)

- \( \neg b_1 \land \neg b_2 \)
- \( \neg b_1 \land b_2 \)
- \( b_1 \land \neg b_2 \)
- \( b_1 \land b_2 \)

Which of them imply \( \phi \) ?

Take the disjunction of all such minterms and simplify. Improvements to this algo. possible.
**Exercise**
- \( b1 \equiv X < 5 \), \( b2 \equiv X = 2 \)
- 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
  - \((^x = p \land 3 > 5) \lor (^x \neq p \land ^*p > 5)\)
- Thus, \( WP(X := e, \varphi(Y)) \) is
  - \((^x = ^Y \land \varphi[Y \rightarrow e]) \lor (^x \neq ^Y \land \varphi(Y))\)
  - If \( X \) and \( Y \) are aliases replace \( Y \) by \( e \) in \( \varphi \)
  - Otherwise, the assignment has no effect
  - If \( \varphi \) refers to several locations, each of them may/may not alias to \( X \).

**Another exponential blowup**
- If \( \varphi \) 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 \( (c) \{ S_1 \} \) else \( \{ S_2 \} \)

- If \( (*) \{ \text{assume} (c) \; \{ S_1 \} \} \) else
  \( \{ \text{assume} (\neg c) \; \{ S_2 \} \} \)

- \( (*) \) denotes non-deterministic choice
- \text{assume}(\varphi) \) terminates exec. if \( \varphi \) is false
- Otherwise, the statement has no effect.

Abstracting Branches

- \text{Abstract}( \text{If} \ (c) \{ S_1 \} \text{else} \{ S_2 \} ) \)

- If \( (*) \) \{ \text{assume} \ G(\ c) ; \text{Abstract}(S_1) \} \) else
  \{ \text{assume} \ G(\neg\ c) ; \text{Abstract}(S_2) \} \)

- 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

- \( G(c) = \neg F(\neg c) \)

- 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

- \text{Abstract(}\text{if} \ (c) \{ S_1 \} \text{else} \{ S_2 \} \)

- If \( (*) \) \{ \text{assume} \ G(\ c) ; \text{Abstract}(S_1) \} \) else
  \{ \text{Abstract}(S_2) \}

- Was the assume statement necessary

- Does the assume statement introduce new paths?

Abstracting Branches - Example

- If \( (*p \leq x) \{ *p := x \} \) else \{ *p := *p + x \}

- Predicates
  - \( b_1 \) is \( *p \leq 0 \)
  - \( b_2 \) is \( *p = 0 \)

- \( G(*p \leq x) = \neg F(*p > x) \)

- To compute \( F(*p > x) \) consider all minterms of \( b_1 \) and \( b_2 \)

Abstracting Branches - Example

- Minterms of \( b_1, b_2 \)
  - \( \neg b_1 \land \neg b_2 \) is \( *p > 0 \land x \neq 0 \)
  - \( b_1 \lor \neg b_2 \) is \( *p \leq 0 \lor x \neq 0 \)
  - \( \neg b_1 \lor b_2 \) is \( *p \geq 0 \lor x = 0 \)
  - \( b_1 \lor b_2 \) is \( *p \leq 0 \lor x = 0 \)

- \( F(*p > x) = \neg b_1 \lor b_2 \)

- &x and \( p \) are considered to be non-aliases
Abstracting Branches-

Example

- \( G(*p \leq x) = \neg F(*p > x) = \neg (b2 /\neg b1) = \neg b2 \lor b1 = b2 \Rightarrow b1 = (x = 0) \Rightarrow (*p \leq 0) \)
- Similarly compute \( G((\neg (*p \leq x)) \)

Abstracted template

- If (*) { assume \((x = 0 \Rightarrow (*p \leq 0)) \); …
  }
- else { assume \((x=0 \Rightarrow (\neg (*p \leq 0)))\); … }

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: \( b1, \ldots, bk \)
  - 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 \( b1, \ldots, bj \) 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 \( y \geq 0 \)
- \( b2 \) is \( *q \leq y \)
- \( b3 \) 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 \textit{input context} for \( R1 \).
- Example:
  - Concrete Parameters: \( q, y \)
  - Abstract Parameters: \( y \geq 0, *q \leq y \)
Return Variables

- Natural notion of **output context** for R1. Pass info. 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.)

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

\[ E_R = \{ y \geq 0, \,*q \leq y, \, y = l_1, \, y > l_2 \} \]
- Concrete ret. Var. : \( l_1 \)
- Concrete Parameters: \( q, y \)
- Abst. Ret. Vars: \( y = l_1, \,*q \leq y \)

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

```c
void main()
{
    int procedure(int *q, int y);
    int l1, l2;
    ...  \( r = \text{procedure}(p, x); \)
    ...  \text{return} l1;
    }
}
```

All predicates of "procedure":
- \( y \geq 0 \)
- \( *q \leq y \)
- \( y = l_1 \)
- \( y > l_2 \)

- Formal parameter preds. of procedure
- \( y \geq 0 \)
- \( *q = y \)

- Replace formals by actuals in \( b \).
  - \( y \geq 0 \) is a formal parameter pred.
  - After replacement, it becomes \( x \geq 0 \)
- If \( F(b[\text{formals} \rightarrow \text{actuals}) \) holds during procedure invocation of the boolean pgm, then pass **true** to the parameter \( b \)
- If \( F(\neg b[\text{formals} \rightarrow \text{actuals}]) \) 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 😊)

Reading(s)

- **Automatic Predicate Abstraction of C Programs**
  - Ball, Majumdar, Millstein, Rajamani
- 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