



**Systems and Software  
Verification Laboratory**

**MANCHESTER**  
1824

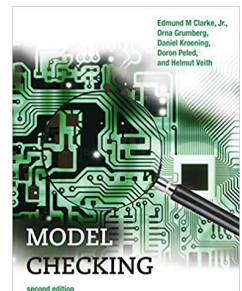
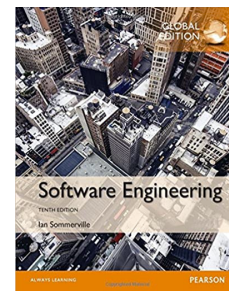
The University of Manchester

# Detection of Software Vulnerabilities: Static Analysis (Part I)

**Lucas Cordeiro**  
**Department of Computer Science**  
[lucas.cordeiro@manchester.ac.uk](mailto:lucas.cordeiro@manchester.ac.uk)

# Static Analysis

- Lucas Cordeiro (Formal Methods Group)
  - [lucas.cordeiro@manchester.ac.uk](mailto:lucas.cordeiro@manchester.ac.uk)
  - Office: 2.28
  - Office hours: 15-16 Tuesday, 14-15 Wednesday
- Textbook:
  - *Model checking* (Chapter 14)
  - *Software model checking*. ACM Comput. Surv., 2009
  - *The Cyber Security Body of Knowledge*, 2019
  - *Software Engineering* (Chapters 8, 13)



# Motivating Example

- **Functionality** demanded **increased significantly**
  - Peer reviewing and testing

# Motivating Example

- **Functionality** demanded **increased significantly**
  - Peer reviewing and testing
- Multi-core processors with scalable **shared memory / message passing**
  - Static and dynamic verification

# Motivating Example

- **Functionality** demanded **increased significantly**
  - Peer reviewing and testing
- Multi-core processors with scalable **shared memory / message passing**
  - Static and dynamic verification

```
void *threadA(void *arg) {  
    lock(&mutex);  
    x++;  
    if (x == 1) lock(&lock);  
    unlock(&mutex);  
    lock(&mutex);  
    x--;  
    if (x == 0) unlock(&lock);  
    unlock(&mutex);  
}
```

```
void *threadB(void *arg) {  
    lock(&mutex);  
    y++;  
    if (y == 1) lock(&lock);  
    unlock(&mutex);  
    lock(&mutex);  
    y--;  
    if (y == 0) unlock(&lock);  
    unlock(&mutex);  
}
```

# Motivating Example

- **Functionality** demanded **increased significantly**
  - Peer reviewing and testing
- Multi-core processors with scalable **shared memory / message passing**
  - Static and dynamic verification

```
void *threadA(void *arg) {  
    lock(&mutex);  
    x++;  
    if (x == 1) lock(&lock);  
    unlock(&mutex); (CS1)  
    lock(&mutex);  
    x--;  
    if (x == 0) unlock(&lock);  
    unlock(&mutex);  
}
```

```
void *threadB(void *arg) {  
    lock(&mutex);  
    y++;  
    if (y == 1) lock(&lock);  
    unlock(&mutex);  
    lock(&mutex);  
    y--;  
    if (y == 0) unlock(&lock);  
    unlock(&mutex);  
}
```

# Motivating Example

- **Functionality** demanded **increased significantly**
  - Peer reviewing and testing
- Multi-core processors with scalable **shared memory / message passing**
  - Static and dynamic verification

```
void *threadA(void *arg) {  
    lock(&mutex);  
    x++;  
    if (x == 1) lock(&lock);  
    unlock(&mutex); (CS1)  
    lock(&mutex);  
    x--;  
    if (x == 0) unlock(&lock);  
    unlock(&mutex);  
}
```

```
void *threadB(void *arg) {  
    lock(&mutex);  
    y++;  
    if (y == 1) lock(&lock); (CS2)  
    unlock(&mutex);  
    lock(&mutex);  
    y--;  
    if (y == 0) unlock(&lock);  
    unlock(&mutex);  
}
```

# Motivating Example

- **Functionality** demanded **increased significantly**
  - Peer reviewing and testing
- Multi-core processors with scalable **shared memory / message passing**
  - Static and dynamic verification

```
void *threadA(void *arg) {  
    lock(&mutex);  
    x++;  
    if (x == 1) lock(&lock);  
    unlock(&mutex); (CS1)  
    lock(&mutex); (CS3)  
    x--;  
    if (x == 0) unlock(&lock);  
    unlock(&mutex);  
}
```

```
void *threadB(void *arg) {  
    lock(&mutex);  
    y++;  
    if (y == 1) lock(&lock); (CS2)  
    unlock(&mutex);  
    lock(&mutex);  
    y--;  
    if (y == 0) unlock(&lock);  
    unlock(&mutex);  
}
```



# Motivating Example

- **Functionality** demanded **increased significantly**
  - Peer reviewing and testing
- Multi-core processors with scalable **shared memory / message passing**
  - Static and dynamic verification

```
void *threadA(void *arg) {  
    lock(&mutex);  
    x++;  
    if (x == 1) lock(&lock);  
    unlock(&mutex); (CS1)  
    lock(&mutex); (CS3)  
    x--;  
    if (x == 0) unlock(&lock);  
    unlock(&mutex);  
}
```

**Deadlock**

```
void *threadB(void *arg) {  
    lock(&mutex);  
    y++;  
    if (y == 1) lock(&lock); (CS2)  
    lock(&mutex);  
    y--;  
    if (y == 0) unlock(&lock);  
    unlock(&mutex);  
}
```

# Intended learning outcomes

- Introduce **software verification** and **validation**

# Intended learning outcomes

- Introduce **software verification** and **validation**
- Understand **soundness** and **completeness** concerning **detection techniques**

# Intended learning outcomes

- Introduce **software verification** and **validation**
- Understand **soundness** and **completeness** concerning **detection techniques**
- Emphasize the difference among **static analysis**, **testing / simulation**, and **debugging**

# Intended learning outcomes

- Introduce **software verification** and **validation**
- Understand **soundness** and **completeness** concerning **detection techniques**
- Emphasize the difference among **static analysis**, **testing / simulation**, and **debugging**
- Explain **bounded model checking** of software

# Intended learning outcomes

- Introduce **software verification** and **validation**
- Understand **soundness** and **completeness** concerning **detection techniques**
- Emphasize the difference among **static analysis**, **testing / simulation**, and **debugging**
- Explain **bounded model checking** of software
- Explain **precise memory model** for software **verification**

# Intended learning outcomes

- Introduce **software verification** and **validation**
- Understand **soundness** and **completeness** concerning **detection techniques**
- Emphasize the difference among **static analysis**, **testing / simulation**, and **debugging**
- Explain **bounded model checking** of software
- Explain **precise memory model** for software **verification**

# Verification vs Validation

- **Verification:** *"Are we building the product right?"*
  - The software should **conform to its specification**



# Verification vs Validation

- **Verification:** *"Are we building the product right?"*
  - The software should **conform to its specification**
- **Validation:** *"Are we building the right product?"*
  - The software should do what the **user requires**

# Verification vs Validation

- **Verification:** *"Are we building the product right?"*
  - The software should **conform to its specification**
- **Validation:** *"Are we building the right product?"*
  - The software should do what the **user requires**
- Verification and validation must be applied at **each stage in the software process**
  - The **discovery of defects** in a system
  - The assessment of whether or not the system is **usable in an operational situation**

# Static and Dynamic Verification

- **Software inspections** are concerned with the analysis of the static system representation to discover problems (**static verification**)
  - Supplement by **tool-based document and code analysis**
  - **Code analysis** can **prove the absence of errors** but might be subject to **incorrect results**

# Static and Dynamic Verification

- **Software inspections** are concerned with the analysis of the static system representation to discover problems (**static verification**)
  - Supplement by **tool-based document and code analysis**
  - **Code analysis can prove the absence of errors** but might be subject to **incorrect results**
- **Software testing** is concerned with exercising and observing product behaviour (**dynamic verification**)
  - The system is executed with **test data**
  - **Operational behaviour is observed**
  - Can reveal the presence of errors **NOT their absence**

# Static and Dynamic Verification



The image cannot be displayed. Your computer may not have enough memory to open the image, or the image may have been corrupted. Restart your computer, and then open the file again. If the red x still appears, you may have to delete the image and then insert it again.

Ian Sommerville. Software Engineering  
(6th,7th or 8th Edn) Addison Wesley

# V & V planning

- **Careful planning** is required to get the most out of **dynamic and static verification**
  - Planning should start **early in the development process**
  - The plan should identify the **balance between static and dynamic verification**

# V & V planning

- **Careful planning** is required to get the most out of **dynamic and static verification**
  - Planning should start **early in the development process**
  - The plan should identify the **balance between static and dynamic verification**
- V & V should establish confidence that the **software is fit for purpose**

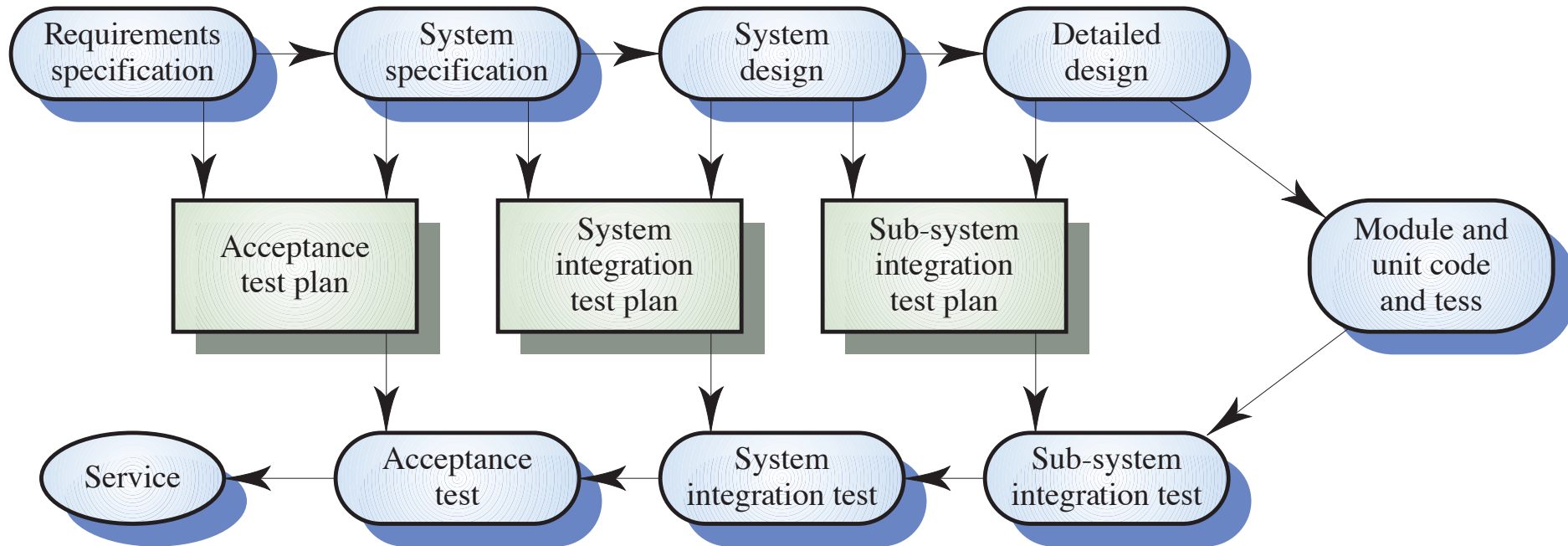
# V & V planning

- **Careful planning** is required to get the most out of **dynamic and static verification**
  - Planning should start **early in the development process**
  - The plan should identify the **balance between static and dynamic verification**
- V & V should establish confidence that the **software is fit for purpose**

V & V planning depends on **system's purpose, user expectations and marketing environment**



# The V-model of development



Ian Sommerville. Software Engineering  
(6th,7th or 8th Edn) Addison Wesley

# Intended learning outcomes

- Introduce **software verification and validation**
- Understand **soundness and completeness** concerning **detection techniques**
- Emphasize the difference among **static analysis, testing / simulation, and debugging**
- Explain **bounded model checking of software**
- Explain **unbounded model checking of software**

# Detection of Vulnerabilities

- Detect the presence of vulnerabilities in the code during the **development, testing, and maintenance**

# Detection of Vulnerabilities

- Detect the presence of vulnerabilities in the code during the **development, testing, and maintenance**
- Trade-off between **soundness** and **completeness**

# Detection of Vulnerabilities

- Detect the presence of vulnerabilities in the code during the **development, testing, and maintenance**
- Trade-off between **soundness** and **completeness**
  - A detection technique is **sound** for a given category if it can correctly conclude that a given program has no vulnerabilities
    - An unsound detection technique may have **false negatives**, i.e., actual vulnerabilities that the detection technique fails to find

# Detection of Vulnerabilities

- Detect the presence of vulnerabilities in the code during the **development, testing, and maintenance**
- Trade-off between **soundness** and **completeness**
  - A detection technique is **sound** for a given category if it can correctly conclude that a given program has no vulnerabilities
    - An unsound detection technique may have **false negatives**, i.e., actual vulnerabilities that the detection technique fails to find
  - A detection technique is **complete** for a given category, if any vulnerability it finds is an actual vulnerability
    - An incomplete detection technique may have **false positives**, i.e., it may detect issues that do not turn out to be actual vulnerabilities

# Detection of Vulnerabilities

- Achieving **soundness** requires reasoning about **all executions** of a program (usually an infinite number)
  - This can be done by static checking of the program code while making suitable abstractions of the executions

# Detection of Vulnerabilities

- Achieving **soundness** requires reasoning about **all executions** of a program (usually an infinite number)
  - This can be done by static checking of the program code while making suitable abstractions of the executions
- Achieving **completeness** can be done by performing actual, **concrete executions** of a program that are witnesses to any vulnerability reported
  - The analysis technique has to come up with concrete inputs for the program that triggers a vulnerability
  - A typical dynamic approach is software testing: the tester writes test cases with concrete inputs and specific checks for the outputs



# Detection of Vulnerabilities

Detection tools can use a **hybrid combination of static and dynamic analysis** techniques to achieve a good trade-off between **soundness and completeness**

# Detection of Vulnerabilities

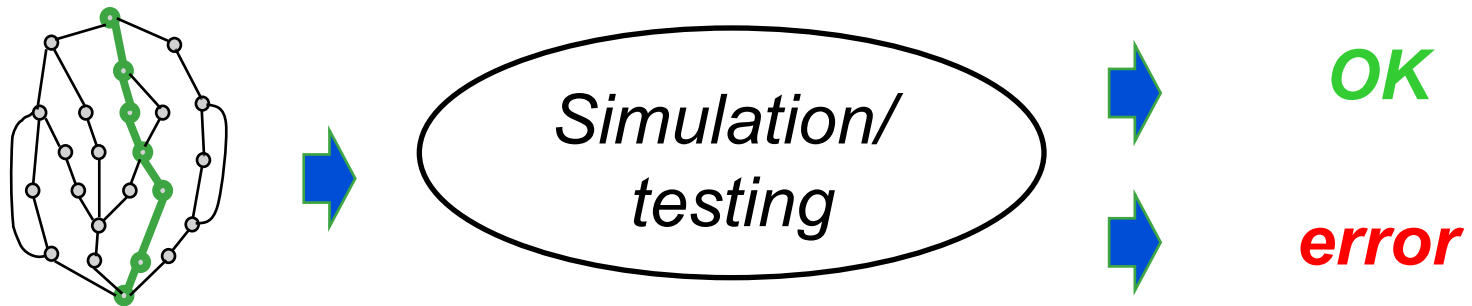
Detection tools can use a **hybrid combination of static and dynamic analysis** techniques to achieve a good trade-off between **soundness and completeness**

**Dynamic verification** should be used in conjunction with **static verification** to provide **full code coverage**

# Intended learning outcomes

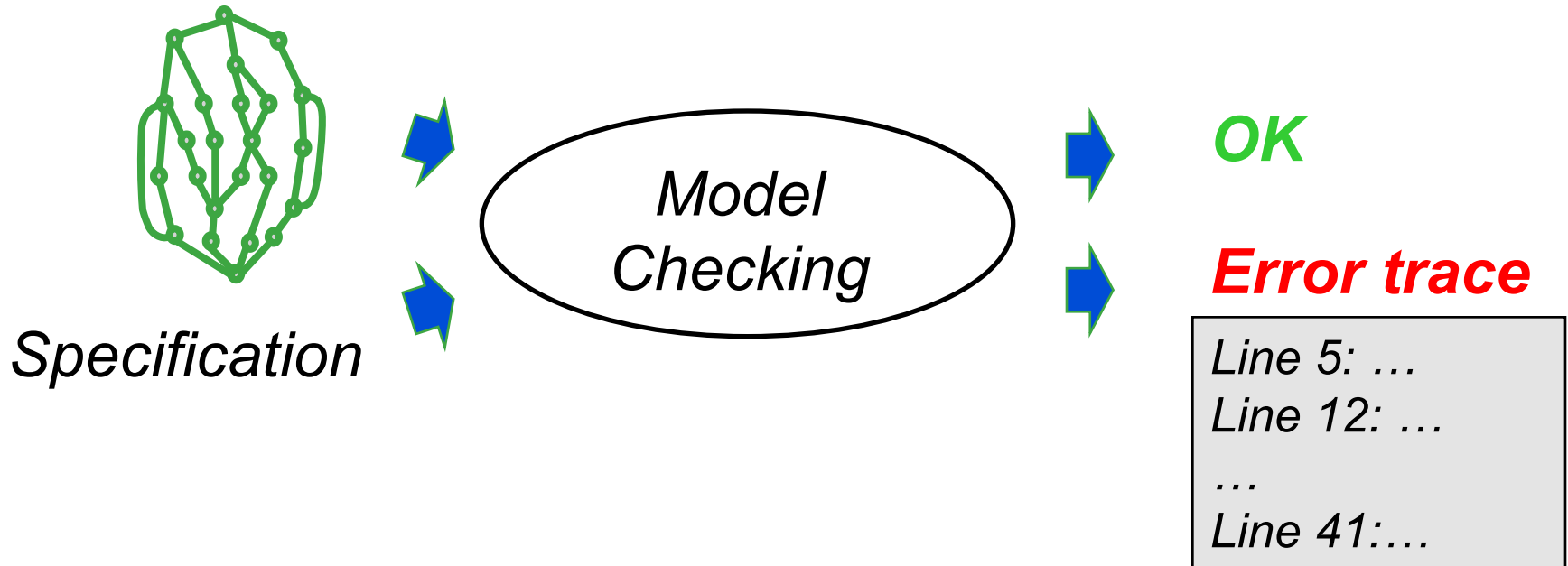
- Introduce **software verification and validation**
- Understand **soundness and completeness** concerning **detection techniques**
- Emphasize the difference among **static analysis, testing / simulation, and debugging**
- Explain **bounded model checking of software**
- Explain **unbounded model checking of software**

# Static analysis vs Testing/ Simulation



- **Checks only some of the system executions**
  - May miss errors
- A **successful execution** is an execution that **discovers one or more errors**

# Static analysis vs Testing/ Simulation



- **Exhaustively explores all executions**
- Report errors as **traces**
- May produce **incorrect results**

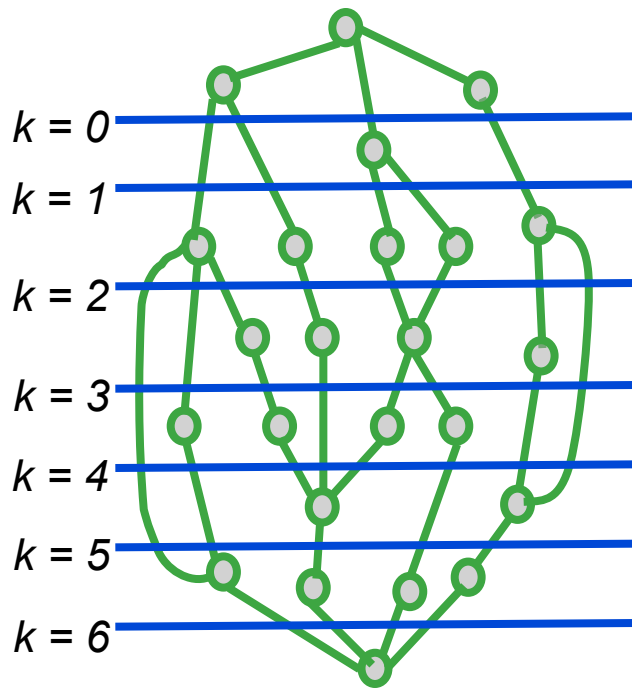
# Avoiding state space explosion

- Bounded Model Checking (BMC)
  - **Breadth-first search (BFS)** approach
- Symbolic Execution
  - **Depth-first search (DFS)** approach

# Bounded Model Checking

A graph  $G = (V, E)$  consists of:

- $V$ : a set of vertices or nodes
- $E \subseteq V \times V$ : set of edges connecting the nodes



- Bounded model checkers explore the state space in depth
- Can only prove correctness if all states are reachable within the bound

# Breadth-First Search (BFS)

**BFS** ( $G, s$ )

```
01 for each vertex  $u \in V[G] - \{s\}$  // anchor ( $s$ )
02     colour[ $u$ ]  $\leftarrow$  white //  $u$  colour
03      $d[u] \leftarrow \infty$  //  $s$  distance
04      $\pi[u] \leftarrow \text{NIL}$  //  $u$  predecessor
```

Initialization of  
graph nodes

```
05 colour[ $s$ ]  $\leftarrow$  grey
06  $d[s] \leftarrow 0$ 
07  $\pi[s] \leftarrow \text{NIL}$ 
08 enqueue( $Q, s$ )
```

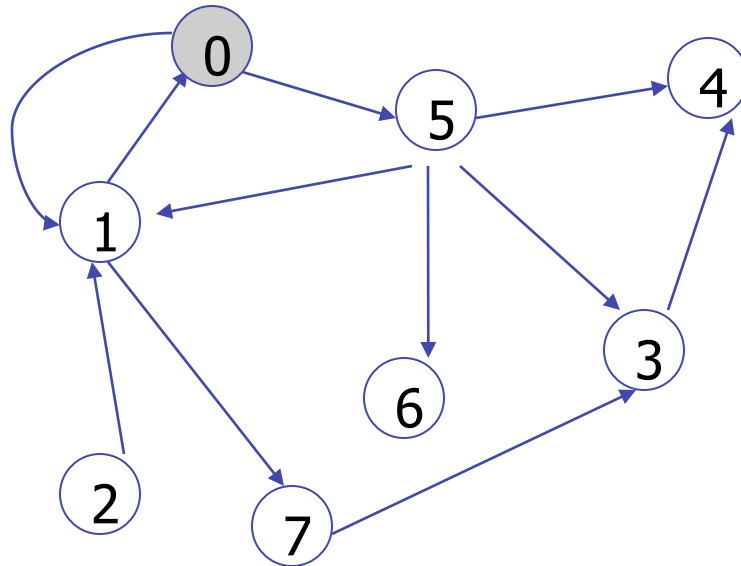
Initializes the  
anchor node ( $s$ )

```
09 while  $Q \neq \emptyset$  do
10      $u \leftarrow$  dequeue( $Q$ )
11     for each  $v \in \text{Adj}[u]$  do
12         if colour[ $v$ ] = white then
13             colour[ $v$ ]  $\leftarrow$  grey
14              $d[v] \leftarrow d[u] + 1$ 
15              $\pi[v] \leftarrow u$ 
16             enqueue( $Q, v$ )
17     colour[ $u$ ]  $\leftarrow$  blue
```

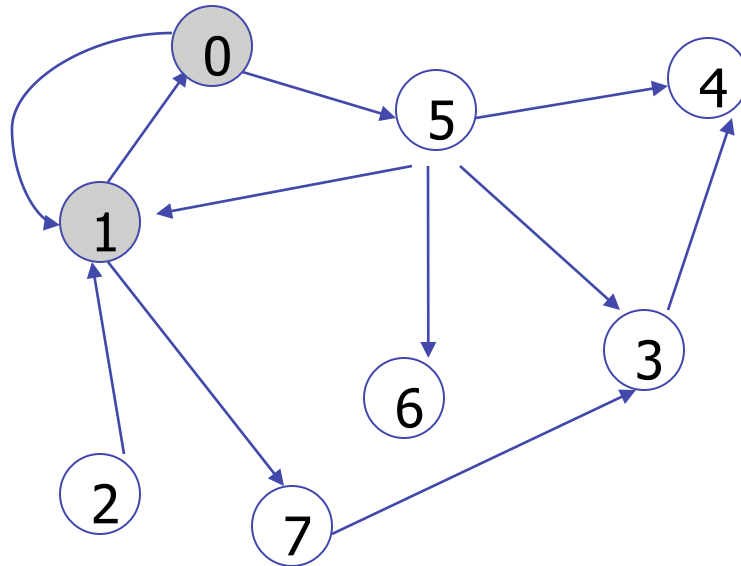
Visit each adjacent  
node of  $u$



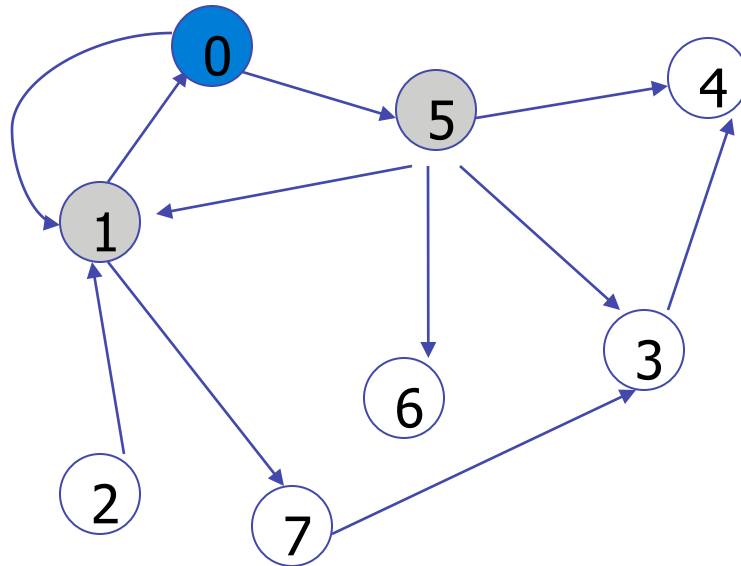
# BFS Example



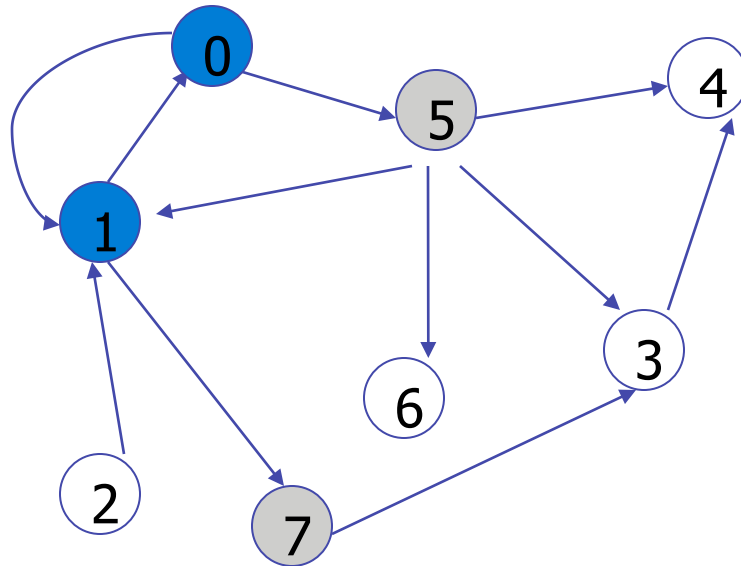
# BFS Example



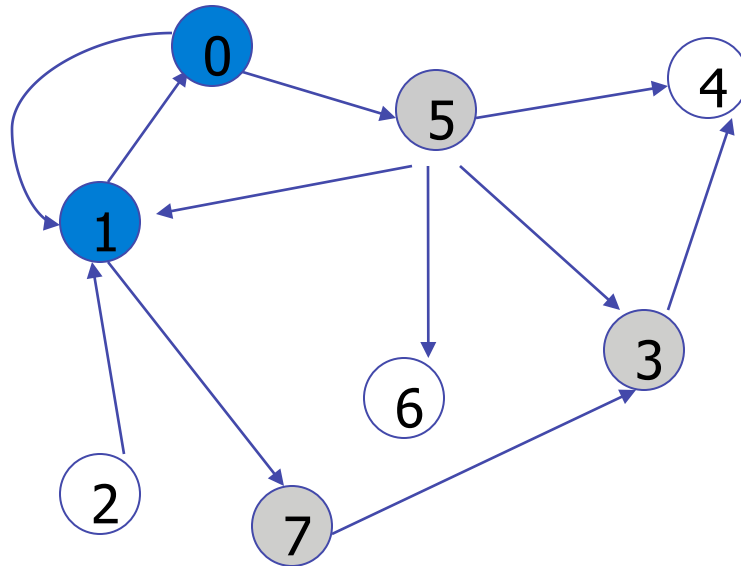
# BFS Example



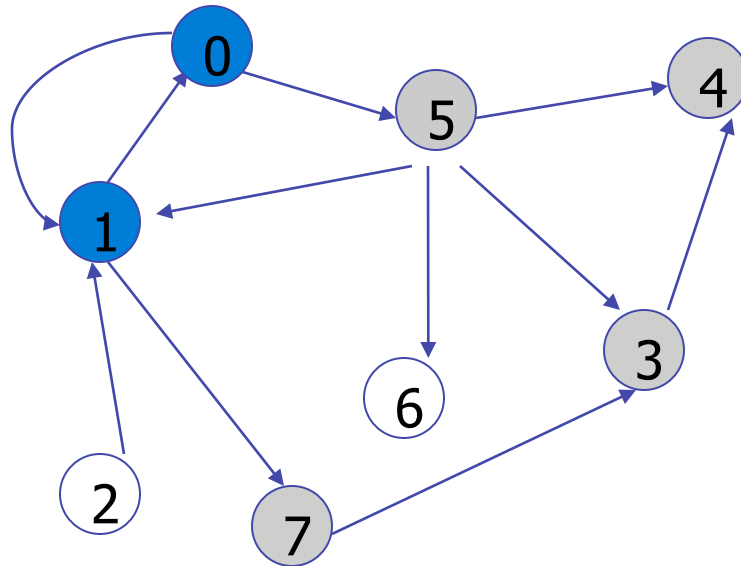
# BFS Example



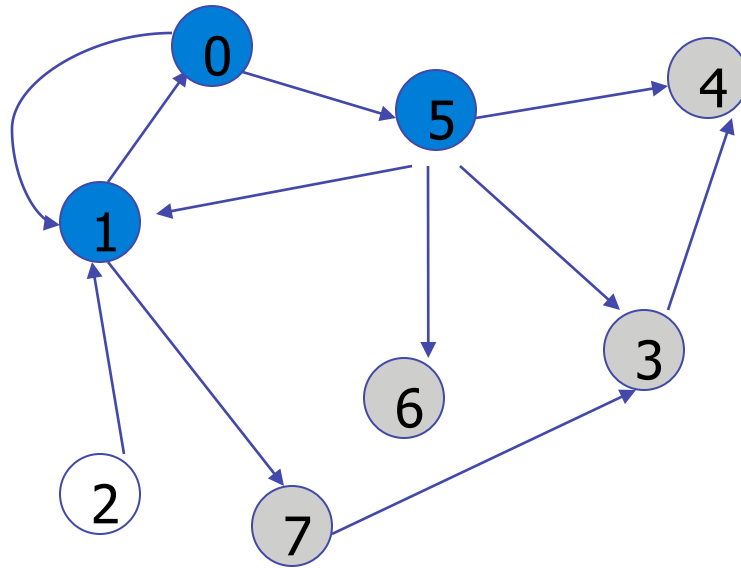
# BFS Example



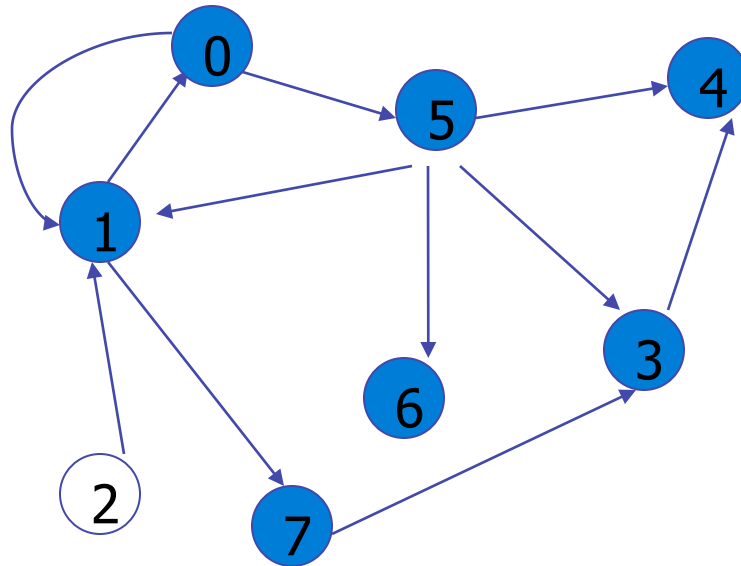
# BFS Example



# BFS Example

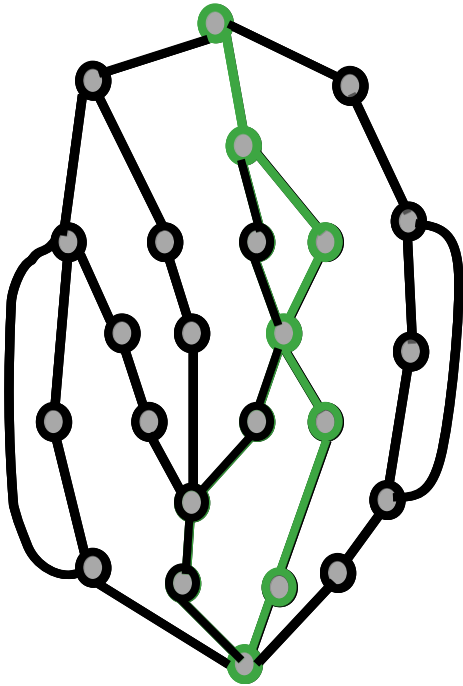


# BFS Example





# Symbolic Execution



- Symbolic execution explores all paths individually
- Can only prove correctness if all paths are explored

# Depth-first search (DFS)

DFS( $G$ )

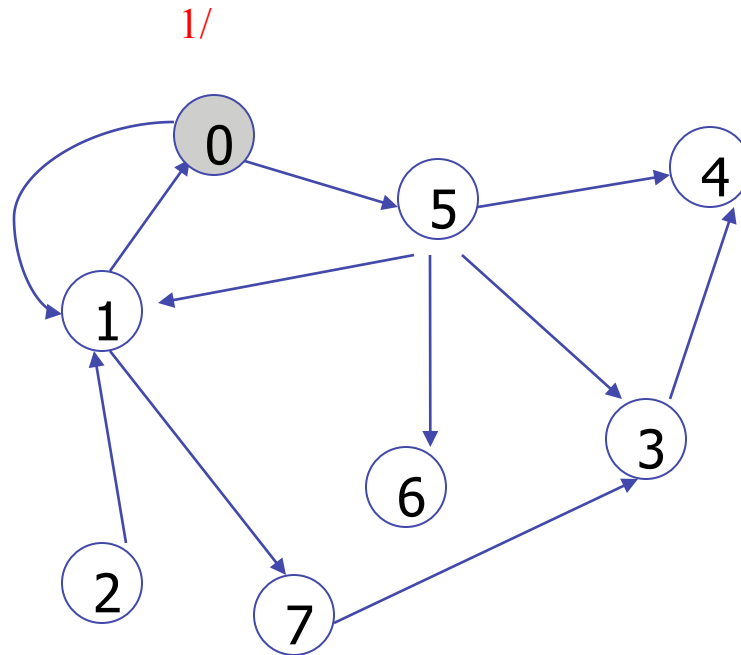
```
1  for each vertex  $u \in V[G]$ 
2      do  $color[u] \leftarrow WHITE$ 
3          $\pi[u] \leftarrow NIL$ 
4   $time \leftarrow 0$ 
5  for each vertex  $u \in V[G]$ 
6      do if  $color[u] = WHITE$ 
7         then DFS-VISIT( $u$ )
```

Paint all vertices white and initialize the fields  $\pi$  with NIL where  $\pi[u]$  represents the predecessor of  $u$

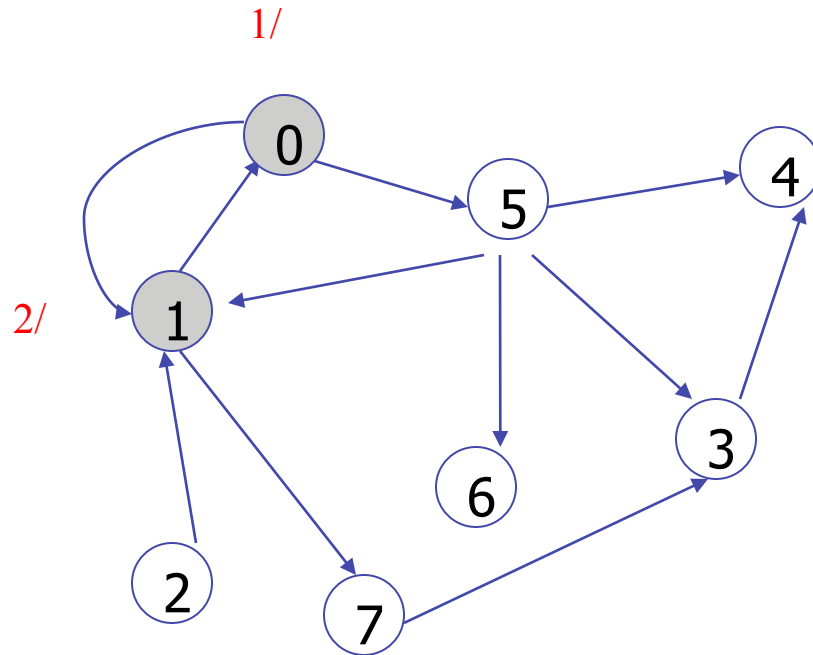
DFS-VISIT( $u$ )

```
1   $color[u] \leftarrow GRAY$        $\triangleright$  White vertex  $u$  has just been discovered.
2   $time \leftarrow time + 1$ 
3   $d[u] \leftarrow time$ 
4  for each  $v \in Adj[u]$        $\triangleright$  Explore edge  $(u, v)$ .
5      do if  $color[v] = WHITE$ 
6         then  $\pi[v] \leftarrow u$ 
7            DFS-VISIT( $v$ )
8   $color[u] \leftarrow BLACK$      $\triangleright$  Blacken  $u$ ; it is finished.
9   $f[u] \leftarrow time \leftarrow time + 1$ 
```

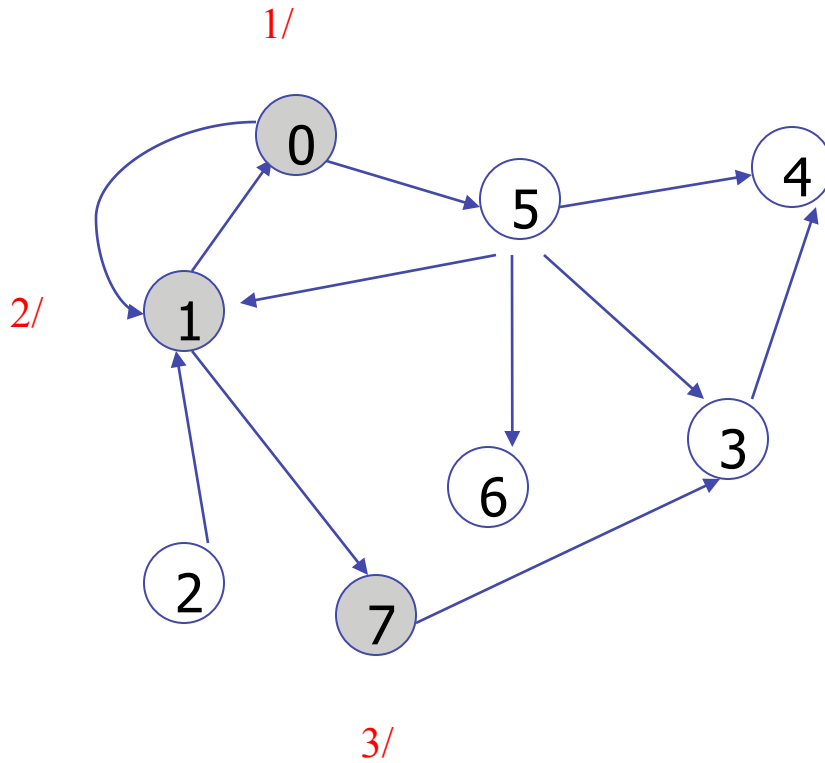
# DFS Example



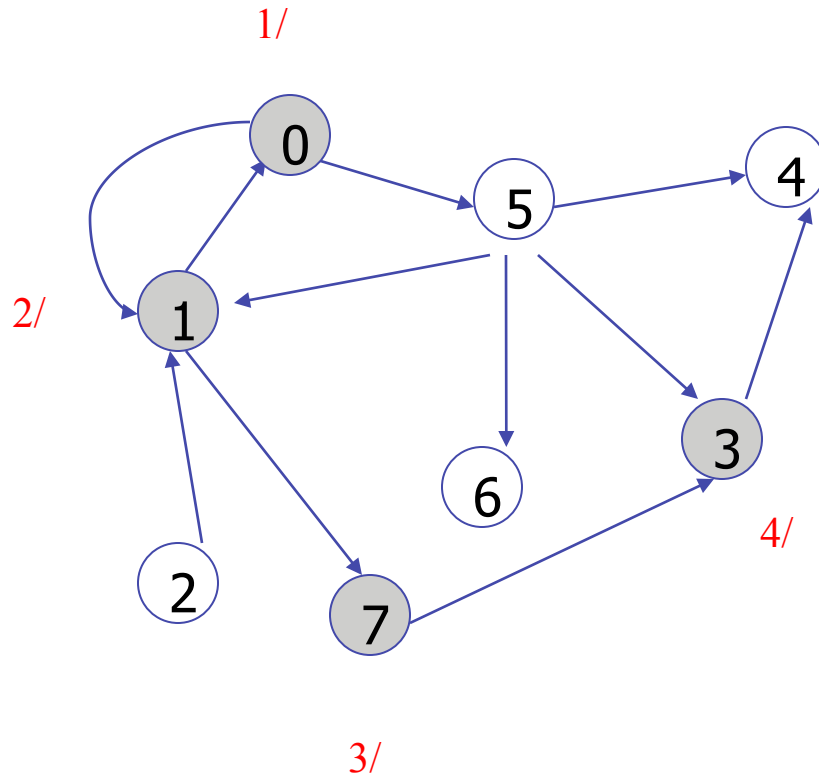
# DFS Example



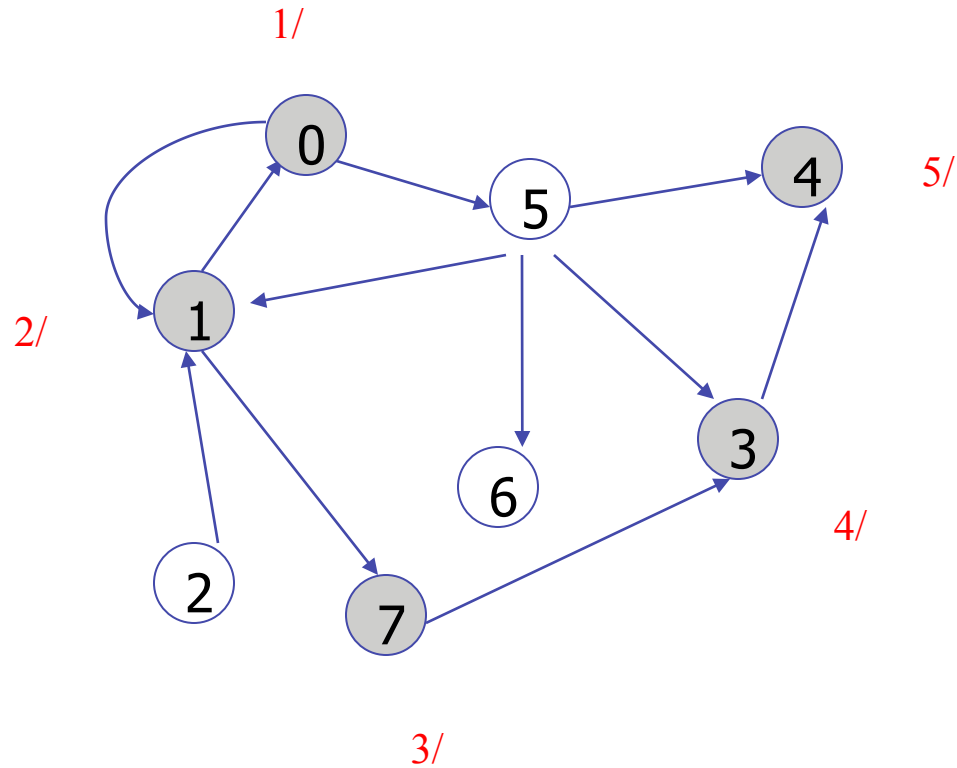
# DFS Example



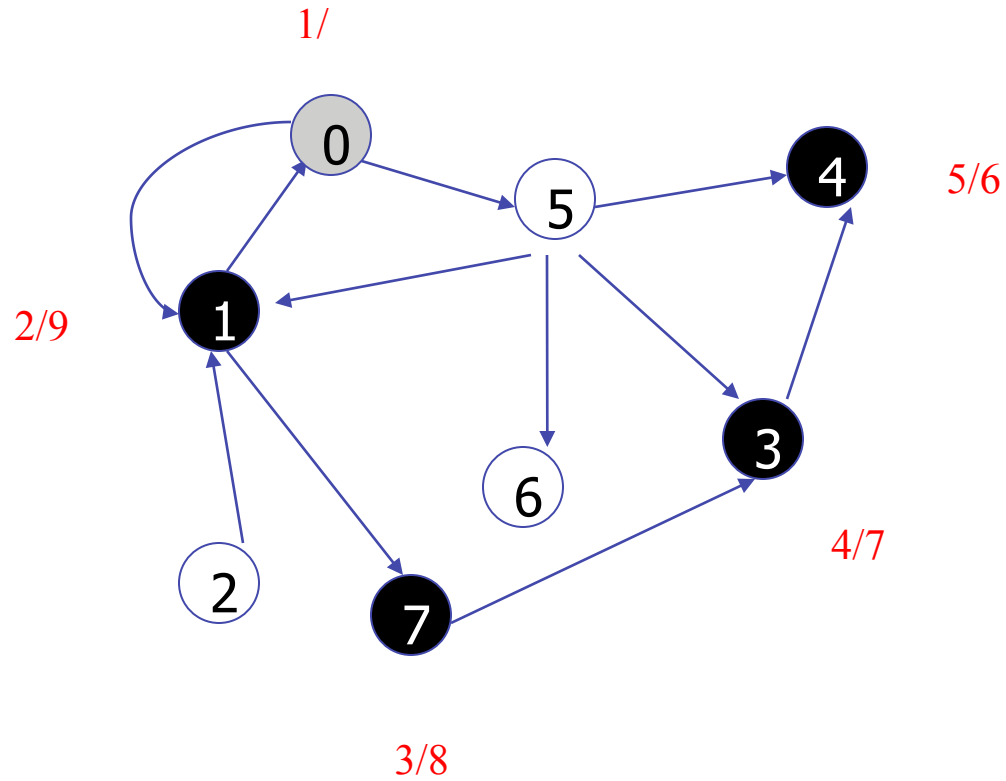
# DFS Example



# DFS Example

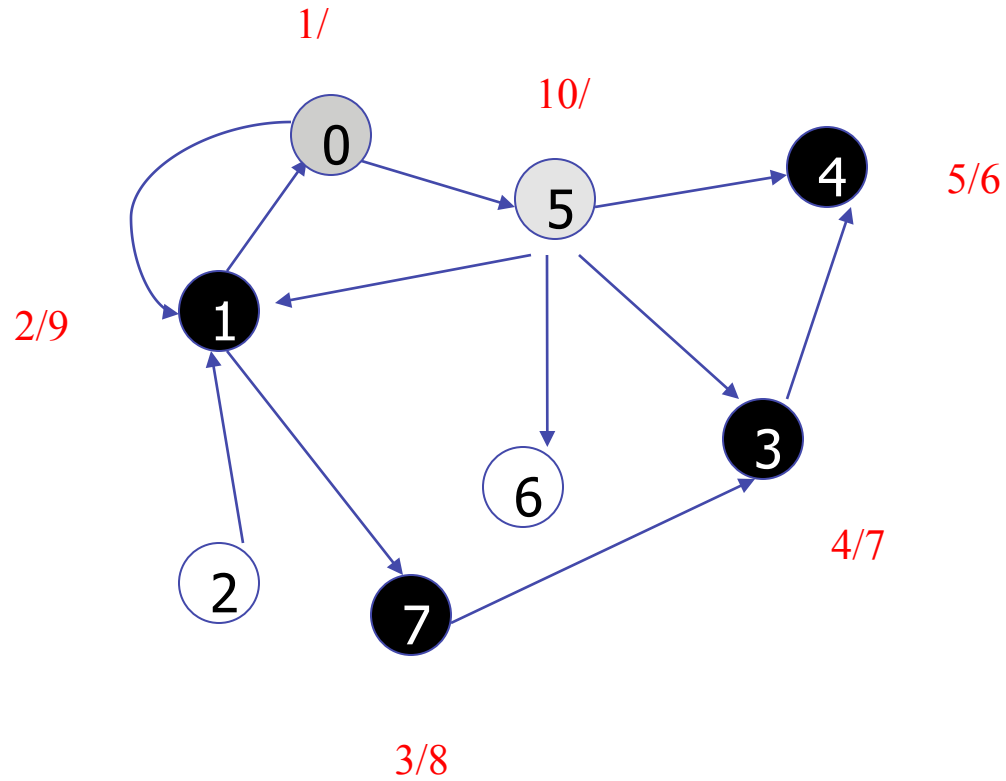


# DFS Example

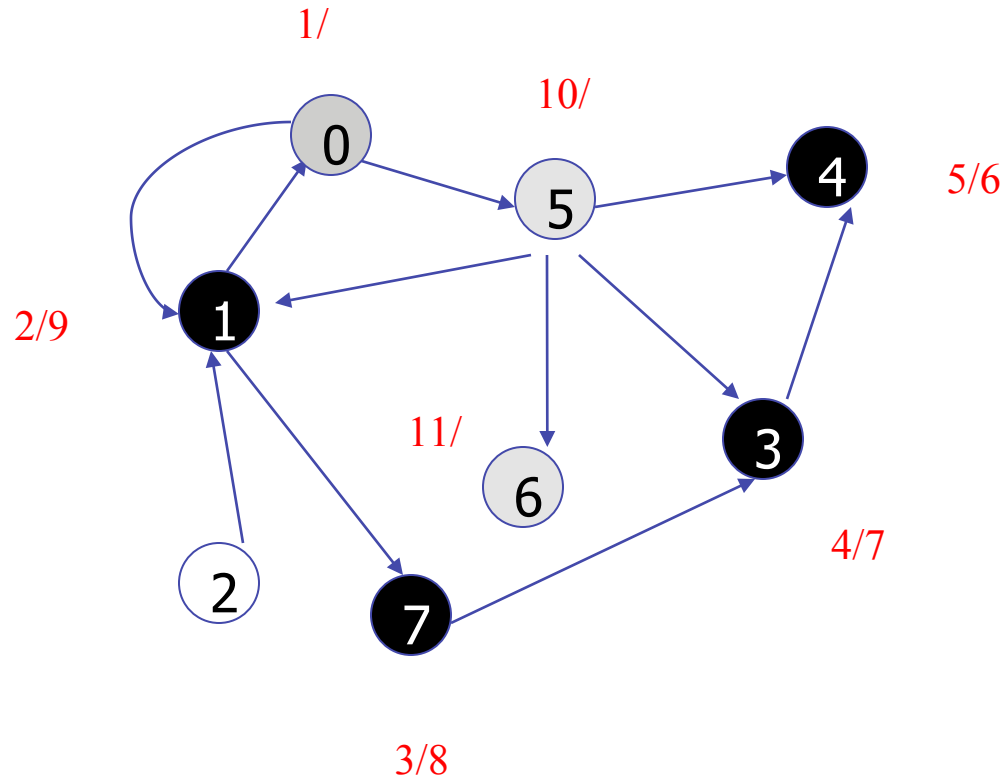




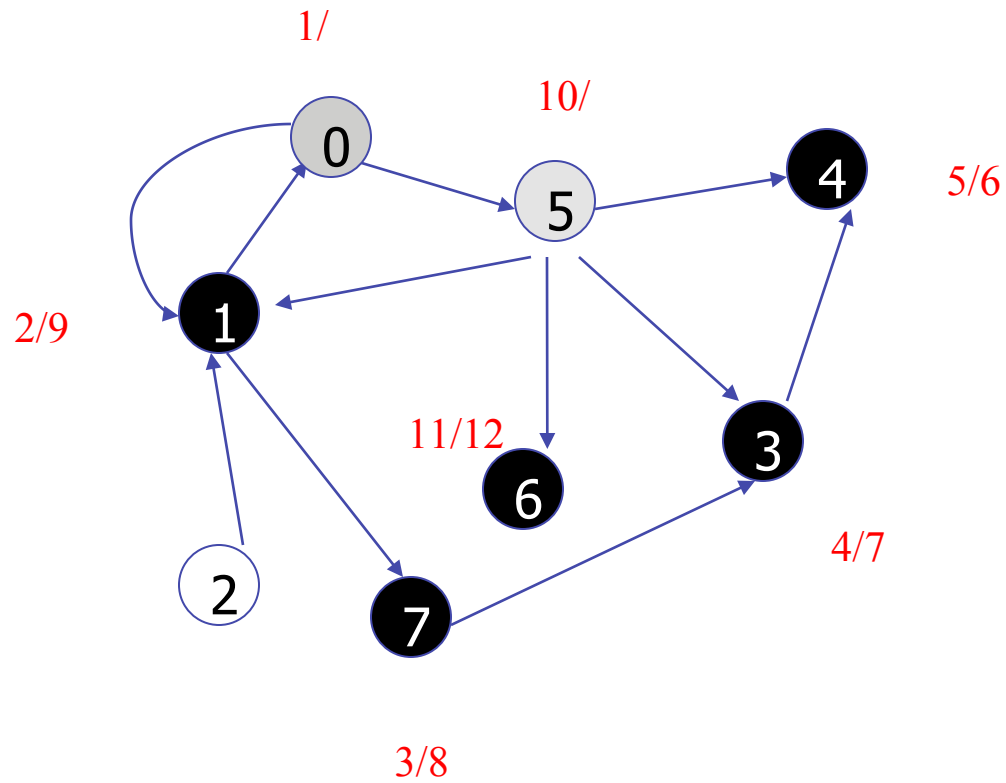
# DFS Example



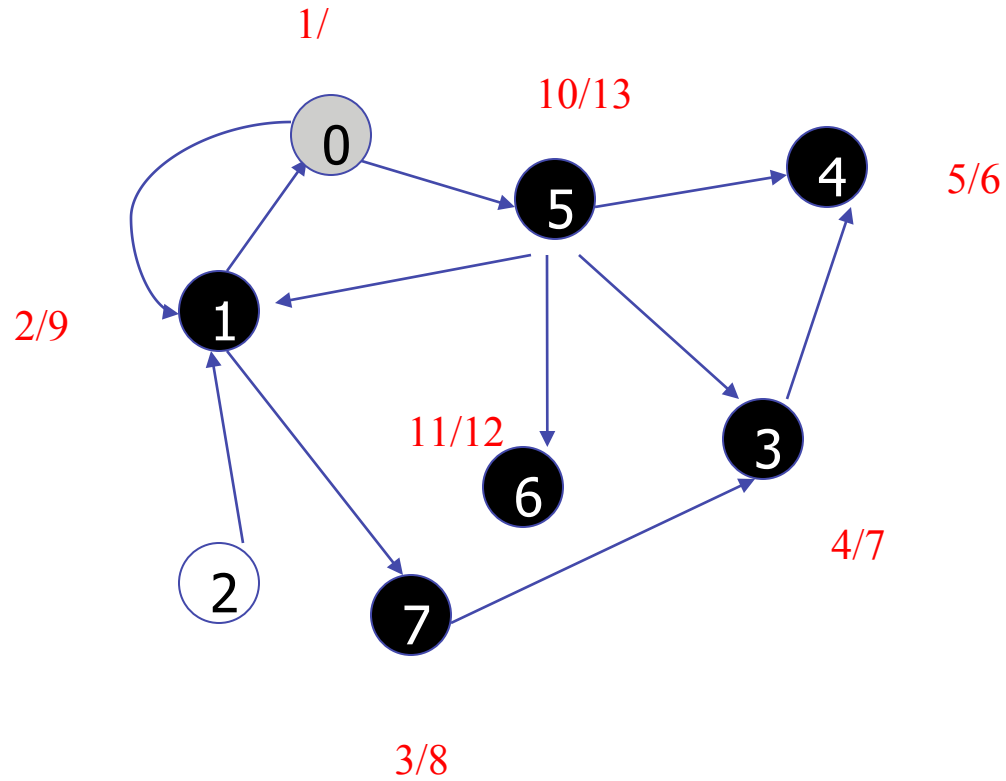
# DFS Example



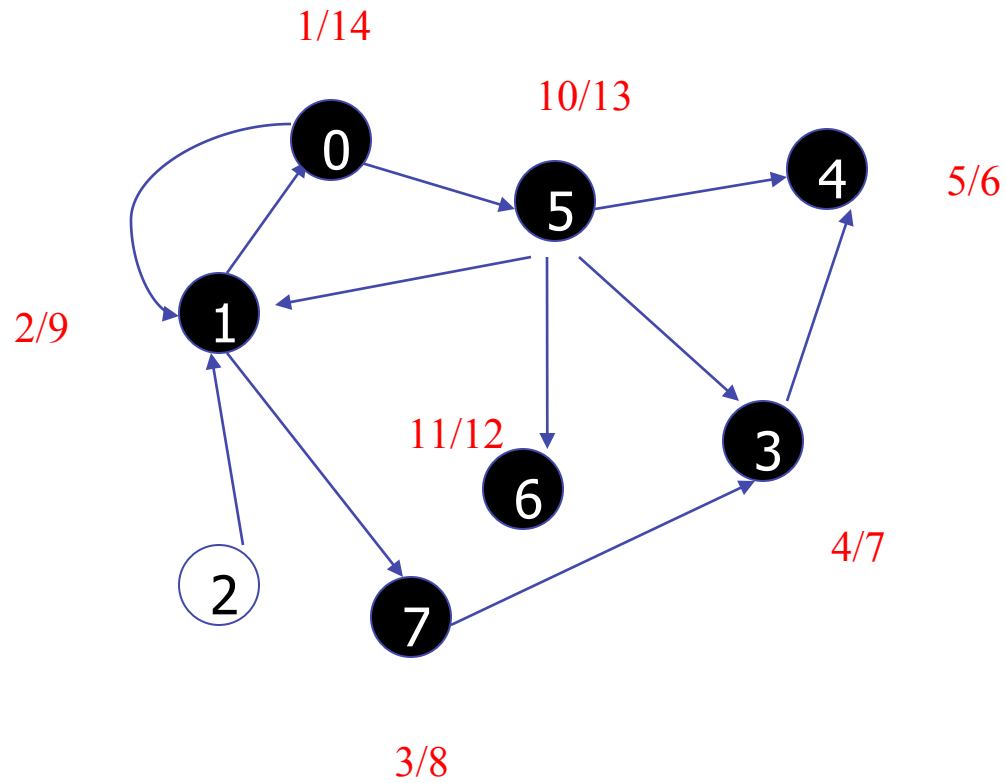
# DFS Example



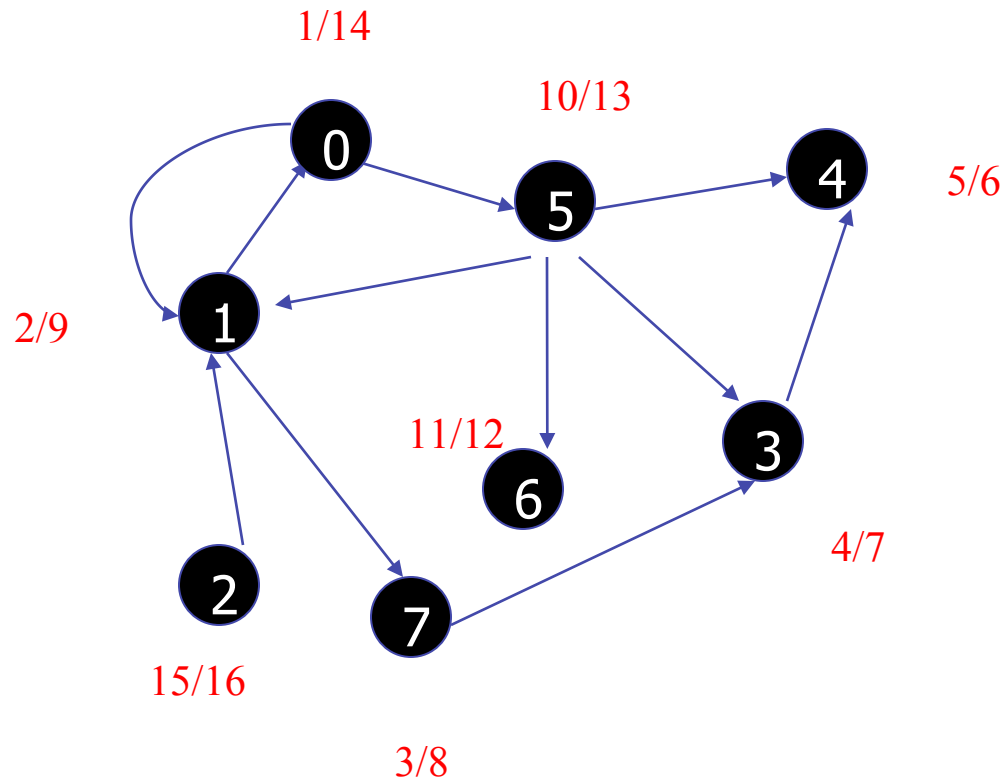
# DFS Example



# DFS Example



# DFS Example



# V&V and debugging

- V & V and debugging are **distinct processes**

# V&V and debugging

- V & V and debugging are **distinct processes**
- **V & V** is concerned with establishing the **absence or existence of defects** in a program, resp.



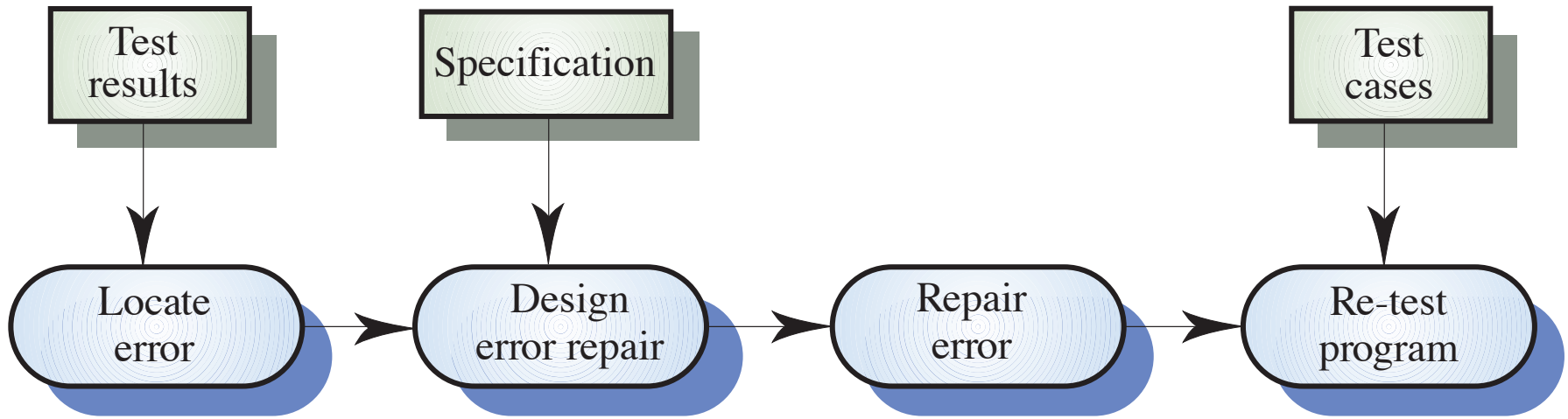
# V&V and debugging

- V & V and debugging are **distinct processes**
- **V & V** is concerned with establishing the **absence or existence of defects** in a program, resp.
- **Debugging** is concerned with two main tasks
  - **Locating and**
  - **Repairing these errors**

# V&V and debugging

- V & V and debugging are **distinct processes**
- **V & V** is concerned with establishing the **absence or existence of defects** in a program, resp.
- **Debugging** is concerned with two main tasks
  - **Locating and**
  - **Repairing these errors**
- Debugging involves
  - Formulating a hypothesis about program behaviour
  - Test these hypotheses to find the system error

# The debugging process



Ian Sommerville. Software Engineering  
(6th,7th or 8th Edn) Addison Wesley

# Intended learning outcomes

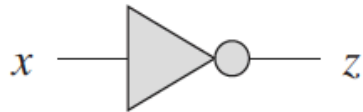
- Introduce **software verification** and **validation**
- Understand **soundness** and **completeness** concerning **detection techniques**
- Emphasize the difference among **static analysis**, **testing / simulation**, and **debugging**
- Explain **bounded model checking of software**
- Explain **precise memory model** for **software verification**

# Circuit Satisfiability

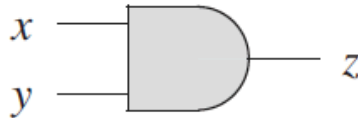
- A Boolean formula contains
  - **Variables** whose values are **0** or **1**

# Circuit Satisfiability

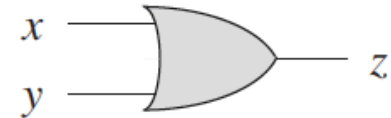
- A Boolean formula contains
  - **Variables** whose values are **0** or **1**
  - **Connectives**:  $\wedge$  (**AND**),  $\vee$  (**OR**), and  $\neg$  (**NOT**)



$x$	$\neg x$
0	1
1	0



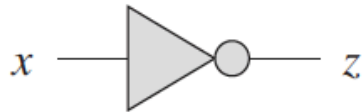
$x$	$y$	$x \wedge y$
0	0	0
0	1	0
1	0	0
1	1	1



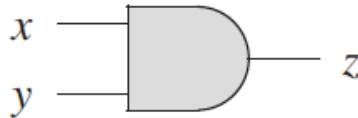
$x$	$y$	$x \vee y$
0	0	0
0	1	1
1	0	1
1	1	1

# Circuit Satisfiability

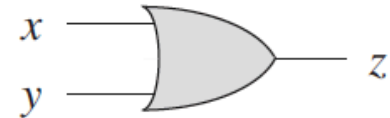
- A Boolean formula contains
  - **Variables** whose values are **0** or **1**
  - **Connectives**:  $\wedge$  (**AND**),  $\vee$  (**OR**), and  $\neg$  (**NOT**)



$x$	$\neg x$
0	1
1	0



$x$	$y$	$x \wedge y$
0	0	0
0	1	0
1	0	0
1	1	1

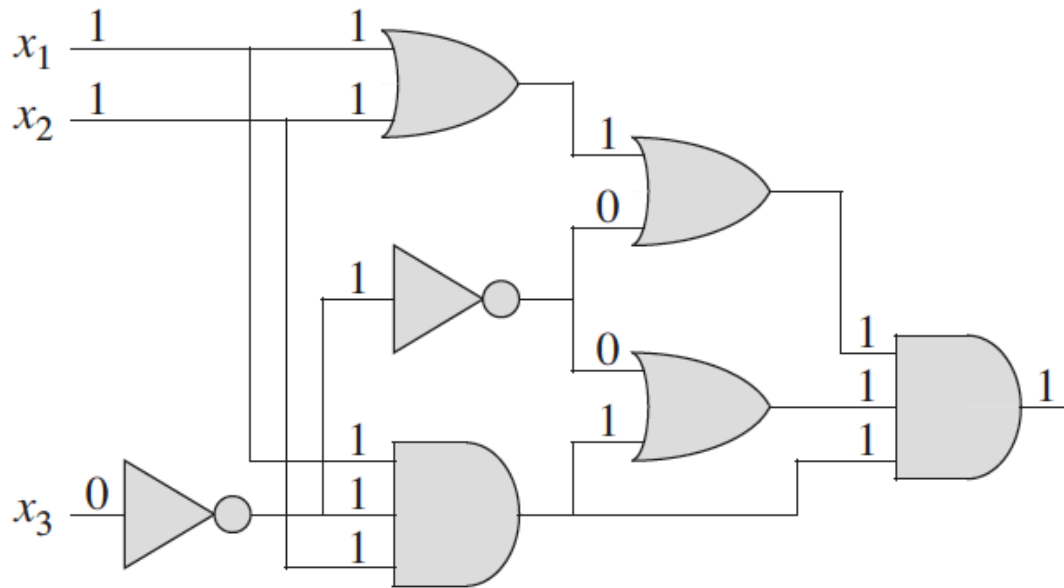


$x$	$y$	$x \vee y$
0	0	0
0	1	1
1	0	1
1	1	1

- A Boolean formula is **SAT** if there exists some assignment to its variables that **evaluates it to 1**

# Circuit Satisfiability

- A **Boolean combinational circuit** consists of one or more **Boolean combinational elements** interconnected by **wires**



*SAT:*  $\langle x_1 = 1, x_2 = 1, x_3 = 0 \rangle$



# Circuit-Satisfiability Problem

- Given a **Boolean combinational circuit** of AND, OR, and NOT gates, is it **satisfiable**?

CIRCUIT-SAT = {<C> : C is a satisfiable Boolean combinational circuit}

# Circuit-Satisfiability Problem

- Given a **Boolean combinational circuit** of AND, OR, and NOT gates, is it **satisfiable**?

CIRCUIT-SAT = {<C> : C is a satisfiable Boolean combinational circuit}

- **Size:** number of **Boolean combinational elements** plus **the number of wires**
  - o if the circuit has  **$k$  inputs**, then we would have to check up to  **$2^k$  possible assignments**

# Circuit-Satisfiability Problem

- Given a **Boolean combinational circuit** of AND, OR, and NOT gates, is it **satisfiable**?

CIRCUIT-SAT = {<C> : C is a satisfiable Boolean combinational circuit}

- **Size:** number of **Boolean combinational elements** plus **the number of wires**
  - o if the circuit has  **$k$  inputs**, then we would have to check up to  **$2^k$  possible assignments**
- When the **size of C is polynomial in  $k$** , checking each one takes  **$\Omega(2^k)$** 
  - o Super-polynomial in the size of  $k$

# Formula Satisfiability (SAT)

- The SAT problem asks whether a given Boolean formula is satisfiable

$SAT = \{ \langle \Phi \rangle : \Phi \text{ is a satisfiable Boolean formula} \}$

# Formula Satisfiability (SAT)

- The SAT problem asks whether a given Boolean formula is satisfiable

SAT = { $\langle \Phi \rangle$  :  $\Phi$  is a satisfiable Boolean formula}

- Example:

$$\circ \Phi = ((x_1 \rightarrow x_2) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg x_2$$

# Formula Satisfiability (SAT)

- The SAT problem asks whether a given Boolean formula is satisfiable

SAT = { $\langle \Phi \rangle$  :  $\Phi$  is a satisfiable Boolean formula}

- Example:

- $\Phi = ((x_1 \rightarrow x_2) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg x_2$

- Assignment:  $\langle x_1 = 0, x_2 = 0, x_3 = 1, x_4 = 1 \rangle$

# Formula Satisfiability (SAT)

- The SAT problem asks whether a given Boolean formula is satisfiable

SAT = { $\langle \Phi \rangle$  :  $\Phi$  is a satisfiable Boolean formula}

- Example:

- $\Phi = ((x_1 \rightarrow x_2) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg x_2$

- Assignment:  $\langle x_1 = 0, x_2 = 0, x_3 = 1, x_4 = 1 \rangle$

- $\Phi = ((0 \rightarrow 0) \vee \neg((\neg 0 \leftrightarrow 1) \vee 1)) \wedge \neg 0$

# Formula Satisfiability (SAT)

- The SAT problem asks whether a given Boolean formula is satisfiable

SAT = { $\langle \Phi \rangle$  :  $\Phi$  is a satisfiable Boolean formula}

- Example:

- $\Phi = ((x_1 \rightarrow x_2) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg x_2$

- Assignment:  $\langle x_1 = 0, x_2 = 0, x_3 = 1, x_4 = 1 \rangle$

- $\Phi = ((0 \rightarrow 0) \vee \neg((\neg 0 \leftrightarrow 1) \vee 1)) \wedge \neg 0$

- $\Phi = (1 \vee \neg(1 \vee 1)) \wedge 1$



# Formula Satisfiability (SAT)

- The SAT problem asks whether a given Boolean formula is satisfiable

SAT = { $\langle \Phi \rangle$  :  $\Phi$  is a satisfiable Boolean formula}

- Example:

- $\Phi = ((x_1 \rightarrow x_2) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg x_2$

- Assignment:  $\langle x_1 = 0, x_2 = 0, x_3 = 1, x_4 = 1 \rangle$

- $\Phi = ((0 \rightarrow 0) \vee \neg((\neg 0 \leftrightarrow 1) \vee 1)) \wedge \neg 0$

- $\Phi = (1 \vee \neg(1 \vee 1)) \wedge 1$

- $\Phi = (1 \vee 0) \wedge 1$

# Formula Satisfiability (SAT)

- The SAT problem asks whether a given Boolean formula is satisfiable

SAT = { $\langle \Phi \rangle$  :  $\Phi$  is a satisfiable Boolean formula}

- Example:

- $\Phi = ((x_1 \rightarrow x_2) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg x_2$

- Assignment:  $\langle x_1 = 0, x_2 = 0, x_3 = 1, x_4 = 1 \rangle$

- $\Phi = ((0 \rightarrow 0) \vee \neg((\neg 0 \leftrightarrow 1) \vee 1)) \wedge \neg 0$

- $\Phi = (1 \vee \neg(1 \vee 1)) \wedge 1$

- $\Phi = (1 \vee 0) \wedge 1$

- $\Phi = 1$

# DPLL satisfiability solving

Given a Boolean formula  $\varphi$  in *clausal form* (an **AND** of **ORs**)

$$\{\{a, b\}, \{\neg a, b\}, \{a, \neg b\}, \{\neg a, \neg b\}\}$$

determine whether a *satisfying assignment* of variables to truth values exists.

# DPLL satisfiability solving

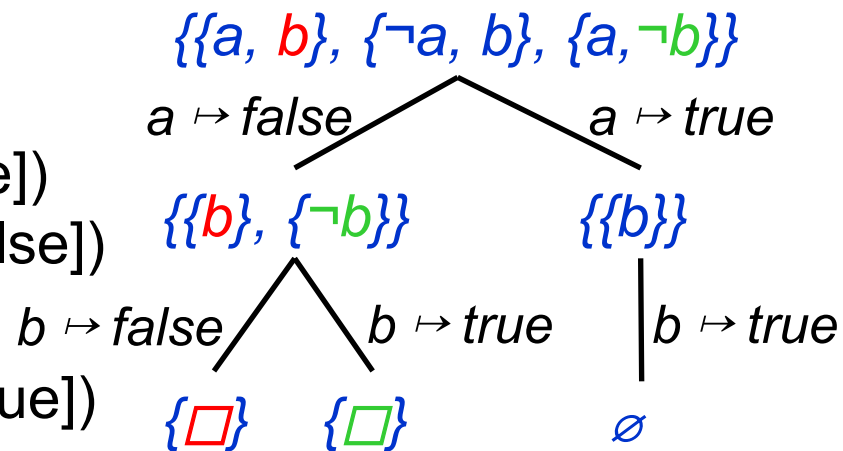
Given a Boolean formula  $\varphi$  in *clausal form* (an AND of ORs)

$$\{\{a, b\}, \{\neg a, b\}, \{a, \neg b\}, \{\neg a, \neg b\}\}$$

determine whether a *satisfying assignment* of variables to truth values exists.

Solvers based on Davis-Putnam-Logemann-Loveland algorithm:

1. If  $\varphi = \emptyset$  then SAT
2. If  $\square \in \varphi$  then UNSAT
3. If  $\varphi = \varphi' \cup \{x\}$  then  $\text{DPLL}(\varphi'[x \mapsto \text{true}])$   
If  $\varphi = \varphi' \cup \{\neg x\}$  then  $\text{DPLL}(\varphi'[x \mapsto \text{false}])$
4. Pick arbitrary  $x$  and return  
 $\text{DPLL}(\varphi[x \mapsto \text{false}]) \vee \text{DPLL}(\varphi[x \mapsto \text{true}])$



# DPLL satisfiability solving

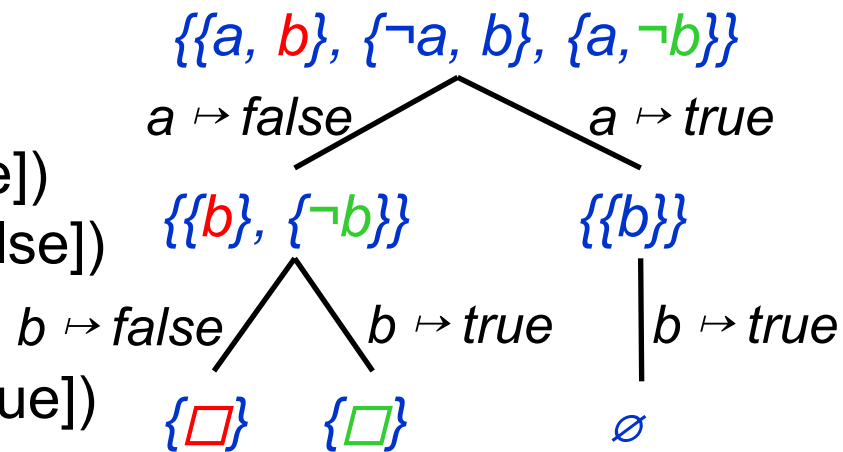
Given a Boolean formula  $\varphi$  in *clausal form* (an AND of ORs)

$$\{\{a, b\}, \{\neg a, b\}, \{a, \neg b\}, \{\neg a, \neg b\}\}$$

determine whether a *satisfying assignment* of variables to truth values exists.

Solvers based on Davis-Putnam-Logemann-Loveland algorithm:

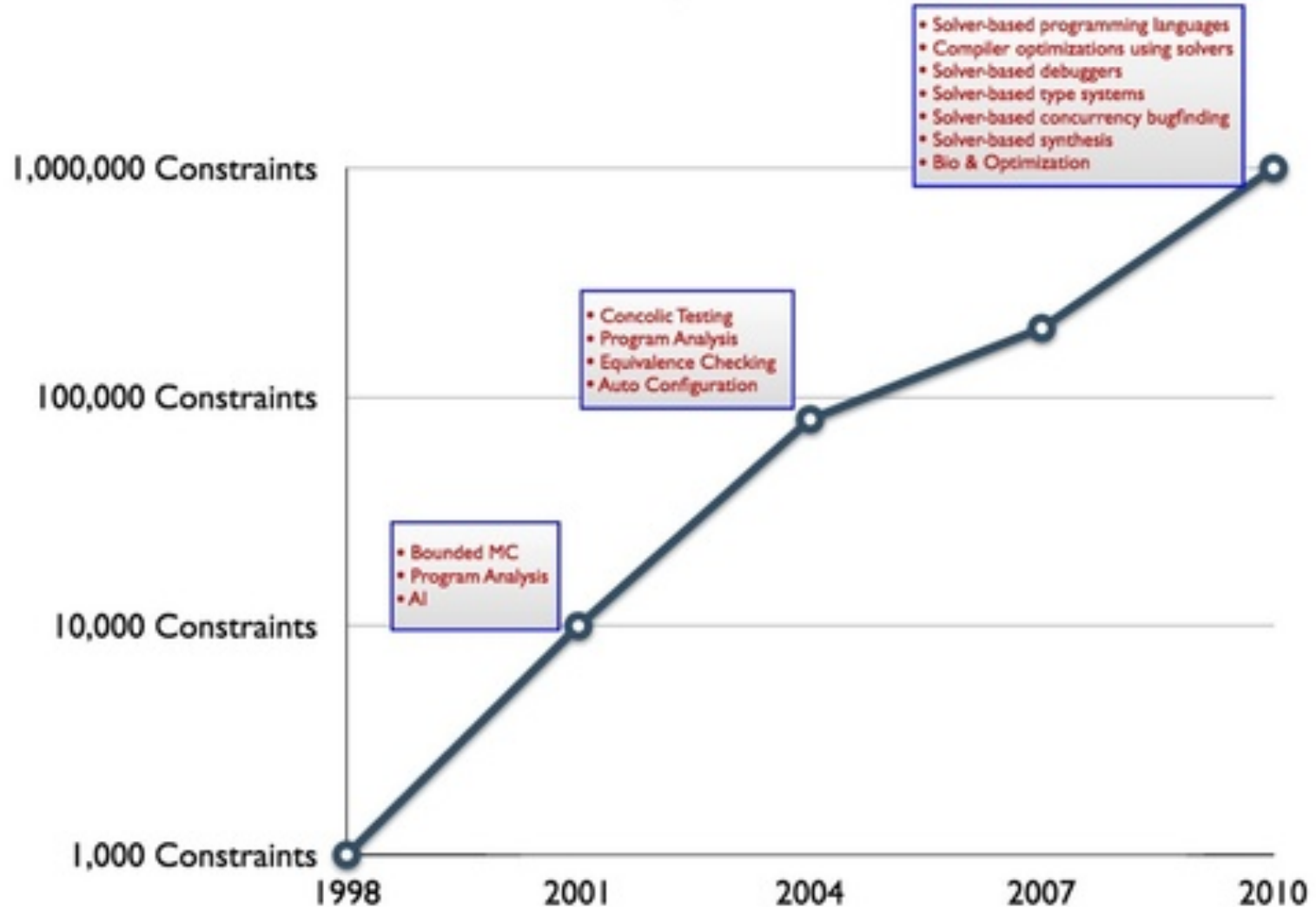
1. If  $\varphi = \emptyset$  then SAT
2. If  $\square \in \varphi$  then UNSAT
3. If  $\varphi = \varphi' \cup \{x\}$  then  $\text{DPLL}(\varphi'[x \mapsto \text{true}])$   
If  $\varphi = \varphi' \cup \{\neg x\}$  then  $\text{DPLL}(\varphi'[x \mapsto \text{false}])$
4. Pick arbitrary  $x$  and return  
 $\text{DPLL}(\varphi[x \mapsto \text{false}]) \vee \text{DPLL}(\varphi[x \mapsto \text{true}])$



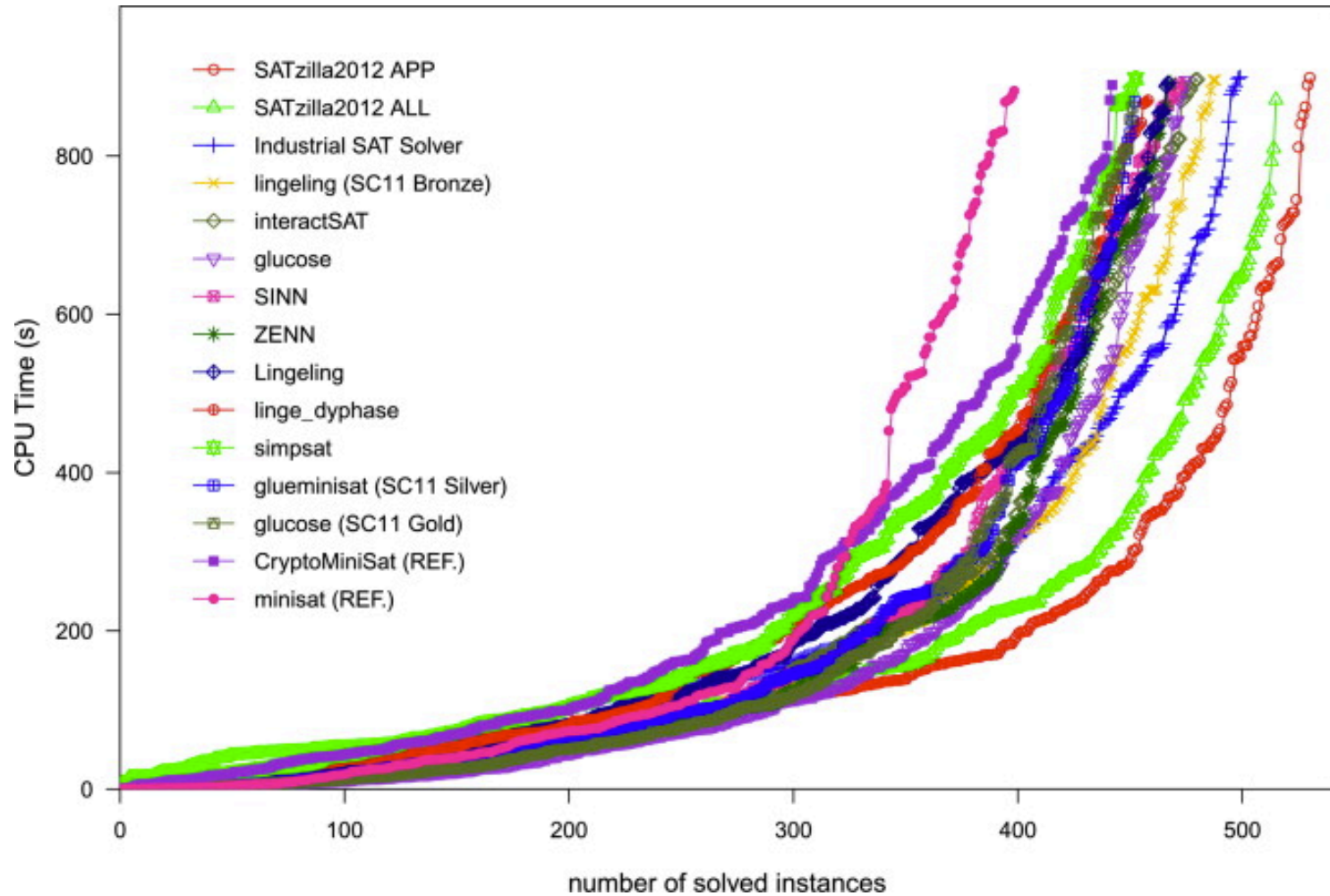
- + NP-complete but many heuristics and optimizations
- ⇒ can handle problems with 100,000's of variables

# SAT solving as enabling technology

## SAT/SMT Solver Research Story A 1000x Improvement

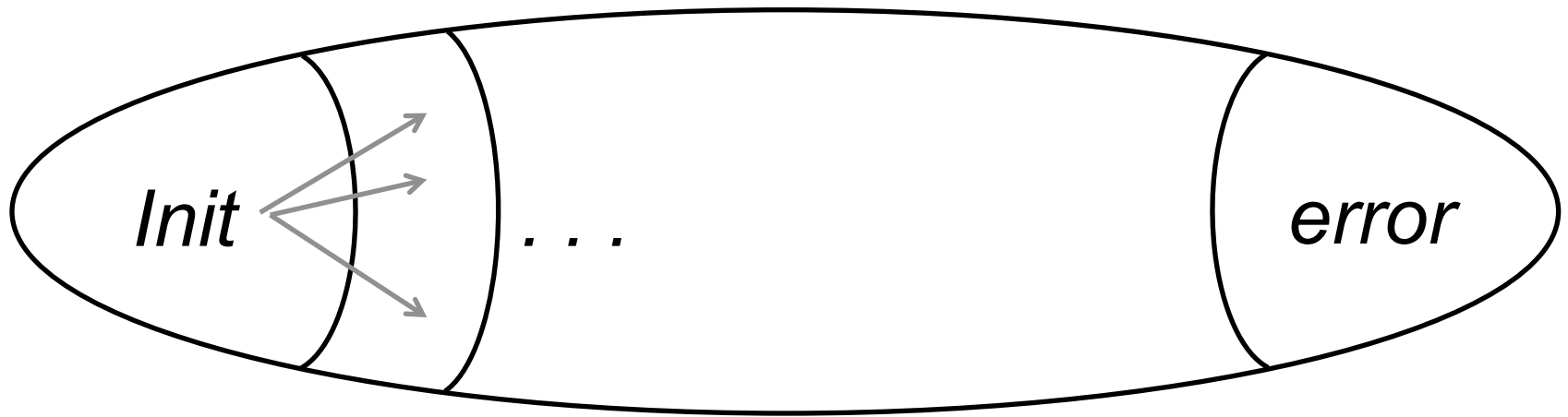


# SAT Competition



# Bounded Model Checking (BMC)

**MC:** check if a property holds for all states

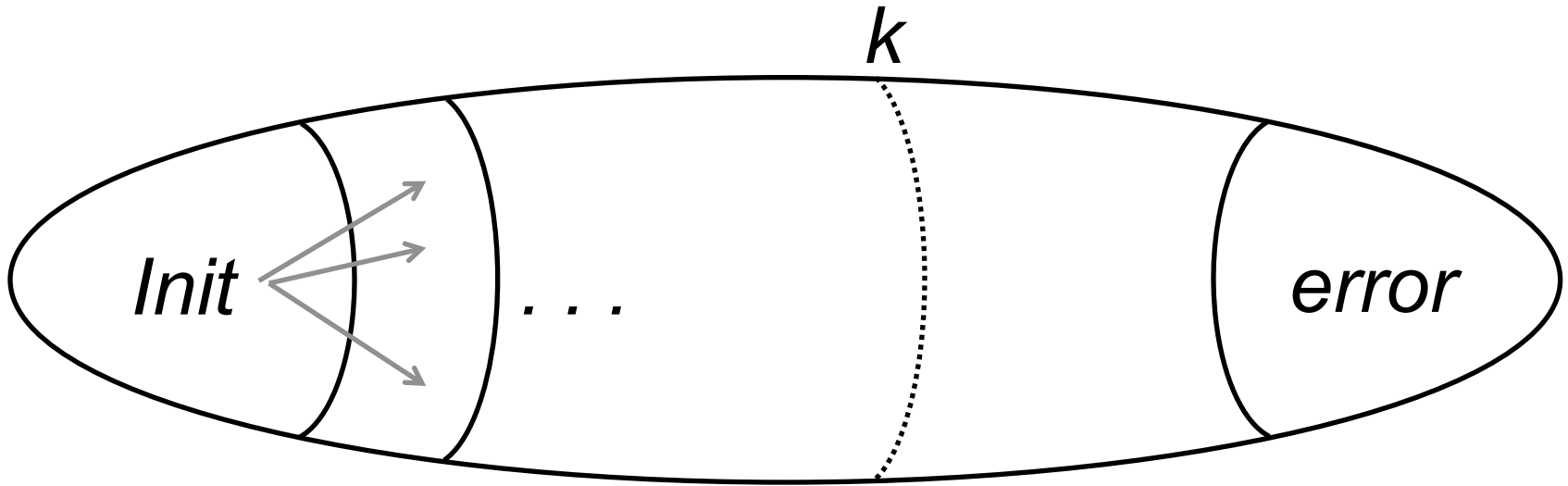




# Bounded Model Checking (BMC)

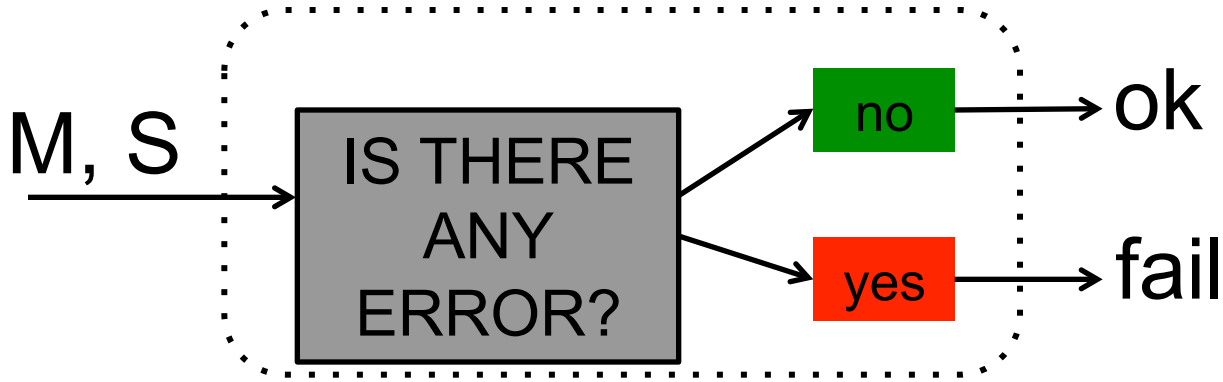
**MC:** check if a property holds for all states

**BMC:** check if a property holds for a subset of states



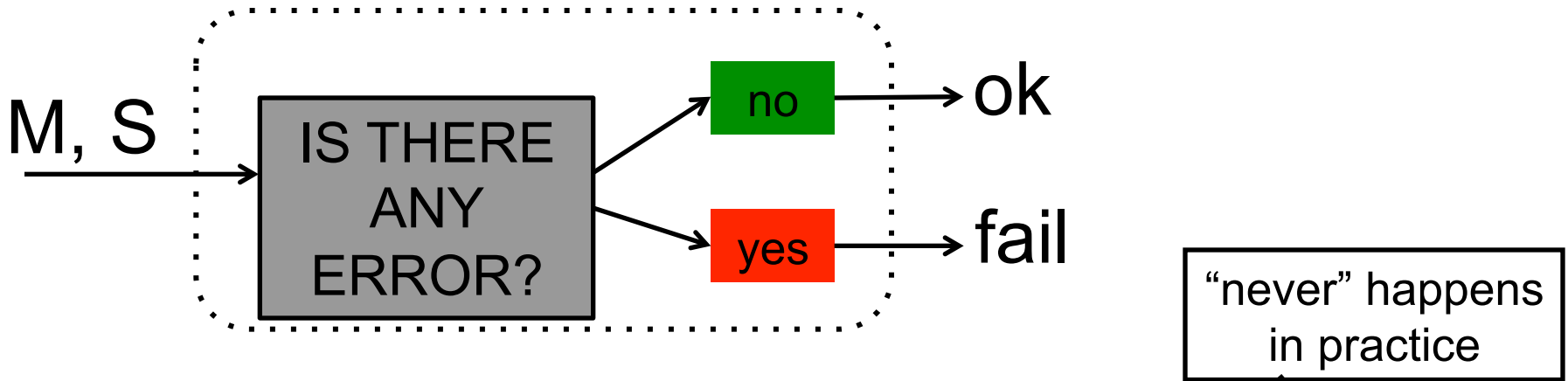
# Bounded Model Checking (BMC)

MC:

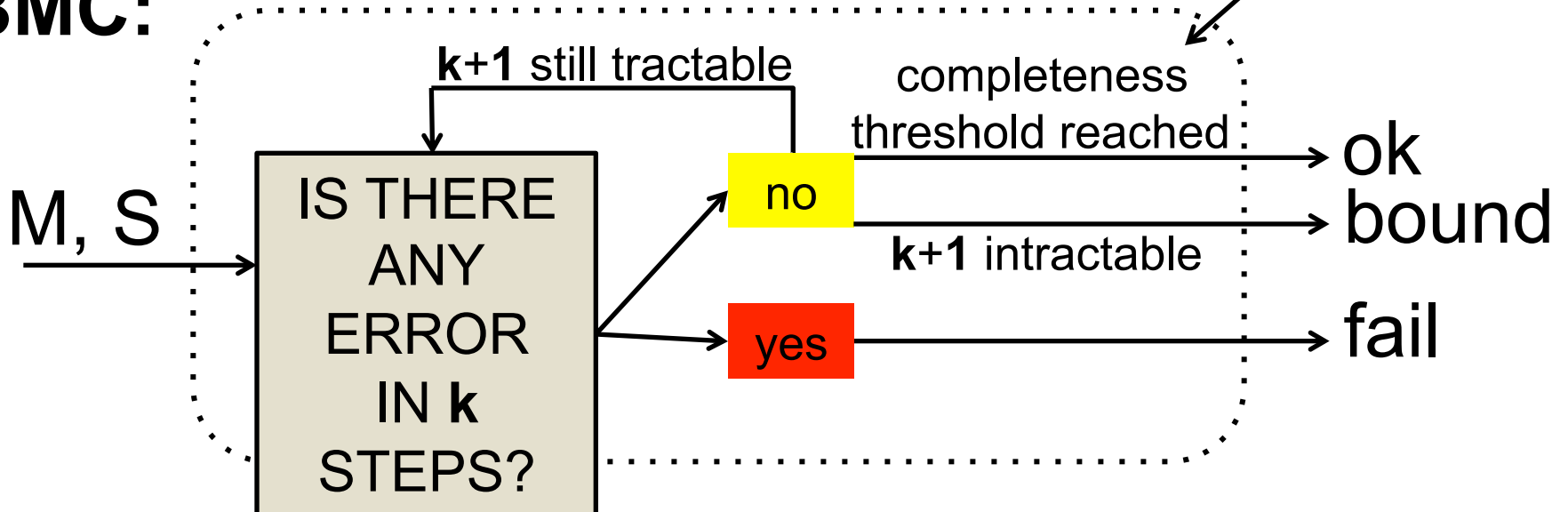


# Bounded Model Checking (BMC)

**MC:**

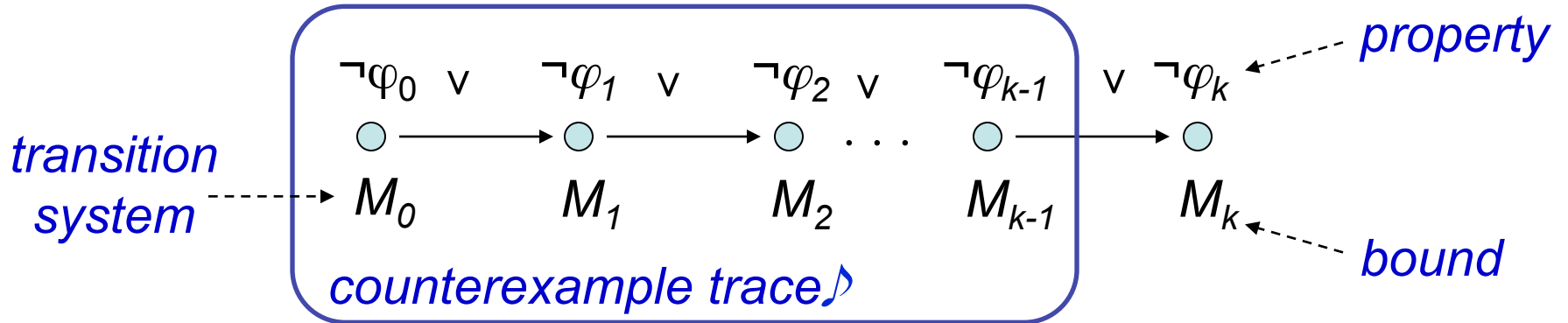


**BMC:**



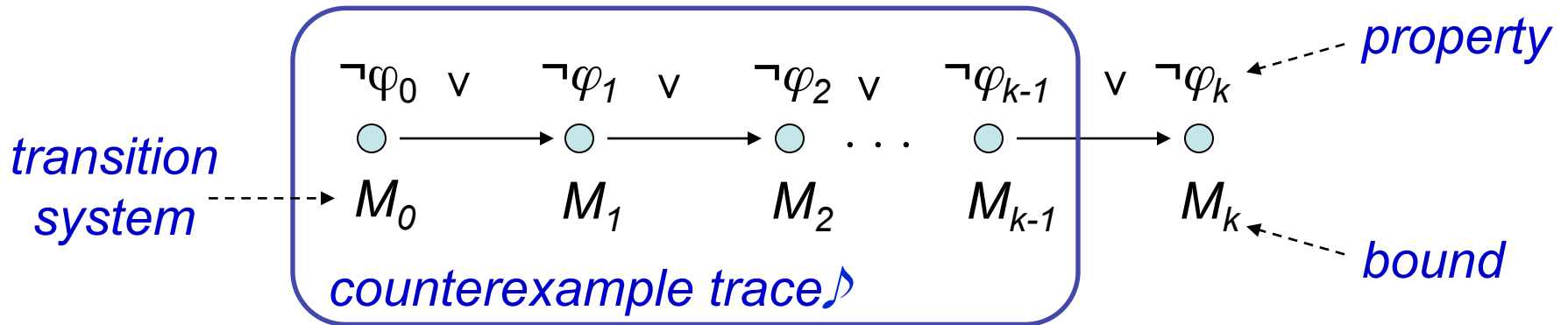
# Bounded Model Checking

Basic Idea: check negation of given property up to given depth



# Bounded Model Checking

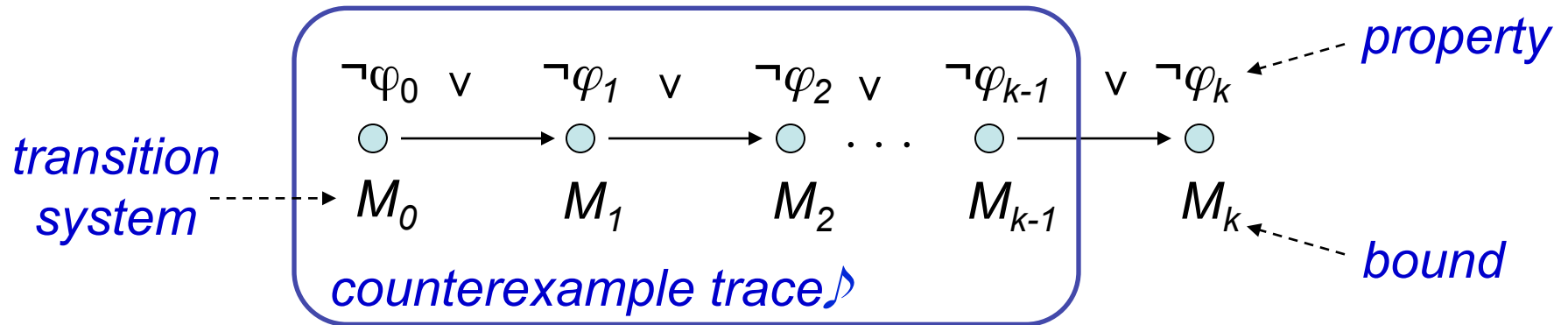
Basic Idea: check negation of given property up to given depth



- transition system  $M$  unrolled  $k$  times
  - for programs: unroll loops, unfold arrays, ...

# Bounded Model Checking

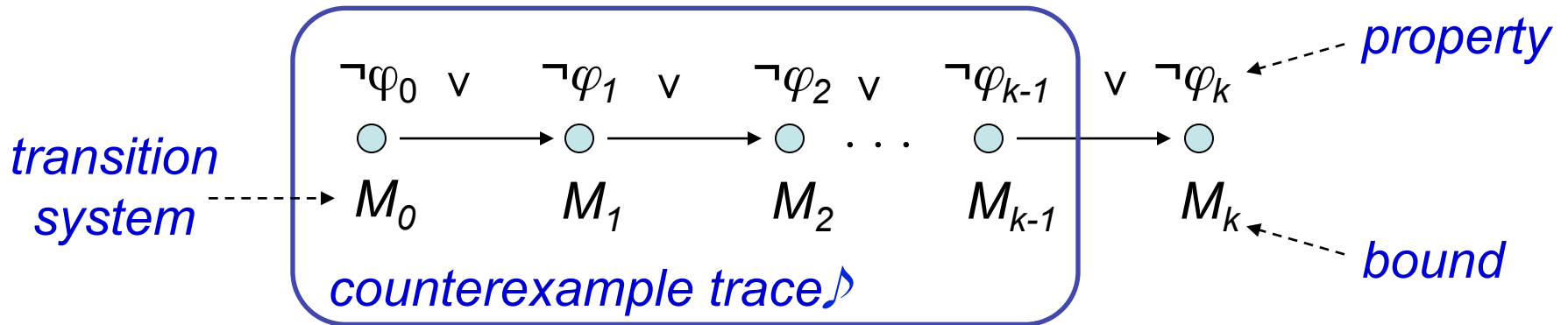
Basic Idea: check negation of given property up to given depth



- transition system  $M$  unrolled  $k$  times
  - for programs: unroll loops, unfold arrays, ...
- translated into verification condition  $\psi$  such that  $\psi$  **satisfiable iff  $\varphi$  has counterexample of max. depth  $k$**

# Bounded Model Checking

Basic Idea: check negation of given property up to given depth



- transition system  $M$  unrolled  $k$  times
  - for programs: unroll loops, unfold arrays, ...
- translated into verification condition  $\psi$  such that
  - $\psi$  **satisfiable iff  $\varphi$  has counterexample of max. depth  $k$**
- has been applied successfully to verify HW/SW systems

# Satisfiability Modulo Theories (1)

SMT decides the **satisfiability** of first-order logic formulae using the combination of different **background theories** (building-in operators)



# Satisfiability Modulo Theories (1)

SMT decides the **satisfiability** of first-order logic formulae using the combination of different **background theories** (building-in operators)

Theory	Example
Equality	$x_1 = x_2 \wedge \neg (x_1 = x_3) \Rightarrow \neg (x_1 = x_3)$

# Satisfiability Modulo Theories (1)

SMT decides the **satisfiability** of first-order logic formulae using the combination of different **background theories** (building-in operators)

Theory	Example
Equality	$x_1 = x_2 \wedge \neg (x_1 = x_3) \Rightarrow \neg (x_1 = x_3)$
Bit-vectors	$(b \gg i) \& 1 = 1$

# Satisfiability Modulo Theories (1)

SMT decides the **satisfiability** of first-order logic formulae using the combination of different **background theories** (building-in operators)

Theory	Example
Equality	$x_1 = x_2 \wedge \neg (x_1 = x_3) \Rightarrow \neg (x_1 = x_3)$
Bit-vectors	$(b \gg i) \& 1 = 1$
Linear arithmetic	$(4y_1 + 3y_2 \geq 4) \vee (y_2 - 3y_3 \leq 3)$

# Satisfiability Modulo Theories (1)

SMT decides the **satisfiability** of first-order logic formulae using the combination of different **background theories** (building-in operators)

Theory	Example
Equality	$x_1 = x_2 \wedge \neg (x_1 = x_3) \Rightarrow \neg (x_1 = x_3)$
Bit-vectors	$(b \gg i) \& 1 = 1$
Linear arithmetic	$(4y_1 + 3y_2 \geq 4) \vee (y_2 - 3y_3 \leq 3)$
Arrays	$(j = k \wedge a[k]=2) \Rightarrow a[j]=2$

# Satisfiability Modulo Theories (1)

SMT decides the **satisfiability** of first-order logic formulae using the combination of different **background theories** (building-in operators)

Theory	Example
Equality	$x_1 = x_2 \wedge \neg (x_1 = x_3) \Rightarrow \neg (x_1 = x_3)$
Bit-vectors	$(b \gg i) \& 1 = 1$
Linear arithmetic	$(4y_1 + 3y_2 \geq 4) \vee (y_2 - 3y_3 \leq 3)$
Arrays	$(j = k \wedge a[k] = 2) \Rightarrow a[j] = 2$
Combined theories	$(j \leq k \wedge a[j] = 2) \Rightarrow a[i] < 3$

# Satisfiability Modulo Theories (2)

- Given

- a decidable  $\Sigma$ -theory  $T$
- a quantifier-free formula  $\varphi$

$\varphi$  is **T-satisfiable** iff  $T \cup \{\varphi\}$  is satisfiable, i.e., there exists a structure that satisfies both formula and sentences of  $T$

# Satisfiability Modulo Theories (2)

- Given

- a decidable  $\Sigma$ -theory  $T$
- a quantifier-free formula  $\varphi$

$\varphi$  is **T-satisfiable** iff  $T \cup \{\varphi\}$  is satisfiable, i.e., there exists a structure that satisfies both formula and sentences of  $T$

- Given

- a set  $\Gamma \cup \{\varphi\}$  of first-order formulae over  $T$

$\varphi$  is a **T-consequence of  $\Gamma$**  ( $\Gamma \models_T \varphi$ ) iff every model of  $T \cup \Gamma$  is also a model of  $\varphi$

# Satisfiability Modulo Theories (2)

- Given

- a decidable  $\Sigma$ -theory  $T$
- a quantifier-free formula  $\varphi$

$\varphi$  is **T-satisfiable** iff  $T \cup \{\varphi\}$  is satisfiable, i.e., there exists a structure that satisfies both formula and sentences of  $T$

- Given

- a set  $\Gamma \cup \{\varphi\}$  of first-order formulae over  $T$

$\varphi$  is a **T-consequence of  $\Gamma$**  ( $\Gamma \vDash_T \varphi$ ) iff every model of  $T \cup \Gamma$  is also a model of  $\varphi$

- Checking  $\Gamma \vDash_T \varphi$  can be reduced in the usual way to checking the T-satisfiability of  $\Gamma \cup \{\neg\varphi\}$



# Satisfiability Modulo Theories (3)

- let **a** be an array, **b**, **c** and **d** be signed bit-vectors of width 16, 32 and 32 respectively, and let **g** be an unary function.

# Satisfiability Modulo Theories (3)

- let **a** be an array, **b**, **c** and **d** be signed bit-vectors of width 16, 32 and 32 respectively, and let **g** be an unary function.

$$g(\text{select}(\text{store}(a, c, 12)), \text{SignExt}(b, 16) + 3)$$

$$\neq g(\text{SignExt}(b, 16) - c + 4) \wedge \text{SignExt}(b, 16) = c - 3 \wedge c + 1 = d - 4$$

# Satisfiability Modulo Theories (3)

- let **a** be an array, **b**, **c** and **d** be signed bit-vectors of width 16, 32 and 32 respectively, and let **g** be an unary function.

$$g(\text{select}(\text{store}(a, c, 12)), \text{SignExt}(b, 16) + 3) \\ \neq g(\text{SignExt}(b, 16) - c + 4) \wedge \text{SignExt}(b, 16) = c - 3 \wedge c + 1 = d - 4$$



**b'** extends **b** to the signed equivalent bit-vector of size 32

$$\text{step 1: } g(\text{select}(\text{store}(a, c, 12), b' + 3)) \neq g(b' - c + 4) \wedge b' = c - 3 \wedge c + 1 = d - 4$$

# Satisfiability Modulo Theories (3)

- let **a** be an array, **b**, **c** and **d** be signed bit-vectors of width 16, 32 and 32 respectively, and let **g** be an unary function.

$$g(\text{select}(\text{store}(a, c, 12)), \text{SignExt}(b, 16) + 3) \\ \neq g(\text{SignExt}(b, 16) - c + 4) \wedge \text{SignExt}(b, 16) = c - 3 \wedge c + 1 = d - 4$$

↓ **b'** extends **b** to the signed equivalent bit-vector of size 32

$$\text{step 1: } g(\text{select}(\text{store}(a, c, 12), b' + 3)) \neq g(b' - c + 4) \wedge b' = c - 3 \wedge c + 1 = d - 4$$

↓ replace **b'** by **c-3** in the inequality

$$\text{step 2: } g(\text{select}(\text{store}(a, c, 12), c - 3 + 3)) \neq g(c - 3 - c + 4) \wedge c - 3 = c - 3 \wedge c + 1 = d - 4$$

# Satisfiability Modulo Theories (3)

- let **a** be an array, **b**, **c** and **d** be signed bit-vectors of width 16, 32 and 32 respectively, and let **g** be an unary function.

$$g(\text{select}(\text{store}(a, c, 12)), \text{SignExt}(b, 16) + 3) \\ \neq g(\text{SignExt}(b, 16) - c + 4) \wedge \text{SignExt}(b, 16) = c - 3 \wedge c + 1 = d - 4$$

↓ **b'** extends **b** to the signed equivalent bit-vector of size 32

$$\text{step 1: } g(\text{select}(\text{store}(a, c, 12), b' + 3)) \neq g(b' - c + 4) \wedge b' = c - 3 \wedge c + 1 = d - 4$$

↓ replace **b'** by **c-3** in the inequality

$$\text{step 2: } g(\text{select}(\text{store}(a, c, 12), c - 3 + 3)) \neq g(c - 3 - c + 4) \wedge c - 3 = c - 3 \wedge c + 1 = d - 4$$

↓ using facts about bit-vector arithmetic

$$\text{step 3: } g(\text{select}(\text{store}(a, c, 12), c)) \neq g(1) \wedge c - 3 = c - 3 \wedge c + 1 = d - 4$$

# Satisfiability Modulo Theories (4)

*step 3:  $g(\text{select}(\text{store}(a, c, 12), c)) \neq g(1) \wedge c - 3 = c - 3 \wedge c + 1 = d - 4$*

# Satisfiability Modulo Theories (4)

*step 3*:  $g(\text{select}(\text{store}(a, c, 12), c)) \neq g(1) \wedge c - 3 = c - 3 \wedge c + 1 = d - 4$

↓ applying the theory of arrays

*step 4*:  $g(12) \neq g(1) \wedge c - 3 \wedge c + 1 = d - 4$

# Satisfiability Modulo Theories (4)

*step 3*:  $g(\text{select}(\text{store}(a, c, 12), c)) \neq g(1) \wedge c - 3 = c - 3 \wedge c + 1 = d - 4$

↓ applying the theory of arrays

*step 4*:  $g(12) \neq g(1) \wedge c - 3 \wedge c + 1 = d - 4$

↓ The function  $g$  implies that for all  $x$  and  $y$ ,  
if  $x = y$ , then  $g(x) = g(y)$  (*congruence rule*).

*step 5*: SAT ( $c = 5, d = 10$ )



# Satisfiability Modulo Theories (4)

*step 3*:  $g(\text{select}(\text{store}(a, c, 12), c)) \neq g(1) \wedge c - 3 = c - 3 \wedge c + 1 = d - 4$

↓ applying the theory of arrays

*step 4*:  $g(12) \neq g(1) \wedge c - 3 \wedge c + 1 = d - 4$

↓ The function  $g$  implies that for all  $x$  and  $y$ ,  
if  $x = y$ , then  $g(x) = g(y)$  (*congruence rule*).

*step 5*: SAT ( $c = 5, d = 10$ )

- SMT solvers also apply:
  - standard algebraic reduction rules
  - contextual simplification

$$r \wedge \text{false} \mapsto \text{false}$$

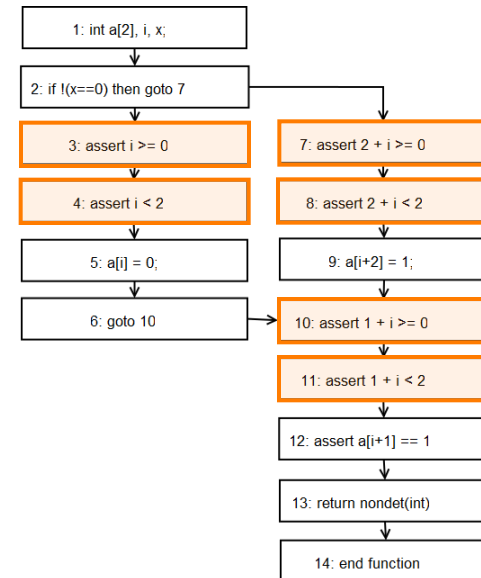
$$a = 7 \wedge p(a) \mapsto a = 7 \wedge p(7)$$

# BMC of Software

- program modelled as state transition system
  - state: program counter and program variables
  - derived from control-flow graph
  - checked safety properties give extra nodes
- program unfolded up to given bounds
  - loop iterations
  - context switches
- unfolded program optimized to reduce blow-up
  - constant propagation
  - forward substitutions

} crucial

```
int main() {  
  int a[2], i, x;  
  if (x==0)  
    a[i]=0;  
  else  
    a[i+2]=1;  
  assert(a[i+1]==1);  
}
```



# BMC of Software

- program modelled as state transition system
  - state: program counter and program variables
  - derived from control-flow graph
  - checked safety properties give extra nodes
- program unfolded up to given bounds
  - loop iterations
  - context switches
- unfolded program optimized to reduce blow-up
  - constant propagation
  - forward substitutions } crucial
- front-end converts unrolled and optimized program into SSA

```
int main() {  
    int a[2], i, x;  
    if (x==0)  
        a[i]=0;  
    else  
        a[i+2]=1;  
    assert(a[i+1]==1);  
}
```



```
 $g_1 = x_1 == 0$   
 $a_1 = a_0$  WITH  $[i_0 := 0]$   
 $a_2 = a_0$   
 $a_3 = a_2$  WITH  $[2+i_0 := 1]$   
 $a_4 = g_1 ? a_1 : a_3$   
 $t_1 = a_4[1+i_0] == 1$ 
```

# BMC of Software

- program modelled as state transition system
  - state: program counter and program variables
  - derived from control-flow graph
  - checked safety properties give extra nodes
- program unfolded up to given bounds
  - loop iterations
  - context switches
- unfolded program optimized to reduce blow-up
  - constant propagation } crucial
  - forward substitutions }
- front-end converts unrolled and optimized program into SSA
- extraction of constraints C and properties P
  - specific to selected SMT solver, uses theories
- satisfiability check of  $C \wedge \neg P$

```

int main() {
  int a[2], i, x;
  if (x==0)
    a[i]=0;
  else
    a[i+2]=1;
  assert(a[i+1]==1);
}

```



$$C := \left[ \begin{array}{l} g_1 := (x_1 = 0) \\ \wedge a_1 := \text{store}(a_0, i_0, 0) \\ \wedge a_2 := a_0 \\ \wedge a_3 := \text{store}(a_2, 2 + i_0, 1) \\ \wedge a_4 := \text{ite}(g_1, a_1, a_3) \end{array} \right]$$

$$P := \left[ \begin{array}{l} i_0 \geq 0 \wedge i_0 < 2 \\ \wedge 2 + i_0 \geq 0 \wedge 2 + i_0 < 2 \\ \wedge 1 + i_0 \geq 0 \wedge 1 + i_0 < 2 \\ \wedge \text{select}(a_4, i_0 + 1) = 1 \end{array} \right]$$

# Encoding of Numeric Types

- SMT solvers typically provide different encodings for numbers:
  - abstract domains (**Z**, **R**)
  - fixed-width bit vectors (unsigned int, ...)
    - ▷ “internalized bit-blasting”

# Encoding of Numeric Types

- SMT solvers typically provide different encodings for numbers:
  - abstract domains (**Z**, **R**)
  - fixed-width bit vectors (unsigned int, ...)
    - ▷ “internalized bit-blasting”
- verification results can depend on encodings

$$(a > 0) \wedge (b > 0) \Rightarrow (a + b > 0)$$

# Encoding of Numeric Types

- SMT solvers typically provide different encodings for numbers:
  - abstract domains (**Z**, **R**)
  - fixed-width bit vectors (unsigned int, ...)
    - ▷ “internalized bit-blasting”
- verification results can depend on encodings

$$(a > 0) \wedge (b > 0) \Rightarrow (a + b > 0)$$

*valid in abstract domains  
such as **Z** or **R***

*doesn't hold for bitvectors,  
due to possible overflows*

# Encoding of Numeric Types

- SMT solvers typically provide different encodings for numbers:
  - abstract domains (**Z**, **R**)
  - fixed-width bit vectors (unsigned int, ...)
    - ▷ “internalized bit-blasting”
- verification results can depend on encodings

$$(a > 0) \wedge (b > 0) \Rightarrow (a + b > 0)$$

*valid in abstract domains  
such as **Z** or **R***

*doesn't hold for bitvectors,  
due to possible overflows*

- majority of VCs solved faster if numeric types are modelled by abstract domains but possible loss of precision
- ESBMC supports both types of encoding and also combines them to improve scalability and precision



# Encoding Numeric Types as Bitvectors

Bitvector encodings need to handle

- type casts and implicit conversions
  - arithmetic conversions implemented using word-level functions (part of the bitvector theory: Extract, SignExt, ...)
    - o different conversions for every pair of types
    - o uses type information provided by front-end

# Encoding Numeric Types as Bitvectors

Bitvector encodings need to handle

- type casts and implicit conversions
  - arithmetic conversions implemented using word-level functions (part of the bitvector theory: Extract, SignExt, ...)
    - o different conversions for every pair of types
    - o uses type information provided by front-end
  - conversion to / from bool via if-then-else operator
    - t = ite(v ≠ k, true, false) //conversion to bool
    - v = ite(t, 1, 0) //conversion from bool

# Encoding Numeric Types as Bitvectors

Bitvector encodings need to handle

- type casts and implicit conversions
  - arithmetic conversions implemented using word-level functions (part of the bitvector theory: Extract, SignExt, ...)
    - o different conversions for every pair of types
    - o uses type information provided by front-end
  - conversion to / from bool via if-then-else operator
    - `t = ite(v ≠ k, true, false) //conversion to bool`
    - `v = ite(t, 1, 0) //conversion from bool`
- arithmetic over- / underflow
  - standard requires modulo-arithmetic for unsigned integer
    - $\text{unsigned\_overflow} \Leftrightarrow (r - (r \bmod 2^w)) < 2^w$

# Encoding Numeric Types as Bitvectors

Bitvector encodings need to handle

- type casts and implicit conversions
  - arithmetic conversions implemented using word-level functions (part of the bitvector theory: Extract, SignExt, ...)
    - o different conversions for every pair of types
    - o uses type information provided by front-end
  - conversion to / from bool via if-then-else operator
    - $t = \text{ite}(v \neq k, \text{true}, \text{false})$  //conversion to bool
    - $v = \text{ite}(t, 1, 0)$  //conversion from bool
- arithmetic over- / underflow
  - standard requires modulo-arithmetic for unsigned integer
    - $\text{unsigned\_overflow} \Leftrightarrow (r - (r \bmod 2^w)) < 2^w$
  - define error literals to detect over- / underflow for other types
    - $\text{res\_op} \Leftrightarrow \neg \text{overflow}(x, y) \wedge \neg \text{underflow}(x, y)$
    - o similar to conversions

# Floating-Point Numbers

- Over-approximate floating-point by fixed-point numbers
  - encode the integral (i) and fractional (f) parts

# Floating-Point Numbers

- Over-approximate floating-point by fixed-point numbers
  - encode the integral (i) and fractional (f) parts
- **Binary encoding:** get a new bit-vector  $b = i @ f$  with the same bitwidth before and after the radix point of a.

$$i = \begin{cases} \text{Extract}(b, n_b + m_a - 1, n_b) & : m_a \leq m_b & // m = \text{number of bits of } i \\ \text{SignExt}(\text{Extract}(b, t_b - 1, n_b), m_a - m_b) & : \text{otherwise} & \text{bits of } i \end{cases}$$
$$f = \begin{cases} \text{Extract}(b, n_b - 1, n_b - n_b) & : n_a \leq n_b & // n = \text{number of bits of } f \\ \text{Extract}(b, n_b, 0) @ \text{SignExt}(b, n_a - n_b) & : \text{otherwise} & \end{cases}$$

# Floating-Point Numbers

- Over-approximate floating-point by fixed-point numbers
  - encode the integral (i) and fractional (f) parts
- **Binary encoding:** get a new bit-vector  $b = i @ f$  with the same bitwidth before and after the radix point of  $a$ .

$$i = \begin{cases} \text{Extract}(b, n_b + m_a - 1, n_b) & : m_a \leq m_b & // m = \text{number of bits of } i \\ \text{SignExt}(\text{Extract}(b, t_b - 1, n_b), m_a - m_b) & : \text{otherwise} & // m = \text{number of bits of } i \end{cases}$$

$$f = \begin{cases} \text{Extract}(b, n_b - 1, n_b - n_b) & : n_a \leq n_b & // n = \text{number of bits of } f \\ \text{Extract}(b, n_b, 0) @ \text{SignExt}(b, n_a - n_b) & : \text{otherwise} & // n = \text{number of bits of } f \end{cases}$$

- **Rational encoding:** convert  $a$  to a rational number

$$a = \begin{cases} \frac{\left( i * p + \left( \frac{f * p}{2^n} + 1 \right) \right)}{p} & : f \neq 0 & // p = \text{number of decimal places} \\ i & : \text{otherwise} \end{cases}$$

# Floating-point SMT Encoding

- The SMT floating-point theory is an addition to the SMT standard, proposed in 2010 and formalises:
  - Floating-point arithmetic



# Floating-point SMT Encoding

- The SMT floating-point theory is an addition to the SMT standard, proposed in 2010 and formalises:
  - Floating-point arithmetic
  - Positive and negative infinities and zeroes

# Floating-point SMT Encoding

- The SMT floating-point theory is an addition to the SMT standard, proposed in 2010 and formalises:
  - Floating-point arithmetic
  - Positive and negative infinities and zeroes
  - NaNs

# Floating-point SMT Encoding

- The SMT floating-point theory is an addition to the SMT standard, proposed in 2010 and formalises:
  - Floating-point arithmetic
  - Positive and negative infinities and zeroes
  - NaNs
  - Comparison operators

# Floating-point SMT Encoding

- The SMT floating-point theory is an addition to the SMT standard, proposed in 2010 and formalises:
  - Floating-point arithmetic
  - Positive and negative infinities and zeroes
  - NaNs
  - Comparison operators
  - Five rounding modes: round nearest with ties choosing the even value, round nearest with ties choosing away from zero, round towards zero, round towards positive infinity and round towards negative infinity

# Floating-point SMT Encoding

- Missing from the standard:
  - Floating-point exceptions
  - Signaling NaNs

# Floating-point SMT Encoding

- Missing from the standard:
  - Floating-point exceptions
  - Signaling NaNs
- Two solvers currently support the standard:
  - Z3: implements all operators
  - MathSAT: implements all but two operators
    - o *fp.rem*: remainder:  $x - y * n$ , where  $n$  in  $Z$  is nearest to  $x/y$
    - o *fp.fma*: fused multiplication and addition;  $(x * y) + z$

# Floating-point SMT Encoding

- Missing from the standard:
  - Floating-point exceptions
  - Signaling NaNs
- Two solvers currently support the standard:
  - Z3: implements all operators
  - MathSAT: implements all but two operators
    - *fp.rem*: remainder:  $x - y * n$ , where  $n$  in  $Z$  is nearest to  $x/y$
    - *fp.fma*: fused multiplication and addition;  $(x * y) + z$
- Both solvers offer non-standard functions:
  - *fp\_as\_ieeebv*: converts floating-point to bitvectors
  - *fp\_from\_ieeebv*: converts bitvectors to floating-point

# How to encode Floating-point programs?

- Most operations performed at program-level to encode FP numbers have a **one-to-one conversion to SMT**
- Special cases being casts to boolean types and the `fp.eq` operator
  - Usually, cast operations are encoded using **extend/extract operation**
  - Extending floating-point numbers is non-trivial because of the format

```
int main()  
{  
    _Bool c;  
  
    double b = 0.0f;  
    b = c;  
    assert(b != 0.0f);  
  
    c = b;  
    assert(c != 0);  
}
```



# Cast to/from booleans

- Simpler solutions:
  - Casting **booleans** to **floating-point numbers** can be done using an ite operator

```
(assert (= (ite |main::c|  
            (fp #b0 #b01111111111 #x0000000000000000)  
            (fp #b0 #b000000000000 #x0000000000000000))  
          |main::b|))
```

# Cast to/from booleans

- Simpler solutions:
  - Casting **booleans** to **floating-point numbers** can be done using an ite operator

*If true, assign 1f to b*

```
(assert (= (ite |main::c|  
  (fp #b0 #b01111111111 #x0000000000000000)  
  (fp #b0 #b00000000000 #x0000000000000000))  
  |main::b|))
```

# Cast to/from booleans

- Simpler solutions:
  - Casting **booleans** to **floating-point numbers** can be done using an ite operator

```
(assert (= (ite |main::c|  
            (fp #b0 #b0111111111 #x0000000000000000)  
            (fp #b0 #b000000000000 #x0000000000000000))  
          |main::b|))
```



*Otherwise, assign 0f to b*

# Cast to/from booleans

- Simpler solutions:
  - Casting **floating-point numbers** to **booleans** can be done using an equality and one not:

```
(assert (= (not (fp.eq |main::b|
                 (fp #b0 #b000000000000 #x0000000000000000)))
         |main::c|))
```

:note

"(fp.eq x y) evaluates to true if x evaluates to -zero and y to +zero, or vice versa. fp.eq and all the other comparison operators evaluate to false if one of their arguments is NaN."

# Cast to/from booleans

- Simpler solutions:
  - Casting **floating-point numbers to booleans** can be done using an equality and one not:

```
(assert (= (not (fp.eq |main::b|  
  (fp #b0 #b000000000000 #x0000000000000000)))  
  |main::c|))
```

*true when the  
floating is not 0.0*

:note

"(fp.eq x y) evaluates to true if x evaluates to -zero and y to +zero, or vice versa. fp.eq and all the other comparison operators evaluate to false if one of their arguments is NaN."

# Cast to/from booleans

- Simpler solutions:
  - Casting **floating-point numbers** to **booleans** can be done using an equality and one not:

```
(assert (= (not (fp.eq |main::b|  
  (fp #b0 #b000000000000 #x0000000000000000)))  
  |main::c|))
```

 *otherwise, the result is false*

:note

"(fp.eq x y) evaluates to true if x evaluates to -zero and y to +zero, or vice versa. fp.eq and all the other comparison operators evaluate to false if one of their arguments is NaN."

# Cast to/from booleans

- Simpler solutions:
  - Casting **floating-point numbers** to **booleans** can be done using an equality and one not:

```
(assert (= (not (fp.eq |main::b|  
                (fp #b0 #b000000000000 #x0000000000000000)))  
         |main::c|))
```

:note

"(fp.eq x y) evaluates to true if x evaluates to -zero and y to +zero, or vice versa. fp.eq and all the other comparison operators evaluate to false if one of their arguments is NaN."

# Floating-point Encoding: Illustrative Example

```
int main()  
{  
    float x;  
    float y = x;  
    assert (x==y) ;  
    return 0;  
}
```



# Floating-point Encoding: Illustrative Example

```
; declaration of x and y
(declare-fun |main::x| () (_ FloatingPoint 8 24))
(declare-fun |main::y| () (_ FloatingPoint 8 24))

; symbol created to represent a nondeterministic number
(declare-fun |nondet_symex::nondet0| () (_ FloatingPoint 8 24))

; Global guard, used for checking properties
(declare-fun |execution_statet::\guard_exec| () Bool)

; assign the nondeterministic symbol to x
(assert (= |nondet_symex::nondet0| |main::x|))

; assign x to y
(assert (= |main::x| |main::y|))

; assert x == y
(assert (let ((a!1 (not (=> true
                        (=> |execution_statet::\guard_exec|
                            (fp.eq |main::x| |main::y|))))))
        (or a!1))))
```

# Floating-point Encoding: Illustrative Example

```
; declaration of x and y
(declare-fun |main::x| () (_ FloatingPoint 8 24))
(declare-fun |main::y| () (_ FloatingPoint 8 24))
```

```
; symbol created to represent a nondeterministic number
(declare-fun |nondet_symex::nondet0| () (_ FloatingPoint 8 24))
```

```
; Global guard, used for checking properties
(declare-fun |execution_statet::\guard_exec| () Bool)
```

```
; assign the nondeterministic symbol to x
(assert (= |nondet_symex::nondet0| |main::x|))
```

```
; assign x to y
(assert (= |main::x| |main::y|))
```

**Variable declarations**

```
; assert x == y
(assert (let ((a!1 (not (=> true
                        (=> |execution_statet::\guard_exec|
                            (fp.eq |main::x| |main::y|))))))
          (or a!1))))
```

# Floating-point Encoding: Illustrative Example

```
; declaration of x and y
(declare-fun |main::x| () (_ FloatingPoint 8 24))
(declare-fun |main::y| () (_ FloatingPoint 8 24))
```

```
; symbol created to represent a nondeterministic number
(declare-fun |nondet_symex::nondet0| () (_ FloatingPoint 8 24))
```

```
; Global guard, used for checking properties
(declare-fun |execution_statet::\guard_exec| () Bool)
```

```
; assign the nondeterministic symbol to x
(assert (= |nondet_symex::nondet0| |main::x|))
```

```
; assign x to y
(assert (= |main::x| |main::y|))
```

```
; assert x == y
(assert (let ((a!1 (not (=> true
                        (=> |execution_statet::\guard_exec|
                            (fp.eq |main::x| |main::y|))))))
          (or a!1))))
```

**Nondeterministic symbol  
declaration (optional)**



# Floating-point Encoding: Illustrative Example

```
; declaration of x and y
(declare-fun |main::x| () (_ FloatingPoint 8 24))
(declare-fun |main::y| () (_ FloatingPoint 8 24))

; symbol created to represent a nondeterministic number
(declare-fun |nondet_symex::nondet0| () (_ FloatingPoint 8 24))
```

```
; Global guard, used for checking properties
(declare-fun |execution_statet::\guard_exec| () Bool)
```

```
; assign the nondeterministic symbol to x
(assert (= |nondet_symex::nondet0| |main::x|))
```

```
; assign x to y
(assert (= |main::x| |main::y|))
```

```
; assert x == y
(assert (let ((a!1 (not (=> true
                        (=> |execution_statet::\guard_exec|
                            (fp.eq |main::x| |main::y|))))))
          (or a!1))))
```

Guard used to check  
satisfiability

# Floating-point Encoding: Illustrative Example

```
; declaration of x and y
(declare-fun |main::x| () (_ FloatingPoint 8 24))
(declare-fun |main::y| () (_ FloatingPoint 8 24))

; symbol created to represent a nondeterministic number
(declare-fun |nondet_symex::nondet0| () (_ FloatingPoint 8 24))

; Global guard, used for checking properties
(declare-fun |execution_statet::\guard_exec| () Bool)

; assign the nondeterministic symbol to x
(assert (= |nondet_symex::nondet0| |main::x|))

; assign x to y
(assert (= |main::x| |main::y|))

; assert x == y
(assert (let ((a!1 (not (=> true
                        (=> |execution_statet::\guard_exec|
                            (fp.eq |main::x| |main::y|))))))
        (or a!1))))
```

Assignment of  
nondeterministic  
value to x

# Floating-point Encoding: Illustrative Example

```
; declaration of x and y
(declare-fun |main::x| () (_ FloatingPoint 8 24))
(declare-fun |main::y| () (_ FloatingPoint 8 24))

; symbol created to represent a nondeterministic number
(declare-fun |nondet_symex::nondet0| () (_ FloatingPoint 8 24))

; Global guard, used for checking properties
(declare-fun |execution_statet::\guard_exec| () Bool)

; assign the nondeterministic symbol to x
(assert (= |nondet_symex::nondet0| |main::x|))

; assign x to y
(assert (= |main::x| |main::y|))

; assert x == y
(assert (let ((a!1 (not (=> true
                        (=> |execution_statet::\guard_exec|
                            (fp.eq |main::x| |main::y|))))))
          (or a!1))))
```

← Assignment x to y

# Floating-point Encoding: Illustrative Example

```
; declaration of x and y
(declare-fun |main::x| () (_ FloatingPoint 8 24))
(declare-fun |main::y| () (_ FloatingPoint 8 24))


; symbol created to represent a nondeterministic number
(declare-fun |nondet_symex::nondet0| () (_ FloatingPoint 8 24))

; Global guard, used for checking properties
(declare-fun |execution_statet::\guard_exec| () Bool)

; assign the nondeterministic symbol to x
(assert (= |nondet_symex::nondet0| |main::x|))

; assign x to y
(assert (= |main::x| |main::y|))
```

Check if the comparison  
*satisfies the guard*



```
; assert x == y
(assert (let ((a!1 (not (=> true
                        (=> |execution_statet::\guard_exec|
                            (fp.eq |main::x| |main::y|))))))
          (or a!1)))
```

# Floating-point Encoding: Illustrative Example

- Z3 produces:

```
sat
(model
  (define-fun |main::x| () (_ FloatingPoint 8 24)
    (_ NaN 8 24))
  (define-fun |main::y| () (_ FloatingPoint 8 24)
    (_ NaN 8 24))
  (define-fun |nondet_symex::nondet0| () (_ FloatingPoint 8 24)
    (_ NaN 8 24))
  (define-fun |execution_statet::\\guard_exec| () Bool
    true)
)
```



# Floating-point Encoding: Illustrative Example

- MathSAT produces:

```
sat
( (|main::x| (_ NaN 8 24))
  (|main::y| (_ NaN 8 24))
  (|nondet_symex::nondet0| (_ NaN 8 24))
  (|execution_statet::\guard_exec| true) )
```

# Floating-point Encoding: Illustrative Example

Counterexample:

```
State 1 file main3.c line 3 function main thread 0  
main
```

```
-----  
main3::main::1::x=-NaN (11111111100000000000000000000001)
```

```
State 2 file main3.c line 4 function main thread 0  
main
```

```
-----  
main3::main::2::y=-NaN (11111111100000000000000000000001)
```

```
State 3 file main3.c line 5 function main thread 0  
main
```

```
-----  
Violated property:  
file main3.c line 5 function main  
assertion  
(_Bool)(x == y)
```

VERIFICATION FAILED

# Intended learning outcomes

- Introduce **software verification** and **validation**
- Understand **soundness** and **completeness** concerning **detection techniques**
- Emphasize the difference among **static analysis**, **testing / simulation**, and **debugging**
- Explain **bounded model checking** of software
- Explain **precise memory model** for **software verification**

# Encoding of Pointers

- arrays and records / tuples typically handled directly by SMT-solver
- pointers modelled as tuples
  - p.o  $\triangleq$  representation of underlying object
  - p.i  $\triangleq$  index (if pointer used as array base)

# Encoding of Pointers

- arrays and records / tuples typically handled directly by SMT-solver
- pointers modelled as tuples
  - p.o  $\triangleq$  representation of underlying object
  - p.i  $\triangleq$  index (if pointer used as array base)

```
int main() {  
    int a[2], i, x, *p;  
    p=a;  
    if (x==0)  
        a[i]=0;  
    else  
        a[i+1]=1;  
    assert(* (p+2)==1);  
}
```

# Encoding of Pointers

- arrays and records / tuples typically handled directly by SMT-solver
- pointers modelled as tuples
  - $p.o \triangleq$  representation of underlying object
  - $p.i \triangleq$  index (if pointer used as array base)

```
int main() {  
  int a[2], i, x, *p;  
  p=a;  
  if (x==0)  
    a[i]=0;  
  else  
    a[i+1]=1;  
  assert(*p+2==1);  
}
```



C :=

$$\left( \begin{array}{l} p_1 := \text{store}(p_0, 0, \&a[0]) \\ \wedge p_2 := \text{store}(p_1, 1, 0) \\ \wedge g_2 := (x_2 == 0) \\ \wedge a_1 := \text{store}(a_0, i_0, 0) \\ \wedge a_2 := a_0 \\ \wedge a_3 := \text{store}(a_2, 1 + i_0, 1) \\ \wedge a_4 := \text{ite}(g_1, a_1, a_3) \\ \wedge p_3 := \text{store}(p_2, 1, \text{select}(p_2, 1) + 2) \end{array} \right)$$

# Encoding of Pointers

- arrays and records / tuples typically handled directly by SMT-solver
- pointers modelled as tuples
  - $p.o \triangleq$  representation of underlying object
  - $p.i \triangleq$  index (if pointer used as array base)

*Store object at position 0*

```
int main() {  
  int a[2], i, x, *p;  
  p=a;  
  if (x==0)  
    a[i]=0;  
  else  
    a[i+1]=1;  
  assert(*p==1);  
}
```



C :=

```
p1 := store(p0, 0, &a[0])  
∧ p2 := store(p1, 1, 0)  
∧ g2 := (x2 == 0)  
∧ a1 := store(a0, i0, 0)  
∧ a2 := a0  
∧ a3 := store(a2, 1+ i0, 1)  
∧ a4 := ite(g1, a1, a3)  
∧ p3 := store(p2, 1, select(p2, 1)+2)
```

# Encoding of Pointers

- arrays and records / tuples typically handled directly by SMT-solver
- pointers modelled as tuples
  - $p.o \triangleq$  representation of underlying object
  - $p.i \triangleq$  index (if pointer used as array base)

*Store object at position 0*

```
int main() {  
  int a[2], i, x, *p;  
  p=a;  
  if (x==0)  
    a[i]=0;  
  else  
    a[i+1]=1;  
  assert(*p==1);  
}
```



C:=

```
{  
  p1 := store(p0, 0, &a[0])  
  ∧ p2 := store(p1, 1, 0)  
  ∧ g2 := (x2 == 0)  
  ∧ a1 := store(a0, i0, 0)  
  ∧ a2 := a0  
  ∧ a3 := store(a2, 1+ i0, 1)  
  ∧ a4 := ite(g1, a1, a3)  
  ∧ p3 := store(p2, 1, select(p2, 1)+2)  
}
```

*Store index at position 1*



# Encoding of Pointers

- arrays and records / tuples typically handled directly by SMT-solver
- pointers modelled as tuples
  - $p.o \triangleq$  representation of underlying object
  - $p.i \triangleq$  index (if pointer used as array base)

```

int main() {
  int a[2], i, x, *p;
  p=a;
  if (x==0)
    a[i]=0;
  else
    a[i+1]=1;
  assert(*p+2==1);
}

```



C

```

p1 := store(p0, 0, &a[0])
∧ p2 := store(p1, 1, 0)
∧ g2 := (x2 == 0)
∧ a1 := store(a0, i0, 0)
∧ a3 := store(a2, 1 + i0, 1)
∧ a4 := ite(g1, a1, a3)
∧ p3 := store(p2, 1, select(p2, 1) + 2)

```

*Store object at position 0*

*Store index at position 1*

*Update index*

# Encoding of Pointers

- arrays and records / tuples typically handled directly by SMT-solver
- pointers modelled as tuples
  - p.o  $\triangleq$  representation of underlying object
  - p.i  $\triangleq$  index (if pointer used as array base)

```
int main() {  
  int a[2], i, x, *p;  
  p=a;  
  if (x==0)  
    a[i]=0;  
  else  
    a[i+1]=1;  
  assert(* (p+2)==1);  
}
```



P:=

$$\left( \begin{array}{l} i_0 \geq 0 \wedge i_0 < 2 \\ \wedge 1 + i_0 \geq 0 \wedge 1 + i_0 < 2 \\ \wedge \text{select}(p_3, 0) == \&a[0] \\ \wedge \text{select}(\text{select}(p_3, 0), \\ \quad \text{select}(p_3, 1)) == 1 \end{array} \right)$$

*negation satisfiable  
(a[2] unconstrained)  
⇒ assert fails*

# Encoding of Memory Allocation

- model memory just as an array of bytes (array theories)
  - read and write operations to the memory array on the logic level

# Encoding of Memory Allocation

- model memory just as an array of bytes (array theories)
  - read and write operations to the memory array on the logic level
- each dynamic object  $d_o$  consists of
  - $m \triangleq$  memory array
  - $s \triangleq$  size in bytes of  $m$
  - $\rho \triangleq$  unique identifier
  - $v \triangleq$  indicate whether the object is still alive
  - $l \triangleq$  the location in the execution where  $m$  is allocated

# Encoding of Memory Allocation

- model memory just as an array of bytes (array theories)
  - read and write operations to the memory array on the logic level
- each dynamic object  $d_o$  consists of
  - $m \triangleq$  memory array
  - $s \triangleq$  size in bytes of  $m$
  - $\rho \triangleq$  unique identifier
  - $v \triangleq$  indicate whether the object is still alive
  - $l \triangleq$  the location in the execution where  $m$  is allocated
- to detect invalid reads/writes, we check whether
  - $d_o$  is a dynamic object
  - $i$  is within the bounds of the memory array

$$l_{is\_dynamic\_object} \Leftrightarrow \left( \bigvee_{j=1}^k d_o.\rho = j \right) \wedge (0 \leq i < n)$$

# Encoding of Memory Allocation

- to check for invalid objects, we
  - set  $v$  to *true* if the function `malloc` can allocate memory ( $d_o$  is alive)
  - set  $v$  to *false* if the function `free` is called ( $d_o$  is not longer alive)

$$I_{valid\_object} \Leftrightarrow (I_{is\_dynamic\_object} \Rightarrow d_o.v)$$

# Encoding of Memory Allocation

- to check for invalid objects, we
  - set  $v$  to *true* if the function `malloc` can allocate memory ( $d_o$  is alive)
  - set  $v$  to *false* if the function `free` is called ( $d_o$  is not longer alive)

$$I_{\text{valid\_object}} \Leftrightarrow (I_{\text{is\_dynamic\_object}} \Rightarrow d_o.v)$$

- to detect forgotten memory, at the end of the (unrolled) program we check
  - whether the  $d_o$  has been deallocated by the function `free`

$$I_{\text{deallocated\_object}} \Leftrightarrow (I_{\text{is\_dynamic\_object}} \Rightarrow \neg d_o.v)$$

# Example of Memory Allocation

```
#include <stdlib.h>
void main() {
    char *p = malloc(5); //  $\rho = 1$ 
    char *q = malloc(5); //  $\rho = 2$ 
    p=q;
    free(p)
    p = malloc(5);      //  $\rho = 3$ 
    free(p)
}
```

Assume that the malloc  
call succeeds



# Example of Memory Allocation

```
#include <std...  
void main() {  
    char *p = ...  
    char *q = malloc(5); // p = ...  
    p=q;  
    free(p)  
    p = malloc(5); // p = 3  
    free(p)  
}
```

memory leak: pointer  
reassignment makes  $d_{o1.v}$   
to become an orphan

# Example of Memory Allocation

```
#include <stdlib.h>
```

```
void main() {
```

```
  char *p = malloc(5); //  $\rho = 1$ 
```

```
  char *q = malloc(5); //  $\rho = 2$ 
```

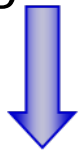
```
  p=q;
```

```
  free(p)
```

```
  p = malloc(5); //  $\rho = 3$ 
```

```
  free(p)
```

```
}
```



$P := (\neg d_{o1}.v \wedge \neg d_{o2}.v \wedge \neg d_{o3}.v)$

$C := \left( \begin{array}{l} d_{o1}.\rho=1 \wedge d_{o1}.s=5 \wedge d_{o1}.v=true \wedge p=d_{o1} \\ \wedge d_{o2}.\rho=2 \wedge d_{o2}.s=5 \wedge d_{o2}.v=true \wedge q=d_{o2} \\ \wedge p=d_{o2} \wedge d_{o2}.v=false \\ \wedge d_{o3}.\rho=3 \wedge d_{o3}.s=5 \wedge d_{o3}.v=true \wedge p=d_{o3} \\ \wedge d_{o3}.v=false \end{array} \right)$

# Example of Memory Allocation

```
#include <stdlib.h>
```

```
void main() {
```

```
  char *p = malloc(5); //  $\rho = 1$ 
```

```
  char *q = malloc(5); //  $\rho = 2$ 
```

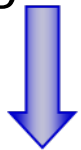
```
  p=q;
```

```
  free(p)
```

```
  p = malloc(5); //  $\rho = 3$ 
```

```
  free(p)
```

```
}
```



$P := (\neg d_{o1}.v \wedge \neg d_{o2}.v \wedge \neg d_{o3}.v)$

$C := \left( \begin{array}{l} d_{o1}.\rho=1 \wedge d_{o1}.s=5 \wedge d_{o1}.v=true \wedge p=d_{o1} \\ \wedge d_{o2}.\rho=2 \wedge d_{o2}.s=5 \wedge d_{o2}.v=true \wedge q=d_{o2} \\ \wedge p=d_{o2} \wedge d_{o2}.v=false \\ \wedge d_{o3}.\rho=3 \wedge d_{o3}.s=5 \wedge d_{o3}.v=true \wedge p=d_{o3} \\ \wedge d_{o3}.v=false \end{array} \right)$

# Align-guaranteed memory mode

- Alignment rules require that any pointer variable must be aligned to at least the alignment of the pointer type
  - E.g., an integer pointer's value must be aligned to at least 4 bytes, for 32-bit integers

# Align-guaranteed memory mode

- Alignment rules require that any pointer variable must be aligned to at least the alignment of the pointer type
  - E.g., an integer pointer's value must be aligned to at least 4 bytes, for 32-bit integers
- Encode **property assertions** when dereferences occur during symbolic execution
  - To guard against executions where an unaligned pointer is dereferenced

# Align-guaranteed memory mode

- Alignment rules require that any pointer variable must be aligned to at least the alignment of the pointer type
  - E.g., an integer pointer's value must be aligned to at least 4 bytes, for 32-bit integers
- Encode **property assertions** when dereferences occur during symbolic execution
  - To guard against executions where an unaligned pointer is dereferenced
  - This is not as strong as the C standard requirement, that a pointer variable may never hold an unaligned value
    - But it provides a guarantee that any pointer dereference will either be correctly aligned or result in a verification failure

# ESBMC's memory model

- statically tracks possible pointer variable targets (objects)
  - dereferencing a pointer leads to the construction of **guarded references** to each **potential target**

# ESBMC's memory model

- statically tracks possible pointer variable targets (objects)
  - dereferencing a pointer leads to the construction of **guarded references** to each **potential target**
- C is very **liberal** about **permitted dereferences**

```
struct foo {  
    uint16_t bar[2];  
    uint8_t baz;  
};
```

```
struct foo qux;  
char *quux = &qux;  
quux++;  
*quux; ← pointer and object types  
do not match
```



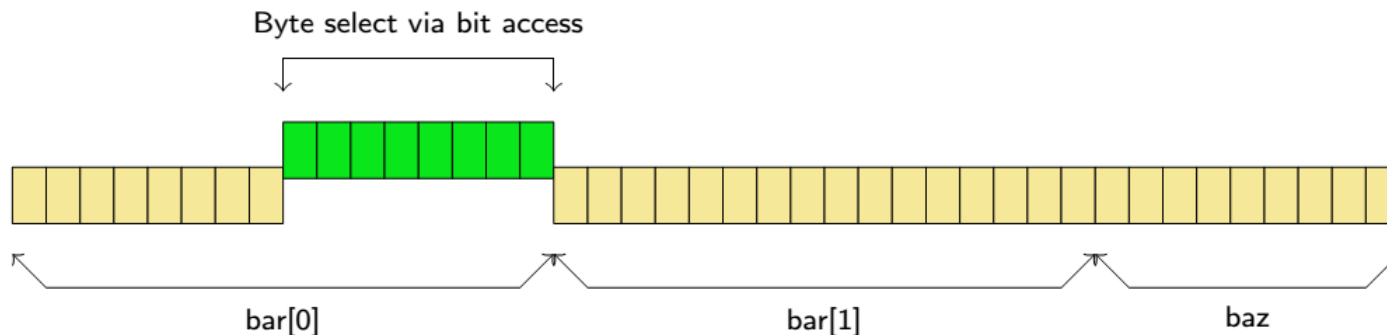
# ESBMC's memory model

- statically tracks possible pointer variable targets (objects)
  - dereferencing a pointer leads to the construction of **guarded references** to each **potential target**
- C is very **liberal** about **permitted dereferences**

```
struct foo {  
    uint16_t bar[2];  
    uint8_t baz;  
};
```

```
struct foo qux;  
char *quux = &qux;  
quux++;  
*quux; ← pointer and object types  
do not match
```

- **SAT: immediate access** to bit-level representation



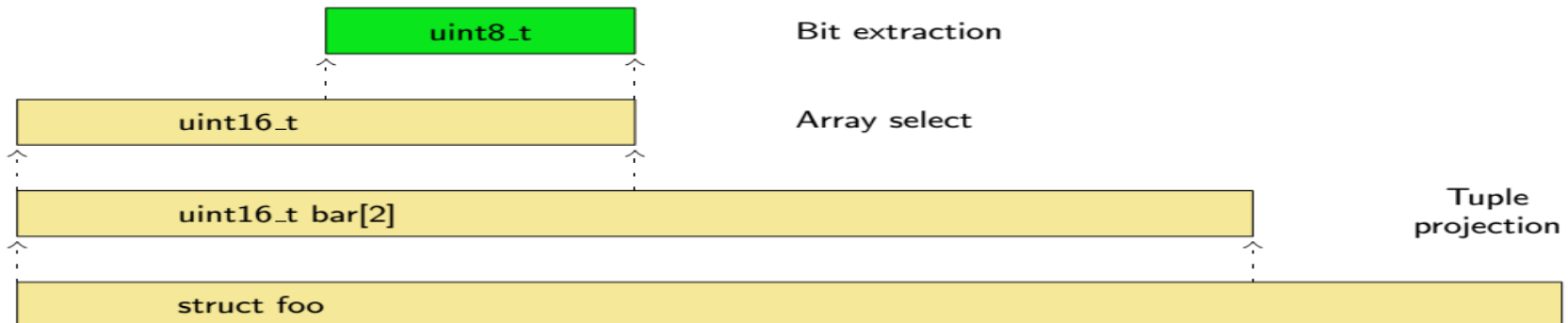
# ESBMC's memory model

- statically tracks possible pointer variable targets (objects)
  - dereferencing a pointer leads to the construction of **guarded references** to each **potential target**
- C is very **liberal** about **permitted dereferences**

```
struct foo {  
    uint16_t bar[2];  
    uint8_t baz;  
};
```

```
struct foo qux;  
char *quux = &qux;  
quux++;  
*quux; ← pointer and object types  
do not match
```

- **SMT**: sorts must be **repeatedly unwrapped**



# Byte-level data extraction in SMT

- access to underlying data bytes is complicated
  - requires manipulation of arrays / tuples

# Byte-level data extraction in SMT

- access to underlying data bytes is complicated
  - requires manipulation of arrays / tuples
- problem is magnified by nondeterministic offsets

```
uint16_t *fuzz;  
if (nondet_bool()) {  
    fuzz = &qux.bar[0];  
} else {  
    fuzz = &qux.baz;  
}
```

- chooses accessed field nondeterministically
- requires a *byte\_extract* expression
- handles the *tuple* that encoded the *struct*

# Byte-level data extraction in SMT

- access to underlying data bytes is complicated
  - requires manipulation of arrays / tuples
- problem is magnified by nondeterministic offsets

```
uint16_t *fuzz;           – chooses accessed field nondeterministically
if (nondet_bool()) {     – requires a byte_extract expression
    fuzz = &qux.bar[0];   – handles the tuple that encoded the struct
} else {
    fuzz = &qux.baz;
}
```

- supporting **all legal behaviors** at SMT layer **difficult**
  - extract (unaligned) 16bit integer from \*fuzz

# Byte-level data extraction in SMT

- access to underlying data bytes is complicated
  - requires manipulation of arrays / tuples
- problem is magnified by nondeterministic offsets

```
uint16_t *fuzz;           – chooses accessed field nondeterministically
if (nondet_bool()) {     – requires a byte_extract expression
    fuzz = &qux.bar[0];  – handles the tuple that encoded the struct
} else {
    fuzz = &qux.baz;
}
```

- supporting **all legal behaviors** at SMT layer **difficult**
  - extract (unaligned) 16bit integer from \*fuzz
- experiments showed significantly increased **memory consumption**

# “Aligned” Memory Model

- framework cannot easily be changed to SMT-level byte representation (a la LLBMC)

# “Aligned” Memory Model

- framework cannot easily be changed to SMT-level byte representation (a la LLBMC)
- push unwrapping of SMT data structures to dereference



# “Aligned” Memory Model

- framework cannot easily be changed to SMT-level byte representation (a la LLBMC)
- push unwrapping of SMT data structures to dereference
- **enforce C alignment rules**
  - static analysis of pointer alignment eliminates need to encode unaligned data accesses
    - reduces number of behaviors that must be modeled

# “Aligned” Memory Model

- framework cannot easily be changed to SMT-level byte representation (a la LLBMC)
- push unwrapping of SMT data structures to dereference
- **enforce C alignment rules**
  - static analysis of pointer alignment eliminates need to encode unaligned data accesses
    - reduces number of behaviors that must be modeled
  - add alignment assertions (if static analysis not conclusive)

# “Aligned” Memory Model

- framework cannot easily be changed to SMT-level byte representation (a la LLBMC)
- push unwrapping of SMT data structures to dereference
- **enforce C alignment rules**
  - static analysis of pointer alignment eliminates need to encode unaligned data accesses
    - reduces number of behaviors that must be modeled
  - add alignment assertions (if static analysis not conclusive)
  - extracting 16-bit integer from \*fuzz if guard is true:
    - offset = 0: project bar[0] out of foo
    - offset = 1: **“unaligned memory access” failure**
    - offset = 2: project bar[1] out of foo
    - offset = 3: **“unaligned memory access” failure**
    - offset = 4: “access to object out of bounds” failure

# Summary

- Described the difference between **soundness** and **completeness** concerning **detection techniques**
  - **False positive** and **false negative**
- Pointed out the difference between **static analysis** and **testing / simulation**
  - **hybrid combination** of static and dynamic analysis techniques to achieve a good trade-off between **soundness** and **completeness**
- Explained **bounded model checking of software**
  - they have been applied successfully to verify **single-threaded software using a precise memory model**