Depth-Bounded Systems: reachability and termination

Thomas A. Henzinger
Damien Zufferey
IST AUSTRIA
Institute of Science and Technology

Kshitij Bansal
Eric Koskinen
Thomas Wies
NEW YORK UNIVERSITY

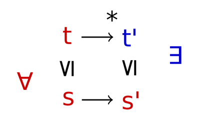
Depth-Bounded Systems form an expressive class of well-structured graph transition systems. They can model a wide range of concurrent infinite-state systems including those with dynamic thread creation, dynamically changing communication topology,and complex shared heap structures. We introduce a symbolic representation, called nested graphs, to capture the families of potentially unbounded graphs generated by DBS. Nested graphs can be used as part of a safety analysis to finitely represent an over-approximation of a system's reachable states. Furthermore, we present a method to automatically prove termination of DBS. Our method is built on top of the safety analysis and uses a numerical abstraction, called structural counter abstraction, which is obtained by systematically augmenting the over-approximation of a BDS's reachable states with a finite set of counters. This numerical abstraction can be analyzed with existing termination provers. What makes our approach unique is the way in which it exploits the well-structuredness of the analyzed system. We have implemented our work in a prototype tool and used it to automatically prove safety and liveness properties of complex concurrent systems, including non blocking algorithms such as Treiber's stack and several distributed processes.

Depth-Bounded Systems (DBS) are well-structured graph transition systems.
-> WSTS: "nice" class of infinite state systems that can be analyzed algorithmically.
-> graph rewriting system: flexible formalism that can capture many systems.
DBS were introduced by [Meyer, 2008] as a fragment of the π-calculus

A WSTS is a transition system (S, →, ≤) such that:

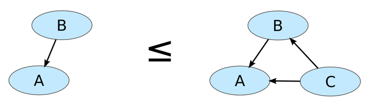
≤ is a well-quasi-ordering: well-founded + no infinite anti-chain

compatibility of ≤ w.r.t. → :



[Abdulla et al., 1996, Finkel and Schnoebelen, 2001]

DBS uses subgraph isomorphism as ordering:



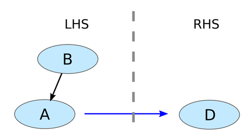
This is a wqo only on families of graphs where the longest acyclic path is bounded.

Hence, we speak of Depth-Bounded Systems.

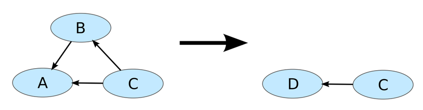
Rewrite rule:
A rewrite rule is composed of:
- a "left hand side" graph (LHS)
- a "right hand side" graph (RHS)
- a partial morphism: LHS → RHS

Applying a rule to the graph G matches the LHS to a subgraph of G and replace it by the RHS. The partial morphism is used to update the edges of nodes in the frame.

Example rule:

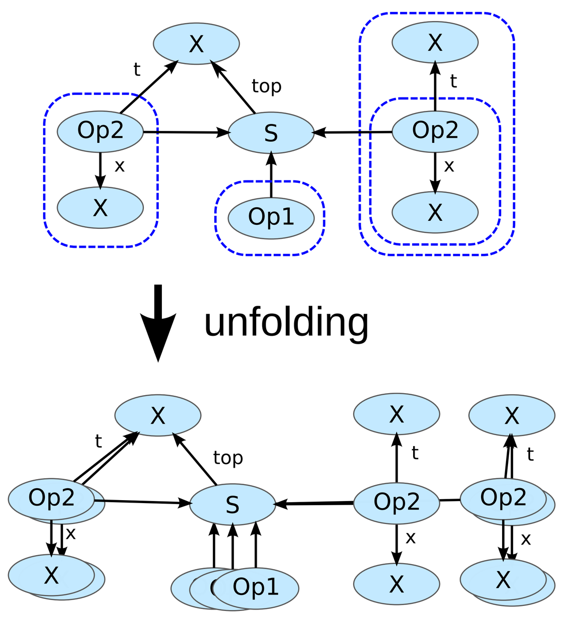


And its application:



Nested graphs. The analysis needs to manipulate potentially unbounded graphs. We introduce nested graphs for this purpose. A nested graph is a graph where some subgraphs marked with a blue dashed box can be copied as much as needed.

A nested graph represents a (infinite) set of graphs. Graphs covered by a nested graph can be obtained by unfolding.



Example. Consider Treiber's stack, a non-blocking algorithm. The algorithm implements a stack with a simple linked-list. The two operations, push and pop use the *compare-and-swap* (CAS) instruction to atomically modify a location in memory. CAS(l,v,v') atomically examines the value at location l and, if it is equivalent to v, sets l to value v'. We are able to prove lock-freedom of this algorithm via a reduction to fair termination of a depth-bounded system. We can represent Treiber's stack algorithm as a depth-bounded system, by abstracting over the values and order of the elements in the stack.

Lock-freedom is a property that allows individual threads to starve but guarantees global progress. Termination proofs for lock-free algorithms tends to be hard to automatize since it requires reasoning globally and considering all the interleavings of threads.

Typical lock-free structure

If the CAS fails (thread might starve)

then another thread succeeded (global progress)

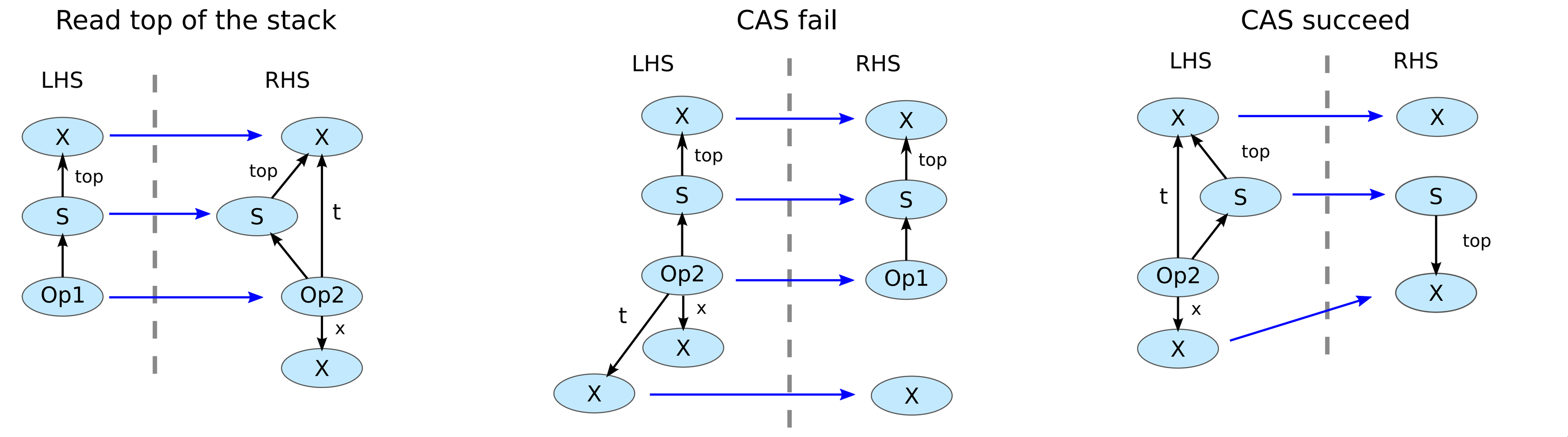
A proof of lock-freedom or fair termination needs to capture that fact.

Treiber's stack
[Treiber, 1986]

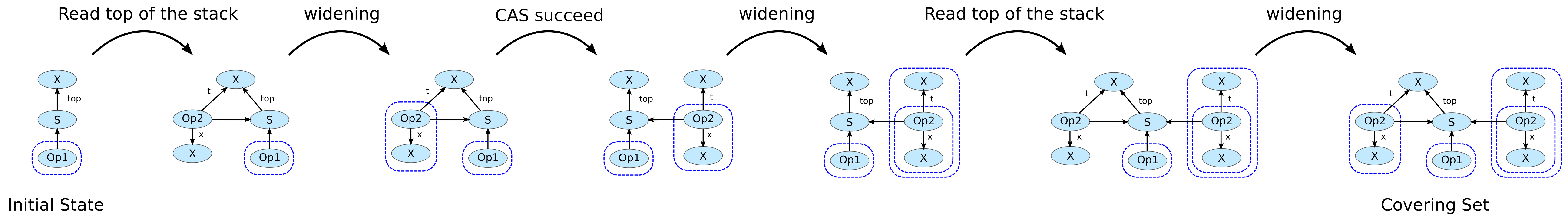
```
struct node {
    struct node *next;
    value data;
};
struct stack { struct node *Top; };
struct stack *S;

void init() {
    S = alloc();
    S->Top = NULL;
}

value pop() {
  struct node *t, *x;
  do {
    t = S->Top;
    if (t == NULL) return EMPTY;
    x = t->next;
  } while (!CAS(&S->Top,t,x));
  return t->data;
}

void push(value v) {
  struct node *t, *x;
  x = alloc();
  x->data = v;
  do {
    t = S->Top;
    x->next = t;
  } while (!CAS(&S->Top,t,x));
}
```

Treiber's stack
simplified

```
     void op() {
         do {
Op1:         t = S->Top;
             x = ... ;
Op2:     } while (!CAS(&S->Top,t,x));
     }
```

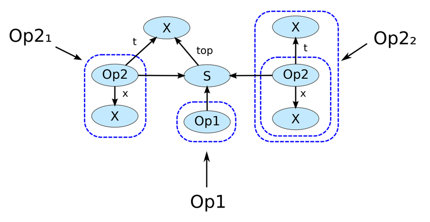


Covering Set. For the safety analysis we compute an over-approximation of the covering set by using an "Ideal abstraction". The covering set is the downward closure of the set of reachable states. In other terms it is an inductive invariant which usually has a compact representation due to the monotonic nature of the system. Our analysis mimics the usual acceleration based algorithm for WSTS but uses widening instead of acceleration. Below we show a possible sequence that the algorithm can explore to build the covering set.

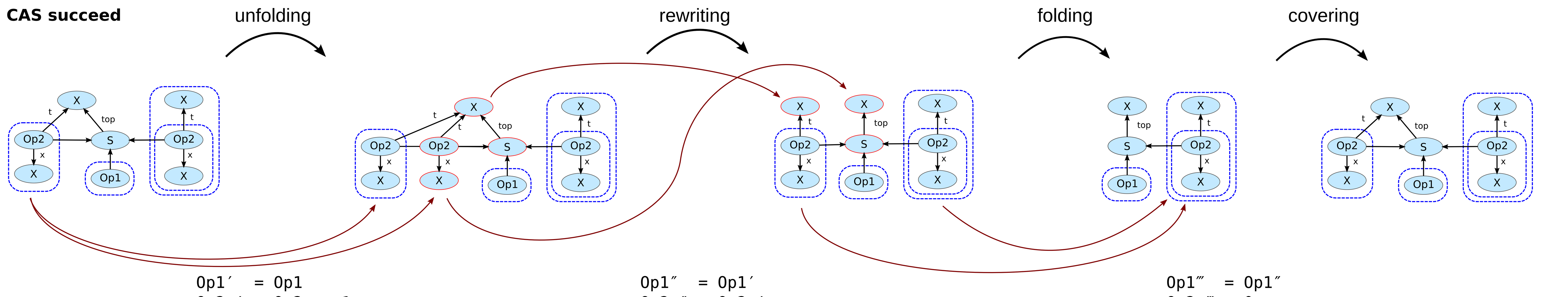


Structural counter abstraction. Using the covering set we can create a numerical abstraction of the system. This numerical abstraction is a counter program. We use off-the-shelf termination prover to show termination of the system. Below we show how the numerical transition corresponding to "CAS succeed" applied to the covering set is generated.

Counters used in the examples:



CAS succeed



Op1′ = Op1
Op2₁′ = Op2₁ - 1
Op2₂′ = Op2₂
Op2₃′ = 1

Op1″ = Op1′
Op2₁″ = Op2₁′
Op2₂″ = Op2₂′
Op2₃″ = 0

Op1‴ = Op1″
Op2₁‴ = 0
Op2₂‴ = Op2₁″ + Op2₂″

Simplification

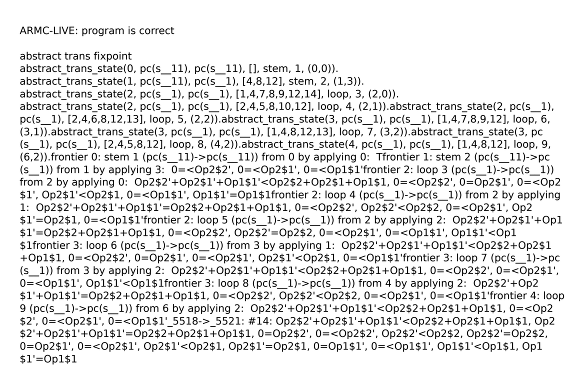
Ingredients of the structural counter abstraction

Nested graphs in the covering set	→	Control locations
Nodes in the nested graph	→	Counters
Transitions (rule + nested graph)	→	Transitions

Structural counter abstraction:

```
Read top of the stack  | CAS succeed             | CAS fails
Op1′  = Op1 - 1        | Op1′  = Op1             | Op1′  = Op1 + 1
Op2₁′ = Op2₁ + 1       | Op2₁′ = 0               | Op2₁′ = Op2₁
Op2₂′ = Op2₂           | Op2₂′ = Op2₁ + Op2₂ - 1 | Op2₂′ = Op2₂ - 1
```

Termination proof by ARMC



This poster is based on:

T. Wies, D. Zufferey, and T. A. Henzinger. Forward analysis of depth-bounded processes. In FOSSACS, 2010.
D. Zufferey, T. Wies, and T. A. Henzinger. Ideal abstractions for well-structured transition systems. In VMCAI, 2012.
K. Bansal, E. Koskinen, T. Wies, D. Zufferey: Structural Counter Abstraction. In TACAS, 2013

Tool and experiments: http://pub.ist.ac.at/~zufferey/picasso/