Structural Counter Abstraction

Kshitij Bansal[1], Eric Koskinen[1,⋆], Thomas Wies[1], and Damien Zufferey[2,⋆⋆]

[1] New York University
[2] IST Austria

Abstract. Depth-Bounded Systems form an expressive class of well-structured 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 present the first method to automatically prove fair termination of depth-bounded systems. Our method uses a numerical abstraction of the system, which we obtain by systematically augmenting an over-approximation of the system'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 liveness properties of complex concurrent systems, including nonblocking algorithms such as Treiber's stack and several distributed processes. Many of these examples are beyond the scope of termination analyses that are based on traditional counter abstractions.

1 Introduction

Graph transformation systems [9] are a well-studied formalism for describing concurrent computations. A *depth-bounded system* [17, 26] is a graph transformation system for which there exists a bound on the length of all simple (i.e. acyclic) paths in all reachable graphs. Depth-bounded systems are also well-structured transition systems (WSTS) [10]. This makes them an attractive target for automated analysis because there are generic algorithms for deciding a number of verification problems for WSTS [1].

Depth-bounded systems are also among the most expressive classes of WSTS, subsuming *e.g.* Petri nets and their monotonic extensions [18]. They can model a wide range of concurrent systems including those with dynamic thread creation, dynamically changing communication topology, and complex shared heap data structures. Many concurrent systems are depth-bounded. For instance, Actor-style message passing systems often fall into this class. Other systems have natural depth-bounded abstractions that preserve important properties. For example, consider the lock-free stack due to Treiber [25] (see Figure 1), which uses atomic *compare-and-swap* instructions to implement nonblocking stack operations. This algorithm can be abstracted to a depth-bounded system by ignoring the order of the elements in the stack. This abstraction

⋆ Supported in part by the CMACS NSF Expeditions in Computing award 0926166.
⋆⋆ Supported in part by the European Research Council (ERC) Advanced Investigator Grant QUAREM and by the Austrian Science Fund (FWF) project S11402-N23.

N. Piterman and S. Smolka (Eds.): TACAS 2013, LNCS 7795, pp. 62–77, 2013.

preserves the termination/progress behavior of the algorithm. Similar depth-bounded abstractions can be obtained for a wide variety of concurrent algorithms.

In this paper, we present the first method to automatically prove fair termination of depth-bounded systems. We focus on a notion of *weak fairness* that is consistent with the *finite delay property* for Petri nets [5]. However, our technique also extends to other fairness conditions. Many liveness properties of practical interest (including progress guarantees: wait-, lock-, and obstruction-freedom) are reducible to termination under weak fairness. The problem is difficult; it subsumes the structural termination problem for transfer nets (i.e. termination for all possible input markings), which is undecidable [16]. Despite this difficulty, we show that one can build on existing verification techniques for WSTS to obtain an approximate analysis for this problem that is both practical and sufficiently precise to prove fair termination of complex systems.

The key technical contribution of this paper is a method that automatically constructs a precise numerical abstraction of a depth-bounded system from a precomputed inductive invariant of the system. The inductive invariant is assumed to be given as a finite set of nested graphs in which nested subgraphs can be unfolded arbitrarily often. Thus, each nested graph is a symbolic representation of the (infinite) set of concrete graphs obtained by such unfoldings. We associate a counter with each of the nested subgraphs, tracking how often it can be unfolded. From these augmented nested graphs we then compute a numerical transition system that simulates the depth-bounded system. This so-called *structural counter abstraction* can then be analyzed using existing termination provers. The number and meaning of counters in the numerical abstraction is not fixed *a priori* but, instead, depends on the structure of the reachable configuration graphs (described by the inductive invariant). Our method thus provides a more precise alternative to traditional counter abstractions [3, 7, 21] for concurrent systems.

The benefit of our approach is that it can utilize existing reachability analyses for depth-bounded systems to obtain the inductive invariant [27], and existing termination analyses for numerical programs [6, 22]. We have implemented our method in a prototype tool and applied it to prove liveness properties of various concurrent systems, including nonblocking algorithms such as Treiber's stack, as well as distributed processes. These systems are beyond the scope of traditional counter abstraction techniques.

Contributions. We present the first automatic technique for proving fair termination of depth-bounded systems. Our technique enables the automated verification of liveness properties for a large class of concurrent infinite-state systems. What makes our approach unique is the way in which it exploits the monotonicity of the system. Our algorithmic technique of computing a numerical abstraction from an inductive invariant, introduced in this paper, promises applications beyond liveness properties. For instance, the same technique can be used to strengthen an inductive invariant of a depth-bounded system with numerical constraints, enabling proofs of complex safety properties.

2 Overview

Motivating example. Consider Treiber's stack [25], a non-blocking algorithm, given in the C-like code in Fig. 1. The algorithm implements a stack with a simple linked-list. The two operations, `push` and `pop` use the *compare-and-swap* (`CAS`) instruction

```
struct node {
   struct node *next;
   value t data;
};

struct stack {
  struct node *Top;
};
struct stack *S;

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

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

value t 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;
}
```

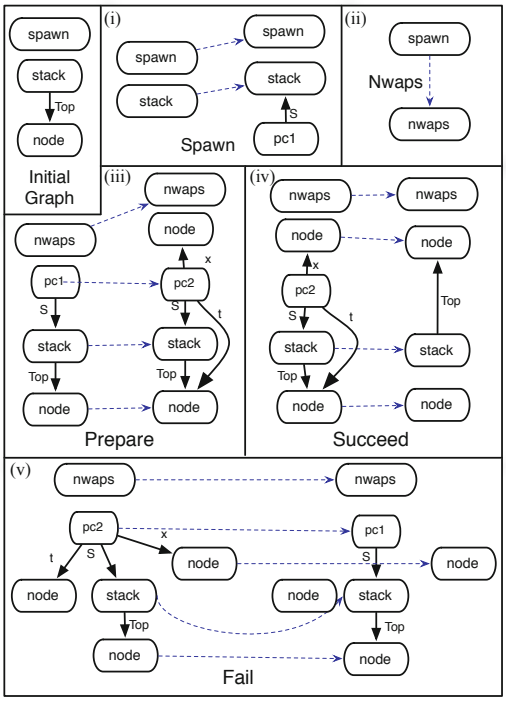


Fig. 1. Source code of Treiber's stack [25] and its abstraction as a graph transformation system

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'. In this section, we will describe how 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. In the depth-bounded abstraction of Treiber's stack, the graphs represent the state of the heap, *i.e.*, the linked list implementing the stack, and thread objects describing the local states of all clients currently executing push and pop operations. The abstraction is obtained from the concrete transition system of Treiber's stack by ignoring the values of `next` pointers connecting the vertices in the linked list of the stack. In this abstraction, there may still be unboundedly many elements in the stack as well as unboundedly many clients operating on the stack. However, since the list vertices are no longer connected, they can no longer form simple paths of arbitrary length in the heap graph. At this level of abstraction, `push` and `pop` become indistinguishable. Both operations have the same control-flow structure: they iteratively read the top of the stack and attempt to modify it until the `CAS` operation succeeds. The actual modification of the stack is non-deterministic in both operations.

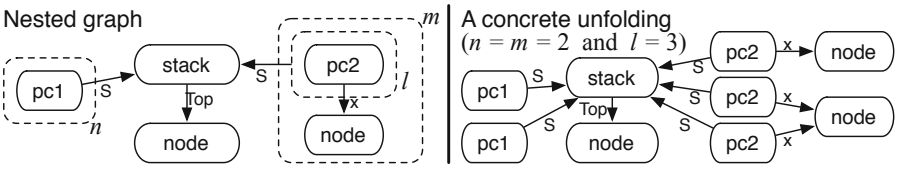
Depth-bounded abstractions of programs can be computed automatically from the program's source code using shape analysis techniques. These techniques are orthogonal

to the contribution of this paper. In Fig. 1 we give the graph rewriting system for the depth-bounded abstraction of Treiber's stack. The initial state is a graph consisting of the vertex spawn, indicating that clients can be spawned, and the stack and its Top element which is some node. There are five rewrite rules. (i) The Spawn rule replaces a stack vertex with an identical stack vertex that is connected to a new vertex pc1 representing a client in an initial thread state before the CAS (pc1 refers to its owning stack via edge S). The dotted line indicates how the left-hand-side of the rule is replaced by the right-hand-side: the stack vertex on the left is replaced with the stack vertex on the right. (ii) Spawning may cease when the Nwaps rule is applied. Here, the spawn vertex is replaced with a nwaps vertex. The effect is that both the Spawn and Nwaps rules are disabled, but the remaining rules now become enabled. (iii) In the Prepare rule, a client reads the stack's Top pointer and prepares a new element (pointed to by x) to be pushed or popped onto the stack. There are then two cases that correspond to whether or not the CAS operation succeeds (depending on whether the local pointer t agrees with Top). (iv) In the Succeed case, the stack is updated to point to the new element and the old element is disregarded. This is a generalization that encompasses both push and pop. (v) Alternatively, the CAS may fail, as given by the Fail case. The stack is unchanged and the client forgets what it read and retries.

We can prove that Treiber's stack is lock-free by showing that its depth-bounded abstraction always terminates modulo a weak fairness constraint. The fairness constraint is that the Nwaps rule cannot be continuously enabled without being applied, i.e., a fair run of the system will only spawn finitely many clients. It does not matter whether we allow process spawning only in an initial phase (as in our model), or at any time.

The key contribution of this paper is a technique that automatically constructs a precise numerical abstraction of a depth-bounded system from a given inductive invariant of the system. We refer to this numerical abstraction as the *structural counter abstraction*. The structural counter abstraction then enables us to prove weakly fair termination of the system. Our approach utilizes existing reachability analyses for well-structured transition systems to obtain the inductive invariant, and existing termination analyses for numerical programs to prove termination of the structural counter abstraction. In the following, we explain the construction of the counter abstraction for Treiber's stack.

Nested graphs. Above we saw that graph rewrite rules transform a subcomponent of a concrete graph into another concrete graph. However, we will need to work with (potentially infinitely many) instances of graph subcomponents. So we instead work with *nested graphs* (formal definitions provided in Section 5) in which subcomponents are given counters that indicate an upper bound on how many times they may be duplicated. For Treiber's stack, consider this abstract graph on the left hand side:



The set of concrete graphs represented by this nested graph are those in which the dotted subcomponents are repeated some number of times but at most as many times as determined by the associated counter. For instance, the left dotted subgraph is repeated at most n times. A component may itself contain nested sub-components. An example of an unfolded concrete graph is given on the right hand side. Notice that the pc2 vertices occur at different frequencies per node vertex. Also note that counters always refer to the *total* number of copies of their component. This representation can be thought of as a more precise alternative to counter abstractions [3,7,21], in that we associate counters with nested graph components rather than merely program locations. We say that a nested graph $\widehat{G}_1$ is *covered* by nested graph $\widehat{G}_2$ if the set of concrete graphs obtainable from unfoldings of $\widehat{G}_2$ is contained within the set of concrete graphs obtainable from unfoldings of $\widehat{G}_1$. Determining whether $\widehat{G}_2$ covers $\widehat{G}_1$ is decidable and, as we will see, helps ensure that the structural counter abstraction can be effectively computed.

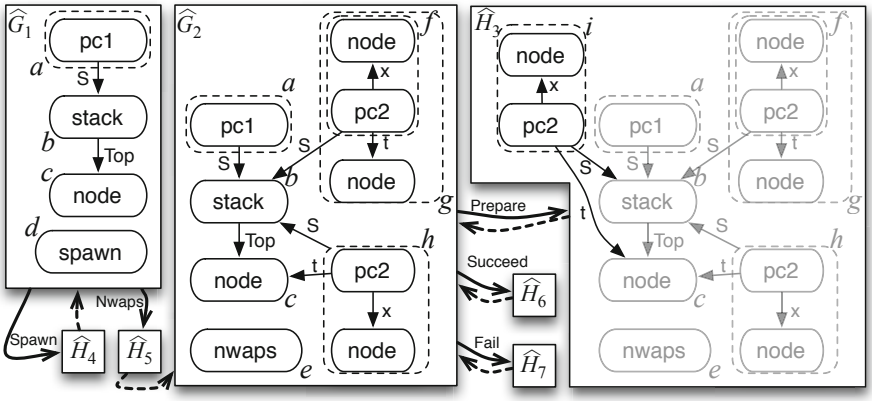


Fig. 2. Structural counter abstraction for Treiber's stack. Numerical transition constraints are omitted for readability. Here the inductive invariant is given by nested graphs $\widehat{G}_1$ and $\widehat{G}_2$.

Obtaining the structural counter abstraction. We begin with a nested graph representation of the inductive invariant. For Treiber's stack the invariant is $\widehat{G}_1$ and $\widehat{G}_2$ in Fig. 2. This invariant (obtained, *e.g.*, via [27]) is a finite set of nested graphs and is an over-approximation of the reachable states of the system. $\widehat{G}_1$ describes states in which spawning may still occur (indicated with a spawn vertex) and $\widehat{G}_2$ describes states in which spawning has ceased (indicated with a nwaps vertex) and arbitrarily many clients have performed Prepare, Suceed or Fail.

We begin to construct the structural counter abstraction by associating a counter variable with each subcomponent of each nested graph in the inductive invariant. For example in Fig. 2, we have established counter variables a, b, c, d with components of $\widehat{G}_1$ and additional counter variables e, f, g, h in $\widehat{G}_2$. In our generation of the structural counter abstraction, we leverage the fact that the invariant is *closed* under rewrite rules. That is, whenever we apply a rewrite rule to a nested graph $\widehat{G}$ in the inductive invariant, we obtain $\widehat{H}$ that is already covered by some other nested graph $\widehat{G}'$ in the invariant.

To construct the abstraction, we apply each rewrite rule, one at a time, for every possible match in one of the nested graphs in the invariant. For example, in Fig. 2 we can apply the Prepare rule as follows. We first unfold one instance of the pc1 vertex a in $\widehat{G_2}$, obtaining a separate pc1 vertex to which we apply the Prepare rule. This produces a new nested graph $\widehat{H_3}$ that extends $\widehat{G_2}$ with a new subgraph. We add a new counter variable i for this new subgraph in $\widehat{H_3}$. Notice that, because the inductive invariant is maximal, $\widehat{H_3}$ is covered by the existing graph $\widehat{G_2}$ (hence the dotted edge from $\widehat{H_3}$ to $\widehat{G_2}$). It is covered because the isomorphic subgraphs with associated counters i and h in $\widehat{H_3}$ can both be represented by the subgraph with associated counter h in $\widehat{H_3}$. From the point of view of the concrete graph transformation system, we can think of this covering edge as an ϵ-transition: every rewrite rule that is susequently applied to $\widehat{H_3}$ can also be applied to $\widehat{G_2}$. The structural counter abstraction is a numerical transition system that reflects the corresponding changes to the counter values when rewrite and covering edges between nested graphs are taken. There are several other possible instances where rules can be applied to this inductive invariant. (These involve graphs $\widehat{H_4}$, $\widehat{H_5}$, $\widehat{H_6}$, and $\widehat{H_7}$ which have been omitted for lack of space.) For example, one can apply the Spawn rule in $\widehat{G_1}$ and obtain $\widehat{H_4}$ which has two pc1 subgraphs. This new graph $\widehat{H_4}$ is, again, covered by $\widehat{G_1}$ and the two pc1 subgraphs can be merged into the pc1 subgraph in $\widehat{G_1}$.

Structural counter abstraction. The structural counter abstraction is represented as a simple control-flow graph program $\mathcal{N} = (Locs, s_0, Vars, \Delta)$. Here, $Locs$ refers to the control locations. There is one location per nested graph in the inductive invariant, respectively, per nested graph obtained by application of a rewriting rule. The variables $Vars$ are the structural counters in the nested graphs, and Δ is a set of commands that change the counter values according to the rewriting and covering steps. s_0 is the initial state. An excerpt of the structural counter abstraction for Treiber's stack that captures parts of Fig. 2 is as follows:

$$\mathcal{N} \equiv (\{\ell_1, \ell_2, \ell_3, \ell_4, \ell_5, \ell_6, \ell_7\}, s_0, \{a, b, c, \dots\}, \{(\ell_2, \delta_{23}, \ell_3), (\ell_3, \delta_{32}, \ell_2), ...\}) \text{ where}$$
$$s_0 \equiv (\ell_1, \{b \mapsto 1, c \mapsto 1, d \mapsto 1, \_ \mapsto 0\})$$
$$\delta_{23} \equiv a' = a - 1 \wedge i' = i + 1 \wedge Id|_{\{a,i\}} \quad \delta_{32} \equiv h' = h + i \wedge i' = 0 \wedge Id|_{\{h,i\}}$$

$Id|_S$ is the identity mapping on the variables, excluding those in S. The transition constraint δ_{23} captures the application of the Prepare rule on $\widehat{G_2}$ yielding $\widehat{H_3}$. The transition constraint δ_{32} captures the covering transition from $\widehat{H_3}$ back to $\widehat{G_2}$. The initial state s_0 encodes the initial graph of the system which consists of one spawn, one stack, and one node vertex. The fairness constraints on the original system can be translated to fairness constraints on the structural counter abstraction in a straightforward manner. The structural counter abstraction we produce is then fit to be analyzed by an existing termination analysis tool such as Terminator [6] or ARMC [22].

Prototype. In Section 6 we describe our prototype tool that automates all steps required to prove fair termination of depth-bounded systems: generation of the inductive invariant, construction of the structural counter abstraction, and the final termination proof. It is able to prove fair termination of the Treiber stack model in less than 10 seconds. A simple counter abstraction that distinguishes only between processes at different control locations would yield a system with fair infinite traces. It is crucial to distinguish

between the processes at location pc2 that may still succeed and those that are bound to fail. This is achieved by our more fine-grained structural counter abstraction.

3 Background

Posets and wqos. A *quasi-ordering* $\leq$ is a reflexive and transitive relation $\leq$ on a set X. In the following $X(\leq)$ is a quasi-ordered set. The *downward closure* of $Y \subseteq X$ is $\downarrow Y = \{\, x \in X \mid \exists y \in Y. x \leq y \,\}$, Y is *downward-closed* if $Y = \downarrow Y$. An *upper bound* $x \in X$ of a set $Y \subseteq X$ is such that for all $y \in Y$, $y \leq x$. A nonempty set $D \subseteq X$ is *directed* if any two elements in D have a common upper bound in D. A set $I \subseteq X$ is an *ideal* of X if I is downward-closed and directed. A quasi-ordering $\leq$ on a set X is a *well-quasi-ordering* (wqo) if any infinite sequence $x_0, x_1, x_2, \ldots$ of elements from X contains an increasing pair $x_i \leq x_j$ with $i < j$.

(Well-Structured) Labeled Transition Systems. A *(labeled) transition system* is a tuple $\mathcal{T} = (S, s_0, Act, \longrightarrow)$ where S is a set of states, $s_0 \in S$ an initial state, Act a set of action labels, and $\longrightarrow \subseteq S \times Act \times S$ is a transition relation. We define $s \xrightarrow{a} s'$ iff $(s, a, s') \in \longrightarrow$. For $A \subseteq Act$, we define $s \xrightarrow{A} s'$ iff $s \xrightarrow{a} s'$ for some $a \in A$. We further define the *post operator* for an action a as $\mathsf{post}_a : \mathcal{P}(S) \to \mathcal{P}(S)$ with $\mathsf{post}_a(X) = \{\, x' \in S \mid \exists x \in X. x \xrightarrow{a} x' \,\}$ and extend it to $\mathsf{post}_{\mathcal{T}}$ by $\mathsf{post}_{\mathcal{T}}(X) = \bigcup_{a \in Act} \mathsf{post}_a(X)$. The *reachability set* of a transition system $\mathcal{T}$, denoted $\mathrm{Reach}(\mathcal{T})$, is defined by $\mathrm{Reach}(\mathcal{T}) = \mathrm{lfp}^{\subseteq}(\lambda X.\{s_0\} \cup \mathsf{post}_{\mathcal{T}}(X))$. A set $X \subseteq S$ is called an *invariant* of $\mathcal{T}$ if $\mathrm{Reach}(\mathcal{T}) \subseteq X$, and X is called *inductive* if $\mathsf{post}_{\mathcal{T}}(X) \subseteq X$. A *well-structured transition system* (WSTS) is a tuple $\mathcal{T} = (S, s_0, Act, \to, \leq)$ where $(S, s_0, Act, \to)$ is a transition system and $\leq \subseteq S \times S$ a wqo that is *monotonic* with respect to $\to$, i.e., for all s_1, s_2, t_1, a such that $s_1 \leq t_1$ and $s_1 \xrightarrow{a} s_2$, there exists t_2 such that $t_1 \xrightarrow{a} t_2$ and $s_2 \leq t_2$. The *covering set* of a well-structured transition system $\mathcal{T}$, denoted $\mathrm{Cover}(\mathcal{T})$, is defined by $\mathrm{Cover}(\mathcal{T}) = \downarrow \mathrm{Reach}(\mathcal{T})$.

Graphs. We use a standard notation for (directed) graphs, denoted as tuples of the form (V, E), with $E \subseteq V \times V$. We define *(vertex) labeled graphs* over a set of labels VL as graphs with labels for each vertex and denote them as (V, E, ν) where $\nu : V \to VL$ is the *vertex-labeling function*. For the rest of the paper we fix VL, a finite set of labels and we denote by *Graphs* the set of all labeled graphs with labels VL. Also, unless explicitly stated otherwise, whenever we say graph, we refer to a labeled graph. We use the standard notions of (partial) homomorphisms, isomorphisms, subgraphs, etc. For a set $V' \subseteq V$ of vertices of a graph $G = (V, E)$, we denote by $G[V'] = (V', E \cap V' \times V')$ the subgraph *induced* by V'. We further denote by $\preceq$ the quasi-ordering induced by subgraph isomorphisms, i.e., $G \preceq H$ iff G is isomorphic to a subgraph of H. We write $G \cong H$ if G and H are isomorphic.

Graph Transformation Systems. We use an adaptation of the standard notion of graph transformation systems with the single pushout approach [9] to labeled directed graphs. A *rewriting rule* is a partial morphism $r : G_L \rightharpoonup G_R$, where G_L is called *left-hand side* and G_R is called *right-hand side*. A *match* of r is a total injective morphism $m : G_L \to G$. Given a rule r and a match $m : G_L \to G$, a *rewriting step* is the *pushout* of r and

m, which consists of a graph H and two graph morphisms $r' : G \rightharpoonup H$, $m' : G_R \to H$ such that $m' \circ r = r' \circ m$ and for every pair of morphisms $r'' : G \rightharpoonup H'$ and $m'' : G_R \rightharpoonup H'$ there exists a unique morphism $f : H \rightharpoonup H'$ with $f \circ m' = m''$ and $f \circ r' = r''$. It is known that pushouts are guaranteed to exist, that they are unique up to isomorphism and that they can be effectively constructed. A *graph transformation system* (GTS) $\mathcal{R}$ is a tuple (R, G_0), where R is a set of rewriting rules and G_0 an initial graph. A GTS $\mathcal{R} = (R, G_0)$ induces a transition system $\mathcal{T}(\mathcal{R}) = (Graphs, G_0, R, \xrightarrow{R})$ where R is a finite set of rewriting rules, and $\xrightarrow{R}$ is the union of all relations $\xrightarrow{r}$, for $r \in R$. The subgraph ordering $\preceq$ is monotonic with respect to graph rewriting.

Lemma 1. *Let $\mathcal{R} = (R, G_0)$ be a GTS, then $\preceq$ is monotonic with respect to $\xrightarrow{R}$.*

4 Weakly Fair Termination of Depth-Bounded Systems

In this section, we formally define the class of systems that we consider in this paper and the type of questions that we answer about these systems.

The *depth* of a graph G is the length of the longest simple path in the undirected version of G, obtained by taking the symmetric closure of the edges. For $k \in \mathbb{N}$, we denote by $\mathcal{G}_{\leq k}$ the set of all graphs with depth at most k. We say that a set of graphs $\mathcal{G}$ is *depth-bounded* if $\mathcal{G} \subseteq \mathcal{G}_{\leq k}$ for some $k \in \mathbb{N}$. A *depth-bounded system* (DBS) is a GTS $\mathcal{R} = (R, G_0)$, whose reachable configuration graphs are depth-bounded, i.e., $\mathrm{Reach}(\mathcal{T}(\mathcal{R})) \subseteq \mathcal{G}_{\leq k}$, for some $k \in \mathbb{N}$. We call k a *bound* of the system. From [26, Proposition 12] it follows that $\preceq$ is a wqo on depth-bounded sets of graphs.

Lemma 2. *For any $k \in \mathbb{N}$, $(\mathcal{G}_{\leq k}, \preceq)$ is a wqo.*

Thus, Lemmas 1 and 2 imply that depth-bounded GTSs induce WSTSs.

Theorem 3. *Let $\mathcal{R} = (R, G_0)$ be a DBS, then $(\mathrm{Cover}(\mathcal{R}), G_0, R, \xrightarrow{R}, \preceq)$ is a WSTS.*

Let $\mathcal{T} = (S, s_0, Act, \to)$ be a transition system. A *finite trace* π of $\mathcal{T}$ is a sequence $s_0\, a_0\, s_1\, a_1 \ldots a_{n-1}\, s_n$, with $s_i \in S$ and $a_i \in Act$ such that $s_i \xrightarrow{a_i} s_{i+1}$, for all $0 \leq i < n$; we define infinite traces $s_0\, a_0\, s_1\, a_1 \ldots$ correspondingly. We say that an action $a \in Act$ is *enabled* in a state s, if there exists a state s' such that $s \xrightarrow{a} s'$. Let $\mathcal{F} = \{A_0, \ldots, A_m\}$ be a set of subsets of Act. An infinite trace $s_0\, a_0\, s_1\, a_1 \ldots$ is *weakly fair* with respect to $\mathcal{F}$ if for every A_j, $0 \leq j \leq m$, there are infinitely many i such that $a_i \in A_j$ or there are infinitely many i such that no action in A_j is enabled in s_i.

Definition 4. *Given a transition system $\mathcal{T}$ and a finite set $\mathcal{F}$ of sets of actions of $\mathcal{T}$, the* weakly fair non-termination problem *asks whether there exists an infinite trace π of $\mathcal{T}$ such that π is weakly fair with respect to $\mathcal{F}$. We refer to the complementary problem as the* weakly fair termination problem *(WFT).*

Theorem 5. *Weakly fair termination is undecidable for depth-bounded systems.*

The proof of Theorem 5 goes by reduction of the structural termination problem for transfer nets to WFT of transfer nets. The former problem is known to be undecidable [16]. Transfer nets are subsumed by depth-bounded systems.

5 Structural Counter Abstraction

We now see the formal treatment of how one obtains the structural abstraction of a given depth-bounded system and how it is used to give approximate answers to the weakly fair termination problem. For the remainder of this section, let $\mathcal{R}$ be a depth-bounded system. We systematically construct the structural counter abstraction of $\mathcal{R}$ from an inductive invariant of $\mathcal{R}$. However, we are not interested in arbitrary inductive invariants but in those that are downward-closed with respect to graph embedding. Since graph embedding is a wqo on depth-bounded graphs, such downward-closed sets are finite unions of ideals of the embedding order [27]. Each ideal can itself be finitely represented and we can compute symbolically the effect of transition on this representation. This enables us to compute a form of closure on the inductive invariant that yields the structural counter abstraction. We start by formalizing this representation of ideals.

Nested graphs. We represent downward-closed depth-bounded sets of graphs as finite sets of *nested graphs*. Formally, a *nested graph* $\widehat{G}$ is a tuple (V, E, ν, l) where (V, E, ν) is a labeled graph and $l : V \to \mathbb{N}$ maps each vertex to its *nesting level*. We abuse notation and denote the labeled graph of a nested graph $\widehat{G}$ by G. We extend the notion of homomorphism to nested graphs as expected, i.e., homomorphisms on nested graphs also preserve the nesting levels of vertices.

Meaning of nested graphs. Intuitively, a nested graph $\widehat{G}$ represents the set of *concrete* graphs that can be obtained by recursively unfolding the nested subgraphs of $\widehat{G}$ arbitrarily often. In the following, we make these notions formal.

We define a *one-step unfolding* relation on nested graphs $\widehat{G} = (V, E, \nu, l)$ and $\widehat{H} = (V', E', \nu', l')$, denoted $\widehat{G} \rightsquigarrow \widehat{H}$, as follows. For $i \geq 1$, denote all vertices at nesting level i or higher by $V_{\geq i} = \{\, v \in V \mid l(v) \geq i \,\}$. Unfolding involves *duplicating* the subgraph induced by $V_{\geq i}$ and reducing the nesting level of all vertices in the copy of $V_{\geq i}$ by one. Formally, we have $\widehat{G} \rightsquigarrow \widehat{H}$ iff for some $i \geq 1$ there exists a partition U, W_1, W_2 of V' and a homomorphism $h : H \to G$ such that $H[U \cup W_1] \cong G \cong H[U \cup W_2]$, $H[W_1] \cong G[V_{\geq i}] \cong H[W_2]$ under (natural restrictions of) h, $W_1 \times W_2 \cap E' = \emptyset$, for all $v' \in V' \setminus W_2$, $l'(v') = l(h(v'))$, and for all $v' \in W_2$, $l'(v') = l(h(v')) - 1$.

We then define the concretization $\gamma(\widehat{G})$ of a nested graph $\widehat{G}$ as the downward closure (with respect to the embedding order) of the set of all unfoldings of $\widehat{G}$: $\gamma(\widehat{G}) = {\downarrow}\{\, H \mid \widehat{G} \rightsquigarrow^* \widehat{H} \,\}$. We extend γ to sets of nested graphs $\widehat{\mathcal{G}}$ as expected: $\gamma(\widehat{\mathcal{G}}) = \bigcup_{\widehat{G} \in \widehat{\mathcal{G}}} \gamma(\widehat{G})$.

Inclusion of Nested Graphs. We next show that inclusion on nested graphs is decidable. Let $\widehat{G} = (V, E, \nu, l)$ and $\widehat{H} = (V', E', \nu', l')$ be nested graphs. Define the relation $\sqsubseteq$ on nested graphs as $\widehat{G} \sqsubseteq \widehat{H}$ iff $\gamma(\widehat{G}) \subseteq \gamma(\widehat{H})$. An *inclusion mapping* for $\widehat{G}$ and $\widehat{H}$ is a homomorphism $\widehat{h} : (V, E, \nu) \to (V', E', \nu')$ satisfying the following additional properties: (i) for all $v \in V$, $l(v) \leq l'(\widehat{h}(v))$; (ii) $\widehat{h}$ is injective with respect to level 0 vertices in V': for all $v, w \in V$, $v' \in V'$, $\widehat{h}(v) = \widehat{h}(w) = v'$ and $l'(v') = 0$ implies $v = w$; (iii) for all distinct $u, v, w \in V$ such that $\widehat{h}(u) = \widehat{h}(v)$, and u and v are both neighbors of w, $l(u) > l(w)$ and $l(v) > l(w)$.

Theorem 6. *Let $\widehat{G}$ and $\widehat{H}$ be nested graphs. Then $\widehat{G} \sqsubseteq \widehat{H}$ iff there exists an inclusion mapping $\widehat{h} : \widehat{G} \to \widehat{H}$. The problem of deciding the existence of $\widehat{h}$ is NP-complete.*

To see that the problem is in NP, note that each of the conditions for inclusion mapping can be checked in polynomial time. NP-hardness follows from the fact that the problem subsumes the subgraph isomorphism problem.

Nested graph rewriting. We lift application of rewrite rules to nested graphs by using inclusion mappings as the notion of a *match*. Intuitively, inclusion mappings allow us to apply the rewrite rule to an unfolding of the graph that contains the left-hand-side of the rule as a subgraph. Formally, we extend the notion of pushout to nested graphs in a natural way by using the homomorphisms defined on nested graphs. For a rewriting rule $r : G_L \to G_R$, naturally lift the notion and define $\widehat{r} : \widehat{G_L} \to \widehat{G_R}$. A *match* of $\widehat{r}$ is an inclusion mapping $\widehat{m} : \widehat{G_L} \to \widehat{G}$.

Lemma 7. *Given a rule $\widehat{r} : \widehat{G_L} \to \widehat{G_R}$ and a match $\widehat{m} : \widehat{G_L} \to \widehat{G}$, there exists a nested graph $\widehat{G}'$ and an injective inclusion mapping $\widehat{h} : \widehat{G_L} \to \widehat{G}'$ such that $\widehat{G} \rightsquigarrow^* \widehat{G}'$. Moreover, $\widehat{G}'$ and $\widehat{h}$ can be constructed in polynomial time.*

Let $\widehat{G}'$ be the nested graph and $\widehat{h} : \widehat{G_L} \to \widehat{G}'$ the injective inclusion mapping, as described in Lemma 7. Then there exists a pushout $\widehat{r}' : \widehat{G}' \rightharpoonup \widehat{H}, \widehat{h}' : \widehat{G_R} \to \widehat{H}$ for $\widehat{r}$ and $\widehat{h}$. This pushout defines a *rewriting step of nested graphs* $\widehat{G} \xrightarrow{\widehat{r}} \widehat{H}$.

Constructing the structural counter abstraction. In the following, we assume that $\widehat{\mathcal{I}}$ is a finite set of nested graphs such that $\gamma(\widehat{\mathcal{I}})$ is a downward-closed inductive invariant of $\mathcal{R}$. From $\widehat{\mathcal{I}}$ we then construct the structural counter abstraction. The precision of this abstraction depends on the precision of $\widehat{\mathcal{I}}$. The most precise downward-closed inductive invariant of $\mathcal{R}$ is the covering set $\mathrm{Cover}(\mathcal{T}(\mathcal{R}))$. Unfortunately, this set is in general not computable for depth-bounded systems[1], even though the covering problem[2] is decidable [26]. However, we can employ existing algorithms [27] that compute downward-closed inductive approximations of the covering set. In practice, these algorithms often yield precisely $\mathrm{Cover}(\mathcal{T}(\mathcal{R}))$. This is confirmed by our experiments in Section 6. In fact, we did not encounter a significant precision loss in any of our examples.

Let G_0 be the initial graph of $\mathcal{R}$ and let $\widehat{G_0}$ be the nested graph obtained by equipping G_0 with a nesting level function mapping all nodes to 0. Further, let R be the set of rewriting rules of $\mathcal{R}$. We define a set of *rewriting edges* E_R as follows: $E_R = \{ (\widehat{G}, r, \widehat{H}) \mid \widehat{G} \in \widehat{\mathcal{I}}, r \in R, \widehat{H} \in \widehat{\mathcal{G}}, \widehat{G} \xrightarrow{\widehat{r}} \widehat{H} \}$. That is, E_R describes the set of one step rule applications on the nested graphs in the inductive invariant. The set E_R is finite up to isomorphism of nested graphs. Next, define the set $\widehat{\mathcal{J}} = \{\widehat{G_0}\} \cup \{ \widehat{H} \mid (\widehat{G}, r, \widehat{H}) \in E_R \}$. From the fact that $\widehat{\mathcal{I}}$ is an inductive invariant it follows that, for all $\widehat{H} \in \widehat{\mathcal{J}}$ there exists $\widehat{G} \in \widehat{\mathcal{I}}$ such that $\widehat{H} \sqsubseteq \widehat{G}$. Fix one such $\widehat{G}$ for each $\widehat{H} \in \widehat{\mathcal{J}}$ and let E_C be the set of all pairs $(\widehat{H}, \widehat{G})$. We call the elements of E_C *covering edges*. Let $\mathcal{E} = E_R \cup E_C$. In Fig 2, we saw this construction for the example of Treiber's stack starting with an inductive invariant. The solid edges between nested graphs correspond to rewrite edges and the dashed ones to covering edges. At the end of Section 2, we also saw an excerpt of the counter abstraction, next we describe how this is done in general.

[1] This follows from the undecidability of place-boundedness of transfer nets [8].

[2] The covering problem for DBS asks whether for given a $\mathcal{R}$ and graph G, $G \in \mathrm{Cover}(\mathcal{T}(\mathcal{R}))$.

The abstraction is a tuple $\mathcal{N} = (Locs, s_0, Vars, \Delta)$ where $Locs = \{ \ell_{\widehat{G}} \mid \widehat{G} \in \widehat{\mathcal{I}} \cup \mathcal{J} \}$ is a set of control locations, $Vars = \{ x_v \mid v \in V(\widehat{G}), \widehat{G} \in \mathcal{I} \cup \mathcal{J} \}$ is a set of counter variables , one for each vertex of a nested graph in $\mathcal{I} \cup \mathcal{J}$, and $\Delta = \{ \delta_e \mid e \in \mathcal{E} \}$ is a set of commands, one for each edge in $\mathcal{E}$. The command δ_e associated with an edge $e = (\widehat{G}, \widehat{H})$ is of the form $(\ell_{\widehat{G}}, \rho_e, \ell_{\widehat{H}})$ where ρ_e is a *transition constraint* over primed and unprimed versions of the variables in $Vars$. The initial state of $\mathcal{N}$ is $s_0 = (\ell_{\widehat{G_0}}, \eta_0)$ where η_0 is a function mapping counters to natural numbers and defined as $\eta_0(x_v) = 1$ if $v \in V(\widehat{G_0})$, and 0 otherwise. Further, let $\sigma_{\mathcal{R}} : \Delta \rightharpoonup R$ be a partial mapping defined as $\sigma_{\mathcal{R}}(\delta_e) = r$ if e is a rewriting edge for rule r.

The definition of the transition constraint δ_e for an edge $e \in \mathcal{E}$ depends on whether e is a rewriting or a covering edge. We first consider the case that e is a rewriting edge $(\widehat{G}, r, \widehat{H})$. In order to perform a rewrite (which only transforms level-0 vertices) we need to unfold the graph $\widehat{G}$. As mentioned in Lemma 7, this can be done efficiently giving us $\widehat{G} \rightsquigarrow^* K$. Each unfolding step gives a homomorphism, which can be composed together to give $h : K \to \widehat{G}$. Further, from the pushout we get a partial homomorphism $r' : K \rightharpoonup \widehat{H}$. Let V be the vertices of $\widehat{G}$, U the vertices of K, and W the vertices of $\widehat{H}$. Further, let U_0 be the level-0 vertices of K and define $\overline{U_0} = U \setminus U_0$. Similarly, let W_0 be the level-0 vertices of $\widehat{H}$. Then, the transition constraint ρ_e for e is given by the conjunction of the following constraints:

$$x_v = \sum_{u \in h^{-1}(v) \cap \overline{U_0}} x'_{r'(u)} + \left| h^{-1}(v) \cap U_0 \right|, \quad \text{for all } v \in V \tag{1}$$

$$x'_w = 1, \quad \text{for all } w \in W_0 \tag{2}$$

$$y' = 0, \quad \text{for all } y \in Vars \setminus \{ x_w \mid w \in W \} \tag{3}$$

During unfolding of $\widehat{G}$ to $\widehat{H}$, if some vertex v with count x_v is duplicated, then constraint (1) ensures that all counts for the duplicates sum up to x_v. Level-0 vertices get a special treatment, since they may be transformed by the rewrite rule. Similarly, (2) takes care of level-0 vertices in the rewritten graph. The constraint (3) encodes that only counters of vertices associated with the successor location have non-zero values.

For covering edges $e = (\widehat{H}, \widehat{G})$, we use the inclusion mapping $\widehat{h} : \widehat{H} \to \widehat{G}$ between the two nested graphs to define the transition constraint δ_e. Let W be the vertices of $\widehat{G}$, W_0 the level-0 vertices of $\widehat{G}$, and V the vertices of $\widehat{H}$. The inclusion mapping encodes which vertices $v \in V$ are collapsed to a single vertex $w \in W$, yielding the constraint

$$x'_w = \sum_{v \in \widehat{h}^{-1}(w)} x_v, \quad \text{for all } w \in W \tag{4}$$

Then δ_e is the conjunction of constraint (4) and constraints (2) and (3), which are the same as in the case of a rewriting edge.

Finally, the fairness constraints $\mathcal{F}_{\mathcal{R}}$ of $\mathcal{R}$ can be translated to fairness constraints $\mathcal{F}_{\mathcal{N}}$ of $\mathcal{N}$ using the partial function $\sigma_{\mathcal{R}}$ as follows: $\mathcal{F}_{\mathcal{N}} = \{ \sigma_{\mathcal{R}}^{-1}(R_i) \mid R_i \in \mathcal{F}_{\mathcal{R}} \}$.

The numerical abstraction induces a transition system $\mathcal{T}(\mathcal{N}) = (S, s_0, \Delta, \xrightarrow{\Delta})$ with states $S = Locs \times \mathbb{N}^{Vars}$, i.e., a program location along with an evaluation of the

counters. The transition relation $\xrightarrow{\Delta}$ is as expected. The details of the following soundness theorem may be found in the technical report [2].

Theorem 8 (Soundness). *If $(\mathcal{T}(\mathcal{R}), \mathcal{F}_{\mathcal{R}})$ has a weakly fair infinite trace, then so does $(\mathcal{T}(\mathcal{N}), \mathcal{F}_{\mathcal{N}})$.*

6 Evaluation

We implemented a prototype of our algorithm as an extension to the PICASSO [20, 27] tool. PICASSO takes as input a depth-bounded systems and computes a so called *abstract coverability tree* (ACT). The nodes of the ACT are nested graphs and its construction is similar to the Karp-Miller tree for Petri nets. The maximal nodes in the ACT form a downward-closed inductive invariant, $\widehat{\mathcal{I}}$, of the input system. From this invariant we generate a structural counter abstraction, $\mathcal{N}$, that is optimized and then analyzed with the ARMC [22] termination prover.

A naive implementation of the method described in Section 5 produced structural counter abstractions that were too big for current termination provers. For instance, for Treiber's stack, having one variable for each vertex of each nested graph in the inductive invariant and those obtained by applying rewrite rules led to an abstraction with over 170 variables and 40 transitions. We therefore optimized the generation of the abstraction to get a smaller counter program with the same termination properties. When we generate the constraints for a transition, we decompose the transition into three steps: unfolding, morphism, and covering. These steps lead to many intermediate locations and transitions. We eliminate the intermediate steps by using the quantifier elimination procedure for linear integer arithmetic in PRINCESS [24]. We collect the constraints generated for each step and quantify away the variables at the intermediate locations. The resulting constraint describes a single transition with the same source and target locations as the original three-step transition, using only the variables at those locations. Furthermore, we observed that in many places constant values are assigned to the variables because they represent nodes on nesting level 0. We propagate the constant values using a combination of lightweight abstract interpretation and constraint propagation. We use an abstract domain that maps the variables to $\mathbb{N} \cup \bot$. A variable v is mapped to a value n in $\mathbb{N}$ when we can deduce that v is always equal to n, otherwise v is mapped to $\bot$.From the abstract fixed point we extract variable/value pairs and eliminate the variables by replacing them with their associated values. Lastly, instead of using one variable per node and graph, we reuse the variables across different graphs. The renaming is done by finding a minimal coloring of a graph where the nodes are variables and there is an edge between two nodes if the corresponding variables are used at the same location.For Treiber's stack, we reduced the abstraction to 6 variables and 4 transitions.

Transition predicates. We observed that ARMC finds easily the predicates that involve one or two variables, but not the predicates requiring more variables. Fortunately, ARMC can take transition predicates as part of its input. We manually give hints to PICASSO in the form of variables names, usually corresponding to control-states. Those names are turned into transition predicates by summing the variables. For example, in the numerical abstraction of Treiber's stack we specified a simple predicate indicating that the sum of all the process counters was either unchanged or decreasing.

Table 1. Experimental results. The columns show the number of locations, variables, and transitions in the counter abstraction, and the running times, in seconds, for computing the inductive invariant, constructing the abstraction, and for proving termination.

Example	#loc	#v	#t	$\widehat{\mathcal{I}}$	$\mathcal{N}$	ARMC	**Total**
Split/merge	4	3	9	1.5	6.8	0.1	8.4
Work stealing, 3 processors	4	4	20	1.7	13.1	0.2	15.0
Work stealing, parameterized	2	3	4	1.5	5.6	0.1	6.2
Compute server job queue	2	5	4	1.6	6.1	0.1	7.8
Chat room	5	34	80	9.8	61.3	5 min	6 min
Map reduce	6	10	15	2.0	8.8	0.2	11.0
Map reduce with failure	6	15	21	2.3	11.1	0.9	14.3
Treiber's stack (coarse-grained)	2	6	4	1.9	7.2	0.2	9.3
Treiber's stack (fine-grained)	3	14	13	2.7	14.2	1.2	17.1
Herlihy/Wing queue	3	16	25	3.8	24.9	6.5	34.2
Michael/Scott queue (dequeue only)	4	7	23	2.8	13.0	0.6	16.4
Michael/Scott queue (enqueue only)	7	15	53	3.8	43.7	7.6	55.1
Michael/Scott queue	9	31	224	25.0	265.0	3 wks	3 wks

Results. Table 1 summarizes the results of our experiments. Our implementation is parallelized and ran on a server using 26 cores. Memory consumption was not an issue. We examined a collection of depth-bounded transition systems, including distributed processes and concurrent algorithms. The examples and the tool can be downloaded from the PICASSO web site [20]. We applied our method to prove global progress properties of those systems. Fairness is used to limit the number of clients, requests, and failures. Details about the encoding of fairness constraints can be found in the technical report [2]. Our experiments show that our approach can quickly prove termination of complex systems. The structural counter abstraction is concise and maintains the necessary information in order to prove termination.

The split/merge example is a parallel computation where a master sends jobs to a pool of workers. We also proved termination of (non-)parameterized versions of a work stealing algorithm. From [13] we considered systems obtained from Scala implementations of a chat room and a map reduce algorithm (with and without failure). As shared memory examples, we considered the model of Treiber's stack [25] described in Section 2 as well as a more fine-grained variant with push and pop modeled independently. We analyzed a model of the Herlihy/Wing concurrent queue [14] which requires an additional fairness constraint to ensure that dequeue operations cannot execute without enqueue operations ever taking steps. This is needed because the dequeue operation retries if the queue is empty. Finally, we also considered the Michael/Scott queue [19] where the order between the elements is abstracted. This example results in an abstraction that is very large for today's termination provers. We therefore also show the results for simpler models where enqueue and dequeue operations are considered in isolation.

7 Related Work

Depth-bounded systems (DBS) were first introduced by Meyer in [17] as a fragment of the π-calculus. In his paper, he showed that DBS are well-structured and that termination (without fairness constraints) is decidable. Termination without fairness has only limited practical applications because the initial state of the system is fixed. With a fixed initial state one cannot model systems with an infinite set of reachable states without losing termination, since we only consider finitely branching systems.

The idea of using reachability analyses to obtain numerical abstractions of programs whose states can be described by graphs is by itself not new. In particular, such techniques have been studied for proving safety and liveness properties of heap manipulating programs [4, 12, 23]. Our technique differs substantially from these approaches in the way the numerical abstraction is computed. Specifically, our technique is based on *ideal abstractions* [27] for computing over-approximations of the covering sets of WSTS and it exploits the monotonicity of the analyzed system, i.e., that the behavior observable from a given graph is subsumed by the behavior observable from any larger graph. Finally, the abstract domain of nested graphs can model unbounded recursive unfolding structures that naturally occur in complex concurrent systems and that are difficult to capture using traditional shape analysis domains.

Joshi and König study graph transformation systems that are well-structured with respect to the graph minor ordering [15]. Our approach targets a different application domain. We consider rewriting rules with injective matching. Systems with this semantics are not monotonic with respect to graph minors and therefore not well-structured under this ordering. On the other hand, the graph minor ordering is a wqo for arbitrary graphs, while the subgraph ordering is a wqo only for graphs bounded in the length of their simple paths. The two approaches thus consider orthogonal classes of WSTS.

An application of our results is proving nonblocking properties of concurrent algorithms. Others have considered approaches directly targeted on this goal. Gotsman *et al.* [11] describe a thread-modular proof technique. While their work enables thread-local reasoning, it is only suitable in instances where there are simple environmental invariants (*i.e.* other threads do not execute certain actions infinitely often).

8 Conclusion

We have shown a novel technique for proving fair termination of algorithms described as depth-bounded systems. Despite the fact that this problem is undecidable, we showed that one can build on existing verification techniques to obtain an approximate analysis that is both practical and sufficiently precise to prove fair termination of complex concurrent systems such as Treiber's stack. We have shown that our method is sound, and demonstrated viability with a prototype implementation.

References

1. Abdulla, P.A., Cerans, K., Jonsson, B., Tsay, Y.-K.: General decidability theorems for infinite-state systems. In: LICS, pp. 313–321 (1996)
2. Bansal, K., Koskinen, E., Wies, T., Zufferey, D.: Structural counter abstraction. Technical Report TR2012-947, New York University (2012)

3. Basler, G., Mazzucchi, M., Wahl, T., Kroening, D.: Symbolic Counter Abstraction for Concurrent Software. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 64–78. Springer, Heidelberg (2009)
4. Berdine, J., Cook, B., Distefano, D., O'Hearn, P.W.: Automatic Termination Proofs for Programs with Shape-Shifting Heaps. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 386–400. Springer, Heidelberg (2006)
5. Carstensen, H.: Decidability Questions for Fairness in Petri Nets. In: Brandenburg, F.J., Wirsing, M., Vidal-Naquet, G. (eds.) STACS 1987. LNCS, vol. 247, pp. 396–407. Springer, Heidelberg (1987)
6. Cook, B., Podelski, A., Rybalchenko, A.: Termination proofs for systems code. In: PLDI (2006)
7. Delzanno, G., Raskin, J.-F., Van Begin, L.: Towards the Automated Verification of Multithreaded Java Programs. In: Katoen, J.-P., Stevens, P. (eds.) TACAS 2002. LNCS, vol. 2280, pp. 173–187. Springer, Heidelberg (2002)
8. Dufourd, C., Finkel, A., Schnoebelen, P.: Reset Nets Between Decidability and Undecidability. In: Larsen, K.G., Skyum, S., Winskel, G. (eds.) ICALP 1998. LNCS, vol. 1443, pp. 103–115. Springer, Heidelberg (1998)
9. Ehrig, H., Heckel, R., Korff, M., Löwe, M., Ribeiro, L., Wagner, A., Corradini, A.: Handbook of graph grammars and computing by graph transformation, pp. 247–312. World Scientific Publishing Co., Inc. (1997)
10. Finkel, A., Schnoebelen, P.: Well-structured transition systems everywhere! Theor. Comput. Sci. 256(1-2), 63–92 (2001)
11. Gotsman, A., Cook, B., Parkinson, M.J., Vafeiadis, V.: Proving that non-blocking algorithms don't block. In: POPL. ACM (2009)
12. Gulwani, S., Lev-Ami, T., Sagiv, M.: A combination framework for tracking partition sizes. In: POPL, pp. 239–251. ACM (2009)
13. Haller, P., Sommers, F.: Actors in Scala. Artima (January 2012)
14. Herlihy, M., Wing, J.M.: Linearizability: A correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. 12(3), 463–492 (1990)
15. Joshi, S., König, B.: Applying the Graph Minor Theorem to the Verification of Graph Transformation Systems. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 214–226. Springer, Heidelberg (2008)
16. Mayr, R.: Undecidable problems in unreliable computations. Theor. Comput. Sci. 297(1-3), 337–354 (2003)
17. Meyer, R.: On Boundedness in Depth in the π-Calculus. In: Ausiello, G., Karhumäki, J., Mauri, G., Ong, L. (eds.) Fifth IFIP International Conference on Theoretical Computer Science–TCS 2008. IFIP, vol. 273, pp. 477–489. Springer, Boston (2008)
18. Meyer, R., Gorrieri, R.: On the Relationship between π-Calculus and Finite Place/Transition Petri Nets. In: Bravetti, M., Zavattaro, G. (eds.) CONCUR 2009. LNCS, vol. 5710, pp. 463–480. Springer, Heidelberg (2009)
19. Michael, M.M., Scott, M.L.: Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In: PODC (1996)
20. Picasso, `http://pub.ist.ac.at/~zufferey/picasso/termination`
21. Pnueli, A., Xu, J., Zuck, L.D.: Liveness with $(0, 1, \infty)$-Counter Abstraction. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 107–122. Springer, Heidelberg (2002)
22. Podelski, A., Rybalchenko, A.: ARMC: The Logical Choice for Software Model Checking with Abstraction Refinement. In: Hanus, M. (ed.) PADL 2007. LNCS, vol. 4354, pp. 245–259. Springer, Heidelberg (2007)
23. Podelski, A., Rybalchenko, A., Wies, T.: Heap Assumptions on Demand. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 314–327. Springer, Heidelberg (2008)

24. Rümmer, P.: A Constraint Sequent Calculus for First-Order Logic with Linear Integer Arithmetic. In: Cervesato, I., Veith, H., Voronkov, A. (eds.) LPAR 2008. LNCS (LNAI), vol. 5330, pp. 274–289. Springer, Heidelberg (2008)
25. Treiber, R.: Systems programming: Coping with parallelism. International Business Machines Incorporated, Thomas J. Watson Research Center (1986)
26. Wies, T., Zufferey, D., Henzinger, T.A.: Forward Analysis of Depth-Bounded Processes. In: Ong, L. (ed.) FOSSACS 2010. LNCS, vol. 6014, pp. 94–108. Springer, Heidelberg (2010)
27. Zufferey, D., Wies, T., Henzinger, T.A.: Ideal Abstractions for Well-Structured Transition Systems. In: Kuncak, V., Rybalchenko, A. (eds.) VMCAI 2012. LNCS, vol. 7148, pp. 445–460. Springer, Heidelberg (2012)