Part 2: Reachability analysis of stack-based systems

From Finite to Infinite-State Systems

- So far, algorithms for systems with finite state spaces
  - semi-algorithms in the presence of recursion

Decidability of reachability analysis

Single thread of control:

Control	Finite		Infinite
Data	Acyclic	Looping	
Finite	Yes	Yes	Yes
Infinite	Yes	No	No

Decidability of reachability analysis

Multiple threads of control:

Control	Finite		Infinite
Data	Acyclic	Looping	
Finite	Yes	Yes	No
Infinite	Yes	No	No

Decidability vs. Expressiveness

- Unbounded state ≠ Undecidable
- Is the unbounded system able to encode a Turing machine?
  - Single-counter machines? NO
  - Two-counter machines? YES
  - Single-stack machines? NO
  - Two-stack machines? YES

State representation

- Explicit representation infeasible
- Symbolic representation is the key
  - For the transition system
  - For the reachable states

Pushdown systems

$(G, L, g_0, l_0, \rightarrow)$

$g, h \in G$: finite set of control states

$l, m \in L$: finite set of stack symbols

g_0 : initial control state

l_0 : initial stack symbol

$\rightarrow$: set of transitions

Remarks

The classical definition of a pushdown system has, in addition, an alphabet I of input symbols.

Each transition depends on the control state, the top of the stack, and the input symbol.

The language $L \subseteq I^*$ of a classical pushdown system contains those input sequences for which there is an execution leading to the empty stack.

We are only concerned with reachability analysis and will therefore ignore I.

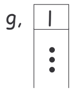
Three kinds of transitions:

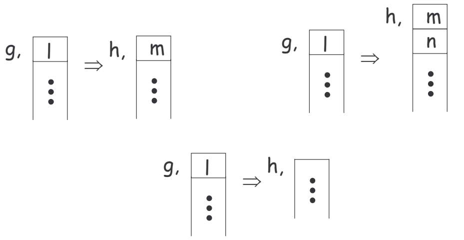
$(g, l) \rightarrow (h, m)$ (step)

$(g, l) \rightarrow (h, m\ n)$ (call)

$(g, l) \rightarrow (h, \varepsilon)$ (return)

Configuration:





Modeling sequential programs

- An element in G is a valuation to global variables
- An element in L is a valuation to local variables and
  - current instruction address for the frame at the top of the stack
  - return instruction address for the other frames

Example

bool a = F;	$(a, \langle x, pc \rangle)$
void main( ) {	$(F, \langle \_, L1 \rangle)$
L1: a = T;	$(T, \langle \_, L2 \rangle)$
L2: flip(a);	$(T, \langle \_, L3 \rangle \langle T, L4 \rangle)$
L3: }	$(F, \langle \_, L3 \rangle \langle T, L5 \rangle)$
void flip(bool x) {	$(F, \langle \_, L3 \rangle)$
L4: a = !x;	(F, ε)
L5: }	

Reachability problem

Given pushdown system $(G, L, g_0, l_0, \rightarrow)$ and control state g, does there exist a stack $ls \in L^*$ such that $(g_0, l_0) \Rightarrow^* (g, ls)$?

Naïve algorithm

$$\frac{}{\text{Add } (g_0, l_0) \text{ to R}}$$

$$\frac{(g, ls) \in R \qquad (g, ls) \Rightarrow (g', ls')}{\text{Add } (g', ls') \text{ to R}}$$

Problem with the naïve algorithm

- R is unbounded so algorithm won't terminate
- Two solutions:
  - Summary-based (a.k.a. interprocedural dataflow analysis)
  - Automata-based

Automata-based algorithm

$$\frac{}{\text{Add } (g_0, l_0) \text{ to R}} \qquad \frac{(g, ls) \in R \qquad (g, ls) \Rightarrow (g', ls')}{\text{Add } (g', ls') \text{ to R}}$$

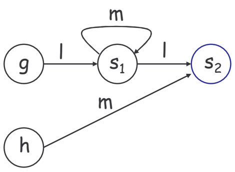
Key idea:
Use a finite automaton to symbolically represent R

Symbolic representation

Pushdown system $(G, L, g_0, l_0, \rightarrow)$

Representation automaton (Q, L, T, G, F)

- Q ($\supseteq$ G) is the set of states
- L is the alphabet
- T is the transition relation
- G is the set of initial states
- F is the set of final states



Represents the set of configurations: { (h, m), (g, l m* l) }

A set C of configurations is regular if it is representable by an automaton

Theorem (Buchi) : The set of configurations reachable from a regular set is also regular.

Remarks

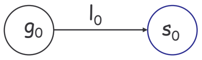
The classical definition of a pushdown system has, in addition, an alphabet I of input symbols.

Buchi's theorem does not contradict the fact that pushdown systems can accept non-regular languages over the input alphabet I.

The language of reachable stack configurations is a language over the alphabet L.
The accepted language is a language over the alphabet I.

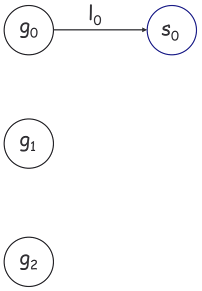
Pushdown system:

$(G, L, g_0, l_0, \rightarrow)$
- $G = \{g_0, g_1, g_2\}$
- $L = \{l_0, l_1, l_2\}$
- $(g_0, l_0) \rightarrow (g_1, l_1 l_0)$
  $(g_1, l_1) \rightarrow (g_2, l_2 l_0)$
  $(g_2, l_2) \rightarrow (g_0, l_1)$
  $(g_0, l_1) \rightarrow (g_0, \varepsilon)$



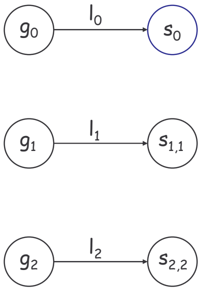
Pushdown system:

$(G, L, g_0, l_0, \rightarrow)$
- $G = \{g_0, g_1, g_2\}$
- $L = \{l_0, l_1, l_2\}$
- $(g_0, l_0) \rightarrow (g_1, l_1 l_0)$
  $(g_1, l_1) \rightarrow (g_2, l_2 l_0)$
  $(g_2, l_2) \rightarrow (g_0, l_1)$
  $(g_0, l_1) \rightarrow (g_0, \varepsilon)$



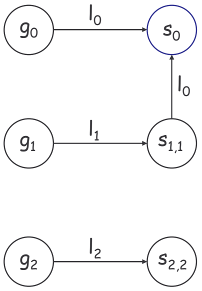
Pushdown system:

$(G, L, g_0, l_0, \rightarrow)$
- $G = \{g_0, g_1, g_2\}$
- $L = \{l_0, l_1, l_2\}$
- $(g_0, l_0) \rightarrow (g_1, l_1 l_0)$
  $(g_1, l_1) \rightarrow (g_2, l_2 l_0)$
  $(g_2, l_2) \rightarrow (g_0, l_1)$
  $(g_0, l_1) \rightarrow (g_0, \varepsilon)$



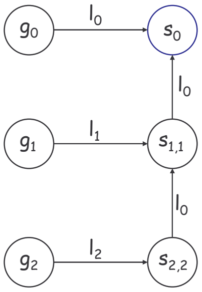
Pushdown system:

$(G, L, g_0, l_0, \rightarrow)$
- $G = \{g_0, g_1, g_2\}$
- $L = \{l_0, l_1, l_2\}$
- $(g_0, l_0) \rightarrow (g_1, l_1 l_0)$
  $(g_1, l_1) \rightarrow (g_2, l_2 l_0)$
  $(g_2, l_2) \rightarrow (g_0, l_1)$
  $(g_0, l_1) \rightarrow (g_0, \varepsilon)$



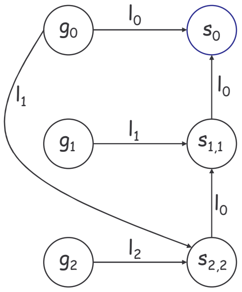
Pushdown system:

$(G, L, g_0, l_0, \rightarrow)$
- $G = \{g_0, g_1, g_2\}$
- $L = \{l_0, l_1, l_2\}$
- $(g_0, l_0) \rightarrow (g_1, l_1 l_0)$
  $(g_1, l_1) \rightarrow (g_2, l_2 l_0)$
  $(g_2, l_2) \rightarrow (g_0, l_1)$
  $(g_0, l_1) \rightarrow (g_0, \varepsilon)$



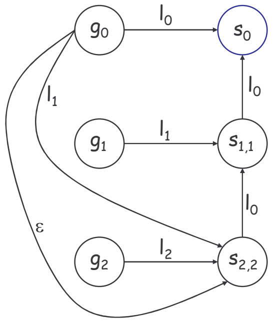
Pushdown system:

$(G, L, g_0, l_0, \rightarrow)$
- $G = \{g_0, g_1, g_2\}$
- $L = \{l_0, l_1, l_2\}$
- $(g_0, l_0) \rightarrow (g_1, l_1 l_0)$
  $(g_1, l_1) \rightarrow (g_2, l_2 l_0)$
  $(g_2, l_2) \rightarrow (g_0, l_1)$
  $(g_0, l_1) \rightarrow (g_0, \varepsilon)$



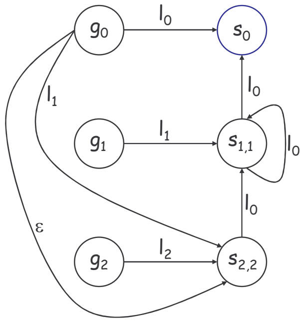
Pushdown system:

$(G, L, g_0, l_0, \rightarrow)$
- $G = \{g_0, g_1, g_2\}$
- $L = \{l_0, l_1, l_2\}$
- $(g_0, l_0) \rightarrow (g_1, l_1 l_0)$
  $(g_1, l_1) \rightarrow (g_2, l_2 l_0)$
  $(g_2, l_2) \rightarrow (g_0, l_1)$
  $(g_0, l_1) \rightarrow (g_0, \varepsilon)$



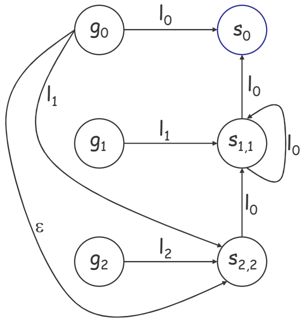
Pushdown system:

$(G, L, g_0, l_0, \rightarrow)$
- $G = \{g_0, g_1, g_2\}$
- $L = \{l_0, l_1, l_2\}$
- $(g_0, l_0) \rightarrow (g_1, l_1 l_0)$
  $(g_1, l_1) \rightarrow (g_2, l_2 l_0)$
  $(g_2, l_2) \rightarrow (g_0, l_1)$
  $(g_0, l_1) \rightarrow (g_0, \varepsilon)$



Pushdown system:

$(G, L, g_0, l_0, \rightarrow)$
- $G = \{g_0, g_1, g_2\}$
- $L = \{l_0, l_1, l_2\}$
- $(g_0, l_0) \rightarrow (g_1, l_1 l_0)$
  $(g_1, l_1) \rightarrow (g_2, l_2 l_0)$
  $(g_2, l_2) \rightarrow (g_0, l_1)$
  $(g_0, l_1) \rightarrow (g_0, \varepsilon)$

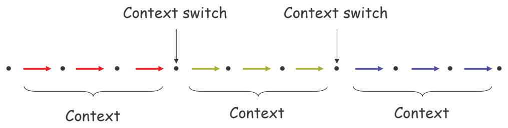


$\{ (g_0, l_0 \cup l_0 l_0^+ \cup l_1 l_0 l_0^+), (g_1, l_1 l_0^+), (g_2, l_2 l_0 l_0^+) \}$

Reachability analysis for concurrent pushdown systems

- Undecidable in general
- Three approaches
  - restrict computation model, e.g., Esparza-Podelski 00
  - sound and imprecise approaches, e.g., Bouajjani-Esparza-Touili 03, Flanagan-Qadeer 03
  - unsound but precise approaches

Context-bounded verification of concurrent software



Context switch

Context switch

Context

Context

Context

Analyze *all* executions with *small* number of context switches !

Different from bounded-depth model checking
- no bound on the computation within each context

Why context-bounded analysis?

- Many subtle concurrency errors are manifested in executions with a small number of context switches
- Context-bounded analysis can be performed efficiently

KISS: a static analysis tool

- Technique to use any sequential checker to perform context-bounded concurrency analysis
- Found a number of concurrency errors in NT device drivers even with a context-switch bound of two

Driver	KLOC	# Fields	# Races
Tracedrv	0.5	3	0
Moufiltr	1.0	14	0
Kbfiltr	1.1	15	0
Imca	1.1	5	1
Startio	1.1	9	0
Toaster/toastmon	1.4	8	1
Diskperf	2.4	16	0
1394diag	2.7	18	0
1394vdev	2.8	18	1
Fakemodem	2.9	39	6
Toaster/bus	5.0	30	0
Serenum	5.9	41	2
Toaster/func	6.6	24	5
Mouclass	7.0	34	1
Kbdclass	7.4	36	1
Mouser	7.6	34	1
Fdc	9.2	92	9

Total:
30 races

Zing: an explicit-state model checker

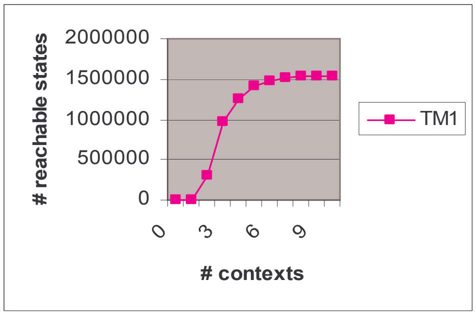
- *Case study (Naik-Rehof 04): Concurrent transaction management code from Microsoft product group*
- *Analyzed by the Zing model checker after automatically translating to the Zing input language*
  - Found three bugs each requiring between three and four context switches

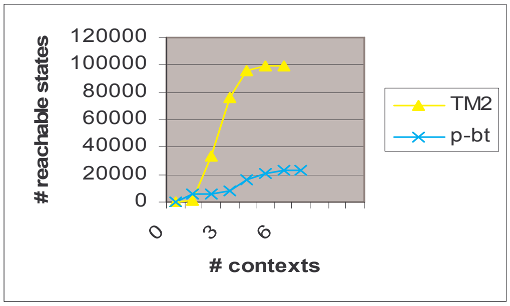
Why context-bounded analysis?

- Many subtle concurrency errors are manifested in executions with a small number of context switches
- Context-bounded analysis can be performed efficiently

Polynomially-bounded executions

- Context bounding leads to polynomial bound on the number of executions
  - n threads, each executing k steps
  - total no. of executions = $\Omega(n^k)$
  - With context bound c, no. of executions = $O((n^2.k)^c)$

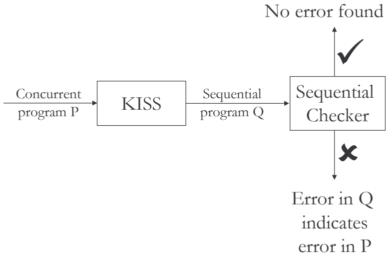




Reachability analysis

- Rechability analysis of finite-data concurrent programs is decidable for bounded number of context switches

KISS: A static checker for concurrent software



No error found

✓

Concurrent program P

KISS

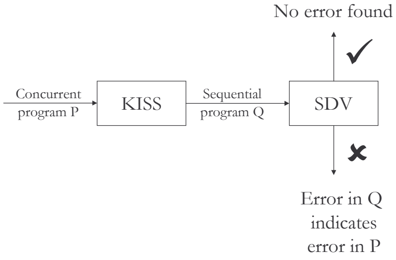
Sequential program Q

Sequential Checker

✗

Error in Q indicates error in P

KISS: A static checker for concurrent software



No error found

✓

Concurrent program P

KISS

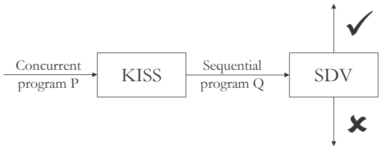
Sequential program Q

SDV

✗

Error in Q indicates error in P

KISS strategy



✓

Concurrent program P

KISS

Sequential program Q

SDV

✗

- Q encodes executions of P with small number of context switches
  - instrumentation introduces lots of extra paths to mimic context switches
- Leverage all-path analysis of sequential checkers

```
DispatchRoutine( ) {
   int t;

   if (! de->stopping) {

      AtomicIncr(& de->count);
      // do useful work
      // …

      t = AtomicDecr(& de->count);

      if (t == 0)
         SetEvent(& de->stopEvent);

   }
}
```

```
PnpStop( ) {
   int t;

   de->stopping = T;

   t = AtomicDecr(& de->count);

   if (t == 0)
      SetEvent(& de->stopEvent);

   WaitEvent(& de->stopEvent);
}
```

```
DispatchRoutine( ) {
   int t;

   if (! de->stopping) {

      AtomicIncr(& de->count);
      // do useful work
      // …

      t = AtomicDecr(& de->count);

      if (t == 0)
         SetEvent(& de->stopEvent);

   }
}
```

```
PnpStop( ) {
   int t;
   if ($) return;
   de->stopping = T;
   if ($) return;
   t = AtomicDecr(& de->count);
   if ($) return;
   if (t == 0)
      SetEvent(& de->stopEvent);
   if ($) return;
   WaitEvent(& de->stopEvent);

}
```

```
bool done = F;
                    CODE  ≡   if ( !done ) {
                                 if ($) { done = T; PnpStop( ); }
                              }
```

```
DispatchRoutine( ) {
   int t;
   CODE;
   if (! de->stopping) {
      CODE;
      AtomicIncr(& de->count);
      // do useful work
      // …
      CODE;
      t = AtomicDecr(& de->count);
      CODE;
      if (t == 0)
         SetEvent(& de->stopEvent);
      CODE;
   }
}
```

```
PnpStop( ) {
   int t;
   if ($) return;
   de->stopping = T;
   if ($) return;
   t = AtomicDecr(& de->count);
   if ($) return;
   if (t == 0)
      SetEvent(& de->stopEvent);
   if ($) return;
   WaitEvent(& de->stopEvent);

}
```

```
bool done = F;
                    CODE  ≡   if ( !done ) {
                                 if ($) { done = T; PnpStop( ); }
                              }
```

```
DispatchRoutine( ) {
   int t;
   CODE;
   if (! de->stopping) {
      CODE;
      AtomicIncr(& de->count);
      // do useful work
      // …
      CODE;
      t = AtomicDecr(& de->count);
      CODE;
      if (t == 0)
         SetEvent(& de->stopEvent);
      CODE;
   }
}
```

```
PnpStop( ) {
   int t;
   if ($) return;
   de->stopping = T;
   if ($) return;
   t = AtomicDecr(& de->count);
   if ($) return;
   if (t == 0)
      SetEvent(& de->stopEvent);
   if ($) return;
   WaitEvent(& de->stopEvent);

}
```

```
main( ) {  DispatchRoutine( ); }
```

```
bool done = F;
                    CODE  ≡   if ( !done ) {
                                 if ($) { done = T; PnpStop( ); }
                              }
```

```
DispatchRoutine( ) {
   int t;
   if ($) return;
   if (! de->stopping) {
      if ($) return;
      AtomicIncr(& de->count);
      // do useful work
      // …
      if ($) return;
      t = AtomicDecr(& de->count);
      if ($) return;
      if (t == 0)
         SetEvent(& de->stopEvent);

   }
}
```

```
PnpStop( ) {
   int t;
   CODE;
   de->stopping = T;
   CODE;
   t = AtomicDecr(& de->count);
   CODE;
   if (t == 0)
      SetEvent(& de->stopEvent);
   CODE;
   WaitEvent(& de->stopEvent);
   CODE;
}
```

```
main( ) {  PnpStop( ); }
```

KISS features (I)

- KISS trades off soundness for scalability
- Sound for event-driven systems
  - embedded software, TinyOS
- Unsoundness is precisely quantifiable for other systems
  - e.g., for 2-thread program, explores *all* executions with up to two context switches

---

KISS features (II)

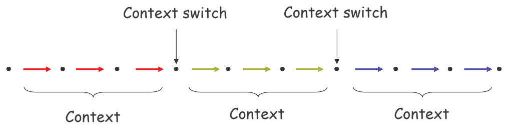
- Cost of analyzing a concurrent program P = cost of analyzing a sequential program Q
  - Size of Q asymptotically same as size of P
- Allows any sequential checker to analyze concurrency

However…

- Hard limit on number of explored contexts
  - e.g., two context switches for concurrent program with two threads

Is a tuning knob possible?

Given a concurrent boolean program P and a positive integer c, does P go wrong by failing an assertion via an execution with at most c contexts?



Problem:
- Unbounded computation possible within each context!
- Unbounded execution depth and reachable state space
- Different from bounded-depth model checking

Sequential boolean program

Global store	g,	valuation to global variables
Local store	l,	valuation to local variables
Stack	s,	sequence of local stores
State	(g, s)	

Sequential boolean program

Global store	g,	valuation to global variables
Local store	l,	valuation to local variables
Stack	s,	sequence of local stores
State	(g, s)	

Transition relation:

$$(g, s) \rightarrow (g', s')$$

Reachability problem for sequential boolean program

$\mathrm{Reach}(g, s) = \{ (g',s') \mid (g, s) \rightarrow^* (g', s') \}$

Given (g, s), is there s' such that $(g, s) \rightarrow^* (\mathrm{error}, s')$?

Aggregate state

Set of stacks	ss
Aggregate state	$(g, ss) = \{ (g,s) \mid s \in ss \}$

$\text{Reach}(g, ss) = \bigcup \{\text{Reach}(g,s) \mid s \in ss\}$

Aggregate transition relation

- Suppose $G = \{ g'_1, \ldots, g'_n \}$
- There is a unique partition of $\text{Reach}(g, ss)$ into aggregate states: $(g'_1, ss'_1) \cup \ldots \cup (g'_n, ss'_n)$

$$(g, ss) \Rightarrow (g'_1, ss'_1)$$
$$\vdots$$
$$(g, ss) \Rightarrow (g'_n, ss'_n)$$

iff $\text{Reach}(g,ss) = (g'_1, ss'_1) \cup \ldots \cup (g'_n, ss'_n)$

Theorem (Buchi, Schwoon00)

- If ss is regular and $(g, ss) \Rightarrow (g', ss')$, then ss' is regular.
- If ss is given as a finite automaton A, then a finite automaton A' for ss' can be constructed from A in polynomial time.

Algorithm

Problem:
Given (g, s), is there s' such that $(g, s) \to^* (\text{error}, s')$?

Solution:
Compute automaton for ss' such that $(g, \{s\}) \Rightarrow (\text{error}, ss')$ and check if ss' is nonempty.

Concurrent boolean program

Global store	g,	valuation to global variables
Local store	l,	valuation to local variables
Stack	s,	sequence of local stores
State	(g, s_1, s_2)	

Transition relation:

$$\frac{(g, s_1) \to (g', s'_1) \text{ in thread 1}}{(g, s_1, s_2) \to_1 (g', s'_1, s_2)} \qquad \frac{(g, s_2) \to (g', s'_2) \text{ in thread 2}}{(g, s_1, s_2) \to_2 (g', s_1, s'_2)}$$

Reachability problem for concurrent boolean program

Given (g, s_1, s_2), are there s'_1 and s'_2 such that (g, s_1, s_2) reaches $(\text{error}, s'_1, s'_2)$ via an execution with at most c contexts?

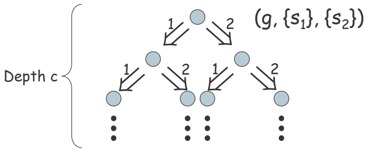
Aggregate transition relation

$$(g, ss_1, ss_2) = \{ (g, s_1, s_2) \mid s_1 \in ss_1, s_2 \in ss_2 \}$$

$$\frac{(g, ss_1) \Rightarrow (g', ss'_1) \text{ in thread 1}}{(g, ss_1, ss_2) \Rightarrow_1 (g', ss'_1, ss_2)}$$

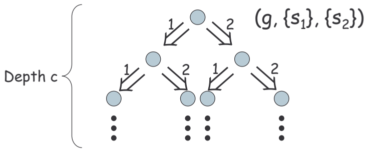
$$\frac{(g, ss_2) \Rightarrow (g', ss'_2) \text{ in thread 2}}{(g, ss_1, ss_2) \Rightarrow_2 (g', ss_1, ss'_2)}$$

Algorithm: 2 threads, c contexts



Compute the set of reachable aggregate states.
Report an error if (g, ss_1, ss_2) is reachable and g = error, ss_1 is nonempty, and ss_2 is nonempty.

Complexity: 2 threads, c contexts



Depth of tree = context bound c
Branching factor bounded by $G \times 2$ (G = # of global stores)
Number of edges bounded by $(G \times 2)^{(c+1)}$
Each edge computable in polynomial time

Results

- Algorithm for checking if a concurrent boolean program P fails an assertion via an execution with at most c contexts
- Algorithm for checking if a concurrent boolean program P with unbounded fork-join parallelism fails an assertion via an execution with at most c contexts