

COMP6210 Automated Software Verification

Symbolic Model Checking

Pavel Naumov

Intended Learning Outcomes

By the end of this lecture, you will be able to

- model systems with variables using Boolean functions
- model Boolean programs using Boolean functions
- understand how simple correctness properties can be verified using a symbolic representation of programs

Symbolic Modelling of Systems

Symbolic Model Checking

Modelling Software (Reminder)

- Running software has:
 - **data**, partitioned into variables, each taking values from a finite set; this includes:
 - a program counter per thread
 - variables on stack
 - variables on heap
 - **instructions** (sequential or concurrent) transforming data over time.
- We saw already how to model data and instructions using transition systems.

Explicit versus Symbolic Modelling of Software Systems

- transition systems model software systems **explicitly**
 - each node represents a system state
 - each edge represents an atomic state change
- this suffers from **state space explosion**
 - number of states grows exponentially with number of variables
 - a system with n Boolean variables can have 2^n states
 - approximately 10^{78} atoms in universe!
- transition systems are inadequate to model systems with more than a few variables
- in contrast, **symbolic model checking** avoids explicitly constructing the state space of a system, and instead represents it as a formula in propositional logic
 - this allows for **compact representations**, e.g. using **Binary Decision Diagrams** (BDDs)

Boolean Connectives and Notation

- boolean connectives: \neg (not), \vee (or), \wedge (and), \rightarrow (implies), \leftrightarrow (if and only if)
- $V = \{v_1, \dots, v_n\}$ set of boolean variables
- $\bigwedge_{j=1, \dots, n} (v_j) : v_1 \wedge v_2 \wedge \dots \wedge v_n$
- $\bigvee_{j=1, \dots, n} (v_j) : v_1 \vee v_2 \vee \dots \vee v_n$
- $\bigwedge_{j=1, \dots, n, j \neq i} (v_j) : v_1 \wedge \dots \wedge v_{i-1} \wedge v_{i+1} \wedge \dots \wedge v_n$

Symbolic Representation of Systems

- finite set of **variables** $V = \{x_1, \dots, x_n\}$, over a finite domain D , to encode system states (nodes)
- describe the **initial states** with a formula over the variables in V :

$$Init(x_1, \dots, x_n)$$

- represent **transitions** with a formula over the variables in V and the variables x'_1, \dots, x'_n (which are a copy of the variables in V):

$$Trans(x_1, \dots, x_n, x'_1, \dots, x'_n)$$

- variables x_1, \dots, x_n describe the state *before* the transition
- variables x'_1, \dots, x'_n describe the state *after* the transition

Example: Noughts and Crosses

					X			X			X		X	X	X	X	X
									O				O	O		O	O
		X	O	X	O	X	O		X	O		X	O		X	O	X

Initial configurations:

Init: $(x_1 = -) \wedge (x_2 = -) \wedge \dots \wedge (x_9 = -) \wedge (t = A \vee t = B)$

(all cells are empty and either player *A* or player *B* can start)

Example: Noughts and Crosses

				X			X			X			X	X	X
								O						O	O
		X		O		X	O		X	O			X	O	X

Transition relation:

(either player A or player B can move, if it is her turn, and mark one of the empty cells)

Trans : $\bigvee_{i=1,\dots,9} (\text{Move}(A, i) \vee \text{Move}(B, i))$

- Move*(A, i) :

$$(t = A \wedge t' = B) \wedge (x_i = - \wedge x'_i = X) \wedge \bigwedge_{j=1,\dots,9, j \neq i} (x'_j = x_j)$$

(it's A's turn to move and B will move next; cell i was empty and now it is marked with X; all other cells remain unchanged)

- Move*(B, i) is similar.

• 公式:

$$\text{Move}(A, i) : (t = A \wedge t' = B) \wedge (x_i = - \wedge x'_i = X) \wedge \bigwedge_{j=1,\dots,9, j \neq i} (x'_j = x_j)$$

• 解释:

- $t = A \wedge t' = B$: 这表示当前轮到A 进行移动, 执行完之后轮到B 移动。
- $x_i = - \wedge x'_i = X$: 表示第 i 个格子在移动前是空的 (-), 在移动后被 A 标记为 "X"。
- $\bigwedge_{j=1,\dots,9, j \neq i} (x'_j = x_j)$: 表示其他所有不等于 i 的格子状态都保持不变。

• 总结:

该公式描述了玩家 A 的一次有效移动: 如果轮到 A, 并且 A 在某个空格 i 上标记 "X", 之后轮到 B, 所有其他格子的状态保持不变。

Example: Noughts and Crosses

					X			X			X		X	X	X	X	X	X
									O			O		O	O		O	O
		X	O		X	O		X	O		X	O		X	O		X	X

Winning condition for player A:

WinA : $(x_1 = X \wedge x_2 = X \wedge x_3 = X)$
 $\vee (x_1 = X \wedge x_5 = X \wedge x_9 = X)$
 $\vee \dots$

How to Represent a Program Symbolically?

```
int x,y = 0,0;  
while (true)  
    x,y = (x+1) mod 3, x+1;
```

- variables: x, y with domain **int** (bounded!)
- initial state:

$$\textit{Init} : (x = 0) \wedge (y = 0)$$

- transition relation:

$$\textit{Trans} : (x' = (x + 1) \bmod 3) \wedge (y' = x + 1)$$

Boolean Programs

- in the previous examples, the domains for the variables are **not** Boolean
- to encode *Init* and *Trans* as *propositional logic formulas*, we would need to encode each program variable using *several* boolean variables
- to avoid doing this (for convenience), we restrict to **Boolean programs**:
 - only Boolean variables
 - only ASSIGNMENT, IF, and WHILE statements
 - no procedure calls

How to Represent a Boolean Program?

begin

L_1 : $x_1 = \text{false};$

L_2 : **while** $(x_1 \wedge x_2)$ **do**

L_3 : $x_1 = \text{true};$

L_4 : **endwhile;**

L_5 : **if** $(x_1 \vee x_2)$ **then**

L_6 : $x_1 = x_1 \leftrightarrow x_2;$

L_7 : **else**

L_8 : $x_2 = x_1 \leftrightarrow x_2;$

L_9 : **endif**

L_{10} : **end**

(program counters added as before)

How to Represent a Boolean Program?

4个初始状态

begin

```
 $L_1$  :  $x_1 = \text{false};$   
 $L_2$  : while  $(x_1 \wedge x_2)$  do  
 $L_3$  :  $x_1 = \text{true};$   
 $L_4$  : endwhile;  
 $L_5$  : if  $(x_1 \vee x_2)$  then  
 $L_6$  :  $x_1 = x_1 \leftrightarrow x_2;$   
 $L_7$  : else  
 $L_8$  :  $x_2 = x_1 \leftrightarrow x_2;$   
 $L_9$  : endif  
 $L_{10}$  : end
```

- variables:

$$V = \{x_1, x_2, PC\}$$

- domain:

$$D = \{F, T, L_1, \dots, L_{10}\}$$

(still not Boolean!)

- transitions:

$$Trans = ?$$

Encoding assignments

$$L_j : x_i = b(x_1, \dots, x_n);$$
$$L_{j+1} : \dots$$

(b is a boolean formula over the program variables)

Trans_j :

$PC = L_j \wedge PC' = L_{j+1}$ (update program counter)

$\wedge x'_i \leftrightarrow b(x_1, \dots, x_n)$ (update variable x_i)

$\wedge \bigwedge_{j=1, \dots, n, j \neq i} (x'_j = x_j)$ (copy all the other variables)

Encoding if-then-else statements

L_i : **if** ($b(x_1, \dots, x_n)$) **then**
 L_{i+1} : first statement then-body
...
 L_j : **else**
 L_{j+1} : first statement else-body
...
 L_k : **endif**
 L_{k+1} : ...

(b is a boolean formula over the program variables)

$$\begin{aligned} \text{Trans}_i : & \left((PC = L_i \wedge PC' = L_{i+1} \wedge b(x_1, \dots, x_n)) \right. \\ & \vee (PC = L_i \wedge PC' = L_{j+1} \wedge \neg b(x_1, \dots, x_n)) \\ & \quad \vee (PC = L_j \wedge PC' = L_{k+1}) \quad \text{(update PC)} \\ & \quad \vee (PC = L_k \wedge PC' = L_{k+1}) \left. \right) \\ & \quad \wedge \bigwedge_{j=1, \dots, n} (x'_j = x_j) \quad \text{(copy variables)} \end{aligned}$$

Encoding while statements

L_i : **while** ($b(x_1, \dots, x_n)$) **do**
 L_{i+1} : first statement while-body
...
 L_j : **endwhile**
 L_{j+1} : ...

(b is a boolean formula over the program variables)

$Trans_i$: (($PC = L_i \wedge PC' = L_{i+1} \wedge b(x_1, \dots, x_n)$) (enter while)
 $\vee (PC = L_i \wedge PC' = L_{j+1} \wedge \neg b(x_1, \dots, x_n))$ (exit while)
 $\vee (PC = L_j \wedge PC' = L_i)$ (back to condition)
 $\wedge \bigwedge_{j=1, \dots, n} (x'_j = x_j)$ (copy variables)

How to Represent a Boolean Program?

begin

L_1 : $x_1 = \text{false};$

L_2 : **while** $(x_1 \wedge x_2)$ **do**

L_3 : $x_1 = \text{true};$

L_4 : **endwhile;**

L_5 : **if** $(x_1 \vee x_2)$ **then**

L_6 : $x_1 = x_1 \leftrightarrow x_2;$

L_7 : **else**

L_8 : $x_2 = x_1 \leftrightarrow x_2;$

L_9 : **endif**

L_{10} : **end**

$\text{Trans}(x_1, x_2, PC, x'_1, x'_2, PC') =$

$(PC = L_1 \wedge PC' = L_2 \wedge x'_1 = F \wedge x'_2 = x_2)$

$\vee ((PC = L_2 \wedge PC' = L_3 \wedge (x_1 \wedge x_2))$

$\vee (PC = L_2 \wedge PC' = L_5 \wedge \neg(x_1 \wedge x_2))$

$\vee (PC = L_4 \wedge PC' = L_2)$

)

$\wedge (x'_1 = x_1) \wedge (x'_2 = x_2))$

$\vee (PC = L_3 \wedge PC' = L_4 \wedge (x'_1 = T) \wedge (x'_2 = x_2))$

$\vee ((PC = L_5 \wedge PC' = L_6 \wedge (x_1 \vee x_2))$

$\vee (PC = L_5 \wedge PC' = L_8 \wedge \neg(x_1 \vee x_2))$

$\vee (PC = L_7 \wedge PC' = L_{10})$

$\vee (PC = L_9 \wedge PC' = L_{10})$

)

$\wedge (x'_1 = x_1) \wedge (x'_2 = x_2))$

$\vee (PC = L_6 \wedge PC' = L_7 \wedge (x'_1 = (x_1 \leftrightarrow x_2)) \wedge (x'_2 = x_2))$

$\vee (PC = L_8 \wedge PC' = L_9 \wedge (x'_2 = (x_1 \leftrightarrow x_2)) \wedge (x'_1 = x_1))$

Encoding Program Counters using Boolean Variables

- representation on the previous slide is still not a Boolean formula!
- **Question:** how do we encode $PC = L_i$ as a boolean formula?
- **Answer:**
 - assume the values of the program counter (L_1, \dots, L_{10} in the previous example) can be represented using n bits (4 in the example)
 - use Boolean variables pc_0, \dots, pc_{n-1} to encode the value of the program counter
 - e.g. use variables pc_0, pc_1, pc_2, pc_3 to encode integers $i \in \{1, \dots, 10\}$ (and thus the formula $PC = L_i$)
 - if i is represented as $c_{n-1} \dots c_0$ in binary, $PC = L_i$ can be encoded using the formula $(pc_0 \leftrightarrow c_0) \wedge \dots \wedge (pc_{n-1} \leftrightarrow c_{n-1})$
 - for $i = 6$ (i.e. 0110 in binary), the formula $PC = L_6$ is encoded as $(pc_3 \leftrightarrow 0) \wedge (pc_2 \leftrightarrow 1) \wedge (pc_1 \leftrightarrow 1) \wedge (pc_0 \leftrightarrow 0)$

Note: similar encodings can be done for programs with variables of types other than Boolean!

Encoding Program Counters using Boolean Variables

- representation on the previous slide is still not a Boolean formula!
- **Question:** how do we encode $PC = L_i$ as a boolean formula?
- **Answer:**
 - assume the values of the program counter (L_1, \dots, L_{10} in the previous example) can be represented using n bits (4 in the example)
 - use Boolean variables pc_0, \dots, pc_{n-1} to encode the value of the program counter
 - e.g. use variables pc_0, pc_1, pc_2, pc_3 to encode integers $i \in \{1, \dots, 10\}$ (and thus the formula $PC = L_i$)
 - if i is represented as $c_{n-1} \dots c_0$ in binary, $PC = L_i$ can be encoded using the formula $(pc_0 \leftrightarrow c_0) \wedge \dots \wedge (pc_{n-1} \leftrightarrow c_{n-1})$
 - for $i = 6$ (i.e. 0110 in binary), the formula $PC = L_6$ is encoded as $(pc_3 \leftrightarrow 0) \wedge (pc_2 \leftrightarrow 1) \wedge (pc_1 \leftrightarrow 1) \wedge (pc_0 \leftrightarrow 0)$

Note: similar encodings can be done for programs with variables of types other than Boolean!

Symbolic Modelling of Systems

Symbolic Model Checking

Model Checking

- verification of (simple) safety properties: can we ever reach a "bad" state?
- verification of temporal properties (e.g. LTL)

Example

Check if the following program can ever reach a state satisfying $x \geq y$:

```
int x,y = 0,0;
while (true)
    x,y = (x+1) mod 3, x+1;
```

- initial states: $Init(x, y) : x = 0 \wedge y = 0$
- transition relation:

$$Trans(x, y, x', y') : (x' = (x + 1) \bmod 3) \wedge (y' = x + 1)$$

- states reachable in one step from the initial state:

$$\begin{aligned} Reach_1(x', y') &= \exists x. \exists y. Init(x, y) \wedge Trans(x, y, x', y') \\ &= \exists x. \exists y. x = 0 \wedge y = 0 \wedge x' = (x + 1) \bmod 3 \wedge y' = x + 1 \\ &= x' = 1 \wedge y' = 1 \end{aligned}$$

Example (Cont'd)

- states reachable in one step from the initial state:

$$\text{Reach}_1(x', y') = x' = 1 \wedge y' = 1$$

- states reachable in **two** steps from the initial state:

$$\begin{aligned}\text{Reach}_2(x', y') &= \exists x. \exists y. \text{Reach}_1(x, y) \wedge \text{Trans}(x, y, x', y') \\ &= \exists x. \exists y. x = 1 \wedge y = 1 \wedge x' = (x + 1) \bmod 3 \wedge y' = x + 1 \\ &= x' = 2 \wedge y' = 2\end{aligned}$$

- states reachable in *three* steps from the initial state:

$$\text{Reach}_3(x', y') = \dots = x' = 0 \wedge y' = 3$$

- at each step i (incl. $i = 0$), we can check if $\text{Reach}_i(x, y) \wedge (x < y)$ is *satisfiable*!

Computing Reachable States

Inputs:

- $Init(x_1, \dots, x_n)$ (boolean formula for initial states)
- $Trans(x_1, \dots, x_n, x'_1, \dots, x'_n)$ (boolean formula for transitions)

Output: $Reach(x_1, \dots, x_n)$ (boolean formula for reachable states)

For $R(x_1, \dots, x_n)$ (boolean formula encoding a set of states), define:

$$Next(R) := (\exists x_1 \dots \exists x_n. R(x_1, \dots, x_n) \wedge Trans(x_1, \dots, x_n, x'_1, \dots, x'_n)) [x_1/x'_1, \dots, x_n/x'_n]$$

Algorithm (in which variables are formulas over variables x_1, \dots, x_n):

```
Reach = ff; //empty set of states
R = Init; //set of initial states
while ( $\neg (R \rightarrow Reach)$ ) { //R  $\not\subseteq$  Reach
  Reach = Reach  $\vee$  R; //add R to Reach;
  R = Next(R); //formula for successors of states in R
}
```

Checking Safety Properties

Input: safety property $\phi(x_1, \dots, x_n)$ (system invariant)

- e.g. $(x_1 \vee x_2) \wedge \neg(x_1 \wedge x_2)$

Output: Does the system described by $Init(x_1, \dots, x_n)$ and $Trans(x_1, \dots, x_n, x'_1, \dots, x'_n)$ satisfy the given property?

Algorithm:

1. Compute reachable states, as a formula $Reach(x_1, \dots, x_n)$
2. Check if $Reach(x_1, \dots, x_n) \wedge \neg\phi(x_1, \dots, x_n)$ is *satisfiable* (i.e. it can be satisfied by an assignment of values to x_1, \dots, x_n)
3. If Yes, return "Safety property violated".
If No, return "Success".

Checking Safety Properties

Input: safety property $\phi(x_1, \dots, x_n)$ (system invariant)

- e.g. $(x_1 \vee x_2) \wedge \neg(x_1 \wedge x_2)$

Output: Does the system described by $Init(x_1, \dots, x_n)$ and $Trans(x_1, \dots, x_n, x'_1, \dots, x'_n)$ satisfy the given property?

Algorithm:

1. Compute reachable states, as a formula $Reach(x_1, \dots, x_n)$
2. Check if $Reach(x_1, \dots, x_n) \wedge \neg\phi(x_1, \dots, x_n)$ is *satisfiable* (i.e. it can be satisfied by an assignment of values to x_1, \dots, x_n)
3. If Yes, return "Safety property violated".
If No, return "Success".

Checking LTL Properties (Basic Idea)

Similar idea to explicit-state model checking for LTL:

- system described using $Trans(x_1, \dots, x_n, x'_1, \dots, x'_n)$
- negation of LTL property ($\neg\phi$) described as **tableau**
(boolean formula which encodes *paths* that satisfy $\neg\phi$)
- conjoin (using \wedge) the transition relations of
 - (i) the system
 - (ii) the tableauto find counter-example...
- algorithm computes the set of states from which a path satisfying $\neg\phi$ exists

BDDs and Symbolic Model Checking

- **binary decision diagrams** (BDDs) are a canonical form to represent Boolean functions
 - often **more compact** than traditional normal forms
 - can be manipulated efficiently
- reachable state space can be represented as a BDD
- property verification uses iterative computations on the reachable state space

Summary

- encoding systems with Boolean variables as boolean formulas
- encoding (Boolean) programs as boolean formulas
- symbolic model checking of simple properties

A Word on the State of the Art (**not Examinable!**)

Counter-example-Guided Abstraction Refinement (CEGAR)

- technique to further improve scalability
 - not covered in this module
1. Model the system at hand.
 2. Build more abstract (smaller!) version of the model, which *over-approximates* system behaviours.
 3. Verify safety properties on the abstract model.
 - 3.1 If Success, the model satisfies safety (as an over-approximation was used).
 - 3.2 Otherwise, check if the counter-example is a valid one (i.e. it can be exhibited in the original model).
 - If yes, safety is violated.
 - If not, use the counter-example to guide a **refinement** of the abstract model, and go back to step 3.

Exercises

Encode the following programs:

```
begin  
  while ( $x_1 \wedge x_2$ ) do  
     $x_1 = \text{true};$   
    while ( $x_1 \leftrightarrow x_2$ ) do  
       $x_1 = x_1 \vee x_2;$   
    endwhile;  
  endwhile;  
  if ( $x_1 \vee x_2$ ) then  
     $x_3 = x_1 \leftrightarrow x_2;$   
     $x_2 = x_1 \vee x_2;$   
  else  
     $x_2 = x_1 \wedge x_2;$   
  endif  
end
```

```
begin  
   $x_1 = \text{false};$   
  while ( $x_1 \wedge x_2$ ) do  
     $x_1 = \text{true};$   
    while ( $x_1 \leftrightarrow x_2$ ) do  
      if ( $x_1 \vee x_2$ ) then  
         $x_3 = x_1 \vee \neg x_2;$   
      else  
         $x_3 = \neg x_1 \wedge x_2;$   
      endif;  
    endwhile;  
  endwhile  
end
```