

Analysis of Concurrent Programs via Sequentializations

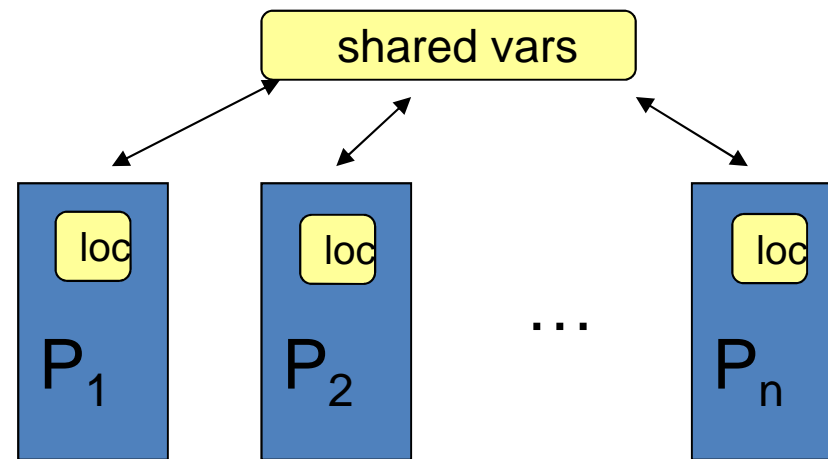
Salvatore La Torre



*Dipartimento di Informatica
Università degli Studi di Salerno*

Concurrent (shared-memory) Programs

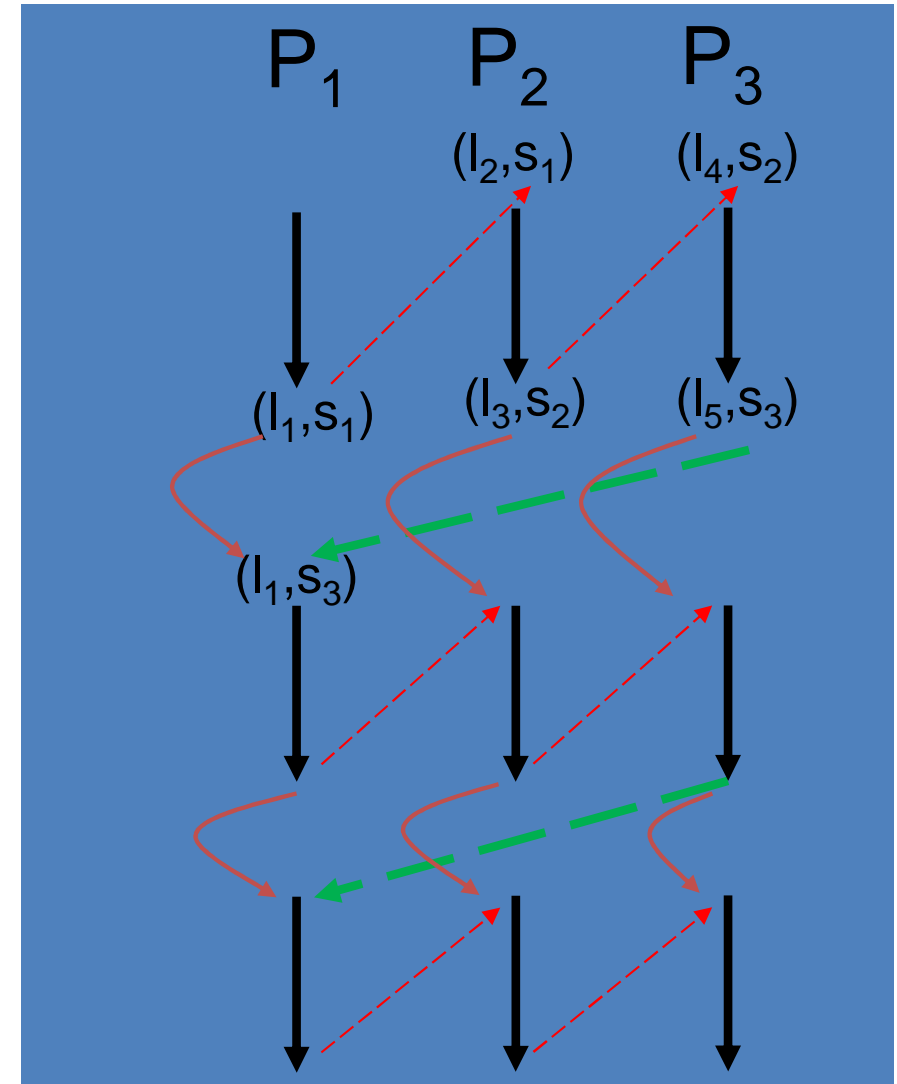
- Formed of sequential programs P_1, \dots, P_n
(each possibly with recursive function calls)



- Each program P_i can read and write shared vars
- We assume sequential consistency
(writes are immediately visible to all the other programs)
- An execution is an interleaving of the executions of each program P_i

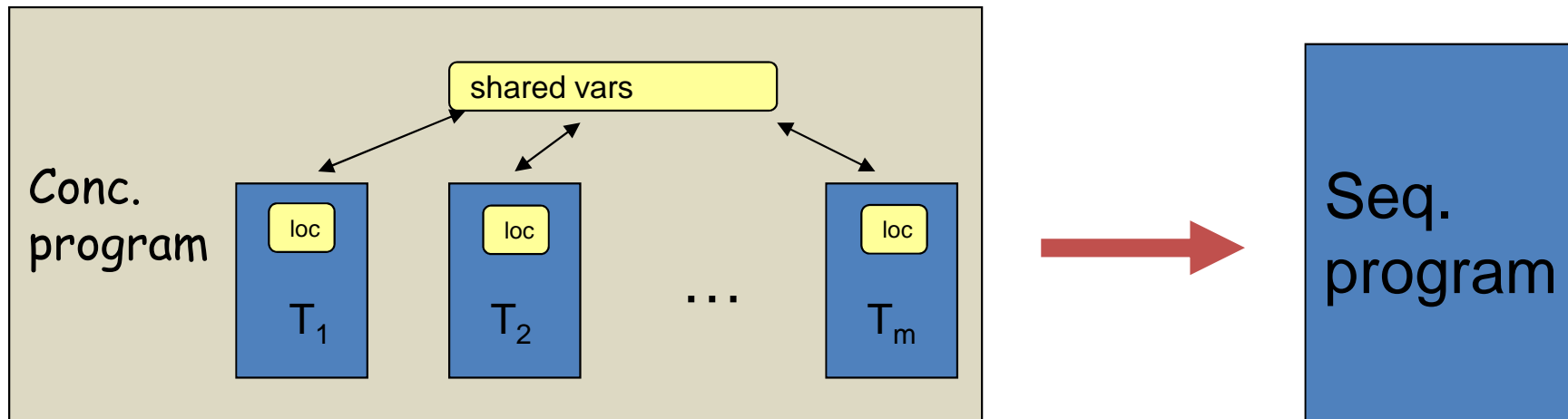
A concurrent execution (n=3)

- Programs are round-Robin scheduled in several rounds
 - **round**: formed of a context of each program
 - **context**: portion of run of a P_i
 - **context-switch**: active thread changes (**global state** is passed on to the next scheduled thread)
 - context-switching back to a thread resumes its **local state**

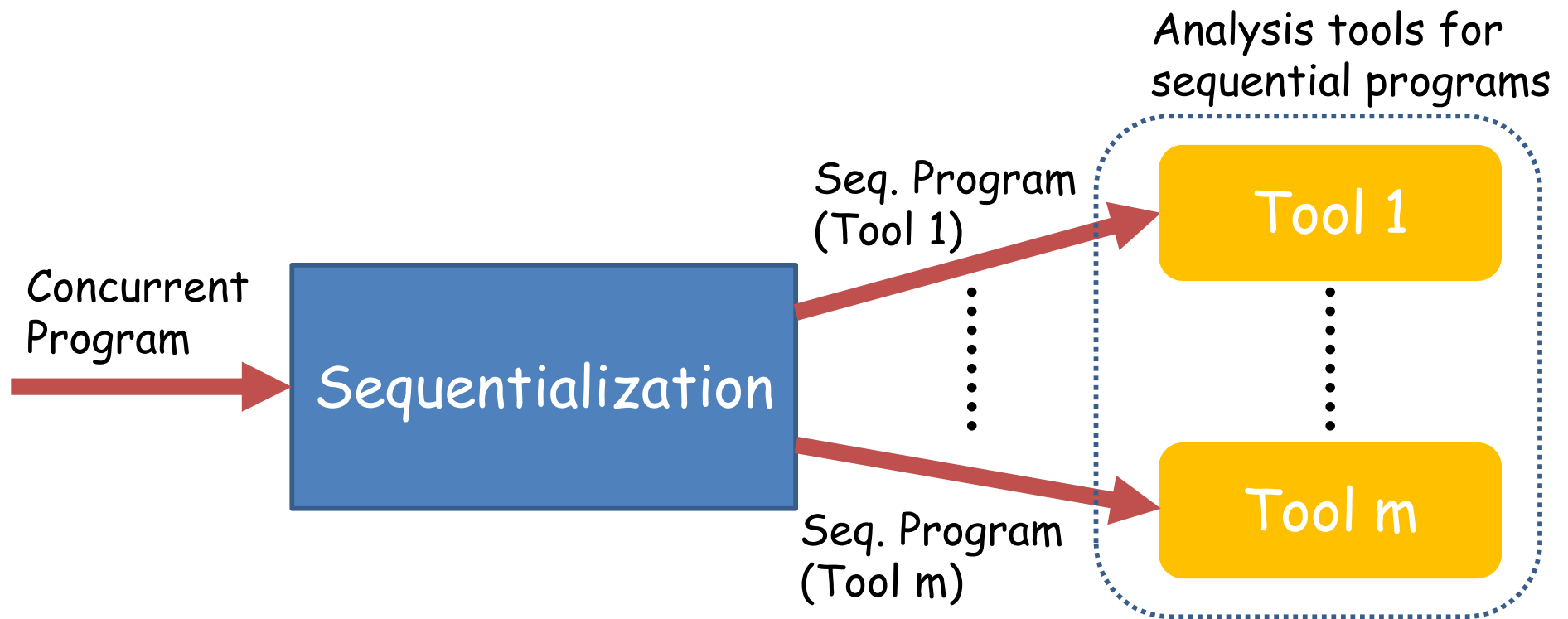


Sequentialization

- Code-to-code translation from a multithreaded program to an "equivalent" sequential one



Why sequentializing?



- Re-use of existing tools (delegate the analysis to the backend tool)
- Fast prototyping (designers can concentrate only concurrency features)
- Can work with different backends

Is this practical?

- Sequentializations inject control code in the original program
 - this can cause some overhead
 - performances of different translations may differ depending on the backend technology
- In the software verification competition (concurrency category) held at TACAS 2014, **gold** and **silver** medal went to tools using **sequentializations**
 - Lazy-CSeq and MU-CSeq
(will be described in the talk of June 5)

Some general observations

- Sequentialization is always possible using unbounded resources
 - Sequential program keeps the call-stacks and just executes threads in time-sharing for any scheduling
- Efficient sequentialization yields an under-approximation of the concurrent programs
 - use prioritized search strategies (e.g., bounded context-switching [Qadeer-Rehof, TACAS'05])
- Full coverage of the state space in very few cases
 - e.g., program abstractions with only two threads sharing only locks acquired/released under contextual locking [Chadha-Madhusudan-Viswanathan, TACAS'12]

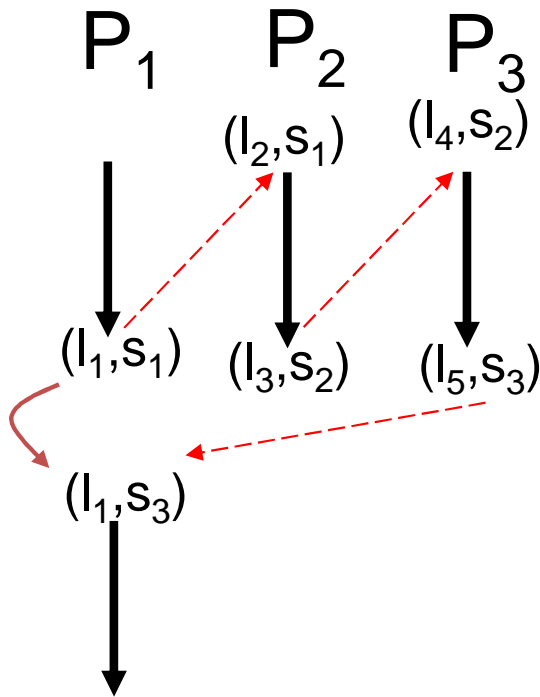
Outline

- First sequentialization
- Bounded context-switching
 - Eager approach
 - Lazy approach
- More sequentializations
- Conclusions

A first sequentialization

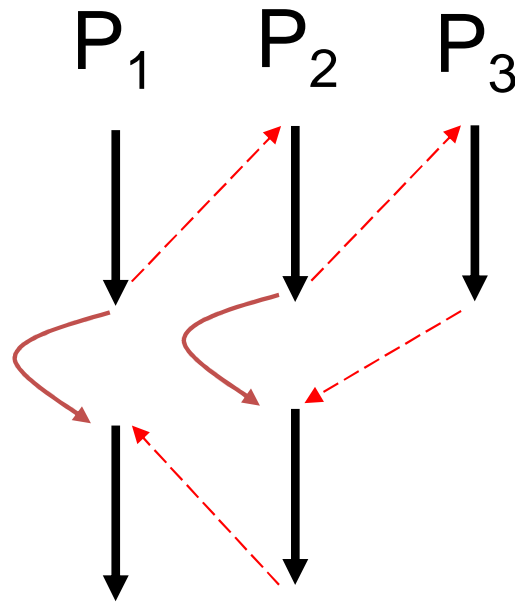
- KISS: Keep It Simple and Sequential (Microsoft tool) [Quadeer-Wu, PLDI'04]
- At context-switches either:
 - the active thread is terminated or
 - a not yet scheduled thread is started (by calling its main function)
- When a thread is terminated either:
 - the thread that has called it is resumed (if any) or
 - a not yet scheduled thread is started

Example (n=3)



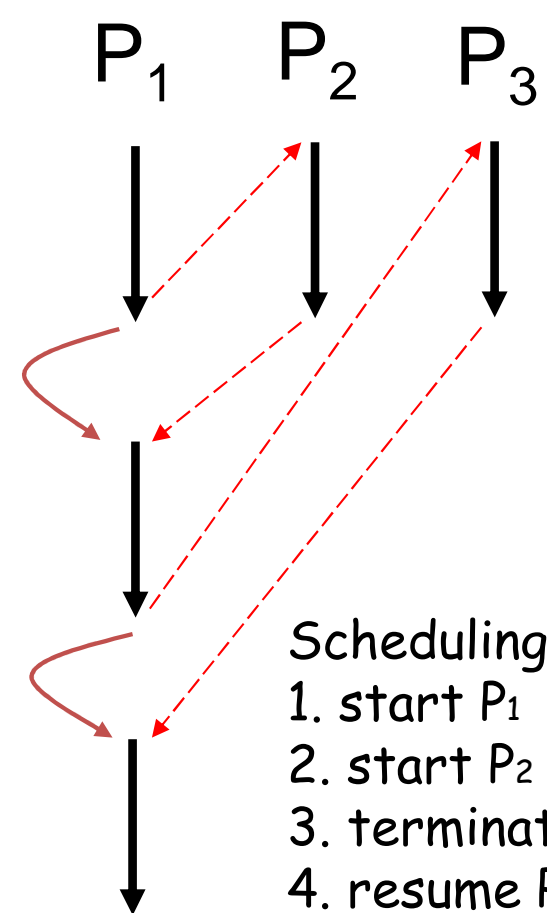
Scheduling 1:

1. start P_1
2. start P_2
3. terminate P_2
4. start P_3
5. terminate P_3
6. resume P_1



Scheduling 2:

1. start P_1
2. start P_2
3. start P_3
4. terminate P_3
5. resume P_2
6. terminate P_2
7. resume P_1



Scheduling 3:

1. start P_1
2. start P_2
3. terminate P_2
4. resume P_1
5. start P_3
6. terminate P_3
7. resume P_1

More on KISS

- Allows dynamic thread allocation in form of asynchronous calls
- Bounds the number of threads that have been created but not started yet
 - Scheduler starts a thread from this set (choosing it nondeterministically) or resumes the last suspended thread (if any)
- Used for assertion checking

Outline

✓ First sequentialization

- Bounded context-switching
 - Eager approach
 - Lazy approach
- More sequentializations
- Conclusions

Bounded context-switching

- Switching between threads is allowed only a bounded number of times [Qadeer-Rehof, TACAS'05]

Under this restriction

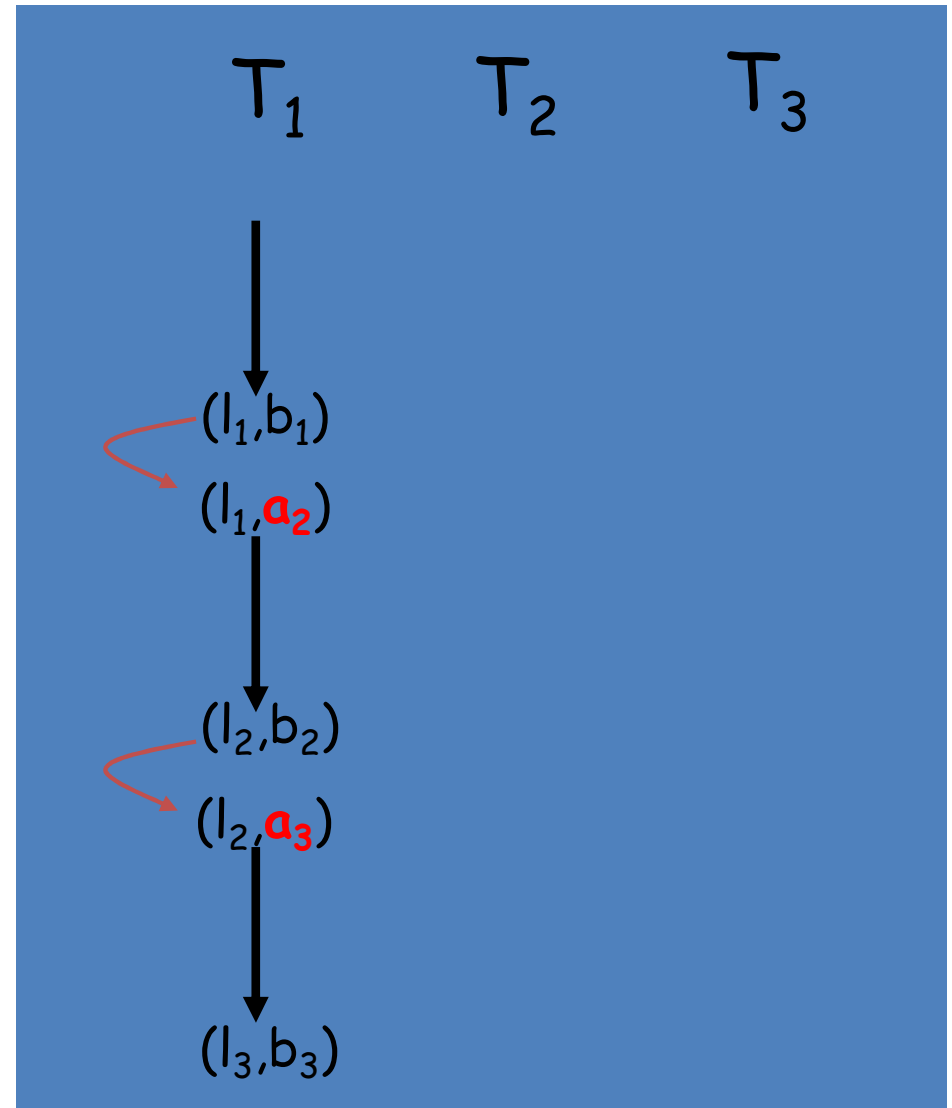
- Analysis is an effective technique for bug detection
 - bugs of concurrent programs are likely to occur within few context-switches [Musuvathi-Qadeer, PLDI'07]
 - Efficient sequentializations can be obtained
 1. Eager approach [Lal-Reps, CAV'08]
 2. Lazy approach [La Torre-Madhusudan-Parlato, CAV'09]

Eager sequentialization

- [Lal-Reps, CAV'08]

Sequential program (k-rounds)

1. Guess a_2, \dots, a_k
2. Execute T_1 to completion
 - Computes local states l_1, \dots, l_k and global states b_1, \dots, b_k

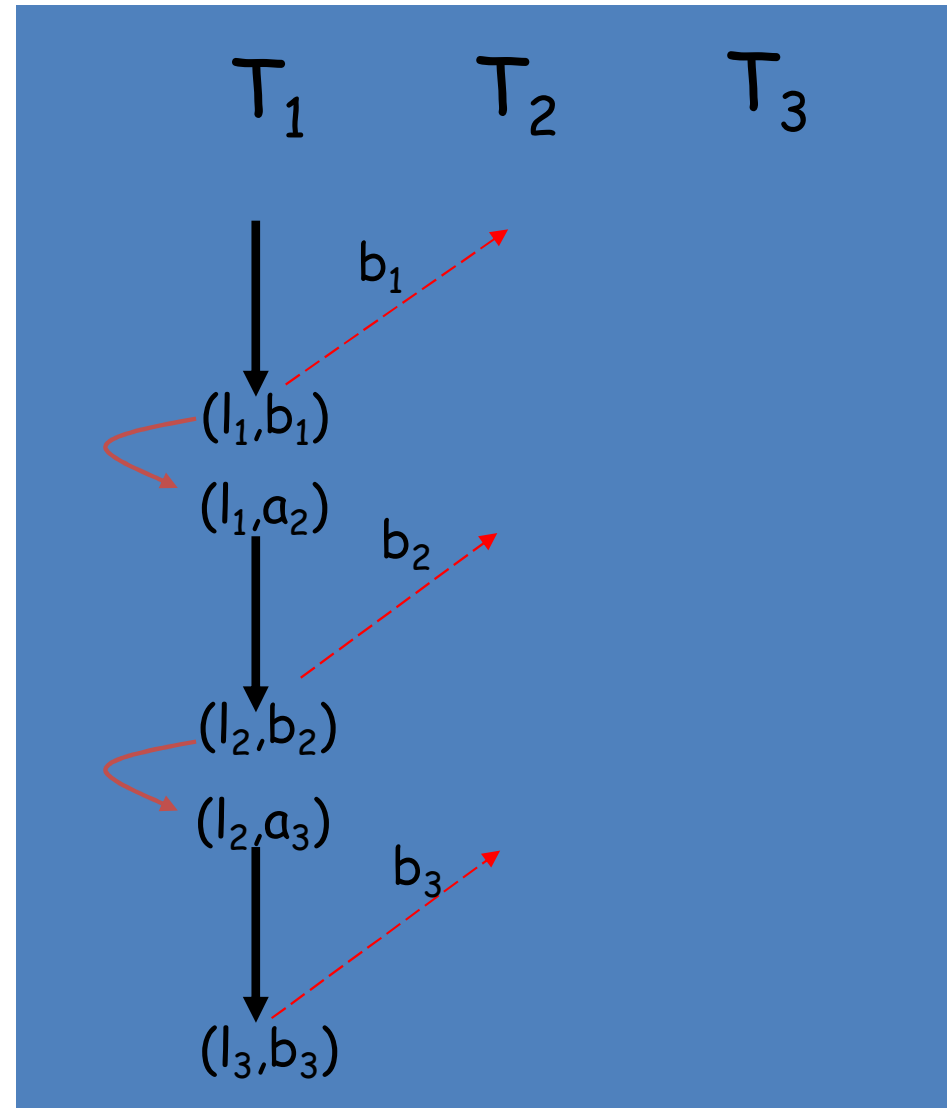


Eager sequentialization

- [Lal-Reps, CAV'08]

Sequential program (k-rounds)

1. Guess a_2, \dots, a_k
2. Execute T_1 to completion
3. Pass b_1, \dots, b_k to T_2

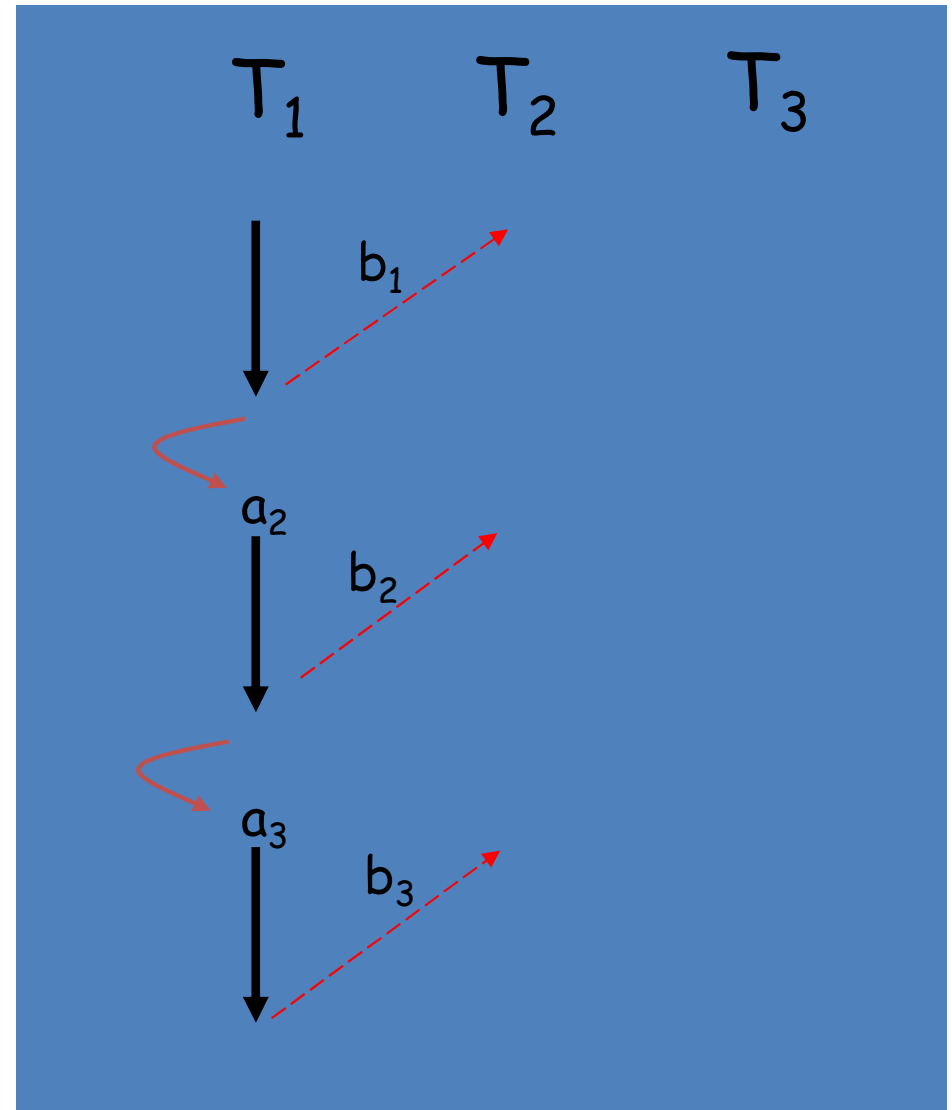


Eager sequentialization

- [Lal-Reps, CAV'08]

Sequential program (k-rounds)

1. Guess a_2, \dots, a_k
 2. Execute T_1 to completion
 3. Pass b_1, \dots, b_k to T_2
- We can forget of locals

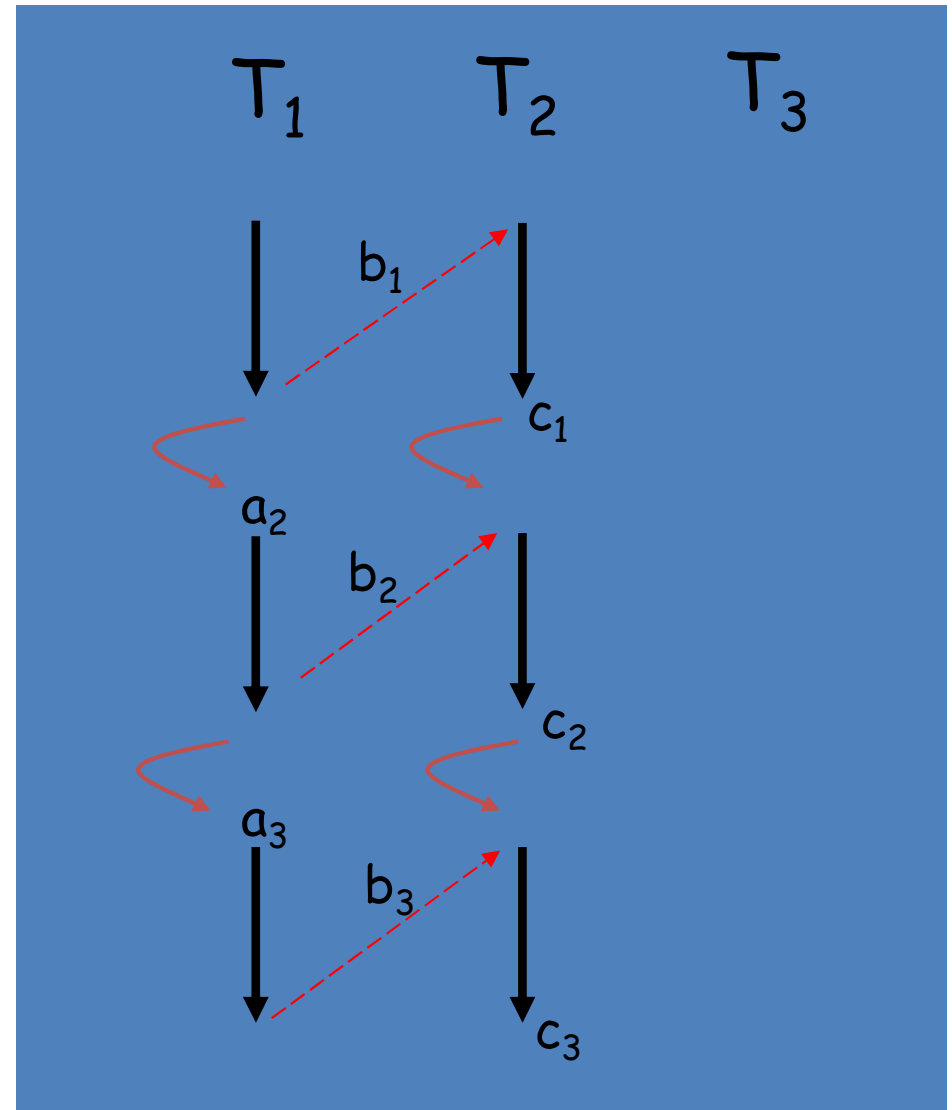


Eager sequentialization

- [Lal-Reps, CAV'08]

Sequential program (k-rounds)

1. Guess a_2, \dots, a_k
2. Execute T_1 to completion
3. Pass b_1, \dots, b_k to T_2
4. Execute T_2 to completion

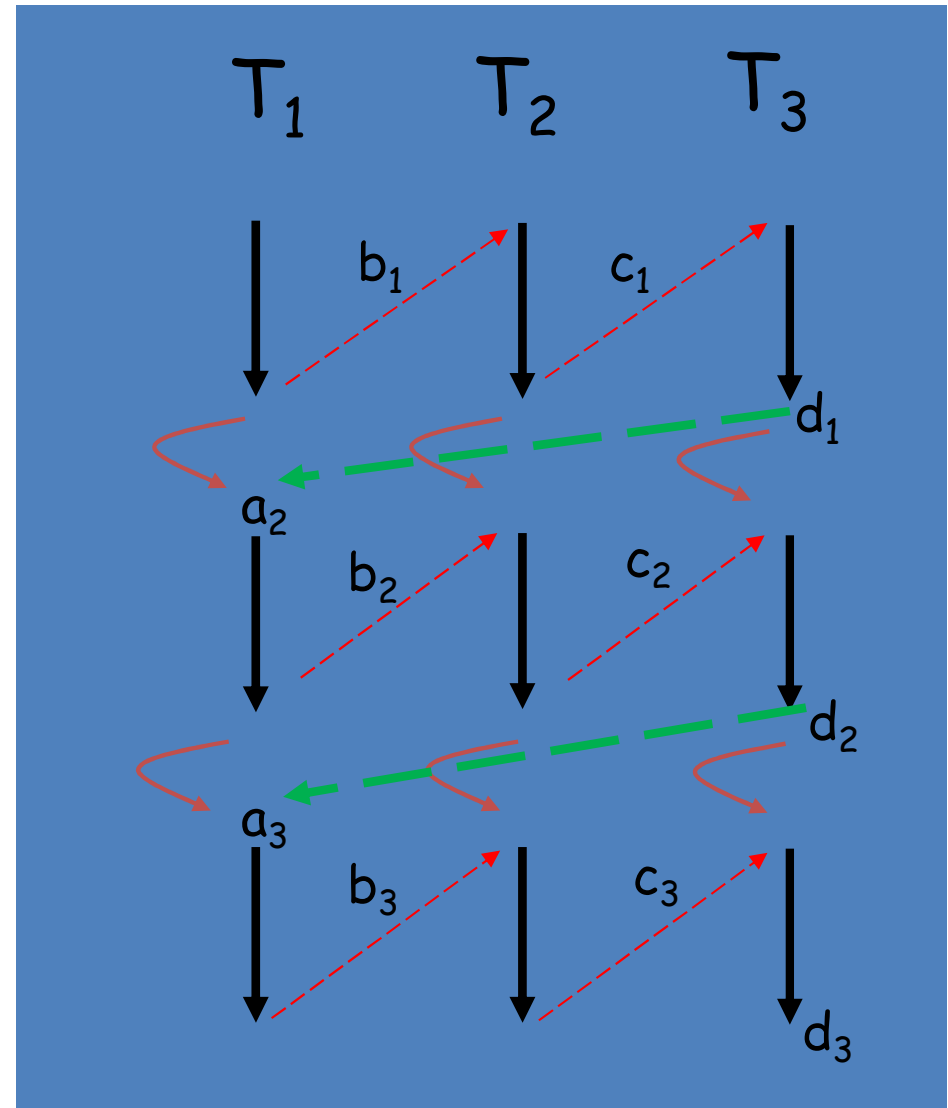


Eager sequentialization

- [Lal-Reps, CAV'08]

Sequential program (k-rounds)

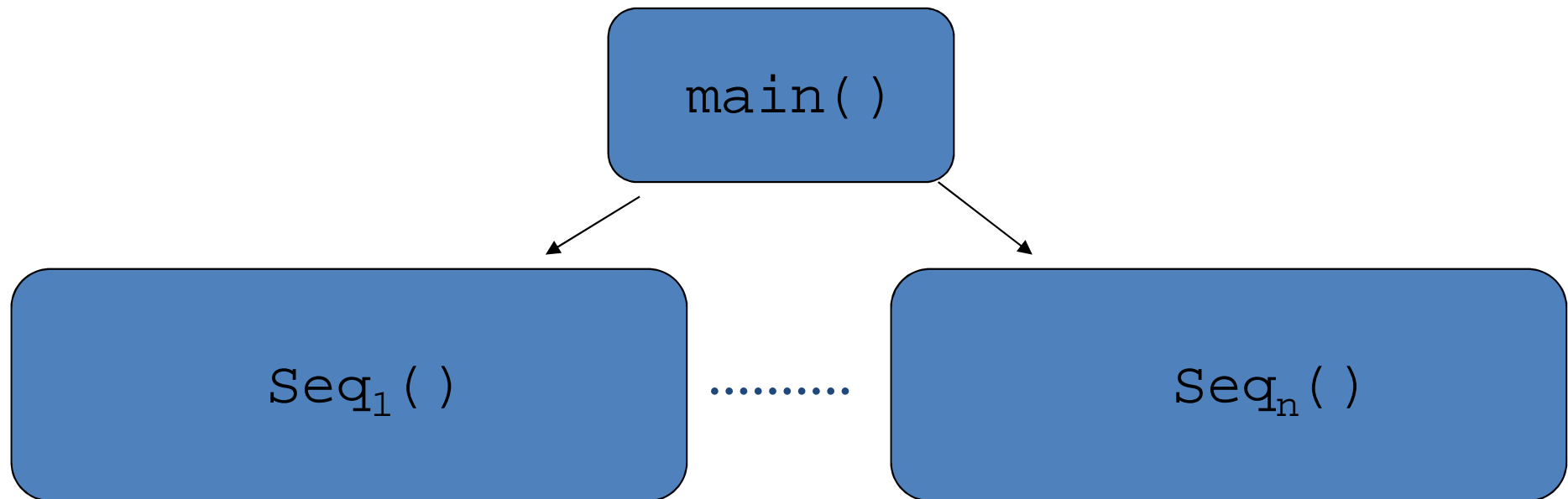
1. Guess a_2, \dots, a_k
2. Execute T_1 to completion
3. Pass b_1, \dots, b_k to T_2
4. Execute T_2 to completion
5. Pass c_1, \dots, c_k to T_3
6. Execute T_3 to completion
7. Computation iff $d_i = a_{i+1} \ \forall i \in [1, k-1]$



Translation scheme

Input: concurrent program P_1, \dots, P_n

Output is a sequential program consisting of:
(Seq_i is the translation of P_i)



Eager translation (k-rounds)

- $2k-1$ copies of shared vars
 - r_2, \dots, r_k (store guessed starting values)
 - s_1, \dots, s_k (copies per round of shared vars)

- main is very simple:

guess r_2, \dots, r_k

$Seq_1()$

.....

$Seq_n()$

Checker()

Error()

- $Seq_i()$:
 - code of P_i using the copy s_j at round j
 - implements round-switching by moving to next copy of shared vars
 - returns to main after last round
- Checker():
 - for $i = 1$ to $K - 1$ do
 - assume ($s_i = r_{i+1}$)
- Error(): assume(goal)

Outline

- First sequentialization
- Bounded context-switching
 - Eager approach
 - Lazy approach
- More sequentializations
- Conclusions

Eager seq. does not preserve assertions

```
// shared variables  
bool blocked=true;  
int x=0, y=0;
```

```
process P1:  
  main() begin  
    while (blocked)  
      skip;  
    assert(y!=0);  
    x = x/y;  
  end
```

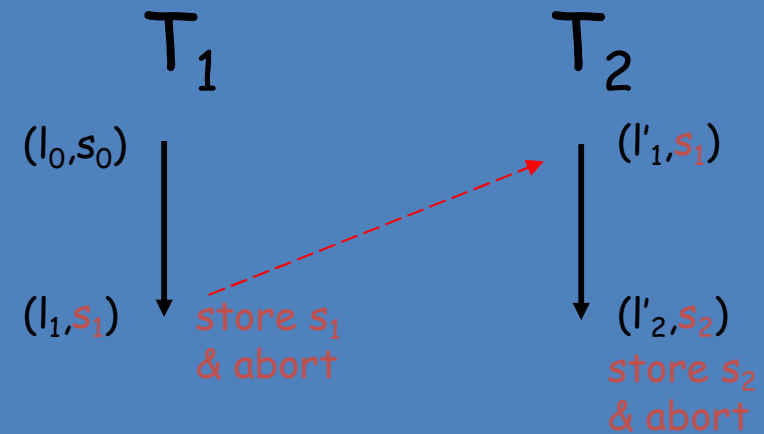
```
process P2:  
  main() begin  
    x=12;  
    y=2;  
    //unlock threads of P1  
    blocked=false;  
  end
```

- $y \neq 0$ is an invariant of the statement $x = x/y$ in the concurrent progr.
 - but not in the sequential program
(blocked can be nondeterministically assigned to false across a context-switch while processing P1)

Lazy transformation: main idea

[La Torre-Madhusudan-Parlato, CAV'09]

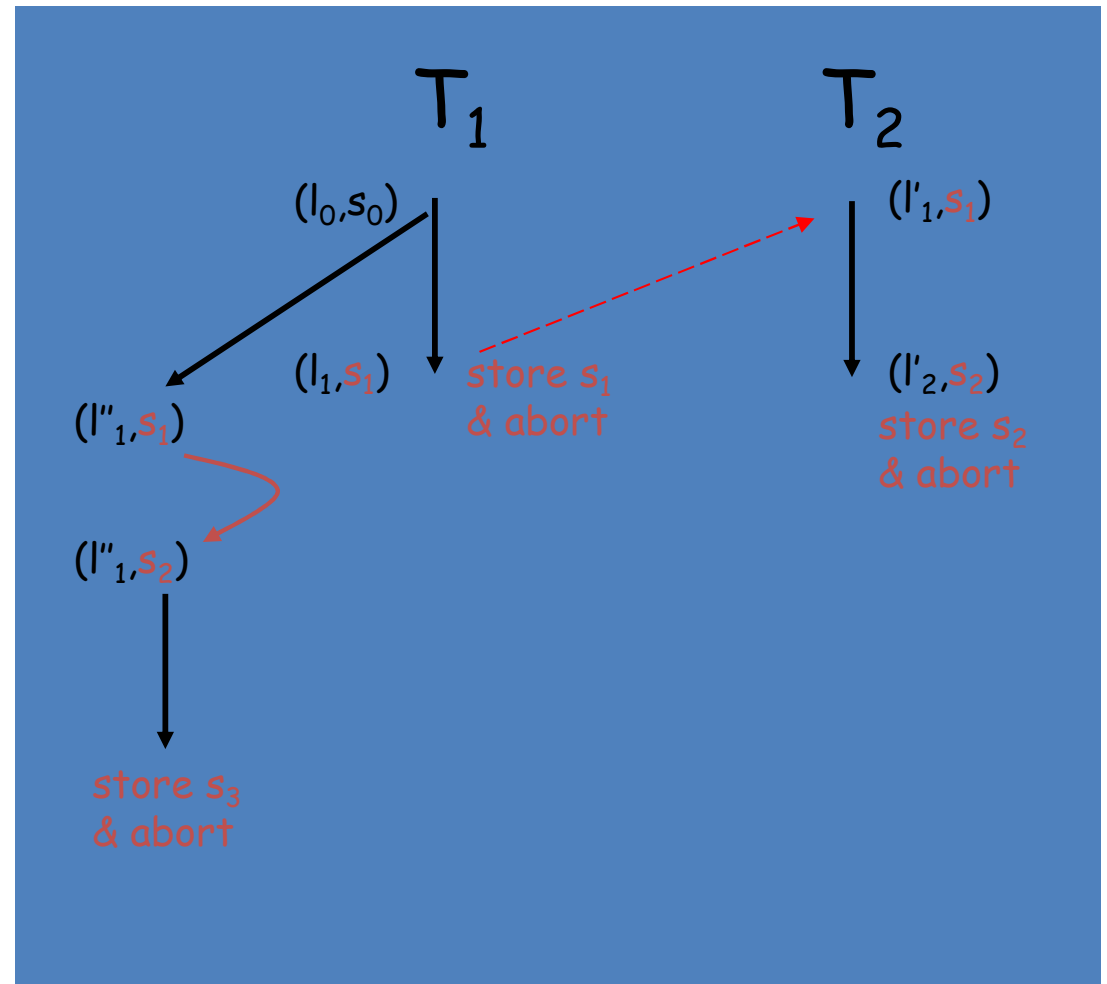
- Execute T_1
- Context-switch:
store s_1 and abort
- Execute T_2 from s_1
- store s_2 and abort



Lazy transformation: main idea

[La Torre-Madhusudan-Parlato, CAV'09]

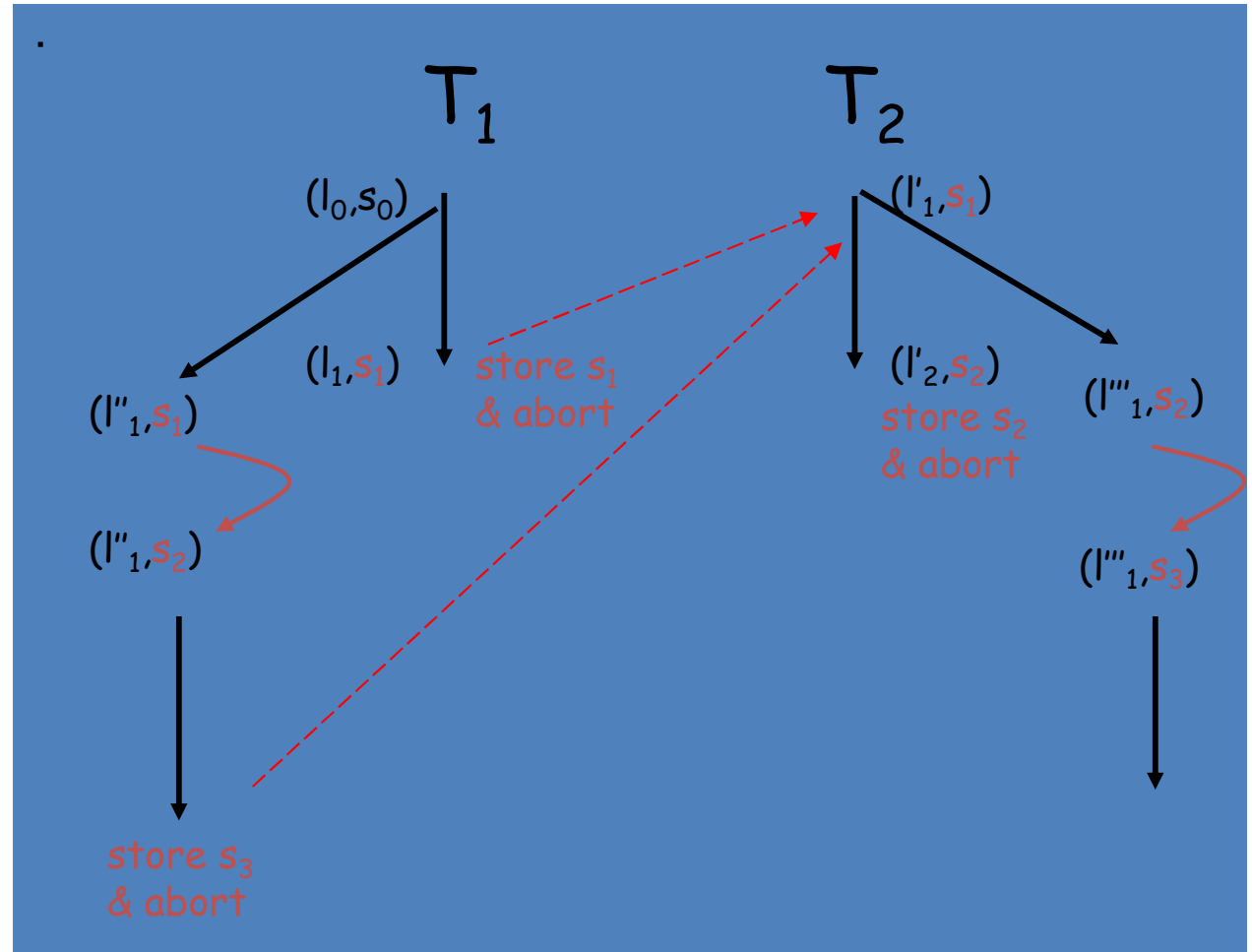
- Re-execute T_1 till it reaches s_1
 - May reach a new local state!
 - Anyway it is correct !!
- Restart from global s_2 and compute s_3



Lazy transformation: main idea

[La Torre-Madhusudan-Parlato, CAV'09]

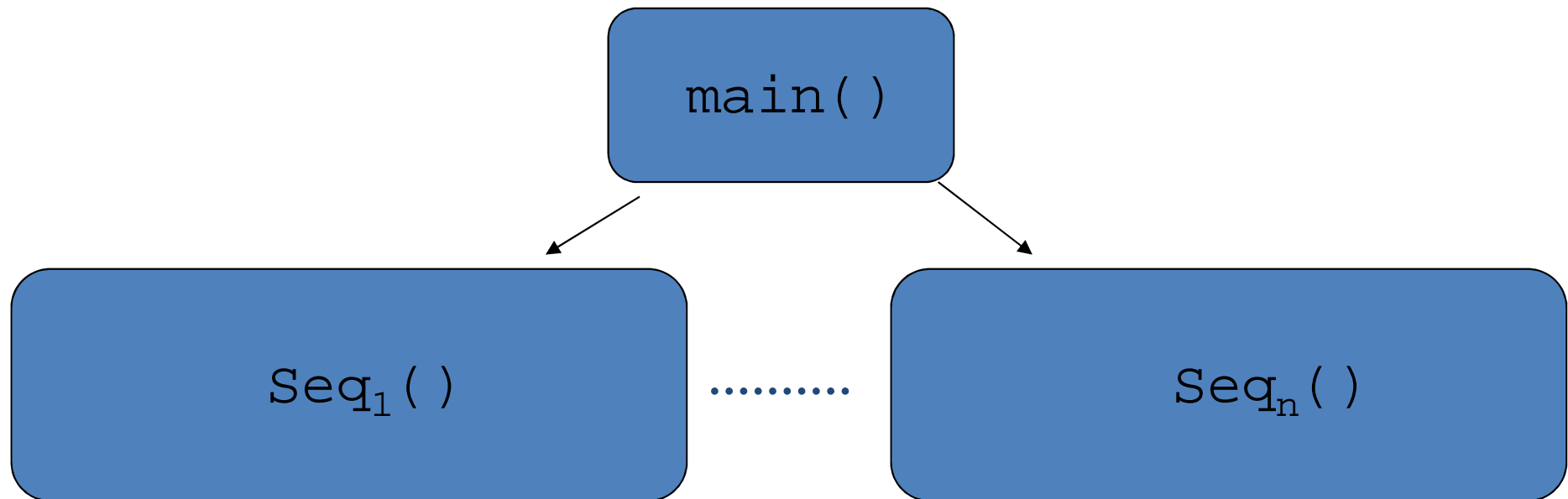
- Switch to T_2
- Execute till it reaches s_2
- Continue computation from global s_3



Translation scheme (as in Eager)

Input: concurrent program P_1, \dots, P_n

Output is a sequential program consisting of:
(Seq_i is the translation of P_i)



Lazy translation (k-contexts)

- k copies of shared vars
 - s_1, \dots, s_k (copies of shared vars to store values at cs)
- main has more control stms:
 - No guessing
 - Keeps track of the current context
 - Starts a thread or its recomputation by assigning the values of sh. vars at first of its contexts

Lazy translation (k-contexts)

- $Seq_i()$:
 - code of P_i interleaved with control code

if (terminate) then return;
else
 if ($*$) then call *contextSwitch*();
 if (terminate) then return;
- No special handling of error condition

- *contextSwitch*()
 - when recomputing contexts:
 1. matches values at cs
 2. set starting values for next context
 - when context-switching out the currently new computed context
 1. stores the sh vars in the appropriate copy
 2. set *terminate* to true

Summarizing lazy translation

- Explores only reachable states
- Preserves invariants across the translation
- Tracks local state of one thread at any time
- Tracks values of shared variables at context switches
 (s_1, s_2, \dots, s_k)
- Requires recomputation of local states

Both translations reduce bounded reachability
to sequential reachability

Theorem:

Let C be a concurrent program, $k > 0$ and
 pc be a program counter of C

pc is reachable in C within k context
switches iff pc is reachable in $\text{SeqProg}_k(C)$

Lazy vs. Eager: performace

- Tool Getafix implements both eager and lazy sequentialization for concurrent Boolean programs
- Lazy outperforms Eager in the experiments
- Sample results on Windows NT Bluetooth driver

Context switches	1-adder 1-stopper			2-adders 1-stopper			1-adder 2-stoppers			2-adders 2-stoppers		
		eager	lazy		eager	lazy		eager	lazy		eager	lazy
1	N	0.1	0.1	N	0.2	0.1	N	0.1	0.1	N	0.2	0.1
2	N	0.3	0.2	N	0.9	0.8	N	0.7	0.9	N	1.6	2.0
3	N	43.3	1.4	N	135.9	6.3	Y	70.1	0.4	Y	177.6	0.8
4	N	73.6	5.5	Y	1601.0	2.6	Y	597.2	2.9	Y	out of mem.	7.5
5	N	930.0	20.2	Y	-	18.0	Y	-	14.0	Y	out of mem.	66.5
6	N	-	66.8	Y	-	122.9	Y	-	66.1	Y	out of mem.	535.9

Lazy vs. Eager: performance

- Getafix uses as verification engine a fixed-point logic solver (Mucke)
 - It stores summaries, recomputations do not cause to repeat exploration
 - Explore the state space lazily gives some advantages
- Experiments using BMC (Bounded Model-checking) backends gives the opposite result
 - Eager outperforms Lazy

[Ghafari-Hu-Rakamaric, SPIN'10]

Tools implementing LR seq.

- CSe
- STC
- Successes
 - Corra
 - Poiron [Qu, 2011]
 - [Emmi-Queer-Rakamaric, POPL'11]

Is this the end of [3]

NO!

"Lazy Returns..."
in June 5 talk

Outline

- First sequentialization
- Bounded context-switching
 - Eager approach
 - Lazy approach
- More sequentializations
- Conclusions

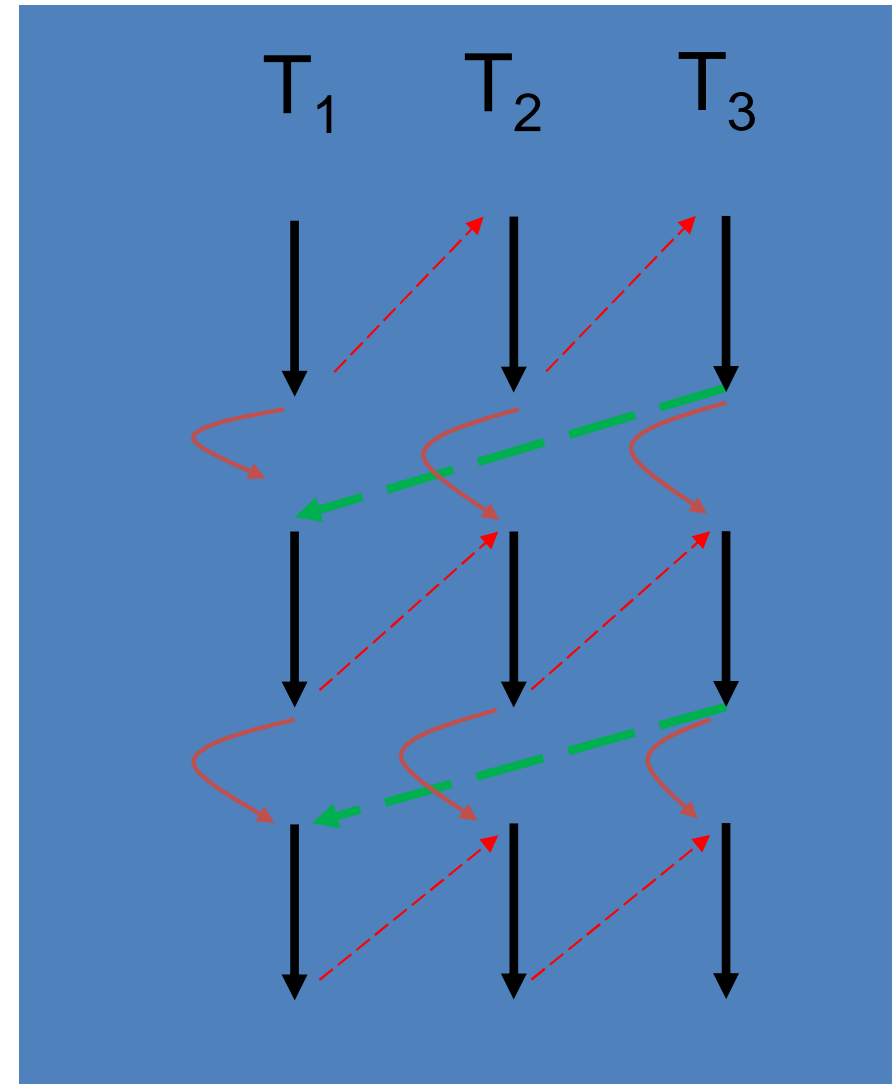
Parameterized programs

- Extend shared-memory concurrent programs
 - Computations can have an arbitrary number of threads
- Complex class of programs (infinite states):
 - each thread can have recursive calls
 - number of threads is unbounded
- Interesting class of programs (e.g., device drivers)
 - can be used to analyze programs with dynamic thread creation

Sequentialization of param. progs

[La Torre-Madhusudan-Parlato, FIT'12]

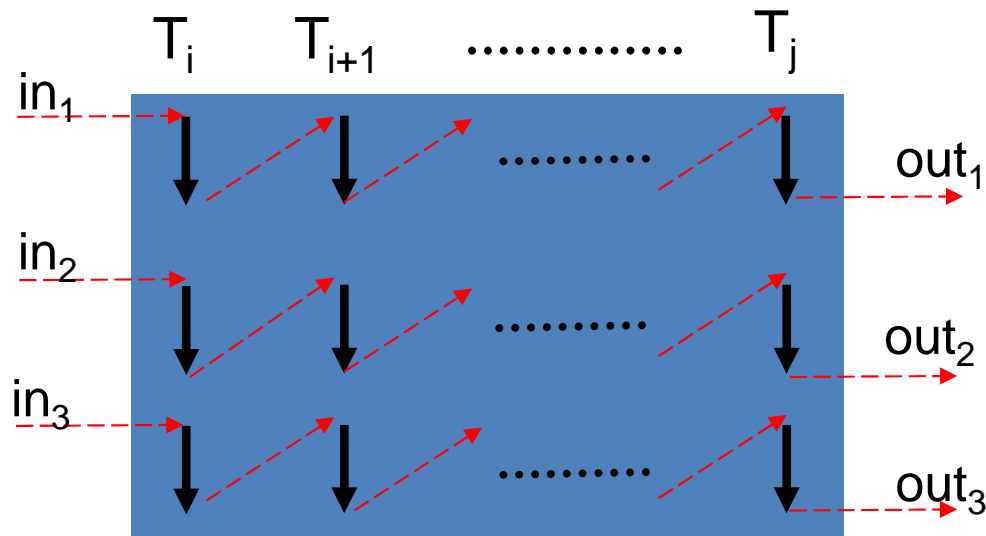
- Eager sequentialization can be easily obtained from that for concurrent programs:
 - each thread is executed up to completion (jumping across context-switches)
 - after computing a thread, nondeterministically (1) terminate and check if all the computed executions form a computation and (2) compute next thread
 - the values of shared variables at context-switches are passed to the next thread



Linear interfaces

- Summarize the effects of a block of unboundedly many threads on the shared variables
 - executions arranged in rounds of round-robin scheduling

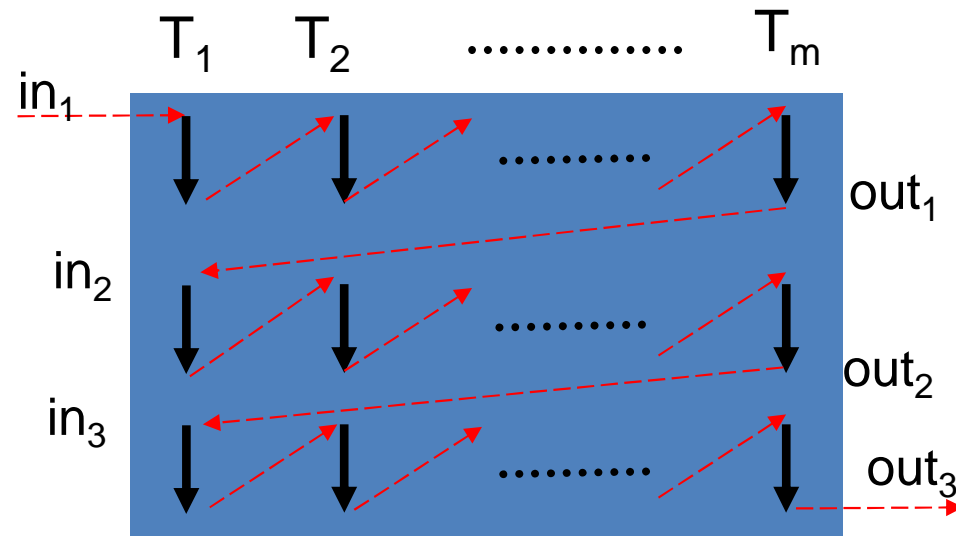
linear interface
(In,Out)
of dim. 3



Linear interface of a run

- (In,Out) s.t. $in_{i+1}=out_i \quad i=1,\dots,k-1$

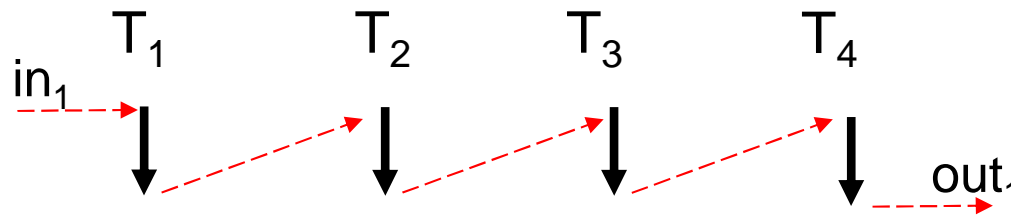
$k=3$



Lazy sequentialization

[La Torre-Madhusudan-Parlato, FIT'12]

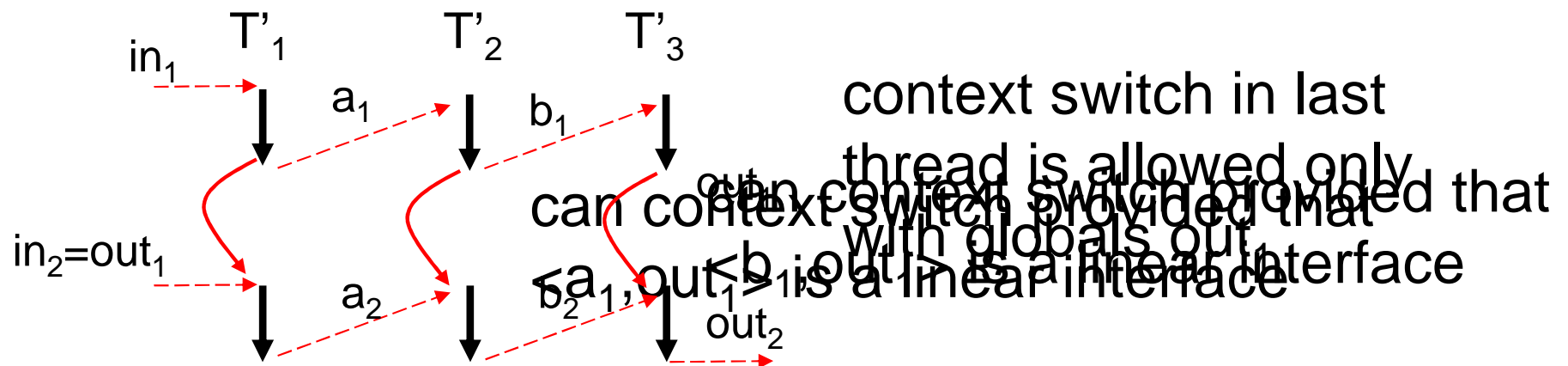
- P_{seq} mimics a computation of P
 - by increasing round numbers and
 - (within each round) by increasing context numbers



- nondeterministically chooses if this is the last thread in the round
- the linear interface $\langle in_1, out_1 \rangle$ is stored

Lazy sequentialization

- Second round is executed matching $\langle in_1, out_1 \rangle$



- Note that threads do not need to be the same we used in the first round and not even in the same number
- The third round is executed similarly by matching $\langle in_1, in_2, out_1, out_2 \rangle$

Dynamic thread creation

- New threads can be instantiated at runtime (e.g., thread creation, asynchronous calls)
- Computations may have unboundedly many threads running at the same time
- Main idea to handle dynamic creation:
 - schedule threads according to a (DFS) visit of the ordered thread-creation tree
 - this allows to use the call stack to explore the pending threads
- This nicely combines with the Eager scheme

Delay-bounded scheduling

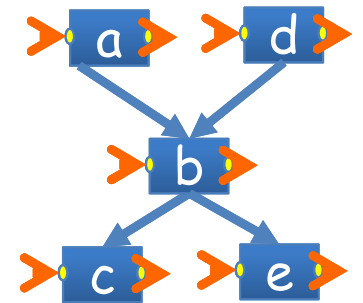
[Emmi-Qadeer-Rakamaric, POPL'11]

- Programs with asynchronous calls (creating tasks)
- Each task is executed to completion (no interleaving with other tasks)
- Sequentialization is according to a DFS scheduler of tasks
- When dispatched, a task can be delayed to next round
 - the total number of delays in a task-creation tree is bounded by k
 - total number of explored rounds is $k+1$
- The beginning of each round is guessed (eager)

General sequentialization

[Bouajjani-Emmi-Parlato, SAS'11]

- Programs with asynchronous calls
- Tasks can be interleaved with other ones
- Sequentialization based on generalization of Linear Interfaces
 - DAGs of contexts
 - Composition and compression operations
- Bound on the size of the DAGs
- Generalizes k-rounds Eager e delay bounded-scheduling sequentialization



Scope-bounded sequentialization

[La Torre-Napoli-Parlato, DLT'14] [La Torre-Parlato, FSTTCS'12]

- No dynamic thread creation
- k-scoped generalizes k-context analysis
 - bounds the number of times a thread is suspended/resumed between each matching call and returns
- Each scope is captured by a linear interface
- Sequentialization maintains a set of linear interfaces (one for each thread)
- Each thread contributes with many LI's in a computation
- Both Eager and Lazy schemes

Outline

- First sequentialization
- Bounded context-switching
 - Eager approach
 - Lazy approach
- More sequentializations
- **Conclusions**

Conclusion

- Sequentialization is an effective approach to analyze concurrent programs
- Main features:
 - Fast prototyping
 - Re-use of mature technologies (tools for sequential programs)
 - Code-to-code translation
 - Introduces some overhead (variables, control code, recursive calls)

Conclusion

- Presented translations:
 - keep track only of the local state of the current thread (no cross product)
 - except for KISS, use # copies of the shared variables depending on the bounding parameter
 - thread creation is implemented with calls
- Eager translations require guessing of values of the shared variables and explore unreachable states
- Lazy translations preserve the invariants and introduces many recursive calls (re-computations)

Conclusions

- Experiments show:
 - Exploring only reachable states impacts positively the size of BDD's in the *Getafix* approach
 - Recursive calls impacts negatively the size of formulas in Bounded Model-Checking backends
- Sequentialization schemes should be targeted to a class of backends

Talk on June 5

- Sequentializations for
Bounded Model Checking backends
- Tool CSeq
<http://users.ecs.soton.ac.uk/gp4/cseq/cseq.html>
- Based on joint work with Bernd Fischer, Omar Inverso, Gennaro Parlato and Ermenegildo Tomasco
 - TACAS-SVCOMP'14, CAV'14 and on-going research