

# Structural Counter Abstraction

Kshitij Bansal<sup>1</sup>, Eric Koskinen<sup>1</sup>, Thomas Wies<sup>1</sup>, and Damien Zufferey<sup>2</sup>

<sup>1</sup> New York University

<sup>2</sup> 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. In this paper, 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. It also provides a more precise alternative to traditional counter abstraction for concurrent systems because the number and meaning of counters is not fixed a priori, but depends on the structure of the reachable states. 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 [8] 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 paths in all reachable configuration 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 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 [4]. 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. Each counter tracks how often the associated subgraph occurs in a represented concrete graph. 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 [2, 6, 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 [5, 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 are the first to present an 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 well-structuredness of the analyzed system. Our algorithmic technique of computing a numerical abstraction from an inductive invariant, which we introduce in this paper, promises interesting applications beyond depth-bounded systems and liveness. For instance, the same technique can be used to strengthen a given 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 to

```

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

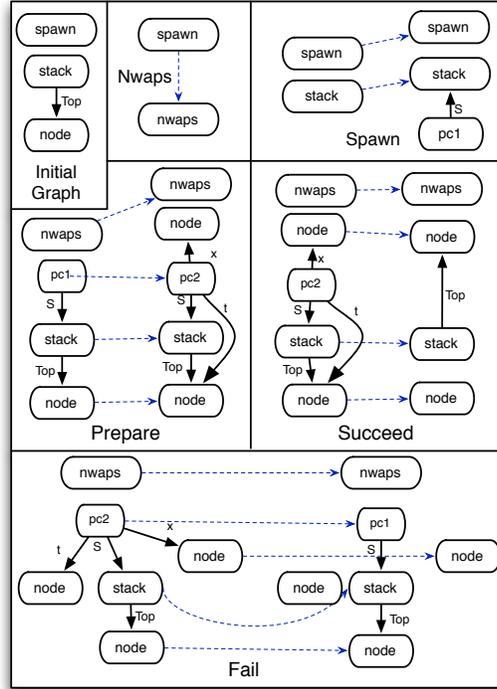
struct stack {
    struct node *Top;
};
struct stack *S;

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] on the left and its abstraction as a depth-bounded graph transformation system on the right.

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 fair-termination of this algorithm.

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. A depth-bounded system is a transition system whose states are graphs such that there is a bound on the length of all simple paths in all reachable graphs of the system. 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.

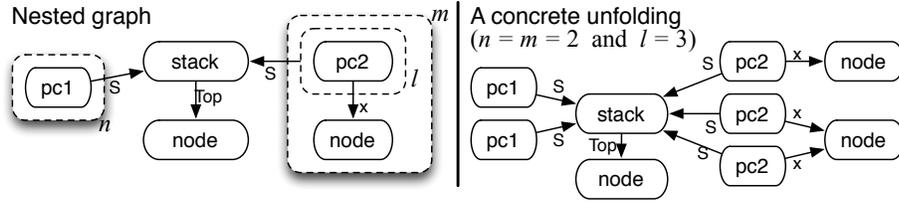
Depth-bounded abstractions of programs can be computed automatically from the program’s source code using shape analysis techniques. These techniques are orthogonal to the contribution of this paper. We refer the interested reader to [9], which describes such a technique in detail. In this paper, we assume that the system to be analyzed is already depth-bounded.

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. 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. 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 head pointer and prepares a new element to be pushed or popped onto the stack. There are then two cases that correspond to whether or not the CAS operation succeeds. (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.

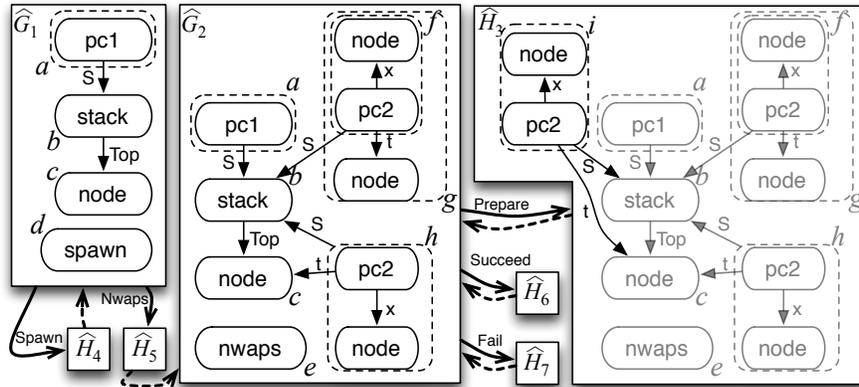
We can prove that Treiber’s stack is lock-free by showing that its depth-bounded abstraction always terminates modulo a weak fairness constraint. The fairness constraint states that the `Nwaps` rule cannot be continuously enabled without being applied, i.e., a fair run of the system will only spawn finitely many clients. Note that it makes no difference whether we only allow spawning of new processes in an initial phase as in our model or whether we allow new processes to be created at any time.

This paper presents an approach for automatically proving weakly fair termination of depth-bounded systems. The key contribution 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*. 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 remainder of this section, we explain the construction of the counter abstraction through the example of Treiber’s stack.

*Nested graphs.* In the example 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* 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 nested 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 [2,6,21] in that we associate counters with nested graph components rather than merely program locations. We say that a nested graph  $\hat{G}_1$  is *covered* by nested graph  $\hat{G}_2$  if the set of concrete graphs obtainable from unfoldings of  $\hat{G}_2$  is contained within the set of concrete graphs obtainable from unfoldings of  $\hat{G}_1$ . Determining whether  $\hat{G}_2$  covers  $\hat{G}_1$  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  $\hat{G}_1$  and  $\hat{G}_2$ .

*Obtaining the structural counter abstraction.* We begin with a nested graph representation of the inductive invariant. For Treiber's stack the invariant is  $\hat{G}_1$  and  $\hat{G}_2$  in Fig. 2. This invariant is a finite set of nested graphs and is an over-approximation of the reachable states of the system. It can be obtained, for example, using the algorithm presented in [27].  $\hat{G}_1$  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`, `Succeed` or `Fail`.

We begin to construct the structural counter abstraction by associating a counter with each subcomponent of each nested graph in the inductive invariant. For example in Fig. 2, we have established counters  $a, b, c, d$  with components of  $\widehat{G}_1$  and additional counters  $e, f, g, h$  in  $\widehat{G}_2$ . In our generation of the structural counter abstraction, we leverage the fact that the inductive invariant is *closed* under application of 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  $i$  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  $\epsilon$ -transition: every subsequent application of a rewrite rule 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 one `pc1` subgraph in  $\widehat{G}_1$ .

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  $\Delta$  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:

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

$Id|_S$  is the identity mapping on the variables, excluding those in  $S$ . The transition constraint  $\delta_{23}$  captures the application of the `Prepare` rule on  $\widehat{G}_2$  yielding  $\widehat{H}_3$ . The transition constraint  $\delta_{32}$  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 [5] or ARMC [22].

*Prototype.* In Section 6 we describe our prototype tool that automates all the steps required to prove fair termination of depth-bounded systems: the generation of the inductive invariant, the 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 Preliminaries

*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  $x_0, x_1, x_2, \dots$  of elements from  $X$  contains an increasing pair  $x_i \leq x_j$  with  $i < j$ . We extend the ordering  $\leq$  to an ordering  $\leq$  on subsets of  $X$  as expected: for  $Y_1, Y_2 \subseteq X$ , we have  $Y_1 \leq Y_2$  iff for all  $y_1 \in Y_1$  there exists  $y_2 \in Y_2$  such that  $y_1 \leq y_2$ .

*(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  $\text{post}_a : \mathcal{P}(S) \rightarrow \mathcal{P}(S)$  with  $\text{post}_a(X) = \{x' \in S \mid \exists x \in X. x \xrightarrow{a} x'\}$  and extend it to  $\text{post}_{\mathcal{T}}$  by  $\text{post}_{\mathcal{T}}(X) = \bigcup_{a \in Act} \text{post}_a(X)$ . The *reachability set* of a transition system  $\mathcal{T}$ , denoted  $\text{Reach}(\mathcal{T})$ , is defined by  $\text{Reach}(\mathcal{T}) = \text{lfp}^{\subseteq}(\lambda X. \{s_0\} \cup \text{post}_{\mathcal{T}}(X))$ . A set  $X \subseteq S$  is called an *invariant* of  $\mathcal{T}$  if  $\text{Reach}(\mathcal{T}) \subseteq X$ , and  $X$  is called *inductive* if  $\text{post}_{\mathcal{T}}(X) \subseteq X$ . A *well-structured transition system* (WSTS) is a tuple  $\mathcal{T} = (S, s_0, Act, \rightarrow, \leq)$  where  $(S, s_0, Act, \rightarrow)$  is a transition system and  $\leq \subseteq S \times S$  a wqo that is *monotonic* with respect to  $\rightarrow$ , 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  $\text{Cover}(\mathcal{T})$ , is defined by  $\text{Cover}(\mathcal{T}) = \downarrow \text{lfp}^{\subseteq}(\lambda X. \downarrow S_0 \cup \text{post}_{\mathcal{T}}(X))$ .

*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  $(V, E, \nu)$  where  $\nu : V \rightarrow VL$  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 [8] to labeled directed graphs. A *rewriting rule* is a partial morphism  $r : G_L \rightarrow G_R$ , 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  $m : G_L \rightarrow G$ . Given a rule  $r$  and a match  $m : G_L \rightarrow G$ , a *rewriting step* is the *pushout* of  $r$  and  $m$ , which consists of a graph  $H$  and two graph morphisms  $r' : G \rightarrow H$ ,  $m' : G_R \rightarrow H$  such that  $m' \circ r = r' \circ m$  and for every pair of morphisms  $r'' : G \rightarrow H'$  and  $m'' : G_R \rightarrow H'$  there exists a unique morphism  $f : H \rightarrow H'$  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. Note that the condition that matches be injective can be dropped leading to a slightly more general but also more involved notion of graph rewriting. Our results carry over to this more general notion. However, injective matches are more natural for the application domains that we consider here. 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  $\mathcal{T}(\mathcal{R}) = (\text{Graphs}, G_0, R, \xrightarrow{R})$  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.,  $\text{Reach}(\mathcal{T}(\mathcal{R})) \subseteq \mathcal{G}_{\leq k}$ , 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  $(\text{Cover}(\mathcal{R}), G_0, R, \xrightarrow{R}, \preceq)$  is a WSTS.*

Let  $\mathcal{T} = (S, s_0, \text{Act}, \rightarrow)$  be a transition system. A *finite trace*  $\pi$  of  $\mathcal{T}$  is a sequence  $s_0 a_0 s_1 a_1 \dots a_{n-1} s_n$ , with  $s_i \in S$  and  $a_i \in \text{Act}$  such that  $s_i \xrightarrow{a_i} s_{i+1}$ , for all  $0 \leq i < n$ ; we define infinite traces  $s_0 a_0 s_1 a_1 \dots$  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  $\mathcal{F} = \{A_0, \dots, A_m\}$  be a set of subsets of  $Act$ . An infinite trace  $s_0 a_0 s_1 a_1 \dots$  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  $\pi$  of  $\mathcal{T}$  such that  $\pi$  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 making this representation of downward-closed depth-bounded sets of graphs formal.

*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  $(V, E, \nu, l)$  where  $(V, E, \nu)$  is a labeled graph and  $l : V \rightarrow \mathbb{N}$  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  $h : H \rightarrow G$  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  $\gamma$  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  $\widehat{h} : (V, E, \nu) \rightarrow (V', E', \nu')$  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  $\widehat{h} : \widehat{G} \rightarrow \widehat{H}$ . Moreover, deciding the existence of inclusion mappings is NP-complete.*

*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  $r : G_L \rightarrow G_R$ , naturally lift the notion and define  $\widehat{r} : \widehat{G}_L \rightarrow \widehat{G}_R$ . A *match* of  $\widehat{r}$  is an inclusion mapping  $\widehat{m} : \widehat{G}_L \rightarrow \widehat{G}$ .

**Lemma 7.** *Given a rule  $\widehat{r} : \widehat{G}_L \rightarrow \widehat{G}_R$  and a match  $\widehat{m} : \widehat{G}_L \rightarrow \widehat{G}$ , there exists a nested graph  $\widehat{G}'$  and an injective inclusion mapping  $\widehat{h} : \widehat{G}_L \rightarrow \widehat{G}'$  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  $\widehat{h} : \widehat{G}_L \rightarrow \widehat{G}'$  the injective inclusion mapping, as described in Lemma 7. Then there exists a pushout  $\widehat{r}' : \widehat{G}' \rightarrow \widehat{H}$ ,  $\widehat{h}' : \widehat{G}_R \rightarrow \widehat{H}$  for  $\widehat{r}$  and  $\widehat{h}$ . This pushout defines an *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  $\text{Cover}(\mathcal{T}(\mathcal{R}))$ . Unfortunately, this set is in general not computable for depth-bounded systems<sup>3</sup>, even though the covering problem<sup>4</sup> 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  $\text{Cover}(\mathcal{T}(\mathcal{R}))$ .

<sup>3</sup> This follows from the undecidability of place-boundedness of transfer nets [7].

<sup>4</sup> The covering problem for DBS asks whether for given a  $\mathcal{R}$  and graph  $G$ ,  $G \in \text{Cover}(\mathcal{T}(\mathcal{R}))$ .

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 such pairs  $(\widehat{H}, \widehat{G})$ . We call the elements of  $E_C$  *covering edges*. Let  $\mathcal{E} = E_R \cup E_C$ .

We now define the actual structural counter abstraction of  $\mathcal{R}$ . The abstraction is a tuple  $\mathcal{N} = (Locs, s_0, Vars, \Delta)$  where  $Locs = \{\ell_{\widehat{G}} \mid \widehat{G} \in \widehat{\mathcal{I}} \cup \widehat{\mathcal{J}}\}$  is a set of control locations,  $Vars = \{x_v \mid v \in V(\widehat{G}), \widehat{G} \in \widehat{\mathcal{I}} \cup \widehat{\mathcal{J}}\}$  is a set of counter variables, one for each vertex of a nested graph in  $\widehat{\mathcal{I}} \cup \widehat{\mathcal{J}}$ , and  $\Delta = \{\delta_e \mid e \in \mathcal{E}\}$  is a set of commands, one for each edge in  $\mathcal{E}$ . The command  $\delta_e$  associated with an edge  $e = (\widehat{G}, \widehat{H})$  is of the form  $(\ell_{\widehat{G}}, \rho_e, \ell_{\widehat{H}})$  where  $\rho_e$  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  $\eta_0$  is a function mapping counters to natural numbers and defined as  $\eta_0(x_v) = 1$  if  $v \in V(\widehat{G}_0)$ , 0 otherwise. Further, let  $\sigma_{\mathcal{R}} : \Delta \rightarrow R$  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  $\delta_e$  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  $h : K \rightarrow \widehat{G}$ . Further, from the pushout we get a partial homomorphism  $r' : K \rightarrow \widehat{H}$ . 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  $\rho_e$  for  $e$  is given by the conjunction of the following constraints:

$$x_v = \sum_{u \in h^{-1}(v) \cap \overline{U}_0} x'_{r'(u)} + |h^{-1}(v) \cap U_0|, \quad \text{for all } v \in V \quad (1)$$

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

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

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. More details for constraints (1) and (2) can be found in the proof of Lemma 9 in Appendix A.

For covering edges  $e = (\widehat{H}, \widehat{G})$ , we use the inclusion mapping  $\widehat{h} : \widehat{H} \rightarrow \widehat{G}$  between the two nested graphs to define the transition constraint  $\delta_e$ . 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 in  $w \in W$ , yielding the constraint

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

Then  $\delta_e$  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  $S = \text{Locs} \times \mathbb{N}^{\text{Vars}}$ , 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 Appendix A.

**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,  $\widehat{\mathcal{L}}$ , 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 naïve 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. To make the method practical, we implemented simplification heuristics. First, using a combination of lightweight abstract interpretation and constraint propagation we eliminated many variables which have a constant value. Second, we compacted sequences of transitions into single transitions by using the quantifier elimination for linear integer arithmetic in PRINCESS [24]. Lastly, instead of using one variable per node and graph, we reused the variables across different graphs. 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.

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	T.O.	T.O.

**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.

*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 Appendix B. Our experiments show that, by generating the structural counter abstraction from an inductive invariant, one can quickly prove termination of complex systems. The abstraction is concise, but 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 applied our analysis to the Michael/Scott queue [19] where the order between the elements is abstracted. This example results in an abstraction that is currently too large for the termination prover. We therefore also show the running times 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  $\pi$ -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 [3, 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 well for arbitrary graphs, while the subgraph ordering is well 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 weakly fair termination is undecidable for such systems, we showed that one can build on existing verification techniques to obtain an approximate analysis for this problem 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. P. A. Abdulla, K. Cerans, B. Jonsson, and Y.-K. Tsay. General decidability theorems for infinite-state systems. In *LICS*, pages 313–321, 1996.

2. G. Basler, M. Mazzucchi, T. Wahl, and D. Kroening. Symbolic counter abstraction for concurrent software. In *CAV*, volume 5643 of *LNCS*, pages 64–78. Springer, 2009.
3. J. Berdine, B. Cook, D. Distefano, and P. W. O’Hearn. Automatic termination proofs for programs with shape-shifting heaps. In *CAV*, pages 386–400. Springer, 2006.
4. H. Carstensen. Decidability questions for fairness in petri nets. In *STACS*, volume 247 of *LNCS*, pages 396–407. Springer, 1987.
5. B. Cook, A. Podelski, and A. Rybalchenko. Termination proofs for systems code. In *PLDI*, 2006.
6. G. Delzanno, J.-F. Raskin, and L. V. Begin. Towards the Automated Verification of Multi-threaded Java Programs. In *TACAS*, volume 2280, pages 173–187. Springer, 2002.
7. C. Dufourd, A. Finkel, and P. Schnoebelen. Reset nets between decidability and undecidability. In *ICALP*, volume 1443 of *LNCS*, pages 103–115. Springer, 1998.
8. H. Ehrig, R. Heckel, M. Korff, M. Löwe, L. Ribeiro, A. Wagner, and A. Corradini. Handbook of graph grammars and computing by graph transformation. pages 247–312. World Scientific Publishing Co., Inc., 1997.
9. S. Esmailsabzali, R. Majumdar, T. Wies, and D. Zufferey. Dynamic package interfaces, 2012. Technical report. Available at <http://cs.nyu.edu/wies/publ/tr-dynamic-package-interfaces.pdf>.
10. A. Finkel and P. Schnoebelen. Well-structured transition systems everywhere! *Theor. Comput. Sci.*, 256(1-2):63–92, 2001.
11. A. Gotsman, B. Cook, M. J. Parkinson, and V. Vafeiadis. Proving that non-blocking algorithms don’t block. In *POPL*. ACM, 2009.
12. S. Gulwani, T. Lev-Ami, and M. Sagiv. A combination framework for tracking partition sizes. In *POPL*, pages 239–251. ACM, 2009.
13. P. Haller and F. Sommers. *Actors in Scala*. Artima, January 2012.
14. M. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.
15. S. Joshi and B. König. Applying the graph minor theorem to the verification of graph transformation systems. In *CAV*, volume 5123 of *LNCS*, pages 214–226. Springer, 2008.
16. R. Mayr. Undecidable problems in unreliable computations. *Theor. Comput. Sci.*, 297(1-3):337–354, 2003.
17. R. Meyer. On boundedness in depth in the  $\pi$ -calculus. In *Fifth Ifip International Conference On Theoretical Computer Science–Tcs 2008*, pages 477–489. Springer, 2008.
18. R. Meyer and R. Gorrieri. On the relationship between pi-calculus and finite place/transition petri nets. In *CONCUR*, pages 463–480, 2009.
19. M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *PODC*, 1996.
20. Picasso: a PI-CALculus-based Static SOFTWARE analyzer. <http://pub.ist.ac.at/~zufferey/picasso/termination>.
21. A. Pnueli, J. Xu, and L. D. Zuck. Liveness with (0, 1, infity)-counter abstraction. In *CAV*, volume 2404 of *LNCS*, pages 107–122. Springer, 2002.
22. A. Podelski and A. Rybalchenko. ARMC: the logical choice for software model checking with abstraction refinement. In *PADL*, 2007.
23. A. Podelski, A. Rybalchenko, and T. Wies. Heap assumptions on demand. In *CAV*, volume 5123 of *LNCS*, pages 314–327. Springer, 2008.
24. P. Rümmer. A constraint sequent calculus for first-order logic with linear integer arithmetic. In *LPAR*, volume 5330 of *LNCS*, pages 274–289. Springer, 2008.
25. R. Treiber. *Systems programming: Coping with parallelism*. International Business Machines Incorporated, Thomas J. Watson Research Center, 1986.
26. T. Wies, D. Zufferey, and T. A. Henzinger. Forward analysis of depth-bounded processes. In *FOSSACS*, pages 94–108, 2010.
27. D. Zufferey, T. Wies, and T. A. Henzinger. Ideal abstractions for well-structured transition systems. In *VMCAI*, pages 445–460, 2012.

## A Details of Soundness Argument

Let  $s = (\ell_{\widehat{G}}, \eta)$  and  $s' = (\ell_{\widehat{H}}, \eta')$  be states in  $S$ . Define  $(\widehat{G}, \eta) \rightsquigarrow (\widehat{H}, \eta')$  iff  $\widehat{G} \rightsquigarrow \widehat{H}$  under  $h$ ,  $\eta(v) = \sum_{w \in h^{-1}(v)} \eta'(w)$  and  $\eta'(w) = 1$  if  $w$  is a level-0 vertex. Let  $\pi(\widehat{G})$  denote the concrete subgraph induced by the level-0 vertices. Formally, if  $\widehat{G} = (V, E, l)$  then the projection,  $\pi(\widehat{G})$  is the concrete graph induced by vertices  $\{v \in V \mid l(v) = 0\}$ . The concretization function  $\gamma$  can be generalized to states of  $\mathcal{T}(\mathcal{N})$  as  $\gamma(s) = \downarrow\{\pi(H) \mid (\widehat{G}, \eta) \rightsquigarrow^* (\widehat{H}, \eta')\}$ .

Define  $s \sqsubseteq s'$  iff  $\gamma(s) \subseteq \gamma(s')$ . An inclusion mapping,  $h : s \rightarrow s'$  will *in addition* need to respect counters: for all  $v$  in  $V(\widehat{H})$  it holds that  $\sum_{w \in h^{-1}(v)} \eta'(w) \leq \eta(x_v)$ . We have extended the notion with the aim that the inclusion mapping corresponds to  $\sqsubseteq$ . (generalization of Theorem 6).

*Properties of transition system  $\mathcal{T}(\mathcal{N})$ .* We state some important properties of the transitions in the numerical abstraction. Define  $\text{post}_r(s)$  as  $\bigcup_{\delta} \text{post}_{\delta}(s)$  where the union is taken over  $\delta \in \sigma_{\mathcal{R}}^{-1}(r)$ . The following proposition says that nested rewriting is sound with respect to the numerical abstraction, i.e., all states reached after applying a rewrite in the concrete are captured in the nested rewrite:

**Proposition 9.** *For all  $r \in R$ ,  $\gamma(\text{post}_r(s)) = \downarrow\text{post}_r(\gamma(s))$ .*

*Proof (Sketch).* Let  $s = (\ell_{\widehat{G}}, \eta)$  be the state and  $G' \in \gamma(s)$  to which some rewrite rule can be applied. Let  $r : G_L \rightarrow G_R$  be the rule and  $m : G_L \rightarrow G'$  the match, giving us a pushout  $r' : G' \rightarrow H'$  and  $m' : G_R \rightarrow H'$ . Then, we need to show that we indeed have a transition  $s \xrightarrow{\delta} s'$  with  $\sigma(\delta) = r$  and  $G' \in \gamma(s')$ .

Since,  $G' \in \gamma(s)$ , we have an inclusion mapping, say  $\widehat{i} : G' \rightarrow (\widehat{H}, \eta)$ . Then, it is easy to see that the inclusion map  $\widehat{m} = \widehat{i} \circ m$  is an nested match  $\widehat{m} : G_L \rightarrow \widehat{G}$ . By construction, we have actions corresponding to all possible rules and matches. Thus, we are guaranteed to have a transition  $\delta$  corresponding to  $\widehat{m}$  and  $r$  at  $s$  with  $\sigma(\delta) = r$ . All that is left to show is that  $H' \in \gamma(s')$  where  $s' = (\ell_{\widehat{H}}, \eta')$ .

In order to do so, we elaborate on constraints (1)-(2). They can be thought of as composition of two set of constraints – one for unfolding so that the rewrite rule can be applied to level-0 nodes and another for rewrite itself. Recalling notation we used to construct the constraints, let  $K$  be the completely unfolded graph under homomorphism  $h$ ;  $V$ ,  $W$  and  $U$  vertices of  $\widehat{G}$ ,  $\widehat{H}$  and  $K$ ; and by  $V_0$ ,  $W_0$  and  $U_0$  we denote the level-0 nodes of each graph. The rewrite homomorphism itself we denote by  $\widehat{r}'$ . Then, if we imagine having variables  $x_u$  for vertices  $u \in U$  in the intermediate graph, the constraints can be thought of as composition of two set of constraints, (5) for unfolding and (6) for rewrite:

$$x_v = \sum_{u \in h^{-1}(v)} x'_u \quad \wedge \quad x'_u = 1 \text{ for } u \in U_0 \quad (5)$$

$$x'_{\widehat{r}'(u)} = x_u \text{ for } u \notin U_0 \quad \wedge \quad x'_w = 1 \text{ for } w \in W_0 \quad (6)$$

The unfolding constraints can itself be thought of as composition of constraints of individual unfolding steps. For a single unfolding step the constraint simply corresponds

to the definition, enforcing  $\eta(v) = \sum_{w \in h^{-1}(v)} \eta'(w)$ . It follows from the definition of  $\gamma$  that if  $G' \in \gamma((\widehat{G}, \eta))$  then there exists  $\eta''$  such that  $G' \in \gamma((K, \eta''))$ .

The rewrite rule itself affects only the part of the graph at level 0. The first part of the conjunction in (6) ensures all counters for non-level-0 nodes are preserved ( $\widehat{r}'$  is a bijection on these nodes), and the second constraint ensures that all counters for the transformed subgraph are 1. Since the multiplicities are preserved on the untransformed part,  $(\widehat{H}, \eta')$  can be unfolded to get  $H'$  as a subgraph in the same way as  $K$  could be unfolded to get  $G'$  as a subgraph. Together with the observation about unfolding, this shows  $G' \in \gamma(s')$ .  $\square$

The next lemma says that the other set of transitions in  $\mathcal{T}(\mathcal{N})$ , ones that correspond to covering edges, do not lose any concrete states.

**Lemma 10.** *Let  $s \xrightarrow{\delta_e} s'$  for a covering edge  $e$ . Then,  $\gamma(s) \subseteq \gamma(s')$ .*

*Proof (Sketch).* Let  $s = (l_{\widehat{G}}, \eta)$ ,  $s' = (l_{\widehat{H}}, \eta')$ , then the way constraint (4) is constructed we get an inclusion mapping  $h : \widehat{G} \rightarrow \widehat{H}$ . The constraint (4) also respect condition on cardinalities for the generalized notion of inclusion mapping, giving us inclusion mapping  $h : s \rightarrow s'$ . If  $G' \in \gamma(s)$ , then there exists an inclusion mapping  $i : G' \rightarrow (\widehat{G}, \eta)$ . It is easy to check that the composition  $h \circ i : G' \rightarrow s'$  is also an inclusion mapping, showing  $G' \in \gamma(s')$ .  $\square$

The next lemma relates the transitions of the numerical abstraction with the transitions in the original system, both for covering and rewriting edges in the ACT.

**Lemma 11.** *For all states  $s \in S$ , either  $\exists \delta \in \Delta \setminus \text{dom}(\sigma_{\mathcal{R}})$  such that  $s \xrightarrow{\delta} s'$ , or  $\forall r \in R, \gamma(\text{post}_r(s)) = \downarrow \text{post}_r(\gamma(s))$ .*

*Proof.* This lemma follows from the fact that the numerical abstraction is constructed from the bipartite graph  $(\widehat{\mathcal{I}}, \widehat{\mathcal{J}}, \mathcal{E})$  where for nested graph in  $\widehat{\mathcal{I}}$  we have an edge corresponding to all rewrite rules (for which the correspondence between abstract and concrete was established in Proposition 9), and for each nested graph in  $\widehat{\mathcal{J}}$  we have a covering edge.  $\square$

*Stuttering simulation.* Let  $\mathcal{T} = (S, s_0, \text{Act}, \xrightarrow{\text{Act}})$  and  $\mathcal{T}' = (S', s'_0, \text{Act}', \xrightarrow{\text{Act}'})$  be two transition systems. Further, let  $\sigma : \text{Act}' \rightarrow \text{Act}$  be a partial map between the actions of the two systems and  $\leq \subseteq S \times S'$ . Define  $A'_\epsilon = \text{Act}' \setminus \text{dom}(\sigma)$ . Given  $s \leq s'_1$ , define a *stuttering step* with respect to  $(\leq, \sigma)$  as a transition  $s'_1 \xrightarrow{A'_\epsilon} s'_2$  such that  $s \leq s'_2$ . We say  $(\leq, \sigma)$  is a *stuttering simulation* if for all  $s_1, s_2 \in S$ ,  $a \in \text{Act}$  with  $s_1 \xrightarrow{a} s_2$  and all  $s'_1 \in S'$  with  $s_1 \leq s'_1$ , there exist  $a', t', s'_2$  such that  $s'_1 \xrightarrow{A'_\epsilon}^* t'$  is a series of stuttering steps, and  $t' \xrightarrow{a'} s'_2$  with  $\sigma(a') = a$  and  $s_2 \leq s'_2$ . In our case, we have two transition system  $\mathcal{T}(\mathcal{R}) = (\text{Graphs}, G_0, R, \xrightarrow{R})$  and  $\mathcal{T}(\mathcal{N}) = (S, s_0, \Delta, \xrightarrow{\Delta})$ . Define  $\leq_{\mathcal{R}}$  as  $G \leq_{\mathcal{R}} s$  if and only if  $G \in \gamma(s)$  and let  $\sigma_{\mathcal{R}} : \Delta \rightarrow R$  be as before. Define  $\Delta_\epsilon = \Delta \setminus \text{dom}(\sigma_{\mathcal{R}})$ . Observe that the transitions in  $\Delta_\epsilon$  are exactly those corresponding to covering edges.

*Claim.* (1) If  $G_1 \leq_{\mathcal{R}} s_1$  and  $s_1 \xrightarrow{\delta_\epsilon} s_2$  with  $\delta_\epsilon \in \Delta_\epsilon$ , then  $G_1 \leq_{\mathcal{R}} s_2$ . In other words, every transition corresponding to a covering edge is a stuttering step with respect to  $(\leq_{\mathcal{R}}, \sigma_{\mathcal{R}})$ .

*Proof.* This is a straightforward corollary of Lemma 10 and the definition of  $\leq_{\mathcal{R}}$ .  $\square$

*Claim.* (2) Let  $G_1 \leq_{\mathcal{R}} s_1$  and  $G_1 \xrightarrow{r} G_2$ . Further, assume that  $\text{post}_r(\gamma(s_1)) \subseteq \gamma(\text{post}_r(s_1))$ . Then, there exists  $s_2 \in S$  and  $\delta \in \text{dom}(\sigma_{\mathcal{R}})$  such that  $\sigma_{\mathcal{R}}(\delta) = r$ ,  $s_1 \xrightarrow{\delta} s_2$  and  $G_2 \leq_{\mathcal{R}} s_2$ .

*Proof.* Since  $G_1 \in \gamma(s_1)$  and  $G_1 \xrightarrow{r} G_2$  we get that  $G_2 \in \text{post}_r(\gamma(s_1))$ . From the second assumption, we can in fact conclude  $G_2 \in \gamma(\text{post}_r(s_1))$ . In other words, there exists  $\delta \in \text{dom}(\sigma_{\mathcal{R}})$  with  $\sigma_{\mathcal{R}}(\delta) = r$  and  $G_2 \in \gamma(\text{post}_\delta(s_1))$ , from which the claim follows.  $\square$

Theorem 12, which captures the main property of our reduction, follows from Lemma 11 and the two claims above.

**Theorem 12.** *The pair  $(\leq_{\mathcal{R}}, \sigma_{\mathcal{R}})$ , where  $\leq_{\mathcal{R}}$  is defined as  $G \leq_{\mathcal{R}} s$  if and only if  $G \in \gamma(s)$ , is a stuttering simulation for  $\mathcal{T}(\mathcal{R})$  and  $\mathcal{T}(\mathcal{N})$ .*

*Proof (of Theorem 12).* Let  $G_1 \leq_{\mathcal{R}} s_1$  and  $G_1 \xrightarrow{r} G_2$ . Then, according to Lemma 11,  $s_1$  has to be such that we meet the assumptions of one of the two claims. If the second claim, we are done – we get  $s_2$  such that  $s_1 \xrightarrow{\delta} s_2$  with  $\sigma(\delta) = r$  and  $G_2 \leq s_2$ . If not, we can repeatedly apply the first claim getting a series of stuttering steps to a state to which second claim can be applied – showing  $(\leq_{\mathcal{R}}, \sigma_{\mathcal{R}})$  is a stuttering simulation.  $\square$

Theorem 8 follows as a corollary of the above theorem.

## B Fairness constraints in PICASSO

*Strong fairness.* PICASSO and ARMC do not directly support weak-fairness. However, we can encode strong fairness constraints in the structural counter abstraction with *fairness counters*. These counters are either decremented by one or incremented by an arbitrary finite amount in the relevant transitions. For example, let  $t_1$  and  $t_2$  be two transitions and  $(t_1, t_2)$  a Streett fairness condition. The Streett condition tell us that if  $t_1$  occurs infinitely often, then  $t_2$  must also occur infinitely often. This can be encoded in the following way. Let  $v$  a fresh variable in the structural counter abstraction. In  $t_1$  we add the constraint  $v' = v - 1$  and in  $t_2$  we add  $v' \geq 0$ . This principle generalizes to Rabin and (co-)Büchi fairness conditions.

Furthermore, we have extended PICASSO to support such fairness constraints, expressed in the input graph rewriting rules. The graph rewriting operation corresponding  $v' \geq 0$  cannot be expressed by the formalism presented in Section 3. For this purpose we extend the graph rewriting rules to allow nodes of non-zero nesting level on the right-hand-side of rewriting rules. This single modification in the parser was enough to express strong fairness constraints.

*When weak and strong fairness meet.* The main difference between strong and weak fairness is that in addition to transition firing, weak fairness also considers whether a transition is enabled. This cannot currently be expressed in PICASSO. Fortunately, the weak-fairness condition that we use are also expressible as strong fairness condition. The key insight is that we can statically know when a transition is enabled. For instance, the transition that spawns client is always enabled until the “Spawn to Nwaps” transition fires. We can rephrase this condition as a co-Büchi condition saying that the spawning of client does not occur infinitely often. For the experimental evaluation we use the co-Büchi condition.