

10001011  
01001100  
101111  
01100100  
10101101



**Systems and Software  
Verification Laboratory**

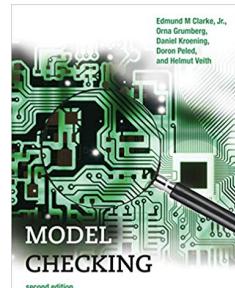


# Detection of Software Vulnerabilities: Static Analysis

**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);  
}
```

```
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);  
}
```

Deadlock

# **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 **unbounded 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 **unbounded model checking** of software

# 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**

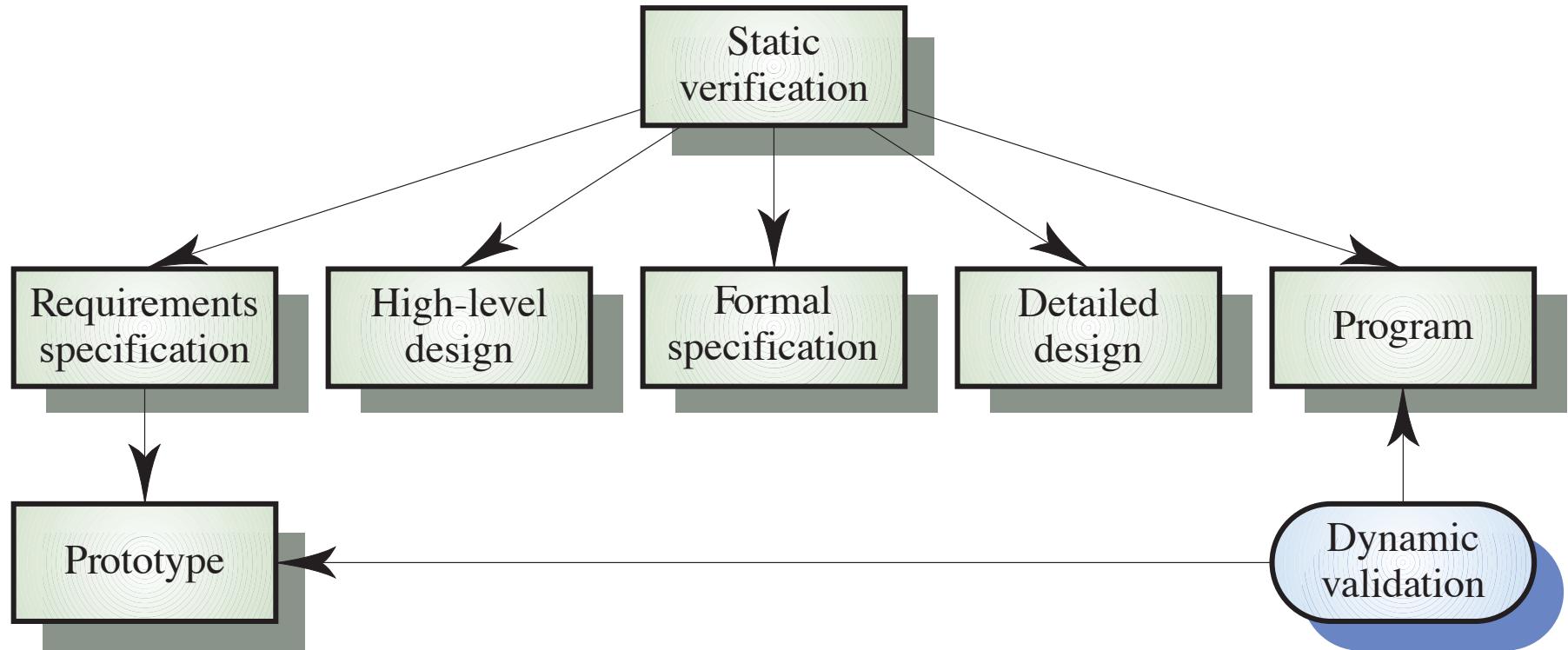
# Static and Dynamic Verification

- **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 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 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



# 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

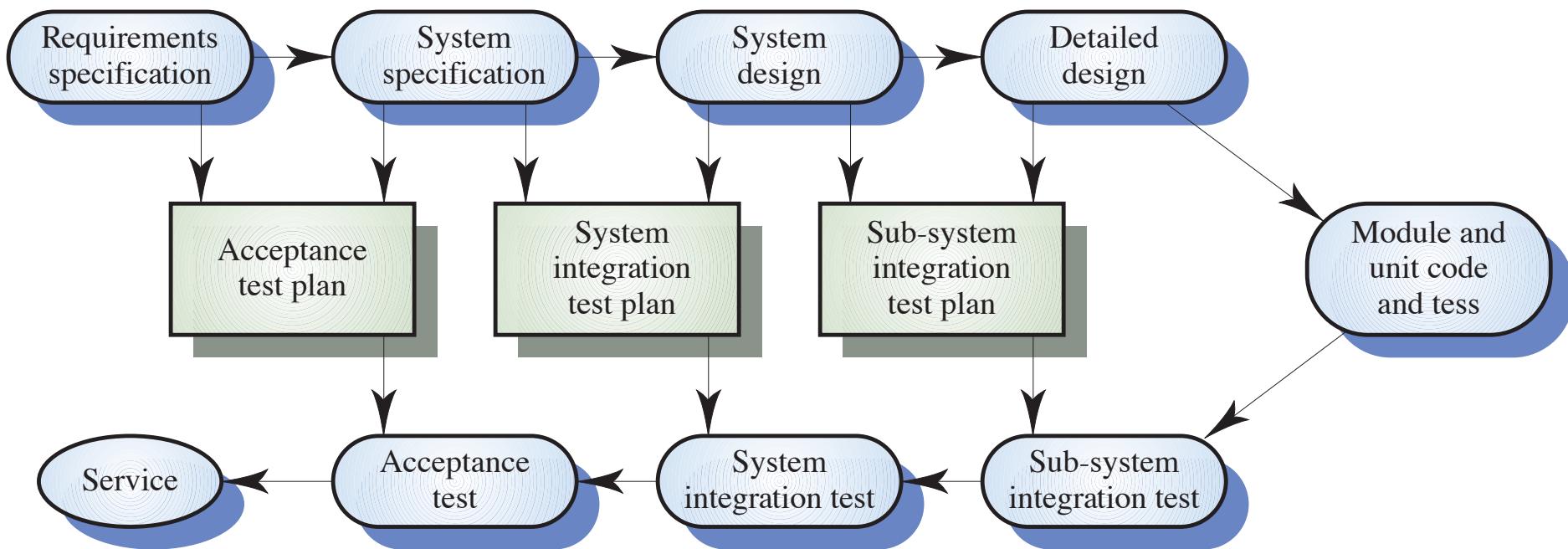
- Careful planning is required to get the most out of dynamic and static verification
  - Planning should start early in the development process
  - The plan should identify the balance between static and dynamic verification
- V & V should establish confidence that the software is fit for purpose

# 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 concludes 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 concludes 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

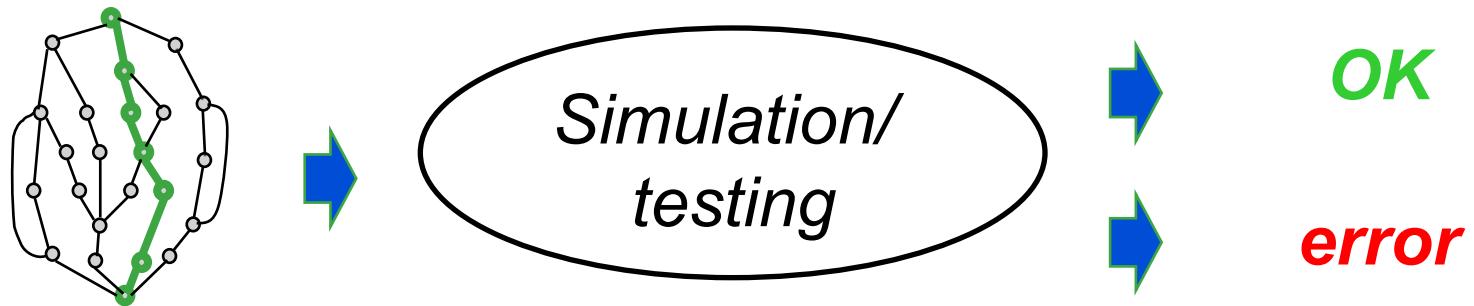
Detection tools can use a **hybrid combination of static and dynamic analysis** techniques to achieve a good trade-off between **soundness and completeness**

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

# Intended learning outcomes

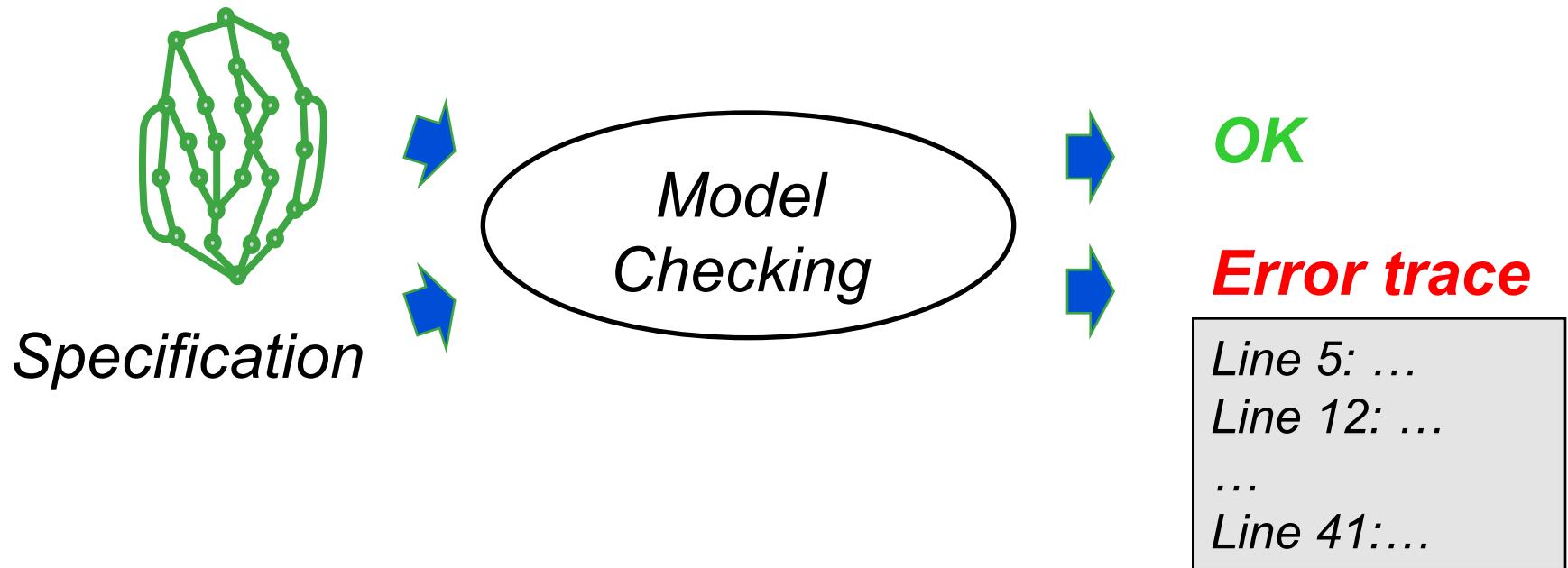
- Introduce **software verification** and **validation**
- Understand **soundness** and **completeness** concerning **detection techniques**
- Emphasize the difference among **static analysis**, **testing / simulation**, and **debugging**
- Explain **bounded model checking** of software
- Explain **unbounded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

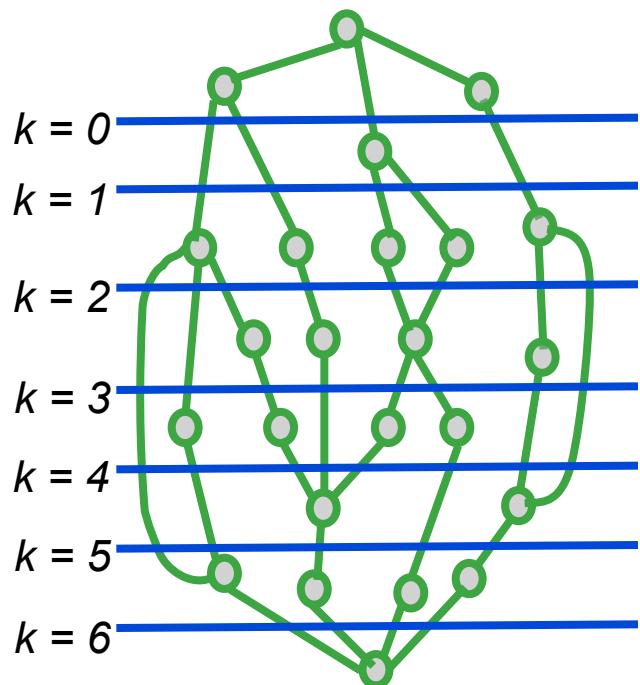


- **Exhaustively explores all executions**
- Report errors as **traces**
- May produce **incorrect results**

# Avoiding state space explosion

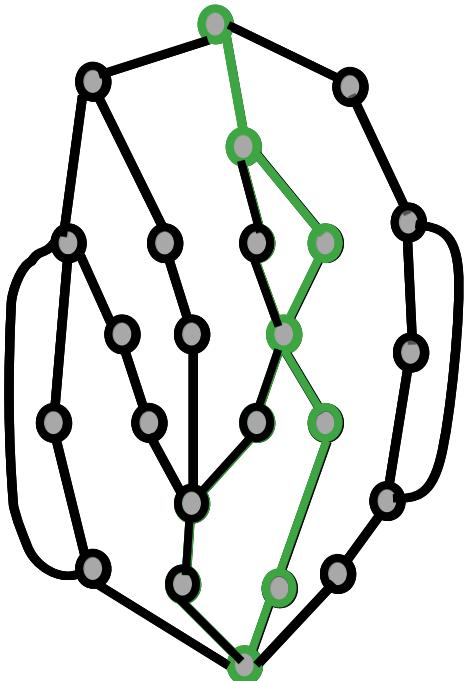
- Bounded Model Checking (BMC)
  - Breadth-first search (BFS) approach
- Symbolic Execution
  - Depth-first search (DFS) approach

# Bounded Model Checking



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

# Symbolic Execution



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

# 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.

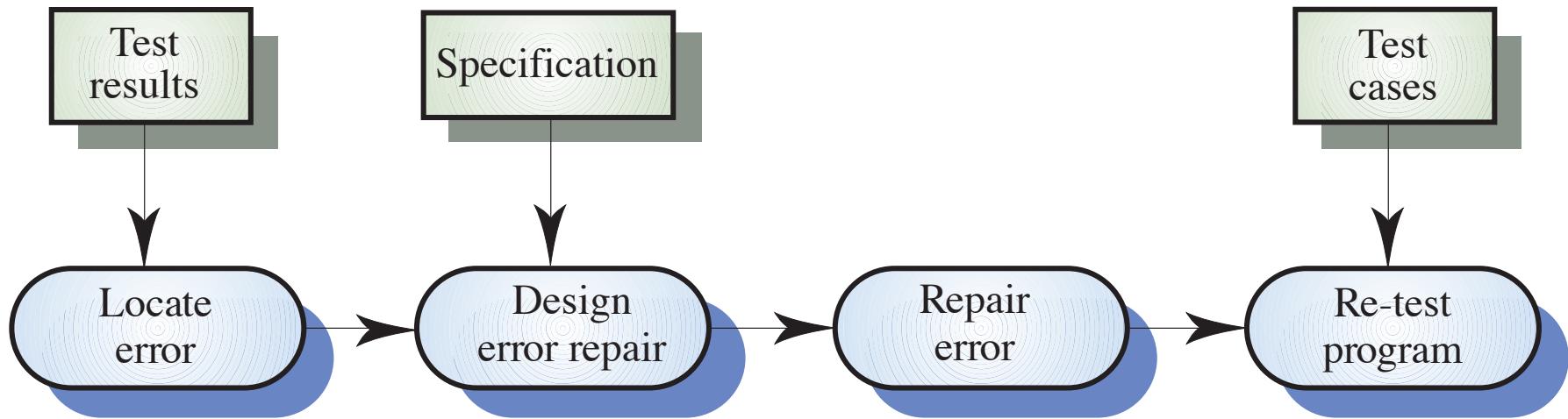
# V&V and debugging

- 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

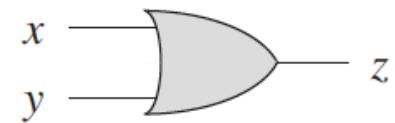
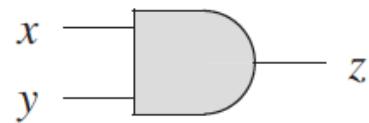
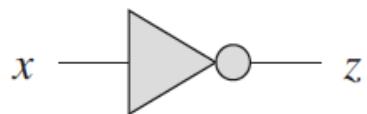
- Introduce **software verification** and **validation**
- Understand **soundness** and **completeness** concerning **detection techniques**
- Emphasize the difference among **static analysis**, **testing / simulation**, and **debugging**
- Explain **bounded model checking of software**
- Explain **unbounded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

# 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)



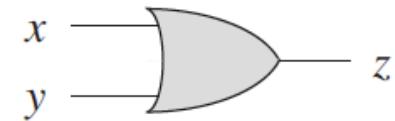
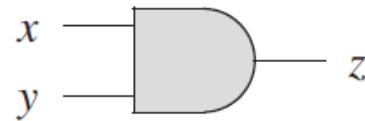
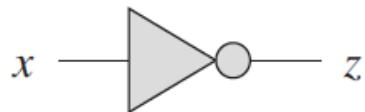
$x$	$\neg x$
0	1
1	0

$x$	$y$	$x \wedge y$
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

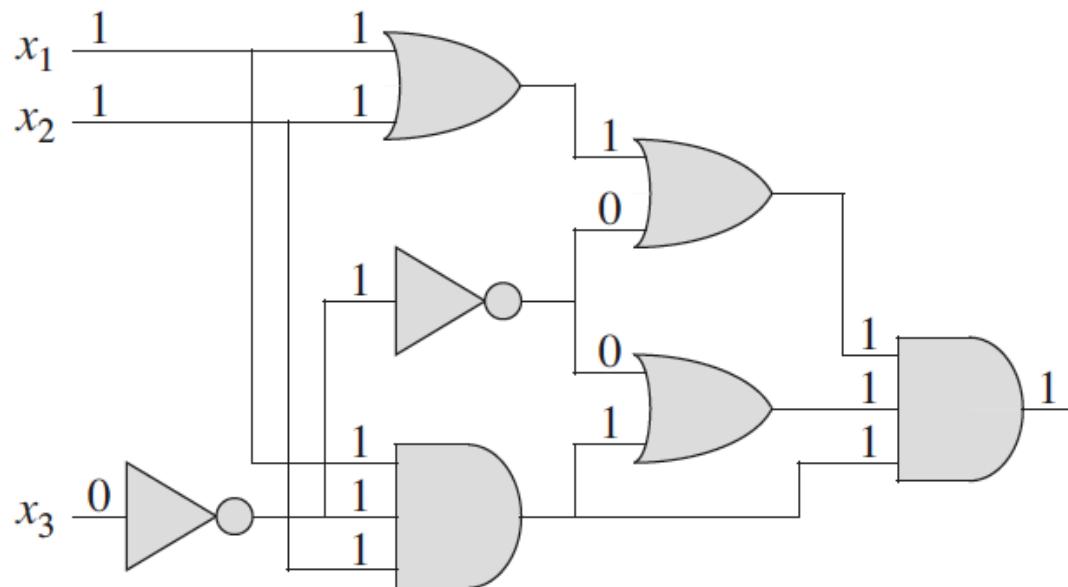
$x$	$y$	$x \wedge y$
0	0	0
0	1	0
1	0	0
1	1	1

$x$	$y$	$x \vee y$
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 = { $\langle C \rangle$  : 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 = { $\langle C \rangle : C$  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 = { $\langle C \rangle : C$  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

$SAT = \{<\Phi> : \Phi \text{ is a satisfiable Boolean formula}\}$

# Formula Satisfiability (SAT)

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

$SAT = \{<\Phi> : \Phi \text{ is a satisfiable Boolean formula}\}$

- Example:

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

# Formula Satisfiability (SAT)

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

$SAT = \{<\Phi> : \Phi \text{ is a satisfiable Boolean formula}\}$

- Example:
  - $\Phi = ((x_1 \rightarrow x_2) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg x_2$
  - Assignment:  $<x_1 = 0, x_2 = 0, x_3 = 1, x_4 = 1>$

# Formula Satisfiability (SAT)

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

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

- 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

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

- 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

$SAT = \{<\Phi> : \Phi \text{ is a satisfiable Boolean formula}\}$

- Example:

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

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

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

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

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

# Formula Satisfiability (SAT)

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

$SAT = \{<\Phi> : \Phi \text{ is a satisfiable Boolean formula}\}$

- Example:

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

- Assignment:  $<x_1 = 0, x_2 = 0, x_3 = 1, x_4 = 1>$

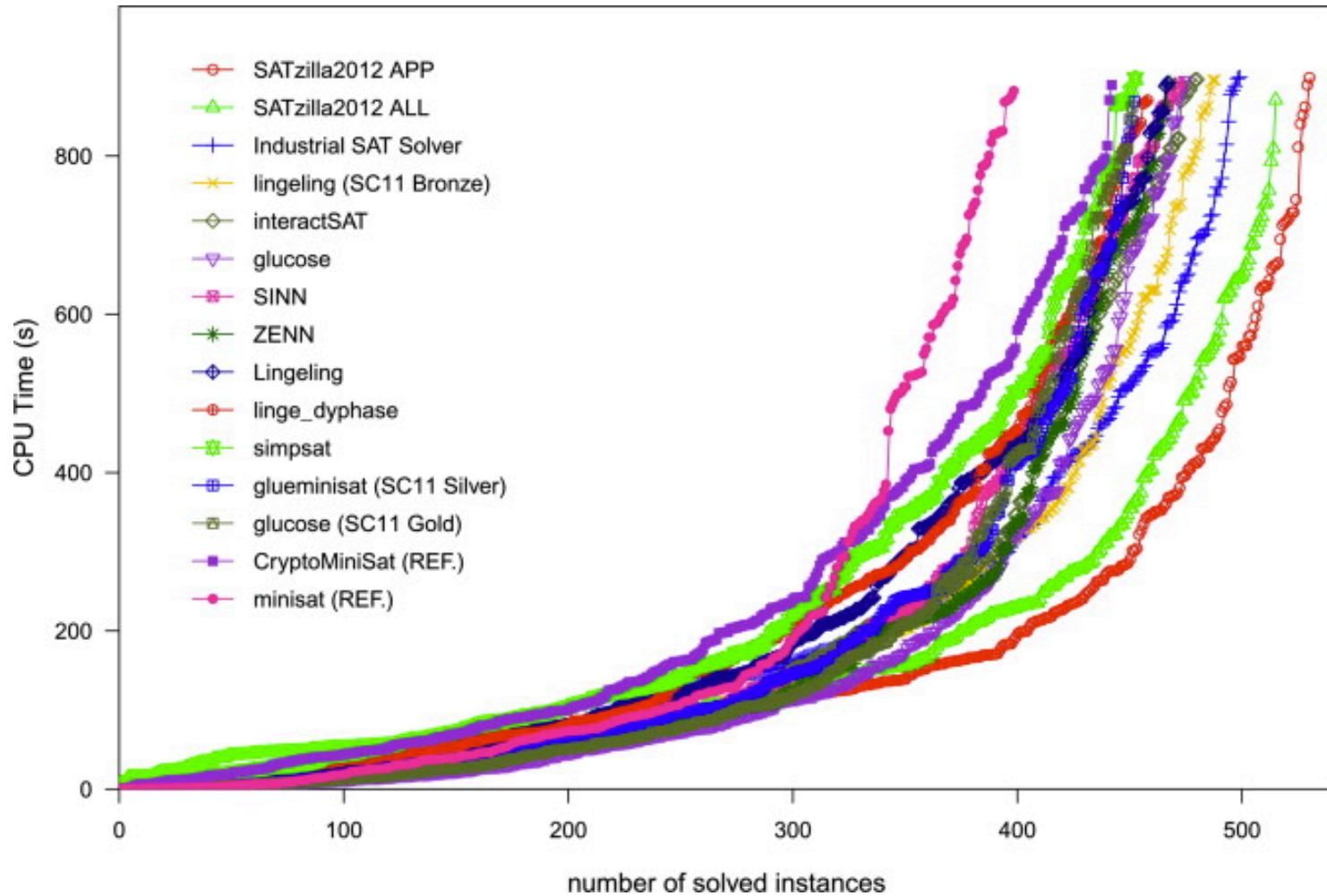
- $\Phi = ((0 \rightarrow 0) \vee \neg((\neg 0 \leftrightarrow 1) \vee 1)) \wedge \neg 0$

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

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

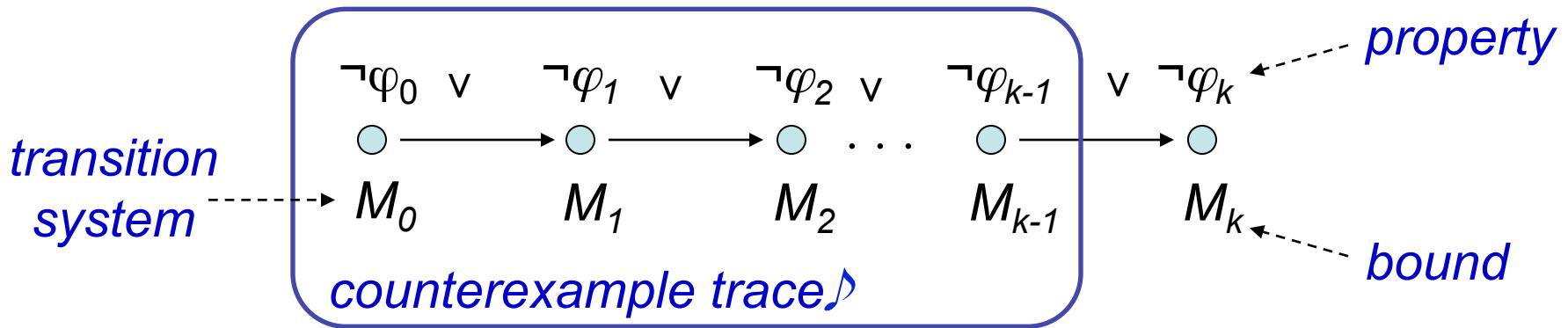
- $\Phi = 1$

# SAT Competition



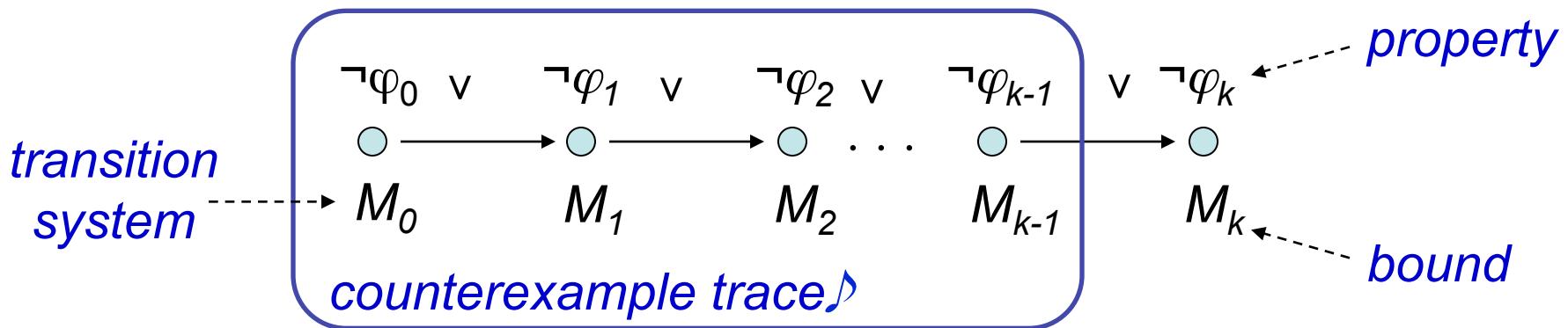
# Bounded Model Checking

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



# Bounded Model Checking

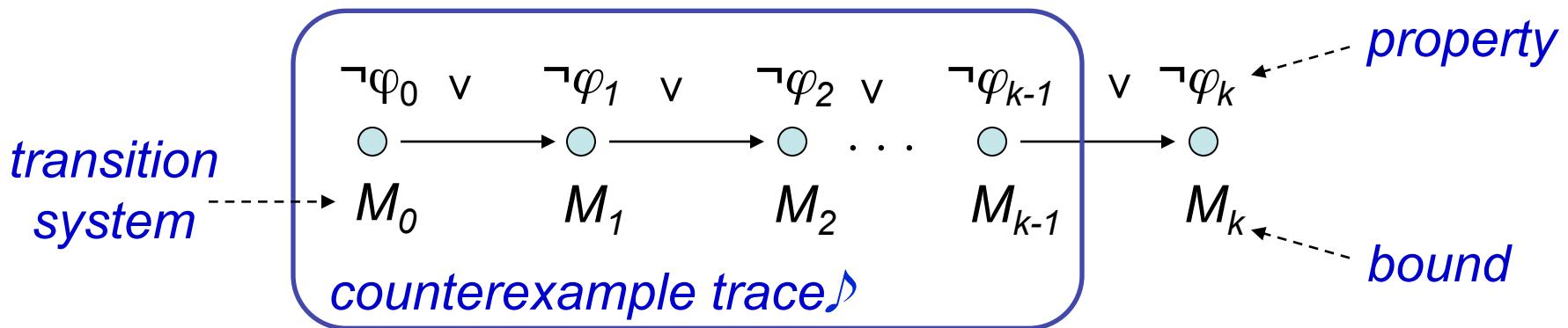
Basic Idea: check negation of given property up to given depth



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

# Bounded Model Checking

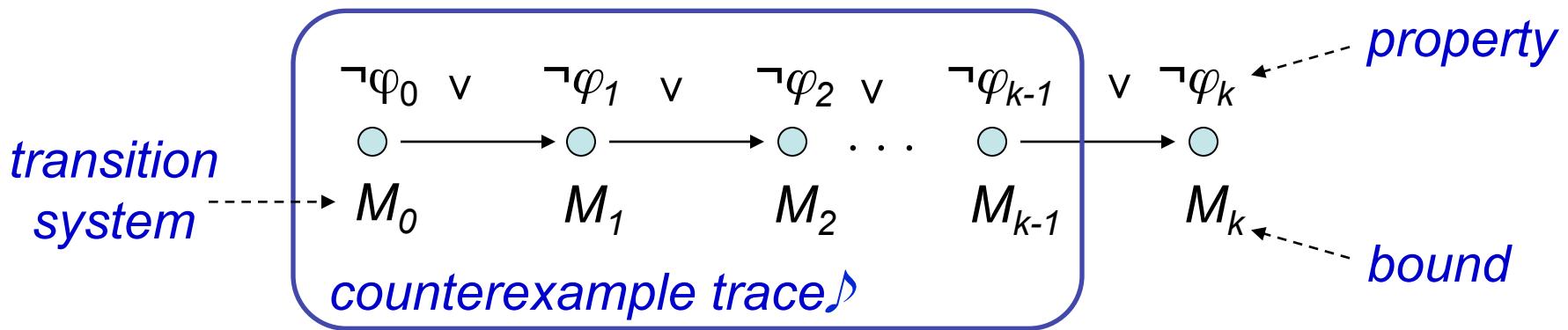
Basic Idea: check negation of given property up to given depth



- 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	$(b >> i) \& 1 = 1$

# 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	$(b >> i) \& 1 = 1$
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	$(b >> i) \& 1 = 1$
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	$(b >> i) \& 1 = 1$
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  $\Gamma$**  ( $\Gamma \models_T \varphi$ ) iff every model of  $T \cup \Gamma$  is also a model of  $\varphi$
- Checking  $\Gamma \models_T \varphi$  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.

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

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

step1:  $g(select(store(a,c,12), b' + 3)) \neq g(b' - c + 4) \wedge b' = c - 3 \wedge c + 1 = d - 4$

# 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.

$$g(select(store(a,c,12)), SignExt(b,16) + 3)$$

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

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

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

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

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

# 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.

$$g(select(store(a,c,12)), SignExt(b,16) + 3)$$

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

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

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

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

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

↓ using facts about bit-vector arithmetic

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

# 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

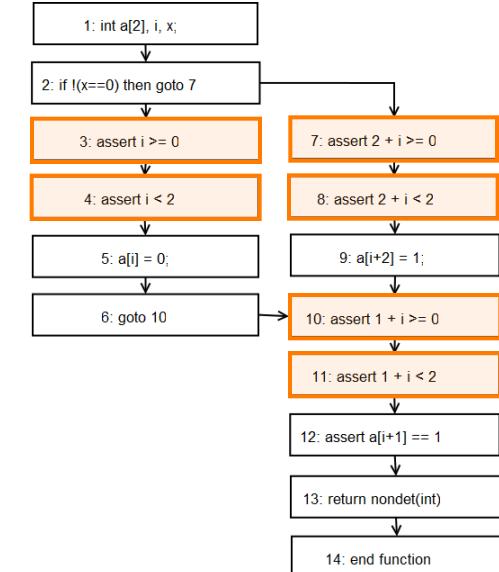
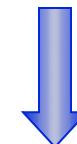
$$r \wedge \text{false} \mapsto \text{false}$$

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

# 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

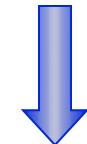
```
int main() {  
    int a[2], i, x;  
    if (x==0)  
        a[i]=0;  
    else  
        a[i+2]=1;  
    assert(a[i+1]==1);  
}
```


$$\begin{aligned}g_1 &= x_1 == 0 \\a_1 &= a_0 \text{ WITH } [i_0:=0] \\a_2 &= a_0 \\a_3 &= a_2 \text{ WITH } [2+i_0:=1] \\a_4 &= g_1 ? a_1 : a_3 \\t_1 &= a_4[1+i_0] == 1\end{aligned}$$

# 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
- 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, ...)
    - different conversions for every pair of types
    - 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, ...)
    - different conversions for every pair of types
    - 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

# 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, ...)
    - different conversions for every pair of types
    - 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
    - $\text{unsigned\_overflow} \Leftrightarrow (r - (r \bmod 2^w)) < 2^w$

# 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, ...)
    - different conversions for every pair of types
    - 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
    - $\text{res\_op} \Leftrightarrow \neg \text{overflow}(x, y) \wedge \neg \text{underflow}(x, y)$
    - 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.

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

# 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.

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

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

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

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

# 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  $\mathbb{Z}$  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
    - o *fp.rem*: remainder:  $x - y * n$ , where  $n$  in  $\mathbb{Z}$  is nearest to  $x/y$
    - o *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 #b11111111111 #x00000000000000)
                (fp #b0 #b00000000000 #x00000000000000))
|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 #b011111111111 #x00000000000000)  
           (fp #b0 #b000000000000 #x00000000000000))  
        |main::b|))
```

*If true, assign 1f to 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 #x00000000000000)
              (fp #b0 #b0000000000 #x00000000000000))
              |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 #b00000000000 #x00000000000000)))
           |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 #x000000000000)))  
           |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 #x00000000000000)))
           |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 #b00000000000 #x00000000000000)))
           |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|))

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

Variable declarations

# 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|))


Assignment x to 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|)) Check if the comparison  
satisfies the guard
; 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

- 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 2 file main3.c line 4 function main thread 0  
main

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

# 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+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)

```
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);  
}
```

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

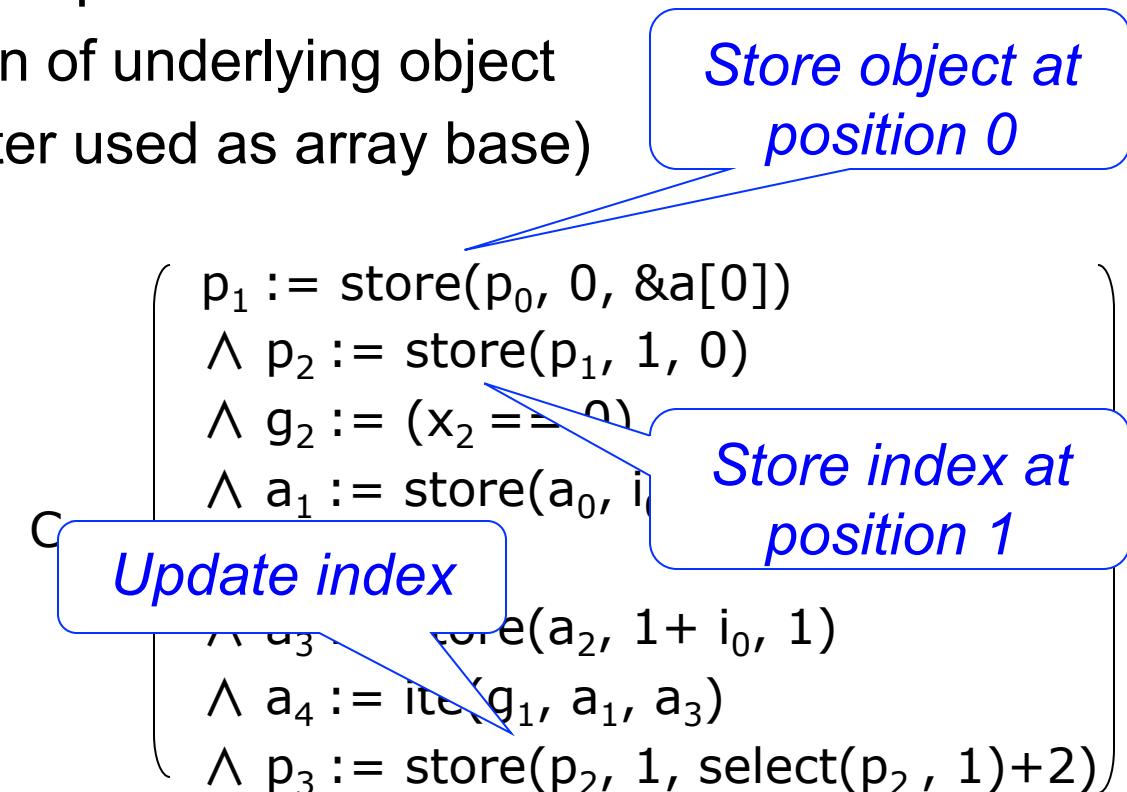
*Store object at position 0*

*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);  
}
```



# 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);  
}
```



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

*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* when the function `malloc` is called ( $d_o$  is alive)
  - set  $v$  to *false* when the function `free` is called ( $d_o$  is not longer alive)

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

# Encoding of Memory Allocation

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

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

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

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

# 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<sub>o1</sub>.v  
to become an orphan*

# Example of Memory Allocation

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

```
void main() {
```

```
    char *p = malloc(5); // ρ = 1
```

```
    char *q = malloc(5); // ρ = 2
```

```
    p=q;
```

```
    free(p)
```

```
    p = malloc(5); // ρ = 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); // ρ = 1
```

```
    char *q = malloc(5); // ρ = 2
```

```
p=q;
```

```
free(p)
```

```
p = malloc(5); // ρ = 3
```

```
free(p)
```

```
}
```



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

↓

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

# 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

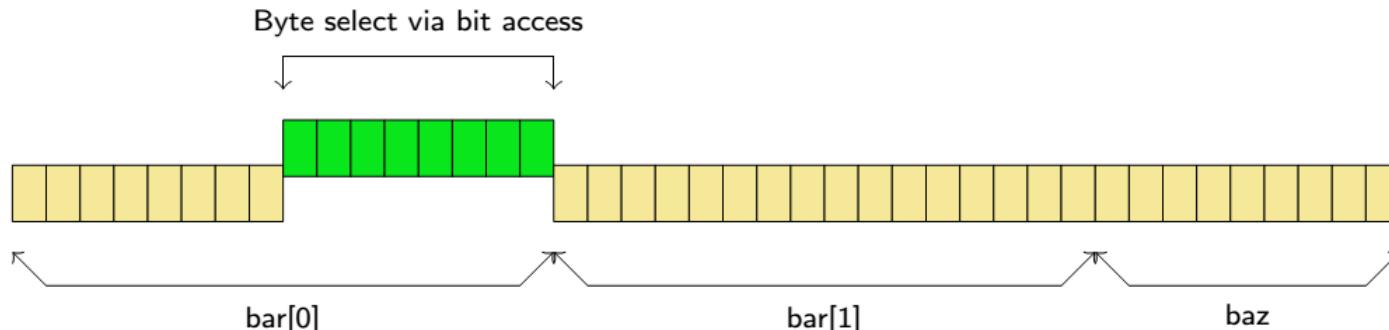
- statically tracks possible pointer variable targets (objects)
  - dereferencing a pointer leads to the construction of **guarded references** to each **potential target**
- C is very **liberal** about **permitted dereferences**

```
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

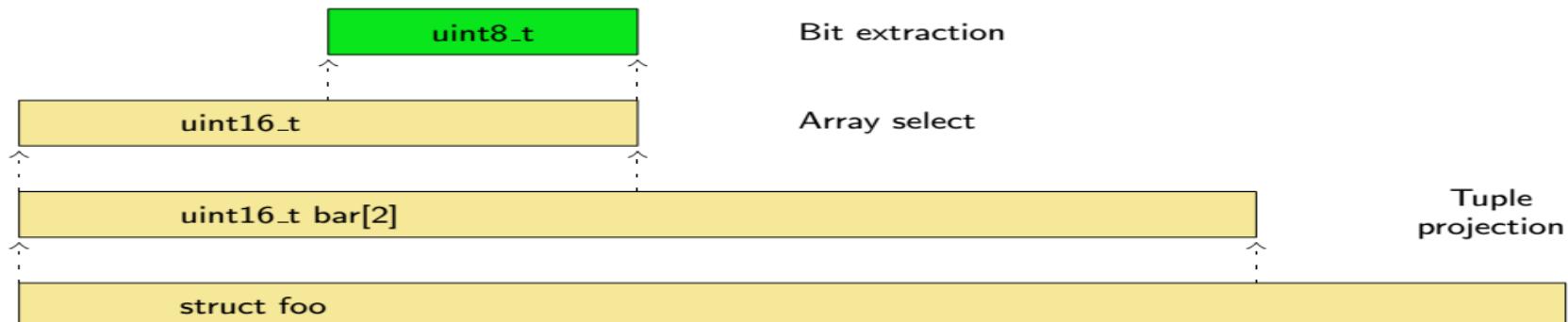
- statically tracks possible pointer variable targets (objects)
  - dereferencing a pointer leads to the construction of **guarded references** to each **potential target**
- C is very **liberal** about **permitted dereferences**

```
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;  
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*
- 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;  
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*
- 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

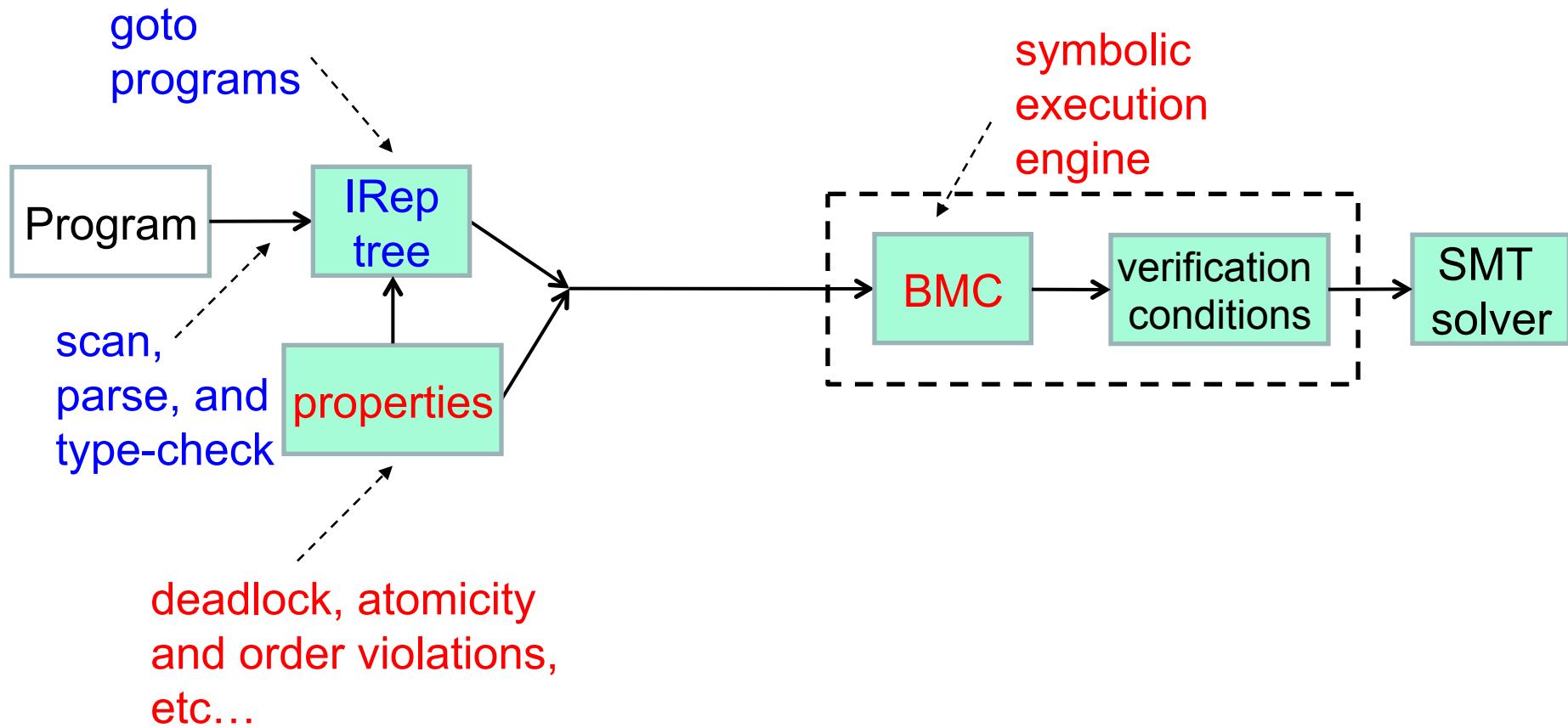
- framework cannot easily be changed to SMT-level byte representation (a la LLBMC)
- push unwrapping of SMT data structures to dereference
- **enforce C alignment rules**
  - static analysis of pointer alignment eliminates need to encode unaligned data accesses
    - reduces number of behaviors that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

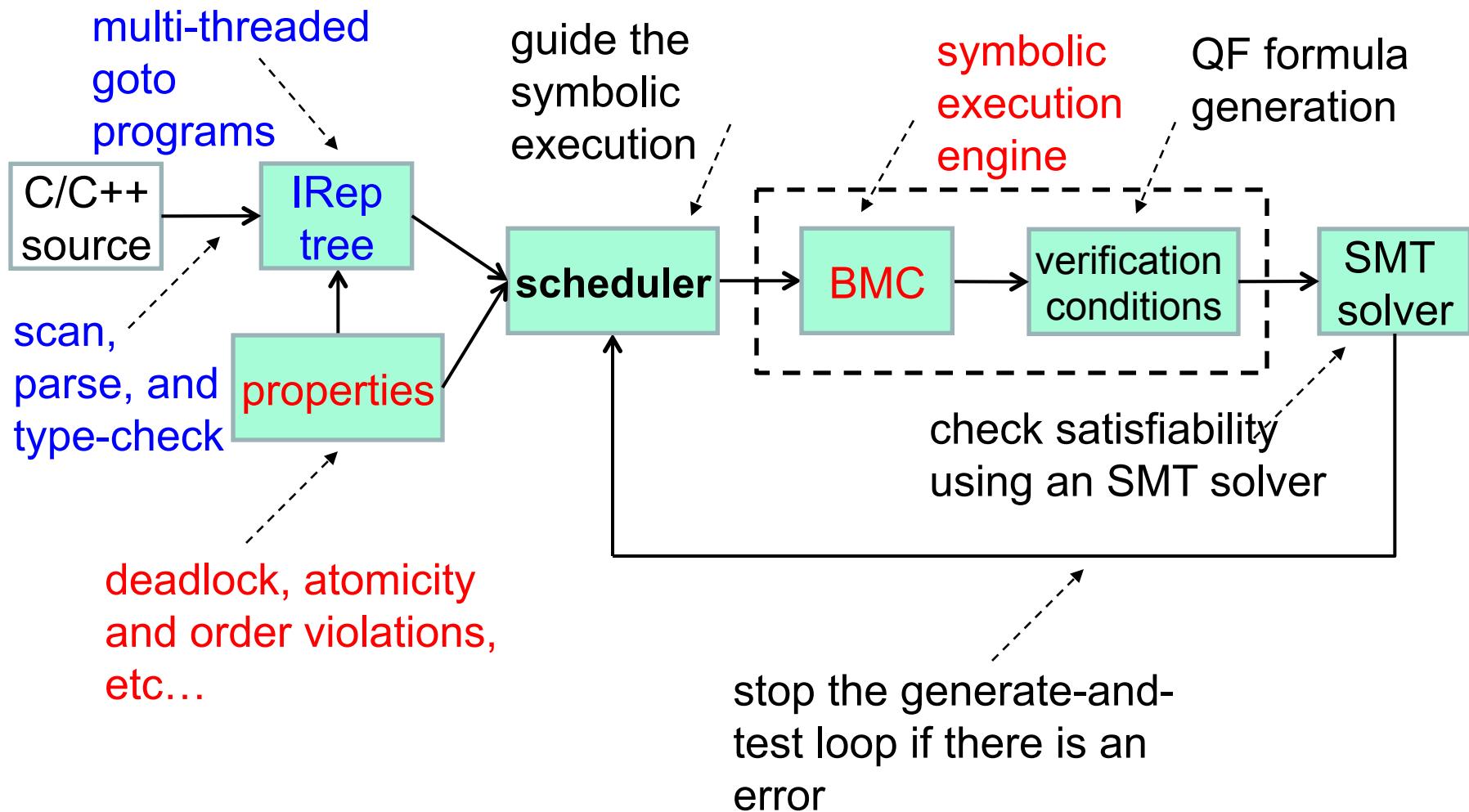
# BMC Architecture

- A typical BMC architecture for verifying programs



# BMC of Multi-threaded Software

**Idea: iteratively generate all possible interleavings and call the BMC procedure on each interleaving**



# Running Example

- the program has sequences of operations that need to be protected together to avoid atomicity violation
  - requirement: the region of code (val1 and val2) should execute atomically

Thread twoStage

```
1: lock(m1);
2: val1 = 1;
3: unlock(m1);
4: lock(m2);
5: val2 = val1 + 1;
6: unlock(m2);
```

program counter: 0

mutexes:  $m1=0; m2=0;$

global variables:  $val1=0; val2=0;$

local variables:  $t1= -1; t2= -1;$

A state  $s \in S$  consists of the value of the program counter pc and the values of all program variables

```
7: unlock(m1);
13: lock(m2);
14: t2 = val2;
15: unlock(m2);
16: assert(t2==(t1+1));
```

# Lazy exploration: interleaving $I_s$

statements:

val1-access:

val2-access:

*Thread twoStage*

```
1: lock(m1);
2: val1 = 1;
3: unlock(m1);
4: lock(m2);
5: val2 = val1 + 1;
6: unlock(m2);
```

*Thread reader*

```
7: lock(m1);
8: if (val1 == 0) {
9:   unlock(m1);
10: return NULL; }
11: t1 = val1;
12: unlock(m1);
13: lock(m2);
14: t2 = val2;
15: unlock(m2);
16: assert(t2==(t1+1));
```

*program counter: 0*

*mutexes: m1=0; m2=0;*

*global variables: val1=0; val2=0;*

*local variables: t1= -1; t2= -1;*

# Lazy exploration: interleaving $I_s$

statements: 1

val1-access:

val2-access:

*Thread twoStage*

- 1: *lock(m1);*
- 2: *val1 = 1;*
- 3: *unlock(m1);*
- 4: *lock(m2);*
- 5: *val2 = val1 + 1;*
- 6: *unlock(m2);*

*Thread reader*

- 7: *lock(m1);*
- 8: *if (val1 == 0) {*
- 9: *unlock(m1);*
- 10: *return NULL; }*
- 11: *t1 = val1;*
- 12: *unlock(m1);*
- 13: *lock(m2);*
- 14: *t2 = val2;*
- 15: *unlock(m2);*
- 16: *assert(t2==(t1+1));*

**program counter: 1**

mutexes: **m1=1**; m2=0;

global variables: val1=0; val2=0;

local variables: t1= -1; t2= -1;

# Lazy exploration: interleaving I<sub>s</sub>

statements: 1-2

val1-access: W<sub>twoStage, 2</sub>

val2-access:

write access to the shared variable val1 in statement 2 of the thread twoStage

Thread twoStage  
1: lock(m1);  
● 2: val1 = 1;  
3: unlock(m1);  
4: lock(m2);  
5: val2 = val1 + 1;  
6: unlock(m2);

Thread reader  
7: lock(m1);  
8: if (val1 == 0) {  
9: unlock(m1);  
10: return NULL; }  
11: t1 = val1;  
12: unlock(m1);  
13: lock(m2);  
14: t2 = val2;  
15: unlock(m2);  
16: assert(t2==(t1+1));

**program counter: 2**

mutexes: m1=1; m2=0;

global variables: val1=1; val2=0;

local variables: t1= -1; t2= -1;

# Lazy exploration: interleaving $I_s$

statements: 1-2-3

val1-access:  $W_{\text{twoStage}, 2}$

val2-access:

*Thread twoStage*

```
1: lock(m1);
2: val1 = 1;
● 3: unlock(m1);
4: lock(m2);
5: val2 = val1 + 1;
6: unlock(m2);
```

*Thread reader*

```
7: lock(m1);
8: if (val1 == 0) {
9:   unlock(m1);
10: return NULL; }
11: t1 = val1;
12: unlock(m1);
13: lock(m2);
14: t2 = val2;
15: unlock(m2);
16: assert(t2==(t1+1));
```

**program counter: 3**

mutexes: **m1=0**; m2=0;

global variables: val1=1; val2=0;

local variables: t1= -1; t2= -1;

# Lazy exploration: interleaving $I_s$

statements: 1-2-3-7

val1-access:  $W_{\text{twoStage}, 2}$

val2-access:

*Thread twoStage*

```
1: lock(m1);
2: val1 = 1;
3: unlock(m1);
4: lock(m2);
5: val2 = val1 + 1;
6: unlock(m2);
```

CS1

*Thread reader*

```
7: lock(m1);
8: if (val1 == 0) {
9:   unlock(m1);
10: return NULL; }
11: t1 = val1;
12: unlock(m1);
13: lock(m2);
14: t2 = val2;
15: unlock(m2);
16: assert(t2==(t1+1));
```

**program counter:** 7

mutexes: **m1=1**; m2=0;

global variables: val1=1; val2=0;

local variables: t1= -1; t2= -1;

# Lazy exploration: interleaving 1

statements: 1-2-3-7-8

val1-access:  $W_{\text{twoStage},2} - R_{\text{reader},8}$

val2-access:

read access to the shared variable **val1** in statement 8 of the thread **reader**

Thread twoStage  
1: lock(m1);  
2: val1 = 1;  
3: unlock(m1);  
4: lock(m2);  
5: val2 = val1 + 1;  
6: unlock(m2);

CS1

Thread reader  
7: lock(m1);  
8: if (val1 == 0) {  
9: unlock(m1);  
10: return NULL; }  
11: t1 = val1;  
12: unlock(m1);  
13: lock(m2);  
14: t2 = val2;  
15: unlock(m2);  
16: assert(t2==(t1+1));

**program counter: 8**

mutexes: m1=1; m2=0;

global variables: val1=1; val2=0;

local variables: t1= -1; t2= -1;

# Lazy exploration: interleaving $I_s$

statements: 1-2-3-7-8-11

val1-access:  $W_{\text{twoStage}, 2} - R_{\text{reader}, 8} - R_{\text{reader}, 11}$

val2-access:

*Thread twoStage*

```
1: lock(m1);
2: val1 = 1;
3: unlock(m1);
4: lock(m2);
5: val2 = val1 + 1;
6: unlock(m2);
```

*CS1*

*Thread reader*

```
7: lock(m1);
8: if (val1 == 0) {
9:   unlock(m1);
10: return NULL; }
```

11: ***t1 = val1;***

```
12: unlock(m1);
13: lock(m2);
14: t2 = val2;
15: unlock(m2);
16: assert(t2==(t1+1));
```

***program counter: 11***

*mutexes: m1=1; m2=0;*

*global variables: val1=1; val2=0;*

*local variables: t1= 1; t2= -1;*

# Lazy exploration: interleaving $I_s$

statements: 1-2-3-7-8-11-12

val1-access:  $W_{\text{twoStage},2} - R_{\text{reader},8} - R_{\text{reader},11}$

val2-access:

*Thread twoStage*

```
1: lock(m1);
2: val1 = 1;
3: unlock(m1);
4: lock(m2);
5: val2 = val1 + 1;
6: unlock(m2);
```

*CS1*

*Thread reader*

```
7: lock(m1);
8: if (val1 == 0) {
9:   unlock(m1);
10: return NULL; }
11: t1 = val1;
12: unlock(m1); 12: unlock(m1);
13: lock(m2);
14: t2 = val2;
15: unlock(m2);
16: assert(t2==(t1+1));
```

**program counter: 12**

mutexes: **m1=0**; m2=0;

global variables: val1=1; val2=0;

local variables: t1= 1; t2= -1;

# Lazy exploration: interleaving $I_s$

statements: 1-2-3-7-8-11-12

val1-access:  $W_{\text{twoStage},2} - R_{\text{reader},8} - R_{\text{reader},11}$

val2-access:

*Thread twoStage*

```
1: lock(m1);
2: val1 = 1;
3: unlock(m1);
4: lock(m2);           CS1
5: val2 = val1 + 1;    CS2
6: unlock(m2);
```

*Thread reader*

```
7: lock(m1);
8: if (val1 == 0) {
9:   unlock(m1);
10: return NULL; }
11: t1 = val1;
12: unlock(m1);
13: lock(m2);
14: t2 = val2;
15: unlock(m2);
16: assert(t2==(t1+1));
```

**program counter: 4**

mutexes:  $m1=0; m2=0;$

global variables:  $val1=1; val2=0;$

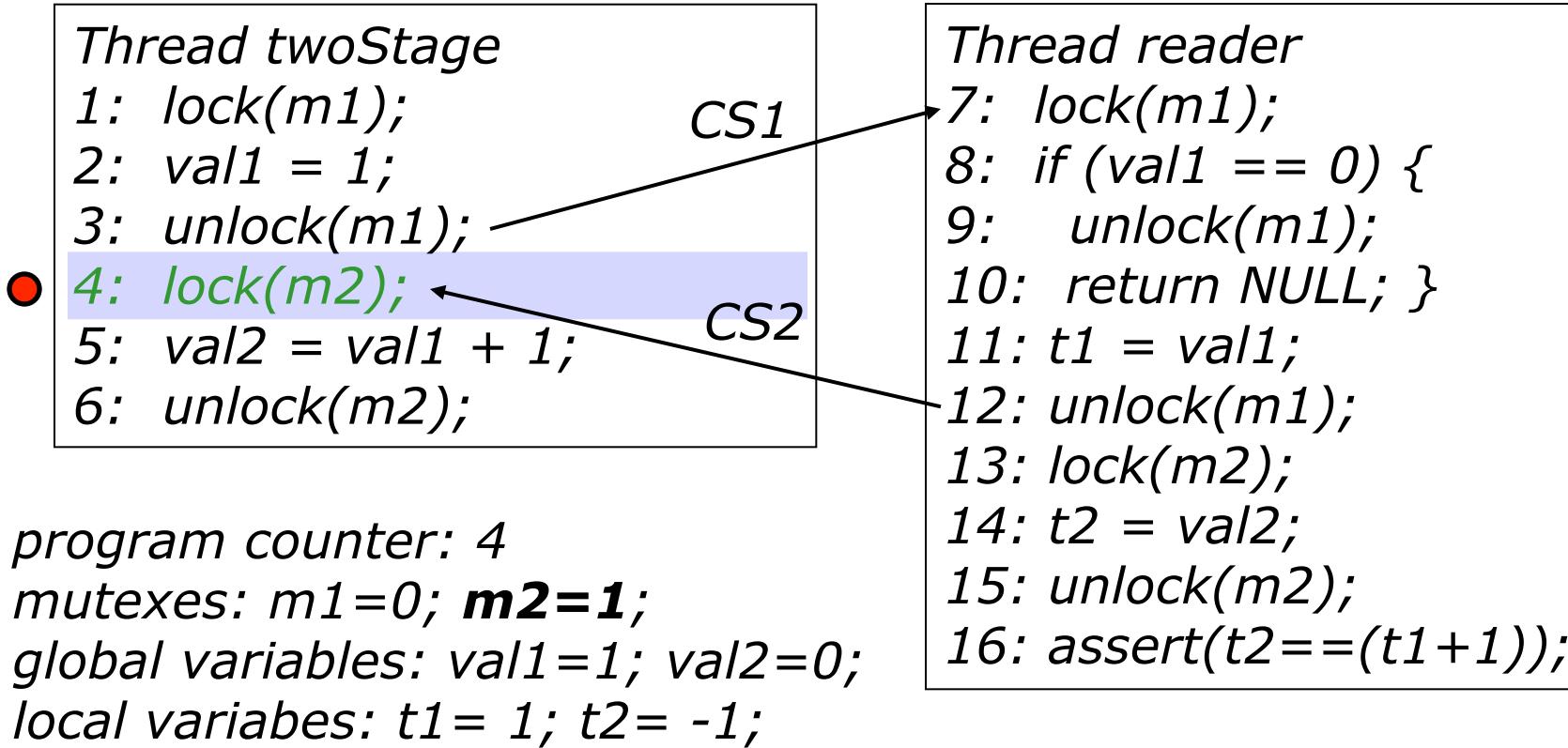
local variables:  $t1= 1; t2= -1;$

# Lazy exploration: interleaving $I_s$

statements: 1-2-3-7-8-11-12-4

val1-access:  $W_{\text{twoStage}, 2} - R_{\text{reader}, 8} - R_{\text{reader}, 11}$

val2-access:



# Lazy exploration: interleaving $I_s$

statements: 1-2-3-7-8-11-12-4-5

val1-access:  $W_{\text{twoStage},2} - R_{\text{reader},8} - R_{\text{reader},11} - R_{\text{twoStage},5}$

val2-access:  $W_{\text{twoStage},5}$

Thread twoStage

```
1: lock(m1);
2: val1 = 1;
3: unlock(m1);
4: lock(m2);
5: val2 = val1 + 1; ●
6: unlock(m2);
```

CS1

CS2

Thread reader

```
7: lock(m1);
8: if (val1 == 0) {
9:   unlock(m1);
10: return NULL; }
11: t1 = val1;
12: unlock(m1);
13: lock(m2);
14: t2 = val2;
15: unlock(m2);
16: assert(t2==(t1+1));
```

**program counter:** 5

mutexes:  $m1=0; m2=1;$

global variables:  $val1=1; \mathbf{val2=2};$

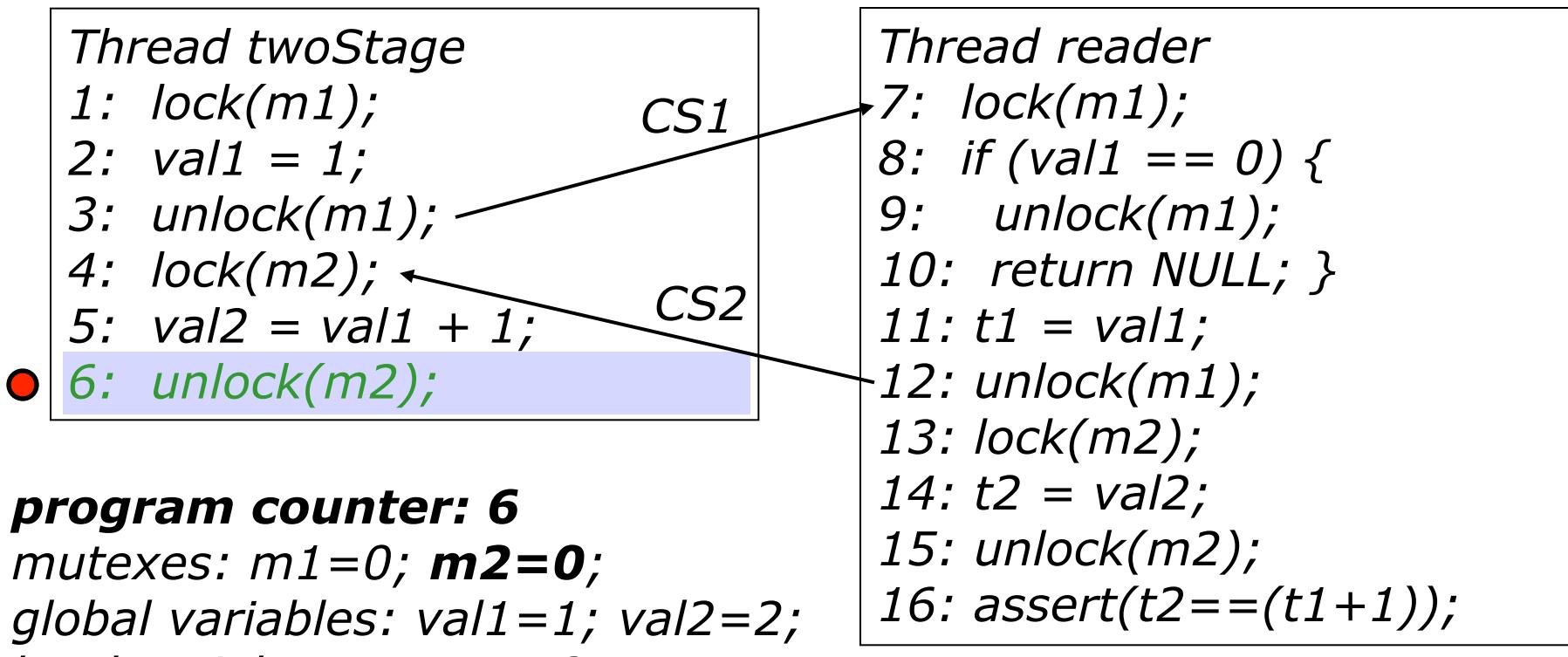
local variables:  $t1= 1; t2= -1;$

# Lazy exploration: interleaving $I_s$

statements: 1-2-3-7-8-11-12-4-5-6

val1-access:  $W_{\text{twoStage},2} - R_{\text{reader},8} - R_{\text{reader},11} - R_{\text{twoStage},5}$

val2-access:  $W_{\text{twoStage},5}$



# Lazy exploration: interleaving $I_s$

statements: 1-2-3-7-8-11-12-4-5-6

val1-access:  $W_{\text{twoStage},2} - R_{\text{reader},8} - R_{\text{reader},11} - R_{\text{twoStage},5}$

val2-access:  $W_{\text{twoStage},5}$

*Thread twoStage*

```
1: lock(m1);
2: val1 = 1;
3: unlock(m1);
4: lock(m2);
5: val2 = val1 + 1;
6: unlock(m2);
```

$CS1$

$CS2$

$CS3$

*Thread reader*

```
7: lock(m1);
8: if (val1 == 0) {
9:   unlock(m1);
10: return NULL; }
11: t1 = val1;
12: unlock(m1);
13: lock(m2);
14: t2 = val2;
15: unlock(m2);
16: assert(t2==(t1+1));
```

**program counter: 13**

mutexes:  $m1=0; m2=0;$

global variables:  $val1=1; val2=2;$

local variables:  $t1= 1; t2= -1;$

# Lazy exploration: interleaving $I_s$

statements: 1-2-3-7-8-11-12-4-5-6-13

val1-access:  $W_{\text{twoStage},2} - R_{\text{reader},8} - R_{\text{reader},11} - R_{\text{twoStage},5}$

val2-access:  $W_{\text{twoStage},5}$

*Thread twoStage*

```
1: lock(m1);
2: val1 = 1;
3: unlock(m1);
4: lock(m2);
5: val2 = val1 + 1;
6: unlock(m2);
```

$CS1$

$CS2$

$CS3$

*Thread reader*

```
7: lock(m1);
8: if (val1 == 0) {
9:   unlock(m1);
10: return NULL; }
11: t1 = val1;
12: unlock(m1);
```

13: lock(m2);

```
14: t2 = val2;
15: unlock(m2);
16: assert(t2==(t1+1));
```

program counter: 13

mutexes:  $m1=0$ ;  **$m2=1$** ;

global variables:  $val1=1$ ;  $val2=2$ ;

local variables:  $t1= 1$ ;  $t2= -1$ ;

# Lazy exploration: interleaving $I_s$

statements: 1-2-3-7-8-11-12-4-5-6-13-14

val1-access:  $W_{\text{twoStage},2} - R_{\text{reader},8} - R_{\text{reader},11} - R_{\text{twoStage},5}$

val2-access:  $W_{\text{twoStage},5} - R_{\text{reader},14}$

*Thread twoStage*

```
1: lock(m1);
2: val1 = 1;
3: unlock(m1);
4: lock(m2);
5: val2 = val1 + 1;
6: unlock(m2);
```

CS1

CS2

CS3

*Thread reader*

```
7: lock(m1);
8: if (val1 == 0) {
9:   unlock(m1);
10: return NULL; }
11: t1 = val1;
12: unlock(m1);
13: lock(m2);
```

14: **t2 = val2;**

```
15: unlock(m2);
16: assert(t2==(t1+1));
```

**program counter: 14**

mutexes:  $m1=0; m2=1;$

global variables:  $val1=1; val2=2;$

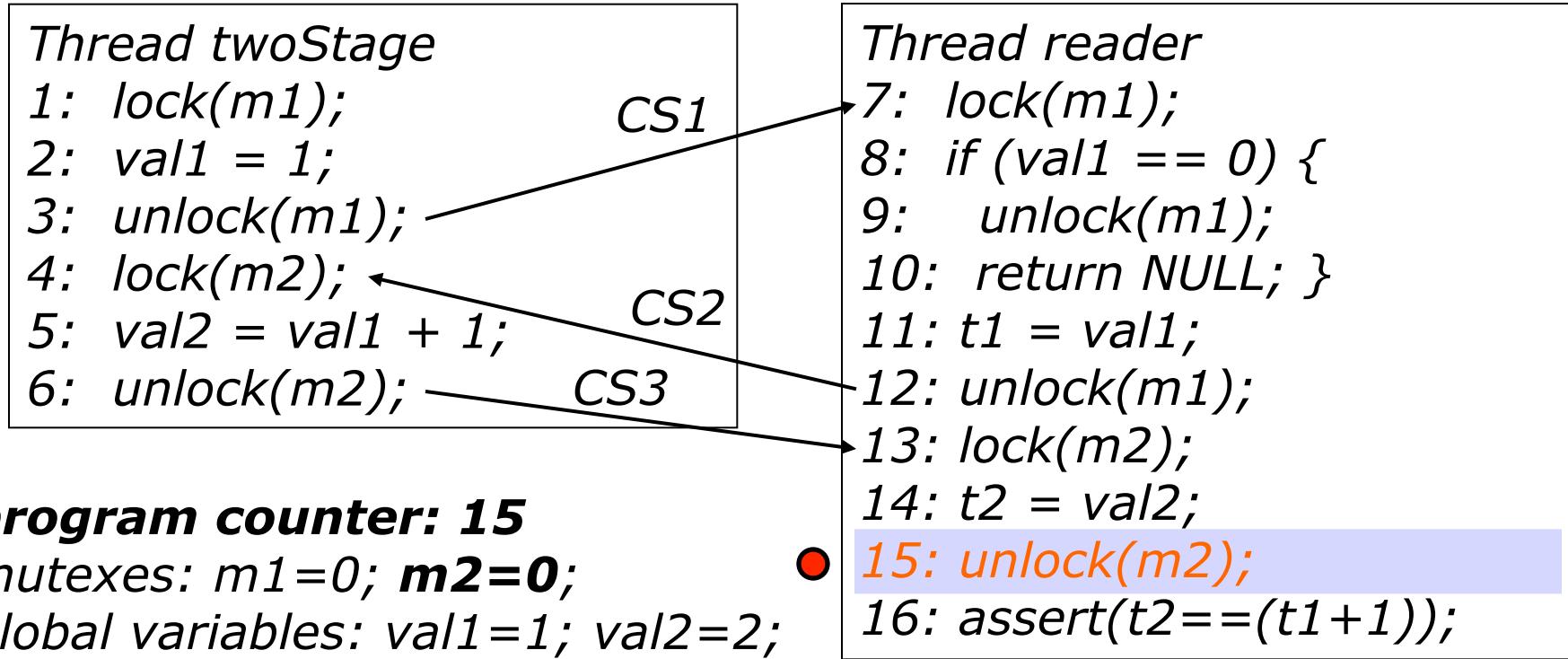
local variables:  $t1= 1; t2= 2;$

# Lazy exploration: interleaving $I_s$

statements: 1-2-3-7-8-11-12-4-5-6-13-14-15

val1-access:  $W_{\text{twoStage},2} - R_{\text{reader},8} - R_{\text{reader},11} - R_{\text{twoStage},5}$

val2-access:  $W_{\text{twoStage},5} - R_{\text{reader},14}$

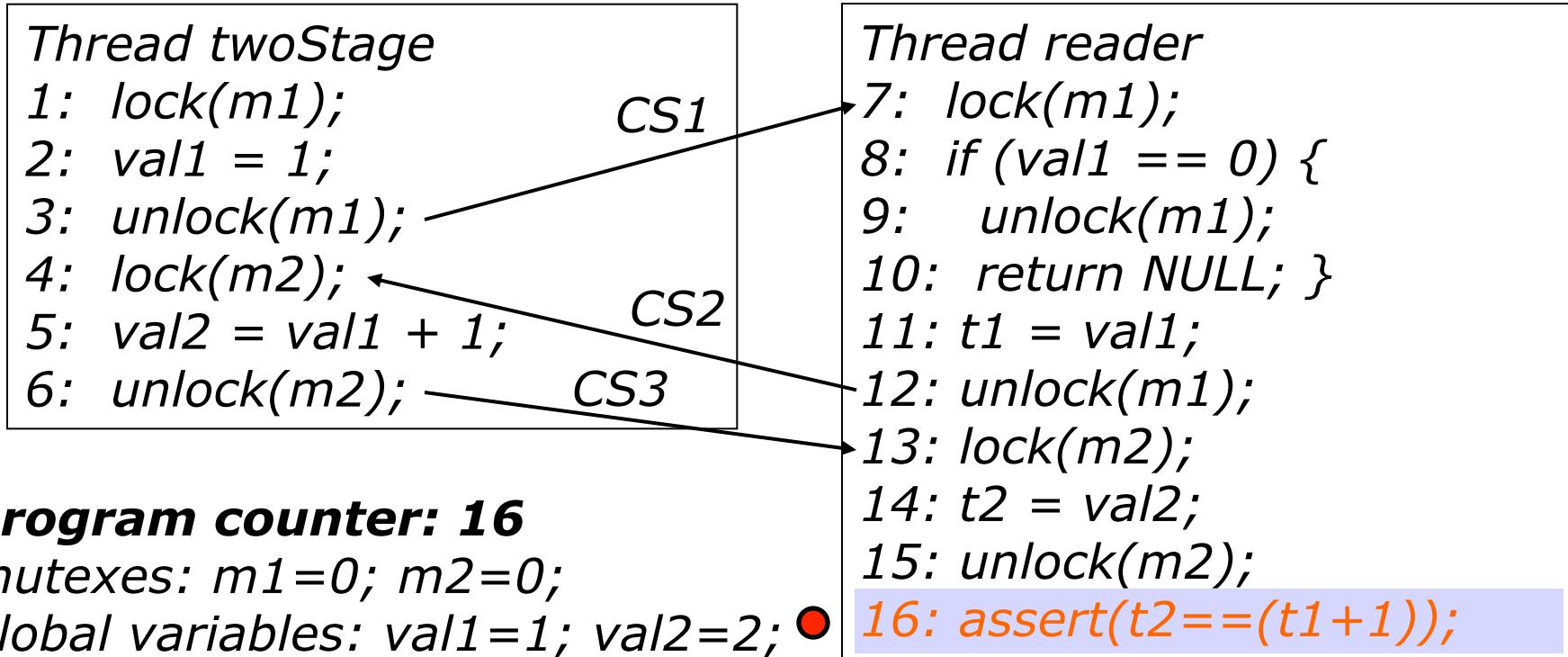


# Lazy exploration: interleaving $I_s$

statements: 1-2-3-7-8-11-12-4-5-6-13-14-15-16

val1-access:  $W_{\text{twoStage},2} - R_{\text{reader},8} - R_{\text{reader},11} - R_{\text{twoStage},5}$

val2-access:  $W_{\text{twoStage},5} - R_{\text{reader},14}$

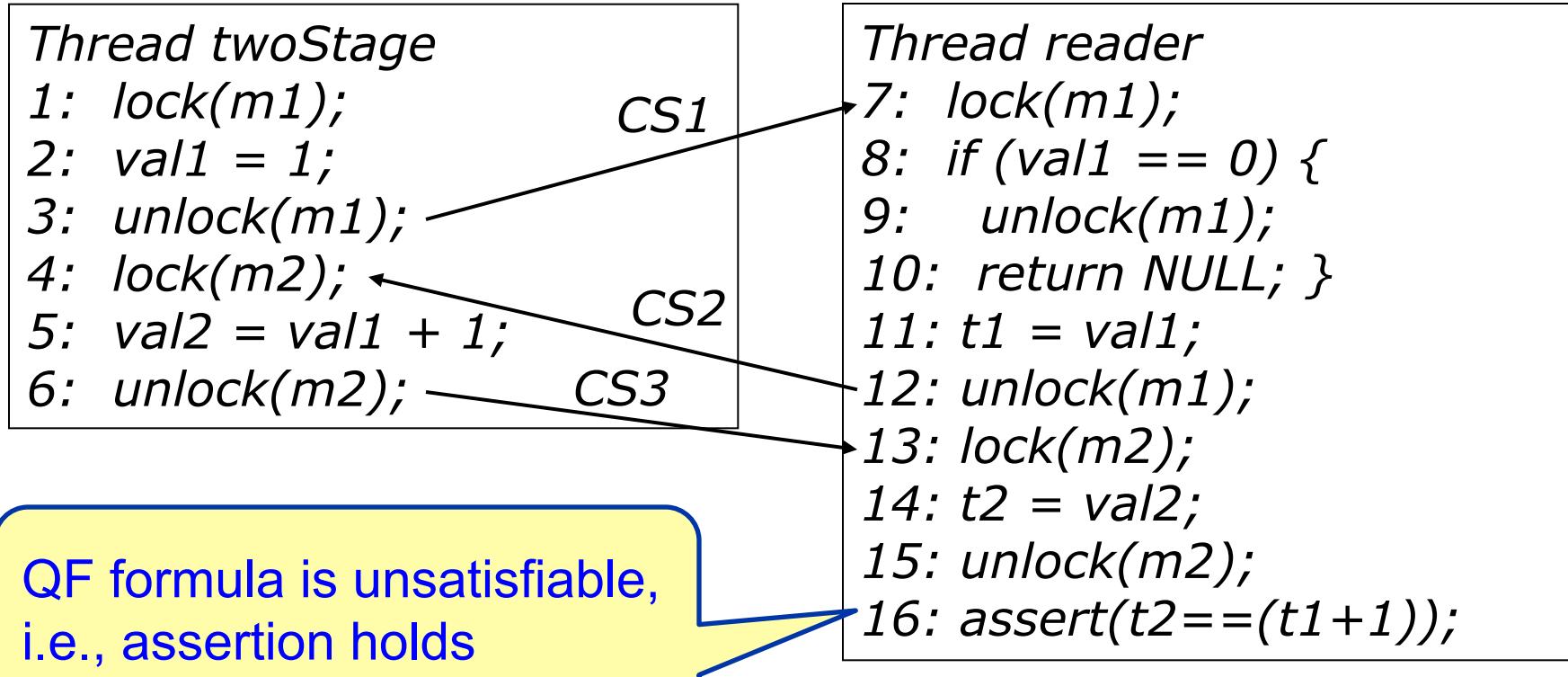


# Lazy exploration: interleaving $I_s$

statements: 1-2-3-7-8-11-12-4-5-6-13-14-15-16

val1-access:  $W_{\text{twoStage},2} - R_{\text{reader},8} - R_{\text{reader},11} - R_{\text{twoStage},5}$

val2-access:  $W_{\text{twoStage},5} - R_{\text{reader},14}$



# Lazy exploration: interleaving $I_f$

statements:

val1-access:

val2-access:

*Thread twoStage*

```
1: lock(m1);
2: val1 = 1;
3: unlock(m1);
4: lock(m2);
5: val2 = val1 + 1;
6: unlock(m2);
```

*program counter: 0*

*mutexes: m1=0; m2=0;*

*global variables: val1=0; val2=0;*

*local variables: t1= -1; t2= -1;*

*Thread reader*

```
7: lock(m1);
8: if (val1 == 0) {
9:   unlock(m1);
10: return NULL; }
11: t1 = val1;
12: unlock(m1);
13: lock(m2);
14: t2 = val2;
15: unlock(m2);
16: assert(t2==(t1+1));
```

# Lazy exploration: interleaving $I_f$

statements: 1-2-3

val1-access:  $W_{\text{twoStage}, 2}$

val2-access:

*Thread twoStage*

```
1: lock(m1);
2: val1 = 1;
3: unlock(m1);
4: lock(m2);
5: val2 = val1 + 1;
6: unlock(m2);
```

*Thread reader*

```
7: lock(m1);
8: if (val1 == 0) {
9:   unlock(m1);
10: return NULL; }
11: t1 = val1;
12: unlock(m1);
13: lock(m2);
14: t2 = val2;
15: unlock(m2);
16: assert(t2==(t1+1));
```

**program counter: 3**

mutexes:  $m1=0$ ;  $m2=0$ ;

global variables: **val1=1**;  $val2=0$ ;

local variables:  $t1 = -1$ ;  $t2 = -1$ ;

# Lazy exploration: interleaving $I_f$

statements: 1-2-3

val1-access:  $W_{\text{twoStage}, 2}$

val2-access:

*Thread twoStage*

```
1: lock(m1);
2: val1 = 1;
3: unlock(m1);
4: lock(m2);
5: val2 = val1 + 1;
6: unlock(m2);
```

*CS1*

*Thread reader*

```
7: lock(m1);
8: if (val1 == 0) {
9:   unlock(m1);
10: return NULL; }
11: t1 = val1;
12: unlock(m1);
13: lock(m2);
14: t2 = val2;
15: unlock(m2);
16: assert(t2==(t1+1));
```

**program counter: 7**

mutexes:  $m1=0$ ;  $m2=0$ ;

global variables:  $val1=1$ ;  $val2=0$ ;

local variables:  $t1= -1$ ;  $t2= -1$ ;

# Lazy exploration: interleaving $I_f$

statements: 1-2-3-7-8-11-12-13-14-15-16

val1-access:  $W_{\text{twoStage},2} - R_{\text{reader},8} - R_{\text{reader},11}$

val2-access:  $R_{\text{reader},14}$

*Thread twoStage*

```
1: lock(m1);
2: val1 = 1;
3: unlock(m1);
4: lock(m2);
5: val2 = val1 + 1;
6: unlock(m2);
```

*CS1*

*Thread reader*

```
7: lock(m1);
8: if (val1 == 0) {
9:   unlock(m1);
10: return NULL; }
11: t1 = val1;
12: unlock(m1);
13: lock(m2);
14: t2 = val2;
15: unlock(m2);
16: assert(t2==(t1+1));
```

**program counter: 16**

mutexes:  $m1=0; m2=0;$

global variables:  $val1=1; val2=0;$

local variables:  **$t1= 1; t2= 0;$**

# Lazy exploration: interleaving $I_f$

statements: 1-2-3-7-8-11-12-13-14-15-16

val1-access:  $W_{\text{twoStage},2} - R_{\text{reader},8} - R_{\text{reader},11}$

val2-access:  $R_{\text{reader},14}$

*Thread twoStage*

```
1: lock(m1);
2: val1 = 1;
3: unlock(m1);
4: lock(m2);
5: val2 = val1 + 1;
6: unlock(m2);
```

*CS1*

*Thread reader*

```
7: lock(m1);
8: if (val1 == 0) {
9:   unlock(m1);
10: return NULL; }
11: t1 = val1;
12: unlock(m1);
13: lock(m2);
14: t2 = val2;
15: unlock(m2);
16: assert(t2==(t1+1));
```

*CS2*

**program counter:** 4

mutexes:  $m1=0; m2=0;$

global variables:  $val1=1; val2=0;$

local variables:  $t1= 1; t2= 0;$

# Lazy exploration: interleaving $I_f$

statements: 1-2-3-7-8-11-12-13-14-15-16-4-5-6

val1-access:  $W_{\text{twoStage},2} - R_{\text{reader},8} - R_{\text{reader},11} - R_{\text{twoStage},5}$

val2-access:  $R_{\text{reader},14} - W_{\text{twoStage},5}$

*Thread twoStage*

```
1: lock(m1);
2: val1 = 1;
3: unlock(m1);
4: lock(m2);
5: val2 = val1 + 1;
6: unlock(m2);
```

*CS1*

*Thread reader*

```
7: lock(m1);
8: if (val1 == 0) {
9:   unlock(m1);
10: return NULL; }
11: t1 = val1;
12: unlock(m1);
13: lock(m2);
14: t2 = val2;
15: unlock(m2);
16: assert(t2==(t1+1));
```

*CS2*

**program counter:** 6

mutexes:  $m1=0; m2=0;$

global variables:  $\text{val1}=1; \text{val2}=2;$

local variables:  $t1= 1; t2= 0;$

# Lazy exploration: interleaving $I_f$

statements: 1-2-3-7-8-11-12-13-14-15-16-4-5-6

val1-access:  $W_{\text{twoStage},2} - R_{\text{reader},8} - R_{\text{reader},11} - R_{\text{twoStage},5}$

val2-access:  $R_{\text{reader},14} - W_{\text{twoStage},5}$

Thread twoStage

```
1: lock(m1);
2: val1 = 1;
3: unlock(m1);
4: lock(m2);
5: val2 = val1 + 1;
6: unlock(m2);
```

CS1

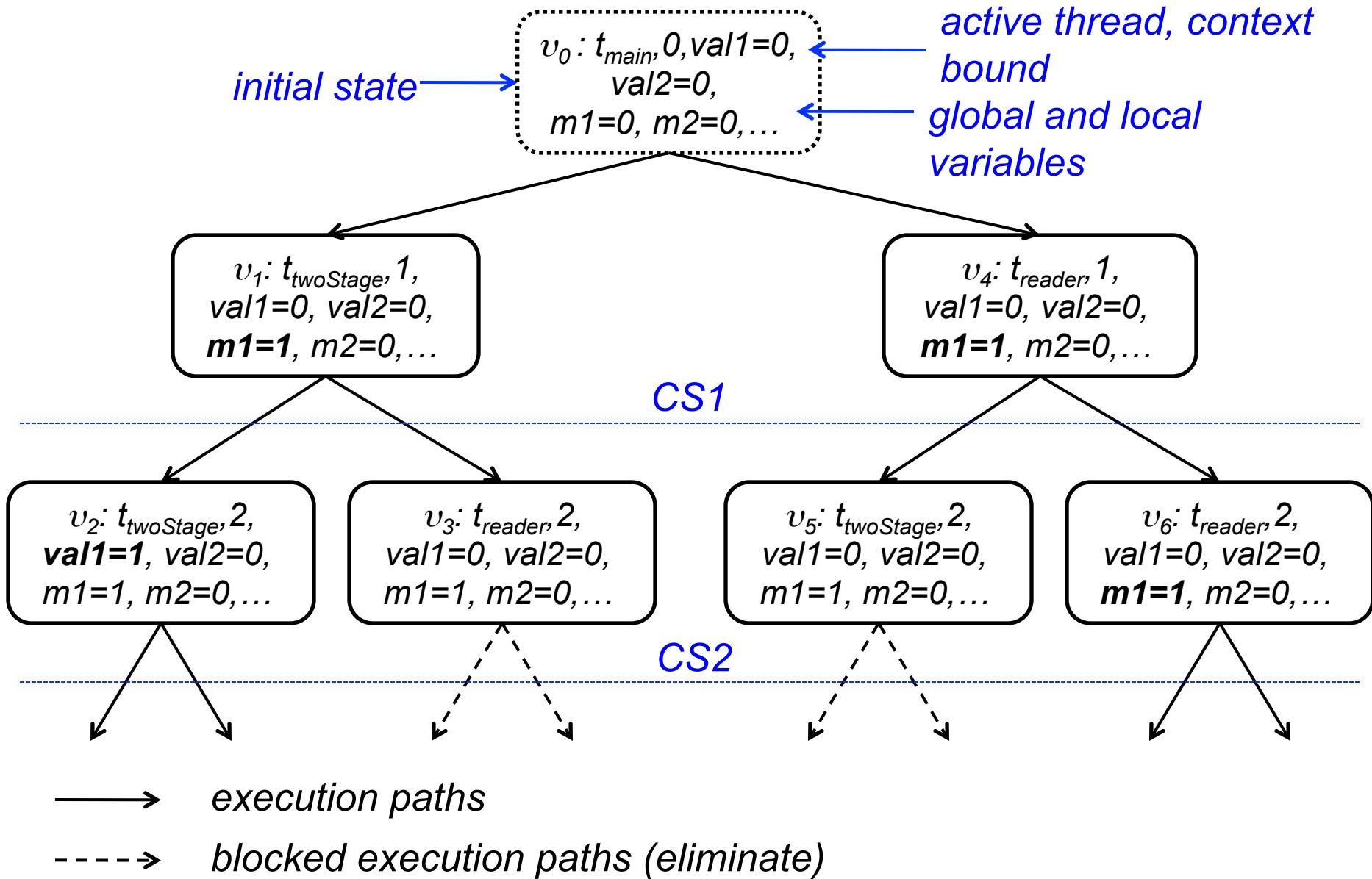
Thread reader

```
7: lock(m1);
8: if (val1 == 0) {
9:   unlock(m1);
10: return NULL; }
11: t1 = val1;
12: unlock(m1);
13: lock(m2);
14: t2 = val2;
15: unlock(m2);
16: assert(t2==(t1+1));
```

CS2

*QF formula is satisfiable,  
i.e., assertion does not hold*

# Lazy Approach: State Transitions



# Exploring the Reachability Tree

- Use a reachability tree (RT) to describe reachable states of a multi-threaded program

# Exploring the Reachability Tree

- Use a reachability tree (RT) to describe reachable states of a multi-threaded program
- Each node in the RT is a tuple  $v = \left( A_i, C_i, s_i, \langle l_i^j, G_i^j \rangle_{j=1}^n \right)_i$  for a given time step  $i$ , where:

# Exploring the Reachability Tree

- Use a reachability tree (RT) to describe reachable states of a multi-threaded program
- Each node in the RT is a tuple  $v = \left( A_i, C_i, s_i, \langle l_i^j, G_i^j \rangle_{j=1}^n \right)_i$  for a given time step  $i$ , where:
  - $A_i$  represents the currently active thread

# Exploring the Reachability Tree

- Use a reachability tree (RT) to describe reachable states of a multi-threaded program
- Each node in the RT is a tuple  $v = \left( A_i, C_i, s_i, \langle l_i^j, G_i^j \rangle_{j=1}^n \right)_i$  for a given time step  $i$ , where:
  - $A_i$  represents the currently active thread
  - $C_i$  represents the context switch number

# Exploring the Reachability Tree

- Use a reachability tree (RT) to describe reachable states of a multi-threaded program
- Each node in the RT is a tuple  $v = \left( A_i, C_i, s_i, \langle l_i^j, G_i^j \rangle_{j=1}^n \right)_i$  for a given time step  $i$ , where:
  - $A_i$  represents the currently active thread
  - $C_i$  represents the context switch number
  - $s_i$  represents the current state

# Exploring the Reachability Tree

- Use a reachability tree (RT) to describe reachable states of a multi-threaded program
- Each node in the RT is a tuple  $v = \left( A_i, C_i, s_i, \langle l_i^j, G_i^j \rangle_{j=1}^n \right)_i$  for a given time step  $i$ , where:
  - $A_i$  represents the currently active thread
  - $C_i$  represents the context switch number
  - $s_i$  represents the current state
  - $l_i^j$  represents the current location of thread  $j$

# Exploring the Reachability Tree

- Use a reachability tree (RT) to describe reachable states of a multi-threaded program
- Each node in the RT is a tuple  $v = \left( A_i, C_i, s_i, \langle l_i^j, G_i^j \rangle_{j=1}^n \right)_i$  for a given time step  $i$ , where:
  - $A_i$  represents the currently active thread
  - $C_i$  represents the context switch number
  - $s_i$  represents the current state
  - $l_i^j$  represents the current location of thread  $j$
  - $G_i^j$  represents the control flow guards accumulated in thread  $j$  along the path from  $l_0^j$  to  $l_i^j$

# Expansion Rules of the RT

**R1 (assign):** If  $l$  is an assignment, we execute  $l$ , which generates  $s_{i+1}$ . We add as child to  $v$  a new node  $v'$

$$v' = \left( A_i, C_i, \underline{s_{i+1}}, \overbrace{\left\langle l_{i+1}^j, G_i^j \right\rangle}_{i+1} \right) \xrightarrow{l_{i+1}^{A_i} = l_i^{A_i} + 1}$$

- we have fully expanded  $v$  if
  - $l$  within an atomic block; or
  - $l$  contains no global variable; or
  - the upper bound of context switches ( $C_i = C$ ) is reached
- if  $v$  is not fully expanded, for each thread  $j \neq A_i$  where  $G_i^j$  is enabled in  $s_{i+1}$ , we thus create a new child node

$$v'_j = \left( \underline{j}, \underline{C_i + 1}, \underline{s_{i+1}}, \left\langle \underline{l_i^j}, \underline{G_i^j} \right\rangle \right)_{i+1}$$

# Expansion Rules of the RT

**R2 (skip):** If  $l$  is a *skip*-statement with target  $l$ , we increment the location of the current thread and continue with it. We explore no context switches:

$$v' = \left( A_i, C_i, s_i, \overbrace{\left\langle l_{i+1}^j, G_i^j \right\rangle}^{l_{i+1}^j} \right)_{i+1} \rightarrow l_{i+1}^j = \begin{cases} l_i^j + 1 & : j = A_i \\ l_i^j & : otherwise \end{cases}$$

# Expansion Rules of the RT

**R2 (skip):** If  $l$  is a *skip*-statement with target  $l$ , we increment the location of the current thread and continue with it. We explore no context switches:

$$v' = \left( A_i, C_i, s_i, \overbrace{\left\langle l_{i+1}^j, G_i^j \right\rangle}^{l_{i+1}^j} \right)_{i+1} \rightarrow l_{i+1}^j = \begin{cases} l_i^j + 1 & : j = A_i \\ l_i^j & : otherwise \end{cases}$$

**R3 (unconditional goto):** If  $l$  is an unconditional *goto*-statement with target  $l$ , we set the location of the current thread and continue with it. We explore no context switches:

$$v' = \left( A_i, C_i, s_i, \overbrace{\left\langle l_{i+1}^j, G_i^j \right\rangle}^{l_{i+1}^j} \right)_{i+1} \rightarrow l_{i+1}^j = \begin{cases} l & : j = A_i \\ l_i^j & : otherwise \end{cases}$$

# Expansion Rules of the RT

**R4 (conditional goto):** If  $l$  is a conditional *goto*-statement with test  $c$  and target  $l$ , we create two child nodes  $v'$  and  $v''$ .

- for  $v'$ , we assume that  $c$  is *true* and proceed with the target instruction of the jump:

$$v' = \left( A_i, C_i, s_i, \overbrace{\left\langle l_{i+1}^j, c \wedge G_i^j \right\rangle}_{i+1} \right)$$
$$l_{i+1}^j = \begin{cases} l & : j = A_i \\ l_i^j & : \text{otherwise} \end{cases}$$

- for  $v''$ , we add  $\neg c$  to the guards and continue with the next instruction in the current thread

$$v'' = \left( A_i, C_i, s_i, \overbrace{\left\langle l_{i+1}^j, \neg c \wedge G_i^j \right\rangle}_{i+1} \right)$$
$$l_{i+1}^j = \begin{cases} l_i^j + 1 & : j = A_i \\ l_i^j & : \text{otherwise} \end{cases}$$

- prune one of the nodes if the condition is determined statically

# Expansion Rules of the RT

**R5 (assume):** If  $l$  is an *assume*-statement with argument  $c$ , we proceed similar to R1.

- we continue with the unchanged state  $s$ , but add  $c$  to all guards, as described in R4
- If  $c \wedge G_i^j$  evaluates to *false*, we prune the execution path

# Expansion Rules of the RT

**R5 (assume):** If  $/$  is an *assume*-statement with argument  $c$ , we proceed similar to R1.

- we continue with the unchanged state  $s$ , but add  $c$  to all guards, as described in R4
- If  $c \wedge G_i^j$  evaluates to *false*, we prune the execution path

**R6 (assert):** If  $/$  is an *assert*-statement with argument  $c$ , we proceed similar to R1.

- we continue with the unchanged state  $s$ , but add  $c$  to all guards, as described in R4
- we generate a verification condition to check the validity of  $c$

# Expansion Rules of the RT

**R5 (start\_thread):** If  $l$  is a *start\_thread* instruction, we add the indicated thread to the set of active threads:

$$v' = \left( A_i, C_i, s_i, \underbrace{\left\langle l_{i+1}^j, G_{i+1}^j \right\rangle}_{j=1}^{n+1} \right)_{i+1}$$

- where  $l_{i+1}^{n+1}$  is the initial location of the thread and  $G_{i+1}^{n+1} = G_i^{A_i}$
- the thread starts with the guards of the currently active thread

# Expansion Rules of the RT

**R5 (start\_thread):** If  $l$  is a *start\_thread* instruction, we add the indicated thread to the set of active threads:

$$v' = \left( A_i, C_i, s_i, \underbrace{\left\langle l_{i+1}^j, G_{i+1}^j \right\rangle}_{j=1}^{n+1} \right)_{i+1}$$

- where  $l_{i+1}^{n+1}$  is the initial location of the thread and  $G_{i+1}^{n+1} = G_i^{A_i}$
- the thread starts with the guards of the currently active thread

**R6 (join\_thread):** If  $l$  is a *join\_thread* instruction with argument  $Id$ , we add a child node:

$$v' = \left( A_i, C_i, s_i, \underbrace{\left\langle l_{i+1}^j, G_i^j \right\rangle}_{j=1} \right)_{i+1}$$

- where  $l_{i+1}^j = l_i^{A_i} + 1$  only if the joining thread Id has exited

# Lazy exploration of interleavings

- Main steps of the algorithm:

1. Initialize the stack with the initial node  $v_0$  and the initial path  $\pi_0 = \langle v_0 \rangle$
2. If the stack is empty, terminate with “no error”.
3. Pop the current node  $v$  and current path  $\pi$  off the stack and compute the set  $v'$  of successors of  $v$  using rules R1-R8.
4. If  $v'$  is empty, derive the VC  $\varphi_k^\pi$  for  $\pi$  and call the SMT solver on it. If  $\varphi_k^\pi$  is satisfiable, terminate with “error”; otherwise, goto step 2.
5. If  $v'$  is not empty, then for each node  $v \in v'$ , add  $v$  to  $\pi$ , and push node and extended path on the stack. goto step 3.

$$\pi = \{v_1, \dots, v_n\}$$
$$\varphi_k^\pi = \overbrace{I(s_0) \wedge R(s_0, s_1) \wedge \dots \wedge R(s_{k-1}, s_k)}^{\text{constraints}} \wedge \overbrace{\neg \phi_k}^{\text{property}}$$

computation path      bound

# Observations about the lazy approach

- naïve but useful:
  - bugs usually manifest with few context switches  
[Qadeer&Rehof'05]

# Observations about the lazy approach

- naïve but useful:
  - bugs usually manifest with few context switches  
[Qadeer&Rehof'05]
  - keep in memory the parent nodes of all unexplored paths only

# Observations about the lazy approach

- naïve but useful:
  - bugs usually manifest with few context switches  
[Qadeer&Rehof'05]
  - keep in memory the parent nodes of all unexplored paths only
  - exploit which transitions are enabled in a given state

# Observations about the lazy approach

- naïve but useful:
  - bugs usually manifest with few context switches  
[Qadeer&Rehof'05]
  - keep in memory the parent nodes of all unexplored paths only
  - exploit which transitions are enabled in a given state
  - bound the number of preemptions (C) allowed per threads
    - ▷ *number of executions:  $O(n^c)$*

# Observations about the lazy approach

- naïve but useful:
  - bugs usually manifest with few context switches  
[Qadeer&Rehof'05]
  - keep in memory the parent nodes of all unexplored paths only
  - exploit which transitions are enabled in a given state
  - bound the number of preemptions ( $C$ ) allowed per threads
    - ▷ *number of executions:*  $O(n^c)$
  - as each formula corresponds to one possible path only, its size is relatively small

# Observations about the lazy approach

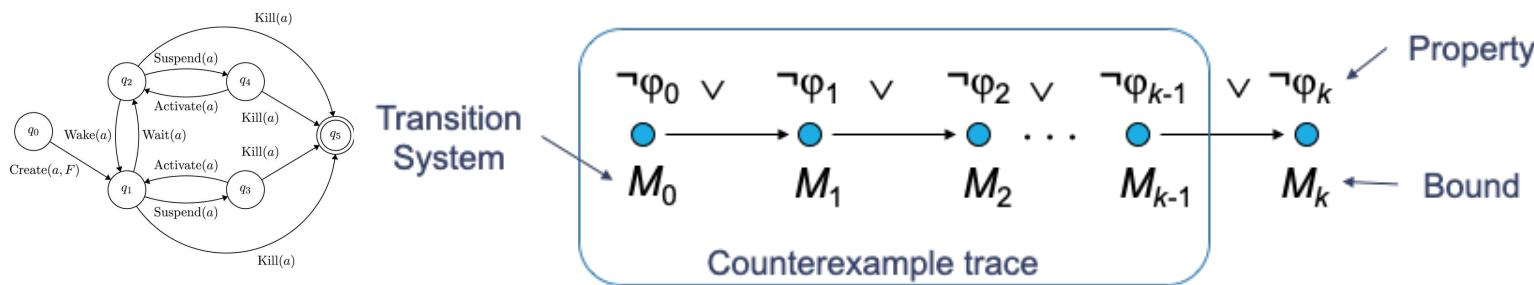
- naïve but useful:
  - bugs usually manifest with few context switches  
[Qadeer&Rehof'05]
  - keep in memory the parent nodes of all unexplored paths only
  - exploit which transitions are enabled in a given state
  - bound the number of preemptions ( $C$ ) allowed per threads
    - ▷ *number of executions:*  $O(n^c)$
  - as each formula corresponds to one possible path only, its size is relatively small
- can suffer performance degradation:
  - in particular for correct programs where we need to invoke the SMT solver once for each possible execution path

# Intended learning outcomes

- Understand **soundness** and **completeness** concerning **detection techniques**
- Emphasize the difference between **static analysis** and **testing / simulation**
- Explain **bounded model checking** of software
- Explain **unbounded model checking** of software

# Revisiting BMC

- Basic Idea: given a transition system  $M$ , check negation of a given property  $\varphi$  up to given depth  $k$ :



- Translated into a VC  $\psi$  such that:  $\psi$  satisfiable iff  $\varphi$  has counterexample of max. depth  $k$

**BMC is aimed at finding bugs; it cannot prove correctness, unless the bound  $k$  safely reaches all program states**

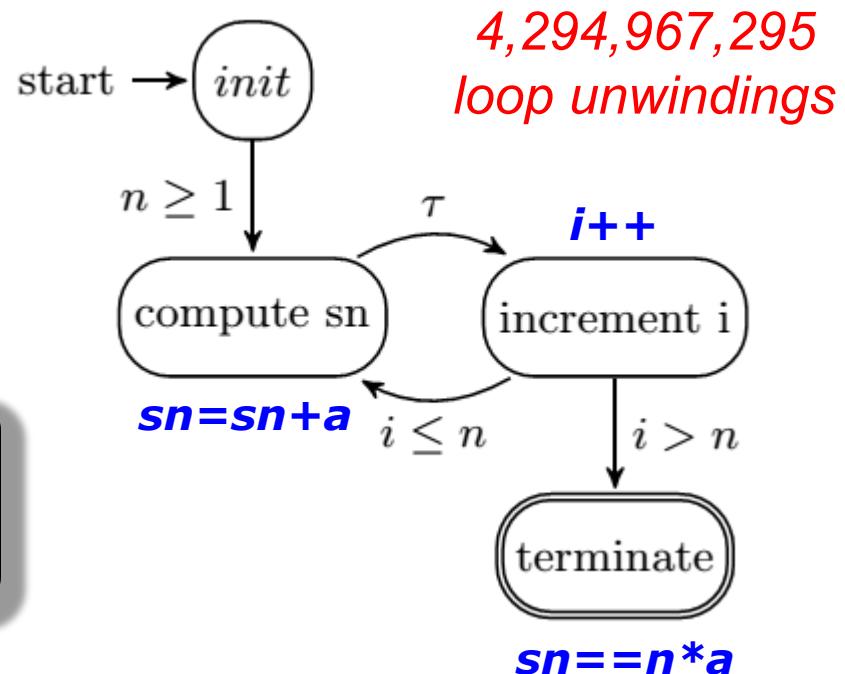
# Difficulties in proving the correctness of programs with loops in BMC

- BMC techniques can falsify properties up to a given depth k
  - they can prove correctness only if an upper bound of k is known (**unwinding assertion**)
    - » BMC tools typically fail to verify programs that contain bounded and unbounded loops

$$S_n = \sum_{i=1}^n a = na, n \geq 1$$



the loop will be unfolded  $2^{n-1}$  times  
(in the worst case,  $2^{32-1}$  times on 32 bits integer)



# Induction-Based Verification

$k$ -induction checks...

- **base case ( $\text{base}_k$ )**: find a counter-example with up to  $k$  loop unwindings (plain BMC)
  - **forward condition ( $\text{fwd}_k$ )**: check that  $P$  holds in all states reachable within  $k$  unwindings
  - **inductive step ( $\text{step}_k$ )**: check that whenever  $P$  holds for  $k$  unwindings, it also holds after next unwinding
    - havoc state
    - run  $k$  iterations
    - assume invariant
    - run final iteration
- ⇒ iterative deepening if inconclusive

# The $k$ -induction algorithm

$k$ =initial bound

**while** *true* **do**

**if**  $base_k$  **then**

**return** *trace*  $s[0..k]$

**else if**  $fwd_k$

**return** *true*

**else if**  $step_k$  **then**

**return** *true*

**end if**

$k = k + 1$

**end**

# The $k$ -induction algorithm

$k$ =initial bound

**while**  $true$  **do**

**if**  $base_k$  **then**

**return**  $trace\ s[0..k]$

**else if**  $fwd_k$

**return**  $true$

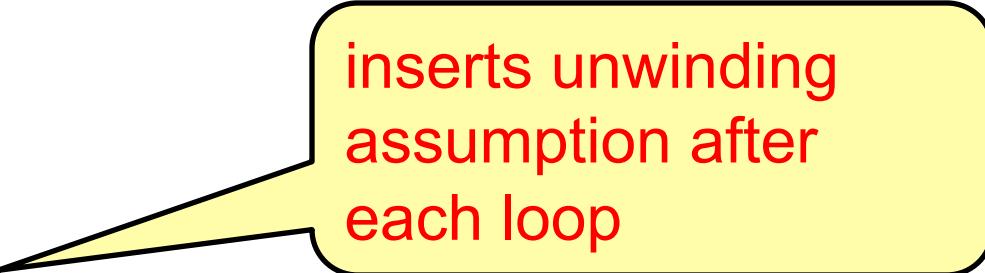
**else if**  $step_k$  **then**

**return**  $true$

**end if**

$k=k+1$

**end**



inserts unwinding  
assumption after  
each loop

# The $k$ -induction algorithm

$k = \text{initial bound}$

**while**  $\text{true}$  **do**

**if**  $\text{base}_k$  **then**

**return**  $\text{trace } s[0..k]$

**else if**  $\text{fwd}_k$

**return**  $\text{true}$

**else if**  $\text{step}_k$  **then**

**return**  $\text{true}$

**end if**

$k = k + 1$

**end**

inserts unwinding  
assumption after  
each loop

inserts unwinding  
assertion after each  
loop

# The $k$ -induction algorithm

$k = \text{initial bound}$

**while** *true* **do**

**if**  $\text{base}_k$  **then**

**return** *trace*  $s[0..k]$

**else if**  $fwd_k$

**return** *true*

**else if**  $step_k$  **then**

**return** *true*

**end if**

$k = k + 1$

**end**

inserts unwinding assumption after each loop

inserts unwinding assertion after each loop

havoc variables that occur in the loop's termination condition

# The $k$ -induction algorithm

$k = \text{initial bound}$

**while** *true* **do**

**if**  $\text{base}_k$  **then**

**return** *trace*  $s[0..k]$

**else if**  $fwd_k$

**return** *true*

**else if**  $step_k$  **then**

**return** *true*

**end if**

$k = k + 1$

**end**

inserts unwinding assumption after each loop

inserts unwinding assertion after each loop

havoc variables that occur in the loop's termination condition

unable to falsify or prove the property

# Running example

Prove that  $S_n = \sum_{i=1}^n a = na$  for  $n \geq 1$

```
unsigned int nondet_uint();
int main() {
    unsigned int i, n=nondet_uint(), sn=0;
    assume (n>=1);
    for(i=1; i<=n; i++)
        sn = sn + a;
    assert(sn==n*a);
}
```

# Running example: *base case*

Insert an **unwinding assumption** consisting of the termination condition after the loop

- find a counter-example with  $k$  loop unwindings

```
unsigned int nondet_uint();
int main() {
    unsigned int i, n=nondet_uint(), sn=0;
    assume (n>=1);
    for(i=1; i<=n; i++)
        sn = sn + a;
    assume(i>n);
    assert(sn==n*a);
}
```

# Running example: *forward condition*

Insert an **unwinding assertion** consisting of the termination condition after the loop

- check that  $P$  holds in all states reachable with  $k$  unwindings

```
unsigned int nondet_uint();
int main() {
    unsigned int i, n=nondet_uint(), sn=0;
    assume (n>=1);
    for(i=1; i<=n; i++)
        sn = sn + a;
    assert(i>n);
    assert(sn==n*a);
}
```

# Running example: *inductive step*

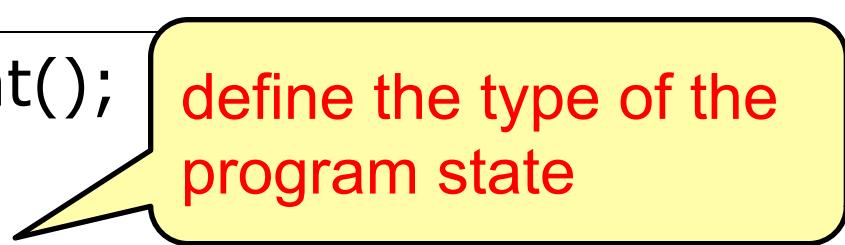
Havoc (only) the variables that occur in the loop's termination and branch conditions

```
unsigned int nondet_uint();
typedef struct state {
    unsigned int i, n, sn;
} statet;
int main() {
    unsigned int i, n=nondet_uint(), sn=0, k;
    assume(n>=1);
    statet cs, s[n];
    cs.i=nondet_uint();
    cs.sn=nondet_uint();
    cs.n=n;
```

# Running example: *inductive step*

Havoc (only) the variables that occur in the loop's termination and branch conditions

```
unsigned int nondet_uint();  
typedef struct state {  
    unsigned int i, n, sn;  
} statet;  
int main() {  
    unsigned int i, n=nondet_uint(), sn=0, k;  
    assume(n>=1);  
    statet cs, s[n];  
    cs.i=nondet_uint();  
    cs.sn=nondet_uint();  
    cs.n=n;
```



define the type of the program state

# Running example: *inductive step*

Havoc (only) the variables that occur in the loop's termination and branch conditions

```
unsigned int nondet_uint();  
typedef struct state {  
    unsigned int i, n, sn;  
} statet;  
int main() {  
    unsigned int i, n=nondet_uint();  
    assume(n>=1);  
    statet cs, s[n];  
    cs.i=nondet_uint();  
    cs.sn=nondet_uint();  
    cs.n=n;
```

define the type of the program state

state vector

# Running example: *inductive step*

Havoc (only) the variables that occur in the loop's termination and branch conditions

```
unsigned int nondet_uint();  
typedef struct state {  
    unsigned int i, n, sn;  
} statet;  
int main() {  
    unsigned int i, n=nondet_uint();  
    assume(n>=1);  
    statet cs, s[n];  
    cs.i=nondet_uint();  
    cs.sn=nondet_uint();  
    cs.n=n;
```

define the type of the program state

state vector

explore all possible values implicitly

# Running example: *inductive step*

BMC is called to verify the assertions where the first arbitrary state is emulated by **nondeterminism**

```
for(i=1; i<=n; i++) {  
    s[i-1]=cs;  
    sn = sn + a;  
    cs.i=i;  
    cs.sn=sn;  
    cs.n=n;  
    assume(s[i-1]!=cs);  
}  
  
assume(i>n);  
assert(sn == n*a);  
}
```

# Running example: *inductive step*

BMC is called to verify the assertions where the first arbitrary state is emulated by **nondeterminism**

```
for(i=1; i<=n; i++) {  
    s[i-1]=cs;  
    sn = sn + a;  
    cs.i=i;  
    cs.sn=sn;  
    cs.n=n;  
    assume(s[i-1]!=cs);  
}  
  
assume(i>n);  
assert(sn == n*a);  
}
```

capture the state *cs* before the iteration

# Running example: *inductive step*

BMC is called to verify the assertions where the first arbitrary state is emulated by **nondeterminism**

```
for(i=1; i<=n; i++) {  
    s[i-1]=cs;  
    sn = sn + a;  
    cs.i=i;  
    cs.sn=sn;  
    cs.n=n;  
    assume(s[i-1]!=cs);  
}  
  
assume(i>n);  
assert(sn == n*a);  
}
```

capture the state *cs*  
before the iteration

capture the state *cs*  
after the iteration

# Running example: *inductive step*

BMC is called to verify the assertions where the first arbitrary state is emulated by **nondeterminism**

```
for(i=1; i<=n; i++) {  
    s[i-1]=cs;  
    sn = sn + a;  
    cs.i=i;  
    cs.sn=sn;  
    cs.n=n;  
    assume(s[i-1]!=cs);  
}  
assume(i>n);  
assert(sn == n*a);  
}
```

capture the state *cs*  
before the iteration

capture the state *cs*  
after the iteration

constraints are  
included by means  
of assumptions

# Running example: *inductive step*

BMC is called to verify the assertions where the first arbitrary state is emulated by **nondeterminism**

```
for(i=1; i<=n; i++) {  
    s[i-1]=cs;  
    sn = sn + a;  
    cs.i=i;  
    cs.sn=sn;  
    cs.n=n;  
    assume(s[i-1]!=cs);  
}  
assume(i>n);  
assert(sn == n)  
}
```

capture the state *cs*  
before the iteration

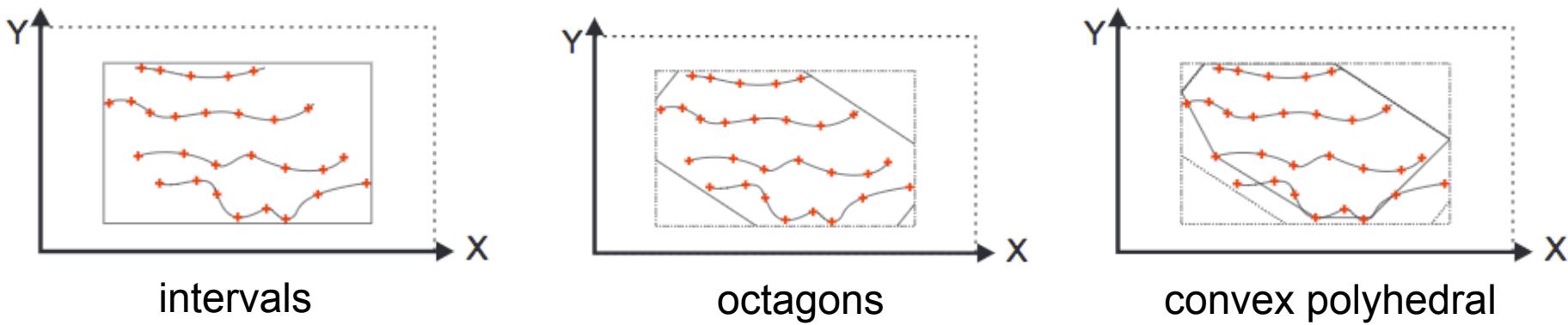
capture the state *cs*  
after the iteration

constraints are  
included by means  
of assumptions

insert unwinding  
assumption

# Automatic Invariant Generation

- Infer invariants using **intervals**, **octagons**, and **convex polyhedral** constraints for the inductive step
  - e.g.,  $a \leq x \leq b$ ;  $x \leq a$ ,  $x-y \leq b$ ; and  $ax + by \leq c$



- Use existing libraries to discover linear/polynomial relations among integer/real variables to infer **loop invariants**
  - compute **pre-** and **post-conditions**

# Running Example: Plain BMC

- Plain BMC unrolls this *while*-loop 100 times...

```
int main() {  
    int x=0, t=0, phase=0;  
    while(t<100) {  
        if(phase==0) x=x+2;  
        if(phase==1) x=x-1;  
        phase=1-phase;  
        t++;  
    }  
    assert(x<=100);  
    return 0;  
}
```

```
$esbmc example.c --clang-frontend  
ESBMC version 4.2.0 64-bit x86_64 macos  
file example.c: Parsing  
Converting  
Type-checking example  
Generating GOTO Program  
GOTO program creation time: 0.232s  
GOTO program processing time: 0.001s  
Starting Bounded Model Checking  
Unwinding loop 1 iteration 1 file example.c line 5 function  
main  
Unwinding loop 1 iteration 2 file example.c line 5 function  
main  
...  
Unwinding loop 1 iteration 100 file example.c line 5 function  
main  
Symex completed in: 0.340s (313 assignments)  
Slicing time: 0.000s  
Generated 1 VCC(s), 0 remaining after simplification  
VERIFICATION SUCCESSFUL  
BMC program time: 0.340s
```

# Running Example: $k$ -induction + invariants

- Inductive step proves correctness for  $k$ -step 2...

```
int main() {
    int x=0, t=0, phase=0;
    while(t<100) {
        assume(-2*x+t+3*phase == 0);
        assume(3-2*x+t >= 0);
        assume(-x+2*t >= 0);
        assume(147+x-2*t >= 0);
        assume(2*x-t >= 0);
        if(phase==0) x=x+2;
        if(phase==1) x=x-1;
        phase=1-phase;
        t++;
    }
    assert(x<=100);
    return 0;
}
```

```
$esbmc example.c --clang-frontend --k-induction
*** K-Induction Loop Iteration 2 ***
*** Checking inductive step
Starting Bounded Model Checking
Unwinding loop 1 iteration 1 file example_pagai.c line 6 function main
Unwinding loop 1 iteration 2 file example_pagai.c line 6 function main
Symex completed in: 0.002s (53 assignments)
Slicing time: 0.000s
Generated 1 VCC(s), 1 remaining after simplification
No solver specified; defaulting to Boolector
Encoding remaining VCC(s) using bit-vector arithmetic
Encoding to solver time: 0.001s
Solving with solver Boolector 2.4.0
Encoding to solver time: 0.001s
Runtime decision procedure: 0.144s
VERIFICATION SUCCESSFUL
BMC program time: 0.148s
Solution found by the inductive step (k = 2)
```

## inductive invariants

reuse k-induction counterexamples to speed-up bug finding  
reuse results of previous steps (caching SMT queries)

# 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** and **unbounded model checking of software**
  - they have been applied successfully to verify **single- and multi-threaded software**