



**Systems and Software
Verification Laboratory**

MANCHESTER
1824

The University of Manchester

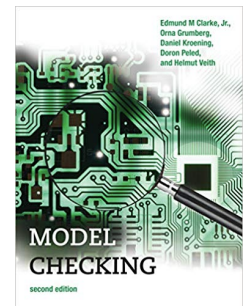
Detection of Software Vulnerabilities: Static Analysis (Part II)

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

Static Analysis (Part II)

- Lucas Cordeiro (Formal Methods Group)
 - lucas.cordeiro@manchester.ac.uk
 - Office: 2.28
 - Office hours: 15-16 Tuesday, 14-15 Wednesday
- References:
 - Clarke et al., *Model checking* (Chapter 14)
 - Cordeiro and Fischer: *Verifying multi-threaded software using smt-based context-bounded model checking*. ICSE 2011

*These slides are based on the lecture notes
“SAT/SMT-Based Bounded Model Checking of
Software” by Fischer, Parlato and La Torre*



Intended learning outcomes

- Introduce typical **BMC architectures** for verifying **software systems**
- Understand **communication models** and **typical errors** when writing **concurrent programs**
- Explain **explicit schedule exploration** of multi-threaded software
- Explain **sequentialization methods** to convert concurrent programs into sequential ones

Intended learning outcomes

- Introduce typical **BMC architectures** for verifying **software systems**
- Understand **communication models** and **typical errors** when writing **concurrent programs**
- Explain **explicit schedule exploration** of multi-threaded software
- Explain **sequentialization methods** to convert concurrent programs into sequential ones

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

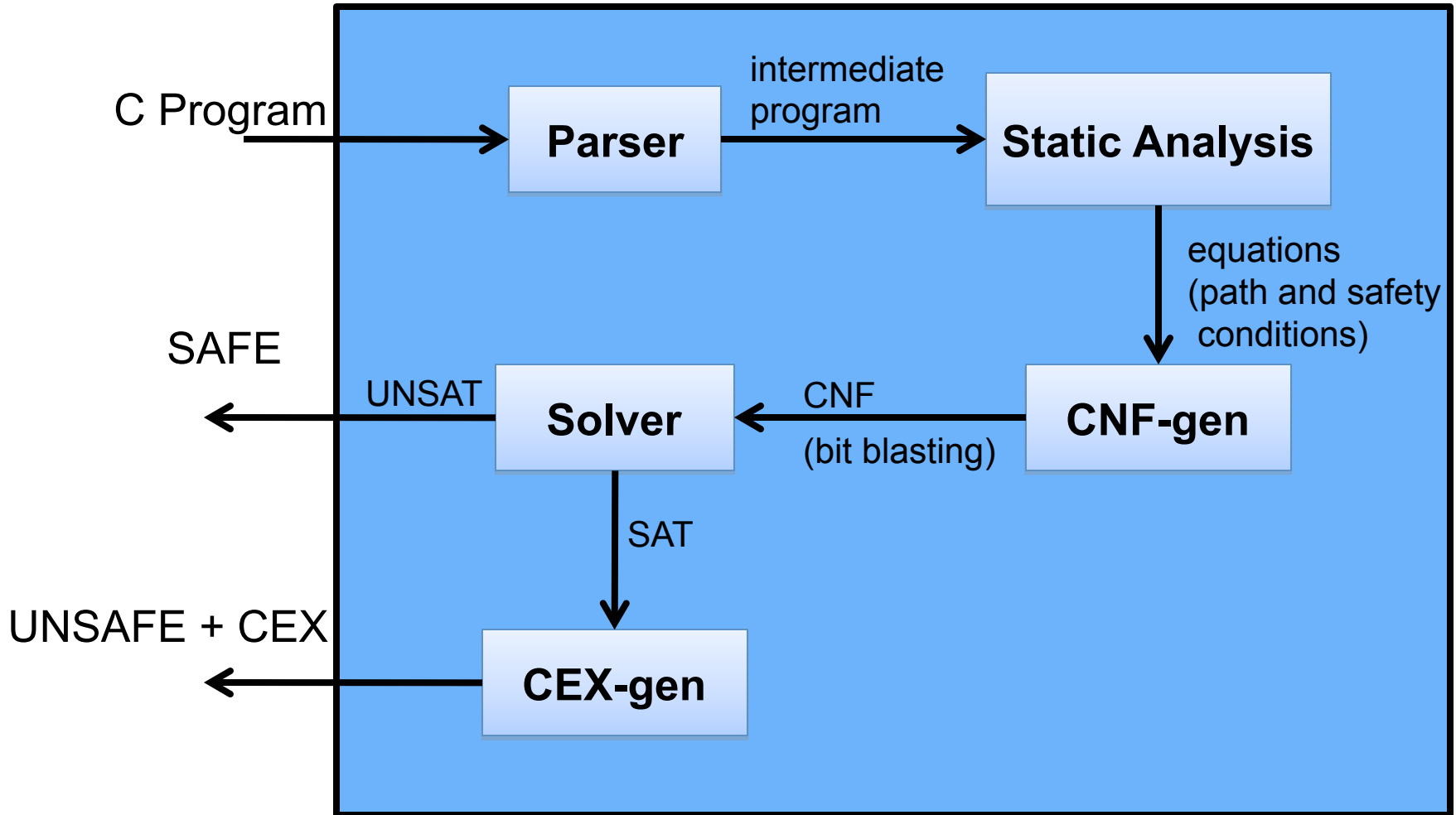
SAT/SMT-based BMC tools for C

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`

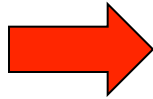
Control flow simplifications

- 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;  
    }  
}  
}
```

unwind one

unwind one

unwind one iteration...

Loop unwinding

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

unwind one

unwind one

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]=...;`

`⇒ assert(0 <= i && i <= N); a[i]=...;`

Safety conditions

- Built-in safety checks converted into explicit assertions:

e.g., array safety:

`a[i]=...;`

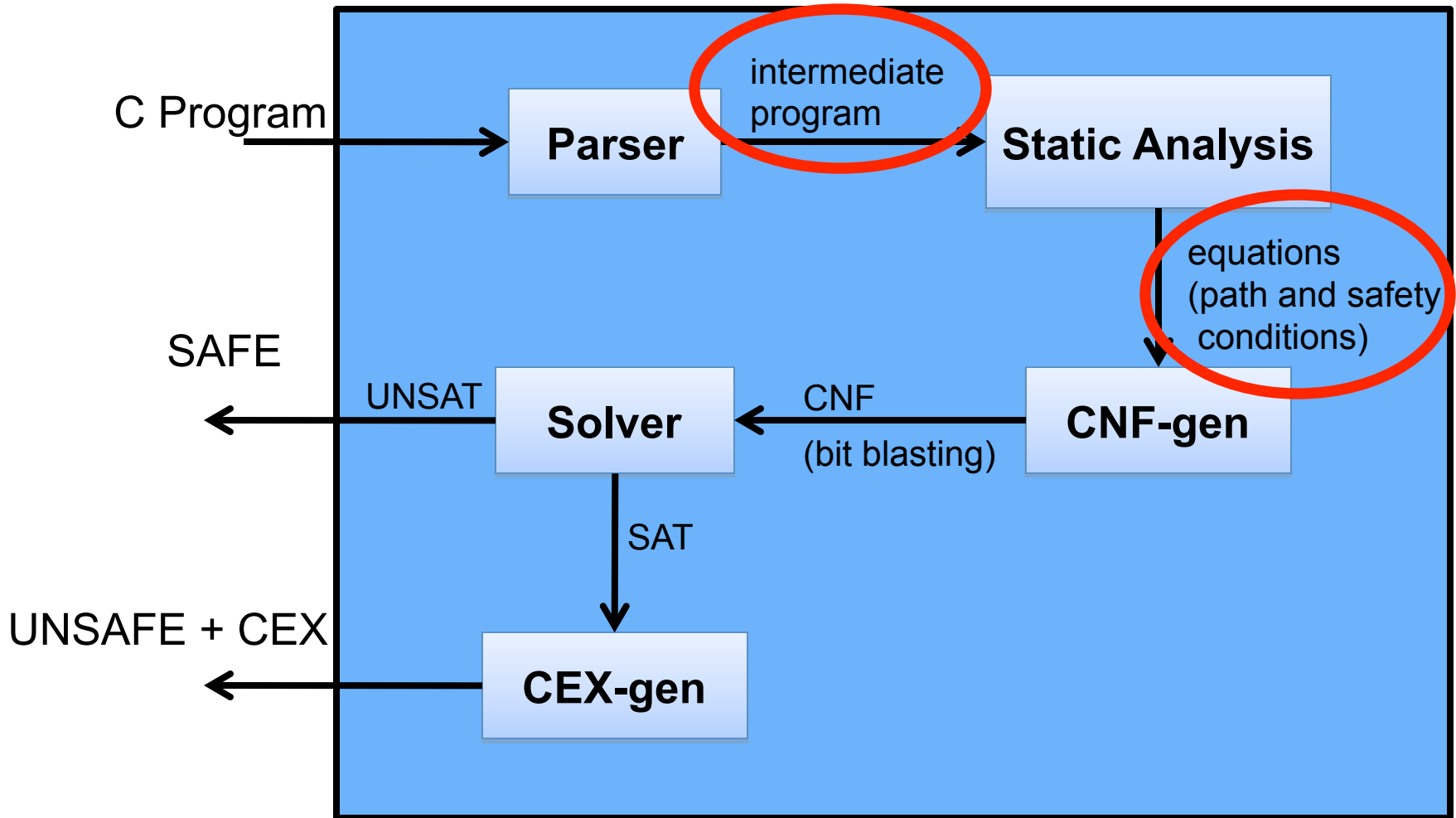
`⇒ assert(0 <= i && i <= N); a[i]=...;`

⇒ 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)?

```
if(v)
  x = y;
else
  x = z;

w = x;
```



```
if(v0)
  x0 = y0;
else
  x1 = z0;

w1 = ?
```

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)?

```
if(v)
  x = y;
else
  x = z;

w = x;
```



```
if(v0)
  x0 = y0;
else
  x1 = z0;
x2 = v0 ? x0 : x1;
w1 = x2;
```

introduce & use
new variable

- 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

\Rightarrow 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:

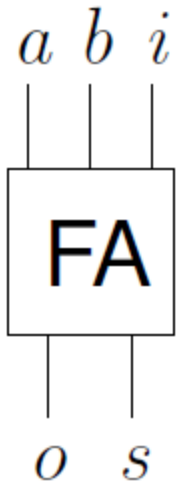
$$| [x = y \mid z] | \triangleq \bigwedge_i x_i \Leftrightarrow (y_i \vee z_i)$$

Exercise: relational operations

Bit-blasting arithmetic operations

Build **circuits** that implement the operations!

1-bit addition:



Full Adder

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

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

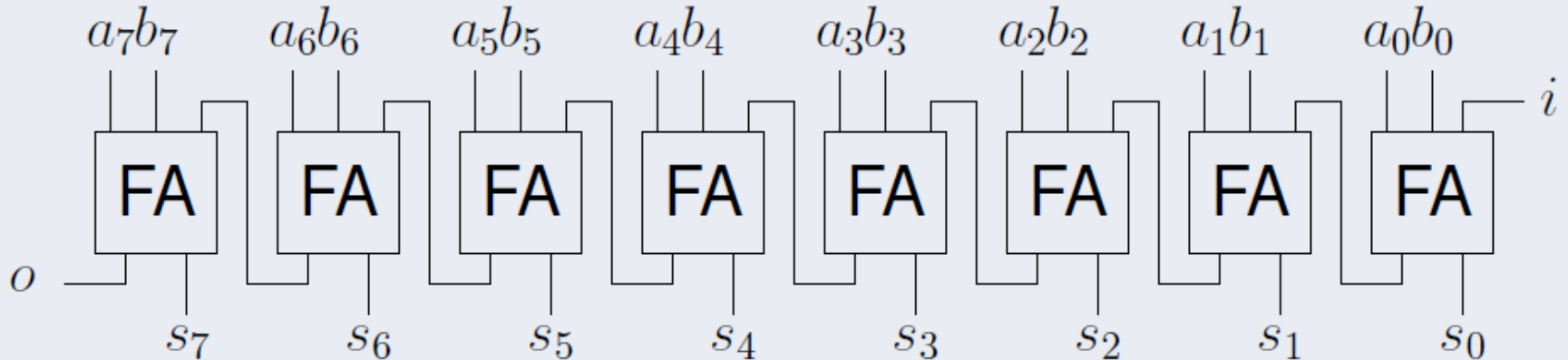
Full adder as CNF:

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

Bit-blasting arithmetic operations

Build **circuits** that implement the operations!

8-Bit ripple carry adder (RCA)



⇒ adds w variables, $6 \cdot w$ clauses

⇒ multiplication / division much more complicated

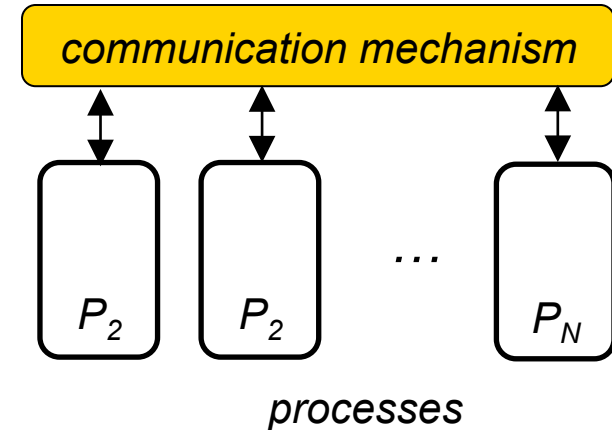
Intended learning outcomes

- Introduce typical **BMC architectures** for verifying **software systems**
- Understand **communication models** and **typical errors** when writing **concurrent programs**
- Explain **explicit schedule exploration** of multi-threaded software
- Explain **sequentialization methods** to convert concurrent programs into sequential ones

Concurrency verification

Writing concurrent programs is DIFFICULT

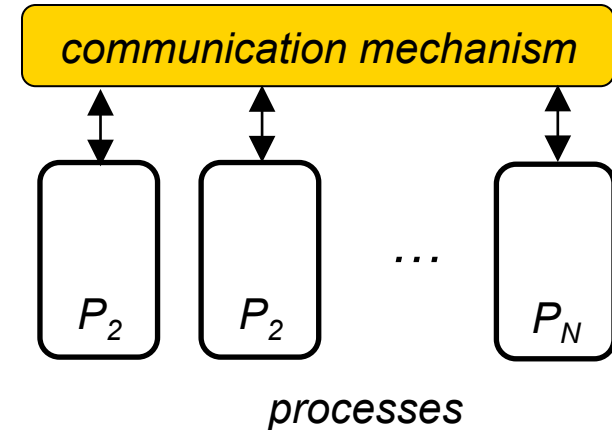
- programmers have to guarantee
 - correctness of sequential execution of each individual process
 - with nondeterministic interferences from other processes (schedules)



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
$./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

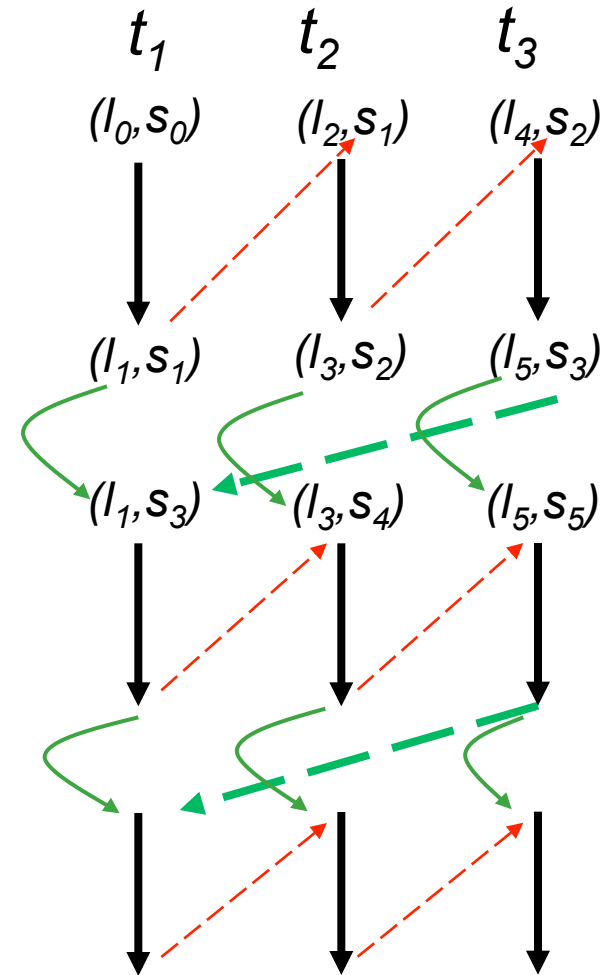
- typical C-implementation: pthreads
- formed of individual sequential programs (**threads**)
 - can be created and destroyed on the fly
 - typically for BMC: assume upper bound
 - each possibly with loops and recursive function calls
 - each with **local variables**
- each thread can read and write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

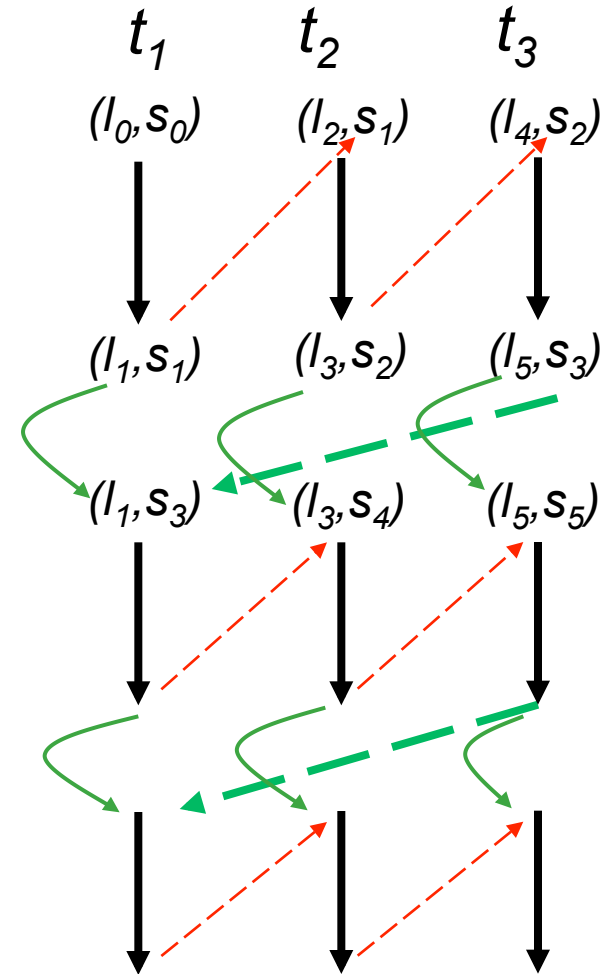
Round-robin scheduling

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



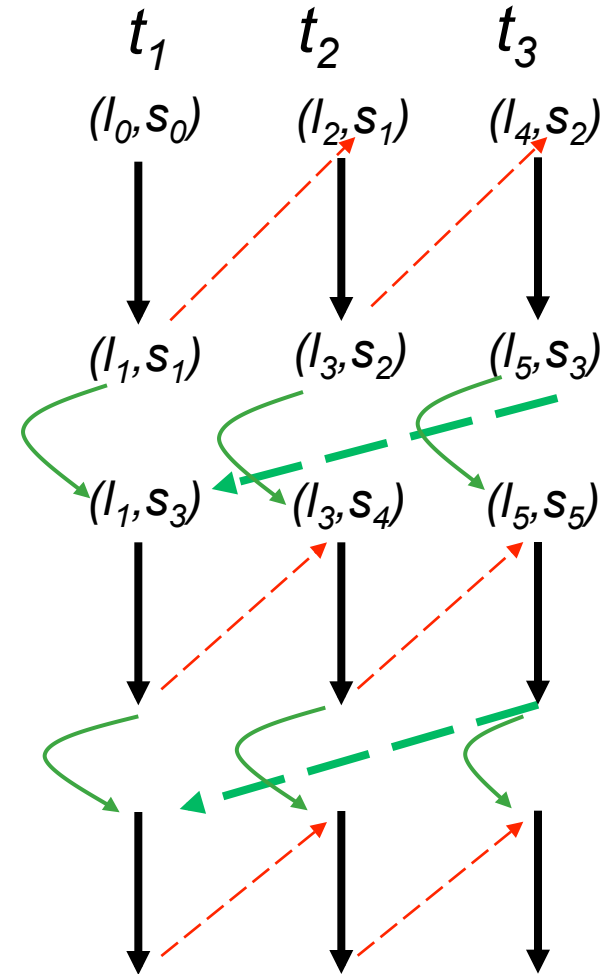
Round-robin scheduling

- **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)



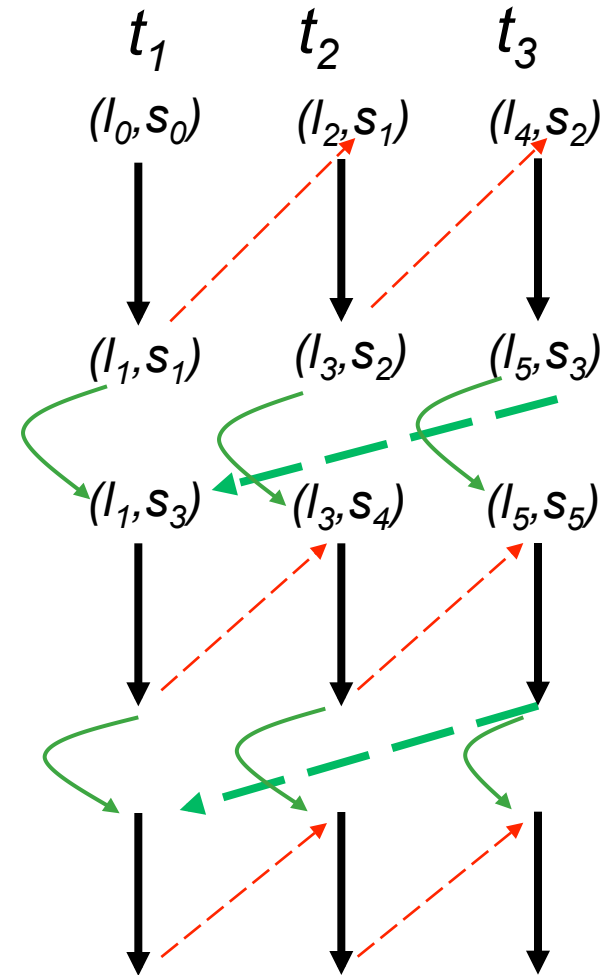
Round-robin scheduling

- **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



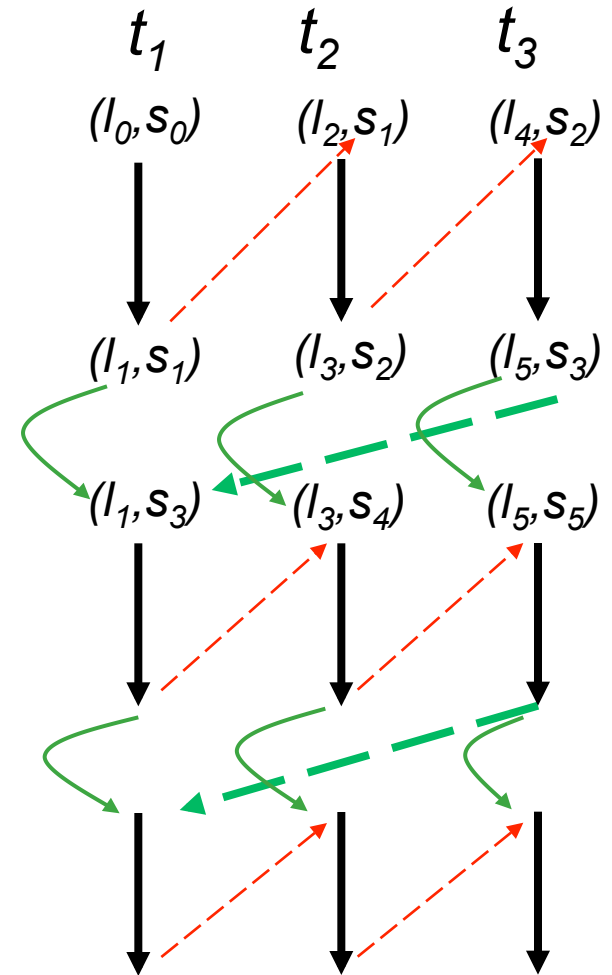
Round-robin scheduling

- **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

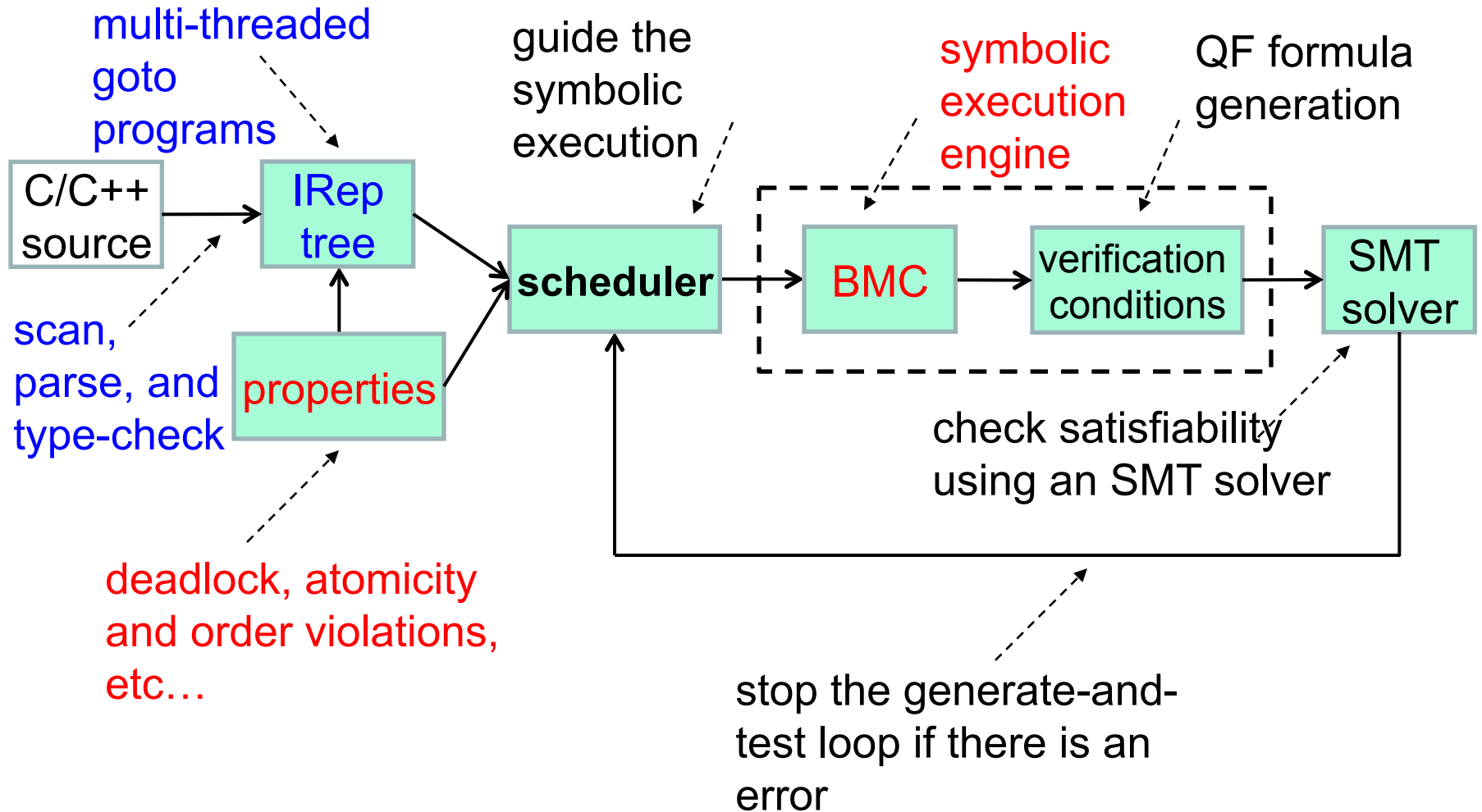
- Explicit schedule exploration (ESBMC)
 - lazy exploration
 - schedule recording
- Partial order methods (CBMC)
- Sequentialization
 - KISS
 - Lal / Reps (eager sequentialization)
 - Lazy CSeq
 - memory unwinding

Intended learning outcomes

- Introduce typical **BMC architectures** for verifying **software systems**
- Understand **communication models** and **typical errors** when writing **concurrent programs**
- Explain **explicit schedule exploration** of multi-threaded software
- Explain **sequentialization methods** to convert concurrent programs into sequential ones

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);  
8: lock(m1);  
9: t1 = val1;  
10: lock(m2);  
11: t2 = val2;  
12: unlock(m2);  
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);
```

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_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);*

program counter: 1

mutexes: m1=1; 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_s

statements: 1-2

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

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 L

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);  
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);

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 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:

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 variables: t1= 1; t2= -1;

Lazy exploration: interleaving I_s

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

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

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

Thread twoStage

1: lock(m1);

2: val1 = 1;

3: unlock(m1);

4: lock(m2);

5: val2 = val1 + 1;

6: unlock(m2);

CS1

CS2

Thread reader

7: lock(m1);

8: if (val1 == 0) {

9: unlock(m1);

10: return NULL; }

11: t1 = val1;

12: unlock(m1);

13: lock(m2);

14: t2 = val2;

15: unlock(m2);

16: assert(t2==(t1+1));

● **program counter: 5**

mutexes: m1=0; m2=1;

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

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

Lazy exploration: interleaving I_s

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

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

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

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 variables: t1= 1; t2= -1;

Lazy exploration: interleaving I_s

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

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

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

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

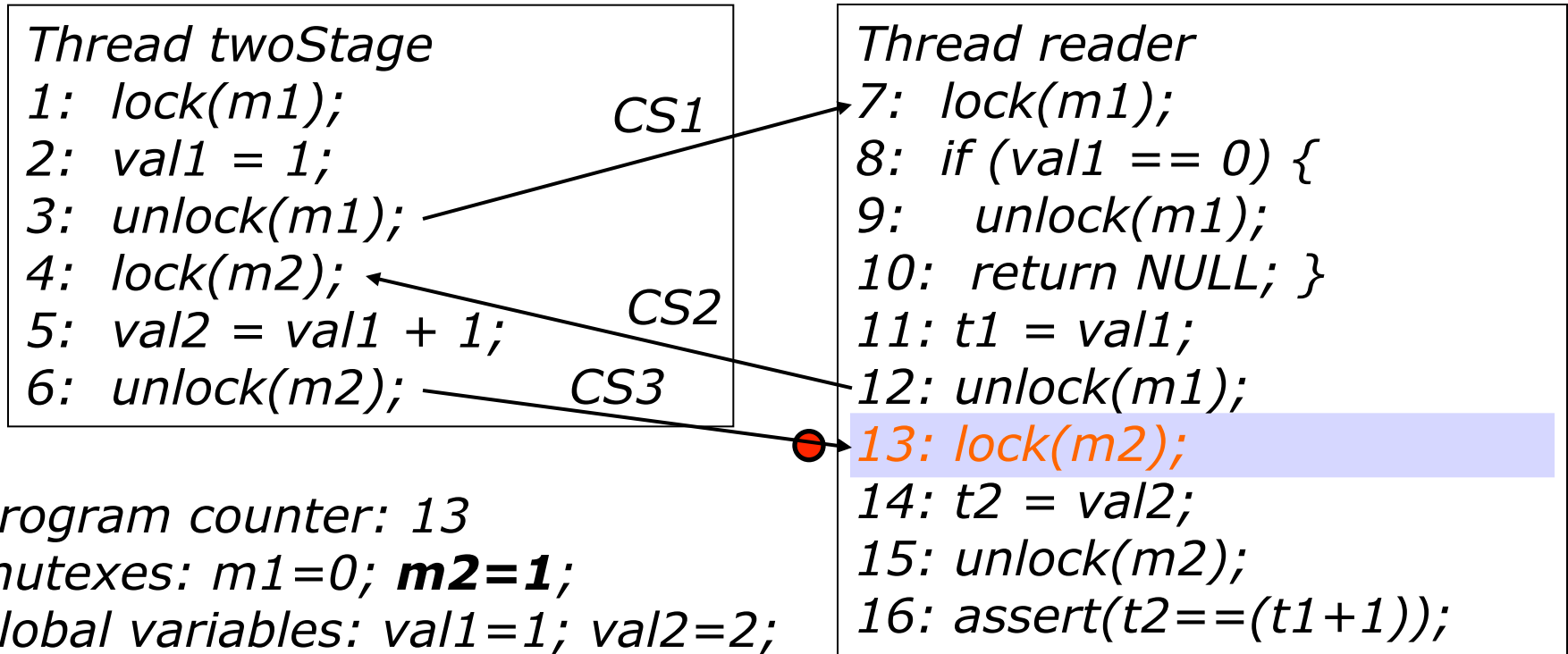
local variables: $t1=1$; $t2=-1$;

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



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

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;

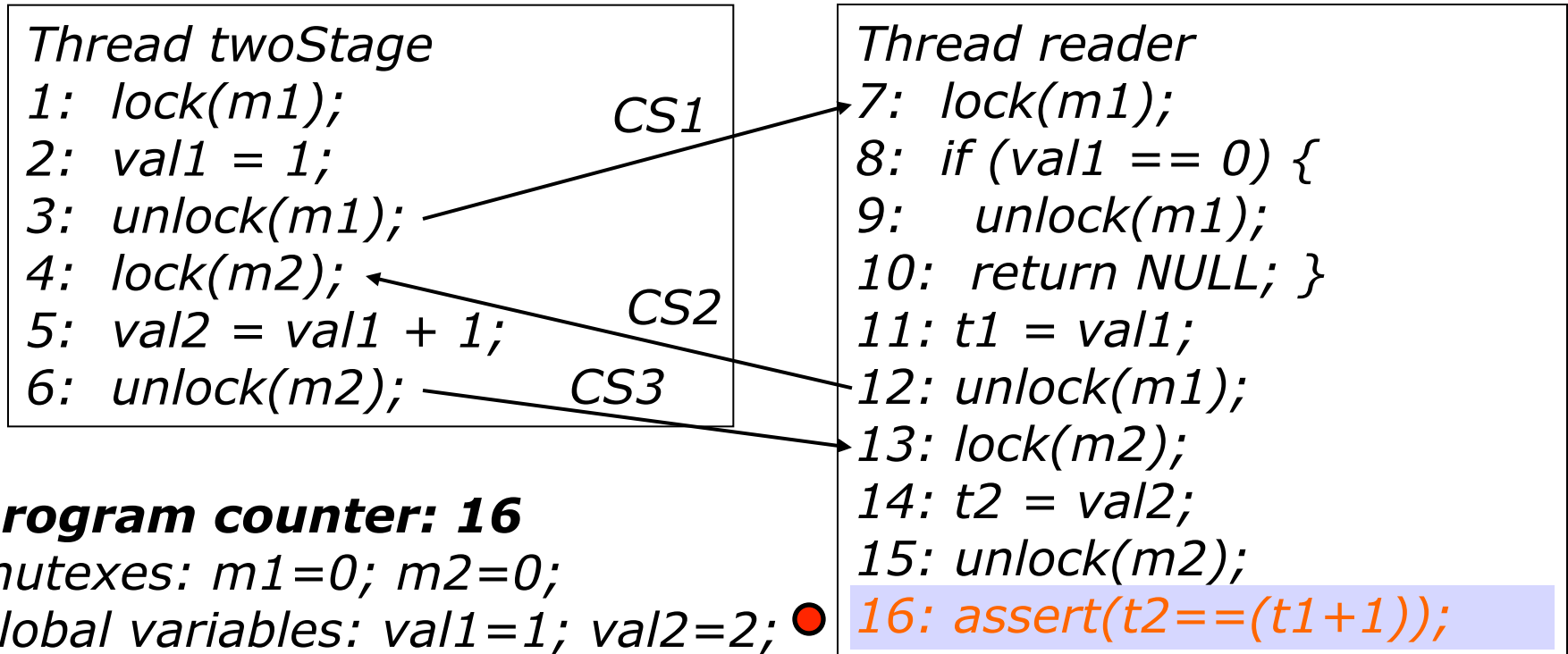
local variables: t1= 1; t2= 2;

Lazy exploration: interleaving I_s

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

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

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

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);
```

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);
```

program counter: 3

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

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

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);
```

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: val1=1; **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

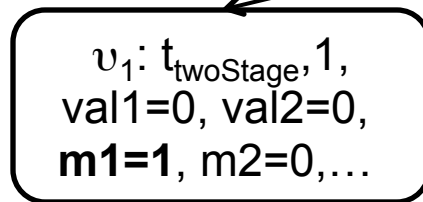
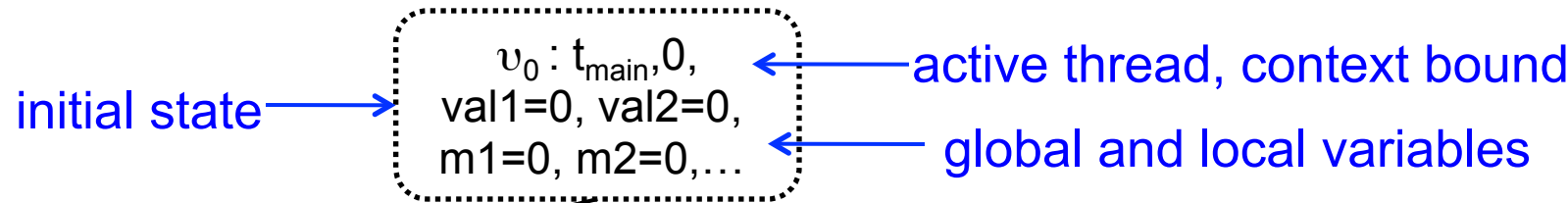
Idea: iteratively generate all possible interleavings and call the BMC procedure on each interleaving

... combines

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

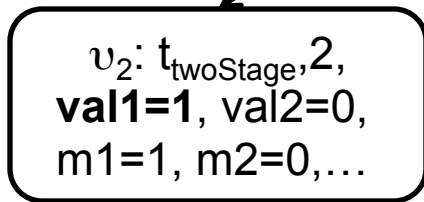
Lazy exploration of interleavings

– Reachability Tree



expansion rules in paper

CS1



interleaving completed, so
call single-threaded BMC

→ execution paths

Lazy exploration of interleavings

– Reachability Tree

initial state

$v_0: t_{\text{main}}, 0,$
 $\text{val1}=0, \text{val2}=0,$
 $m1=0, m2=0, \dots$

active thread, context bound

global and local variables

$v_1: t_{\text{twoStage}}, 1,$
 $\text{val1}=0, \text{val2}=0,$
 $m1=1, m2=0, \dots$

backtrack to last unexpanded node
and continue

$v_2: t_{\text{twoStage}}, 2,$
val1=1, $\text{val2}=0,$
 $m1=1, m2=0, \dots$

$v_3: t_{\text{reader}}, 2,$
 $\text{val1}=0, \text{val2}=0,$
 $m1=1, m2=0, \dots$

symbolic execution can statically
determine that path is blocked
(encoded in instrumented mutex-op)



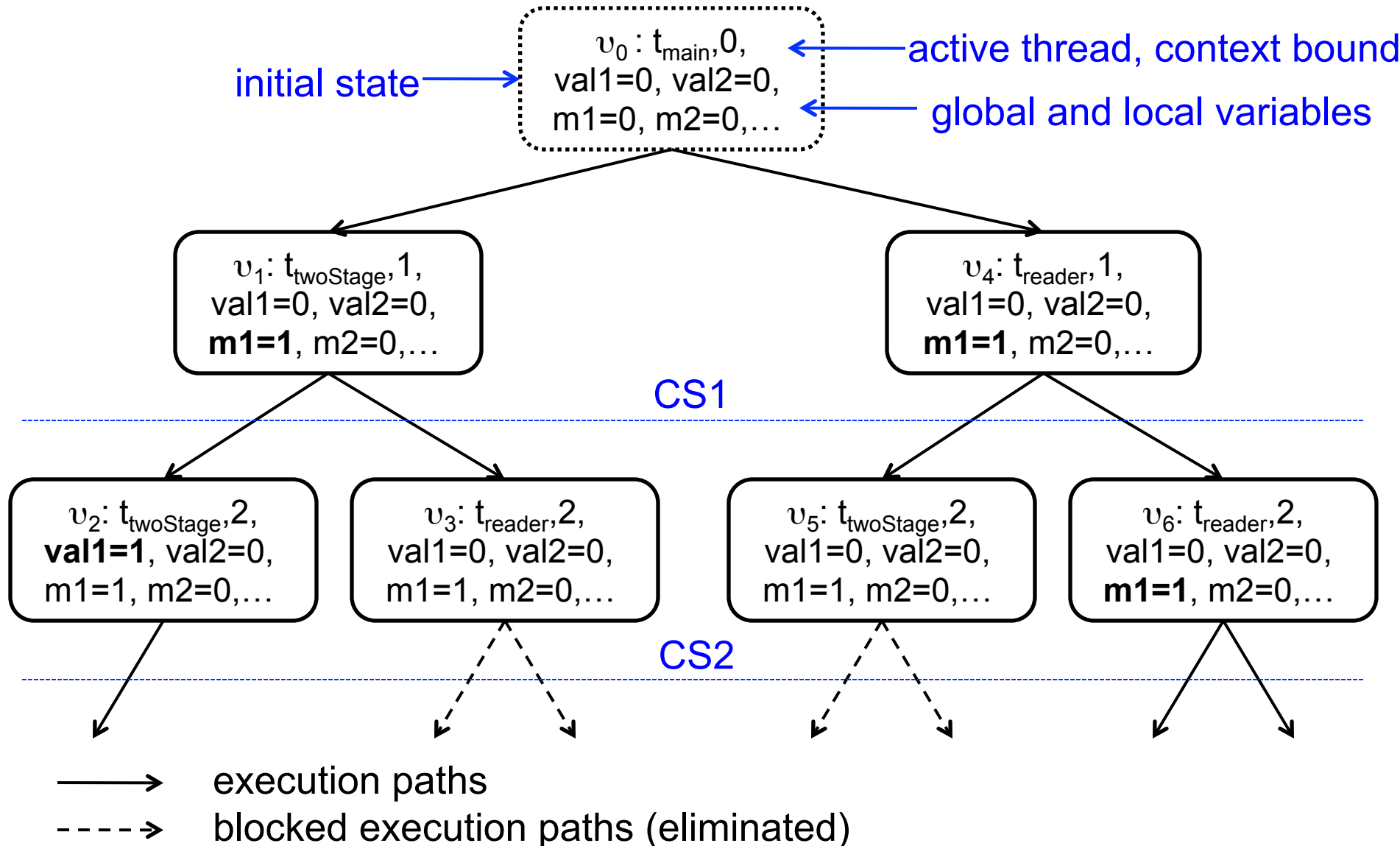
execution paths



blocked execution paths (eliminated)

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 $v = \left(A_i, C_i, s_i, \left\langle l_i^j, G_i^j \right\rangle_{j=1}^n \right)_i$ 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 $v = \left(A_i, C_i, s_i, \left\langle l_i^j, G_i^j \right\rangle_{j=1}^n \right)_i$ for a given time step i , where:
 - A_i represents the currently active thread

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 $v = \left(A_i, C_i, s_i, \left\langle l_i^j, G_i^j \right\rangle_{j=1}^n \right)_i$ for a given time step i , where:
 - A_i represents the currently active thread
 - C_i represents the context switch number

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 $v = \left(A_i, C_i, s_i, \left\langle l_i^j, G_i^j \right\rangle_{j=1}^n \right)_i$ 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

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 $v = \left(A_i, C_i, s_i, \left\langle l_i^j, G_i^j \right\rangle_{j=1}^n \right)_i$ 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

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 $v = \left(A_i, C_i, s_i, \left\langle l_i^j, G_i^j \right\rangle_{j=1}^n \right)_i$ 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

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'

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

- 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

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

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:

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

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:

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

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:

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

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:

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

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

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

- 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_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

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_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 l 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 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:

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

- 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 l is a *join_thread* instruction with argument ld , 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 ld 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.

computation path

$$\pi = \{v_1, \dots, v_n\}$$

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

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

Schedule Recording

Idea: systematically encode all possible interleavings into one formula

- explore reachability tree in same way as lazy approach
- ... but call SMT solver only once
- add a **schedule guard** ts_i for each context switch block i ($0 < ts_i \leq \#threads$)
 - record in which order the scheduler has executed the program
 - SMT solver determines the order in which threads are simulated
- add scheduler guards only to **effective statements** (assignments and assertions)
 - record **effective context switches (ECS)**
 - ECS block: sequence of program statements that are executed with no intervening ECS

Schedule Recording – Interleaving #1

statements:

twoStage-ECS:

reader-ECS:

Thread twoStage

1: lock(m1);

2: val1 = 1;

3: unlock(m1);

4: lock(m2);

5: val2 = val1 + 1;

6: unlock(m2);

ECS block

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));

Schedule Recording – Interleaving #1

statements: 1

twoStage-ECS: (1,1)

reader-ECS:

guarded statement can only be executed if **statement 1** is scheduled in **ECS block 1**

Thread twoStage

```
1: lock(m1);       $ts_1 == 1$ 
2: val1 = 1;
3: unlock(m1);
4: lock(m2);
5: val2 = val1 + 1;
6: unlock(m2);
```

Thread reader

7: test(m1);

each program statement is then prefixed by a schedule guard $ts_i = j$, where:

- i is the **ECS block number**
- j is the **thread identifier**

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

Schedule Recording – Interleaving #1

statements: 1-2

twoStage-ECS: (1,1)-(2,2)

reader-ECS:

Thread twoStage

1: lock(m1); $ts_1 == 1$

2: val1 = 1; $ts_2 == 1$

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));

Schedule Recording – Interleaving #1

statements: 1-2-3

twoStage-ECS: (1,1)-(2,2)-(3,3)

reader-ECS:

Thread twoStage

1: lock(m1); $ts_1 == 1$

2: val1 = 1; $ts_2 == 1$

3: unlock(m1); $ts_3 == 1$

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));

Schedule Recording – Interleaving #1

statements: 1-2-3-7

twoStage-ECS: (1,1)-(2,2)-(3,3)

reader-ECS: (7,4)

Thread twoStage

```
1: lock(m1);          ts1 == 1
2: val1 = 1;          ts2 == 1
3: unlock(m1);        ts3 == 1
4: lock(m2);
5: val2 = val1 + 1;
6: unlock(m2);
```

CS

Thread reader

```
7: lock(m1);          ts4 == 2
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));
```


Schedule Recording – Interleaving #1

statements: 1-2-3-7-8

twoStage-ECS: (1,1)-(2,2)-(3,3)

reader-ECS: (7,4)-(8,5)

Thread twoStage

```
1: lock(m1);          ts1 == 1
2: val1 = 1;          ts2 == 1
3: unlock(m1);        ts3 == 1
4: lock(m2);
5: val2 = val1 + 1;
6: unlock(m2);
```

CS

Thread reader

```
7: lock(m1);          ts4 == 2
8: if (val1 == 0) {    ts5 == 2
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));
```

Schedule Recording – Interleaving #1

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

twoStage-ECS: (1,1)-(2,2)-(3,3)

reader-ECS: (7,4)-(8,5)-(11,6)

Thread twoStage

```
1: lock(m1);          ts1 == 1
2: val1 = 1;          ts2 == 1
3: unlock(m1);        ts3 == 1
4: lock(m2);
5: val2 = val1 + 1;
6: unlock(m2);
```

CS

Thread reader

```
7: lock(m1);          ts4 == 2
8: if (val1 == 0) {    ts5 == 2
9:   unlock(m1);
10:  return NULL; }
11: t1 = val1;         ts6 == 2
12: unlock(m1);
13: lock(m2);
14: t2 = val2;
15: unlock(m2);
16: assert(t2==(t1+1));
```

Schedule Recording – Interleaving #1

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

twoStage-ECS: (1,1)-(2,2)-(3,3)

reader-ECS: (7,4)-(8,5)-(11,6)-(12,7)

Thread twoStage

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

CS

Thread reader

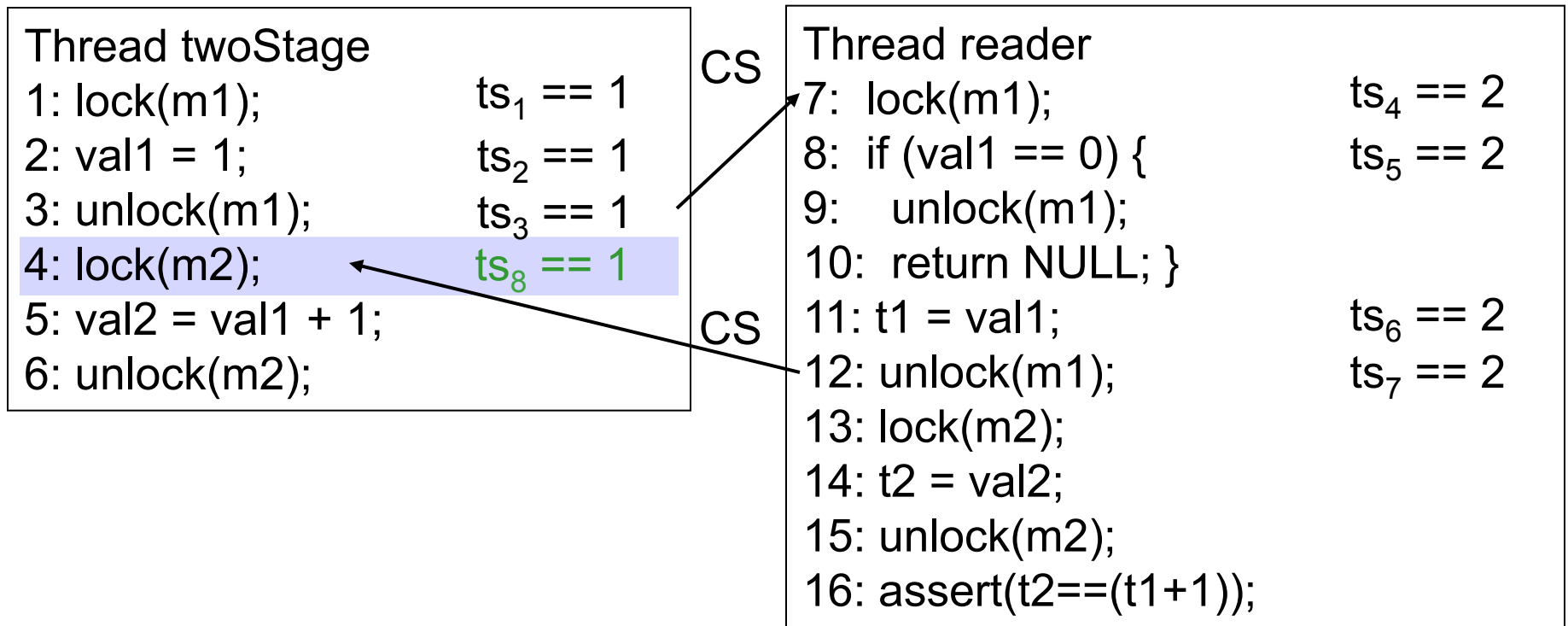
7: lock(m1);	ts ₄ == 2
8: if (val1 == 0) {	ts ₅ == 2
9: unlock(m1);	
10: return NULL; }	
11: t1 = val1;	ts ₆ == 2
12: unlock(m1);	ts ₇ == 2
13: lock(m2);	
14: t2 = val2;	
15: unlock(m2);	
16: assert(t2==(t1+1));	

Schedule Recording – Interleaving #1

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

twoStage-ECS: (1,1)-(2,2)-(3,3)-(4,8)

reader-ECS: (7,4)-(8,5)-(11,6)-(12,7)

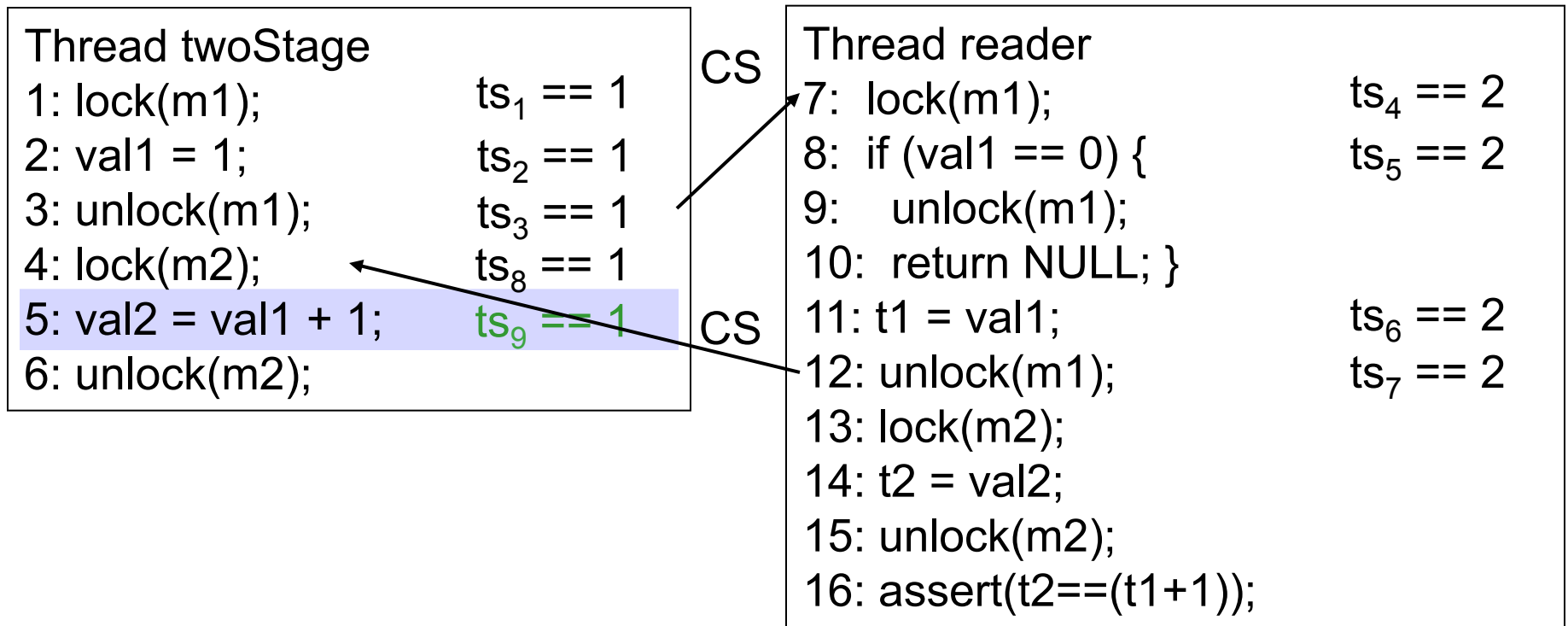


Schedule Recording – Interleaving #1

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

twoStage-ECS: (1,1)-(2,2)-(3,3)-(4,8)-(5,9)

reader-ECS: (7,4)-(8,5)-(11,6)-(12,7)

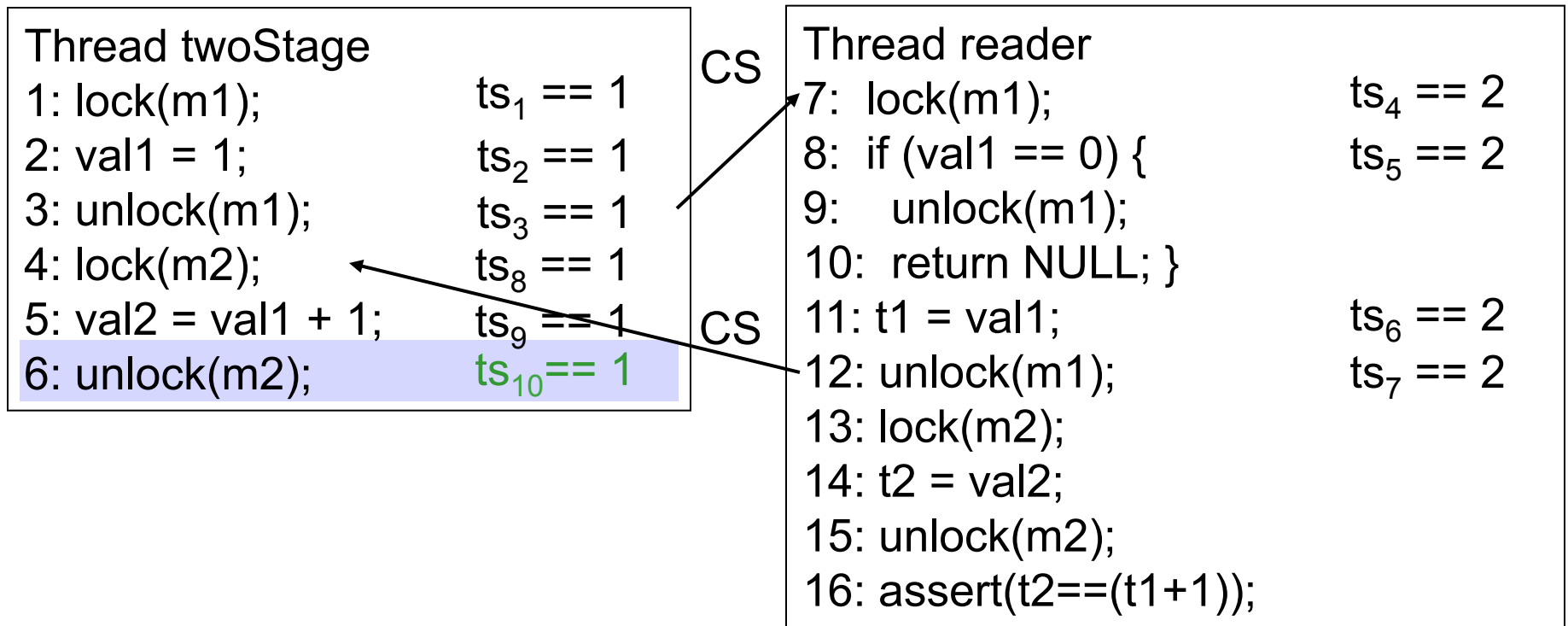


Schedule Recording – Interleaving #1

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

twoStage-ECS: (1,1)-(2,2)-(3,3)-(4,8)-(5,9)-(6,10)

reader-ECS: (7,4)-(8,5)-(11,6)-(12,7)

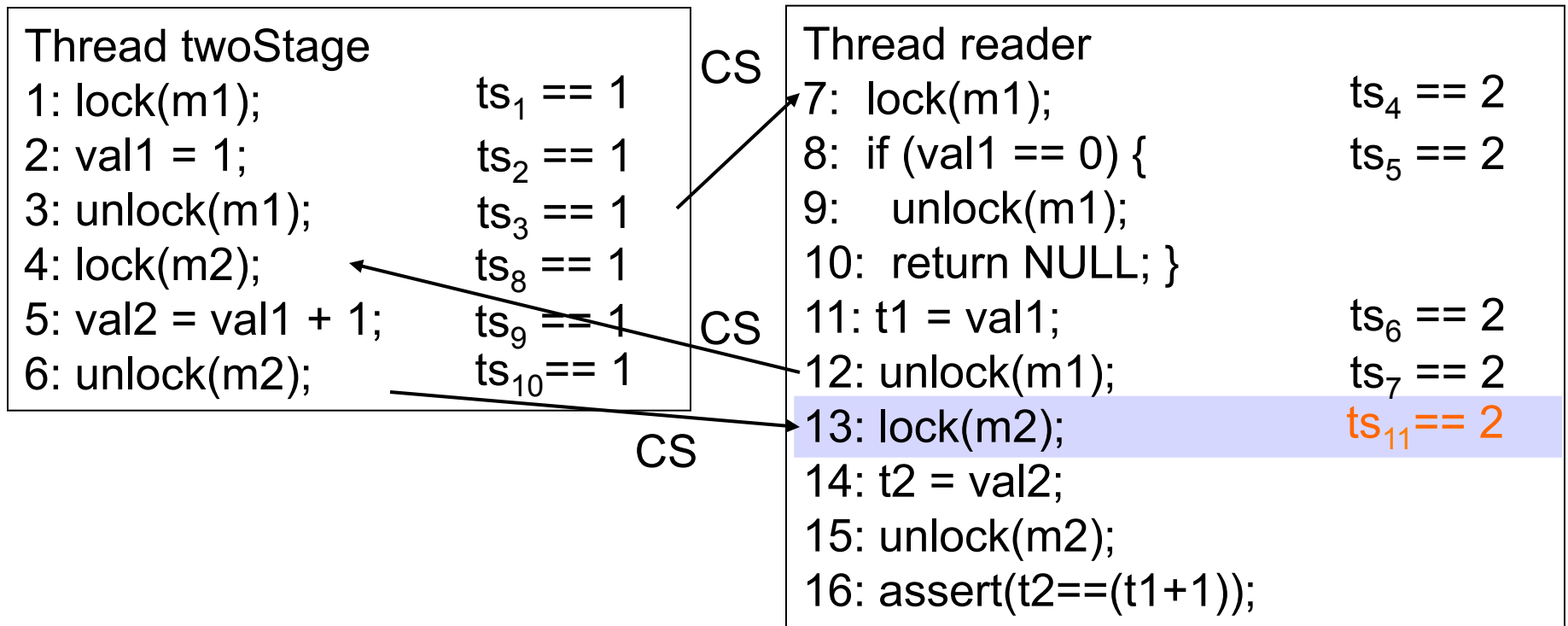


Schedule Recording – Interleaving #1

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

twoStage-ECS: (1,1)-(2,2)-(3,3)-(4,8)-(5,9)-(6,10)

reader-ECS: (7,4)-(8,5)-(11,6)-(12,7)-(13,11)

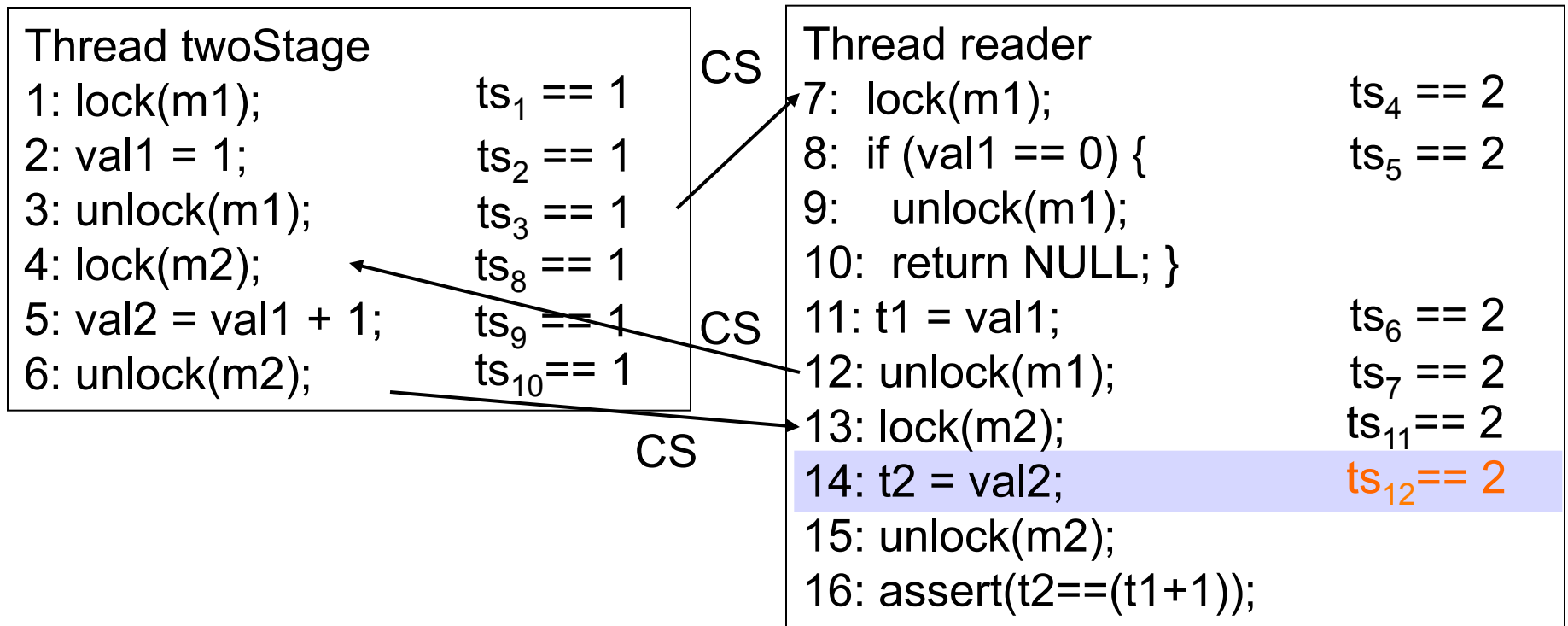


Schedule Recording – Interleaving #1

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

twoStage-ECS: (1,1)-(2,2)-(3,3)-(4,8)-(5,9)-(6,10)

reader-ECS: (7,4)-(8,5)-(11,6)-(12,7)-(13,11)-(14,12)

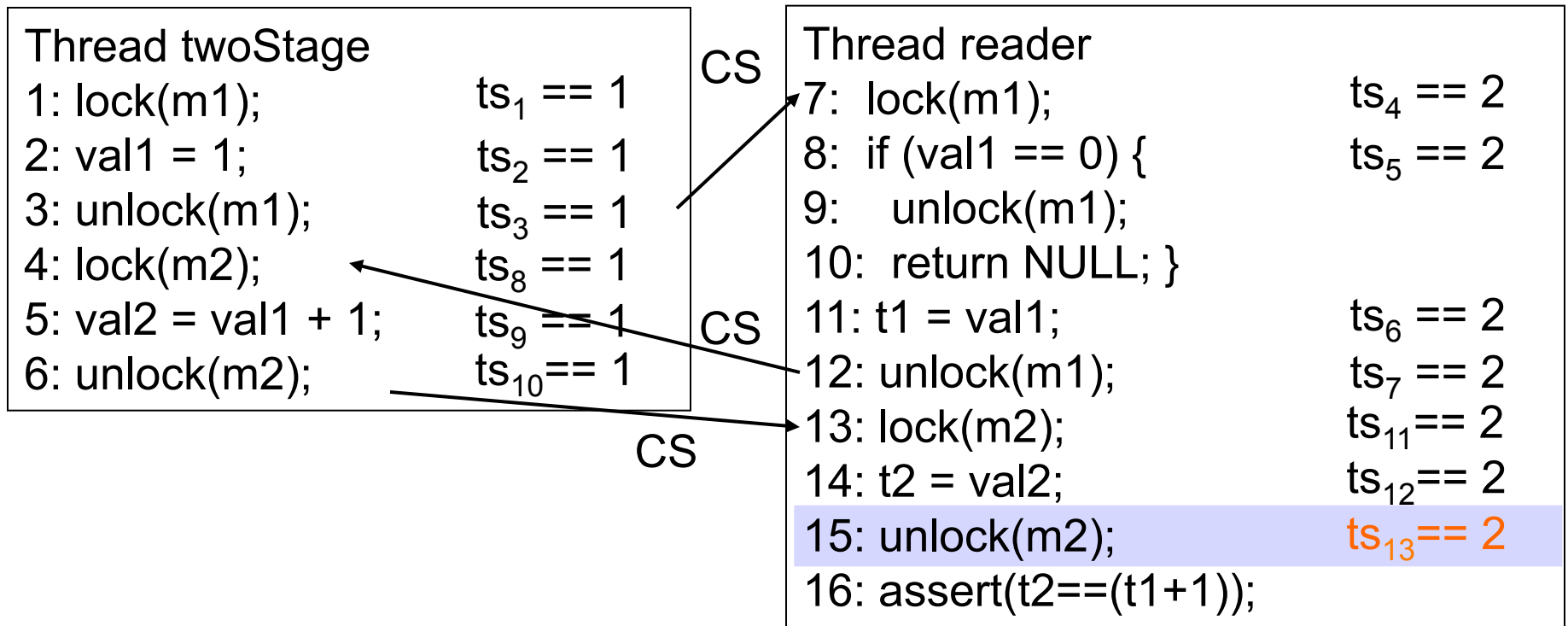


Schedule Recording – Interleaving #1

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

twoStage-ECS: (1,1)-(2,2)-(3,3)-(4,8)-(5,9)-(6,10)

reader-ECS: (7,4)-(8,5)-(11,6)-(12,7)-(13,11)-(14,12)-(15,13)

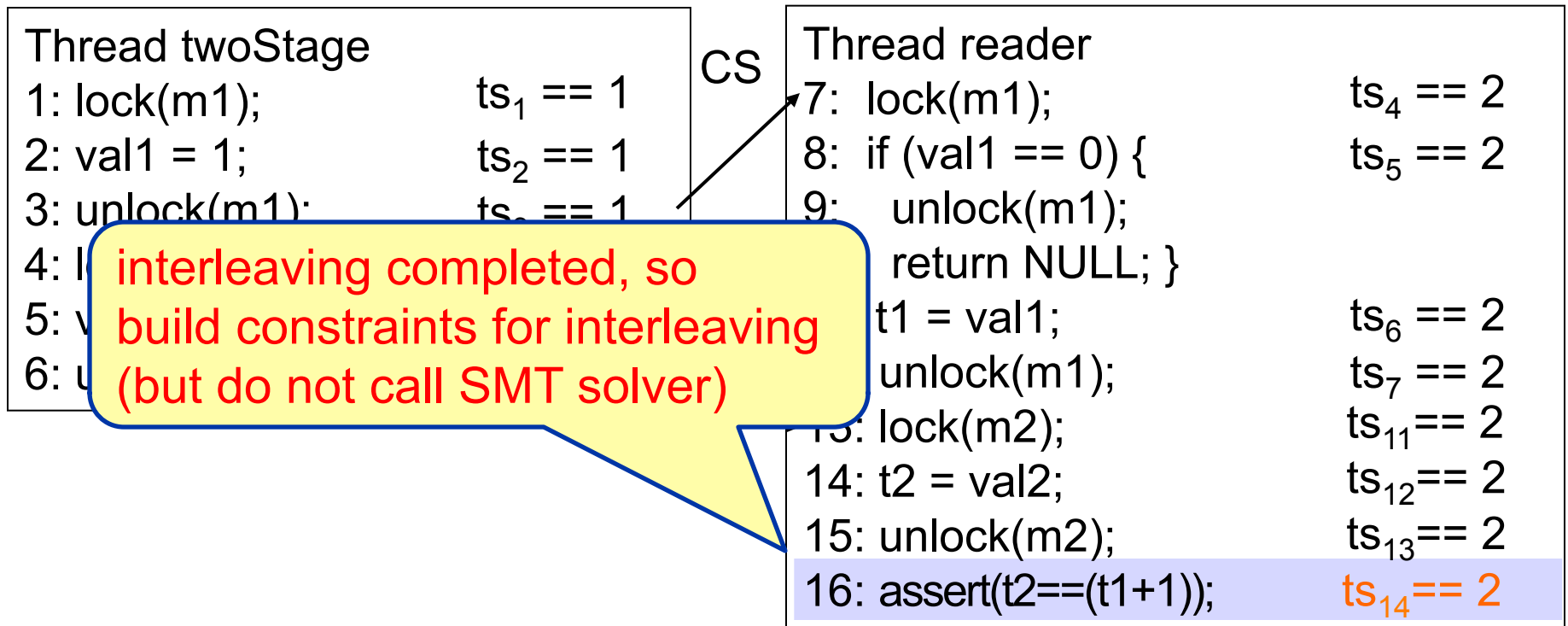


Schedule Recording – Interleaving #1

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

twoStage-ECS: (1,1)-(2,2)-(3,3)-(4,8)-(5,9)-(6,10)

reader-ECS: (7,4)-(8,5)-(11,6)-(12,7)-(13,11)-(14,12)-(15,13)-(16,14)

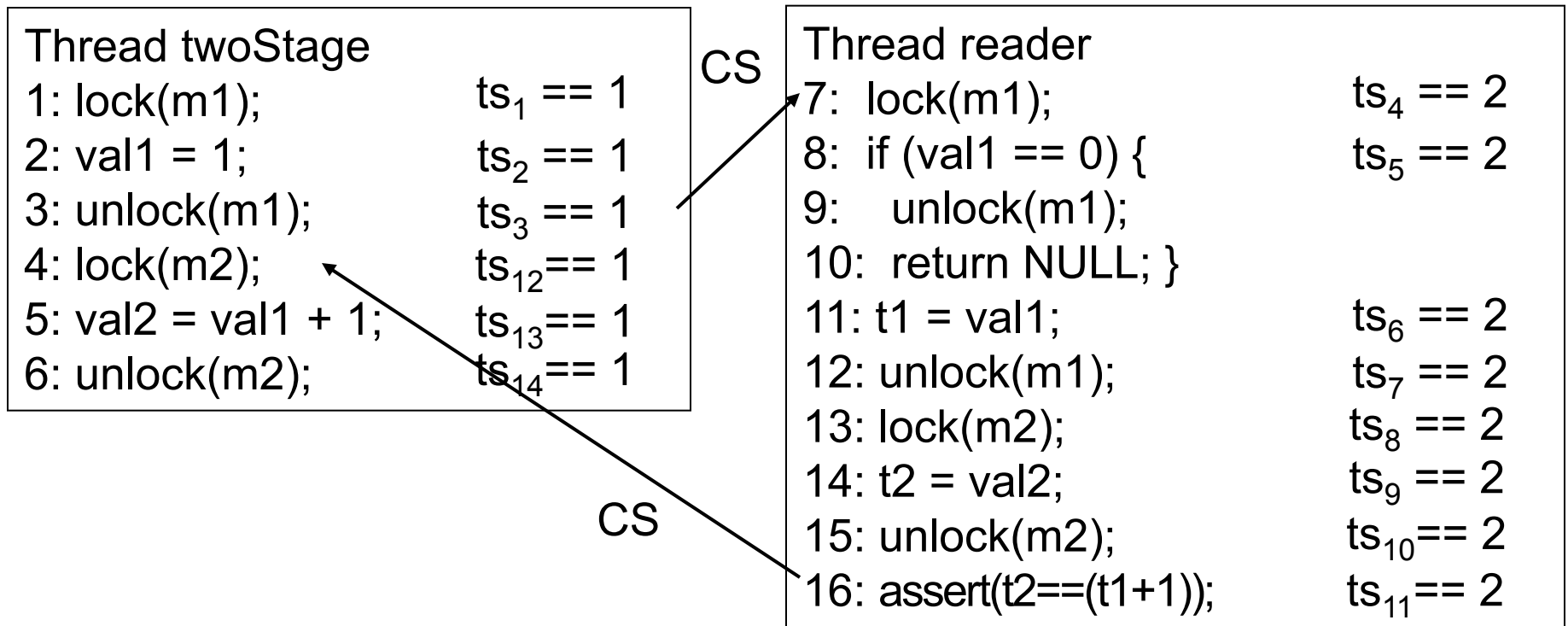


Schedule Recording – Interleaving #1

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

twoStage-ECS: (1,1)-(2,3)-(3,4)-(4,12)-(5,13)-(6,14)

reader-ECS: (7,4)-(8,5)-(11,6)-(12,7)-(13,8)-(14,9)-(15,10)-(16,11)



Schedule Recording: Execution Paths

SMT solver
instantiates ts to
evaluate all possible
interleavings

twoStage, reader

thread
identifiers

program
statement

twoStage, reader
 $ts_1 == 1 \rightarrow \text{lock}(m1)$

twoStage, reader
 $ts_1 == 2 \rightarrow \text{lock}(m1)$

CS1

twoStage, reader
 $ts_1 == 1 \wedge ts_2 == 1$
 $\rightarrow \text{val} = 1$

twoStage, reader
 $ts_1 == 1 \wedge ts_2 == 2$
 $\rightarrow \text{lock}(m1)$

twoStage, reader
 $ts_1 == 2 \wedge ts_2 == 1$
 $\rightarrow \text{lock}(m1)$

twoStage, reader
 $ts_1 == 2 \wedge ts_2 == 2$
 $\rightarrow \text{unlock}(m1)$

CS2

If the guard of the parent node is
false then the guard of the child
node is false as well

Observations about the schedule recoding approach

- systematically explore the thread interleavings as before, but:
 - add schedule guards to record in which order the scheduler has executed the program
 - encode all execution paths into one formula
 - o bound the number of context switches
 - o exploit which transitions are enabled in a given state
- number of threads and context switches grows very large quickly, and easily “blow-up” the solver:
 - there is a clear trade-off between usage of time and memory resources

Intended learning outcomes

- Introduce typical **BMC architectures** for verifying **software systems**
- Understand **communication models** and **typical errors** when writing **concurrent programs**
- Explain **explicit schedule** exploration of multi-threaded software
- Explain **sequentialization methods** to convert concurrent programs into sequential ones

Sequentialization

Observation:

Building verification tools for full-fledged concurrent languages is difficult and expensive...

... but scalable verification techniques exist for sequential languages

- Abstraction techniques
- SAT/SMT techniques (i.e., bounded model checking)

⇒ How can we leverage these?

Sequentialization

⇒ How can we leverage these?

Sequentialization:

convert **concurrent** programs into **sequential** programs such that reachability is preserved

- replace **control non-determinism** by **data non-determinism**
- **P'** simulates all computations (within certain bounds) of **P**
- **source-to-source transformation**: $T_1 \parallel T_2 \rightsquigarrow T'_1 ; T'_2$

⇒ reuse existing tools (largely) unchanged

⇒ easy to target multiple back-ends

⇒ easy to experiment with different approaches

A first sequentialization: KISS

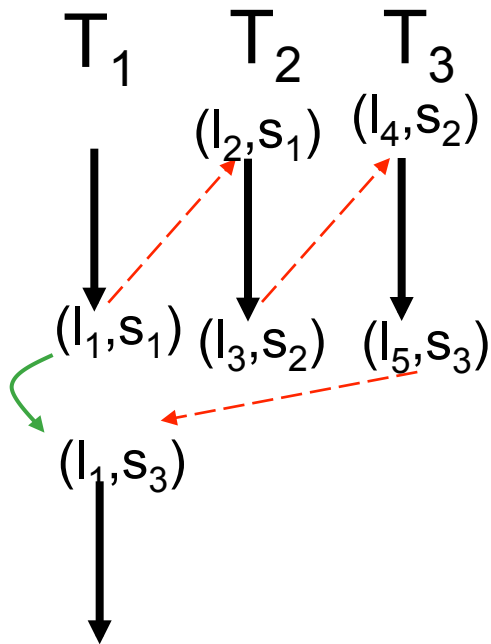
KISS: Keep It Simple and Sequential [\[Quadeer-Wu, PLDI' 04\]](#)

Under-approximation (subset of interleavings)

Thread creation → function call

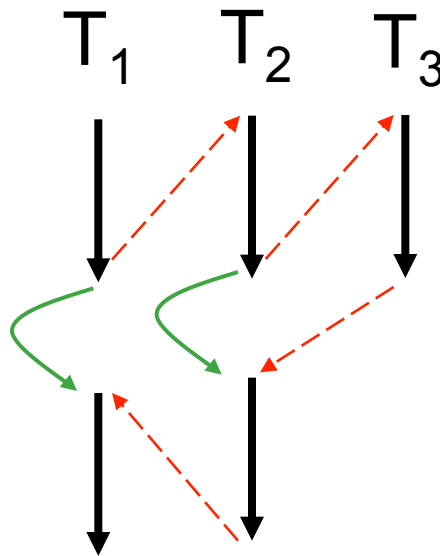
- **at context-switches either:**
 - o the active thread is terminated or
 - o a not yet scheduled thread is started
(by calling its main function)
- **when a thread is terminated either:**
 - o the thread that has called it is resumed (if any) or
 - o a not yet scheduled thread is started

KISS schedules



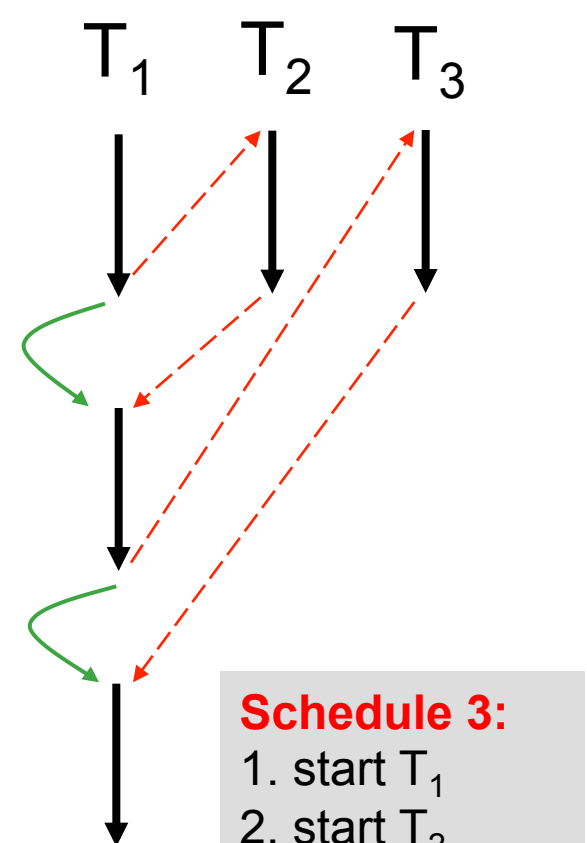
Schedule 1:

1. Start T_1
2. Start T_2
3. Terminate T_2
4. start T_3
5. terminate T_3
6. Resume T_1



Schedule 2:

1. start T_1
2. start T_2
3. start T_3
4. terminate T_3
5. resume T_2
6. terminate T_2
7. resume T_1

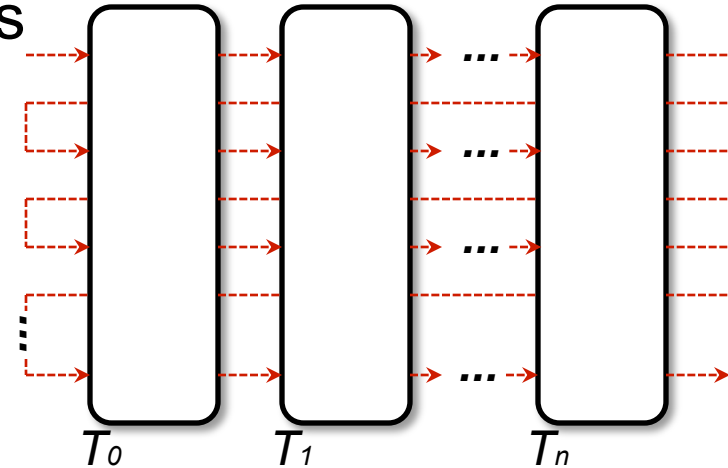


Schedule 3:

1. start T_1
2. start T_2
3. terminate T_2
4. resume T_1
5. start T_3
6. terminate T_3
7. resume T_1

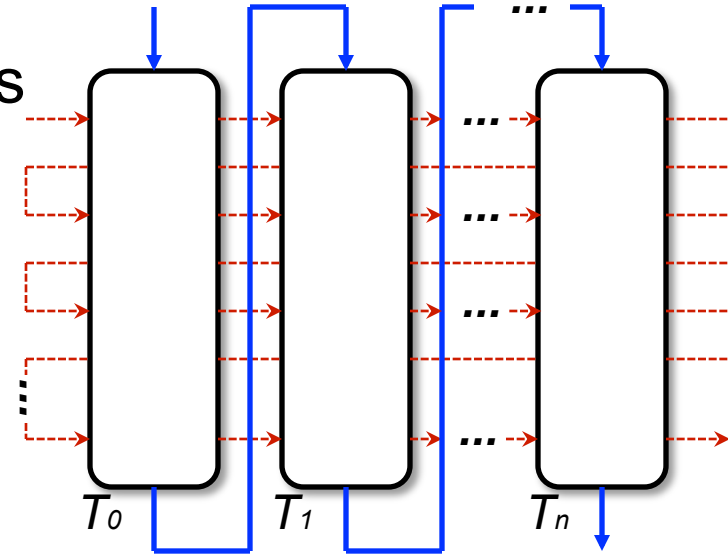
LR sequentialization

- considers only round-robin schedules with k rounds



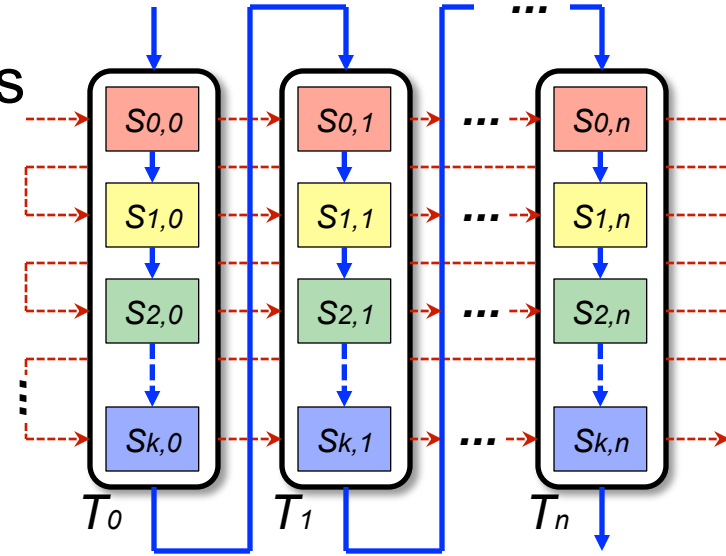
LR sequentialization

- considers only round-robin schedules with k rounds
 - **thread** → **function**, run to completion



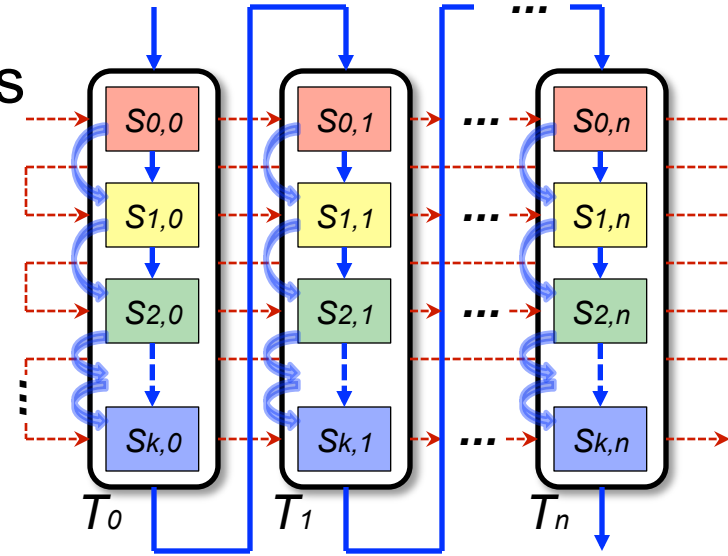
LR sequentialization

- considers only round-robin schedules with k rounds
 - **thread** \rightarrow **function**, run to completion
- global memory copy for each round
 - **scalar** \rightarrow **array**



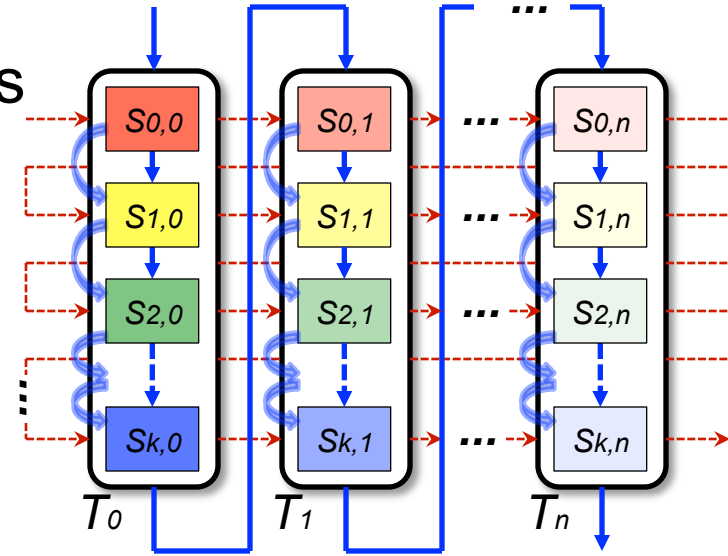
LR sequentialization

- considers only round-robin schedules with k rounds
 - **thread** \rightarrow **function**, run to completion
- global memory copy for each round
 - **scalar** \rightarrow **array**
- **context switch** \rightarrow **round counter++**



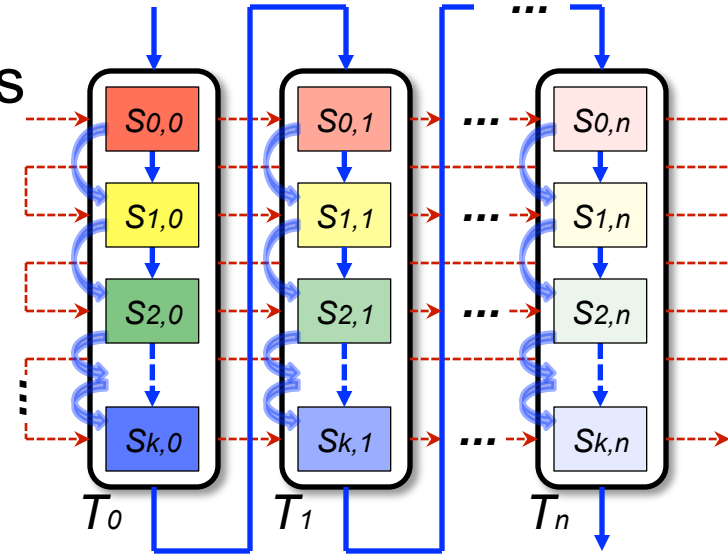
LR sequentialization

- considers only round-robin schedules with k rounds
 - **thread** \rightarrow **function**, run to completion
- global memory copy for each round
 - **scalar** \rightarrow **array**
- **context switch** \rightarrow **round counter++**
- first thread starts with non-deterministic memory contents
 - other threads continue with content left by predecessor



LR sequentialization

- considers only round-robin schedules with k rounds
 - **thread** \rightarrow **function**, run to completion
- global memory copy for each round
 - **scalar** \rightarrow **array**
- **context switch** \rightarrow **round counter++**
- first thread starts with non-deterministic memory contents
 - other threads continue with content left by predecessor
- checker prunes away inconsistent simulations
 - **assume**($S_{k+1,0} == S_{k,n}$);
 - requires second set of memory copies
 - errors can only be checked at end of simulation
 - o requires explicit error checks



LR sequentialization - implementation

```
//shared vars
typeg1 g1; typeg2 g2; ...

//thread functions
t(){
    typex1 x1; typex2 x2; ...
    stmt1 ;
    stmt2 ;
    ...
} ...

main(){
    ...
}
```

```
//shared vars
typeg1 g1[K]; typeg2 g2[K]; ...
uint round=0; bool ret=0; //aux vars

// context-switch simulation
cs() {
    unsigned int j; j= nondet();
    assume(round +j < K); round+=j;
    if (round==K-1 && nondet()) ret=1;
}

//thread functions
t(){
    typex1 x1; typex2 x2; ...
    cs(); if (ret) return; stmt1[round];
    cs(); if (ret) return; stmt2[round];
    ...
} ...

main_thread(){
    ...
}

main(){ ... } //next slide
```

LR sequentialization - implementation

```
main(){
    typeg1 _g1[K]; typeg2 _g2[K]; ...
    // first thread starts with non-deterministic memory contents
    for (i=1; i<K; i++){
        _g1[i] = g1[i] = nondet();
        _g2[i] = g2[i] = nondet();
        ...
    }
    // thread simulations
    t[0] = main_thread;
    round_born[0] = 0; is_created[0] = 1;
    for (i=0; i<N; i++){
        if(is_created[i]){
            ret=0;
            round = round_born[i];
            t[i](); }
    }
    // consistency check
    for (i=0; i<K-1; i++){
        assume(_g1[i+1] == g1[i]);
        assume(_g2[i+1] == g2[i]);
        ...
    }
    // error detection
    assert(err == 0); }
```

LR sequentialization - implementation

- **Corral** (SMT-based analysis for Boogie programs)
 - [Lal–Qadeer–Lahiri, CAV'12]
 - [Lal–Qadeer, FSE'14]
- **CSeq** (code-to-code translation for C + pthreads)
 - [Fischer–Inverso–Parlato, ASE'13]
- **Rek** (for Real-time Embedded Software Systems)
 - [Chaki–Gurfinkel–Strichman, FMCAD'11]
- **Storm**: implementation for C programs
 - [Lahiri–Qadeer–Rakamaric, CAV'09]
 - [Rakamaric, ICSE'10]

Summary

- Described **typical architectures** employed by BMC tools (e.g., CBMC, ESBMC and LLBMC):
 - language support, built-in safety checks, and non-deterministic modelling
 - general approach to verify programs, including program transformations and bit-blasting

Summary

- Described **typical architectures** employed by BMC tools (e.g., CBMC, ESBMC and LLBMC):
 - language support, built-in safety checks, and non-deterministic modelling
 - general approach to verify programs, including program transformations and bit-blasting
- Introduced the difficulties to write concurrent programs, typical **concurrency errors** and **communication models**

Summary

- Described **typical architectures** employed by BMC tools (e.g., CBMC, ESBMC and LLBMC):
 - language support, built-in safety checks, and non-deterministic modelling
 - general approach to verify programs, including program transformations and bit-blasting
- Introduced the difficulties to write concurrent programs, typical **concurrency errors** and **communication models**
- Presented state-of-the-art concurrency verification approaches, including: **explicit schedule exploration** and **sequentialization**