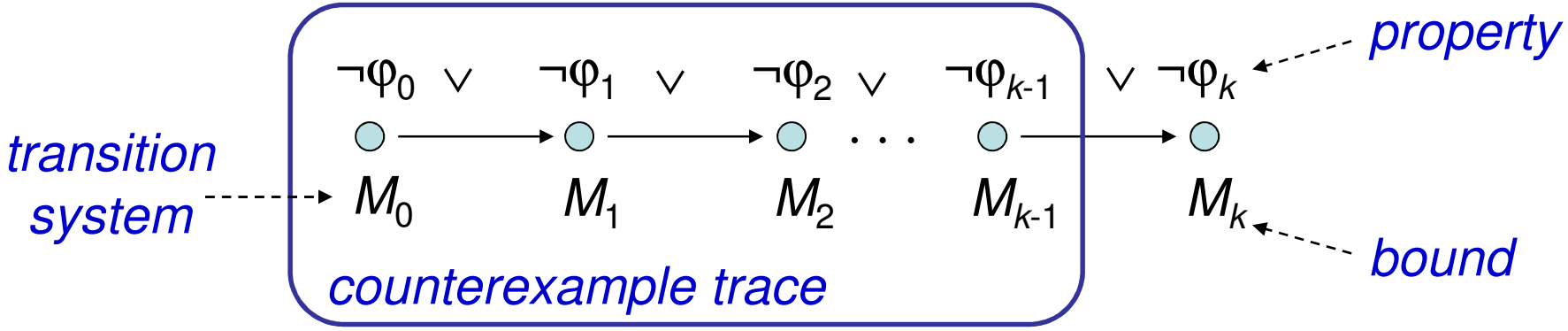
Verifying Multi-threaded Software using SMT-based Context-Bounded Model Checking

Lucas Cordeiro and Bernd Fischer
lucascordeiro@ufam.edu.br

Bounded Model Checking (BMC)

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



- transition system *M* unrolled *k* times
  - for programs: unroll loops, unfold arrays, …
- translated into verification condition ψ such that

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

BMC of Multi-threaded Software

- concurrency bugs are tricky to **reproduce/debug** because they usually occur under specific thread interleavings
  - most common errors: *67% related to atomicity and order violations, 30% related to deadlock* [Lu et al.'08]
- problem: the number of interleavings grows exponentially with the number of threads and program statements
  - context switches among threads increase the number of possible executions
- two important observations help us:
  - concurrency bugs are shallow [Qadeer&Rehof'05]
  - SAT/SMT solvers produce unsatisfiable cores that allow us to remove logic that is not relevant

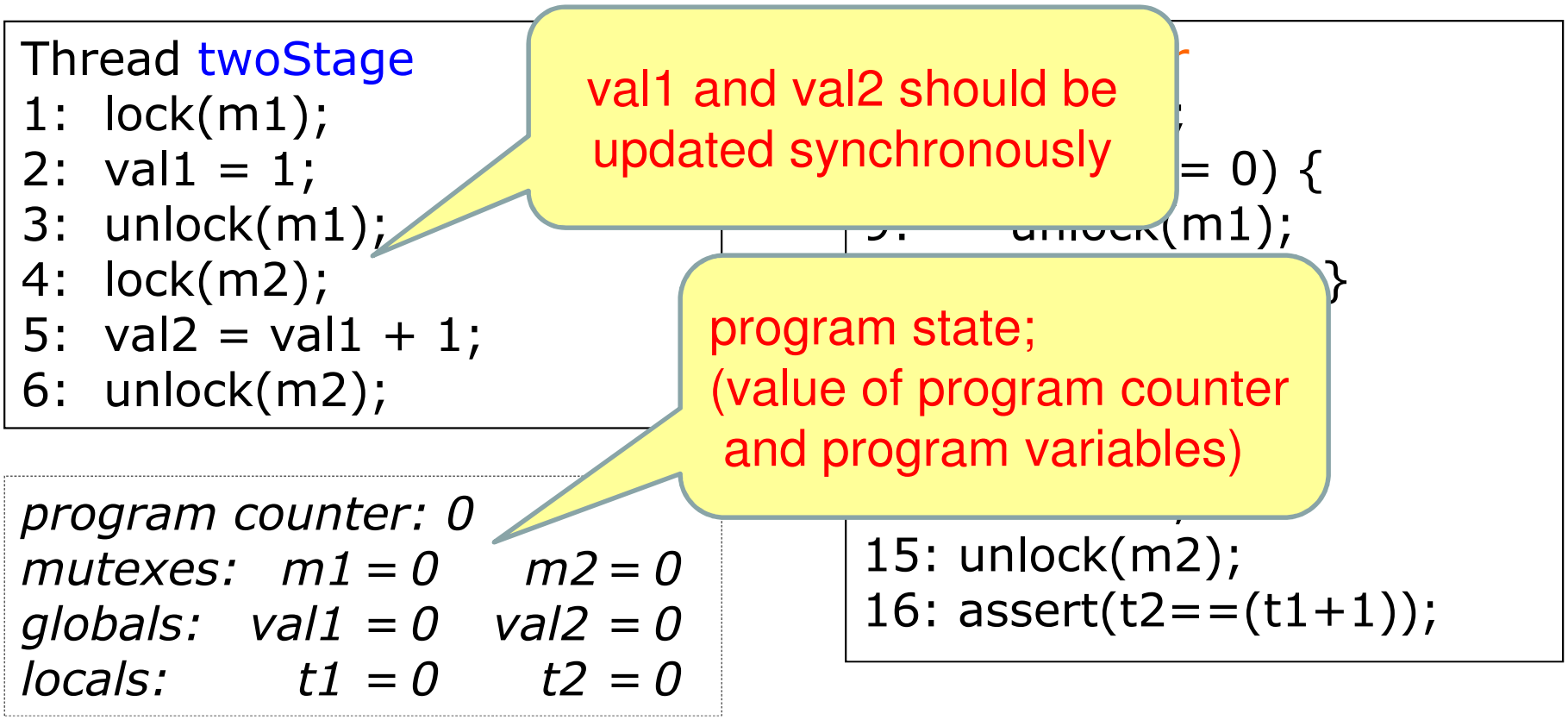
Objective of this work

Exploit SMT to improve BMC of multi-threaded software

- exploit SMT solvers to:
  - prune the *property and data dependent* search space (non-chronological backtracking and conflict clauses learning)
  - remove interleavings that are not relevant by analyzing the proof of unsatisfiability
- propose three approaches to SMT-based BMC:
  - *lazy exploration* of the interleavings
  - *schedule guards* to encode all interleavings
  - *underapproximation and widening (UW) [Grumberg&et al.'05]*
- implement these approaches in ESBMC and evaluate them using multi-threaded applications

Lazy exploration of interleavings

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



Lazy exploration of interleavings

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

interleaving #1: 1

```
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*
globals: val1 = 0 val2 = 0
locals: t1 = 0 t2 = 0

Lazy exploration of interleavings

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

interleaving #1: 1-2

```
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
globals: ***val1 = 1*** *val2 = 0*
locals: t1 = 0 t2 = 0

Lazy exploration of interleavings

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

interleaving #1: 1-2-3

```
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*
globals: val1 = 1 val2 = 0
locals: t1 = 0 t2 = 0

Lazy exploration of interleavings

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

interleaving #1: 1-2-3–7

```
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*
globals: val1 = 1 val2 = 0
locals: t1 = 0 t2 = 0

Lazy exploration of interleavings

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

interleaving #1: 1-2-3–7-8

```
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
globals: val1 = 1 val2 = 0
locals: t1 = 0 t2 = 0

Lazy exploration of interleavings

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

interleaving #1: 1-2-3–7-8-11

```
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
globals: val1 = 1 val2 = 0
locals: ***t1 = 1*** *t2 = 0*

Lazy exploration of interleavings

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

interleaving #1: 1-2-3–7-8-11-12

```
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*
globals: val1 = 1 val2 = 0
locals: t1 = 1 t2 = 0

Lazy exploration of interleavings

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

interleaving #1: 1-2-3–7-8-11-12–4

```
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***
globals: val1 = 1 val2 = 0
locals: t1 = 1 t2 = 0

Lazy exploration of interleavings

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

interleaving #1: 1-2-3–7-8-11-12–4-5

```
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
globals: val1 = 1 ***val2 = 2***
locals: t1 = 1 t2 = 0

Lazy exploration of interleavings

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

interleaving #1: 1-2-3–7-8-11-12–4-5-6

```
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***
globals: val1 = 1 val2 = 2
locals: t1 = 1 t2 = 0

Lazy exploration of interleavings

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

interleaving #1: 1-2-3–7-8-11-12–4-5-6–13

```
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***
globals: val1 = 1 val2 = 2
locals: t1 = 1 t2 = 0

Lazy exploration of interleavings

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

interleaving #1: 1-2-3–7-8-11-12–4-5-6–13-14

```
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 = 1 m2 = 1
globals: val1 = 1 val2 = 2
locals: t1 = 1 ***t2 = 2***

Lazy exploration of interleavings

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

interleaving #1: 1-2-3–7-8-11-12–4-5-6–13-14-15

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

CS1

CS2

CS3

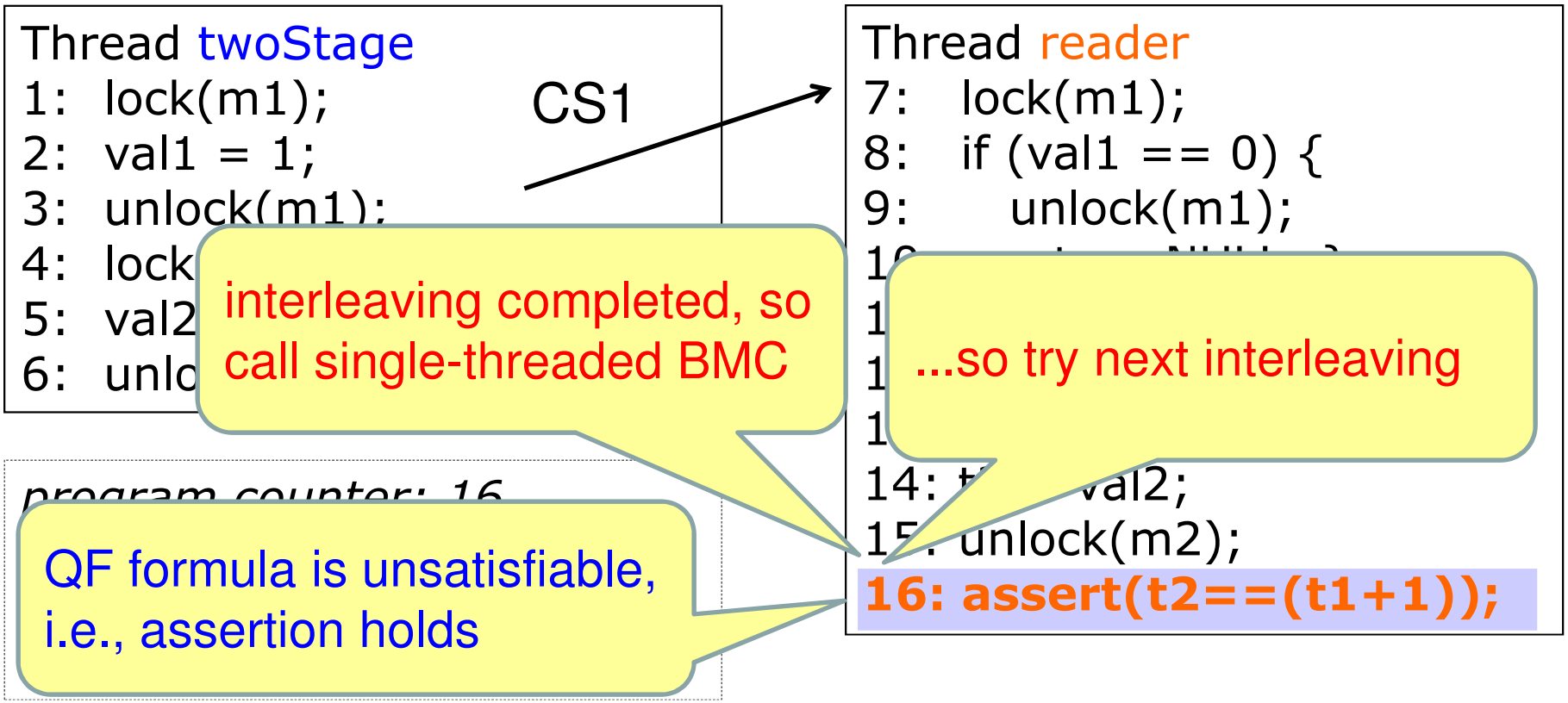
```
Thread reader
7:   lock(m1);
8:   if (val1 == 0) {
9:      unlock(m1);
10:    return NULL; }
11: t1 = val1;
12: unlock(m1);
13: lock(m2);
14: t2 = val2;
15: unlock(m2);
16: assert(t2==(t1+1));
```

program counter: 15
mutexes: m1 = 1 ***m2 = 0***
globals: val1 = 1 val2 = 2
locals: t1 = 1 t2 = 2

Lazy exploration of interleavings

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

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



Lazy exploration of interleavings

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

interleaving #2:

```
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
globals: val1 = 0 val2 = 0
locals: t1 = 0 t2 = 0

Lazy exploration of interleavings

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

interleaving #2: 1-2-3

```
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
globals: val1 = 1 val2 = 0
locals: t1 = 0 t2 = 0

Lazy exploration of interleavings

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

interleaving #2: 1-2-3–7

```
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*
globals: val1 = 1 val2 = 0
locals: t1 = 0 t2 = 0

Lazy exploration of interleavings

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

interleaving #2: 1-2-3–7-8-11-12-13-14-15-16

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

CS1

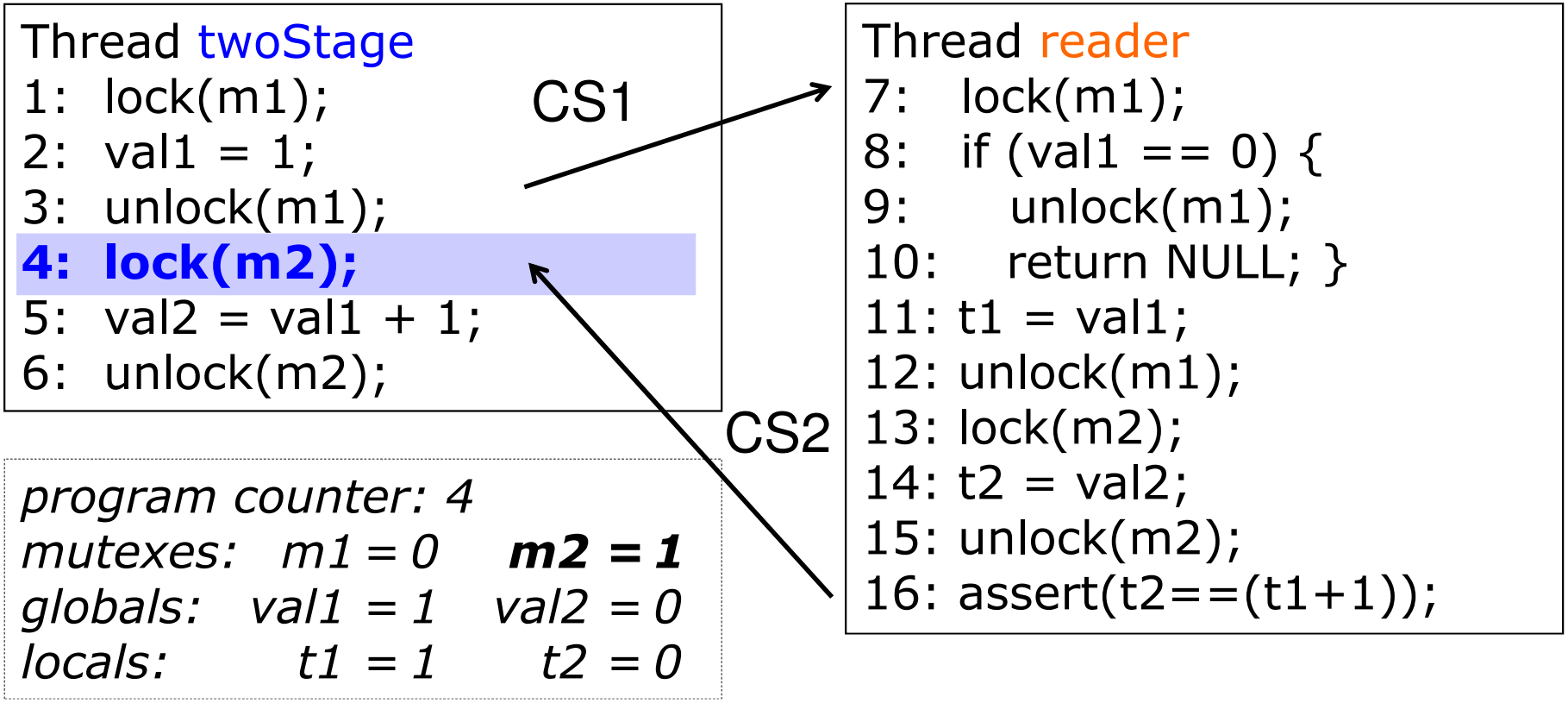
```
Thread reader
7:   lock(m1);
8:   if (val1 == 0) {
9:      unlock(m1);
10:    return NULL; }
11: t1 = val1;
12: unlock(m1);
13: lock(m2);
14: t2 = val2;
15: unlock(m2);
16: assert(t2==(t1+1));
```

program counter: 16
mutexes: m1 = 0 m2 = 0
globals: val1 = 1 val2 = 0
locals: t1 = 1 t2 = 0

Lazy exploration of interleavings

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

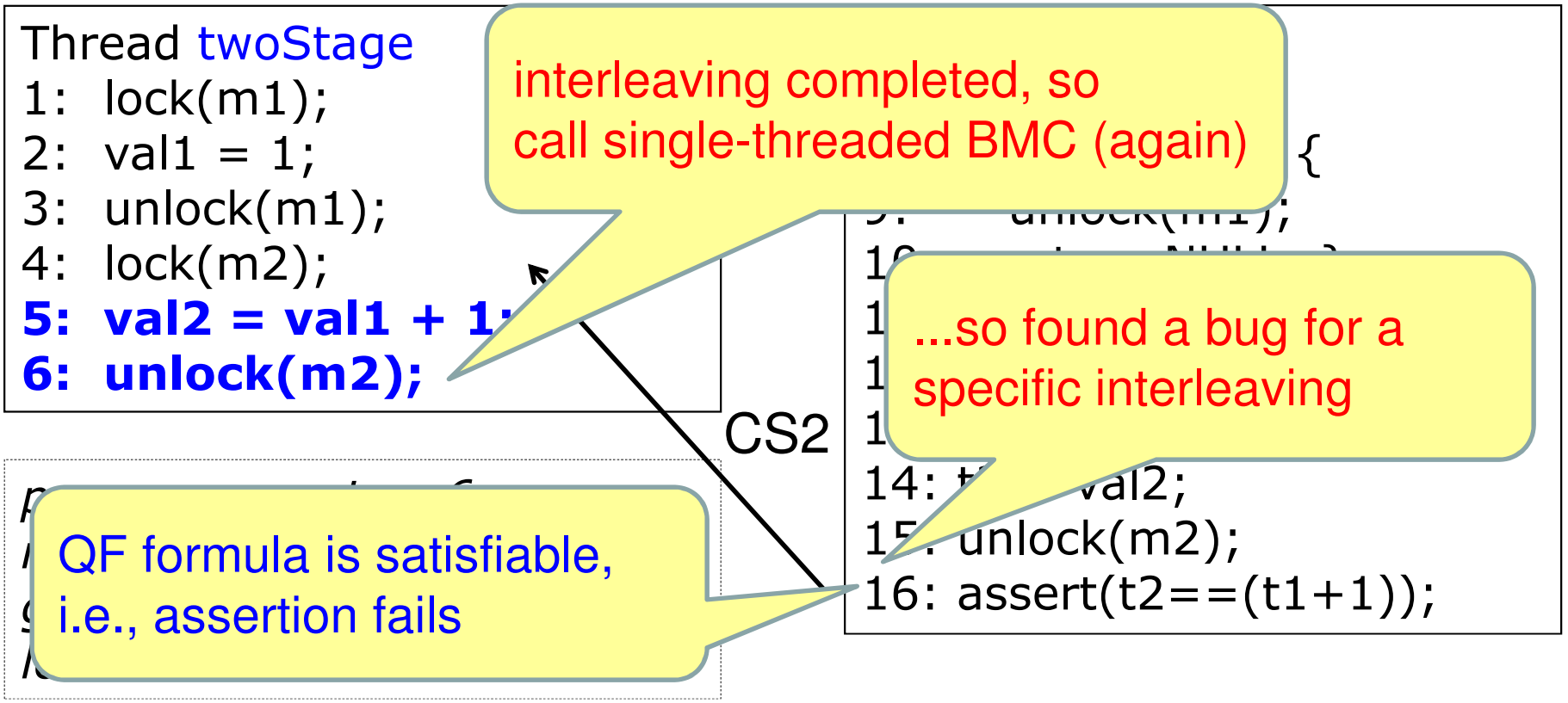
interleaving #2: 1-2-3–7-8-11-12-13-14-15-16–4



Lazy exploration of interleavings

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

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



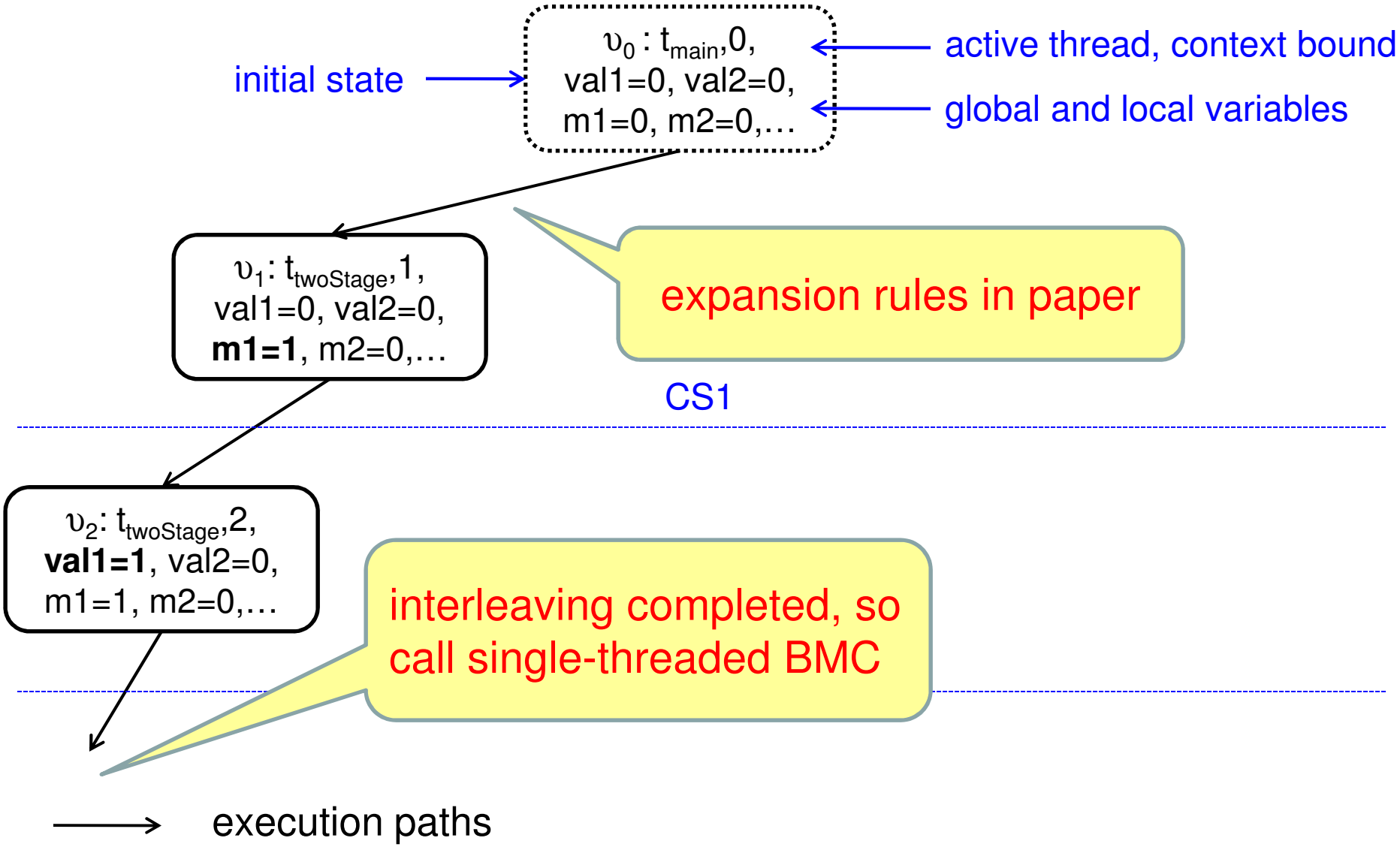
Lazy exploration of interleavings

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

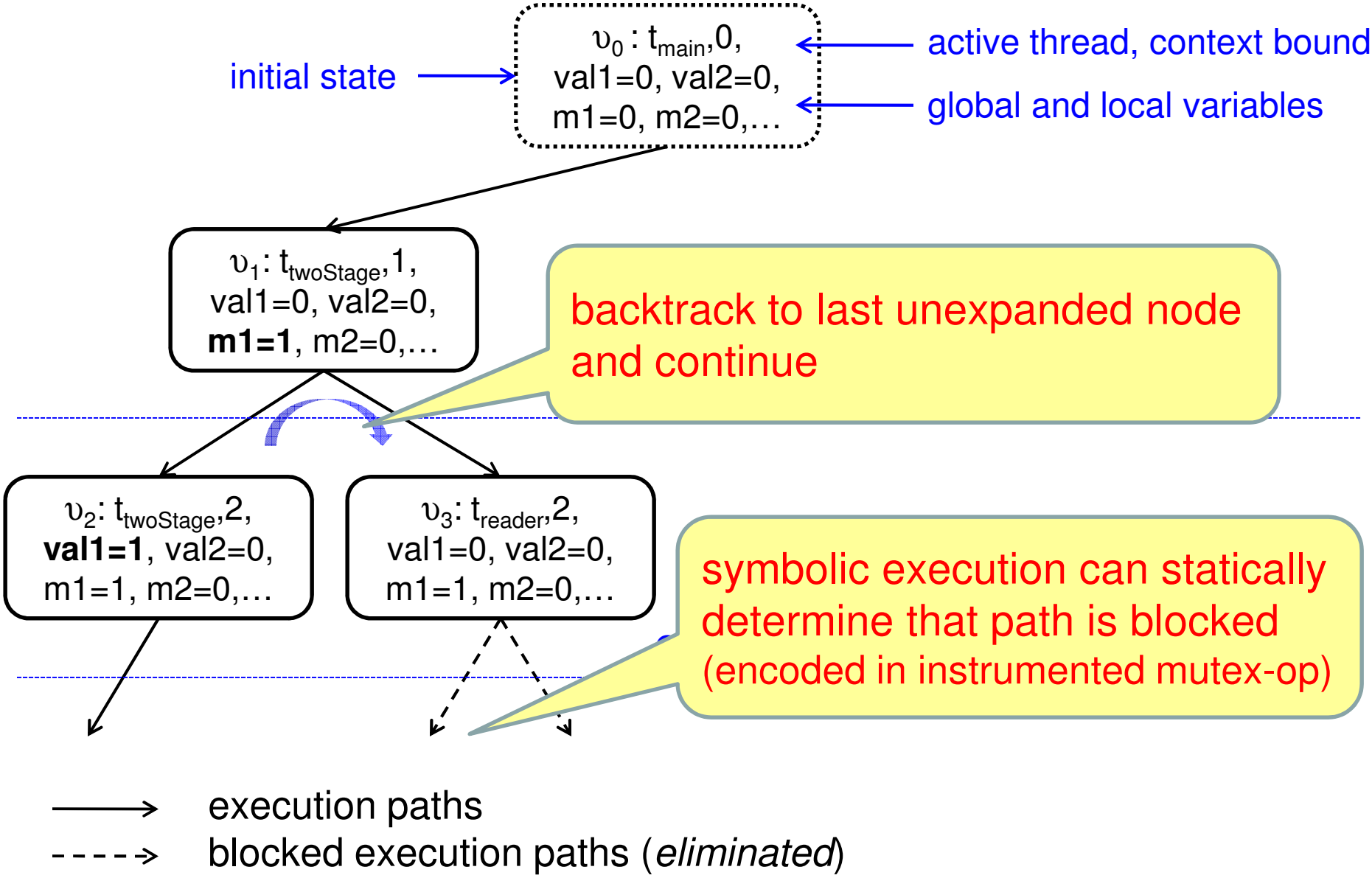
... combines

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

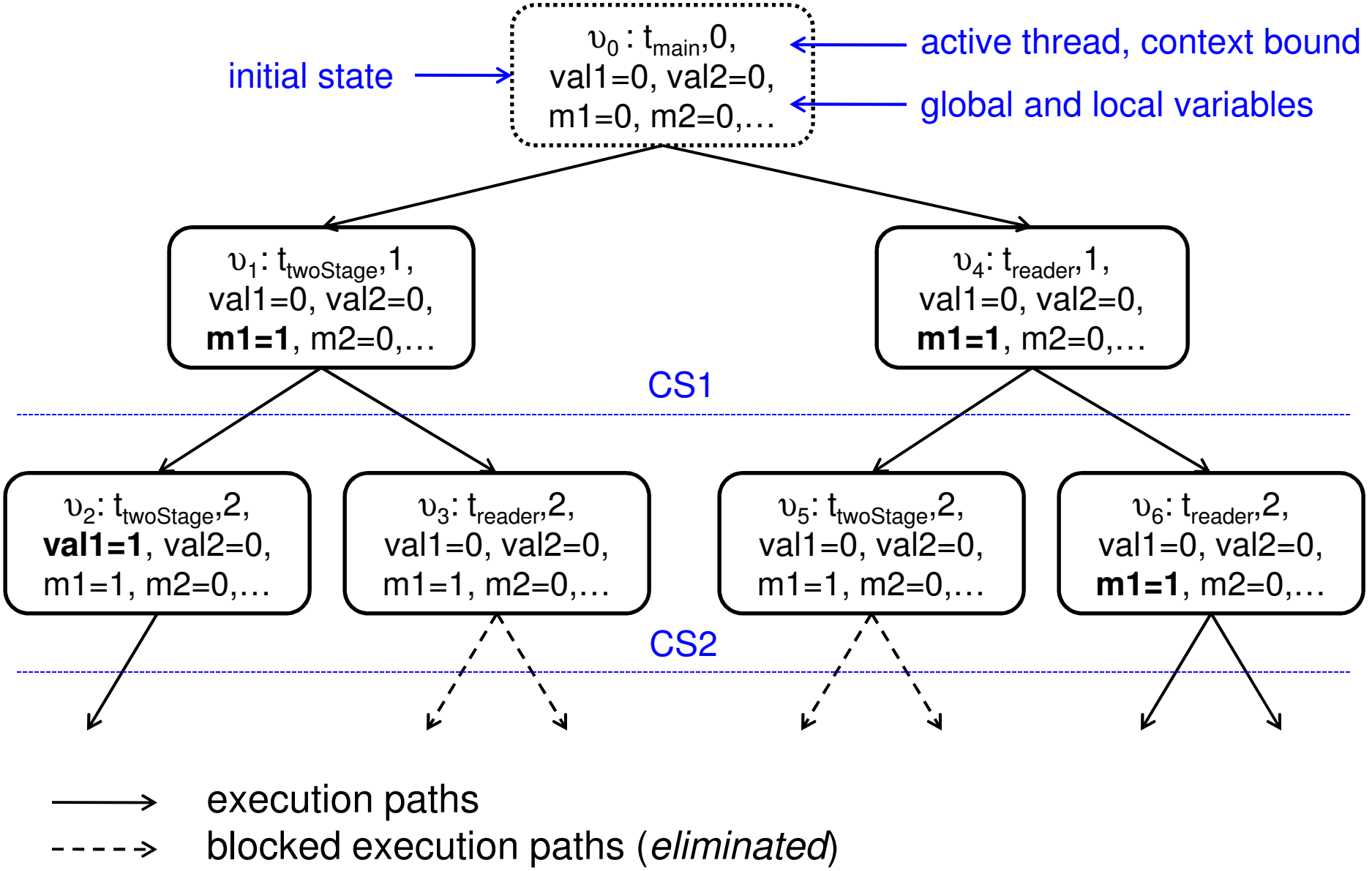
Lazy exploration of interleavings – Reachability Tree



Lazy exploration of interleavings – Reachability Tree



Lazy exploration of interleavings – Reachability Tree



Lazy approach is naïve but useful

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

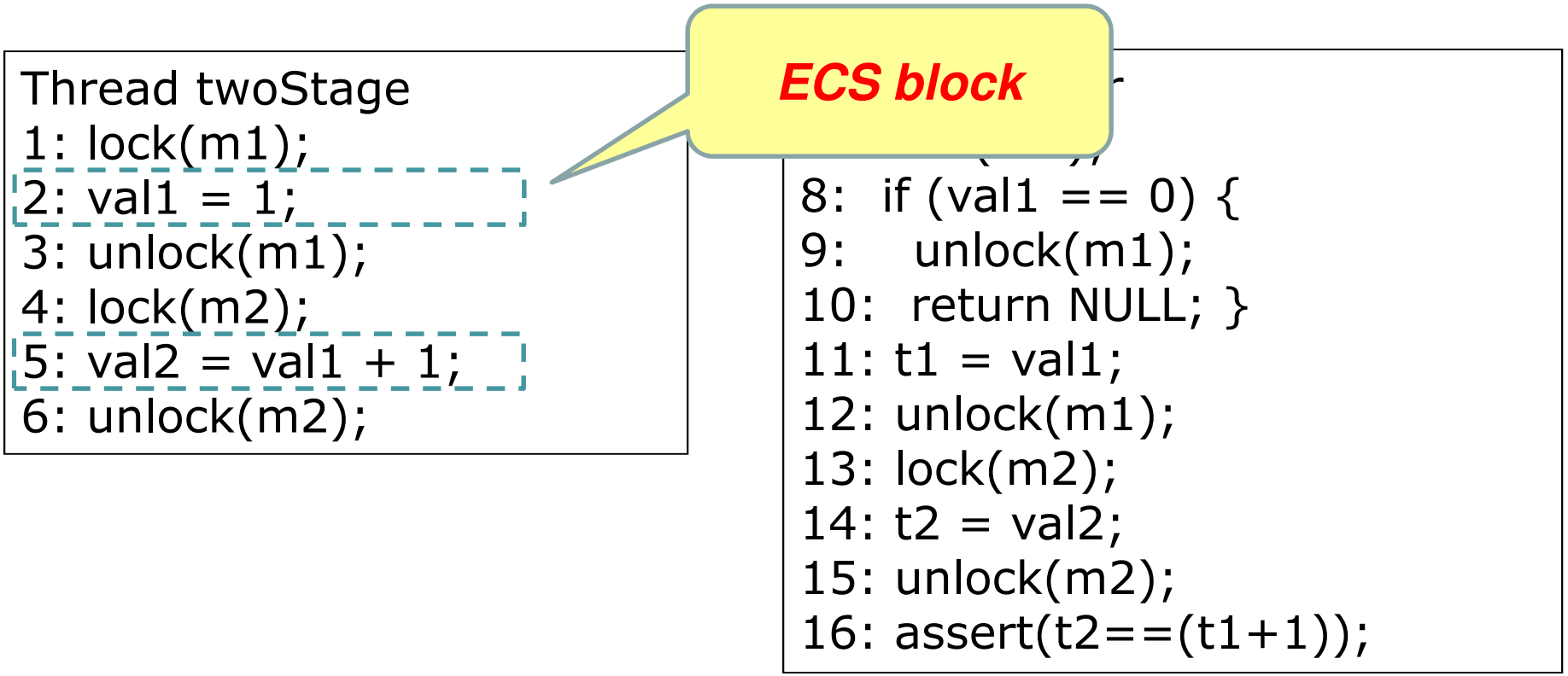
- explore reachability tree in same way as lazy approach
- ... but call SMT solver only once
- add a ***schedule guard*** ts_i for each context switch block i ($0 < ts_i \leq$ #threads)
  - record in which order the *scheduler* has executed the program
  - SMT solver determines the order in which threads 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:



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

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

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*

Schedule Recording – Interleaving #1

statements: 1-2

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

reader-ECS:

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

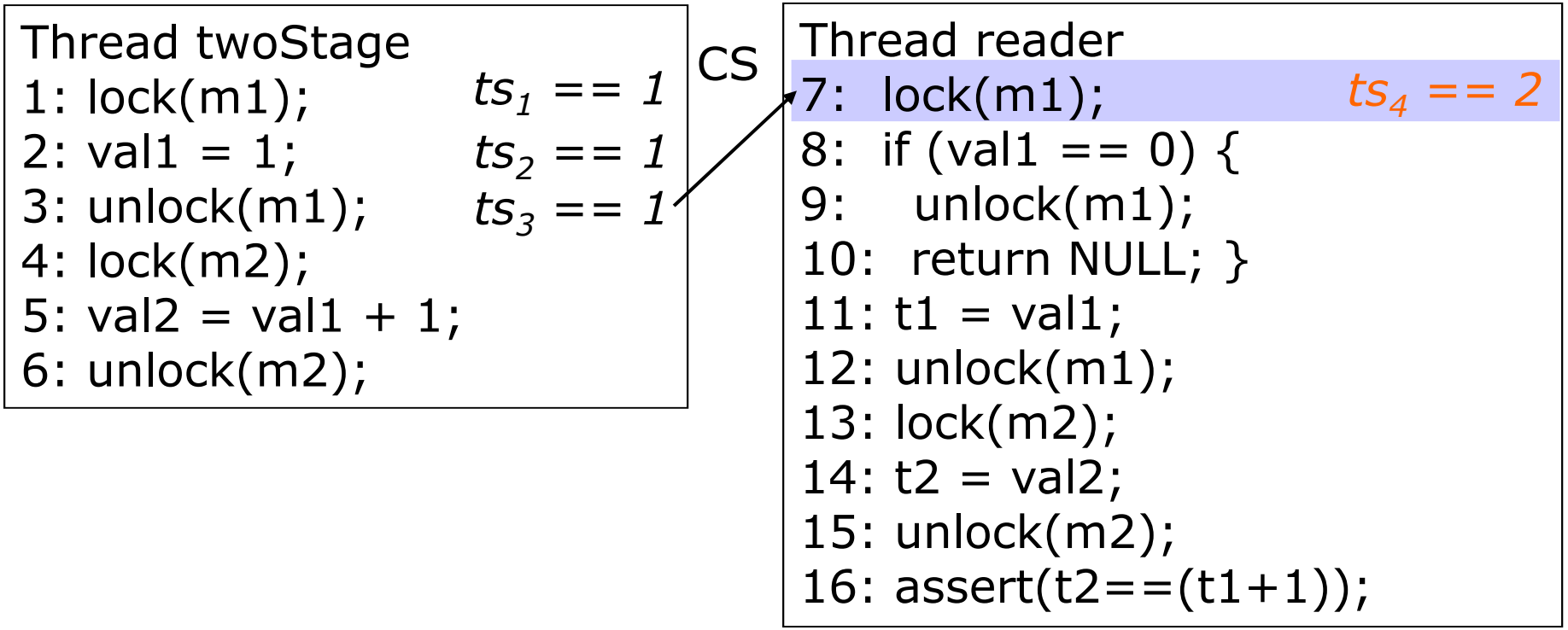
```
Thread reader
7:  lock(m1);
8:  if (val1 == 0) {
9:    unlock(m1);
10:  return NULL; }
11: t1 = val1;
12: unlock(m1);
13: lock(m2);
14: t2 = val2;
15: unlock(m2);
16: assert(t2==(t1+1));
```

Schedule Recording – Interleaving #1

statements: 1-2-3-7

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

reader-ECS: (7,4)

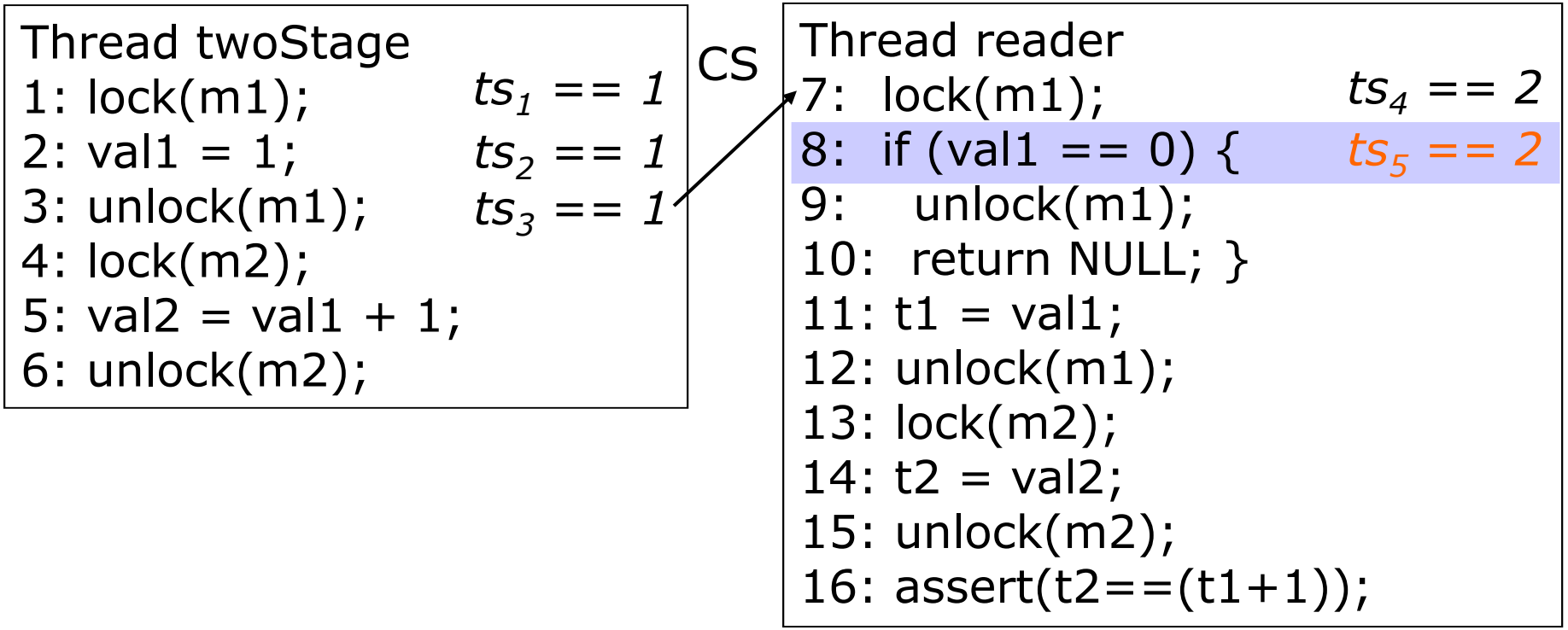


Schedule Recording – Interleaving #1

statements: 1-2-3-7-8

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

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

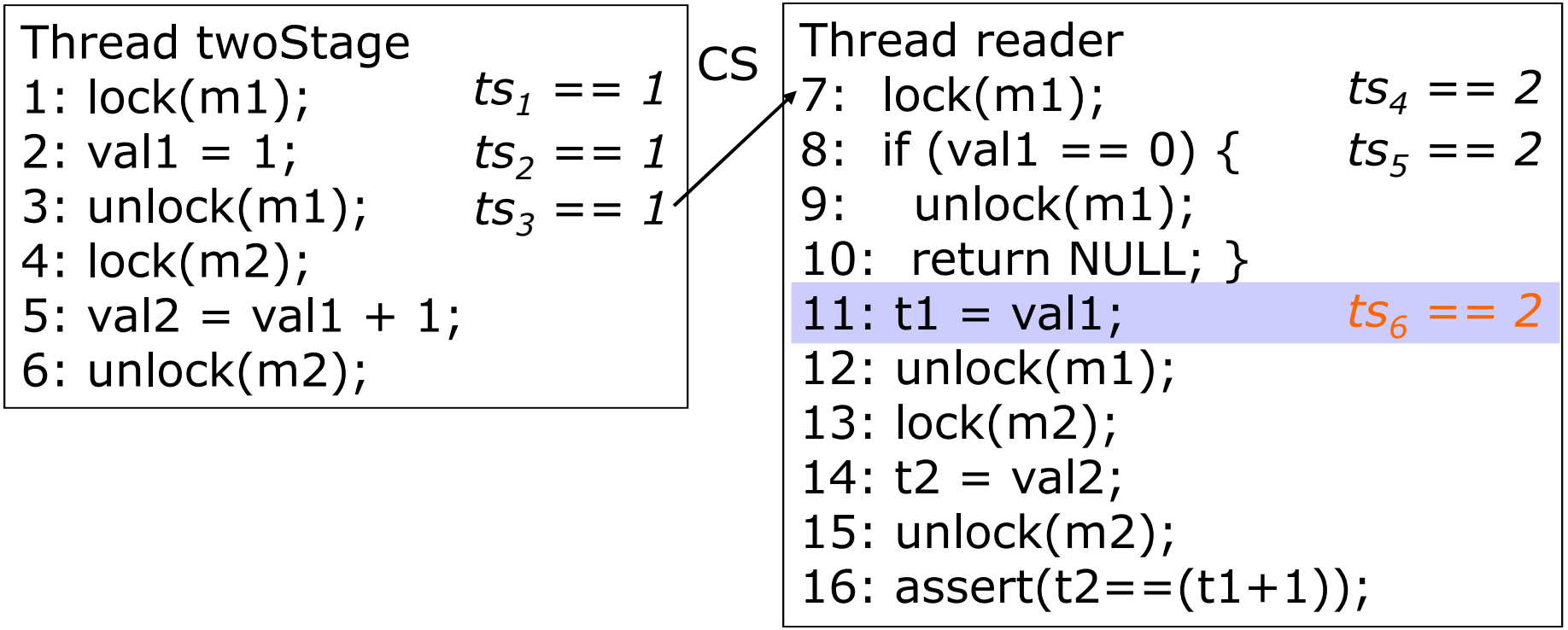


Schedule Recording – Interleaving #1

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

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

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

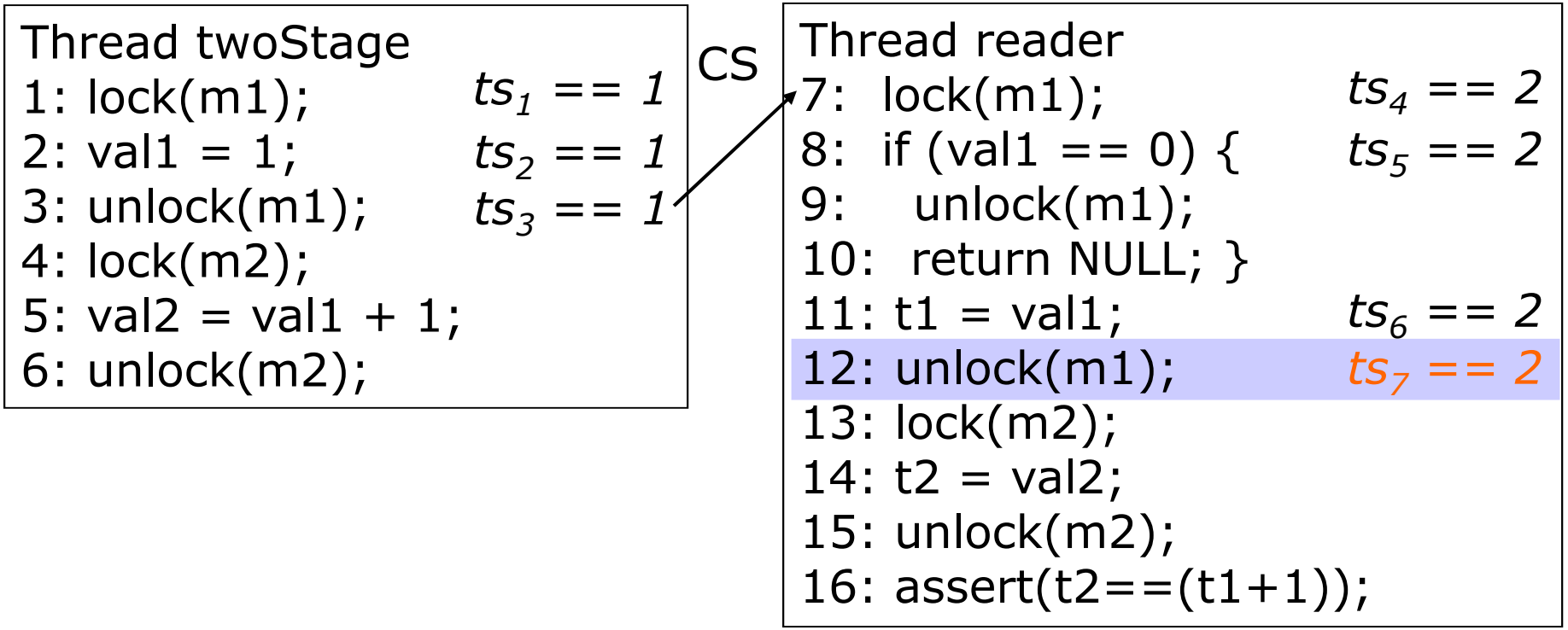


Schedule Recording – Interleaving #1

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)

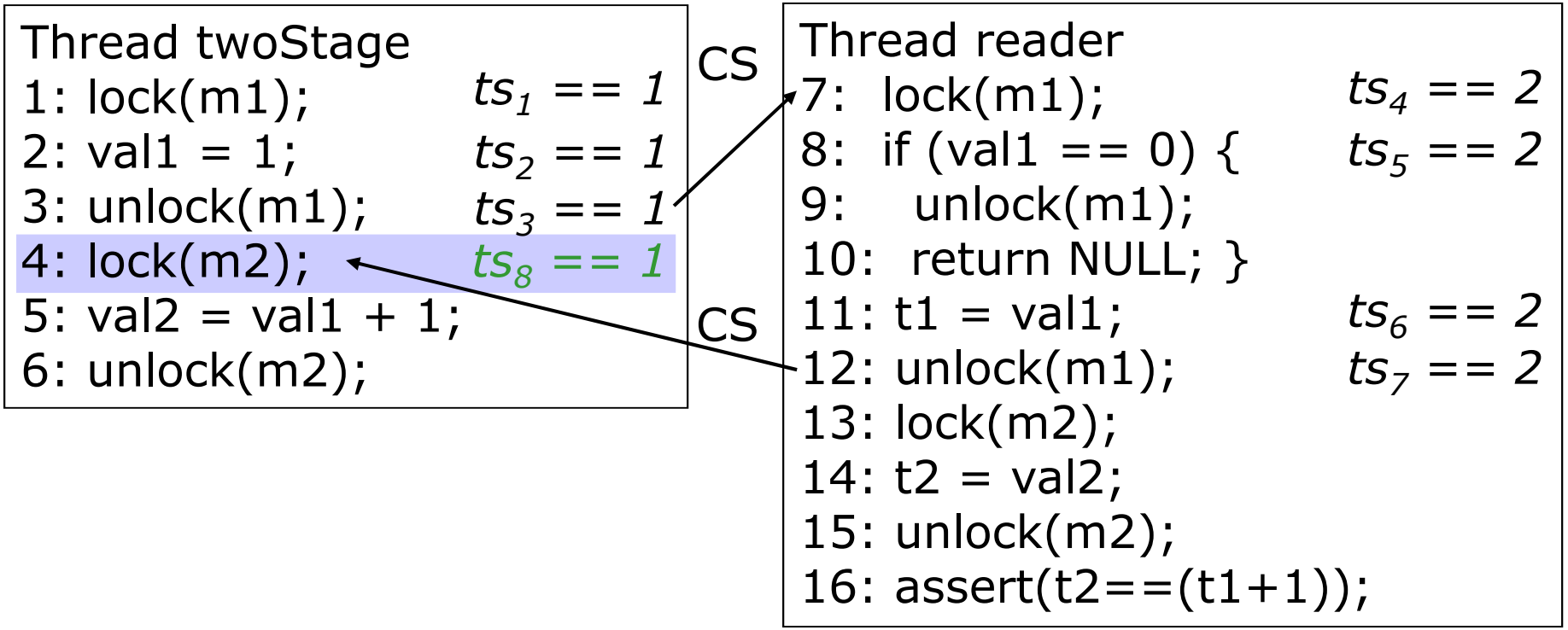


Schedule Recording – Interleaving #1

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)

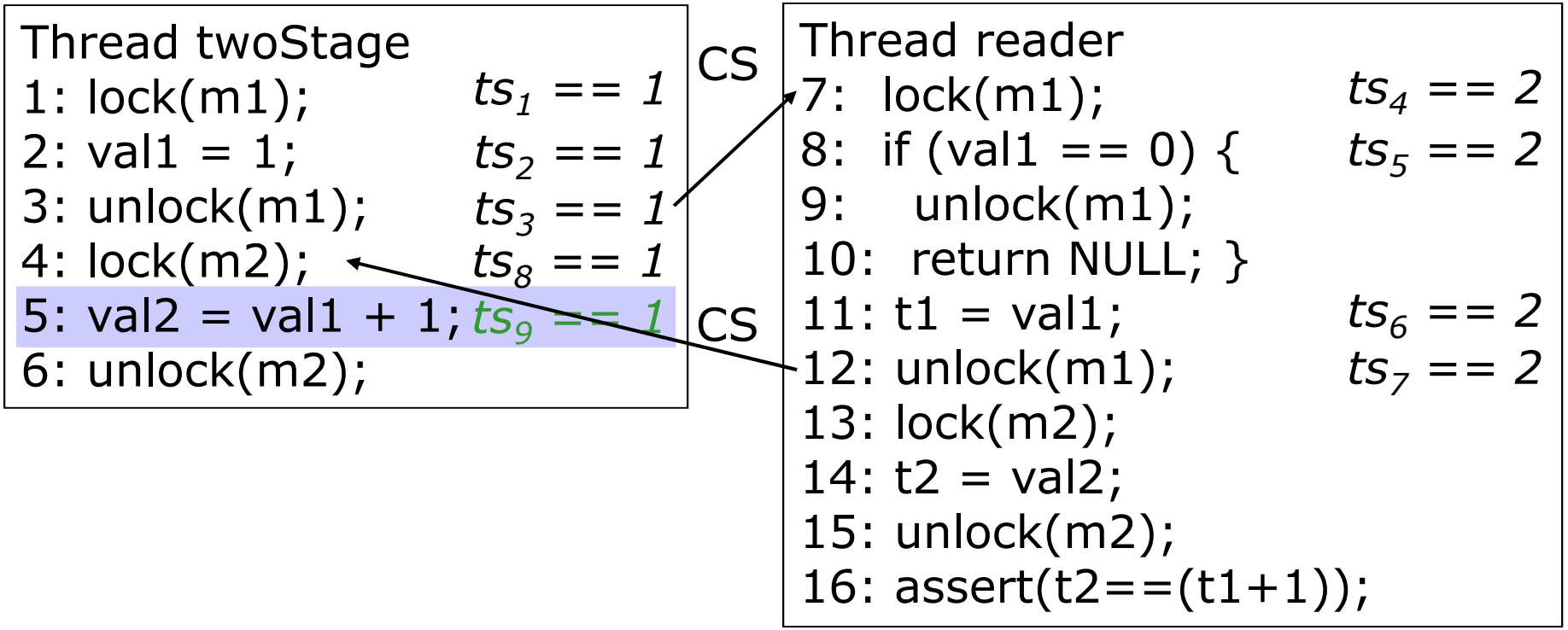


Schedule Recording – Interleaving #1

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)

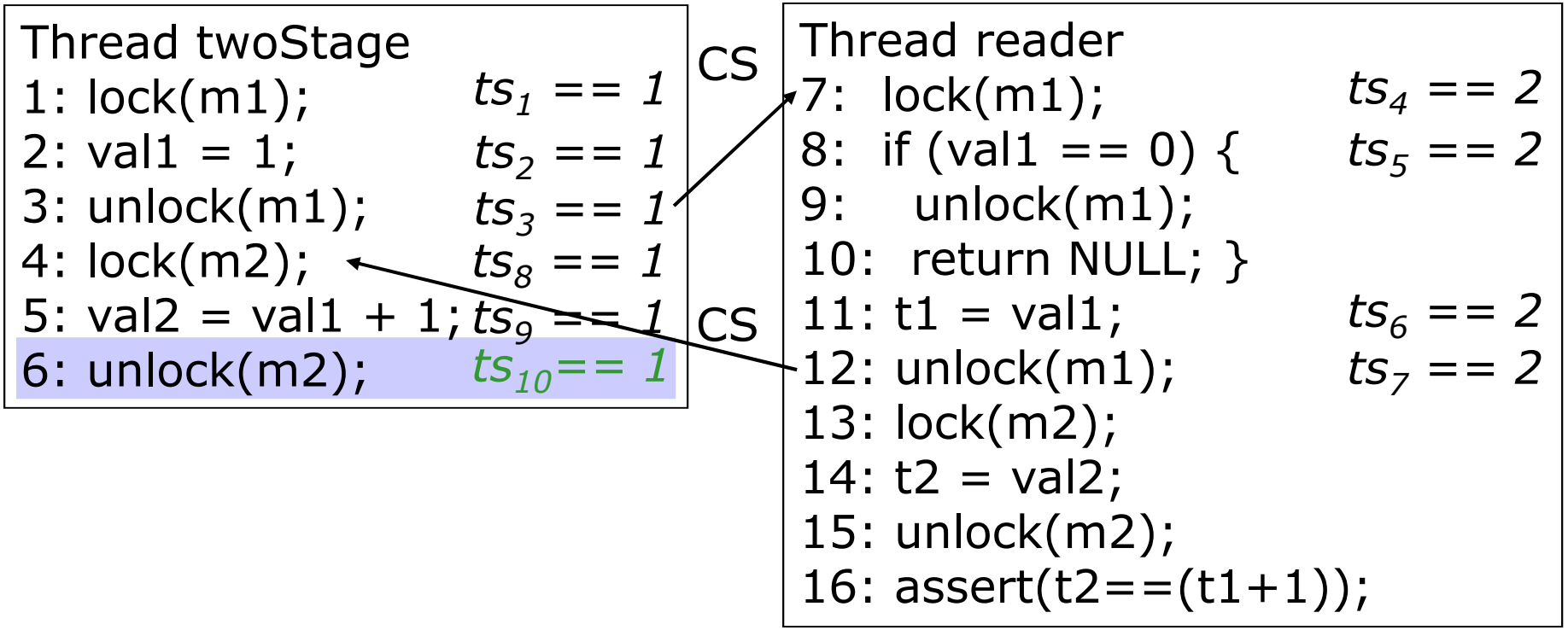


Schedule Recording – Interleaving #1

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)

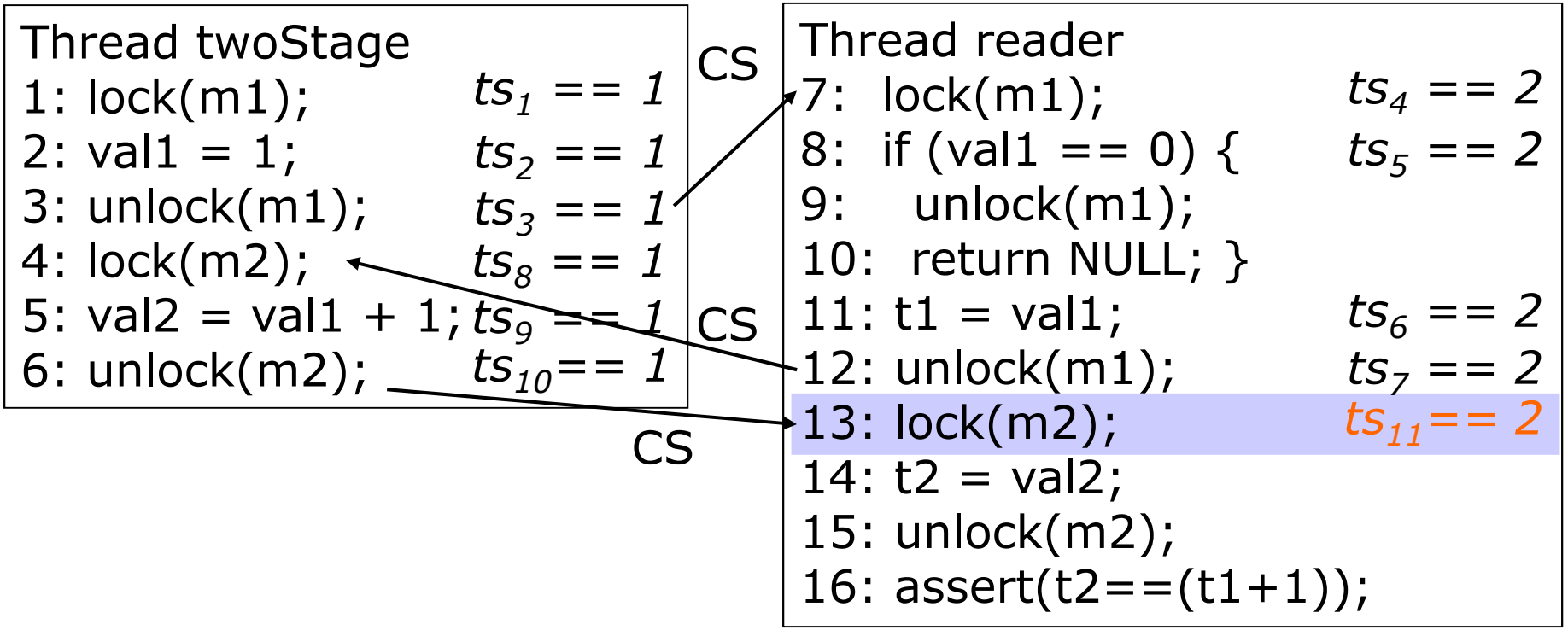


Schedule Recording – Interleaving #1

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)

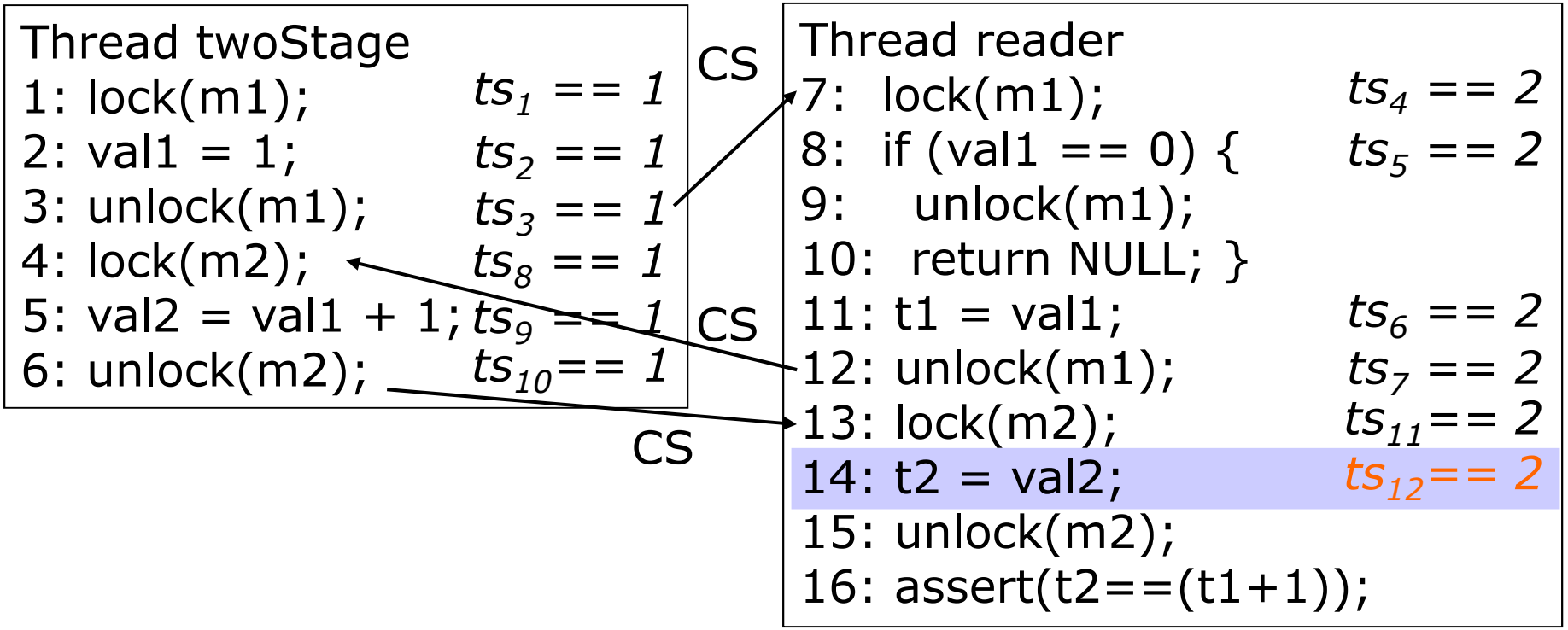


Schedule Recording – Interleaving #1

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)

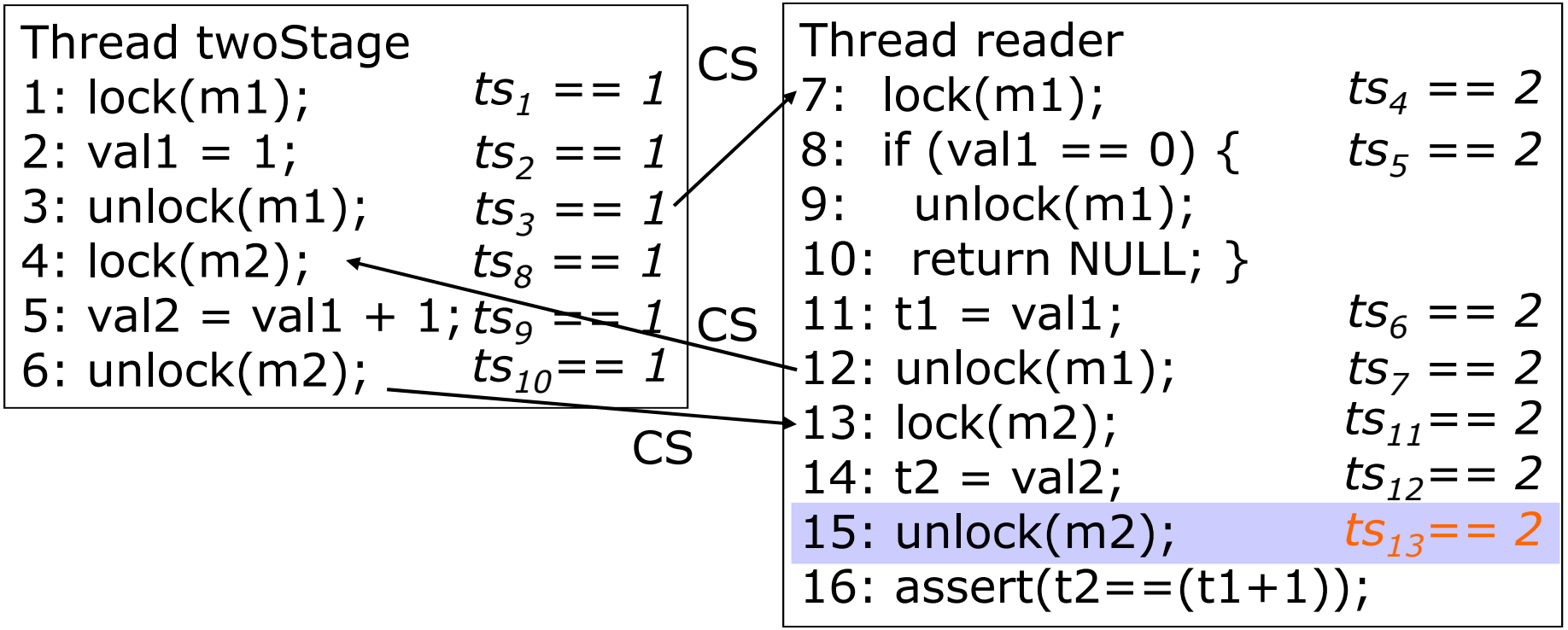


Schedule Recording – Interleaving #1

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)

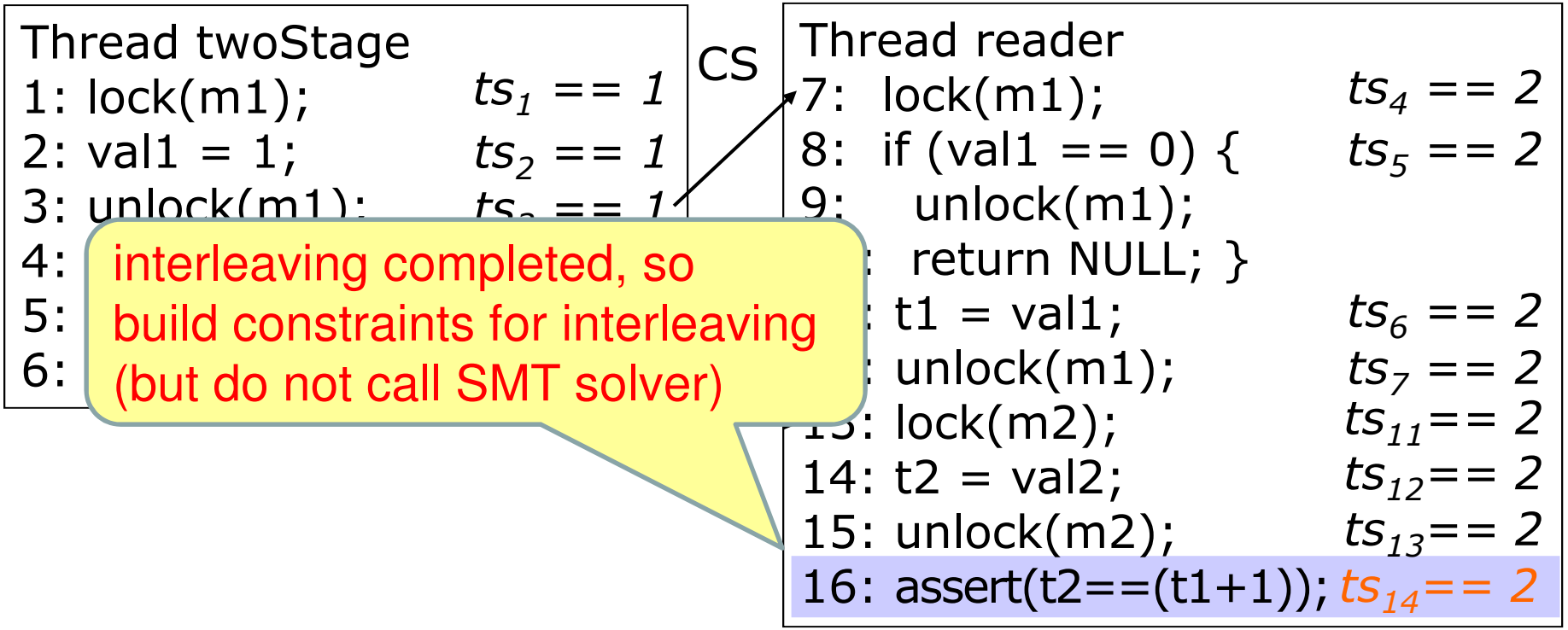


Schedule Recording – Interleaving #1

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)

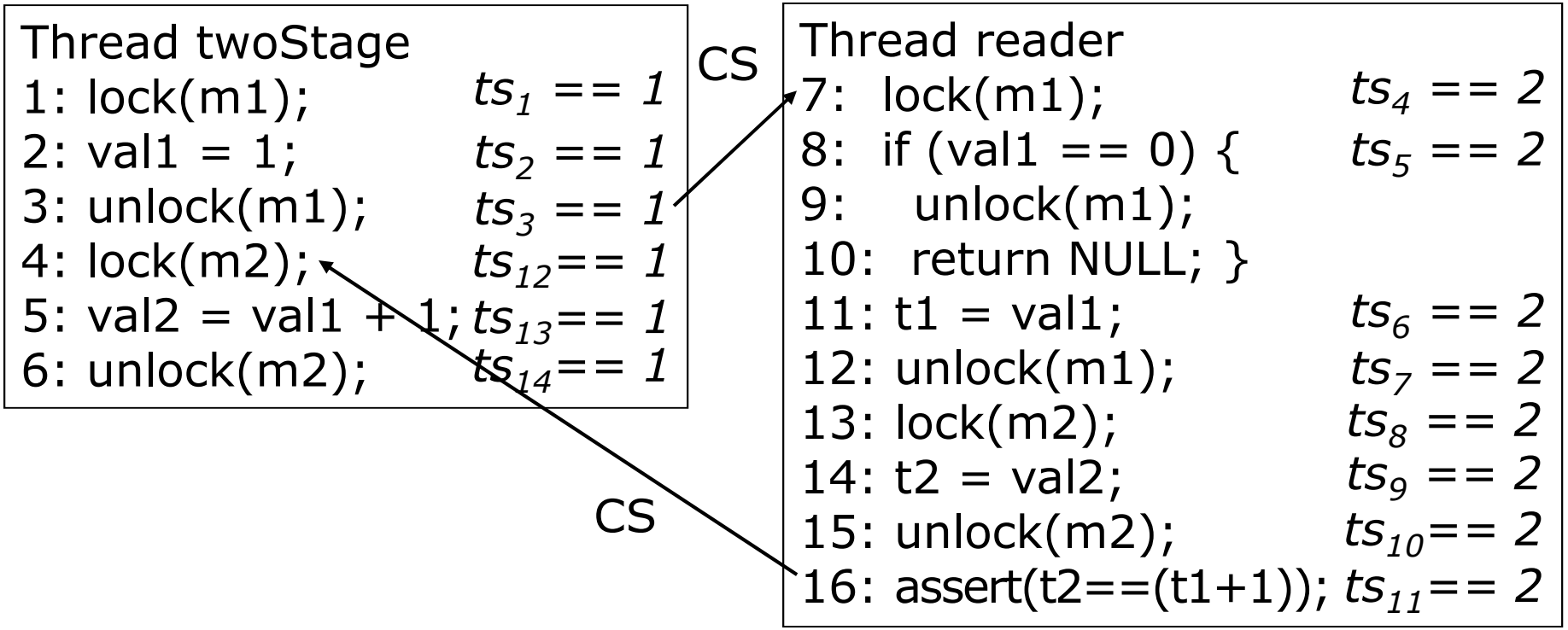


Schedule Recording – Interleaving #2

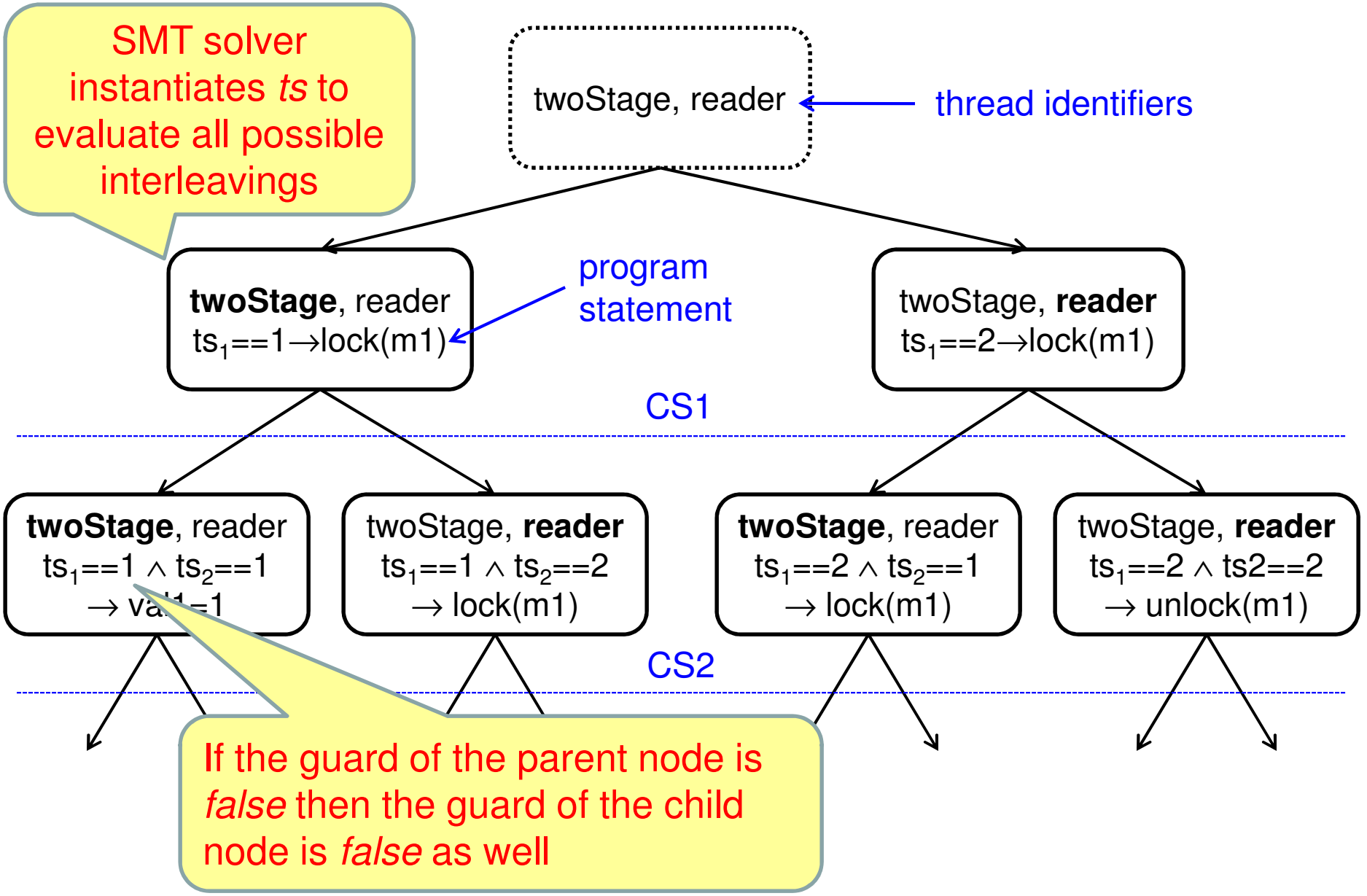
statements: 1-2-3-7-8-11-12-13-14-15-16-4-5-6

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



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
    - ▷ *bound the number of context switches*
    - ▷ *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

Under-approximation and Widening

Idea: check models with an increased set of allowed interleavings [Grumberg&et al.'05]

- start from a single interleaving (under-approximation) and widen the model by adding more interleavings incrementally

Main steps of the algorithm:

1. encode control literals ($cl_{i,j}$) into the verification condition ψ
   - ▷ $cl_{i,j}$ where *i* is the ECS block number and *j* is the thread identifier
2. check the satisfiability of ψ (stop if ψ is satisfiable)
3. extract proof objects generated by the SMT solver
4. check whether the proof depends on the control literals (stop if the proof does not depend on the control literals)
5. remove literals that participated in the proof and go to step 2

UW Approach: Running Example

- use the same guards as in the schedule recording approach as control literals
  - but here the schedule is updated based on the information extracted from the proof

Thread twoStage	
1: lock(m1);	$cl_{1,twoStage} \rightarrow ts_1 == 1$
2: val1 = 1;	$cl_{2,twoStage} \rightarrow ts_2 == 1$
3: unlock(m1);	$cl_{3,twoStage} \rightarrow ts_3 == 1$
4: lock(m2);	$cl_{8,twoStage} \rightarrow ts_8 == 1$
5: val2 = val1 + 1;	$cl_{9,twoStage} \rightarrow ts_9 == 1$
6: unlock(m2);	$cl_{10,twoStage} \rightarrow ts_{10} == 1$

- reduce the number of control points from m x n to e x n
  - m is the number of program statements; n is the number of threads, and e is the number of ECS blocks

Evaluation

Comparison of the Approaches

- Goal: compare efficiency of the proposed approaches
  - lazy exploration
  - schedule recording
  - underapproximation and widening
- Set-up:
  - ESBMC v1.15.1 together with the SMT solver Z3 v2.11
  - support the logics *QF_AUFBV* and *QF_AUFLIRA*
  - standard desktop PC, time-out 3600 seconds

About the benchmarks

	Module	#L	#T	#P	B	#C	
		81	26	47	26	2	Fran
2	fsbench_bad		27		27	2	ipani file system with array
3	indexer_ok	77	15		29	4	hash
4	aget-0.4_bad	1233					eaded download tor
5	bzip2smp_ok	6366					mpressor
6	reorder_bad	84	10	7	10	11	Data race
7	twostage_bad	128	100	13	100	4	Atomicity violation
8	wronglock_bad	110	8	8	8	8	Wrong lock acquisition order
9	exStbHDMI_ok	1060	2	24	16	20	Configures the HDMI device
10	exStbLED_ok	425	2	45	10	10	Front panel LED display
11	exStbThumb_bad	1109	2	249	2	1	Demonstrate how thumbnail images can be manipulated
12	micro_10_ok	1171	10	10	1	17	synthetic micro-benchmark

lines of code

number of threads

the number of properties checked

number of BMC unrolling steps

number of context switches

About the benchmarks

Inspect benchmark suite

	Module	#				C	Description
1	**fsbench_ok**					**2**	**Frangipani file system**
2	**fsbench_bad**					**2**	**Frangipani file system with array out of bounds**
3	**indexer_ok**	**77**	**13**	**21**	**129**	**4**	**Insert messages into a hash table concurrently**
4	**aget-0.4_bad**	**1233**	**3**	**279**	**200**	**2**	**Multi-threaded download accelerator**
5	**bzip2smp_ok**	**6366**	**3**	**8568**	**1**	**9**	**Data compressor**
6	reorder_bad	84	10	7	10	11	Data race
7	twostage_bad	128	100	13	100	4	Atomicity violation
8	wronglock_bad	110	8	8	8	8	Wrong lock acquisition order
9	exStbHDMI_ok	1060	2	24	16	20	Configures the HDMI device
10	exStbLED_ok	425	2	45	10	10	Front panel LED display
11	exStbThumb_bad	1109	2	249	2	1	Demonstrate how thumbnail images can be manipulated
12	micro_10_ok	1171	10	10	1	17	synthetic micro-benchmark

About the benchmarks

	Module	#L	#T	#P	B	#C	Description
1	fsbench_ok	81	26	47	26	2	Frangipani file system
2	fsbench_bad	80	27	48	27	2	Frangipani file system with array out of bounds
3	indexer_ok					4	Insert messages into a hash table concurrently
4	aget-0.4_bad	1				2	Multi-threaded download accelerator
5	bzip2smp_ok	6	3	8568	1	9	Data compressor
6	**reorder_bad**	**84**	**10**	**7**	**10**	**11**	**Data race**
7	**twostage_bad**	**128**	**100**	**13**	**100**	**4**	**Atomicity violation**
8	**wronglock_bad**	**110**	**8**	**8**	**8**	**8**	**Wrong lock acquisition order**
9	exStbHDMI_ok	1060	2	24	16	20	Configures the HDMI device
10	exStbLED_ok	425	2	45	10	10	Front panel LED display
11	exStbThumb_bad	1109	2	249	2	1	Demonstrate how thumbnail images can be manipulated
12	micro_10_ok	1171	10	10	1	17	synthetic micro-benchmark

VV-lab benchmark suite

About the benchmarks

	Module	#L	#T	#P	B	#C	Description
1	fsbench_ok	81	26	47	26	2	Frangipani file system
2	fsbench_bad	80	27	48	27	2	Frangipani file system with array out of bounds
3	indexer_ok	77	13	21	129	4	Insert messages into a hash table concurrently
4	aget-0.4_bad	1233	3	279	200	2	Multi-threaded download accelerator
5	bzip2smp_ok						ata compressor
6	reorder_bad						ata race
7	twostage_bad						tomicity violation
8	wronglock_bad	1	3	8	8	8	Wrong lock acquisition order
9	**exStbHDMI_ok**	**1**	**2**	**24**	**16**	**20**	**Configures the HDMI device**
10	**exStbLED_ok**	**425**	**2**	**45**	**10**	**10**	**Front panel LED display**
11	**exStbThumb_bad**	**1109**	**2**	**249**	**2**	**1**	**Demonstrate how thumbnail images can be manipulated**
12	micro_10_ok	1171	10	10	1	17	synthetic micro-benchmark

Set-top box applications from NXP semiconductors

About the benchmarks

	Module	#L	#T	#P	B	#C	Description
1	fsbench_ok	81	26	47	26	2	Frangipani file system
2	fsbench_bad	80	27	48	27	2	Frangipani file system with array out of bounds
3	indexer_ok	77	13	21	129	4	Insert messages into a hash table concurrently
4	aget-0.4_bad	1233	3	279	200	2	Multi-threaded download accelerator
5	bzip2smp_ok	6366	2	8568	1	9	Data compressor
6	reorder_bad						
7	twostage_bad						iolation
8	wronglock_bad						k acquisition order
9	exStbHDMI_ok						the HDMI device
10	exStbLED_ok	42		45	10	10	Front panel LED display
11	exStbThumb_bad	1	2	249	2	1	Demonstrate how thumbnail images can be manipulated
12	**micro_10_ok**	**1171**	**10**	**10**	**1**	**17**	**synthetic micro-benchmark**

It is used to check the scalability of multi-threaded software verification tools [Ghafari 2010]

Comparison of the appro

encoding and solver time

number of generated and failed interleavings

error detected in module "+" **GOOD THING**

error occurred in tool "–" **BAD THING**

number of iterations

Module	Lazy							
	Time	Result	#FI/#I	Time	Result	Time	Result	Iter
fsbench_ok					+	301		1
fsbench_ba					+			2
indexer_ok					+			1
aget-0.4_ba					+	125	+	1
bzip2smp_ok	1800	+	0/1294	MO	-	MO	-	1
reorder_bad	<1	+	1/154574	MO	-	MO	-	1
twostage_bad	88	+	1/139	93	+	195	+	5
wronglock_bad	90	+	6/104015	MO	-	MO	-	1
exStbHDMI_ok	229	+	0/1	226	+	213	+	1
exStbLED_ok	73	+	0/11	73	+	787	+	1
exStbThumb_bad	95	+	3/3	14	+	12	+	1
micro_10_ok	254	+	0/29260	MO	-	MO	-	1

Comparison of the approaches (1)

lazy encoding often more efficient than schedule recording and UW

	Lazy			Schedule		UW		
	ne	Result	#Fl/#l	Time	Result	Time	Result	Iter
fsbench_ok	282	+	0/676	304	+	301	+	1
fsbench_bad	<1	+	729/729	360	+	786	+	2
indexer_ok	595	+	0/17160	220	+	218	+	1
aget-0.4_bad	137	+	1/1	127	+	125	+	1
bzip2smp_ok	1800	+	0/1294	MO	-	MO	-	1
reorder_bad	<1	+	1/154574	MO	-	MO	-	1
twostage_bad	88	+	1/139	93	+	195	+	5
wronglock_bad	90	+	6/104015	MO	-	MO	-	1
exStbHDMI_ok	229	+	0/1	226	+	213	+	1
exStbLED_ok	73	+	0/11	73	+	787	+	1
exStbThumb_bad	95	+	3/3	14	+	12	+	1
micro_10_ok	254	+	0/29260	MO	-	MO	-	1

Comparison of the approaches (2)

lazy encoding often more efficient than schedule recording and UW, ***but not always***

		Lazy			Schedule		UW		
		sult	#Fl/#l	Time	Result	Time	Result	Iter
t		+	0/676	304	+	301	+	1
fsbench_	<1	+	729/729	360	+	786	+	2
indexer_ok	595	+	0/17160	220	+	218	+	1
aget-0.4_bad	137	+	1/1	127	+	125	+	1
bzip2smp_ok	1800	+	0/1294	MO	-	MO	-	1
reorder_bad	<1	+	1/154574	MO	-	MO	-	1
twostage_bad	88	+	1/139	93	+	195	+	5
wronglock_bad	90	+	6/104015	MO	-	MO	-	1
exStbHDMI_ok	229	+	0/1	226	+	213	+	1
exStbLED_ok	73	+	0/11	73	+	787	+	1
exStbThumb_bad	95	+	3/3	14	+	12	+	1
micro_10_ok	254	+	0/29260	MO	-	MO	-	1

Comparison of the approaches (3)

lazy encoding is extremely fast for ***satisfiable instances***

Module		Lazy		Schedule		UW		
	e	Result	#FI/#I	Time	Result	Time	Result	Iter
fsbench_ok	282	+	0/676	304	+	301	+	1
fsbench_bad	<1	+	729/729	360	+	786	+	2
indexer_ok	595	+	0/17160	220	+	218	+	1
aget-0.4_bad	137	+	1/1	127	+	125	+	1
bzip2smp_ok	1800	+	0/1294	MO	-	MO	-	1
reorder_bad	<1	+	1/154574	MO	-	MO	-	1
twostage_bad	88	+	1/139	93	+	195	+	5
wronglock_bad	90	+	6/104015	MO	-	MO	-	1
exStbHDMI_ok	229	+	0/1	226	+	213	+	1
exStbLED_ok	73	+	0/11	73	+	787	+	1
exStbThumb_bad	95	+	3/3	14	+	12	+	1
micro_10_ok	254	+	0/29260	MO	-	MO	-	1

Comparison to CHESS [Musuvathi and Qadeer]

- CHESS (v0.1.30626.0) is a concurrency testing tool for C# programs; also works for C/C++ (Windows API) .
  - implements iterative context-bounding
  - requires unit tests that it repeatedly executes in a loop, exploring a different interleaving on each iteration
  - performs state hashing based on a happens-before graph
    - avoids exploring the same state repeatedly

- Goal: compare efficiency of the approaches
  - on identical verification problems taken from standard benchmark suites of multi-threaded software

CHESS is effective for programs where there are a small number of threads

SS [Musuvathi and Qadeer]

		B	C	CHESS		Lazy	
				Time	Tests	Time	#FI/#I
reorder_4_bad (3,1)	4	4	5	**98**	130000	**<1**	1/82
reorder_5_bad (4,1)	5	5	6	TO	429000	<1	1/277
reorder_6_bad (5,1)	6	6	7	TO	396000	<1	1/853
reorder_6_bad (5,1)	6	6	8	TO	371000	<1	1/2810
reorder_6_bad (5,1)	6	6	9	TO	367000	<1	1/8124
twostage_4_bad (3,1)	4	4	4	**215**	27000	**2**	1/42
twostage_5_bad (4,1)	5	5	4	TO	384000	2	1/44
twostage_6_bad (5,1)	6	6	4	TO	366000	2	1/45
wronglock_4_bad (1,3)	4	4	8	**21**	3000	**5**	2/489
wronglock_5_bad (1,4)	5	5	8	**724**	93000	**10**	3/2869
wronglock_6_bad (1,5)	6	6	8	TO	356000	18	4/12106
micro_2_ok (100)	2	1	2	**316**	35855	**<1**	0/4
micro_2_ok (100)	2	1	17	TO	40000	1095	0/131072

CHESS is effective for programs where there are a small number of threads, **but it does not scale that well and consistently runs out of time when we increase the number of threads**

[Musuvathi and Qadeer]

				CHESS		Lazy	
				Time	Tests	Time	#FI/#I
			5	98	130000	<1	1/82
reorder_5_bad (4,1)	5	5	6	TO	429000	<1	1/277
reorder_6_bad (5,1)	6	6	7	TO	396000	<1	1/853
reorder_6_bad (5,1)	6	6	8	TO	371000	<1	1/2810
reorder_6_bad (5,1)	6	6	9	TO	367000	<1	1/8124
twostage_4_bad (3,1)	4	4	4	215	27000	2	1/42
twostage_5_bad (4,1)	5	5	4	TO	384000	2	1/44
twostage_6_bad (5,1)	6	6	4	TO	366000	2	1/45
wronglock_4_bad (1,3)	4	4	8	21	3000	5	2/489
wronglock_5_bad (1,4)	5	5	8	724	93000	10	3/2869
wronglock_6_bad (1,5)	6	6	8	TO	356000	18	4/12106
micro_2_ok (100)	2	1	2	316	35855	<1	0/4
micro_2_ok (100)	2	1	17	TO	40000	1095	0/131072

Results

- lazy, schedule recording, and UW algorithms
  - *lazy:* check constraints lazily is *fast for satisfiable instances and to a lesser extent even for safe programs*

    $\Rightarrow$ *it has not been described or evaluated in the literature*
  - *schedule recording:* the number of threads and context switches can grow quickly (and easily "blow-up" the model checker)

    $\Rightarrow$ *combines symbolic with explicit state space exploration*
  - *UW:* memory overhead and slowdowns to extract the *unsat core*

    $\Rightarrow$ *it has not been used for BMC of multi-threaded software*

Future Work

- fault localization in multi-threaded C programs
- verify real-time software using SMT techniques
- interpolants to prove non-interference of context switches