Context-Switch-Directed Verification in DIVINE

MEMICS 2014

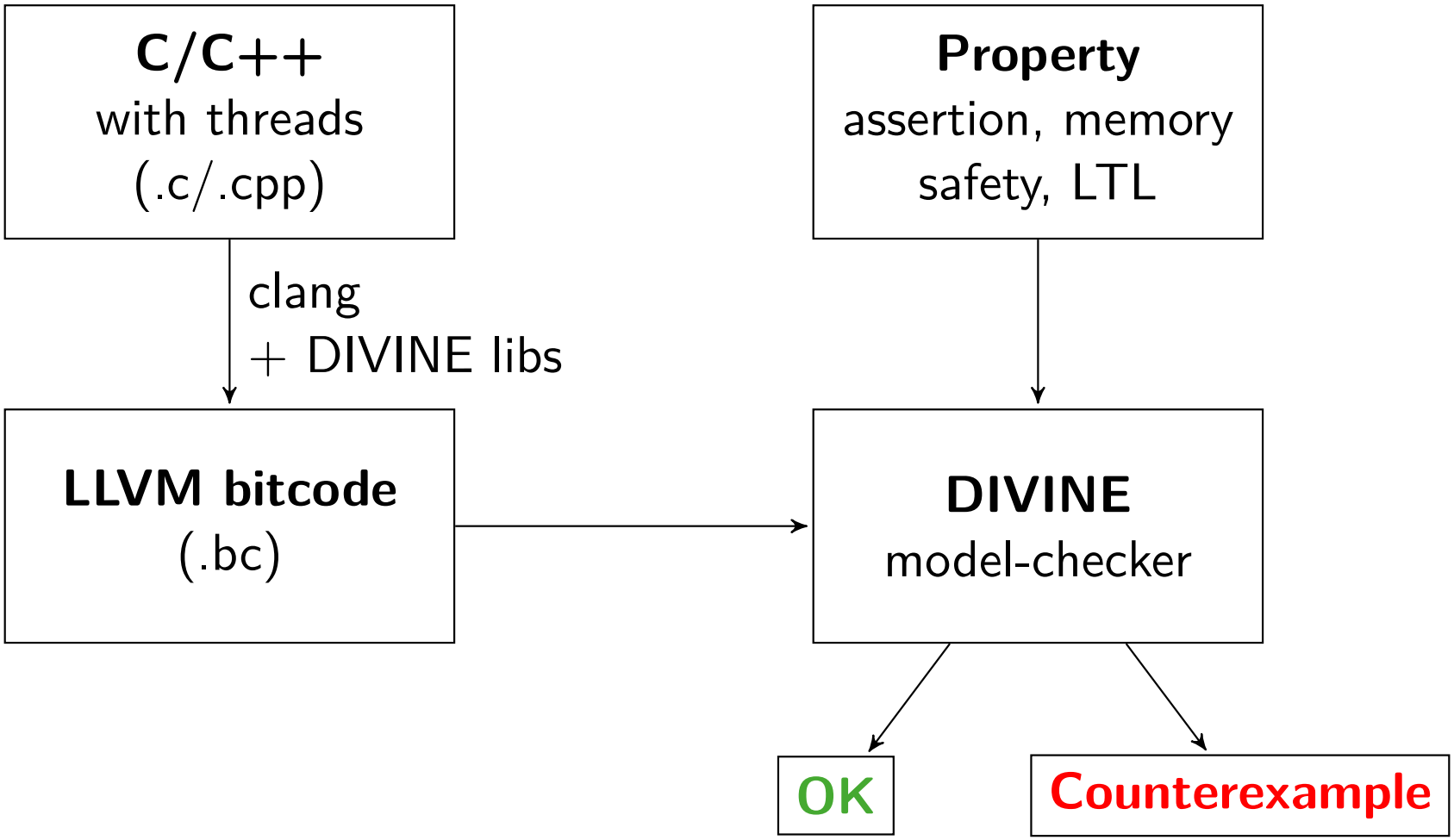
Vladimír Štill Petr Ročkai Jiří Barnat

Faculty of Informatics

Masaryk University, Brno

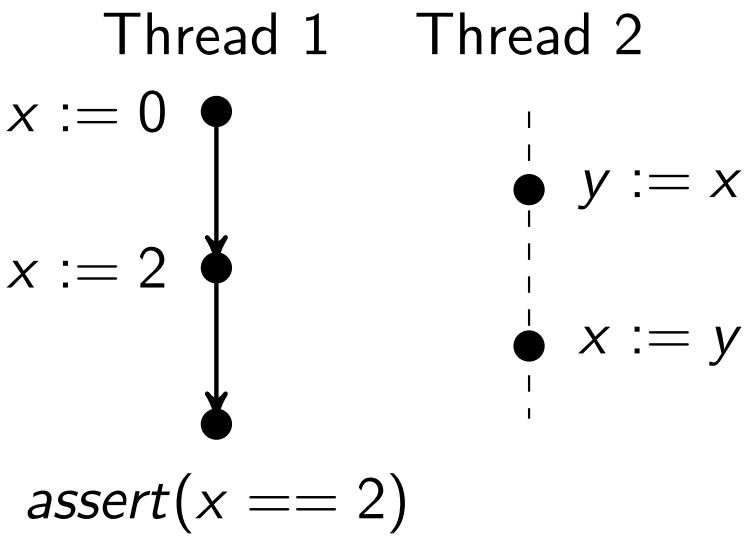
October 18, 2014

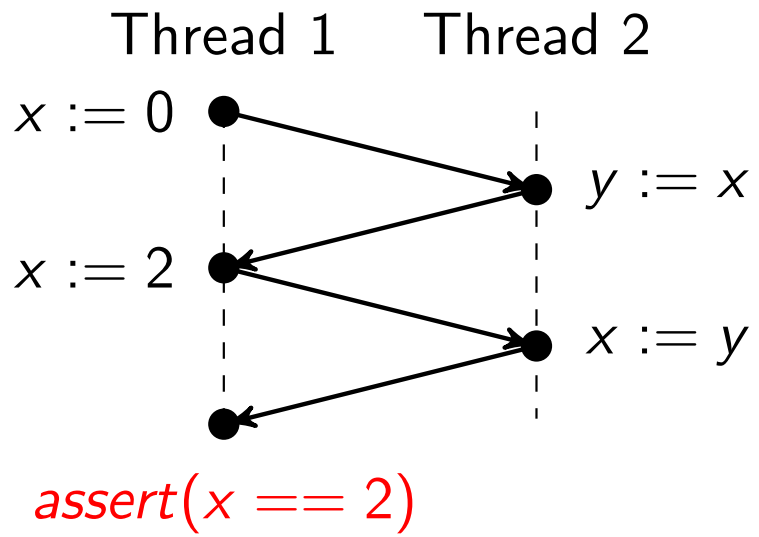
Introduction



Automatic verification of real-world parallel C & C++ unit tests and programs

- race conditions are hard to detect (depend on timing)
  - nondeterministic test results
  - one failure in several thousand runs
- explicit-state model checking can explore all possible outcomes
- produces counterexample = path to point in the program violating the property
  - should be short and simple



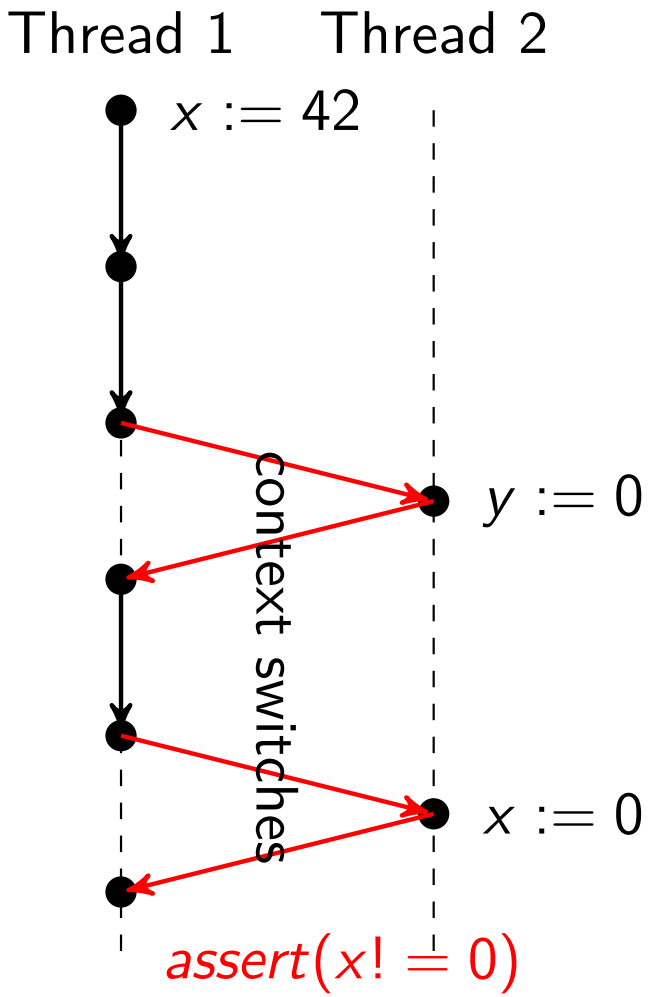


Model checking of real-world parallel C++ programs in DIVINE: what we have

- explicit-state model checking
- support for C and C++ including C++11, pthreads (using Low Level Virtual Machine bitcode)
- use case: parallel unit test $\rightarrow$ detect assertion, memory failure, mutex deadlocks and Linear Temporal Logic violations in any possible interleaving
- gives a counterexample if property does not hold
- time and memory consuming – multiple reduction strategies implemented

Goal of this work

- improve on-the-fly exploration efficiency
- improve counterexample readability
  - should be short
  - should not contain unnecessary context switches
- we primarily target safety properties
- without negatively affecting verification of bug-free programs



Bounded Context Switch Model Checking as an inspiration

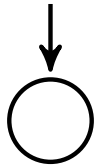
- well established (especially) in symbolic model checking
- limit number of context switches to some preset value
- idea: common bugs usually require only a few context switches
- state space size is reduced
- may not find all bugs

Context-Switch-Directed Exploration in DIVINE

Our contribution: Context-Switch-Directed Reachability

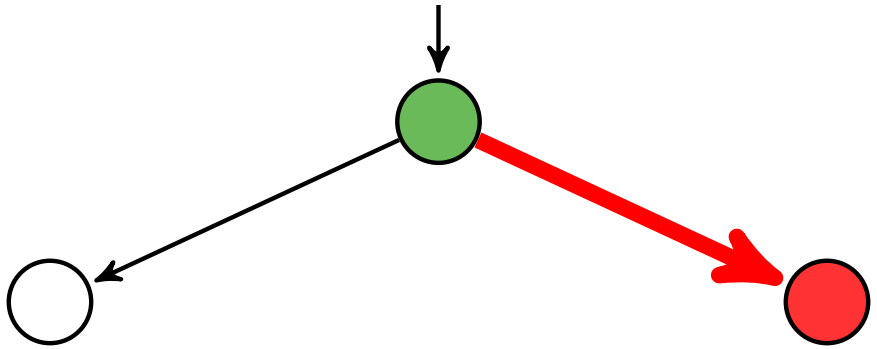
- used as exploration heuristic for explicit-state model checking
- implemented for C & C++ models in latest development version of DIVINE
- alternative to parallel breath-first-search-based reachability
- explores state space in layers
- explores in parallel
- number of context switches can be limited (unlimited by default)

Context-Switch-Directed Reachability: simplified example



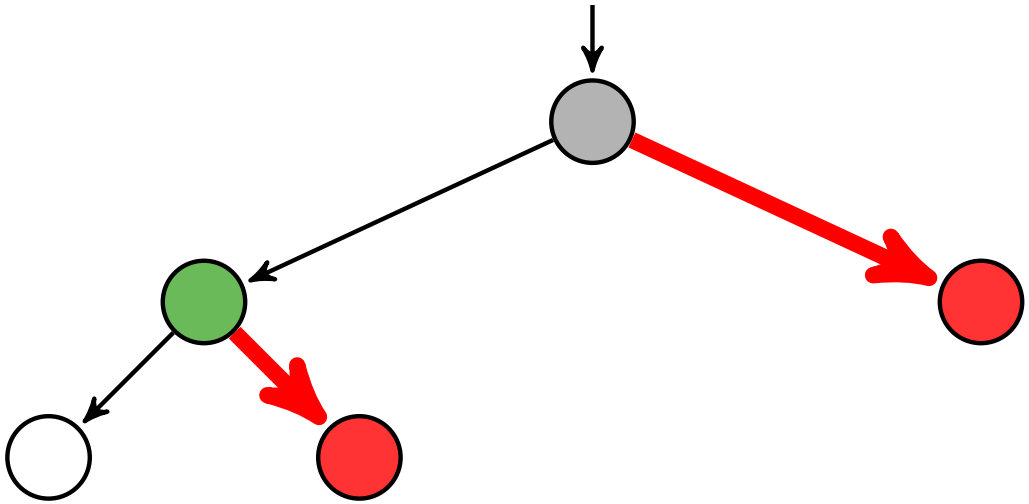
Layer: 0

Context-Switch-Directed Reachability: simplified example



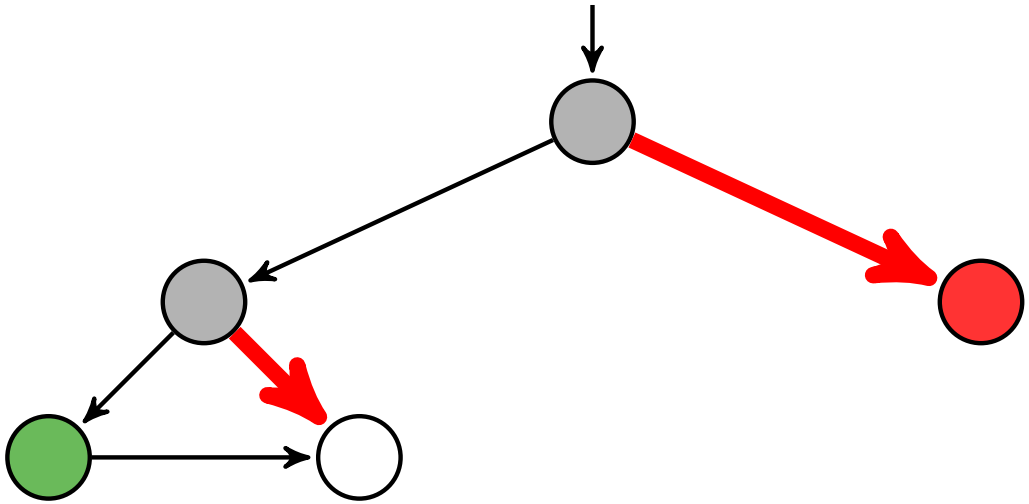
Layer: 0

Context-Switch-Directed Reachability: simplified example



Layer: 0

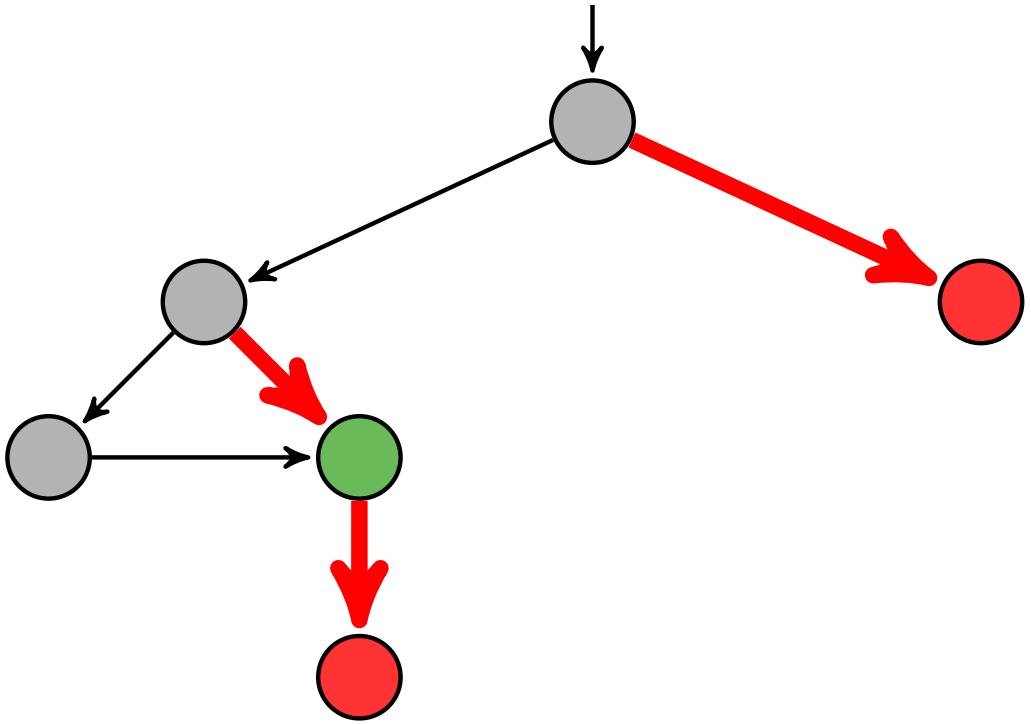
Context-Switch-Directed Reachability: simplified example



Layer: 0

Exploration Order

Context-Switch-Directed Reachability: simplified example

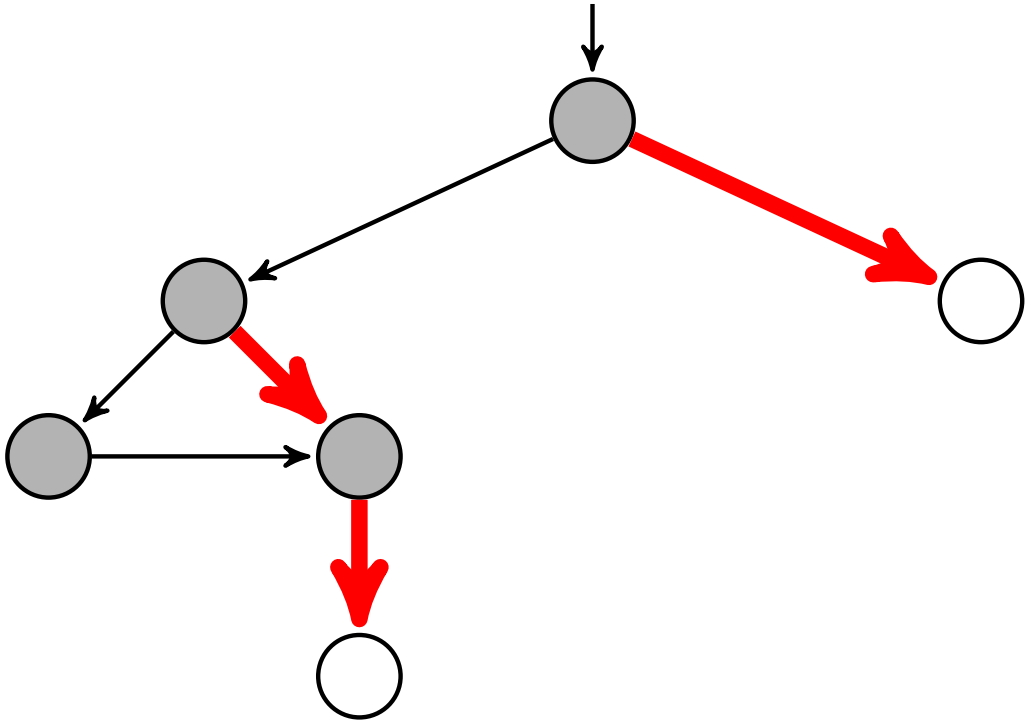


Layer: 0

Layer done

Exploration Order

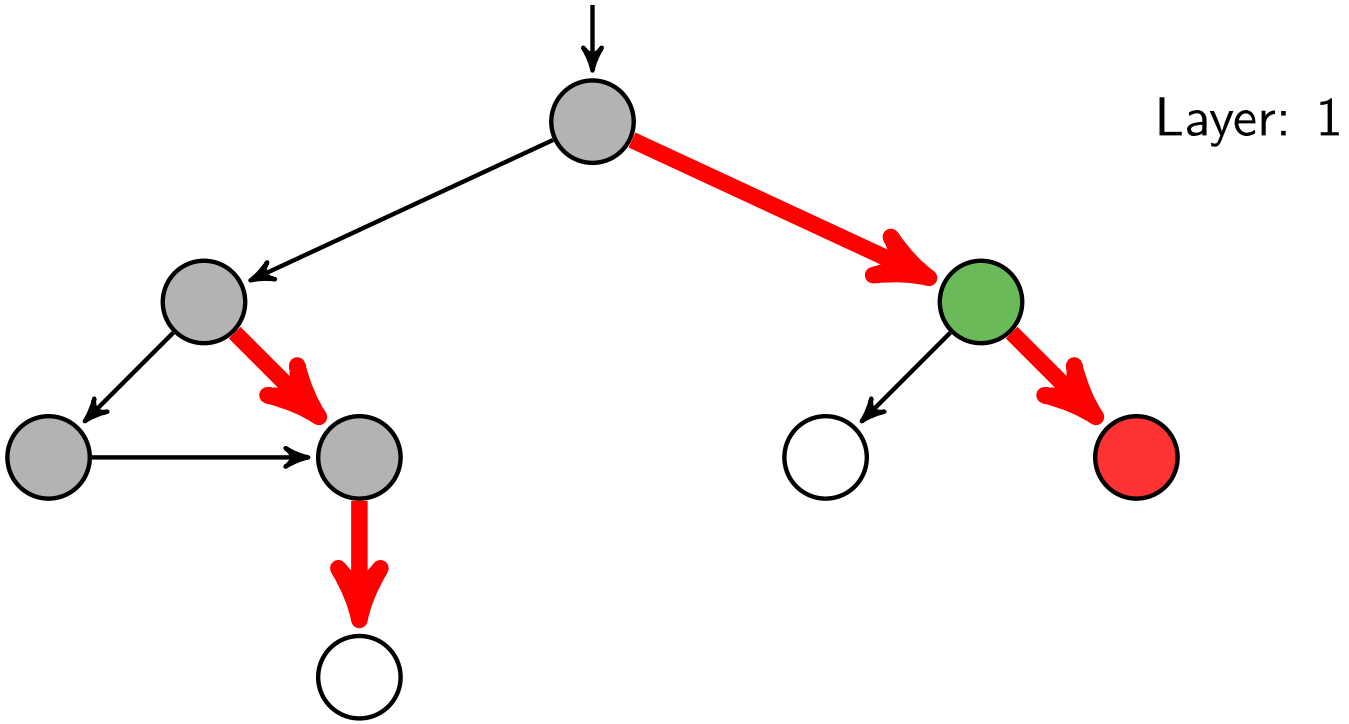
Context-Switch-Directed Reachability: simplified example



Layer: 1

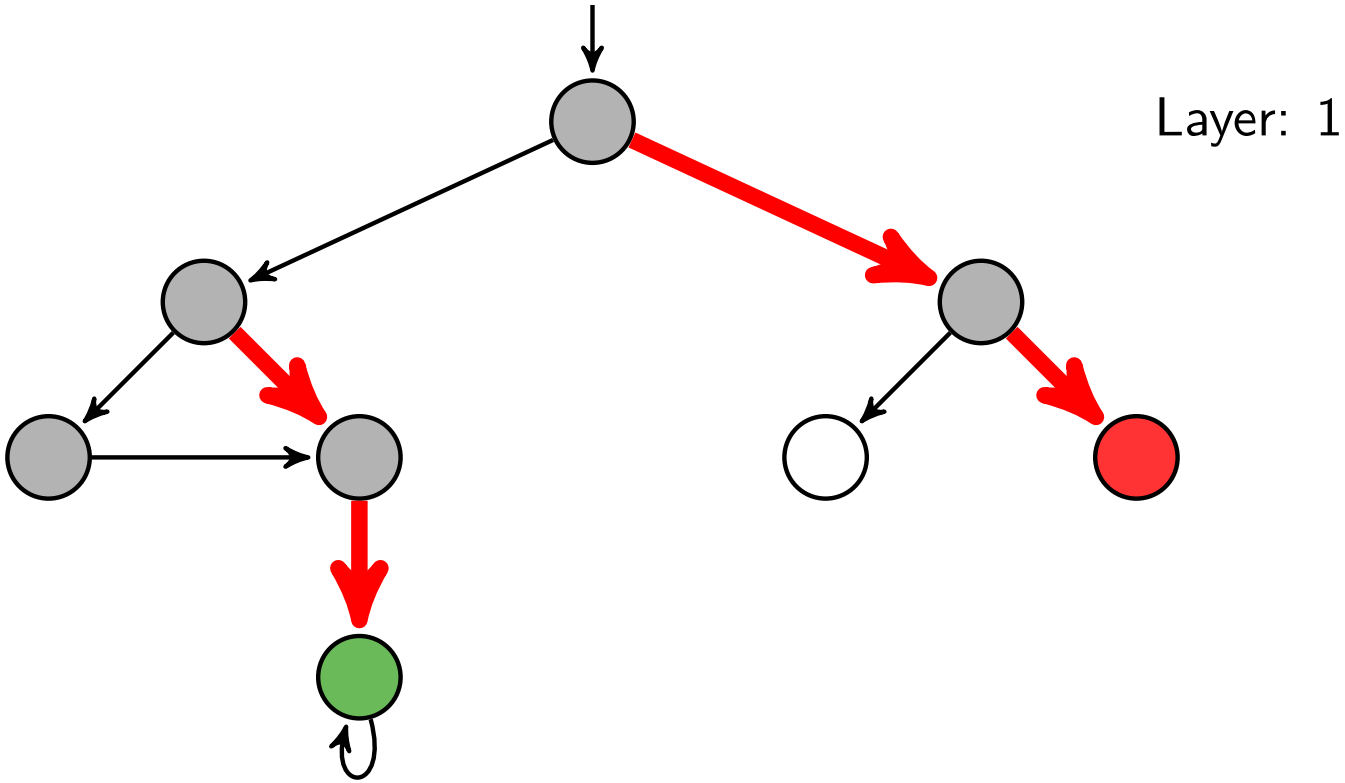
Layer init

Context-Switch-Directed Reachability: simplified example



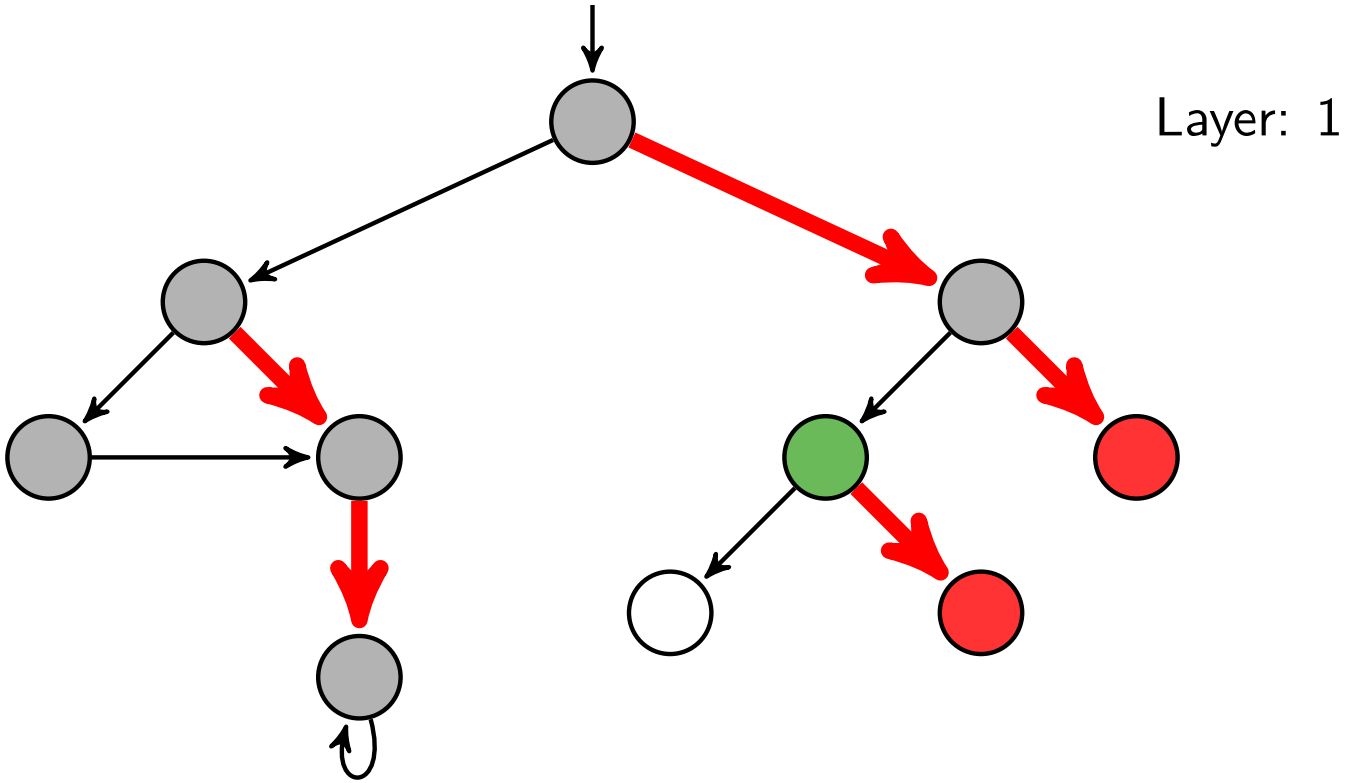
Exploration Order

Context-Switch-Directed Reachability: simplified example

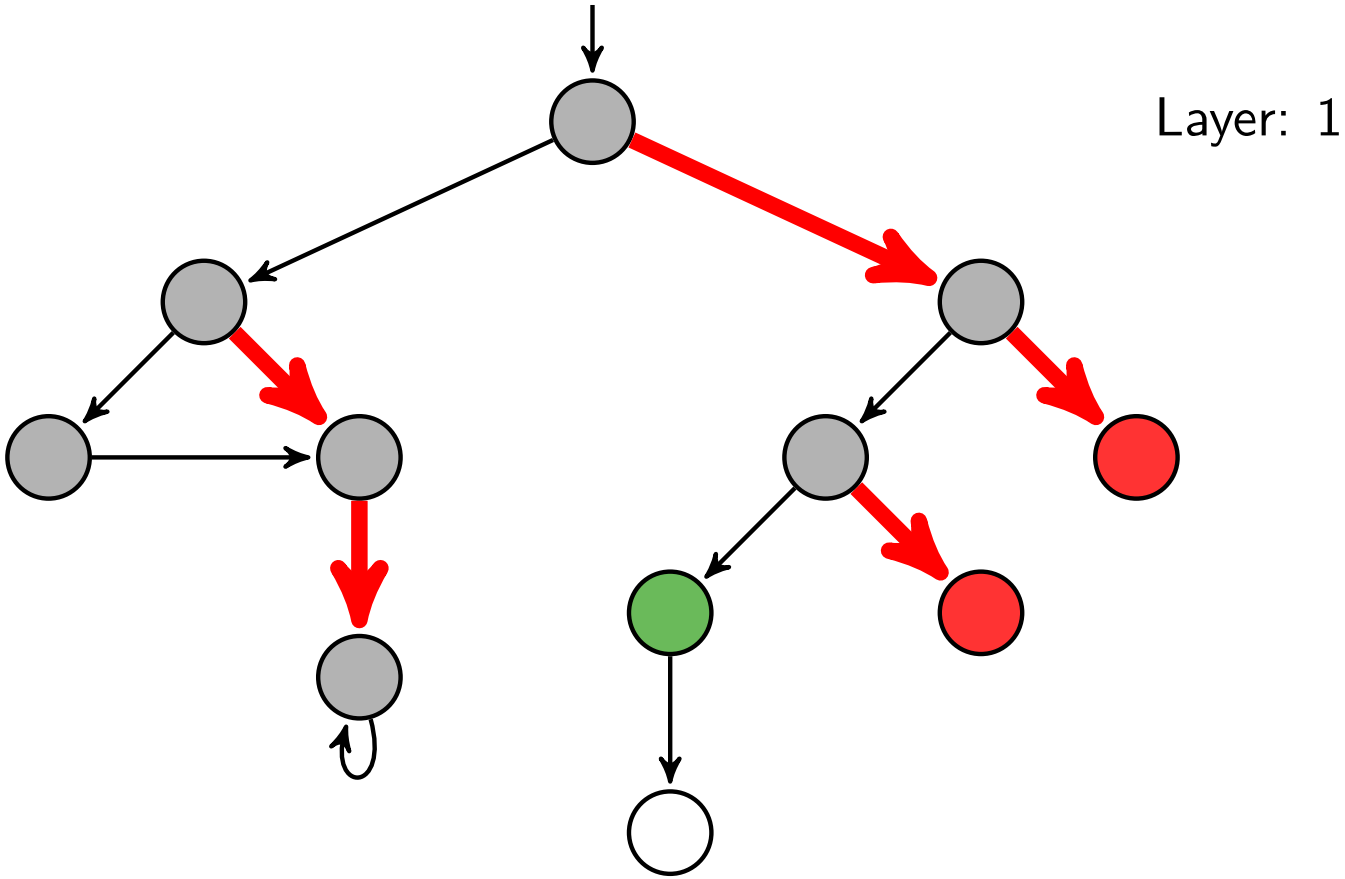


Exploration Order

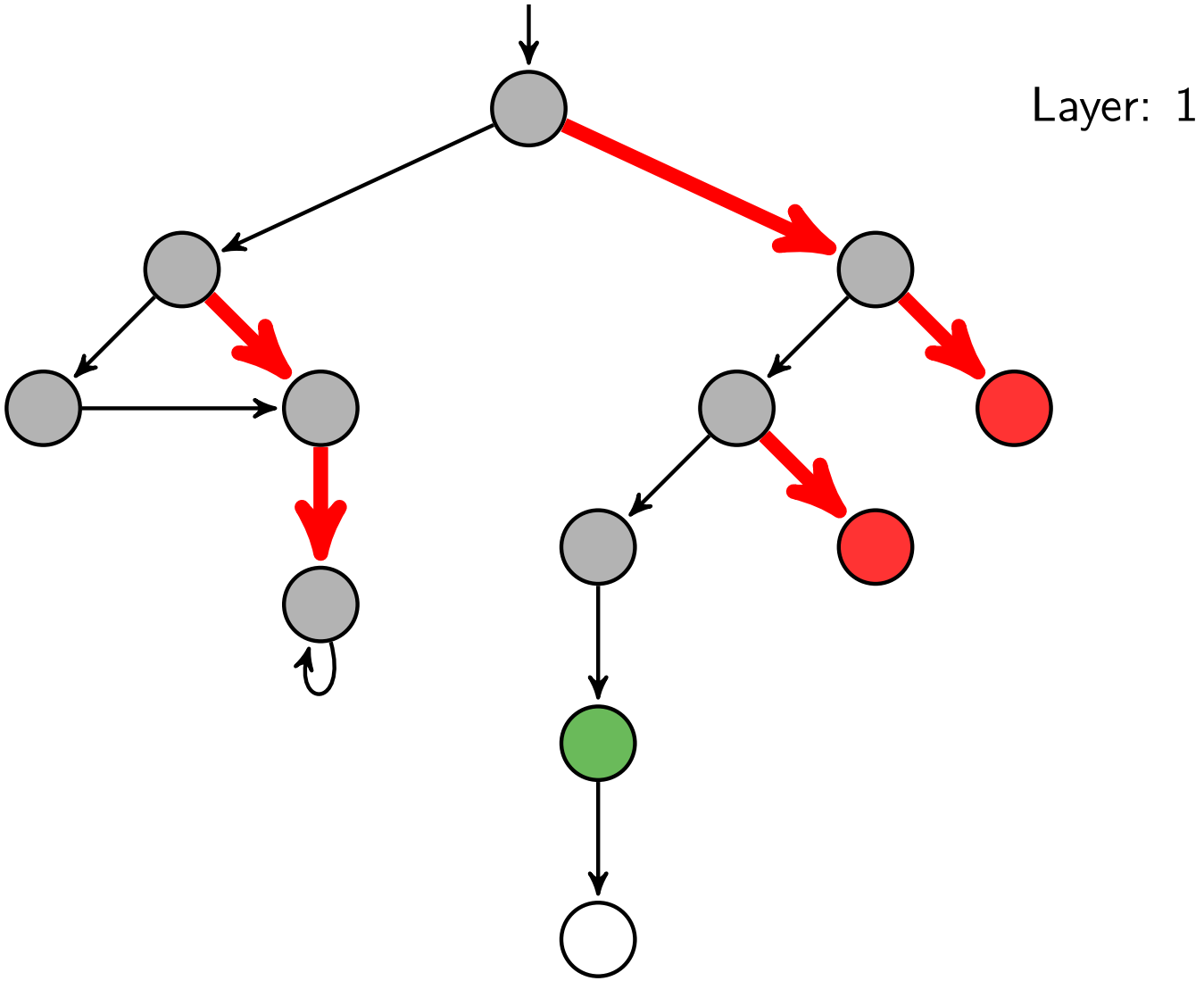
Context-Switch-Directed Reachability: simplified example



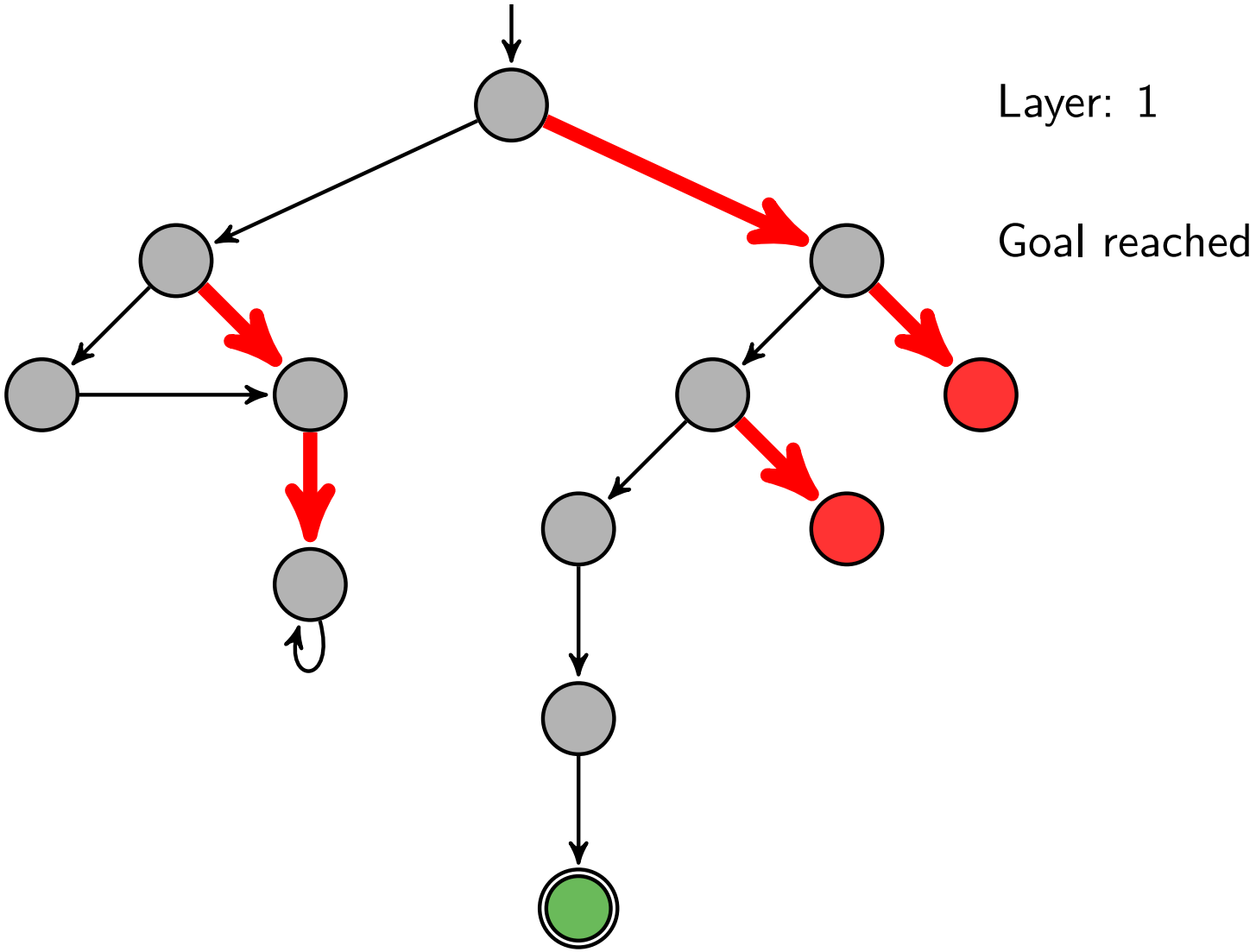
Context-Switch-Directed Reachability: simplified example



Context-Switch-Directed Reachability: simplified example



Context-Switch-Directed Reachability: simplified example



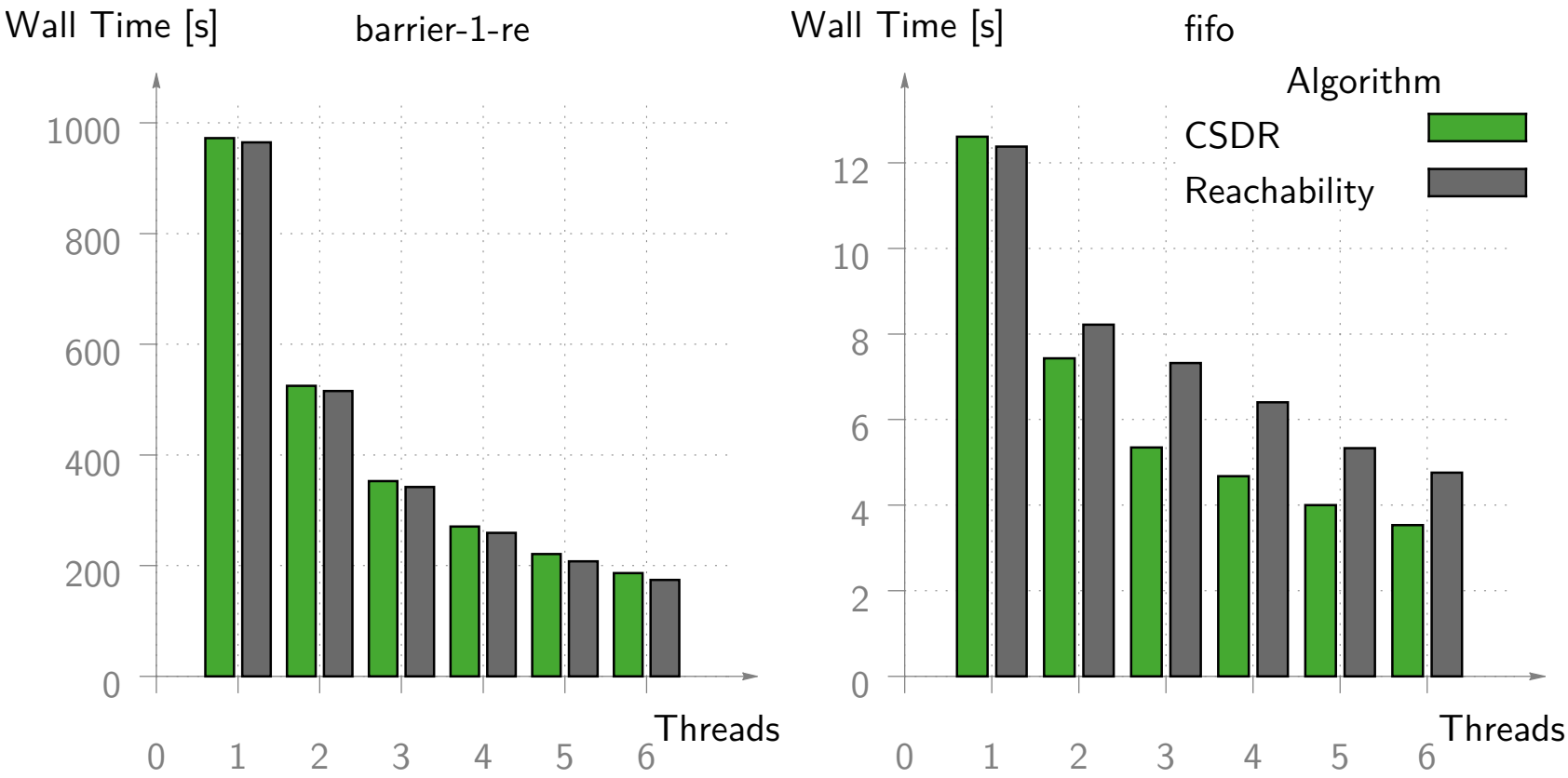
Results

- 11 C++ test cases, both real-world bugs and crafted
  - 8 with bugs
- compare new Context-Switch-Directed Reachability (= CSDR) with breath-first-search-based parallel reachability (= reach.)

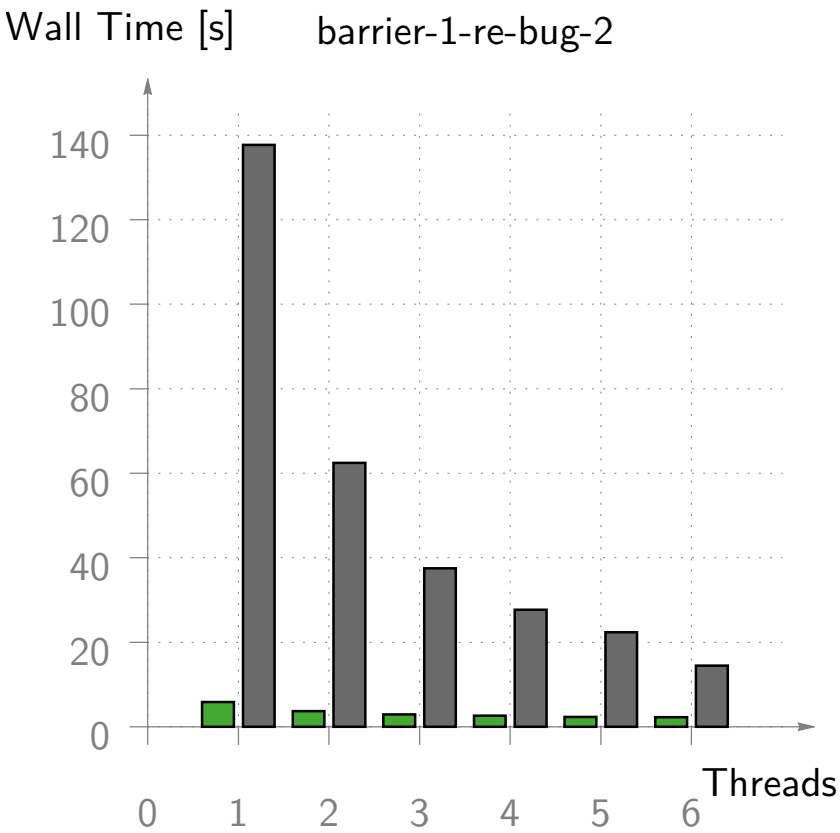
Counterexample length and number of context switches (CS = context switch, CE = counterexample):

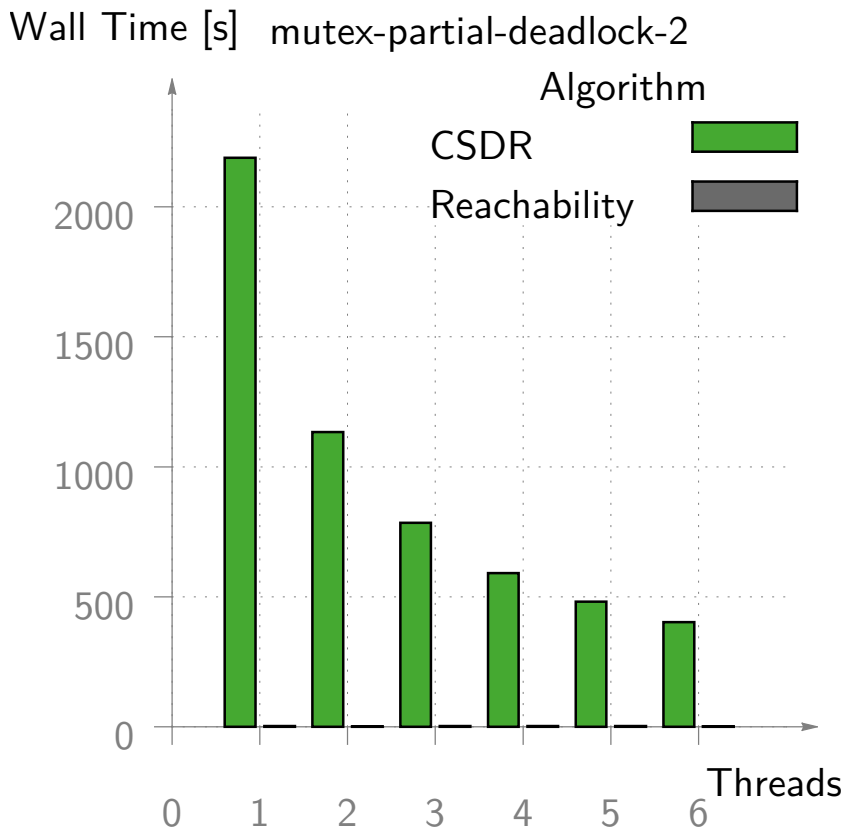
model	CE length		# of CSs in CEs	
	CSDR	reach.	CSDR	reach.
barrier-1-bug	22	22 – 27	1	1 – 4
barrier-1-re-bug-2	68	**65** – 98	**4**	8 – 18
barrier-1-re-bug	90 – 91	90 – 98	**12**	16 – 28
barrier-1-re-bug-test	120	120 – 123	**8**	14 – 31
barrier-n-bug	38	38 – 53	1	1 – 10
fifo-bug	165 – 171	**164** – 172	**4**	10 – 16
mutex-part-deadlock	29	**23** – 31	**8**	9 – 15
mutex-part-deadlock-2	32 – 78	**22** – 37	**9**	10 – 17

Time and scalability: without counterexample



Time and scalability: with counterexample





Conclusion

- verification of parallel unit tests in C & C++
- more readable counterexamples
- better scalability and result stability
- viable heuristic, can speed up counterexample search
- little to no overhead in case of correct model

Conclusion

Conclusion

- verification of parallel unit tests in C & C++
- more readable counterexamples
- better scalability and result stability
- viable heuristic, can speed up counterexample search
- little to no overhead in case of correct model

Future work

- extension to Linear Temporal Logic model checking
- extension to other input formalisms

Conclusion

Conclusion

- verification of parallel unit tests in C & C++
- more readable counterexamples
- better scalability and result stability
- viable heuristic, can speed up counterexample search
- little to no overhead in case of correct model

Future work

- extension to Linear Temporal Logic model checking
- extension to other input formalisms

`http://divine.fi.muni.cz` Thank you.