

10001011
01001100
10110111
01100100
10101101



**Systems and Software
Verification Laboratory**



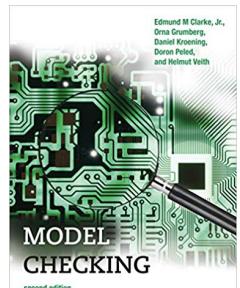
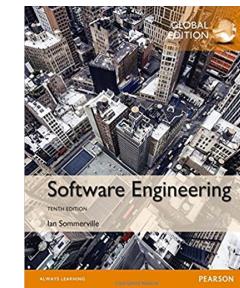
Detection of Software Vulnerabilities: Static Analysis (Part II)

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

Static Analysis

- 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)
 - *Software model checking*. ACM Comput. Surv., 2009
 - *The Cyber Security Body of Knowledge*, 2019
 - *Software Engineering* (Chapters 8, 13)

Lecture notes “SAT/SMT-Based Bounded Model Checking of Software” by Bernd Fischer



Intended learning outcomes

- Introduce typical **BMC architectures** for verifying **software systems**
- Explain **bounded model checking** of multi-threaded software
- Explain **unbounded model checking** of software

Intended learning outcomes

- Introduce typical **BMC architectures** for verifying **software systems**
- Explain bounded model checking of multi-threaded software
- Explain unbounded model checking of software

SAT/SMT-based BMC tools for C

- CBMC (C Bounded Model Checker)
 - <http://www.cprover.org/>
 - SAT-based (MiniSat) “workhorse”
 - also SystemC frontend

SAT/SMT-based BMC tools for C

- CBMC (C Bounded Model Checker)
 - <http://www.cprover.org/>
 - SAT-based (MiniSat) “workhorse”
 - also SystemC frontend
- ESBMC (Embedded Systems Bounded Model Checker)
 - <http://esbmc.org>
 - SMT-based (Z3, Boolector)
 - branched off CBMC, also (rudimentary) C++ frontend

SAT/SMT-based BMC tools for C

- CBMC (C Bounded Model Checker)
 - <http://www.cprover.org/>
 - SAT-based (MiniSat) “workhorse”
 - also SystemC frontend
- ESBMC (Embedded Systems Bounded Model Checker)
 - <http://esbmc.org>
 - SMT-based (Z3, Boolector)
 - branched off CBMC, also (rudimentary) C++ frontend
- LLBMC (Low-level Bounded Model Checker)
 - <http://llbmc.org>
 - SMT-based (Boolector or STP)
 - uses LLVM intermediate language

⇒ share common high-level architecture

SAT/SMT-based BMC tools for C

Typical features:

- full language support
 - bit-precise operations, structs, arrays, ...
 - heap-allocated memory
 - concurrency

SAT/SMT-based BMC tools for C

Typical features:

- full language support
 - bit-precise operations, structs, arrays, ...
 - heap-allocated memory
 - concurrency
- built-in safety checks
 - overflow, div-by-zero, array out-of-bounds indexing, ...
 - memory safety: nil pointer deref, memory leaks, ...
 - deadlocks, race conditions

SAT/SMT-based BMC tools for C

Typical features:

- full language support
 - bit-precise operations, structs, arrays, ...
 - heap-allocated memory
 - concurrency
- built-in safety checks
 - overflow, div-by-zero, array out-of-bounds indexing, ...
 - memory safety: nil pointer deref, memory leaks, ...
 - deadlocks, race conditions
- user-specified assertions and error labels

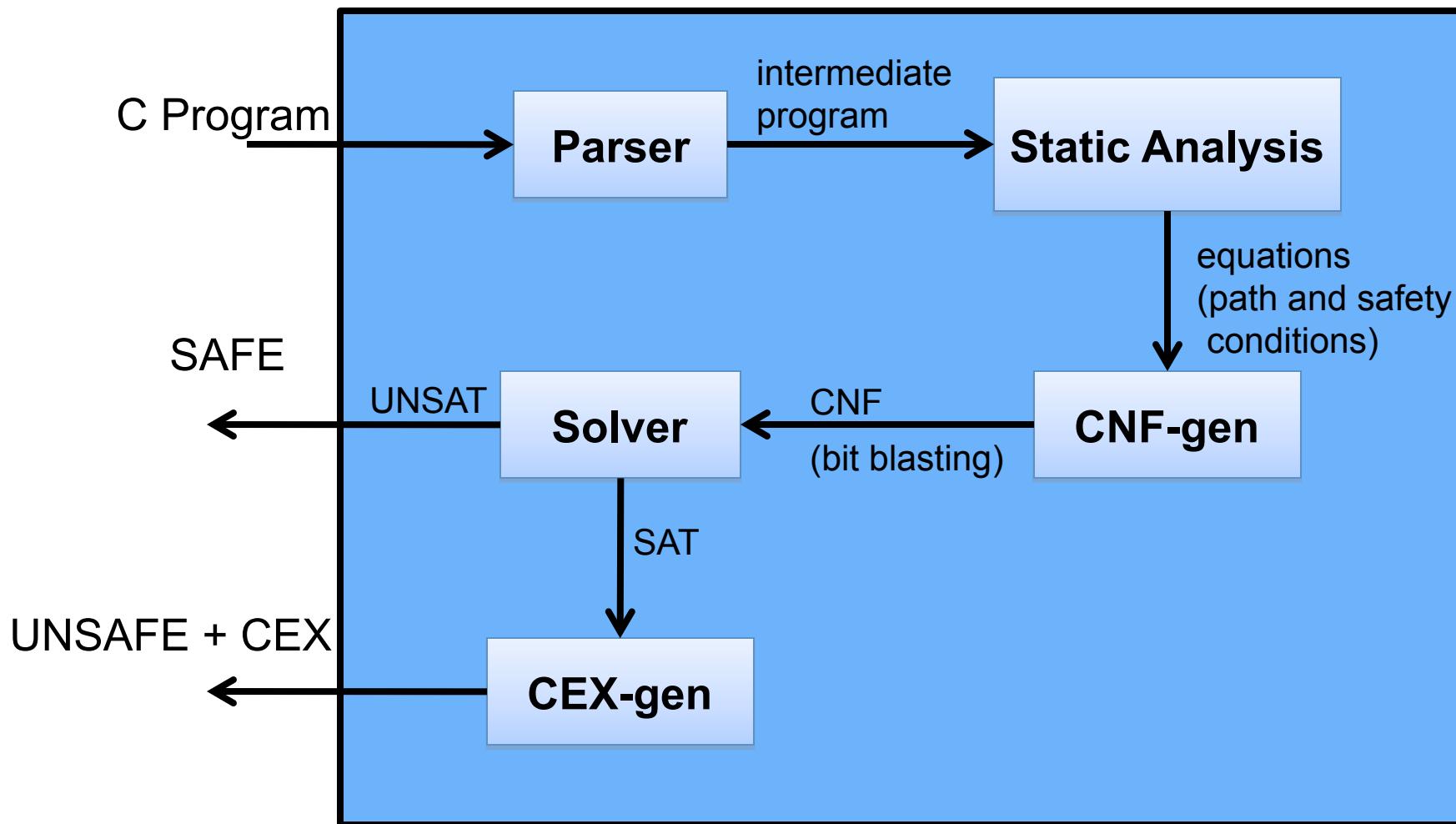
SAT/SMT-based BMC tools for C

Typical features:

- full language support
 - bit-precise operations, structs, arrays, ...
 - heap-allocated memory
 - concurrency
- built-in safety checks
 - overflow, div-by-zero, array out-of-bounds indexing, ...
 - memory safety: nil pointer deref, memory leaks, ...
 - deadlocks, race conditions
- user-specified assertions and error labels
- non-deterministic modelling
 - nondeterministic assignments
 - assume-statements

SAT/SMT-based BMC tools for C

High-level architecture:



SAT/SMT-based BMC tools for C

General approach:

1. Simplify control flow
2. Unwind all of the loops
3. Convert into single static assignment (SSA) form
4. Convert into equations and simplify
5. (Bit-blast)
6. Solve with a SAT/SMT solver
7. Convert SAT assignment into a counterexample

Control flow simplifications

- remove all side effects
 - e.g., $j = ++i$; becomes $i = i + 1$; $j = i$;

Control flow simplifications

- remove all side effects
 - e.g., $j = ++i$; becomes $i = i + 1$; $j = i$;
- simplify all control flow structures into core forms
 - e.g., replace for, do while by while
 - e.g., replace case by if

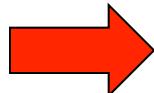
Control flow simplifications

- remove all side effects
 - e.g., $j = ++i$; becomes $i = i + 1$; $j = i$;
- simplify all control flow structures into core forms
 - e.g., replace for, do while by while
 - e.g., replace case by if
- make control flow explicit
 - e.g., replace continue, break by goto
 - e.g., replace if, while by goto

Control flow simplifications

Demo: esbmc --goto-functions-only example-1.c

```
int main() {  
    int i,j;  
    for(i=0; i<6; i++) {  
        j=i;  
    }  
    assert(j==i);  
    return j;  
}
```



```
main (c::main):  
    int i;  
    int j;  
    i = 0;  
    1: IF !(i < 6) THEN GOTO 2  
        j = i;  
        i = i + 1;  
        GOTO 1  
    2: ASSERT j == i  
        RETURN: j  
        END_FUNCTION
```

Loop unwinding

- all loops are “unwound”, i.e., replaced by several guarded copies of the loop body
 - same for backward **gotos** and recursive functions
 - can use different unwinding bounds for different loops

⇒ each statement is executed at most once

Loop unwinding

- all loops are “unwound”, i.e., replaced by several guarded copies of the loop body
 - same for backward **gotos** and recursive functions
 - can use different unwinding bounds for different loops

⇒ each statement is executed at most once

- to check whether unwinding is sufficient special “unwinding assertion” claims are added

⇒ if a program satisfies all of its claims and all unwinding assertions then it is correct!

Loop unwinding

```
void f(...) {  
    ...  
    while(cond) {  
        Body;  
    }  
    Remainder;  
}
```

Loop unwinding

```
void f(...) {  
    ...  
    if(cond) {  
        Body;  
        while(cond) {  
            Body;  
        }  
    }  
    Remainder;  
}
```

unwind one iteration

Loop unwinding

```
void f(...) {  
    ...  
    if(cond) {  
        Body;  
        if(cond) {  
            Body;  
            while(cond) {  
                Body;  
            }  
        }  
    }  
    Remainder;  
}
```

unwind one
iteration

unwind one
iteration

Loop unwinding

```
void f(...) {  
    ...  
    if(cond) {  
        Body;  
        if(cond) {  
            Body;  
            if(cond) {  
                Body;  
                while(cond) {  
                    Body;  
                }  
            }  
        }  
    }  
    Remainder;  
}
```

unwind one
iteration...

unwind one
iteration...

unwind one
iteration...

Loop unwinding

```
void f(...) {  
    ...  
    if(cond) {  
        Body;  
        if(cond) {  
            Body;  
            if(cond) {  
                Body;  
                assert(!cond);  
            }  
        }  
    }  
    Remainder;  
}
```

unwind one
iteration

unwind one
iteration

unwind one
iteration...

unwinding
assertion

- unwinding assertion
 - inserted after last unwound iteration
 - violated if program runs longer than bound permits
- ⇒ if not violated: (real) correctness result!

Loop unwinding

```
void f(...) {  
    ...  
    for(i=0; i<N; i++) {  
        ...  
        b[i]=a[i];  
        ...  
    };  
    ...  
    for(i=0; i<N; i++) {  
        ...  
        assert(b[i]-a[i]>0);  
        ...  
    };  
    ...  
    Remainder;  
}
```

- unwinding assertion
 - inserted after last unwound iteration
 - violated if program runs longer than bound permits
 - ⇒ if not violated: (real) correctness result!
- ⇒ what about multiple loops?
 - use --partial-loops to suppress insertion
 - ⇒ unsound

Safety conditions

- Built-in safety checks converted into explicit assertions:
e.g., array safety:

`a[i]=...;`

$\Rightarrow \text{assert}(0 \leq i \ \&\& \ i \leq N); \ a[i]=...;$

Safety conditions

- Built-in safety checks converted into explicit assertions:
e.g., array safety:

$a[i] = \dots;$

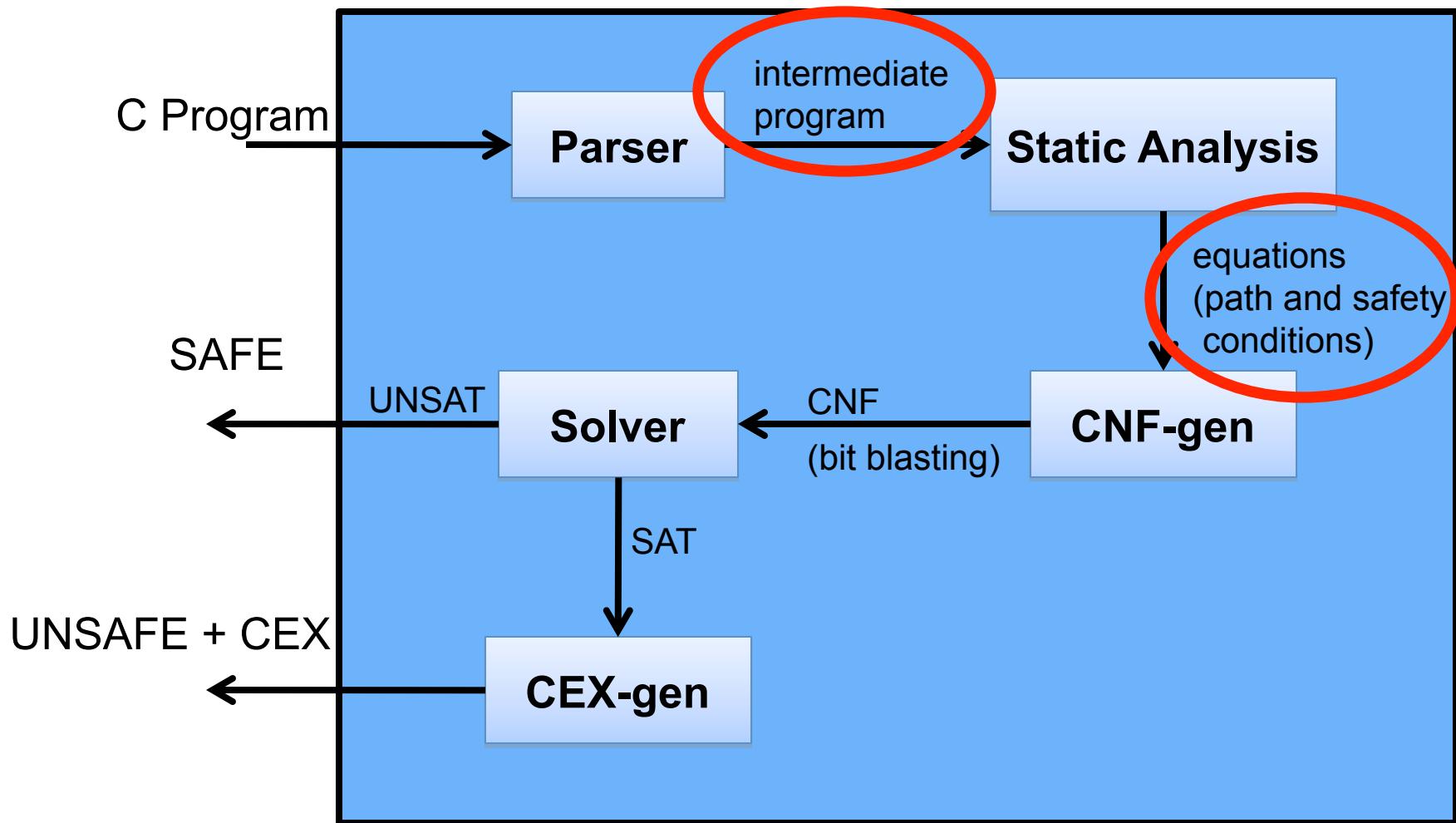
$\Rightarrow \text{assert}(0 \leq i \ \&\& \ i \leq N); \ a[i] = \dots;$

\Rightarrow sometimes easier at intermediate representation or formula level

e.g., word-aligned pointer access, overflow, ...

SAT/SMT-based BMC tools for C

High-level architecture:



Transforming straight-line programs into equations

- simple if each variable is assigned only once:

```
x = a;  
y = x + 1;  
z = y - 1;
```

program



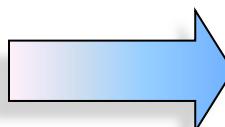
```
x = a      &&  
y = x + 1  &&  
z = y - 1
```

constraints

- still simple if variables are assigned multiple times:

```
x = a;  
x = x + 1;  
x = x - 1;
```

program



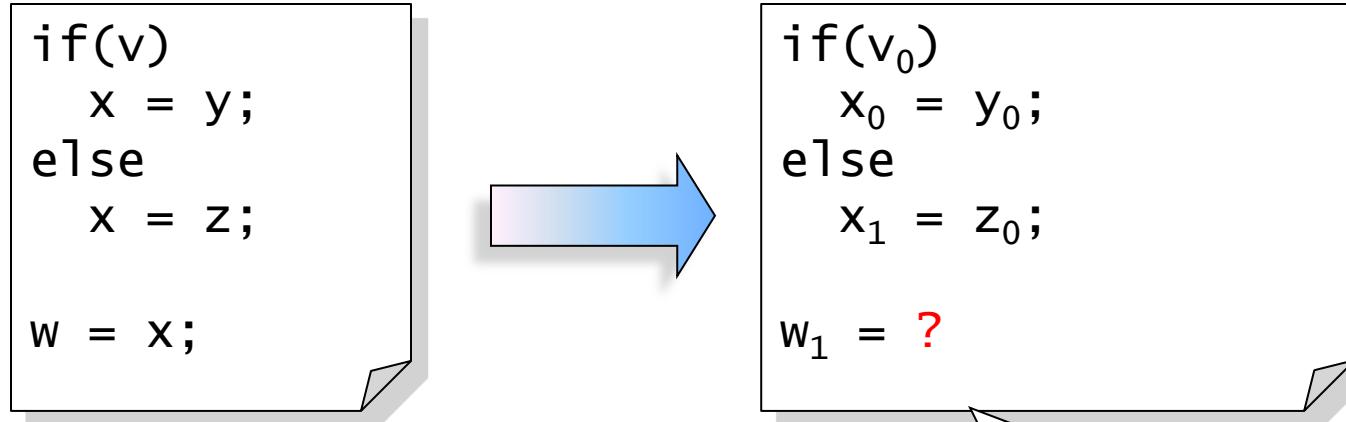
```
x0 = a;  
x1 = x0 + 1;  
x2 = x1 - 1;
```

program in SSA-form

introduce fresh copy for each occurrence
(*static single assignment (SSA) form*)

Transforming loop-free programs into equations

But what about control flow branches (if-statements)?

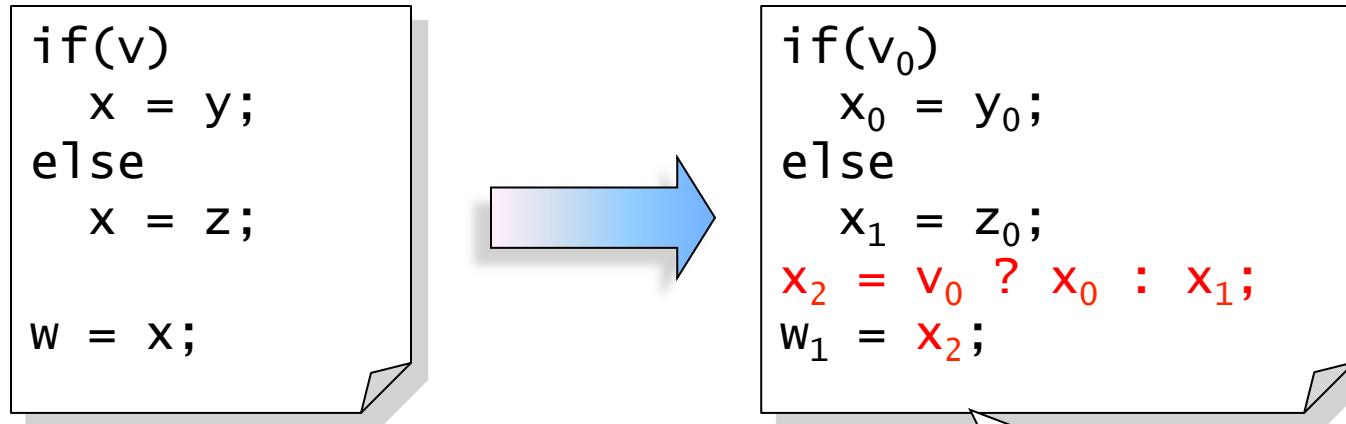


introduce & use
new variable

- for each control flow join point, add a new variable with guarded assignment as definition
 - also called ϕ -function

Transforming loop-free programs into equations

But what about control flow branches (if-statements)?



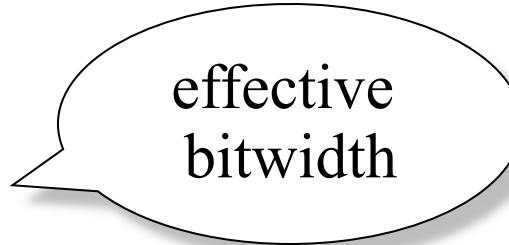
- for each control flow join point, add a new variable with guarded assignment as definition
 - also called ϕ -function

Bit-blasting

Conversion of equations into SAT problem:

- simple assignments:

$$| [x = y] | \triangleq \bigwedge_i x_i \Leftrightarrow y_i$$



effective
bitwidth

⇒ static analysis must approximate effective bitwidth well

- ϕ -functions:

$$| [x = v ? y : z] | \triangleq (v \Rightarrow | [x = y] |) \wedge (\neg v \Rightarrow | [x = z] |)$$

- Boolean operations:

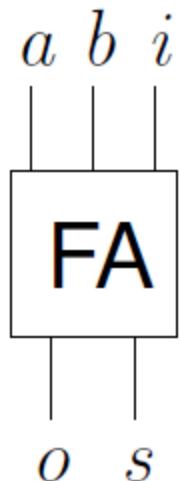
$$| [x = y \mid z] | \triangleq \bigwedge_i x_i \Leftrightarrow (y_i \vee z_i)$$

Exercise: relational operations

Bit-blasting arithmetic operations

Build **circuits** that implement the operations!

1-bit addition:



Full Adder

$$s \equiv (a + b + i) \bmod 2 \equiv a \oplus b \oplus i$$

$$o \equiv (a + b + i) \text{ div } 2 \equiv a \cdot b + a \cdot i + b \cdot i$$

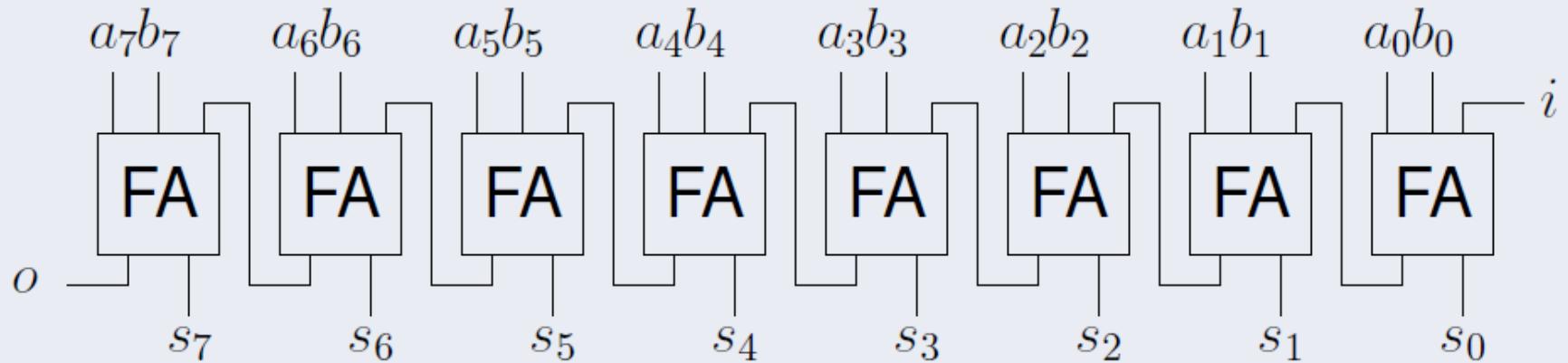
Full adder as CNF:

$$(a \vee b \vee \neg o) \wedge (a \vee \neg b \vee i \vee \neg o) \wedge (a \vee \neg b \vee \neg i \vee o) \wedge \\ (\neg a \vee b \vee i \vee \neg o) \wedge (\neg a \vee b \vee \neg i \vee o) \wedge (\neg a \vee \neg b \vee o)$$

Bit-blasting arithmetic operations

Build **circuits** that implement the operations!

8-Bit ripple carry adder (RCA)



- ⇒ adds w variables, 6^*w clauses
- ⇒ multiplication / division much more complicated

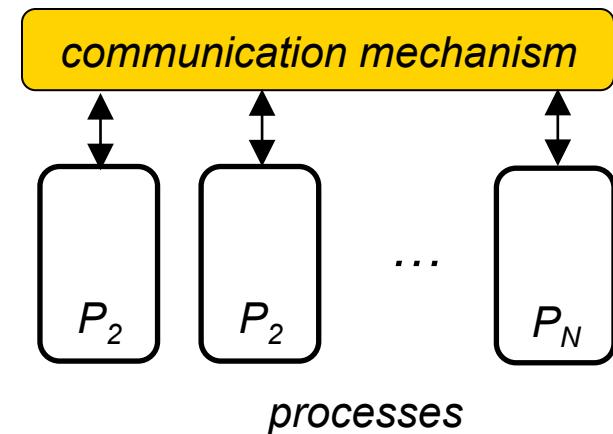
Intended learning outcomes

- Introduce typical BMC architectures for verifying software systems
- Explain **bounded model checking of multi-threaded software**
- Explain unbounded model checking of software

Concurrency verification

Writing concurrent programs is DIFFICULT

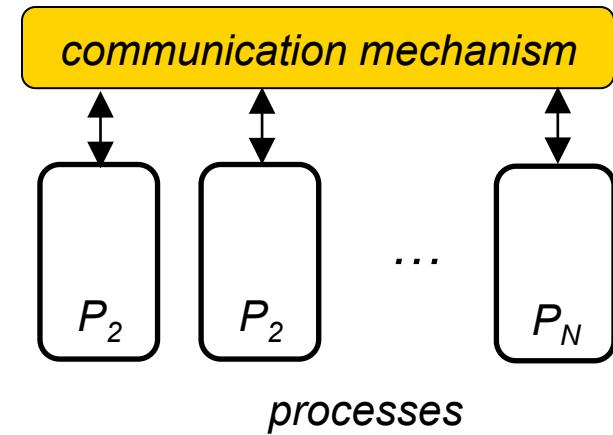
- programmers have to guarantee
 - correctness of sequential execution of each individual process
 - with nondeterministic interferences from other processes (schedules)



Concurrency verification

Writing concurrent programs is DIFFICULT

- programmers have to guarantee
 - correctness of sequential execution of each individual process
 - with nondeterministic interferences from other processes (schedules)
 - rare schedules result in errors that are difficult to find, reproduce, and repair
 - testers can spend weeks chasing a single bug
- ⇒ ***huge productivity problem***



Concurrency verification

What happens here...???

```
int n=0; //shared variable

void* P(void* arg) {
    int tmp, i=1;
    while (i<=10) {
        tmp = n;
        n = tmp + 1;
        i++;
    }
    return NULL;
}

int main (void) {
    pthread_t id1, id2;
    pthread_create(&id1, NULL, P, NULL);
    pthread_create(&id2, NULL, P, NULL);
    pthread_join(id1, NULL);
    pthread_join(id2, NULL);
    assert(n == 20);
}
```

Which values can n actually have?

Concurrency verification

What happens here...???

```
int n=0; //shared variable

void* P(void* arg) {
    int tmp, i=1;
    while (i<=10) {
        tmp = n;
        n = tmp + 1;
        i++;
    }
    return NULL;
}

int main (void) {
    pthread_t id1, id2;
    pthread_create(&id1, NULL, P, NULL);
    pthread_create(&id2, NULL, P, NULL);
    pthread_join(id1, NULL);
    pthread_join(id2, NULL);
    assert(n == 20);
}
```

```
$gcc example-2.c -o
example-2
$./example-2
$./example-2
$./example-2
$./example-2
$./example-2
$./example-2
Assertion failed: (n
== 20), function main,
file example-2.c, line
22.
```

Which values can n actually have?

Concurrency verification

What happens here...???

```
int n=0; //shared variable

void* P(void* arg) {
    int tmp, i=1;
    while (i<=10) {
        tmp = n;
        n = tmp + 1;
        i++;
    }
    return NULL;
}

int main (void) {
    pthread_t id1, id2;
    pthread_create(&id1, NULL, P, NULL);
    pthread_create(&id2, NULL, P, NULL);
    pthread_join(id1, NULL);
    pthread_join(id2, NULL);
    assert(n >= 10 && n <= 20);
}
```

Concurrency verification

What happens here...???

```
int n=0; //shared variable
pthread_mutex_t mutex;
void* P(void* arg) {
    int tmp, i=1;
    while (i<=10) {
        pthread_mutex_lock(&mutex);
        tmp = n;
        n = tmp + 1;
        pthread_mutex_unlock(&mutex);
        i++;
    }
    return NULL;
}
int main (void) {
    pthread_t id1, id2;
    pthread_mutex_init(&mutex, NULL);
    pthread_create(&id1, NULL, P, NULL);
    pthread_create(&id2, NULL, P, NULL);
    pthread_join(id1, NULL);
    pthread_join(id2, NULL);
    assert(n == 20);
}
```

Concurrency errors

There are two main kinds of concurrency errors:

- **progress** errors: deadlock, starvation, ...
 - typically caused by wrong **synchronization**
 - requires modeling of synchronization primitives
 - mutex locking / unlocking
 - requires modeling of (global) error condition

Concurrency errors

There are two main kinds of concurrency errors:

- **progress** errors: deadlock, starvation, ...
 - typically caused by wrong **synchronization**
 - requires modeling of synchronization primitives
 - mutex locking / unlocking
 - requires modeling of (global) error condition
- **safety** errors: assertion violation, ...
 - typically caused by **data races** (i.e., unsynchronized access to shared data)
 - requires modeling of synchronization primitives
 - can be checked locally

Concurrency errors

There are two main kinds of concurrency errors:

- **progress** errors: deadlock, starvation, ...
 - typically caused by wrong **synchronization**
 - requires modeling of synchronization primitives
 - mutex locking / unlocking
 - requires modeling of (global) error condition
 - **safety** errors: assertion violation, ...
 - typically caused by **data races** (i.e., unsynchronized access to shared data)
 - requires modeling of synchronization primitives
 - can be checked locally
- ⇒ focus here on safety errors

Shared memory concurrent programs

Concurrent programming styles:

- communication via **message passing**
 - “truly” parallel **distributed** systems
 - multiple computations advancing simultaneously

Shared memory concurrent programs

Concurrent programming styles:

- communication via **message passing**
 - “truly” parallel **distributed** systems
 - multiple computations advancing simultaneously
- communication via **shared memory**
 - **multi-threaded** programs
 - only one thread active at any given time (conceptually), but active thread can be changed at any given time
 - o active == uncontested access to shared memory
 - o can be single-core or multi-core

Shared memory concurrent programs

Concurrent programming styles:

- communication via **message passing**
 - “truly” parallel **distributed** systems
 - multiple computations advancing simultaneously
 - communication via **shared memory**
 - **multi-threaded** programs
 - only one thread active at any given time (conceptually), but active thread can be changed at any given time
 - o active == uncontested access to shared memory
 - o can be single-core or multi-core
- ⇒ focus here on **multi-threaded, shared memory** programs

Multi-threaded programs

- typical C-implementation: pthreads

Multi-threaded programs

- typical C-implementation: pthreads
- formed of individual sequential programs (**threads**)
 - can be created and destroyed on the fly
 - typically for BMC: assume upper bound
 - each possibly with loops and recursive function calls
 - each with **local variables**

Multi-threaded programs

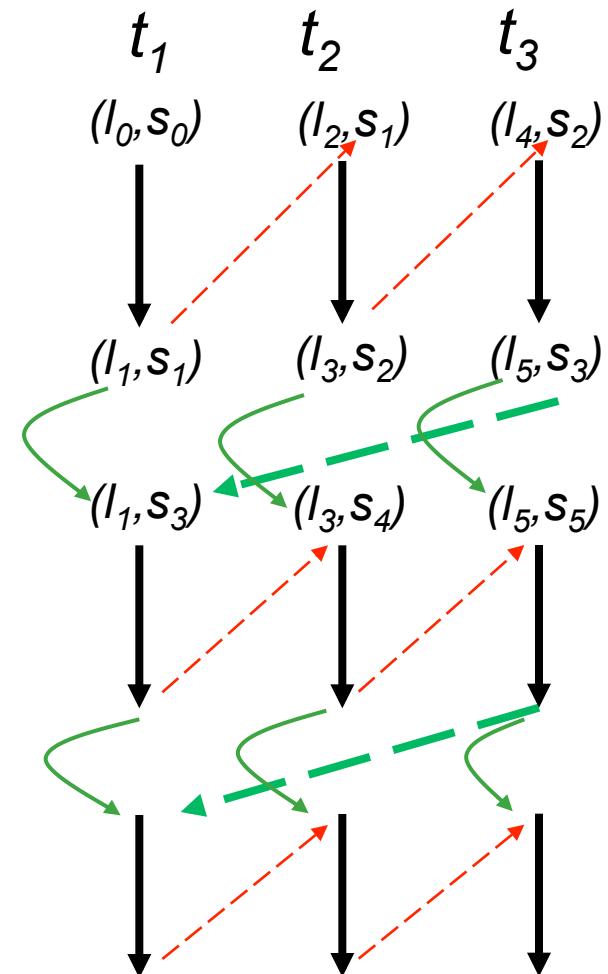
- typical C-implementation: pthreads
- formed of individual sequential programs (**threads**)
 - can be created and destroyed on the fly
 - typically for BMC: assume upper bound
 - each possibly with loops and recursive function calls
 - each with **local variables**
- each thread can read and write **shared variables**
 - assume **sequential consistency**: writes are immediately visible to all the other programs
 - weak memory models can be modeled

Multi-threaded programs

- typical C-implementation: pthreads
- formed of individual sequential programs (**threads**)
 - can be created and destroyed on the fly
 - typically for BMC: assume upper bound
 - each possibly with loops and recursive function calls
 - each with **local variables**
- each thread can read and write **shared variables**
 - assume **sequential consistency**: writes are immediately visible to all the other programs
 - weak memory models can be modeled
- execution is **interleaving** of thread executions
 - only valid for sequential consistency

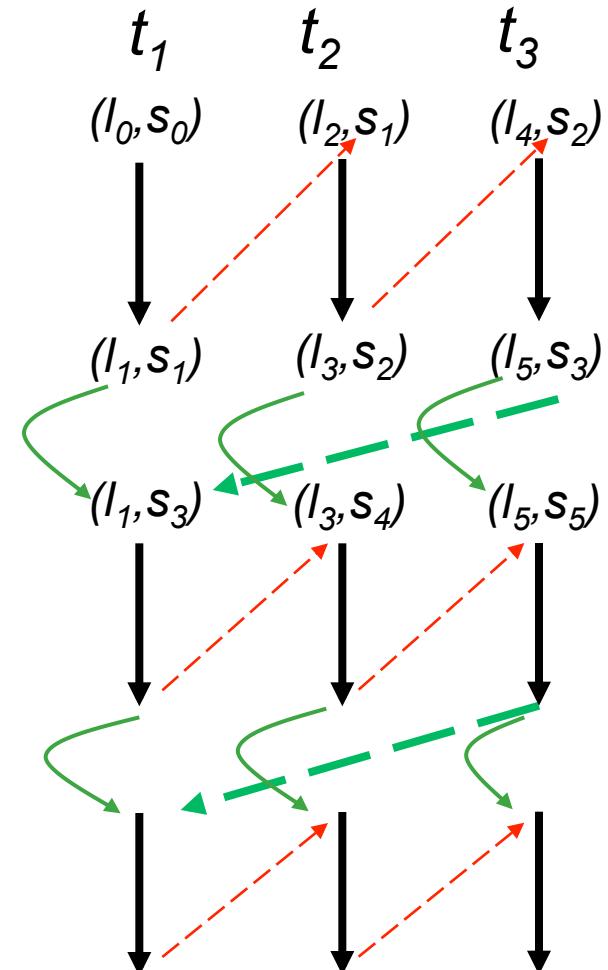
Round-robin scheduling

- **context**: segment of a run of an active thread t_i



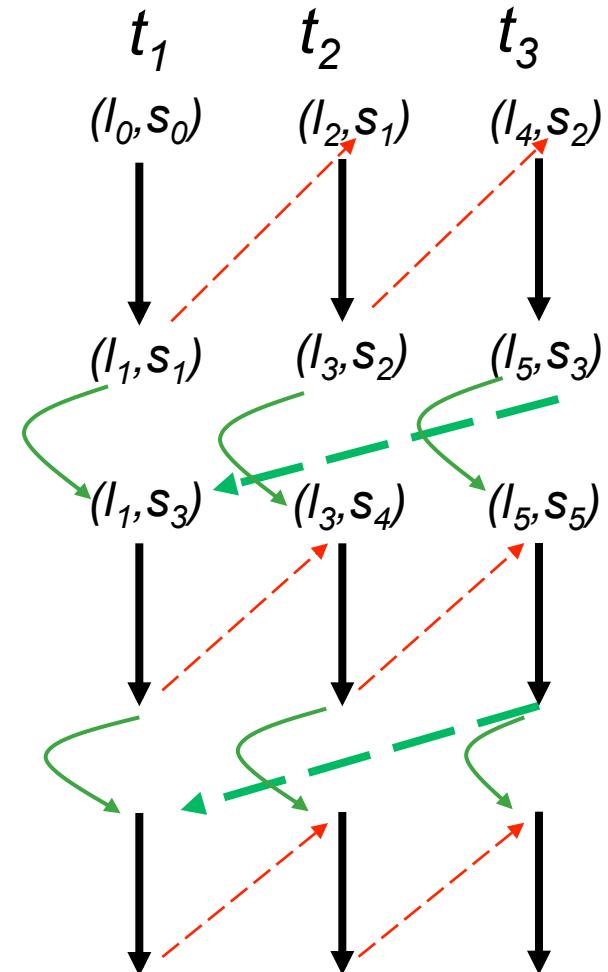
Round-robin scheduling

- **context**: segment of a run of an active thread t_i
- **context switch**: change of active thread from t_i to t_k
 - **global state** is passed on to t_k
 - context switch back to t_i resumes at old **local state** (incl. pc)



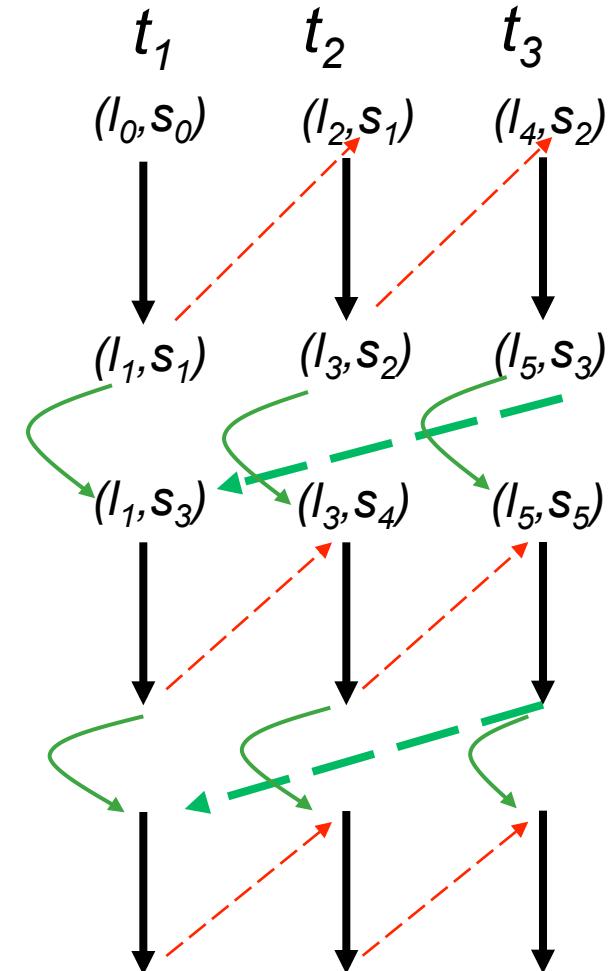
Round-robin scheduling

- **context**: segment of a run of an active thread t_i
- **context switch**: change of active thread from t_i to t_k
 - **global state** is passed on to t_k
 - context switch back to t_i resumes at old **local state** (incl. pc)
- **round**: formed of one context of each thread



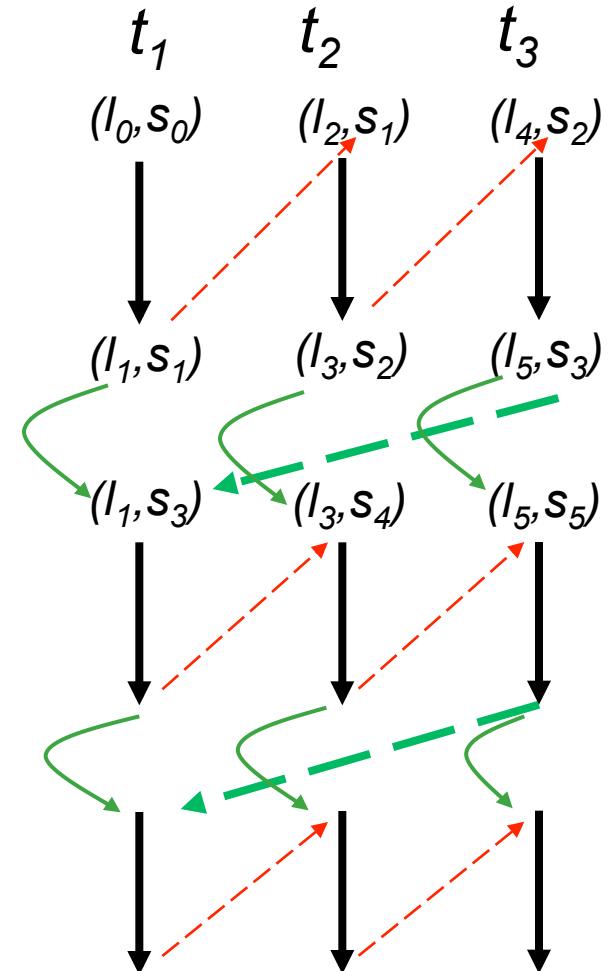
Round-robin scheduling

- **context**: segment of a run of an active thread t_i
- **context switch**: change of active thread from t_i to t_k
 - global state is passed on to t_k
 - context switch back to t_i resumes at old **local state** (incl. pc)
- **round**: formed of one context of each thread
- **round robin schedule**: same order of threads in each round



Round-robin scheduling

- **context**: segment of a run of an active thread t_i
- **context switch**: change of active thread from t_i to t_k
 - global state is passed on to t_k
 - context switch back to t_i resumes at old **local state** (incl. pc)
- **round**: formed of one context of each thread
- **round robin schedule**: same order of threads in each round
- can simulate all schedules by round robin schedules



Context-bounded analysis

Important observation:

Most concurrency errors are shallow!

i.e., require only few context switches

⇒ limit the search space by bounding the number of

- context switches
- rounds

Concurrency verification approaches

- Explicit schedule exploration (ESBMC)
 - lazy exploration
 - schedule recording

Concurrency verification approaches

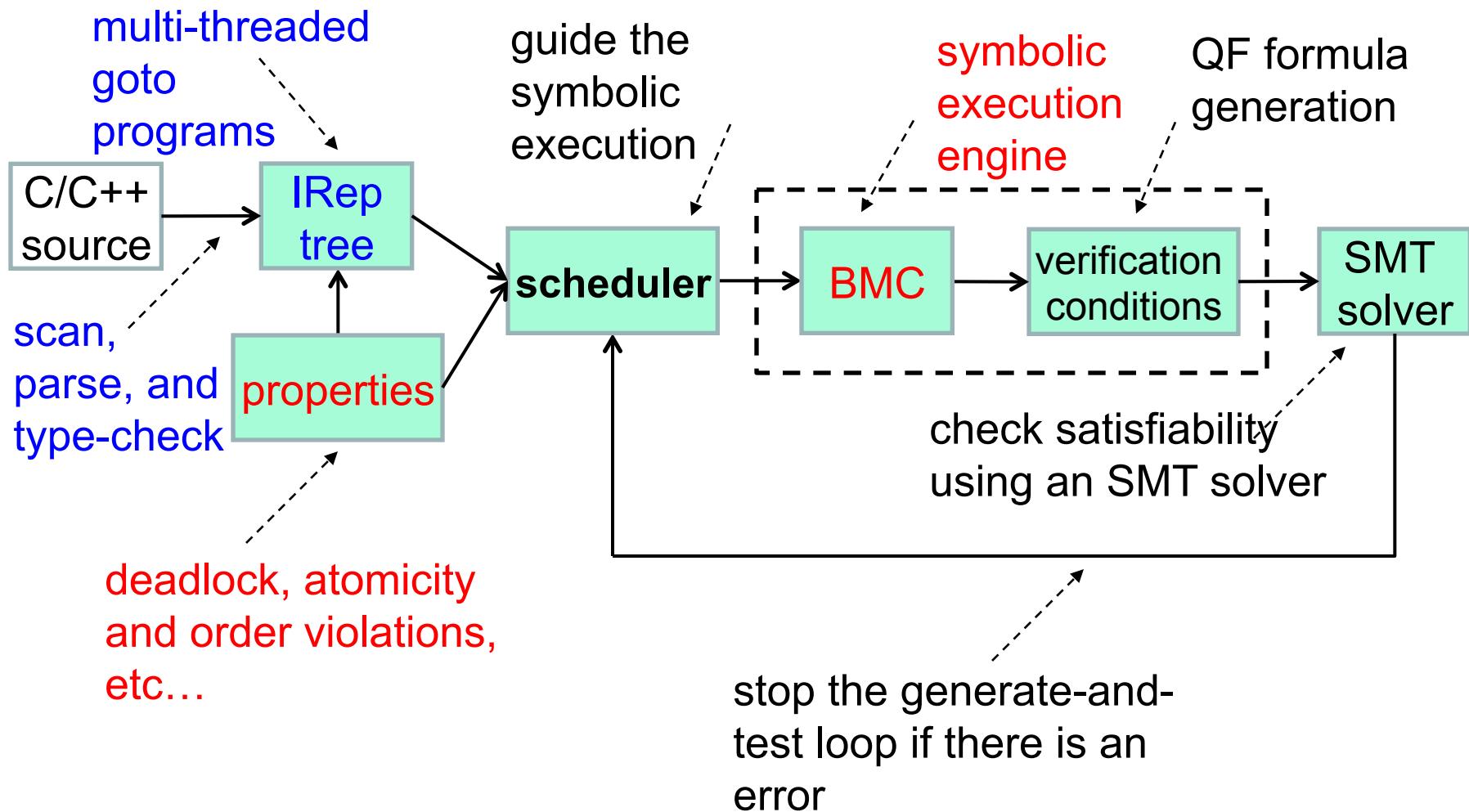
- Explicit schedule exploration (ESBMC)
 - lazy exploration
 - schedule recording
- Partial order methods (CBMC)

Concurrency verification approaches

- Explicit schedule exploration (ESBMC)
 - lazy exploration
 - schedule recording
- Partial order methods (CBMC)
- Sequentialization
 - KISS
 - Lal / Reps (eager sequentialization)
 - Lazy CSeq
 - memory unwinding

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_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 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: $val1=1; val2=0;$

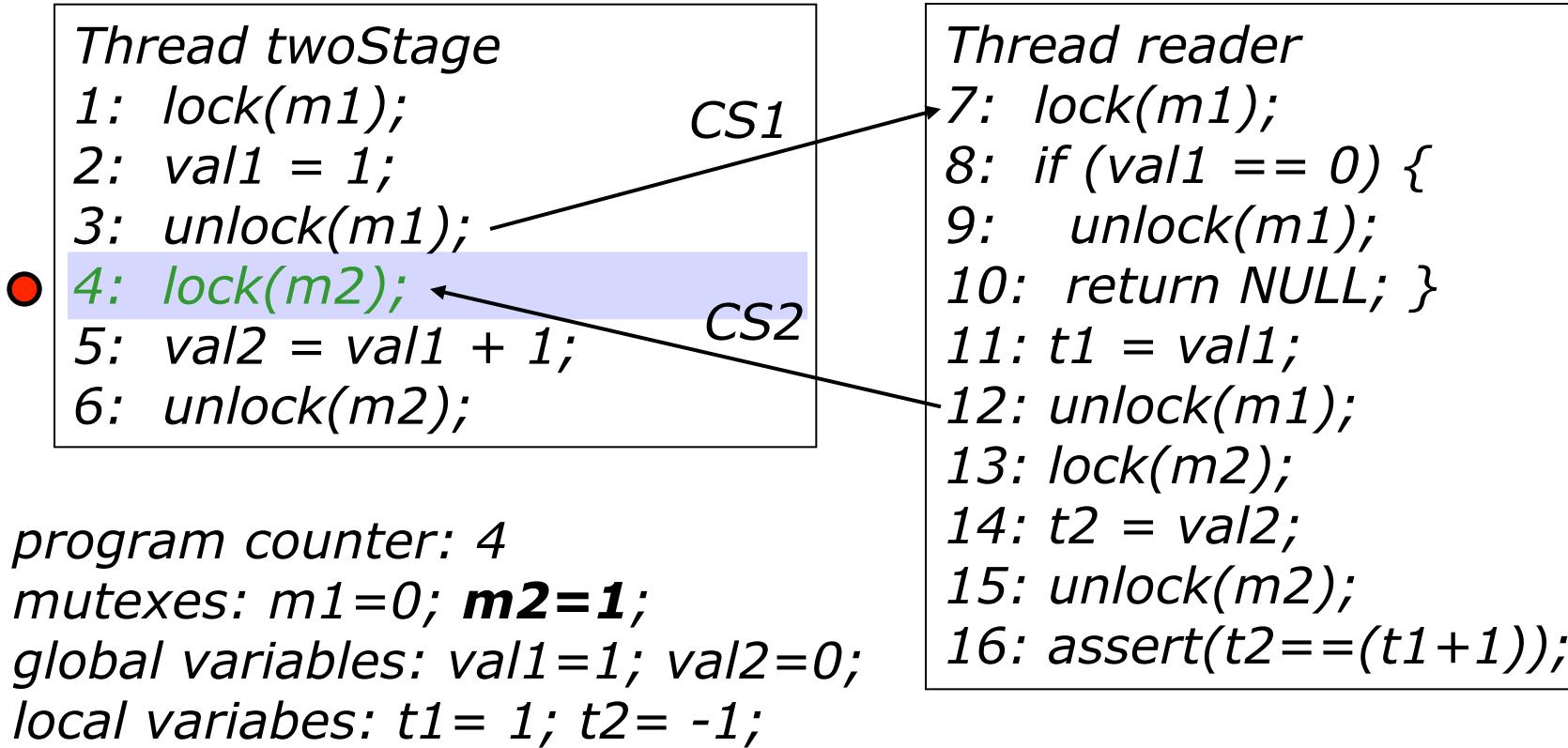
local variables: $t1= 1; t2= -1;$

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:

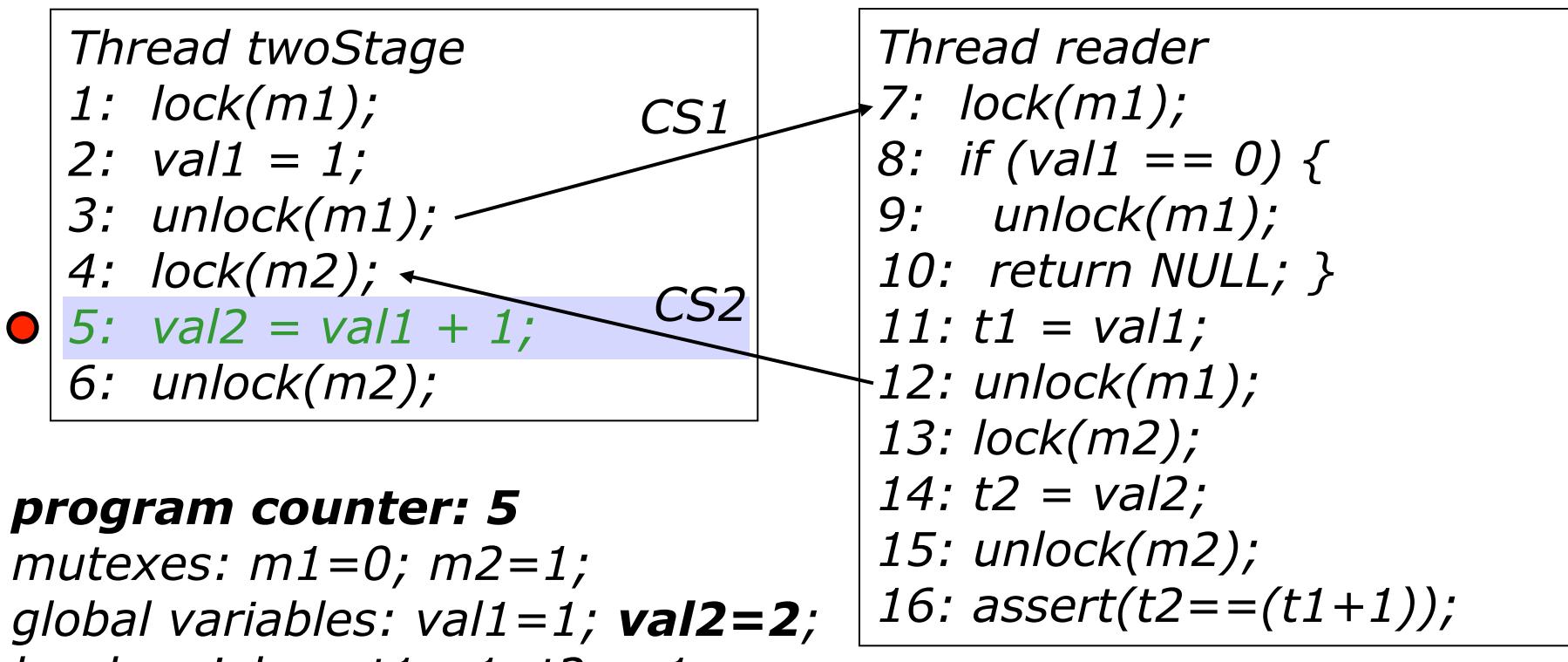


Lazy exploration: interleaving I_s

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: $W_{\text{twoStage},5}$

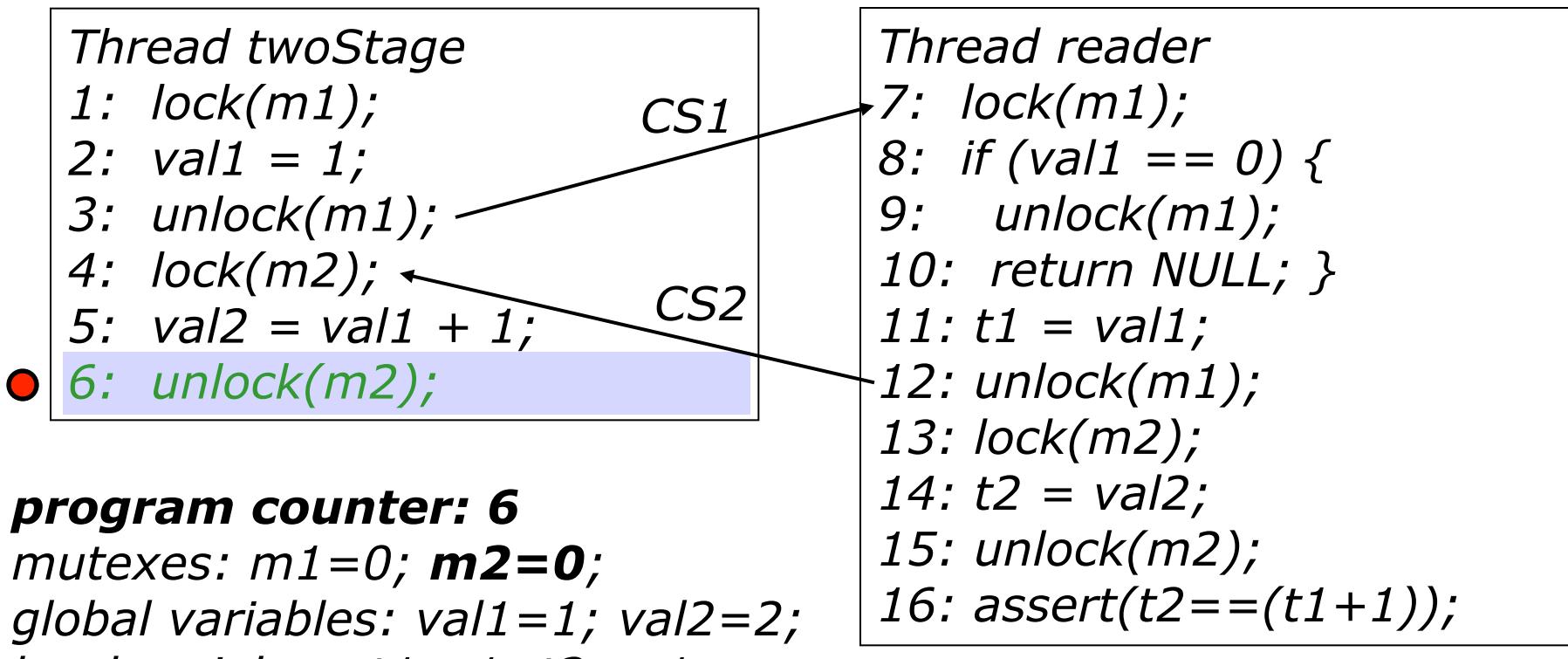


Lazy exploration: interleaving I_s

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: $val1=1; val2=2;$

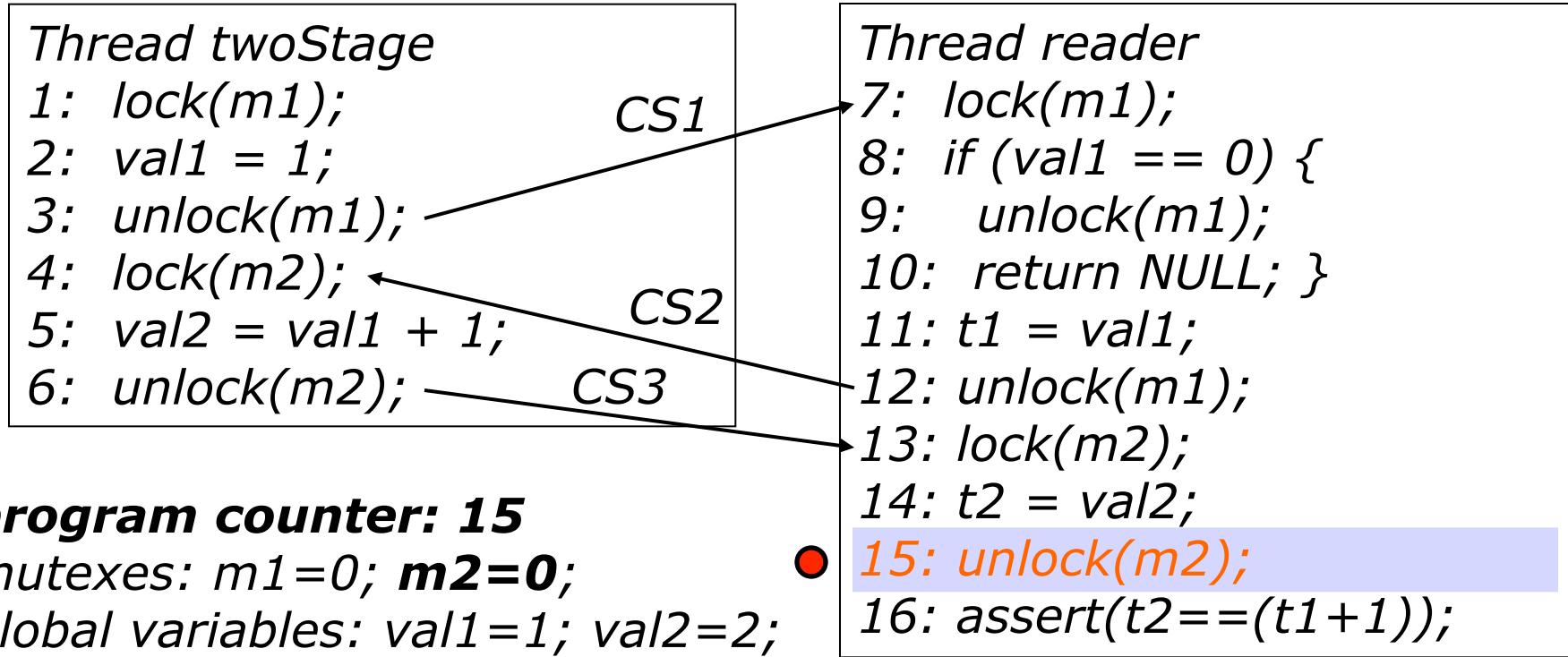
local variables: $t1= 1; t2= 2;$

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

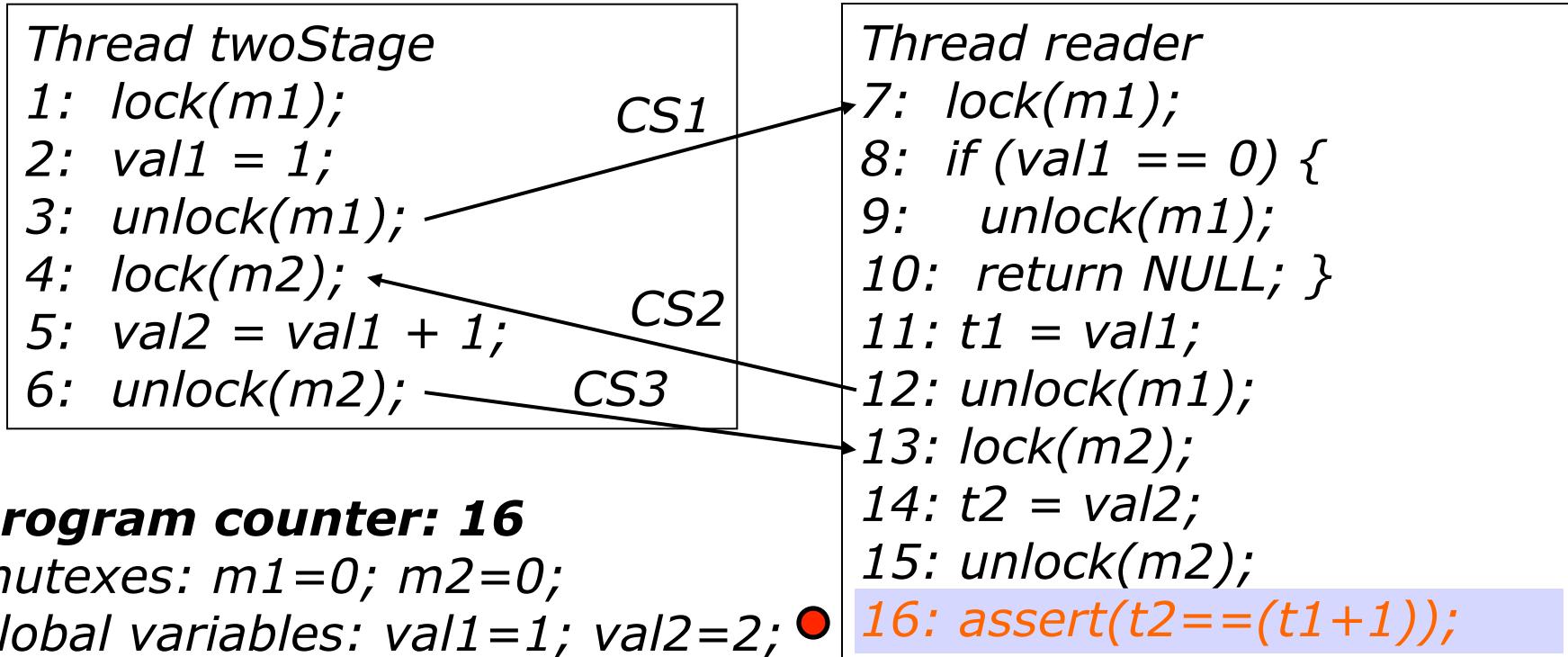


Lazy exploration: interleaving I_s

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

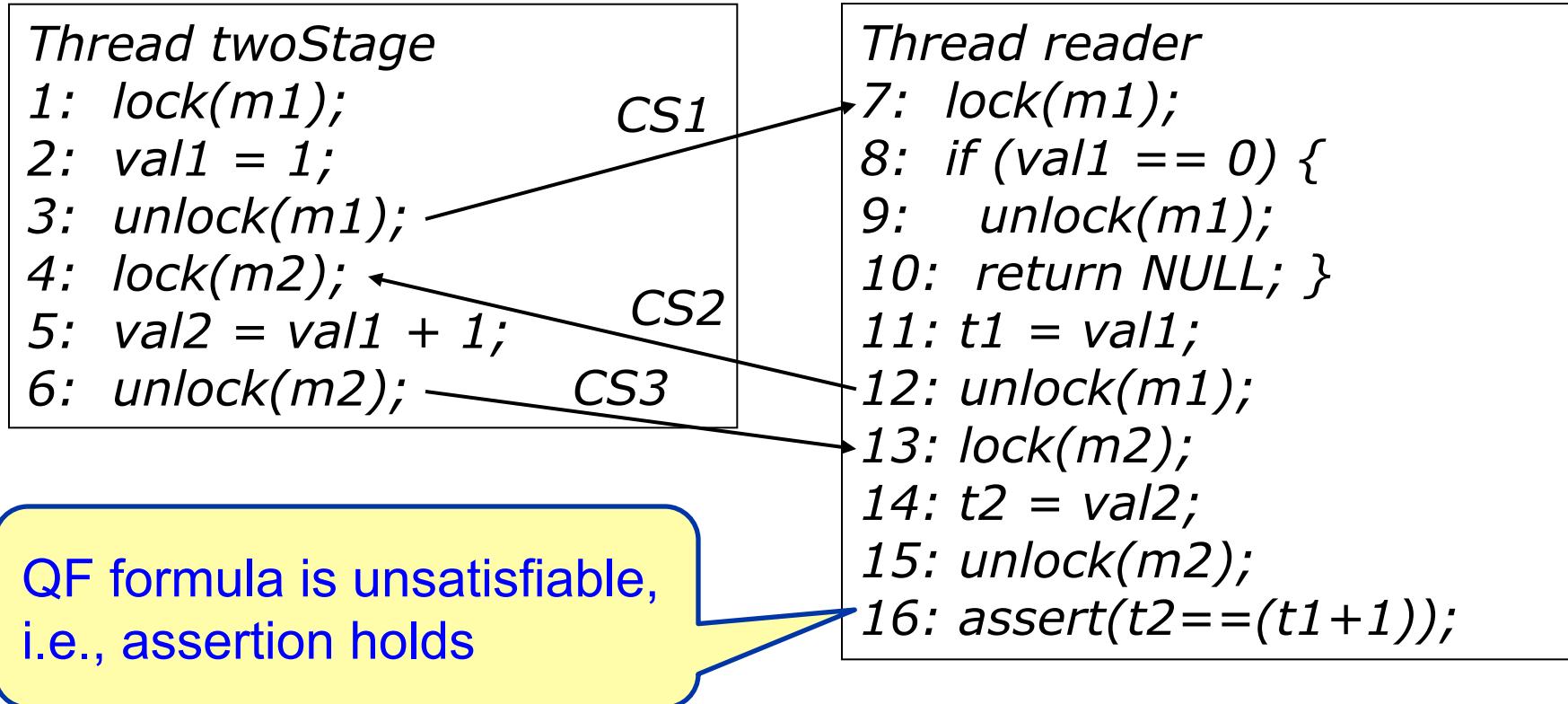


Lazy exploration: interleaving I_s

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 exploration of interleavings

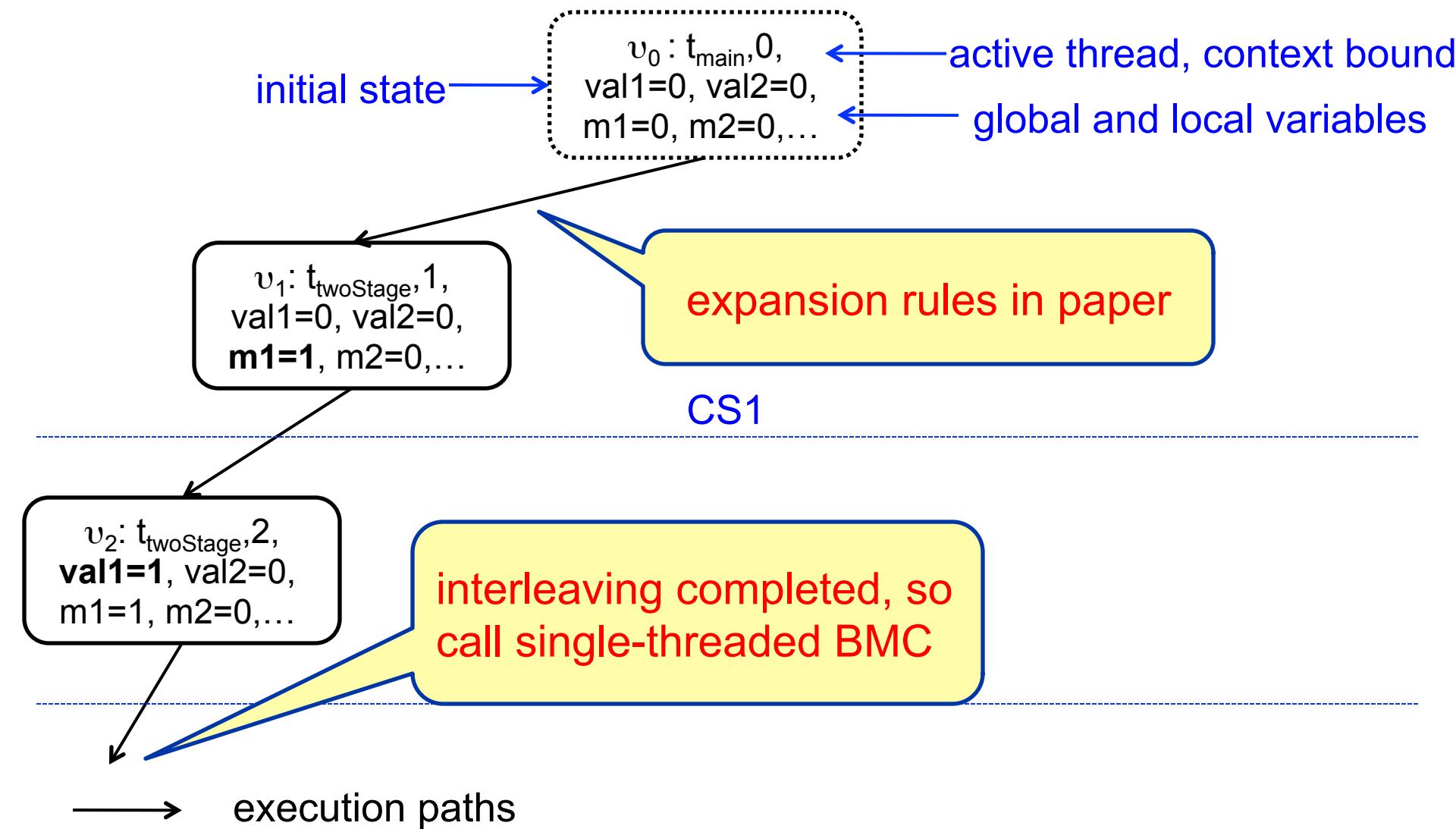
Idea: iteratively generate all possible interleavings and call the BMC procedure on each interleaving

... combines

- **symbolic** model checking: on each individual interleaving
- **explicit state** model checking: explore all interleavings

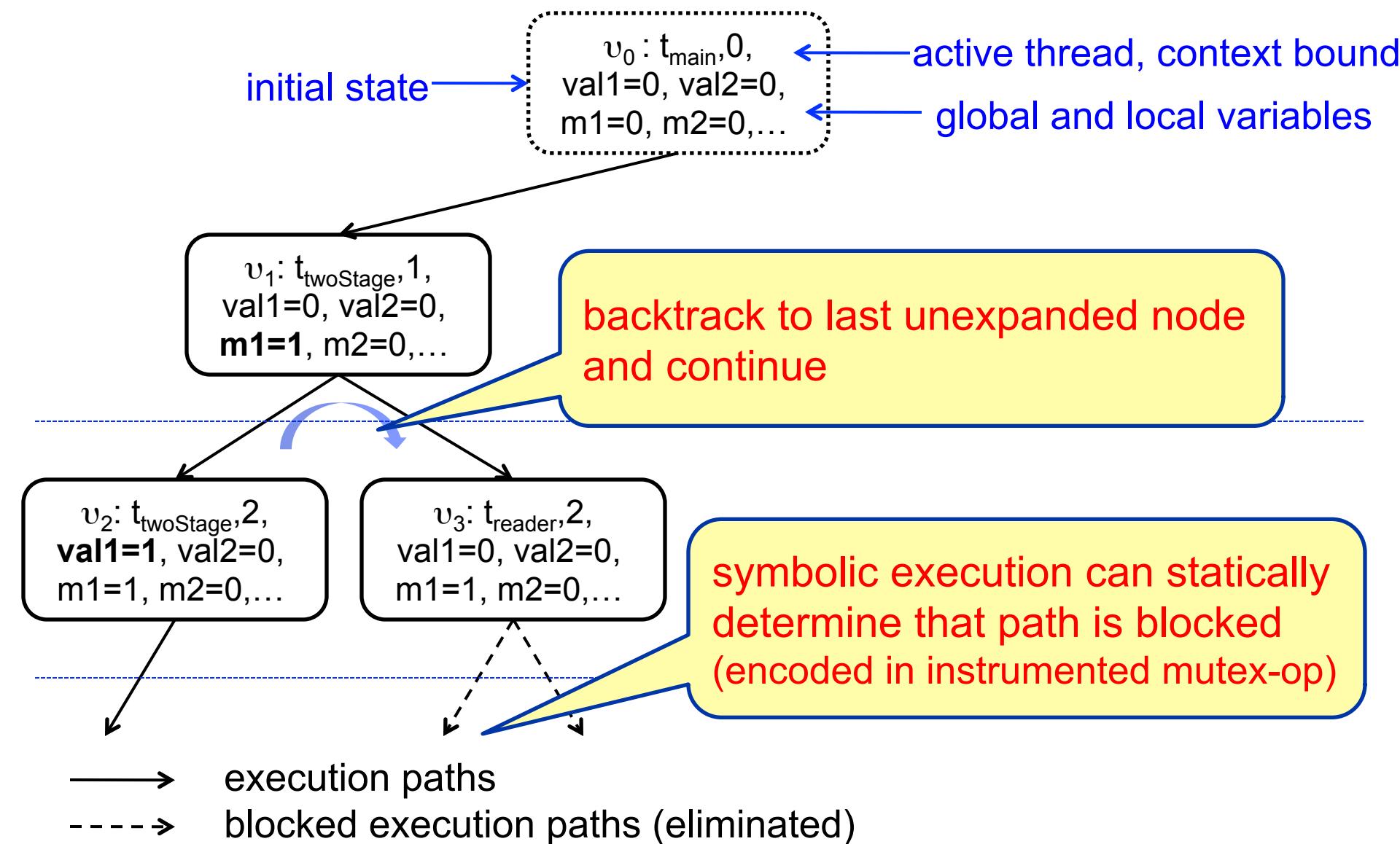
Lazy exploration of interleavings

– Reachability Tree



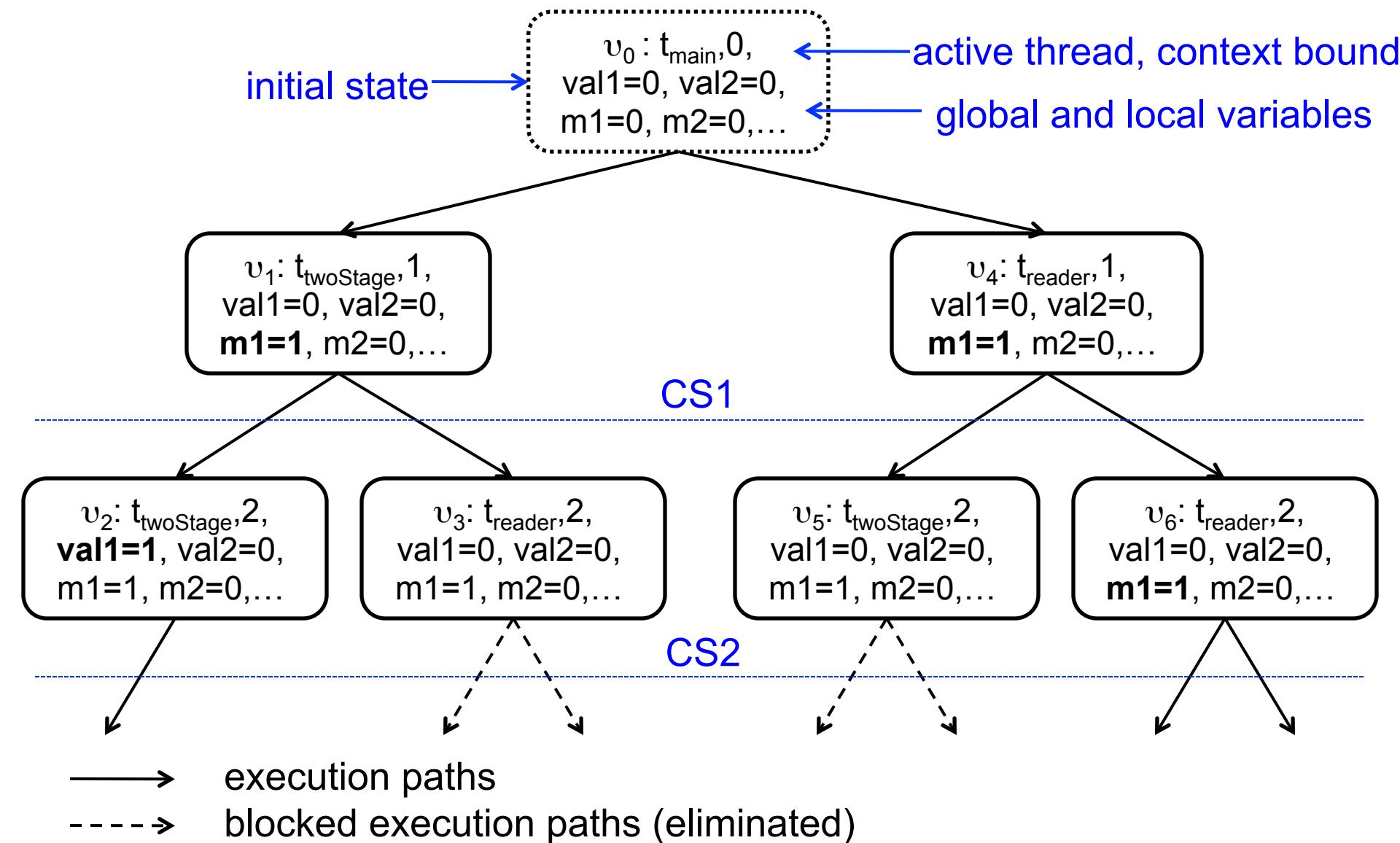
Lazy exploration of interleavings

– Reachability Tree



Lazy exploration of interleavings

– Reachability Tree



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 π off the stack and compute the set v' of successors of v using rules R1-R8.
4. If v' 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 v' is not empty, then for each node $v \in v'$, add v to π , and push node and extended path on the stack. goto step 3.

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

\nearrow computation path
 \searrow 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

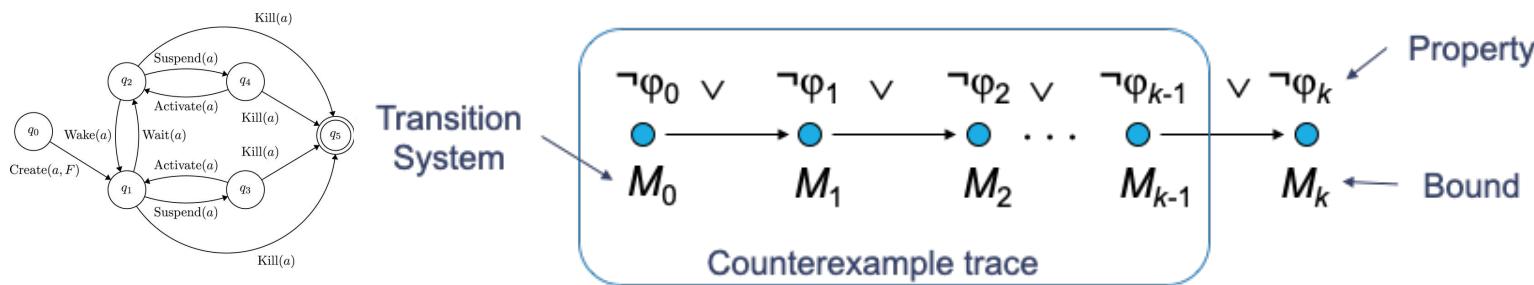
- naïve but useful:
 - bugs usually manifest with few context switches
[Qadeer&Rehof'05]
 - keep in memory the parent nodes of all unexplored paths only
 - exploit which transitions are enabled in a given state
 - bound the number of preemptions (C) allowed per threads
 - ▷ *number of executions:* $O(n^c)$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

- Introduce typical **BMC** architectures for verifying **software systems**
- Explain **bounded model checking** of multi-threaded 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

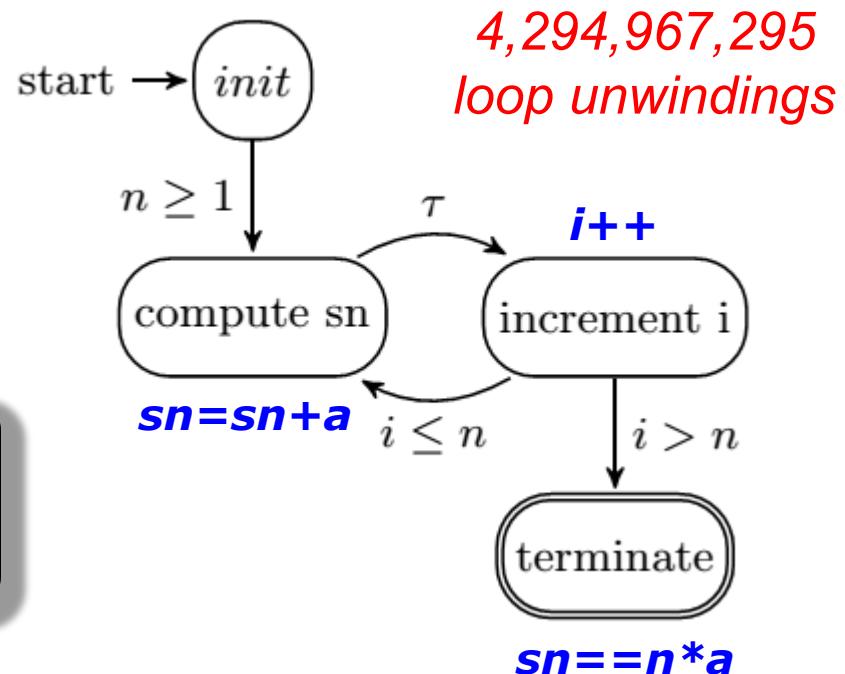
Difficulties in proving the correctness of programs with loops in BMC

- 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

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

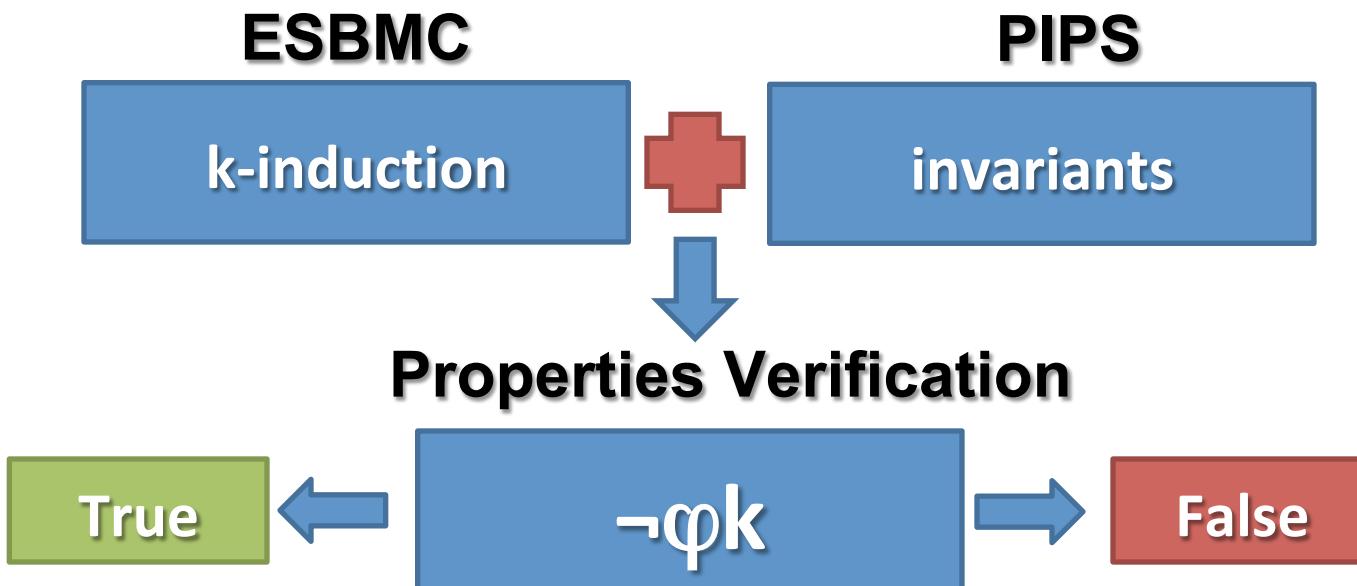


the loop will be unfolded 2^{n-1} times
(in the worst case, 2^{32-1} times on 32 bits integer)



Handling loops in BMC of C programs via k -induction and invariants

- Algorithmic method to prove correctness of C programs
 - combining k -induction with invariants
 - in a completely automatic way



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 $true$

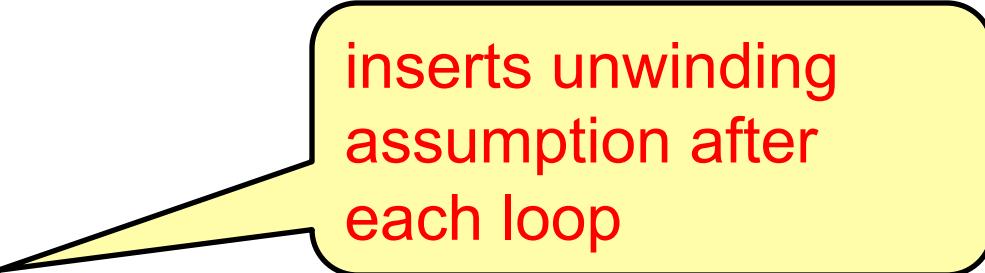
else if $step_k$ **then**

return $true$

end if

$k=k+1$

end



inserts unwinding
assumption after
each loop

The k -induction algorithm

$k = \text{initial bound}$

while true **do**

if base_k **then**

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

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

$k = \text{initial bound}$

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

$k = \text{initial bound}$

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

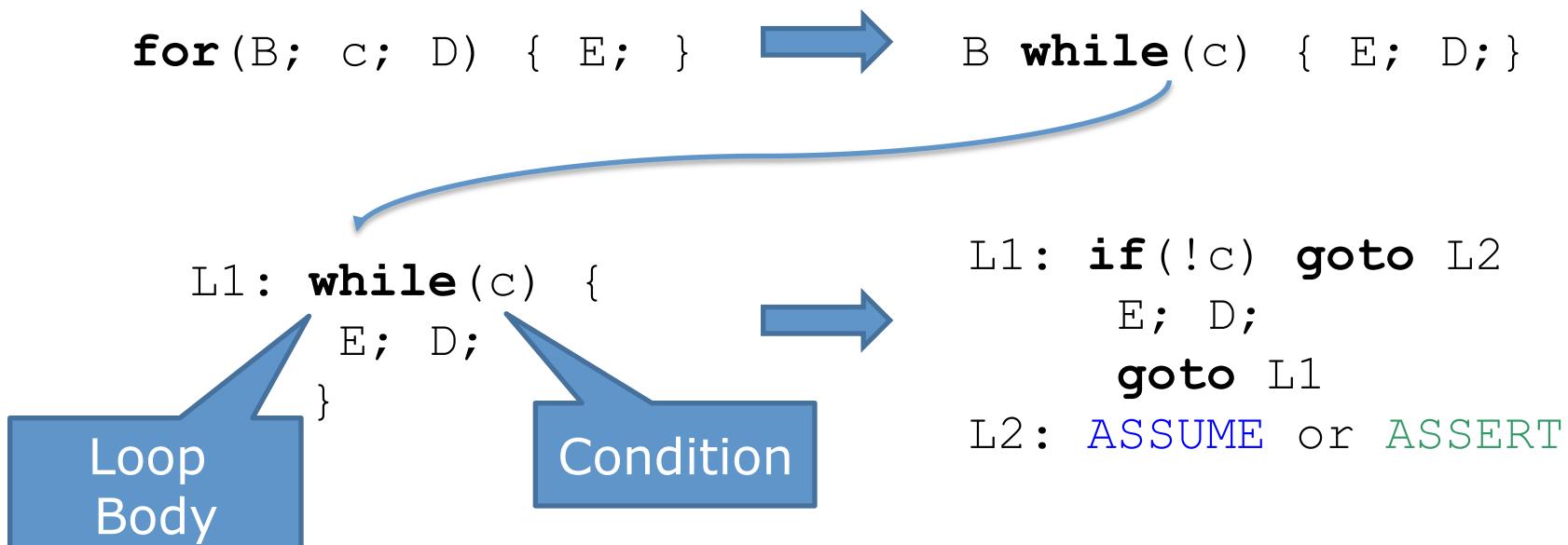
havoc variables that occur in the loop's termination condition

unable to falsify or prove the property

Loop-free Programs

(**base_k** and **fwd_k**)

- A loop-free program is represented by a **straight-line program** (without loops) using *if*-statements



Loop-free Programs (step_k)

In the inductive step, loops are converted into:

the code to
remove redundant
states

`while (c) { E; }`  A `while (c) { S; E; U; } R;`

- **A**: assigns **non-deterministic values** to all loops variables (the state is havoced before the loop)
- **c**: is the **halt condition** of the loop
- **S**: **stores the current state** of the program variables before executing the statements of **E**
- **E**: is the actual **code inside the loop**
- **U**: **updates all program variables** with local values after executing **E**

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*

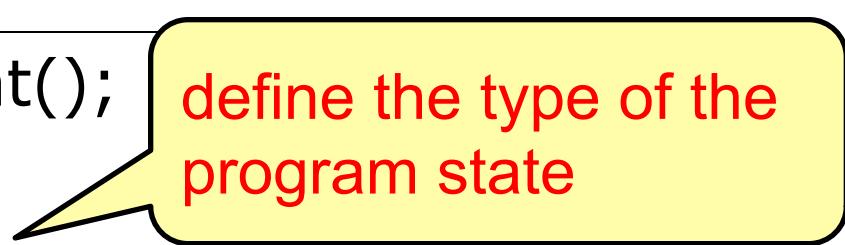
Havoc (only) the variables that occur in the loop's termination and branch conditions

```
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

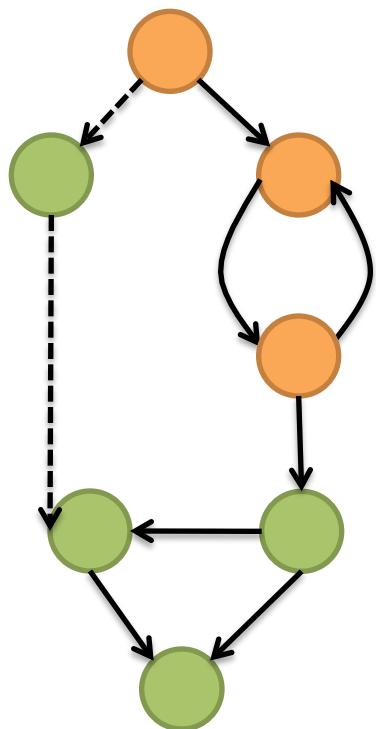
capture the state *cs*
after the iteration

constraints are
included by means
of assumptions

insert unwinding
assumption

Program Invariant

- Invariants are properties of program variables and relationships between these variables in a specific line of code (program point)



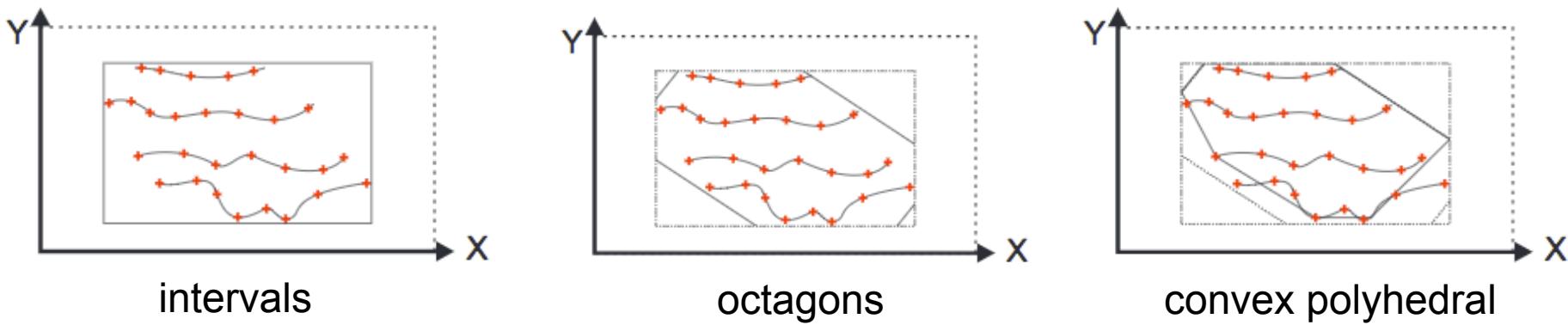
```
i := 0;  
s := 0;  
while i ≠ n{  
    s := s+b[i];  
    i := i+1;  
}
```

Annotations with orange arrows pointing to specific lines of code:

- An annotation **n = size(b)** is connected to the line `i := 0;`.
- An annotation **s = sum(b[0..i-1])** is connected to the line `s := s+b[i];`.
- An annotation **s = sum(b)** is connected to the line `}`.

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**