Systems and Software
Verification Laboratory

MANCHESTER
1824
The University of Manchester

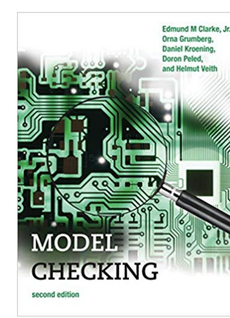
Detection of Software Vulnerabilities: Static Analysis (Part II)

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**

Intended learning outcomes

- Introduce typical **BMC architectures** for verifying **software systems**
- Understand **communication models** and **typical errors** when writing **concurrent programs**

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

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

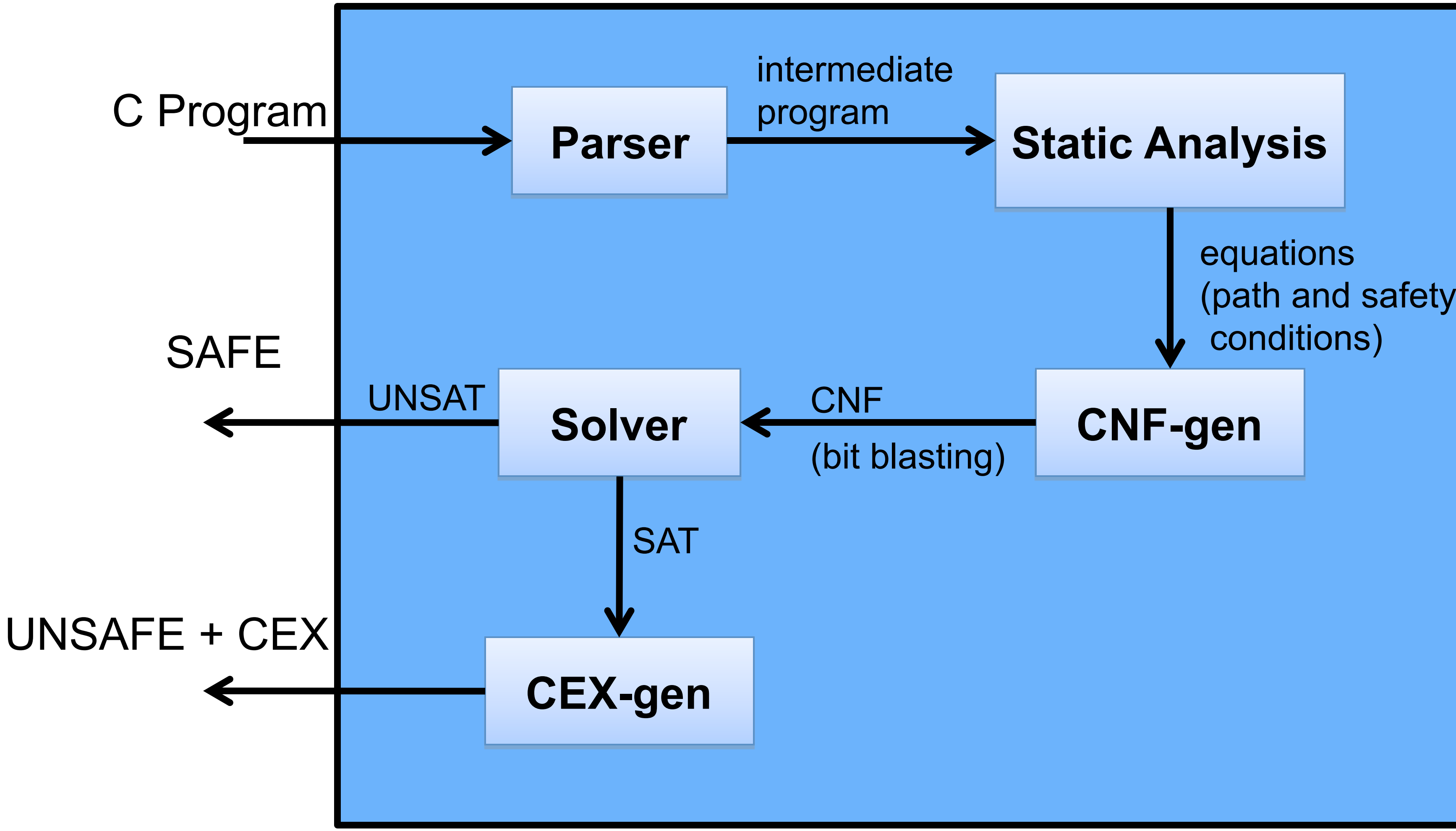
SAT/SMT-based BMC tools for C

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

unwind one
iteration

Loop unwinding

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

unwind one it…

unwind one it…

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:

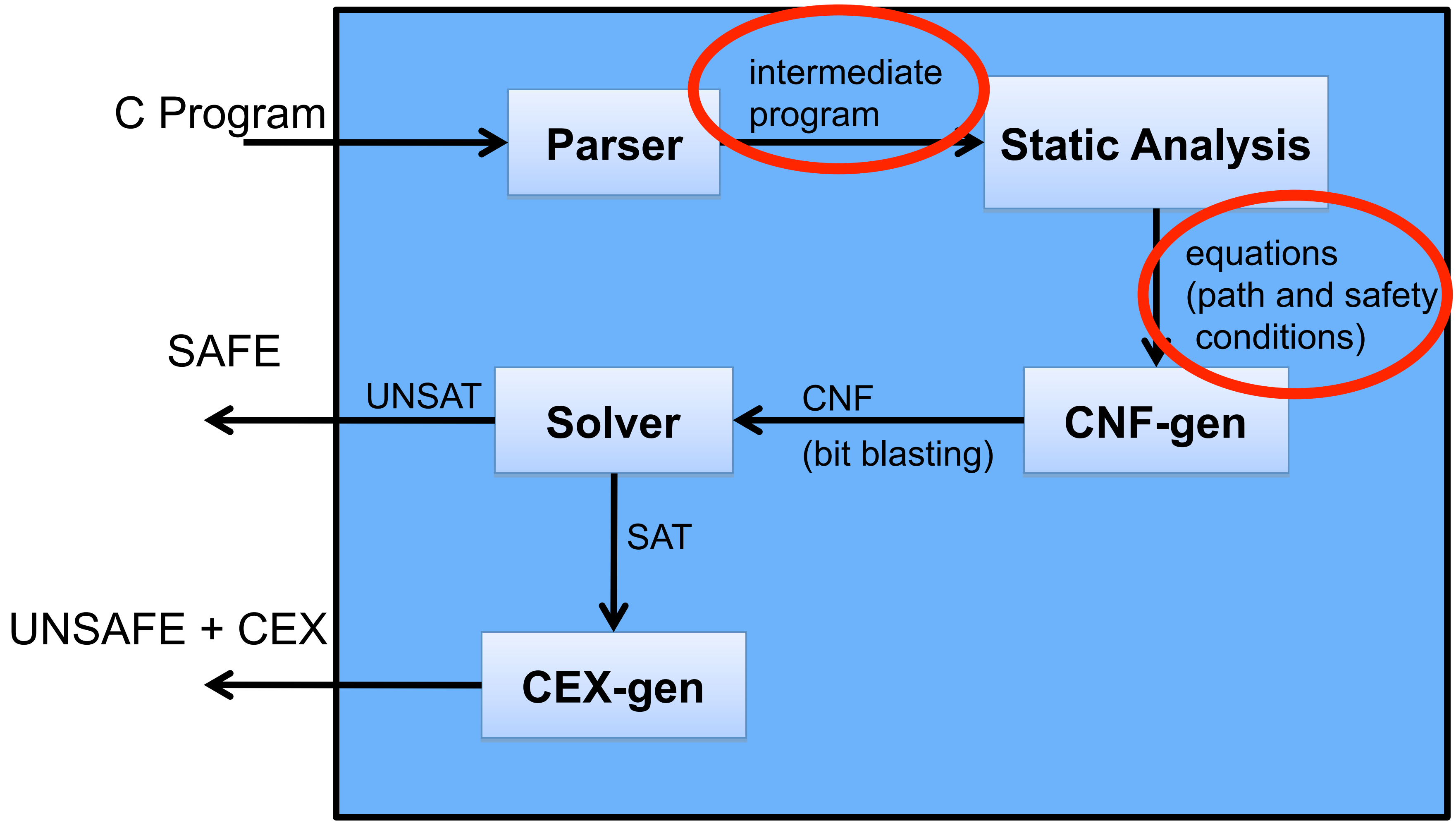
  ```
  a[i]=...;
   ⇒ assert(0 <= i && i <= N); a[i]=...;
  ```

⇒ 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

$\Rightarrow$

$x = a \quad \&\&$
$y = x + 1 \quad \&\&$
$z = y - 1$

constraints

- still simple if variables are assigned multiple times:

x = a;
x = x + 1;
x = x – 1;

program

$\Rightarrow$

$x_0 = a;$
$x_1 = x_0 + 1;$
$x_2 = x_1 - 1;$

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(v_0)
  x_0 = y_0;
else
  x_1 = z_0;

w_1 = ?
```

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;
```

$\texttt{if(}v_0\texttt{)}$
$\quad x_0 = y_0;$
$\texttt{else}$
$\quad x_1 = z_0;$
$x_2 = v_0 \; ? \; x_0 : x_1;$
$w_1 = x_2;$

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:

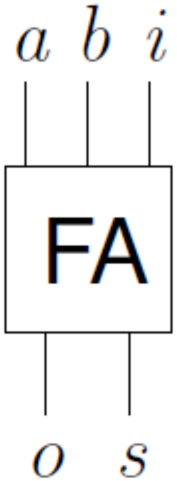
  $|[\, x = y \mid z \,]| \triangleq \bigwedge_i x_i \Leftrightarrow (y_i \vee z_i)$

Exercise: relational operations

Bit-blasting arithmetic operations

Build **circuits** that implement the operations!

1-bit addition:



Full Adder

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

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

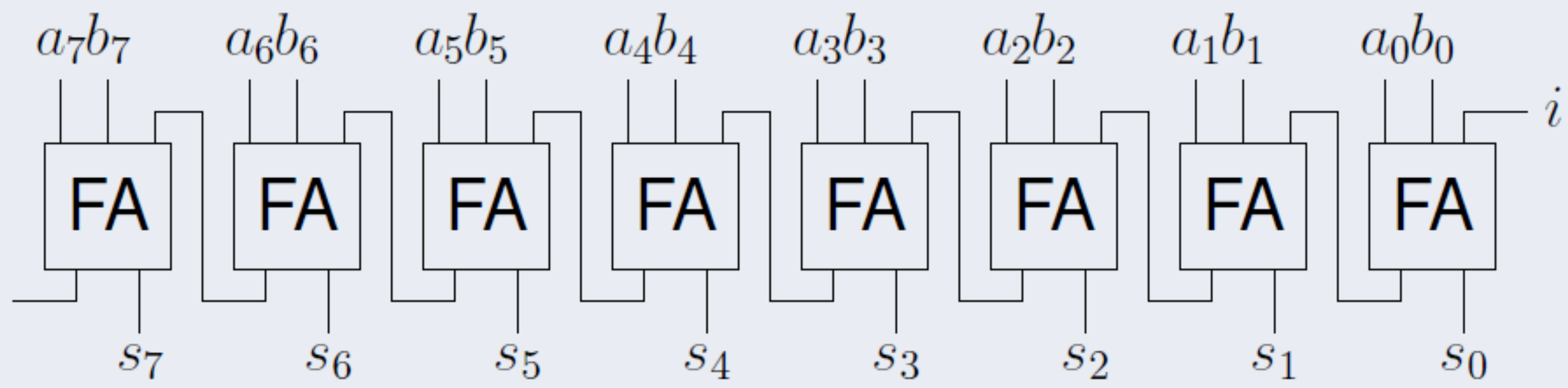
Full adder as CNF:

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

Bit-blasting arithmetic operations

Build **circuits** that implement the operations!

8-Bit ripple carry adder (RCA)



⇒ adds w variables, 6*w clauses

⇒ multiplication / division much more complicated

Handling arrays

Arrays can be replaced by individual variables, with a “demux” at each access:

```
int a[10];
...
x = a[i];
```

```
int a_0, a_1, a_2, ... a_9;
...
x = (i==0 ? a_0
      : (i==1 ? a_1
          : (i==2 ? a_2
              : ...);
```

⇒ surprisingly effective (for N<1000) because value of *i* can often be determined statically

– due to constant propagation

Handling arrays with theories

Arrays can be seen as ADT with two operations:

- read: Array x Index → Element “select”
- write: Array x Index x Element → Array “update”

```
...
a[i]=a[i]+1;
...
```

...

a_1=write(a_0,i,read(a_0,i)+1);

...

Handling arrays with theories

Arrays can be seen as ADT with two operations:

- read: Array x Index → Element *"select"*
- write: Array x Index x Element → Array *"update"*

```
...
a[i]=a[i]+1;
...
```

⇒

```
...
a1=write(a0,i,read(a0,i)+1);
...
```

Axioms describe intended semantics:

a write modifies the position written to …

$$p = r \implies \mathtt{read}(\mathtt{write}(a, p, v), r) = v$$

$$\neg(p = r) \implies \mathtt{read}(\mathtt{write}(a, p, v), r) = \mathtt{read}(a, r)$$

… and nothing else

⇒ requires support by **SMT-solver**

Handling arrays with λ-terms

How to handle `memset` and `memcpy`?

```
void *memset(void *dst, int c, size_t n);

void *memcpy(void *dst, const void *src, size_t n);
```

Handling arrays with λ-terms

How to handle `memset` and `memcpy`?

void *memset(void *dst, int c, size_t n);

void *memcpy(void *dst, const void *src, size_t n);

```
...
memcpy(a,b,4);
...
```

→

$$
\begin{aligned}
&\ldots\\
&a_1 = \mathtt{write}(a_0, 0, \mathtt{read}(b, 0));\\
&a_2 = \mathtt{write}(a_1, 1, \mathtt{read}(b, 1));\\
&a_3 = \mathtt{write}(a_2, 2, \mathtt{read}(b, 2));\\
&a_4 = \mathtt{write}(a_3, 3, \mathtt{read}(b, 3));\\
&\ldots
\end{aligned}
$$

Handling arrays with λ-terms

How to handle `memset` and `memcpy`?

void *memset(void *dst, int c, size_t n);

void *memcpy(void *dst, const void *src, size_t n);

```
...
memcpy(a,b,4);
...
```

$$\begin{array}{l}
\texttt{...}\\
a_1\texttt{=write(}a_0\texttt{,0,read(b,0));}\\
a_2\texttt{=write(}a_1\texttt{,1,read(b,1));}\\
a_3\texttt{=write(}a_2\texttt{,2,read(b,2));}\\
a_4\texttt{=write(}a_3\texttt{,3,read(b,3));}\\
\texttt{...}
\end{array}$$

- not scalable for large constants
- need to encode as loop for non-constant block sizes
  - same problems for normal array-copy operations

Handling arrays with λ-terms

How to handle `memset` and `memcpy`?

void *memset(void *dst, int c, size_t n);

void *memcpy(void *dst, const void *src, size_t n);

Abuse of notation

```
...
memcpy(a,b,4);
...
```

```
...
a_1=λi·(0<=i && i<4) ?
        read(b,i) : read(a_0,i));
...
```

- similar for `memset` and array-copy loops
- additional axiom describes intended semantics

$$\mathtt{read}(\lambda i.\, s, r) = s[i/r]$$

β-reduction

⇒ requires integration into **SMT-solver**

Lambdas, Arrays and Quantifiers

- **Parallel updates**

  Update n elements of array a with value c starting from index i, which yields a new array b, e.g.,

  $b = memset(a, i, n, c)$

  $\forall x\,.\,(\mathrm{read}(b,\ x) = \mathrm{ite}(i \leq x < i + n,\ c,\ \mathrm{read}(a,\ x)))$

  $\lambda x\,.\,\mathrm{ite}(i \leq x < i + n,\ c,\ \mathrm{read}(a,\ x))$

- **Copy operations**

  Copy n elements of array a starting from index i to array b at index j, which yields a new array b', e.g.,

  $b' = memcpy(a, b, i, j, n)$

  $\forall x\,.\,(\mathrm{read}(b',\ x) = \mathrm{ite}(j \leq x < j + n,\ \mathrm{read}(a,\ i + x - j),\ \mathrm{read}(b,\ x)))$

  $\lambda x\,.\,\mathrm{ite}(j \leq x < j + n,\ \mathrm{read}(a,\ i + x - j),\ \mathrm{read}(b,\ x))$

Mathias Preiner, Aina Niemetz, Armin Biere: Better Lemmas with Lambda Extraction. FMCAD 2015: 128-135

Handling arrays with λ-terms

```
int i, j, n = ...;
int *a = malloc(2 * n * sizeof(int));
for (i = 0; i < n; ++i) {
    a[i] = i + 1;
}
for (j = n; j < 2 * n; ++j) {
    a[j] = 2 * j;
}
```

$$a' = \lambda i.\, \mathsf{ITE}(0 \le i < n,\ i + 1,\ \mathtt{read}(a, i))$$

$$a'' = \lambda j.\, \mathsf{ITE}(n \le j < 2 * n,\ 2 * j,\ \mathtt{read}(a', j))$$

Stephan Falke, Florian Merz, Carsten Sinz: Extending the Theory of Arrays: memset, memcpy, and Beyond. VSTTE 2013: 108-128

SAT vs. SMT

BMC tools use both propositional satisfiability (SAT) and satisfiability modulo theories (SMT) solvers:

SAT vs. SMT

BMC tools use both propositional satisfiability (SAT) and satisfiability modulo theories (SMT) solvers:

- SAT solvers require encoding everything in CNF
  - limited support for high-level operations
  - easier to reflect machine-level semantics
  - can be extremely efficient (SMT falls back to SAT)

SAT vs. SMT

BMC tools use both propositional satisfiability (SAT) and satisfiability modulo theories (SMT) solvers:

- SAT solvers require encoding everything in CNF
  - limited support for high-level operations
  - easier to reflect machine-level semantics
  - can be extremely efficient (SMT falls back to SAT)
- SMT solvers support built-in theories
  - equality, free function symbols, arithmetics, arrays,...
  - sometimes even quantifiers
  - very flexible, extensible, front-end easier
  - requires extra effort to enforce precise semantics
  - can be slower

Modeling with non-determinism

Extend C with three modeling features:

- **assert(e)**: aborts execution when e is false, no-op otherwise

```
void assert (_Bool b) { if (!b)  exit(); }
```

Modeling with non-determinism

Extend C with three modeling features:

- **assert(e)**: aborts execution when e is false, no-op otherwise

```
void assert (_Bool b) { if (!b)  exit(); }
```

- **nondet_int()**: returns non-deterministic int-value

```
int nondet_int () { int x; return x; }
```

Modeling with non-determinism

Extend C with three modeling features:

- **assert(e)**: aborts execution when e is false, no-op otherwise

```
void assert (_Bool b) { if (!b)  exit(); }
```

- **nondet_int()**: returns non-deterministic int-value

```
int nondet_int () { int x; return x; }
```

- **assume(e)**: “ignores” execution when e is false, no-op otherwise

```
void assume (_Bool e) { while (!e) ;  }
```

Modeling with non-determinism

General approach:

- use C program to set up structure and deterministic computations
- use non-determinism to set up search space
- use assumptions to constrain search space
- use failing assertion to start search

```
int main() {
  int x=nondet_int(),y=nondet_int(),z=nondet_int();
  __ESBMC_assume(x > 0 && y > 0 && z > 0);
  __ESBMC_assume(x < 16384 && y < 16384 && z < 16384);
  assert(x*x + y*y != z*z);
  return 0;
}
```

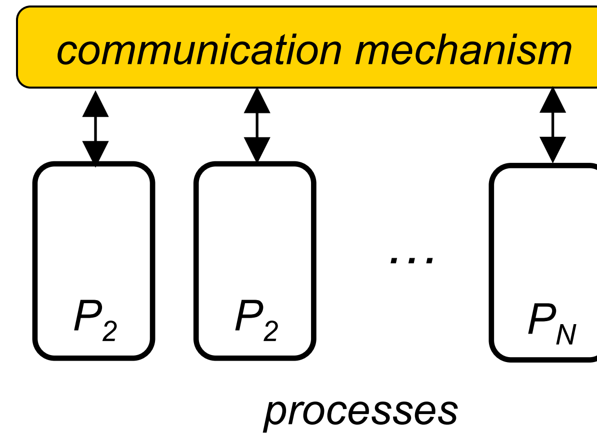
Intended learning outcomes

- 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

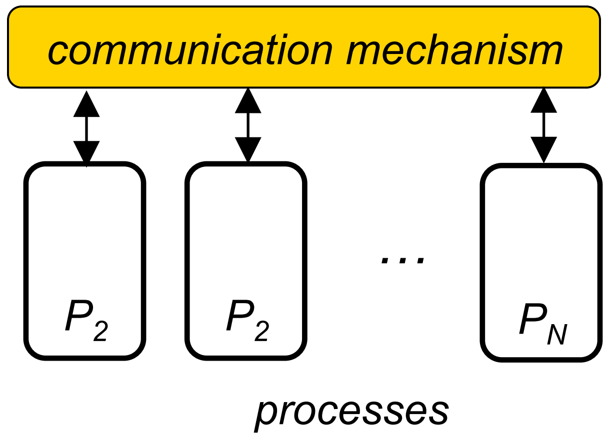
- programmers have to guarantee
  - correctness of sequential execution of each individual process
  - with nondeterministic interferences from other processes (schedules)



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
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

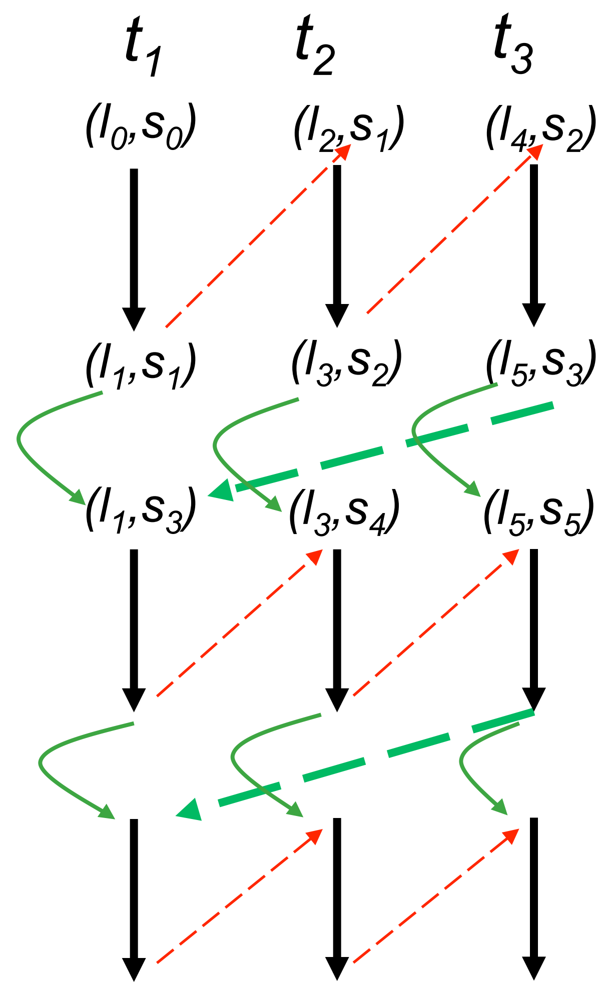
- typical C-implementation: `pthreads`
- formed of individual sequential programs (threads)
  - can be created and destroyed on the fly
  - typically for BMC: assume upper bound
  - each possibly with loops and recursive function calls
  - each with local variables
- each thread can read and write shared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

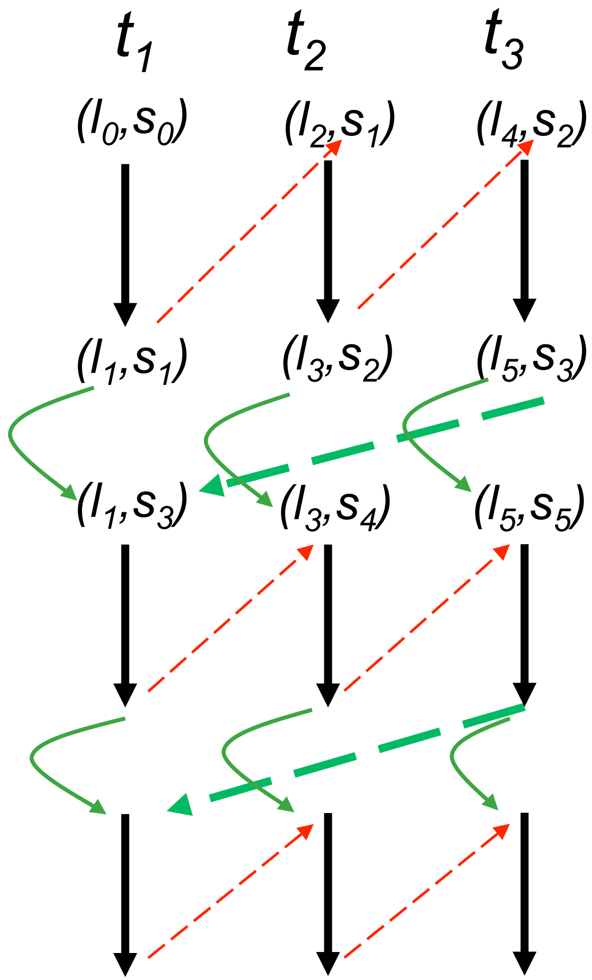
Round-robin scheduling

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



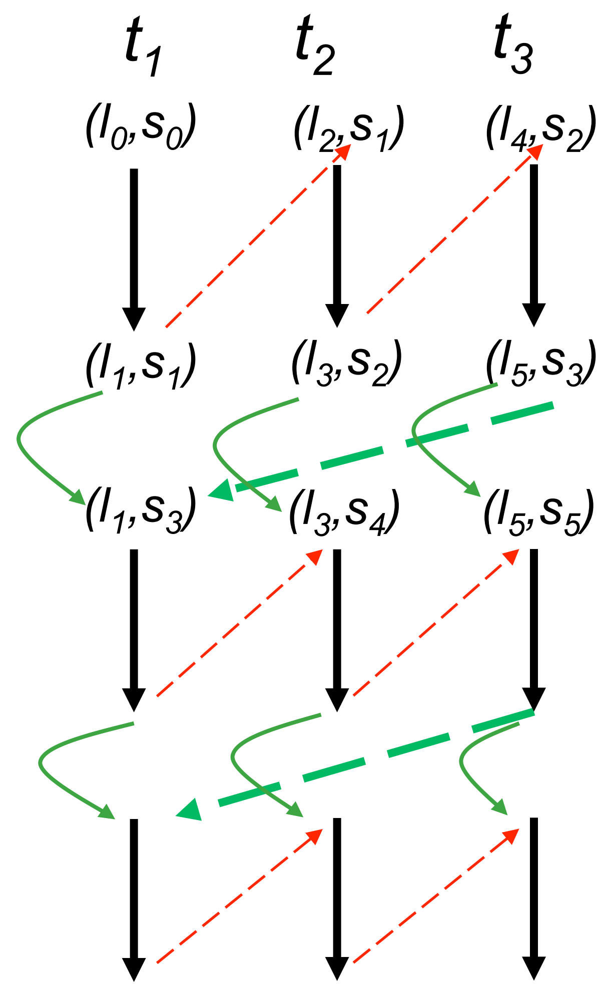
Round-robin scheduling

- 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)



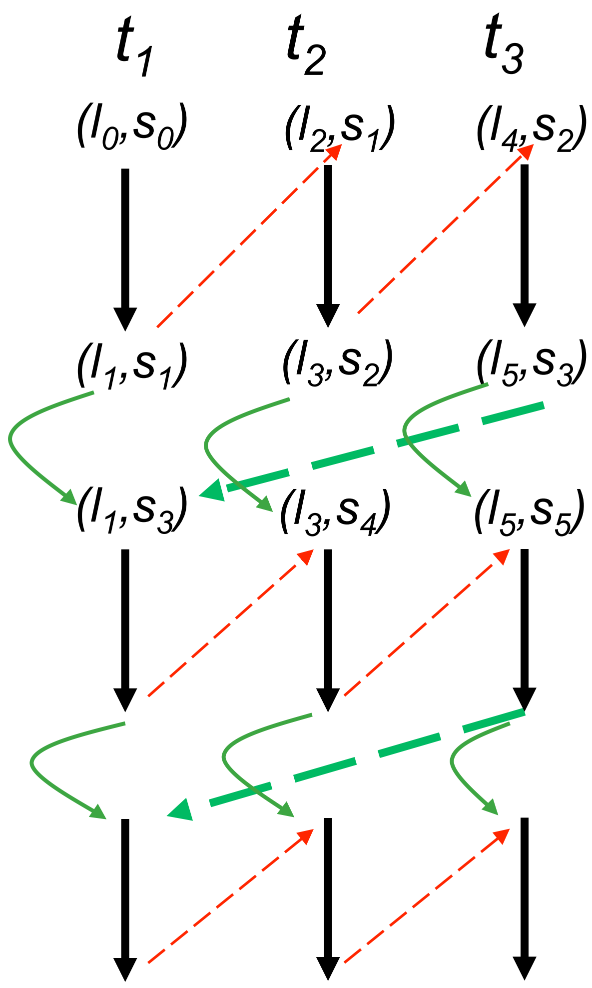
Round-robin scheduling

- 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



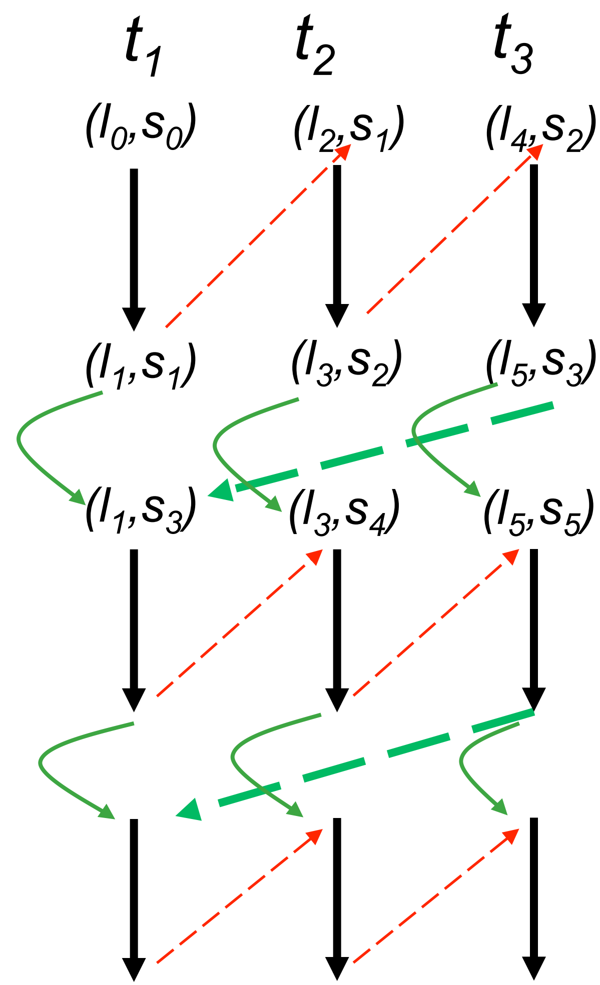
Round-robin scheduling

- 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

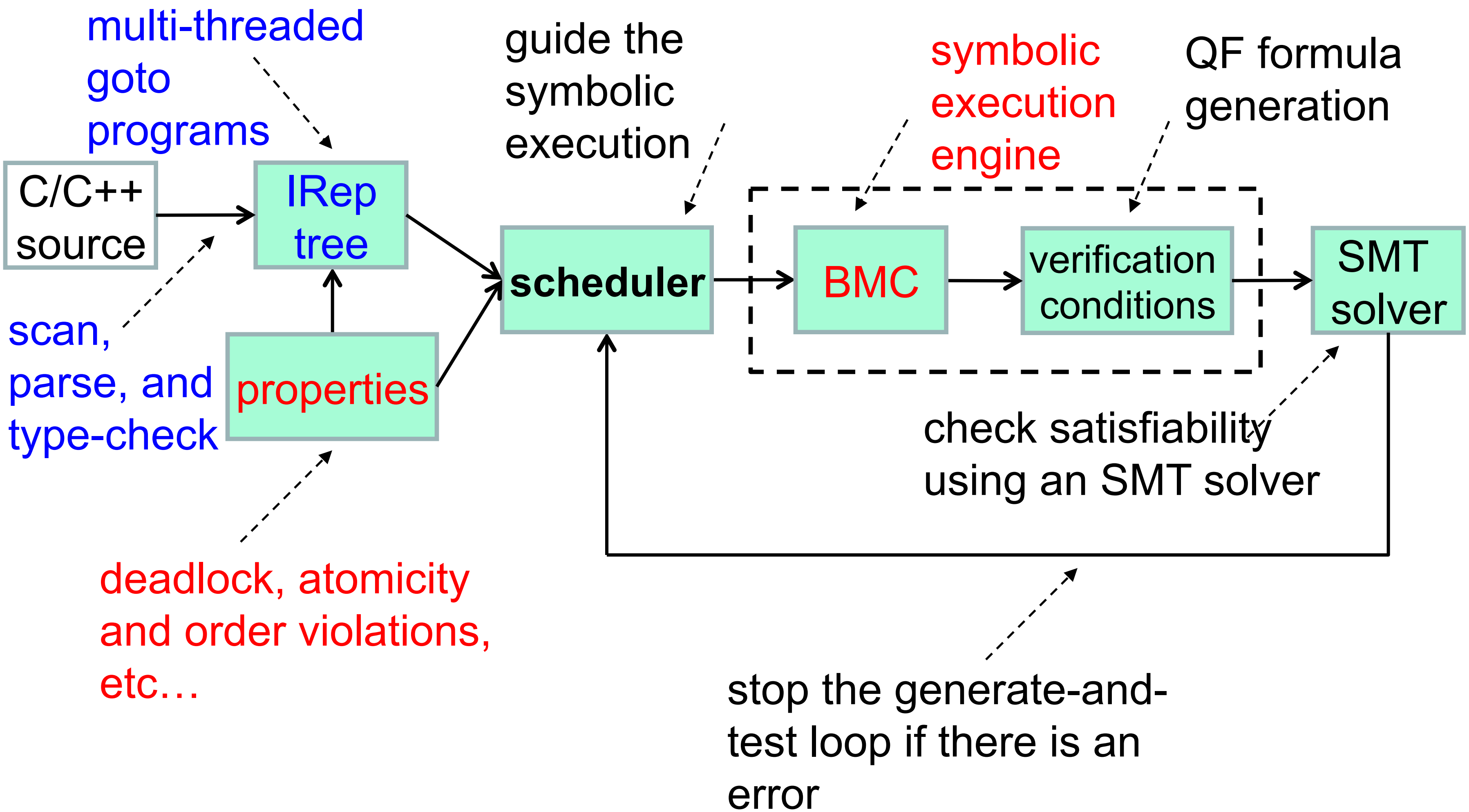
- Explicit schedule exploration (ESBMC)
  - lazy exploration
  - schedule recording
- Partial order methods (CBMC)
- Sequentialization
  - KISS
  - Lal / Reps (eager sequentialization)
  - Lazy CSeq
  - memory unwinding

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);
```

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

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

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

Lazy exploration: interleaving I_s

statements:

val1-access:

val2-access:

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

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

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

Lazy exploration: interleaving I_s

statements: 1

val1-access:

val2-access:

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

program counter: 1

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

global variables: val1=0; val2=0;

local variabes: 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: $W_{twoStage,2}$

val2-access:

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

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

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

program counter: 2

mutexes: m1=1; m2=0;

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

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

Lazy exploration: interleaving I_s

statements: 1-2-3

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

val2-access:

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

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

program counter: 3

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

global variables: val1=1; val2=0;

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

Lazy exploration: interleaving I_s

statements: 1-2-3-7

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

val2-access:

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

CS1

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

program counter: 7

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

global variables: val1=1; val2=0;

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

Lazy exploration: interleaving I

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

statements: 1-2-3-7-8

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

val2-access:

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

CS1

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

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

Lazy exploration: interleaving I_s

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

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

val2-access:

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

CS1

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

program counter: 11

mutexes: m1=1; m2=0;

global variables: val1=1; val2=0;

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

Lazy exploration: interleaving I_s

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

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

val2-access:

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

CS1

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

program counter: 12

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

global variables: val1=1; val2=0;

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

Lazy exploration: interleaving I_s

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

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

val2-access:

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

CS1

CS2

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

program counter: 4

mutexes: m1=0; m2=0;

global variables: val1=1; val2=0;

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

Lazy exploration: interleaving I_s

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

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

val2-access:

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

CS1

CS2

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

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

Lazy exploration: interleaving I_s

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

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

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

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

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 variabes: t1= 1; t2= -1;

Lazy exploration: interleaving I_s

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

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

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

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

CS1

CS2

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

program counter: 6
mutexes: m1=0; ***m2=0****;*
global variables: val1=1; val2=2;
local variabes: t1= 1; t2= -1;

Lazy exploration: interleaving I_s

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

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

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

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

CS1

CS2

CS3

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

program counter: 13
mutexes: m1=0; m2=0;
global variables: val1=1; val2=2;
local variabes: t1= 1; t2= -1;

Lazy exploration: interleaving I_s

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

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

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

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

CS1

CS2

CS3

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

program counter: 13
mutexes: m1=0; ***m2=1***;
global variables: val1=1; val2=2;
local variabes: t1= 1; t2= -1;

Lazy exploration: interleaving I_s

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

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

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

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

CS1

CS2

CS3

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

program counter: 14

mutexes: m1=0; m2=1;

global variables: val1=1; val2=2;

local variabes: t1= 1; ***t2= 2****;*

Lazy exploration: interleaving I_s

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

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

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

Thread twoStage

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

CS1

CS2

CS3

Thread reader

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

program counter: 15

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

global variables: val1=1; val2=2;

local variabes: t1= 1; t2= 2;

Lazy exploration: interleaving I_s

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

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

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

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

CS1

CS2

CS3

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

program counter: 16

mutexes: m1=0; m2=0;

global variables: val1=1; val2=2;

local variabes: t1= 1; t2= 2;

Lazy exploration: interleaving I_s

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

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

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

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

CS1

CS2

CS3

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

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

Lazy exploration: interleaving I_f

statements:

val1-access:

val2-access:

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

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

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

Lazy exploration: interleaving I_f

statements: 1-2-3

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

val2-access:

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

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

program counter: 3
mutexes: m1=0; m2=0;
*global variables: **val1=1**; val2=0;*
local variabes: t1= -1; t2= -1;

Lazy exploration: interleaving I_f

statements: 1-2-3

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

val2-access:

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

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

program counter: 7
mutexes: m1=0; m2=0;
global variables: val1=1; val2=0;
local variabes: t1= -1; t2= -1;

Lazy exploration: interleaving I_f

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

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

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

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

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

program counter: 16
mutexes: m1=0; m2=0;
global variables: val1=1; val2=0;
*local variabes: **t1= 1**; **t2= 0**;*

Lazy exploration: interleaving I_f

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

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

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

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

CS2

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

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

Lazy exploration: interleaving I_f

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

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

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

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

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

CS2

program counter: 6

mutexes: m1=0; m2=0;

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

local variabes: t1= 1; t2= 0;

Lazy exploration: interleaving I_f

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

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

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

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

CS1

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

CS2

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

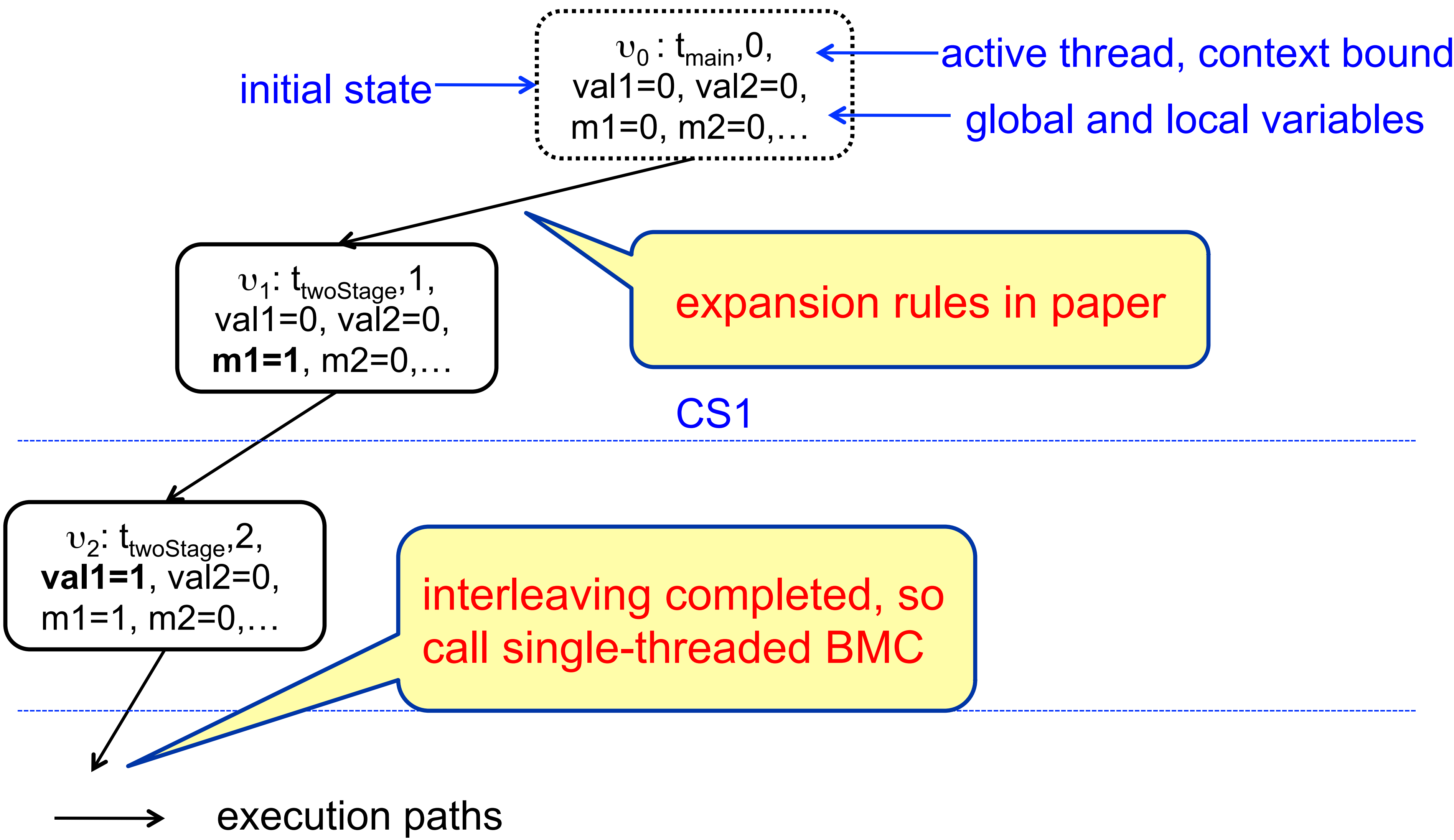
Lazy exploration of interleavings

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

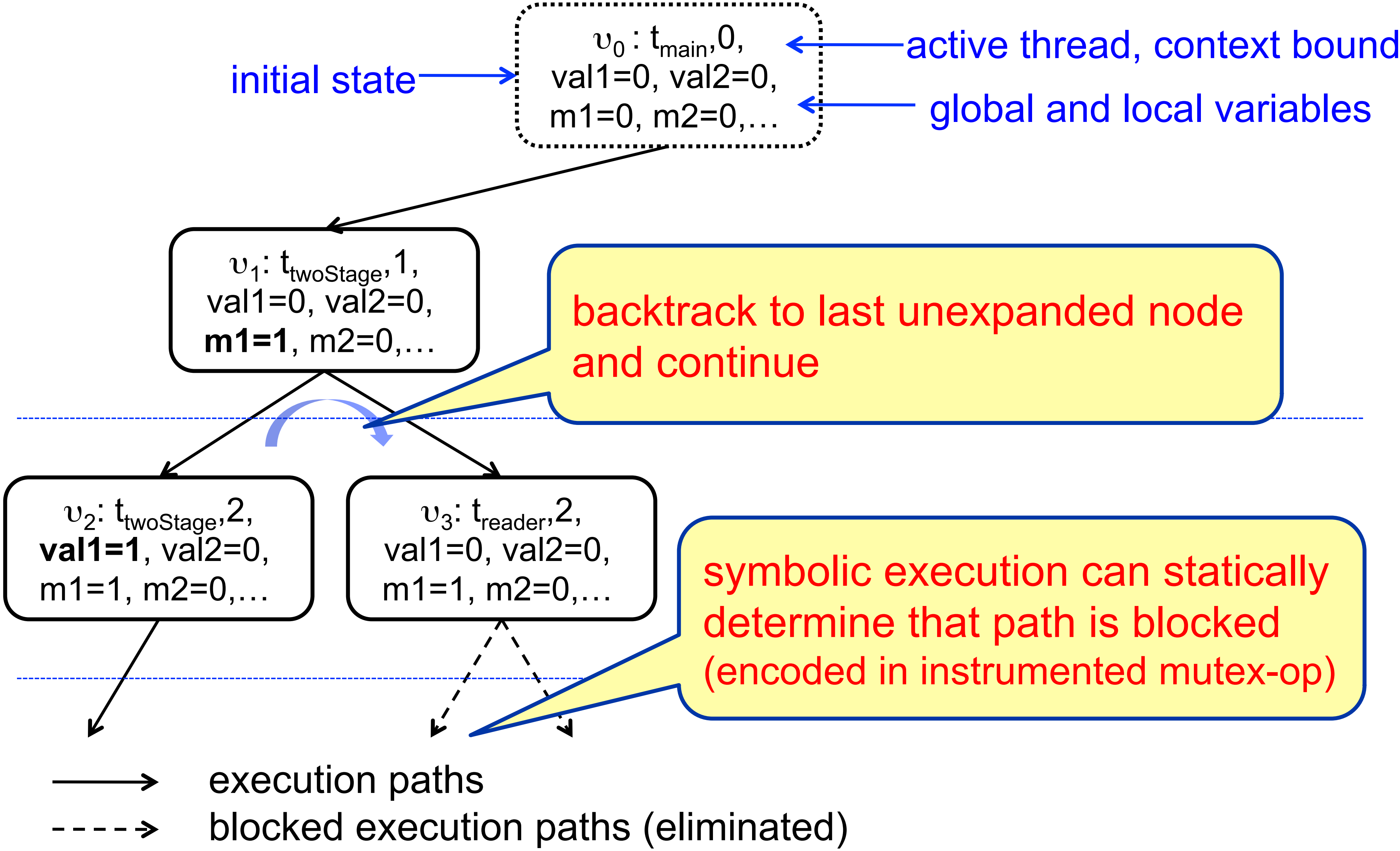
... combines

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

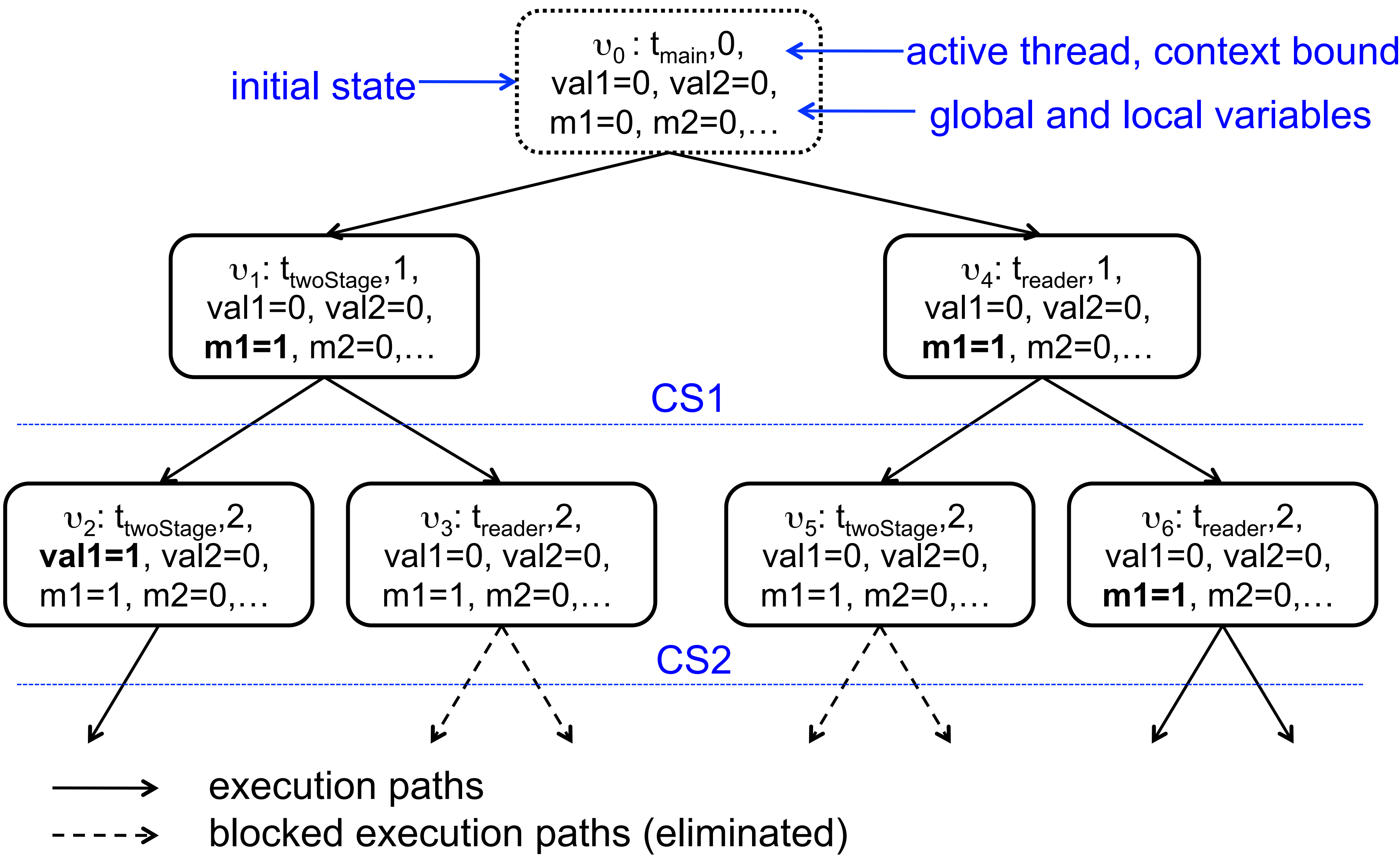
Lazy exploration of interleavings – Reachability Tree



Lazy exploration of interleavings – Reachability Tree



Lazy exploration of interleavings – Reachability Tree



Exploring the Reachability Tree

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

Exploring the Reachability Tree

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

Exploring the Reachability Tree

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

Exploring the Reachability Tree

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

Exploring the Reachability Tree

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

Exploring the Reachability Tree

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

Exploring the Reachability Tree

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

Expansion Rules of the RT

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

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

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

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

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

Expansion Rules of the RT

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

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

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:

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

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

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

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

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

Expansion Rules of the RT

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

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

$$\upsilon' = \left(A_i, C_i, s_i, \left\langle \underline{l^j_{i+1}}, c \underline{\wedge\, G^j_i} \right\rangle\right)_{i+1} \qquad l^j_{i+1} = \begin{cases} l & : \quad j = A_i \\ l^j_i & : \quad otherwise \end{cases}$$

– for υ'', we add ¬*c* to the guards and continue with the next instruction in the current thread

$$\upsilon'' = \left(A_i, C_i, s_i, \left\langle \underline{l^j_{i+1}}, \neg c \underline{\wedge\, G^j_i} \right\rangle\right)_{i+1} \qquad l^j_{i+1} = \begin{cases} l^j_i + 1 & : \quad j = A_i \\ l^j_i & : \quad otherwise \end{cases}$$

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

Expansion Rules of the RT

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

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

Expansion Rules of the RT

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

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

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

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

Expansion Rules of the RT

R5 (start_thread): If *I* is a *start_thread* instruction, we add the indicated thread to the set of active threads:

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

- 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 *I* is a *start_thread* instruction, we add the indicated thread to the set of active threads:

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

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

R6 (join_thread): If *I* is a *join_thread* instruction with argument *Id*, we add a child node:

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

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

Lazy exploration of interleavings

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

computation path

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

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

bound

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: 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);	$ts_1 == 1$
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);              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;
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);	$ts_1 == 1$
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);	
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)

CS

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);	$ts_4 == 2$
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)

Thread twoStage	
1: lock(m1);	$ts_1 == 1$
2: val1 = 1;	$ts_2 == 1$
3: unlock(m1);	$ts_3 == 1$
4: lock(m2);	$ts_8 == 1$
5: val2 = val1 + 1;	
6: unlock(m2);	

CS

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

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

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);	$ts_8 == 1$
5: val2 = val1 + 1;	$ts_9 == 1$
6: unlock(m2);	

CS

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

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

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

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

CS

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

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)

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

CS

CS

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);	$ts_{11} == 2$
14: t2 = val2;	
15: unlock(m2);	
16: assert(t2==(t1+1));	

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)

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

CS

CS

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);	$ts_{11} == 2$
14: t2 = val2;	$ts_{12} == 2$
15: unlock(m2);	
16: assert(t2==(t1+1));	

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)

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

CS

CS

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);	$ts_{11} == 2$
14: t2 = val2;	$ts_{12} == 2$
15: unlock(m2);	$ts_{13} == 2$
16: assert(t2==(t1+1));	

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)

Thread twoStage
1: lock(m1); $ts_1 == 1$
2: val1 = 1; $ts_2 == 1$
3: unlock(m1); $ts_3 == 1$
4: l
5: v
6: u

CS

Thread reader
7: lock(m1); $ts_4 == 2$
8: if (val1 == 0) { $ts_5 == 2$
9: unlock(m1);
return NULL; }
t1 = val1; $ts_6 == 2$
unlock(m1); $ts_7 == 2$
: lock(m2); $ts_{11} == 2$
14: t2 = val2; $ts_{12} == 2$
15: unlock(m2); $ts_{13} == 2$
16: assert(t2==(t1+1)); $ts_{14} == 2$

interleaving completed, so build constraints for interleaving (but do not call SMT solver)

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)

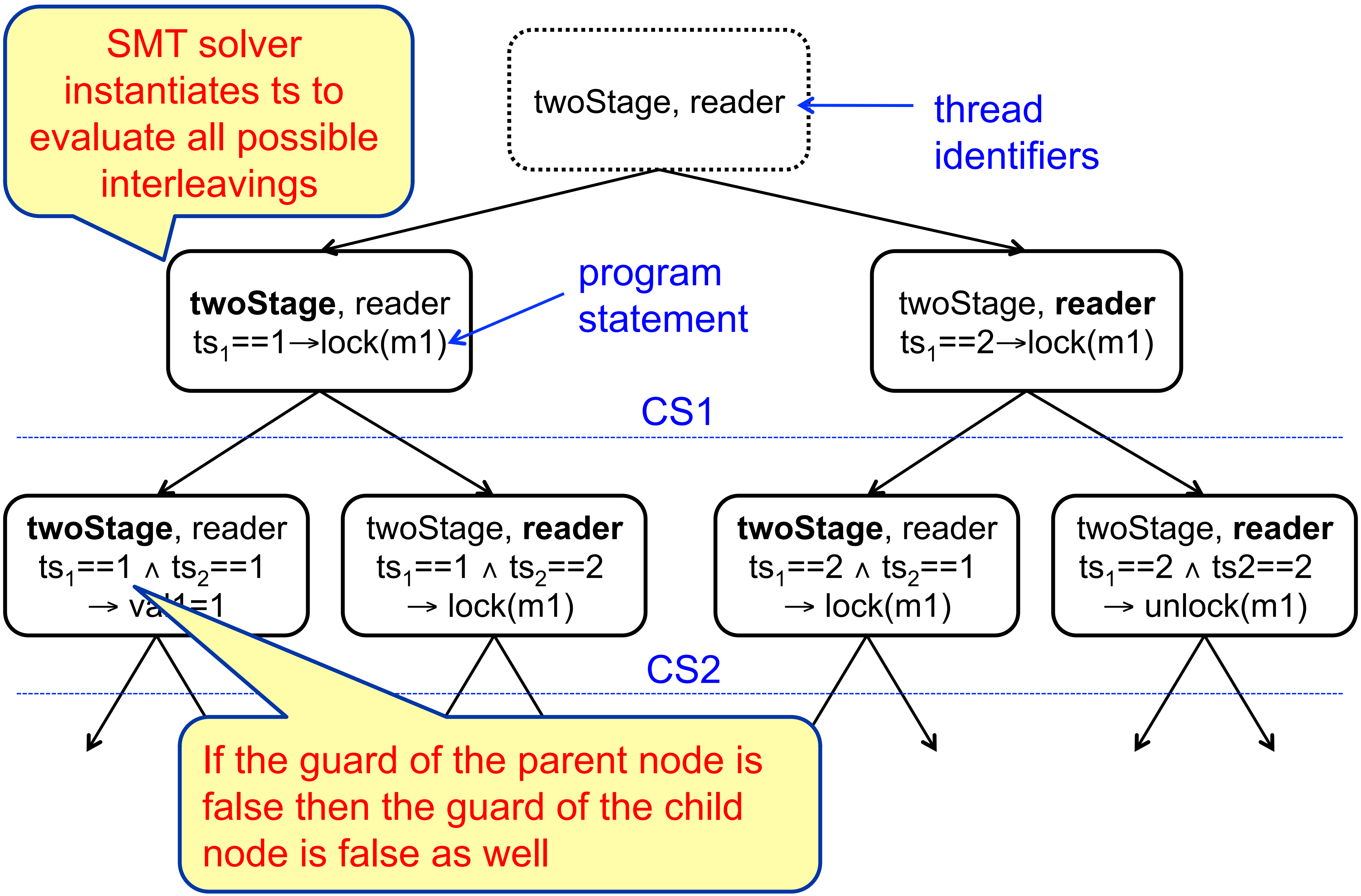
Thread twoStage	
1: lock(m1);	$ts_1 == 1$
2: val1 = 1;	$ts_2 == 1$
3: unlock(m1);	$ts_3 == 1$
4: lock(m2);	$ts_{12} == 1$
5: val2 = val1 + 1;	$ts_{13} == 1$
6: unlock(m2);	$ts_{14} == 1$

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);	$ts_8 == 2$
14: t2 = val2;	$ts_9 == 2$
15: unlock(m2);	$ts_{10} == 2$
16: assert(t2==(t1+1));	$ts_{11} == 2$

CS

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

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)

$\Rightarrow$ 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**: $T_1 \, /\!/ \, T_2 \rightsquigarrow T'_1 \, ; \, T'_2$

⇒ reuse existing tools (largely) unchanged

⇒ easy to target multiple back-ends

⇒ easy to experiment with different approaches

A first sequentialization: KISS

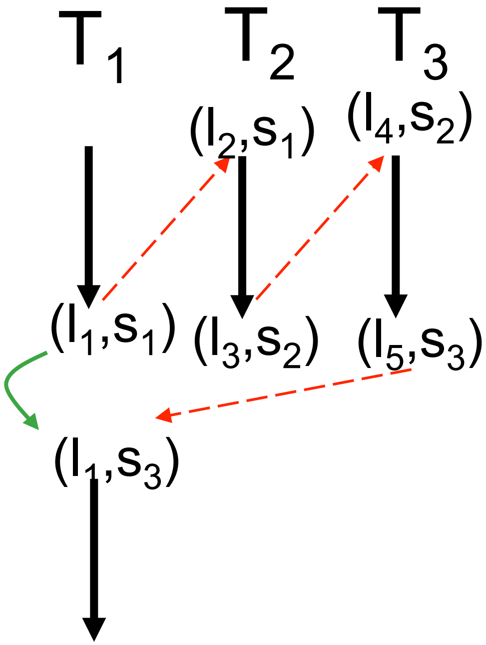
KISS: Keep It Simple and Sequential **[Quadeer-Wu, PLDI' 04]**

Under-approximation (subset of interleavings)

Thread creation → function call

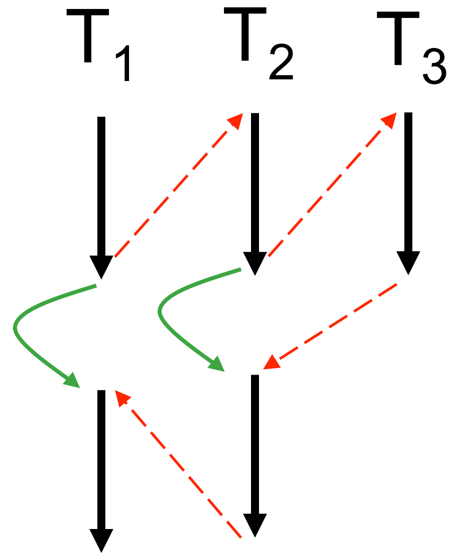
- **at context-switches either:**
  - o the active thread is terminated or
  - o a not yet scheduled thread is started (by calling its main function)
- **when a thread is terminated either:**
  - o the thread that has called it is resumed (if any) or
  - o a not yet scheduled thread is started

KISS schedules



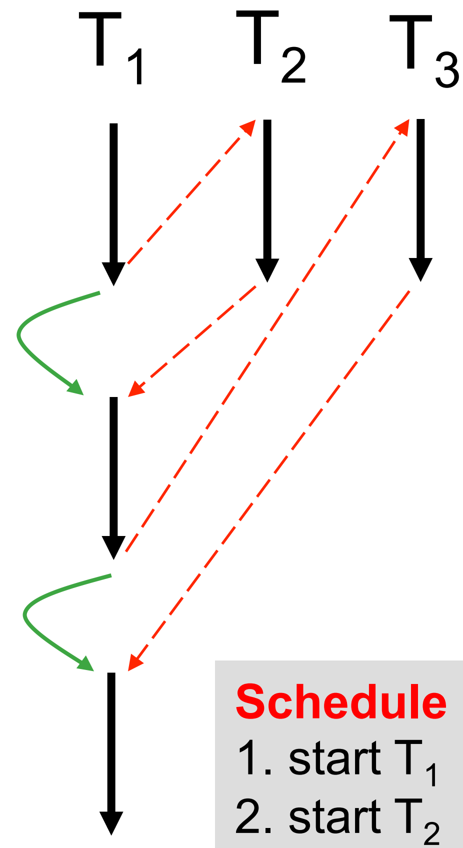
Schedule 1:

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

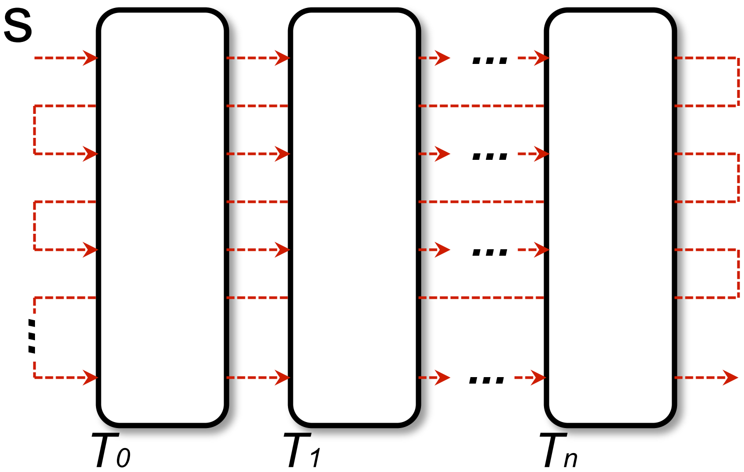


Schedule 3:

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

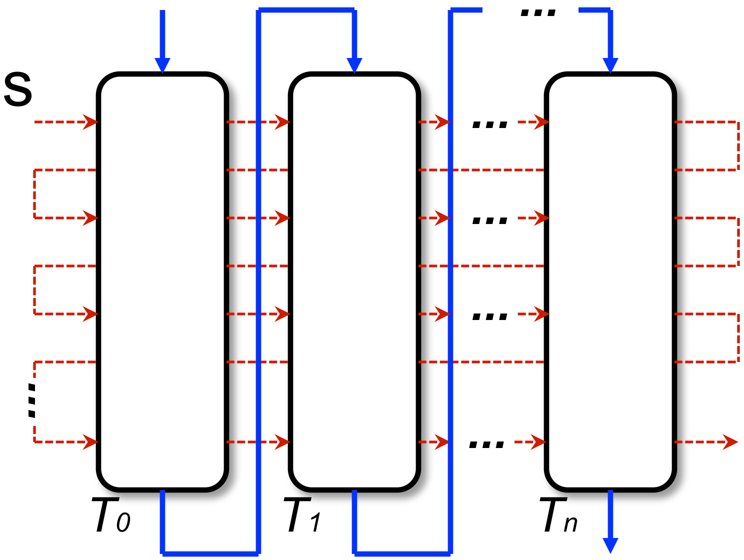
LR sequentialization

- considers only round-robin schedules with *k* rounds



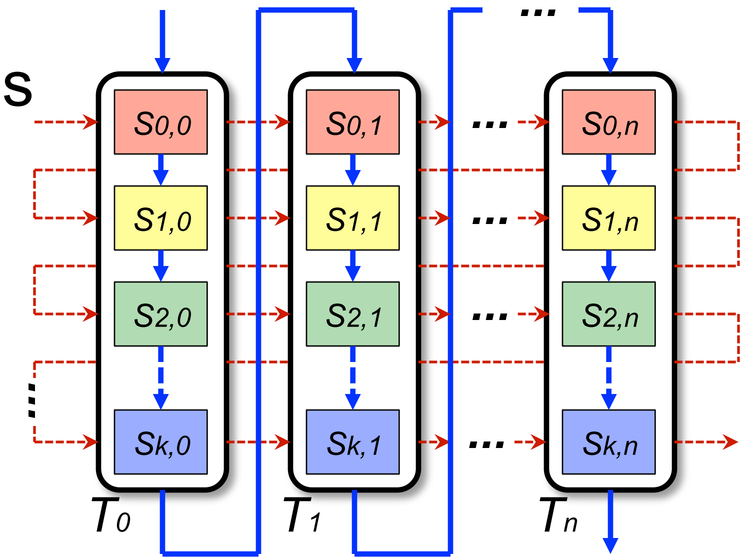
LR sequentialization

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



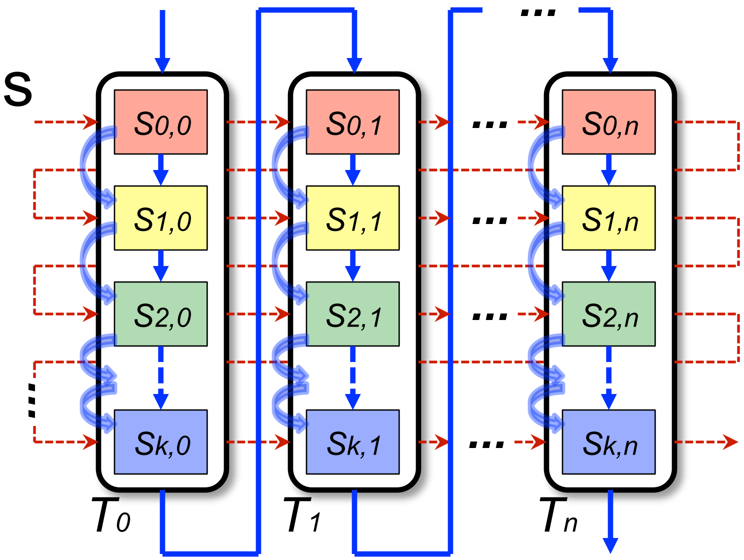
LR sequentialization

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



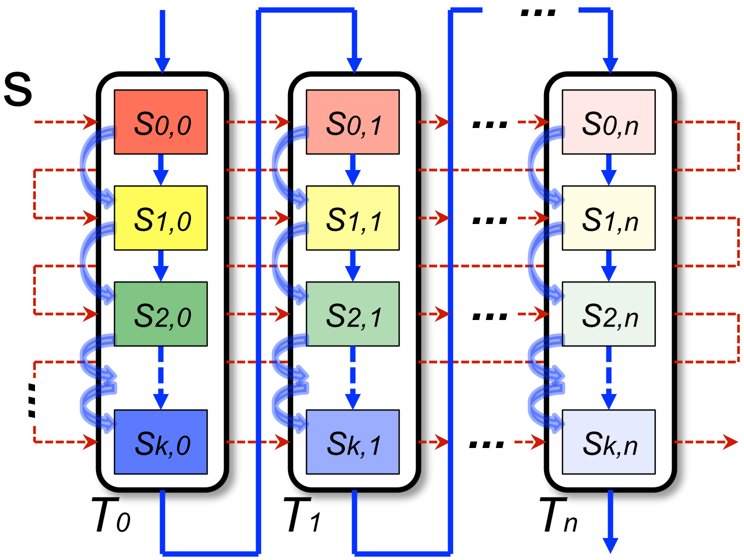
LR sequentialization

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

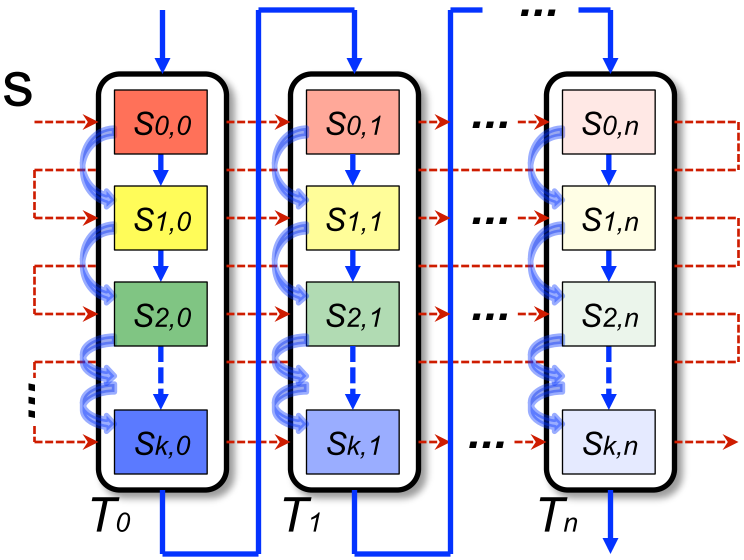


LR sequentialization

- considers only round-robin schedules with *k* rounds
  - thread → function, run to completion
- global memory copy for each round
  - scalar → array
- context switch → 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 → function, run to completion
- global memory copy for each round
  - scalar → array
- context switch → 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
type_g1 g1; type_g2 g2; …




//thread functions
t(){
  type_x1 x1; type_x2 x2; …
  stmt_1 ;
  stmt_2 ;
  …
} …

main(){
  …
}
```

```
//shared vars
type_g1 g1[K]; type_g2 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(){
  type_x1 x1; type_x2 x2; …
  cs(); if (ret) return; stmt_1[round];
  cs(); if (ret) return; stmt_2[round];
  …
} …

main_thread(){
  …
}

main(){ … }      //next slide
```

LR sequentialization - implementation

```
main(){
   type_g1 _g1[K]; type_g2 _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**