Systems and Software Verification Laboratory

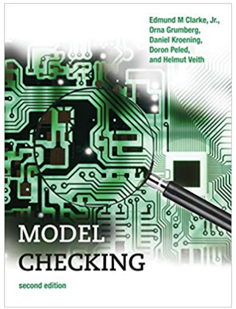
Detection of Software Vulnerabilities: Static Analysis

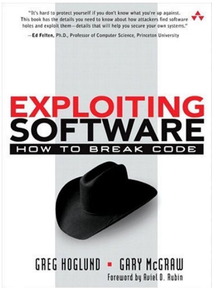
Lucas Cordeiro
Department of Computer Science
lucas.cordeiro@manchester.ac.uk

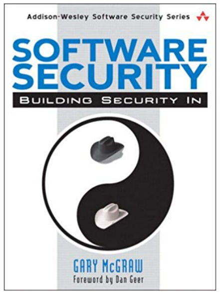
Detection of Software Vulnerabilities

- Lucas Cordeiro (Formal Methods Group)
  - *lucas.cordeiro@manchester.ac.uk*
  - Office: 2.28
  - Office hours: 15-16 Tuesday, 14-15 Wednesday
- Textbook:
  - *Model checking* (Chapter 14)
  - *Exploiting Software: How to Break Code* (Chapter 7)
  - *C How to Program* (Chapter 1)

Rashid et al.: *The Cyber Security Body of Knowledge*, CyBOK, v1.0, 2019







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**

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**

Motivating Example

- functionality demanded increased significantly
  - peer reviewing and testing
- multi-core processors with scalable shared memory / message passing
  - software model checking and testing

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

Deadlock

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

Detection of Vulnerabilities

- Detect the presence of vulnerabilities in the code during the development, testing and maintenance
- Techniques to detect vulnerabilities must make trade-offs 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 is can 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 trigger a vulnerability
    - A common dynamic approach is software testing: the tester writes test cases with concrete inputs, and specific checks for the outputs

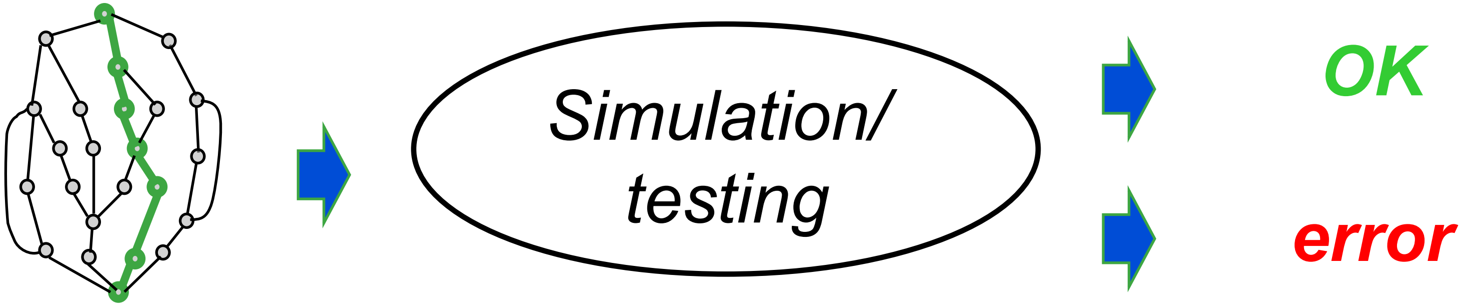
Detection of Vulnerabilities

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

Intended learning outcomes

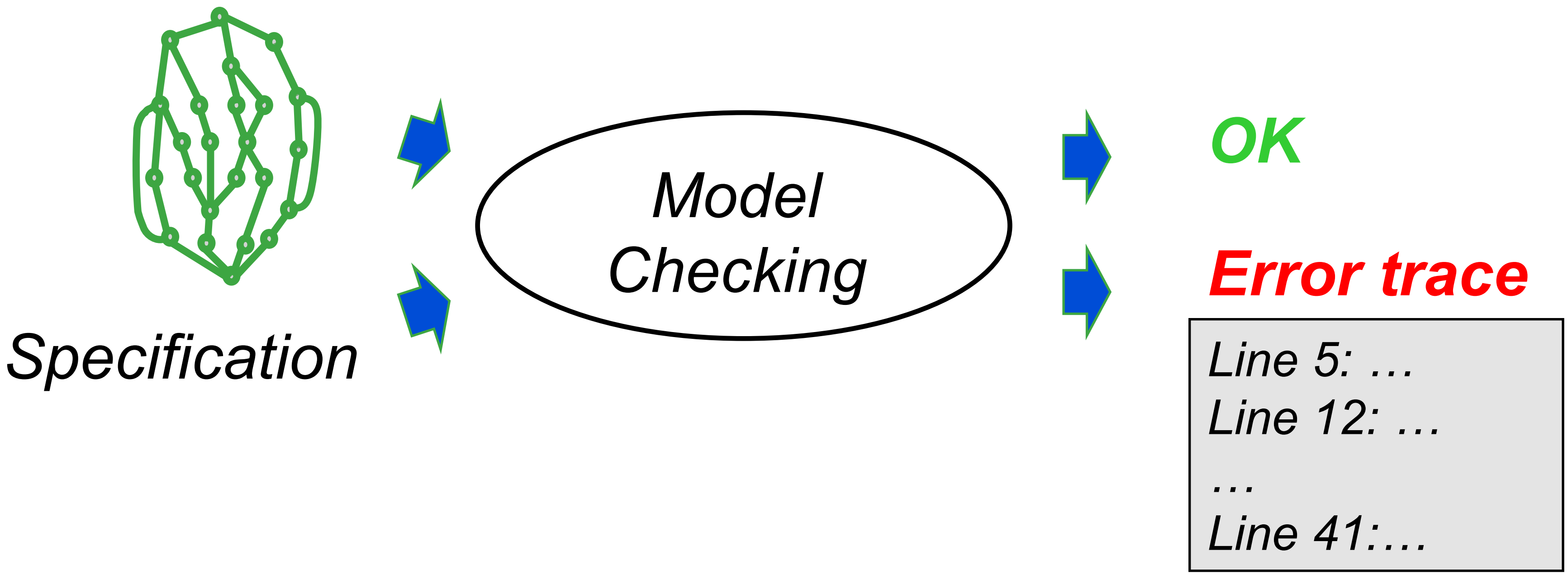
- Understand **soundness** and **completeness** concerning **detection techniques**
- Emphasize the difference between **static analysis** and **testing / simulation**
- Explain **bounded model checking of software**
- Explain **unbounded model checking of software**

Static analysis vs Testing/ Simulation



- Checks only some of the system executions
- May miss errors

Static analysis vs Testing/ Simulation

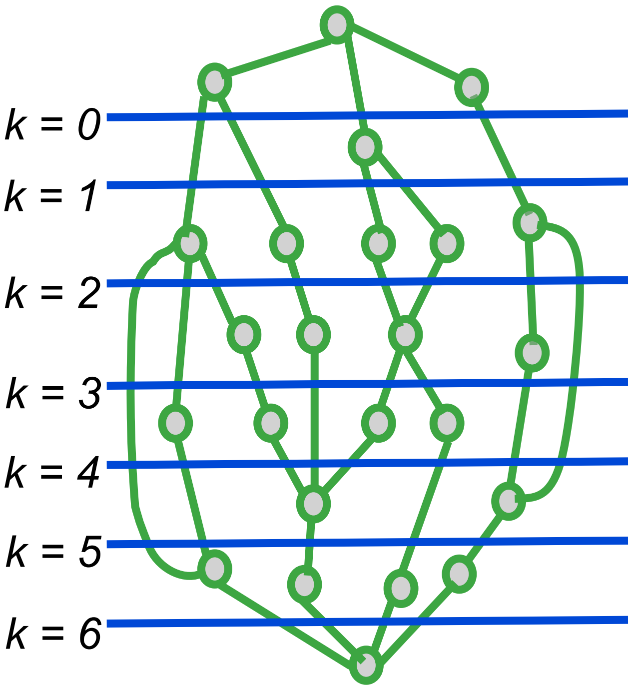


- Exhaustively explores all executions
- Report errors as traces

Avoiding state space explosion

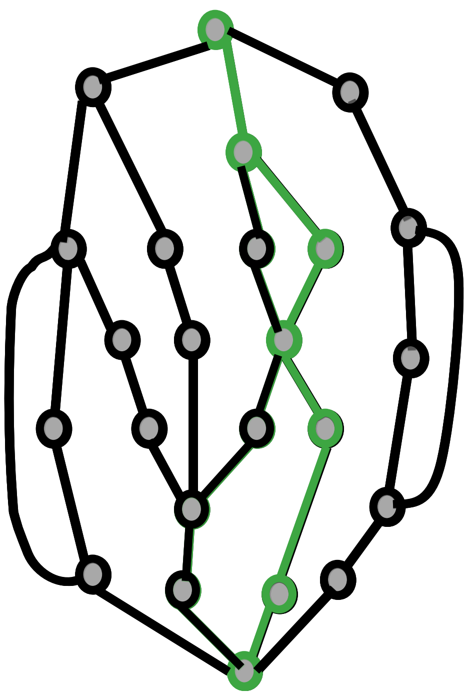
- Bounded Model Checking (BMC)
  - Breadth-first search (BFS) approach
- Symbolic Execution
  - Depth-first search (DFS) approach

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



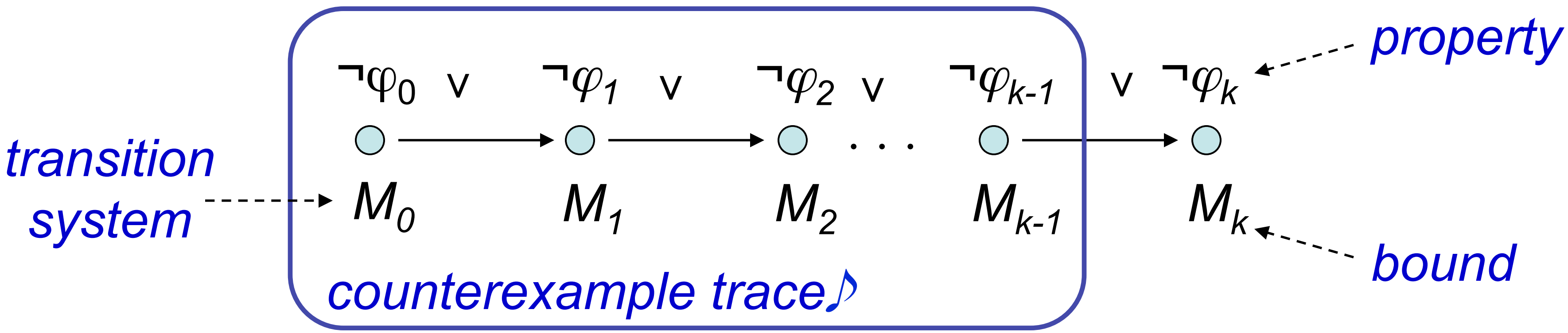
- Symbolic execution explores all paths individually
- Can only prove correctness if all paths are explored

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**

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 ψ such that

  ψ satisfiable iff φ has counterexample of max. depth k
- has been applied successfully to verify (sequential) software

BMC of Multi-threaded Software

- concurrency bugs are tricky to **reproduce/debug** because they usually occur under specific thread interleavings
  - most common errors: 67% related to atomicity and order violations, 30% related to deadlock [Lu et al.' 08]
- problem: the number of interleavings grows exponentially with the number of threads (n) and program statements (s)
  - number of executions: $O(n^s)$
  - context switches among threads increase the number of possible executions

BMC of single- and multi-threaded software

Bounded Model Checking of Software:

- symbolically executes programs into SSA, produces QF formulae
- unrolls loops and recursions up to a maximum bound *k*
- check whether corresponding formula is satisfiable
  - safety properties (array bounds, pointer dereferences, overflows,...)
  - user-specified properties

multi-threaded programs:

- combines explicit-state with symbolic model checking
- symbolic state hashing & monotonic POR
- context-bounded analysis (optional context bound)

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

Satisfiability Modulo Theories (2)

- Given
  - a decidable Σ-theory T
  - a quantifier-free formula φ

  φ 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

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

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

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

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

replace b' by c−3 in the inequality

$$step2: g(select(store(a,c,12), c-3+3)) \neq g(c-3-c+4) \wedge c-3=c-3 \wedge c+1=d-4$$

using facts about bit-vector arithmetic

$$step3: g(select(store(a,c,12), c)) \neq g(1) \wedge c-3=c-3 \wedge c+1=d-4$$

Satisfiability Modulo Theories (4)

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

⇩ 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*).

$$step\,5: SAT\,(\mathrm{c}=5, \mathrm{d}=10)$$

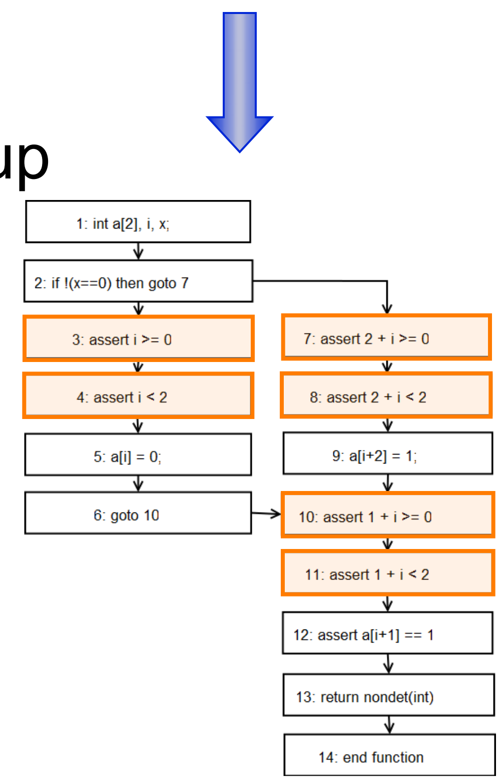
- SMT solvers also apply:
  - standard algebraic reduction rules $\boxed{r \wedge false \mapsto false}$
  - contextual simplification $\boxed{a = 7 \wedge p(a) \mapsto a = 7 \wedge p(7)}$

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

  $\Big\}$ crucial
- front-end converts unrolled and optimized program into SSA

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

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

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
- extraction of constraints C and properties P
  – specific to selected SMT solver, uses theories
- satisfiability check of C ∧ ¬P

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

$$C := \begin{bmatrix} g_1 := (x_1 = 0) \\ \wedge\ a_1 := store(a_0, i_0, 0) \\ \wedge\ a_2 := a_0 \\ \wedge\ a_3 := store(a_2, 2 + i_0, 1) \\ \wedge\ a_4 := ite(g_1, a_1, a_3) \end{bmatrix}$$

$$P := \begin{bmatrix} i_0 \geq 0 \wedge i_0 < 2 \\ \wedge\ 2 + i_0 \geq 0 \wedge 2 + i_0 < 2 \\ \wedge\ 1 + i_0 \geq 0 \wedge 1 + i_0 < 2 \\ \wedge\ select(a_4, i_0 + 1) = 1 \end{bmatrix}$$

Encoding of Numeric Types

- SMT solvers typically provide different encodings for numbers:
  - abstract domains ($\mathbf{Z}$, $\mathbf{R}$)
  - 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 $\mathbf{Z}$ or $\mathbf{R}$

doesn't hold for bitvectors, due to possible overflows

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

Encoding Numeric Types as Bitvectors

Bitvector encodings need to handle

- type casts and implicit conversions
  - arithmetic conversions implemented using word-level functions (part of the bitvector theory: Extract, SignExt, …)
    - o different conversions for every pair of types
    - o uses type information provided by front-end
  - conversion to / from bool via if-then-else operator

    $t = ite(v \neq k, true, false)$ //conversion to bool

    $v = ite(t, 1, 0)$ //conversion from bool
- arithmetic over- / underflow
  - standard requires modulo-arithmetic for unsigned integer

    $unsigned\_overflow \Leftrightarrow (r - (r \bmod 2^w)) < 2^w$
  - define error literals to detect over- / underflow for other types

    $$res\_op \Leftrightarrow \neg\, overflow(x, y) \wedge \neg\, underflow(x, y)$$

    - o similar to conversions

Floating-Point Numbers

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

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

// m = number of bits of i

$$f = \begin{cases} Extract(b, n_b - 1, n_b - n_b) & : \quad n_a \leq n_b \\ Extract(b, n_b, 0) \mathbin{@} SignExt(b, n_a - n_b) & : \quad otherwise \end{cases}$$

// n = number of bits of f

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

$$a = \begin{cases} \dfrac{\left(i * p + \left(\dfrac{f * p}{2^n} + 1\right)\right)}{p} & : \quad f \neq 0 \\ i \quad : \quad otherwise \end{cases}$$

// p = number of decimal places

Encoding of Pointers

- arrays and records / tuples typically handled directly by SMT-solver
- pointers modelled as tuples
  - p.o ≙ representation of underlying object
  - p.i ≙ 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

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

Store index at position 1

Update index

Encoding of Pointers

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

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

```
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);
}
```

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

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 ≜ memory array
  - s ≜ size in bytes of m
  - ρ ≜ unique identifier
  - υ ≜ indicate whether the object is still alive
  - l ≜ 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

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

Encoding of Memory Allocation

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

$$l_{valid\_object} \Leftrightarrow (l_{is\_dynamic\_object} \Rightarrow d_o.\upsilon)$$

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

$$l_{deallocated\_object} \Leftrightarrow (l_{is\_dynamic\_object} \Rightarrow \neg\, d_o.\upsilon)$$

Example of Memory Allocation

```
#include <std
void main() {
  char *p = m
  char *q = malloc(5); // ρ = 2
  p=q;
  free(p)
  p = malloc(5);          // ρ = 3
  free(p)
}
```

memory leak: pointer reassignment makes $d_{o1}.\upsilon$ 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)
}
```

$$P := \left(\neg d_{o1}.\upsilon \wedge \neg d_{o2}.\upsilon \; \neg d_{o3}.\upsilon \right)$$

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

Example of Memory Allocation

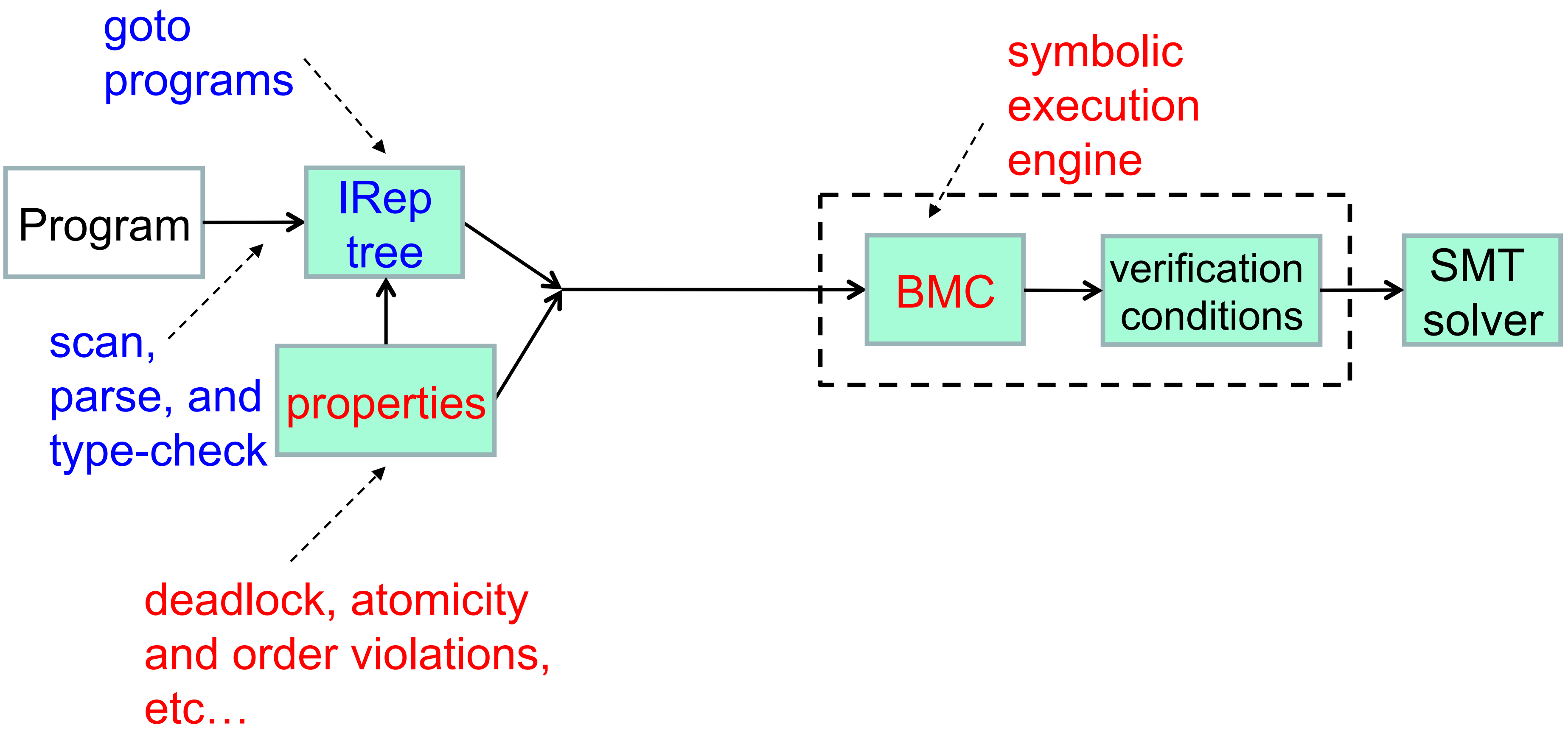
```
#include <stdlib.h>
void main() {
  char *p = malloc(5);  // ρ = 1
  char *q = malloc(5);  // ρ = 2
  p=q;
  free(p)
  p = malloc(5);        // ρ = 3
  free(p)
}
```

$$P := \left(\neg d_{o1}.\upsilon \wedge \neg d_{o2}.\upsilon \; \neg d_{o3}.\upsilon \right)$$

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

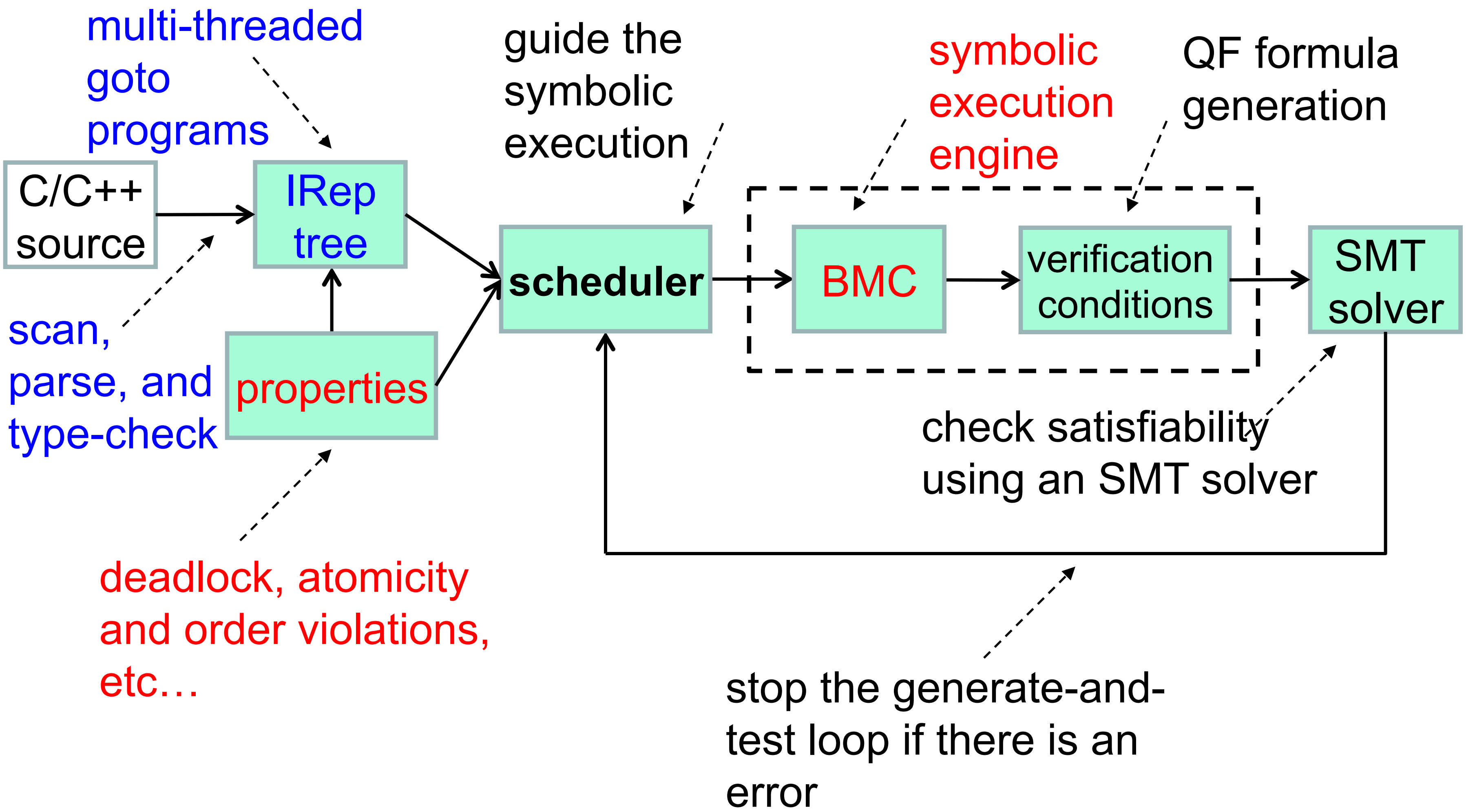
BMC Architecture

- 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);
```

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

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

program counter: 0
mutexes: m1=0; m2=0;
global variables: val1=0; val2=0;
local variabes: t1= -1; t2= -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 variabes: 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 variabes: t1= -1; t2= -1;

Lazy exploration: interleaving I_s

statements: 1-2

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

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 variabes: t1= -1; t2= -1;

Lazy exploration: interleaving I_s

statements: 1-2-3

val1-access: $W_{twoStage,_2}$

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 variabes: t1= -1; t2= -1;

Lazy exploration: interleaving I_s

statements: 1-2-3-7

val1-access: $W_{twoStage,_2}$

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 variabes: t1= -1; t2= -1;

Lazy exploration: interleaving I$_2$

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

statements: 1-2-3-7-8

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

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: 8
mutexes: m1=1; m2=0;
global variables: val1=1; val2=0;
local variabes: t1= -1; t2= -1;

Lazy exploration: interleaving I_s

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

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

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 variabes: **t1= 1**; t2= -1;*

Lazy exploration: interleaving I_s

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

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

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: 12

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

global variables: val1=1; val2=0;

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

Lazy exploration: interleaving I_s

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

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

val2-access:

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

CS1

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: 4

mutexes: m1=0; m2=0;

global variables: val1=1; val2=0;

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

Lazy exploration: interleaving I_s

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

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

val2-access:

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

CS1

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: 4
mutexes: m1=0; ***m2=1****;*
global variables: val1=1; val2=0;
local variabes: t1= 1; t2= -1;

Lazy exploration: interleaving I_s

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

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

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

```
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));
```

CS1

CS2

program counter: 5

mutexes: m1=0; m2=1;

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

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

Lazy exploration: interleaving I_s

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

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

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

```
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: 6

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

global variables: val1=1; val2=2;

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

Lazy exploration: interleaving I_s

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

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

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

```
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 variabes: t1= 1; t2= -1;

Lazy exploration: interleaving I_s

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

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

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

```
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 variabes: t1= 1; t2= -1;

Lazy exploration: interleaving I_s

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

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

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

```
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 variabes: 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: $W_{twoStage,2}$ - $R_{reader,8}$ - $R_{reader,11}$ - $R_{twoStage,5}$

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

```
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: 15

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

global variables: val1=1; val2=2;

local variabes: t1= 1; t2= 2;

Lazy exploration: interleaving I_s

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

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

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

```
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: 16

mutexes: m1=0; m2=0;

global variables: val1=1; val2=2;

local variabes: t1= 1; t2= 2;

Lazy exploration: interleaving I_s

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

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

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

```
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));
```

QF formula is unsatisfiable, i.e., assertion holds

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);
```

```
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 variabes: t1= -1; t2= -1;

Lazy exploration: interleaving I_f

statements: 1-2-3

val1-access: $W_{twoStage,_2}$

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 variabes: t1= -1; t2= -1;

Lazy exploration: interleaving I_f

statements: 1-2-3

val1-access: $W_{twoStage,_2}$

val2-access:

```
Thread twoStage
1:  lock(m1);                CS1
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: 7

mutexes: m1=0; m2=0;

global variables: val1=1; val2=0;

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

Lazy exploration: interleaving I_f

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

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

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

```
Thread twoStage
1:  lock(m1);                CS1
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: 16
mutexes: m1=0; m2=0;
global variables: val1=1; val2=0;
*local variabes: **t1= 1**; **t2= 0**;*

Lazy exploration: interleaving I_f

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

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

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

```
Thread twoStage
1:  lock(m1);                CS1
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));
```

CS2

program counter: 4

mutexes: m1=0; m2=0;

global variables: val1=1; val2=0;

local variabes: 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: $W_{twoStage,2}$ - $R_{reader,8}$ - $R_{reader,11}$ - $R_{twoStage,5}$

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

```
Thread twoStage
1:  lock(m1);                CS1
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));
```

CS2

program counter: 6

mutexes: m1=0; m2=0;

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

local variabes: 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: $W_{twoStage,2}$ - $R_{reader,8}$ - $R_{reader,11}$ - $R_{twoStage,5}$

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

Thread twoStage

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

CS1

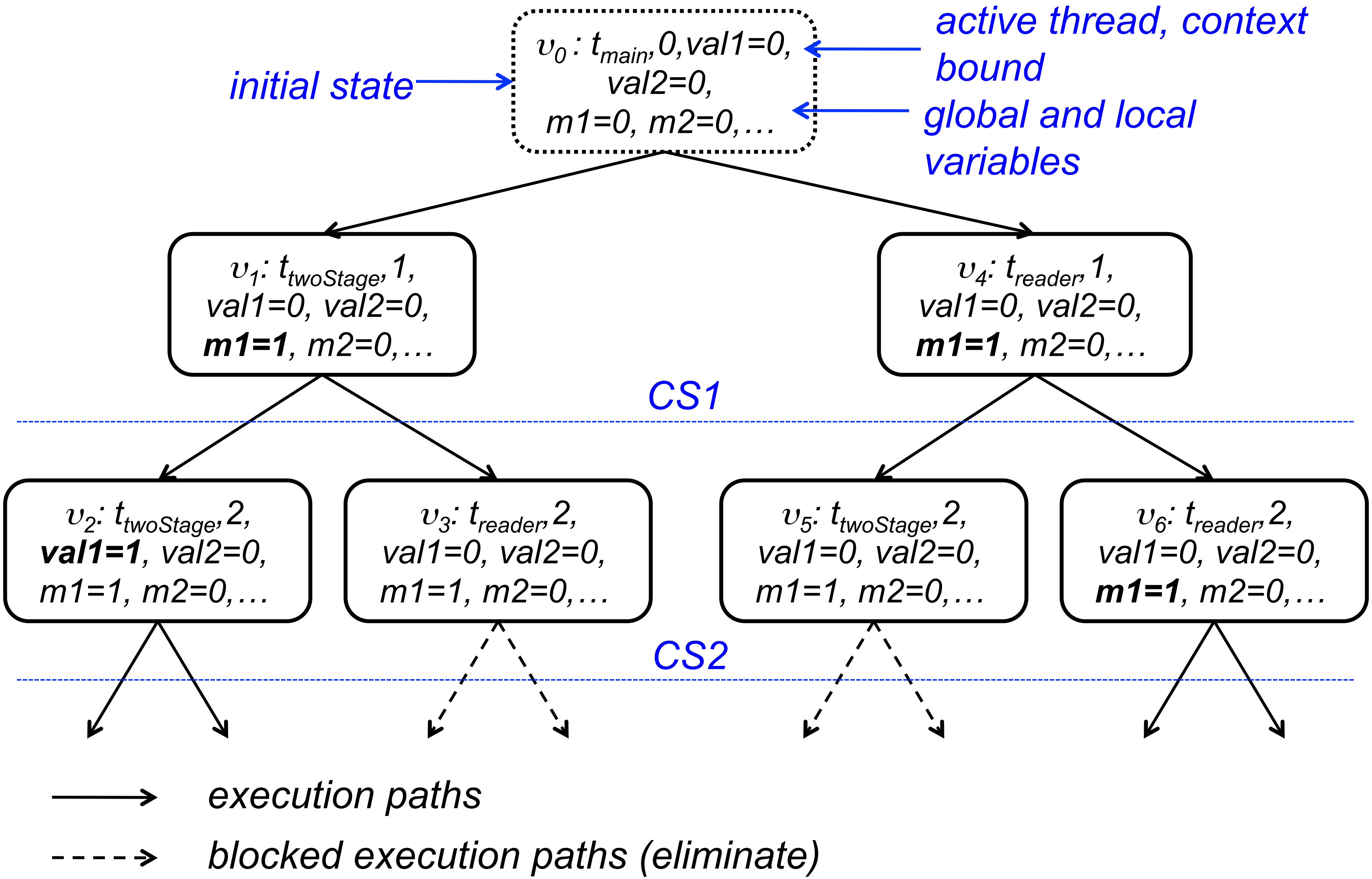
Thread reader

```
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



Lazy exploration of interleavings

- Main steps of the algorithm:

1. Initialize the stack with the initial node ν_0 and the initial path $\pi_0 = \langle \upsilon_0 \rangle$
2. If the stack is empty, terminate with "no error".
3. Pop the current node υ and current path π off the stack and compute the set υ' of successors of υ using rules R1-R8.
4. If υ' is empty, derive the VC φ_k^π for π and call the SMT solver on it. If φ_k^π is satisfiable, terminate with "error"; otherwise, goto step 2.
5. If υ' is not empty, then for each node $\upsilon \in \upsilon'$, add ν to π, and push node and extended path on the stack. goto step 3.

computation path

$\pi = \{ \upsilon_1, \ldots \upsilon_n \}$

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

bound

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 $\upsilon = \left(A_i, C_i, s_i, \left\langle l_i^j, G_i^j \right\rangle_{j=1}^{n} \right)_i$ 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
- expand the RT by executing symbolically each instruction of the multi-threaded program

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 υ a new node υ'

$$l_{i+1}^{A_i} = l_i^{A_i} + 1$$

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

- we have fully expanded υ if
  - I within an atomic block; or
  - l contains no global variable; or
  - the upper bound of context switches ($C_i = C$) is reached
- if υ 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

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

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:

$$l_{i+1}^{j} = \begin{cases} l_i^j + 1 & : \quad j = A_i \\ l_i^j & : \quad otherwise \end{cases}$$

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

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:

$$l_{i+1}^{j} = \begin{cases} l & : \quad j = A_i \\ l_i^j & : \quad otherwise \end{cases}$$

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

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 υ' and υ''.

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

$$l_{i+1}^{j} = \begin{cases} l & : \quad j = A_i \\ l_i^j & : \quad otherwise \end{cases}$$

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

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

$$l_{i+1}^{j} = \begin{cases} l_i^j + 1 & : \quad j = A_i \\ l_i^j & : \quad otherwise \end{cases}$$

$$\upsilon'' = \left(A_i, C_i, s_i, \left\langle \underline{l_{i+1}^{j}}, \neg c \underline{\wedge G_i^j} \right\rangle\right)_{i+1}$$

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

Expansion Rules of the RT

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

- we continue with the unchanged state s_i 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 *I* is an *assert*-statement with argument *c*, we proceed similar to R1.

- we continue with the unchanged state s_i 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 *I* is a *start_thread* instruction, we add the indicated thread to the set of active threads:

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

- 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 *I* is a *join_thread* instruction with argument *Id*, we add a child node:

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

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

Observations about the lazy approach

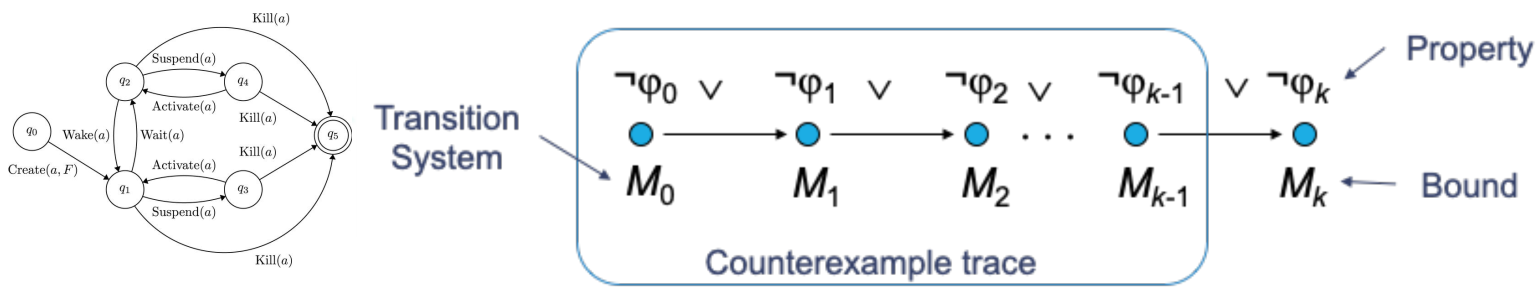
- naïve but useful:
  - bugs usually manifest with few context switches [Qadeer&Rehof'05]
  - keep in memory the parent nodes of all unexplored paths only
  - exploit which transitions are enabled in a given state
  - bound the number of preemptions (C) allowed per threads
    - ▹ *number of executions: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



- Translated into a VC ψ such that: ψ *satisfiable iff* φ *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

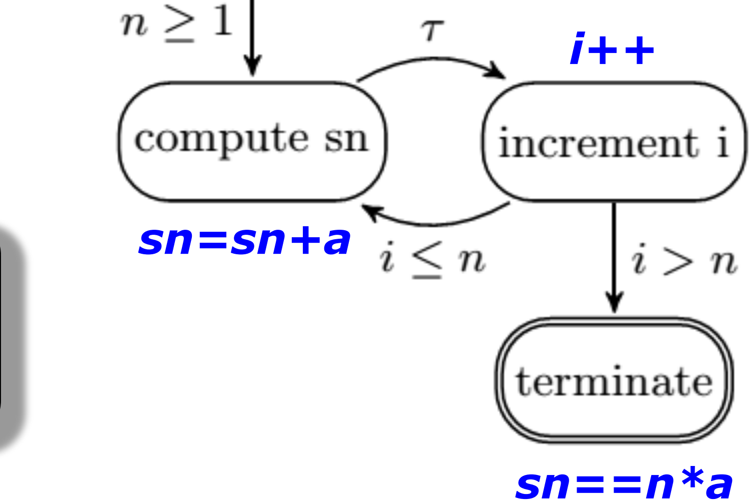
- BMC techniques can falsify properties up to a given depth k

– 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

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

4,294,967,295 loop unwindings



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 (*base*$_k$):** find a counter-example with up to *k* loop unwindings (plain BMC)
- **forward condition (*fwd*$_k$):** check that *P* holds in all states reachable within *k* unwindings
- **inductive step (*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=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

inserts unwinding assertion after each loop

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

inserts unwinding assertion after each loop

havoc variables that occur in the loop's termination condition

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

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(), 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

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(), 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

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*a);
}
```

capture the state *cs* before the iteration

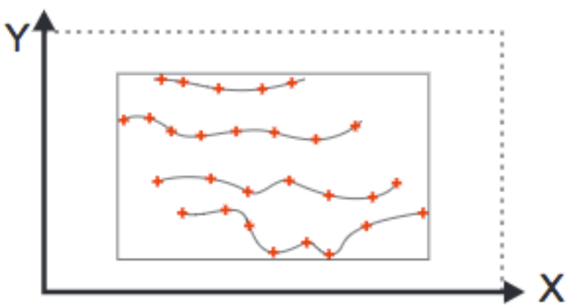
capture the state *cs* after the iteration

constraints are included by means of assumptions

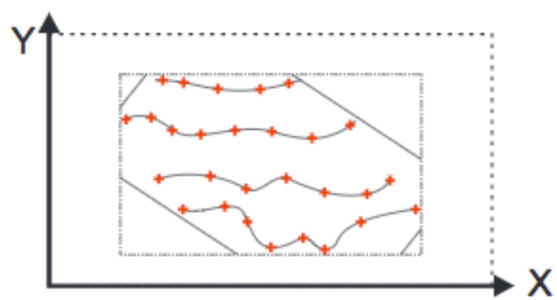
insert unwinding assumption

Automatic Invariant Generation

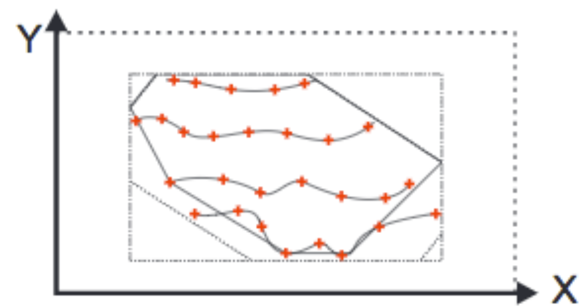
- Infer invariants using **intervals**, **octagons**, and **convex polyhedral** constraints for the inductive step
  - *e.g., $a \leq x \leq b$*; *$x \leq a$, x-$y \leq b$*; and *$ax + by \leq c$*



intervals



octagons



convex polyhedral

- 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…

```c
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**