Verifying Programs by Bounded Tree-Width Behavior Graphs

Omar Inverso[1], Salvatore La Torre[2], Gennaro Parlato[3(✉)], and Ermenegildo Tomasco[1,2,3]

[1] Gran Sasso Science Institute, L'Aquila, Italy
[2] Università degli Studi di Salerno, Fisciano, Italy
[3] Università degli Studi del Molise, Pesche, Italy
gennaro.parlato@unimol.it

Abstract. We present a novel framework to reason about programs based on encodings of computations as graphs. The main insight here is to rearrange the programs such that given a bound k, each computation can be explored according to any tree decomposition of width k of the corresponding behaviour graph. This produces under-approximations parameterized on k, which result in a complete method when we restrict to classes of behaviour graphs of bounded tree-width. As an additional feature, the transformation of the input program can be targeted to existing tools for the analysis. Thus, off-the-shelf tools based on fixed-point, or capable of analyzing sequential programs with scalar variables and nondeterminism, can be used. To illustrate our approach, we develop this framework for sequential programs and discuss how to extend it to handle concurrency. For the case of sequential programs, we develop a compositional approach to generate on-the-fly tree decompositions of nested words, which is based on graph-summaries.

1 Introduction

Program computations are typically described as runs of flat transition systems with possibly infinitely many states. The basic information stored in a state is the current control location and the valuation of the statically allocated variables. Depending on the class of programs, a state can also store heap structures, the call stack and in general, additional *data structures* to handle concurrency (multiple call stacks, FIFO channels, etc.).

Computations can also be represented as graphs (*behaviour graphs*) where the nodes capture the basic (finite) information, and different types of edges are used to capture the transitions and the relations deriving from the use of the additional data structures (see [17]). For example, a stack can be captured by

This work was partially supported by INDAM-GNCS 2022 and 2023, AWS 2021 Amazon Research Awards, the MUR project 'Innovation, digitalisation and sustainability for the diffused economy in Central Italy', Spoke 1 MEGHALITIC, VITALITY Ecosystem, and FARB 2021–2023 grants Università degli Studi di Salerno.

V. Malvone and A. Murano (Eds.): EUMAS 2023, LNAI 14282, pp. 116–132, 2023.
https://doi.org/10.1007/978-3-031-43264-4_8

linking the pair of states corresponding to a push and a matching pop, a queue by linking an enqueue to a matching dequeue, and so on.

Several classes of behavior graphs can be defined depending on the aspects and the granularity of the information we wish to capture. Nested words naturally capture the control-flow structure of sequential programs [1], multiply nested words that of shared-memory multi-threaded programs and stack-queue graphs that of both distributed programs with recursive calls and sequential programs with queues and stacks [17], and more definitions are possible.

A very general result on the decidability of problems on classes of behaviour graphs is the decidability of MSO for all MSO-definable classes of graphs of bounded tree-width [5,6,17], which generalizes Courcelle/Seese's theorem [3,19]. For interesting classes of programs, many decidability results of relevant decision problems in verification, such as reachability, model-checking and decidability of linear temporal logics, are indeed subsumed by this general result.

A class of graphs has tree-width k if for each graph there is a tree decomposition of *width* at most $k+1$, that is, the graph can be rearranged on a tree by assigning to each node a set of at most $k+1$ graph vertices (*bag*) such that all vertices and edges are covered and vertices replicated in two nodes also belong to all the bags on the path connecting them. Essentially, for a behaviour graph G, a tree decomposition T of width k ensures that we can execute the computation described by G by checking the consistency of the information at each vertex (i.e., its program counter and variable valuation) locally to a node and its neighbors: in fact, to check the consistency across the edges of G, it is sufficient to consider one bag at a time, and to ensure that each vertex has the same information in any bag, it is sufficient to compare for each node the bags at the node and at its children.

This way of looking at the tree decompositions is the crucial intuition of the approach we present in this paper. We design a general framework for analyzing programs where given a parameter k, we transform an input program P such that the resulting encoding P' interprets the behaviours of P as described above, according to all the tree decompositions of width k of P behaviour graphs, and then P' is analyzed in a separate tool.

Our approach gives a novel and natural way of representing and analyzing systems and has several other features. First, each tree decomposition rearranges the transitions of a computation and gets a way to explore them in a totally independent order, and thus our approach is likely to be less sensitive to "pathological patterns". Second, the width of a tree decomposition gives a natural parameter for bounding the additional storage needed to explore the program computations, and thus we get under-approximation methods of adjustable precision for arbitrary classes of systems. Third, and probably more importantly, we get a general way to encode unbounded heap structures captured in configurations into a *thick* tree where we need to associate a fixed amount of data stored at each node. The tree encodings of computation graphs will allow us to encode complex features of programs (recursion, concurrency, heap, etc.) in a uniform way where we can exploit off-the-shelf solutions that compute the fixed point of

finite relations or can check sequential programs with only scalar variables and non-determinism, to analyze these otherwise complex systems.

We develop our framework for sequential programs with recursive procedure calls and use as behaviour graphs the nested words augmented with the program counters and the variable valuations (*program nested words*). A crucial aspect is to find an efficient way to generate tree decompositions for this class of graphs. We introduce the concept of *shape of a program nested word* (pnw-shape) to summarize portions of program nested words. Namely, a pnw-shape is either a fragment of a nested word (*ground pnw-shape*), or a *merge* of two "compatible" pnw-shapes, or a *contraction* of a pnw-shape on a set of vertices. By compatible we essentially mean that the pnw-shapes can share nodes but edges do not overlap, and the contraction has the effect of keeping only the vertices in the contraction set and projecting on them the edges of the starting pnw-shape. Essentially, in the construction of a tree decomposition, we use these pnw-shapes to summarize the information of the portion of the nested word covered by the nodes of a subtree. In particular, we start from the leaves labeled each with a ground pnw-shape. At each internal node v, we add a ground pnw-shape which is compatible with the pnw-shape of the children of v, and compute the pnw-shape of v by first merging these three pnw-shapes and then contracting the resulting pnw-shape on the vertices of the ground one (which form the bag of v). We also provide a test for the root to ensure that the constructed tree is indeed a tree decomposition.

We outline a simple implementation of our approach as a code-to-code translation of C-like sequential programs that do not make use of dynamic memory allocation. The output of the transformation is a program that nondeterministically builds a tree decomposition using recursive procedure calls as in a DFS traversal of the tree. The ground program pnw-shapes are nondeterministically guessed, and the consistency of program counters and variable valuations associated to each vertex is checked for each edge (according to the semantics of the input program). Moreover, there are procedures to implement the tests, and the merging and contraction operations.

As a further contribution, we give an informal though detailed description on how this approach can be extended to handle concurrent shared-memory programs and how this relates to the sequentialization algorithms (see [2,4,9,16,18] for a sample research). We believe that our approach can be extended to many other classes of programs and systems, such as concurrent programs with a weak memory model assumption, distributed programs, and programs with dynamic data-structures, to mention some.

2 Programs with Recursive Procedure Calls

We consider sequential programs with possibly recursive procedure calls. For the sake of simplicity and without loss of generality, we omit local variables and procedure parameters (in a procedure call, when needed, the values are passed through the global variables). Since we only admit global variables, henceforth we refer to them simply as variables.

```
Var x, y;

procedure main begin
  0: assume(x=1 || x=2);
  1: call boo;
  2: return;
end

procedure boo begin
  3: y := x;
  4: call foo;
  5: assert(x=1);
  6: call foo;
  7: return;
end

procedure foo begin
  8: if (y > 0) then
  9:     y := y − 1;
  A:     call foo;
  B: else skip;    fi
  C: return;
end
```

Fig. 1. A sample program.

In the rest of the paper, we use program P of Fig. 1 as a running example. P is a simple program with three possible behaviours depending on the initial values of the variable x being 1, 2 or an other value. In the last case, the condition of the `assume` statement does not hold and thus the computation immediately halts. In the remaining cases, the procedures *boo* and *foo* get recursively called until the `assert` statement at program counter 5 is reached. Now, a computation with $x = 2$ violates the assertion, and thus reaches an *error* state, while a computation with $x = 1$ continues through the end of the procedure `main`.

Syntax. The BNF grammar on the right gives the formal syntax of programs (Fig. 2). A program starts with the declaration of a finite set of variables *Var* that are visible to all procedures. We assume variables range over some (potentially infinite) data domain $\mathbb{D}$, a language of expressions $\langle expr \rangle$ interpreted over $\mathbb{D}$, and a language of predicates $\langle pred \rangle$ over the variables. Thereafter, there is a declaration of a non empty list of *procedures*, among which one called `main` that is initially executed to start the program. Each procedure is formed by a nonempty sequence of labeled statements of the form $pc : \langle stmt \rangle$ where pc is the *program counter* (or *program location*) and $\langle stmt \rangle$ defines a simple language of C-like statements. We assume that each procedure has `return` as last statement.

$\langle prgm \rangle$ ::= Var; $\langle proc \rangle^+$
$\langle proc \rangle$ ::= `procedure` p `begin` $\langle pc\_stmt \rangle^+$ `end`
$\langle pc\_stmt \rangle$ ::= pc : $\langle stmt \rangle$;
$\langle stmt \rangle$ ::= g := $\langle expr \rangle$ | `skip`
| `assume(`$\langle pred \rangle$`)` | `assert(`$\langle pred \rangle$`)`
| `if` $\langle pred \rangle$ `then` $\langle pc\_stmt \rangle^+$ `else` $\langle pc\_stmt \rangle^+$ `fi`
| `while` $\langle expr \rangle$ `do` $\langle pc\_stmt \rangle$ `do`
| `call` p | `return`

Fig. 2. BNF grammar of programs.

For a program P, we denote with PC (resp., *Call*, *Ret*) the set of all program counters pc such that $pc : stmt$ (resp., $pc :$ `call` p, $pc :$ `return`) is a labeled statement of P. Furthermore, for every $pc \in Call$ we denote with $afterCall(pc)$ the (unique) program counter pc' such that $pc' : stmt$ is the statement that is executed after returning the procedure call with program counter pc.

Semantics. The semantics is given as a transition system. Each program can make procedure calls and manipulate variables. Thus, a state is a *configuration* of the form $\langle \nu, pc, St \rangle$ where ν is a valuation of the variables (i.e., $\nu : Var \mapsto \mathbb{D}$), $pc \in PC$ is a program counter, and St is the content of the call stack (i.e., the control locations of the pending procedure calls). A configuration $C = \langle \nu, pc, St \rangle$ is *initial* if pc is the program counter of the first statement of the procedure `main` and St is the empty stack.

The *transition relation*, denoted $\hookrightarrow$, is defined as usual. The control-flow statements update the program counter, possibly depending on a predicate (condition). The assignment statements update the variable valuation other than moving to the next program counter. At a procedure call, the current location of the caller (*pc*) is pushed onto the stack, and the control moves to the first location of the called procedure. At a return statement the control location at the top of the stack is popped, say *pc*, and the control moves to location *afterCall*(*pc*).

A *computation* of a program is a sequence of configurations $C_0C_1 \dots C_n$ such that C_0 is initial, and $C_{i-1} \hookrightarrow C_i$ for every $i \in [1, n]$.

3 Graphs Representing Program Executions

In this section, we recall some definitions on graphs and define the notion of program nested word that we use in the rest of the paper to represent the executions of a program.

Multigraphs. A multigraph is a structure $G = (V, E_1, \dots, E_n)$, where V is a finite set of vertices, and for each i, $E_i \subseteq V \times V$ is a set of directed edges. An edge $(u, v) \in E_i$ is also denoted as uE_iv. A multigraph $G = (V, E_i)$ is a *line graph* if there is an ordering of all vertices of G, say $v_0, v_1, \dots, v_m$, such that $E_i = \{v_{j-1}E_iv_j \mid j \in [m]\}$.

Nested Words. A *nested word*[1] is a multigraph $(V, \rightarrow, \curvearrowright)$ where $(V, \rightarrow)$ is a line graph and $\curvearrowright$ is a matching edge relation such that for every $u, v, u', v' \in V$:

- if $u \curvearrowright v$ then $u \rightarrow^+ v$;
- if $u \curvearrowright v$, $u' \curvearrowright v'$ are distinct edges then (1) u, v, u', v' are all distinct nodes, and (2) if $u \rightarrow^+ u'$ then either $v \rightarrow^+ u'$ or $v' \rightarrow^+ v$.

Program Nested Words. We wish to look at the computations of a program via their behaviour graphs, i.e., finite graphs that carefully model with their edges the control-flow structure. In particular, we use as behaviour graphs the nested words annotated with the program counter (*pc*, for short) and the valuation of the variables of each *state*. We refer to such annotated nested words as *program nested words*. In Fig. 3, we give the behaviour graph of a computation of the program from Fig. 1 when $x = 1$ holds. The vertices of the nested word $v_0, \dots, v_{17}$ are labeled with the corresponding *pc* in the program. Also, in the figure, we report the variable valuation at the beginning and update it at each node after an assignment. Moreover, the $\curvearrowright$-edges are represented as curved arrows and the $\rightarrow$-edges are represented as straight arrows. The $\rightarrow$-edges capture the linear ordering of the program states in the computation (and thus the *transitions* of the computation). The $\curvearrowright$-edges match the vertices corresponding to the *pc* of a call to a procedure to the *pc* of the next statement of the procedure that will be executed after returning the call (*return location*). Consider for example

[1] We assume that there are no unmatched calls and returns, differently from [1].

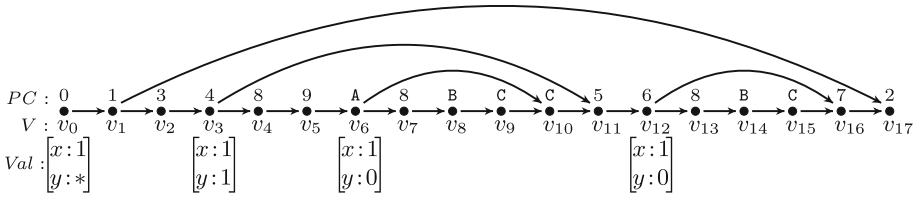

Fig. 3. Program nested word of a run of the program in Fig. 1.

$v_1 \curvearrowright v_{17}$, v_1 corresponds to the state that precedes the call to *boo* from `main` (with *pc* 1) and v_{17} corresponds to the state after returning from this call (with *pc* 2); also we have $v_1 \rightarrow v_2$, and v_2 corresponds to the begin of the first activation of *boo* (with *pc* 3).

Below, we give a logical characterization of program nested words. For the ease of presentation we assume that all the procedure calls in the computations are returned. Note that this is without loss of generality, since we can always append a possibly empty sequence of additional transitions (which are not actual program transitions and thus can be recognized as such) to match all pending calls in the call stack.

Definition 1 (Program Nested Word)**.** *A* program nested word *of a program P with set of variables Var and set of program counters PC, is a tuple $(nw, Val, \overline{pc})$ where*

- *$nw = (V, \rightarrow, \curvearrowright)$ is a nested word; let $V = \{v_0, v_1, \ldots, v_n\}$ such that $v_{i-1} \rightarrow v_i$ for $i \in [1, n]$;*
- *Val and $\overline{pc}$ are labeling functions that map each vertex of V respectively with a valuation of Var and a program counter in PC such that $\overline{pc}(v_0)$ is the program counter of the first statement of procedure* `main` *and for $u, v, z \in V$:*
  - *if $u \rightarrow v$ then $\langle Val(u), \overline{pc}(u), st\rangle \hookrightarrow \langle Val(v), \overline{pc}(v), st'\rangle$, for some st, st';*
  - *if $u \curvearrowright v$ then $\overline{pc}(u) \in Call$, $\overline{pc}(v) = afterCall(\overline{pc}(u))$, and $\overline{pc}(z) \in Ret$ where $z \rightarrow v$;*
  - *if $\overline{pc}(u) \in Call$, then $u \curvearrowright v$ exists;*
  - *if $u \rightarrow v$ and $\overline{pc}(u) \in Ret$, then $z \curvearrowright v$ exists.* □

From Executions to Program Nested Words and Back: Let $\pi = C_0 C_1 \ldots C_n$ be a computation of P, where $C_i = \langle \nu_i, pc_i, St_i\rangle$ for $i \in [0, n]$. For each $i \in [0, n]$ with $pc_i \in Call$, we say that i *matches* j if j is the smallest index such that $j > i$ and $St_j = St_i$. We denote with $NW(\pi)$ the tuple $(nw, Val, \overline{pc})$ where $nw = (\{v_0, \ldots, v_n\}, \rightarrow, \curvearrowright)$ is a nested word such that (1) $v_{i-1} \rightarrow v_i$ for $i \in [1, n]$, (2) $v_i \curvearrowright v_j$ iff i matches j in π, and (3) $Val(v_i) = \nu_i$ and $\overline{pc}(v_i) = pc_i$, for $i \in [0, n]$. We can show that $NW(\pi)$ is indeed a program nested word of P.

Vice-versa, consider a program nested word $pnw = (nw, Val, \overline{pc})$ of P and let $\{v_0, \ldots, v_n\}$ be the set of vertices of nw such that $v_{i-1} \rightarrow v_i$ for $i \in [1, n]$. We denote with $RUN(pnw)$ the sequence of configurations $C_0 C_1 \ldots C_n$ where denoting $C_i = \langle Val(v_i), \overline{pc}(v_i), St_i\rangle$, St_0 is the empty stack and for $i \in [1, n]$:

(1) if $\overline{pc}(v_{i-1}) \in Call$ then $St_i = \overline{pc}(v_{i-1}).St_{i-1}$ (procedure call), (2) if $v_j \curvearrowright v_i$ then $St_{i-1} = \overline{pc}(v_j).St_i$ (return from a call), (3) otherwise $St_i = St_{i-1}$ (internal move). We can show that $RUN(\pi)$ is indeed a computation of P.

Thus the following holds:

Theorem 1. *Given a program P there is a one-to-one mapping (modulo a vertex renaming) between the computations and the program nested words of P.*

4 Bounded Tree-Width Analysis of Programs

The main intuition behind our methodology is to use the tree decompositions of the behaviour graphs of a program, to guide the exploration of its computations.

Informally, a *tree decomposition* of a graph G is a binary tree whose nodes are labeled with sets of G vertices, which are called *bags*, such that every edge or vertex of G is covered by at least one bag, and if a vertex v belongs to two bags labeling two different nodes then all the bags on the unique path connecting such nodes also contain v. Figure 4(a) gives a tree decomposition of the program nested word of Fig. 3 where each bag is implicitly defined by the vertices of the graph that labels the node. A formal definition of tree decomposition is given at the end of this section.

We illustrate the role played by tree decompositions in our methodology on the sample program nested word of Fig. 3 and with respect to the above mentioned tree decomposition.

We augment the tree decompositions by adding to each node some edges of the graph such that each edge is mapped exactly to a node whose bag contains both of its endpoints (note that such a labeling is always possible since by definition each edge is covered by at least one bag). The tree decomposition of Fig. 4(a) is augmented in such a way.

Now, by assuming that we have an augmented tree decomposition for a program nested word of a program P (we will discuss in the next section how to generate efficiently such tree decompositions), we check that a labeling of each vertex in each bag with a program counter and a variable valuation of P forms a computation of P: that is, we can reconstruct a program nested word of P from the additional labeling and the tree decomposition.

Starting from the leaves, we locally check the consistency of the transitions captured by the edges in the bag. For example, in node n_5, we can check for the consistency of the program counters and variable evaluations of the vertices $v_6, v_7, v_8, v_9, v_{10}, v_{11}$ according to (1) the transitions of P corresponding to the $\rightarrow$-edges (v_6, v_7), (v_7, v_8), (v_9, v_{10}) and (v_{10}, v_{11}) and (2) the $\curvearrowright$-edge (v_6, v_{10}). Then, moving up to the parent of n_5, i.e., n_2, we do not need to keep the information for v_7, v_8, v_9, v_{10}, since all the edges involving them have already been examined (this is carefully captured by the tree decomposition that does not contain these vertices in the bag of n_2). However, we need to check that the program counter and the variable valuation associated with the vertices that are kept, i.e., v_6 and v_{11}, are the same as in n_2 and n_5.

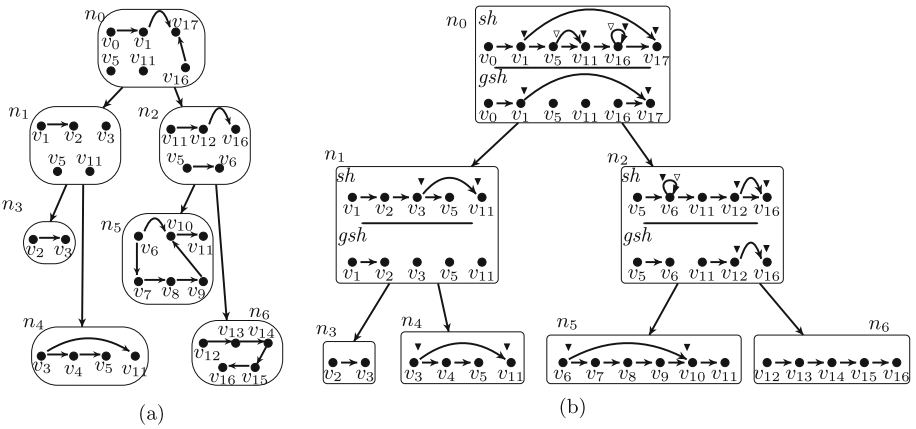

Fig. 4. Example of a tree decomposition (a) and an nw-shape tree (b) for the program nested word of Fig. 3.

In this use of tree decompositions, a bag associated with a node n is the *interface*, or the sticking vertices, of the portion of nested word corresponding to the subtree rooted at n with the rest of the computation. Thus, if we restrict to tree decompositions with bags of size bounded by a parameter k (the width), at each node we only need to track $O(k)$ information. By choosing $k \geq 3$, for the class of programs that we have defined, we can explore the whole set of computations of a program, as stated by Theorem 2 at the end of this section.

Tree Decompositions and Tree-Width. A *tree-decomposition* of a multigraph $G = (V, E_1, \ldots, E_n)$ is a pair $D = (T, \{bag_t\}_{t \in N})$, where T is a binary tree with set of nodes N, and $bag_t \subseteq V$ satisfying the following:

- for every $v \in V$, there is a node $t \in N$ such that $v \in bag_t$;
- for every i and $(u, v) \in E_i$, there is a node $t \in N$ such that $u, v \in bag_t$;
- if $u \in (bag_t \cap bag_{t'})$ then $u \in bag_{t''}$ for every T node t'' that lies on the unique path connecting t to t' in T.

The *width* of a tree decomposition $(T, \{bag_t\}_{t \in N})$ is the size of the largest bag in it, minus one; i.e. $max_{t \in N}\{|bag_n|\} - 1$. The *tree-width* of a graph is the *smallest* of the widths overall its tree decompositions.

Theorem 2 ([17])**.** *Any nested word has tree-width at most* 2.

5 Getting Tree Decompositions for Program Nested Words

In the description of our approach from the previous section, we assume that a tree-decomposition of a program nested word is given. Indeed, we compute such decompositions on the fly. For this, we need to carry in our *summaries* (at each

node) some structural information about the portion of the nested word so far explored, such that we can correctly check portions of nested words separately, then combine them in the same structure, and in the end claim that we have constructed a valid tree decomposition for a computation of the program.

The Informal Scenario. Our notion of *summary* for nested words is a shape of a nested word, *nw-shape* for short. Informally, an nw-shape is either a fragment of a nested word (*ground nw-shape*), or a *merge* of two "compatible" nw-shapes, or a *contraction* of an nw-shape on a set of vertices. By compatible, we essentially mean that the nw-shapes can share nodes but the edges do not overlap. For example, each of the graphs labeling the nodes of the tree in Fig. 4(b) is an nw-shape.

In addition to the notation used for nested words, we also use the symbols ▼ and ▽ to annotate respectively that an end-point of a $\curvearrowright$-edge is the actual one or it replaces one that has been abstracted away. In particular, for a ground nw-shape all the endpoints of a $\curvearrowright$-edge are marked with ▼.

To generate a tree decomposition, we construct an nw-shape tree. In Fig. 4, we give an nw-shape tree for the nested word from Fig. 3. We start from the leaves that are assigned each with a ground nw-shape. For each internal node n, we add a ground nw-shape (marked with *gsh* below the line in each node) and compute a summary of the ground nw-shapes in the subtree rooted at n (marked with *sh* above the line). The summary is computed by merging the summaries at the children and the ground nw-shape of the current node, and then contracting the resulting nw-shape on the vertices of the ground nw-shape (note that at each node the nw-shape and the ground nw-shape have the same set of vertices).

For example, consider node n_1. The nw-shapes from nodes n_3 and n_4 just share the vertex v_3 and therefore can be merged (in fact they are compatible because when glued to v_3 there are no overlaps between the edges). Similarly, the resulting nw-shape can be merged with the ground nw-shape of n_1, and the resulting nw-shape has vertices v_1, v_2, v_3, v_4, v_5 and v_{11}. In the contraction, only v_4 gets abstracted away. The effect of the contraction is thus to remove v_4 and connect v_3 to v_5 to store the information that has already been explored in the space between v_3 and v_5.

The contraction is slightly more intricate when an endpoint of a $\curvearrowright$-edge is abstracted away. In fact, in this case, the new endpoint (if any) is selected, among those that have not been removed, as the closest one which is included in the portion of the graph below the edge. In particular, the $\curvearrowright$-edge from v_3 to v_{11} in n_1 is replaced (in the contraction) by the $\curvearrowright$-edge from v_5 to v_{11} in n_0, and since v_5 is not the actual left endpoint of this edge, we annotate the left end of $v_5 \curvearrowright v_{11}$ with ▽. Also observe that in case there is no such vertex (both the endpoints are abstracted away and no vertices in the between are kept in the new set of vertices), the edge is canceled. This is in fact the case for the self-loop on v_6 in node n_2, which does not appear in the nw-shape of n_0.

There are two more conditions to ensure to obtain an nw-shapetree. First, the vertices of a ground shapein an internal node must contain all the vertices at the "borders" of the maximal lines defined by $\rightarrow$-edges in the nw-shapes of its

children and the endpoints of the $\curvearrowright$-edges that have not yet been added to the shapes of the subtree (see, for example, vertex v_{11} in n_2). Second, the nw-shape of the root must be entirely connected through the $\rightarrow$-edges (*linearly connected*) and should have all the endpoints of $\curvearrowright$ matched (*fully matched*).

An augmented tree decomposition is easily obtained from an nw-shape tree by retaining for each node just its ground shape. Since vice versa also holds, i.e., for each augmented tree decomposition there is a corresponding nw-shape tree, our method captures all the augmented tree decompositions of a program. Moreover, it can be implemented on the fly, with all operations being local.

Nested Word Shapes. Let $T = \{\triangledown, \blacktriangledown\}$ be an alphabet, where the symbol $\triangledown$ stands for *abstract*, and $\blacktriangledown$ stands for *concrete*.

Definition 2 (Nested Word Shapes)**.** *A* nested word shape *(nw-shape) is a tuple* $S = (V, \Rightarrow, \rightarrow, \{\overset{t,z}{\curvearrowright}\}_{t,z \in T})$ *where*

- V *is a finite set of vertices;*
- $(V, \Rightarrow)$ *is a line graph;*
- *the set of* linear edges $\rightarrow$ *is a subset of* $\Rightarrow$*;*
- *the set of the* matching edges $\curvearrowright = (\bigcup_{t,z \in T} \overset{t,z}{\curvearrowright})$ *where* $\overset{t,z}{\curvearrowright} \subseteq V \times V$ *and is such that (where* $a, b, c, d, \in T$ *and* $u, v, x, y \in V$*):*
  - *if* $u \curvearrowright v$ *then also* $u \Rightarrow^* v$*;*
  - *if* $u \curvearrowright v$ *and* $x \curvearrowright y$*, then the following does not hold:*
    - $u \Rightarrow^+ x \Rightarrow^+ v \Rightarrow^+ y$ *(matching edges do not cross);*
    - $(u, v) \neq (x, y)$ *and* $v = x$ *(call and return of distinct matching edges must not coincide);*
  - $u \overset{\blacktriangledown,\blacktriangledown}{\curvearrowright} u$ *does not hold;*
  - *at most one among* $u \overset{\blacktriangledown,\triangledown}{\curvearrowright} v$, $u \overset{\triangledown,\blacktriangledown}{\curvearrowright} v$ *and* $u \overset{\blacktriangledown,\blacktriangledown}{\curvearrowright} v$ *holds;*
  - *if* $u \overset{a,b}{\curvearrowright} v$, $u \overset{c,d}{\curvearrowright} y$ *and* $y \Rightarrow^+ v$ *then* $a = \triangledown$*;*
  - *if* $u \overset{a,b}{\curvearrowright} v$, $x \overset{c,d}{\curvearrowright} v$ *and* $u \Rightarrow^+ x$ *then* $b = \triangledown$*.*

S *is* ground *if all of its matching edges are concrete, that is,* $\curvearrowright$ *is exactly* $\overset{\blacktriangledown,\blacktriangledown}{\curvearrowright}$*.* S *is* linearly connected *if* $\rightarrow$ *is exactly* $\Rightarrow$*.* □

A *linear border* of a shape S is a vertex $u \in V$ without a linear successor or a linear predecessor, i.e., either $u \not\rightarrow v$ for each $v \in V$ or $v \not\rightarrow u$ for each $v \in V$.

Operations on Shapes. In the following, we fix $S = (V, \Rightarrow, \rightarrow, \{\overset{t,z}{\curvearrowright}\}_{t,z \in T})$, $S' = (V', \Rightarrow', \rightarrow', \{\overset{t,z}{\curvearrowright'}\}_{t,z \in T})$, and $S_i = (V_i, \Rightarrow_i, \rightarrow_i, \{\overset{t,z}{\curvearrowright_i}\}_{t,z \in T})$ for $i = 1, 2$.

An nw-shape S' is the *contraction* of an nw-shape S on a set of vertices $V' \subseteq V$, denoted $S' = \mathit{contraction}(S, V')$, if the following holds:

- $\Rightarrow'^*$ is the total order on V' induced by $\Rightarrow^*$;
- $\rightarrow'$ is the set of all pairs $(x, y) \in V' \times V'$ such that either (1) $x \rightarrow y$, or (2) there is a sequence of vertices $u_1, u_2, \ldots, u_m \in (V \setminus V')$ such that $x \rightarrow u_1 \rightarrow u_2 \rightarrow \ldots \rightarrow u_m \rightarrow y$.

– for each matching edge (u, v) of S, denote with $\mathit{contraction}_{\curvearrowright}(u, v)$ the pair (x, y) if the following holds:
  - $u \Rightarrow^* x \Rightarrow^* y \Rightarrow^* v$;
  - x is the smallest vertex $x' \in V'$ with $u \Rightarrow^* x'$; and
  - y is the greatest vertex $y' \in V'$ with $y' \Rightarrow^* v$.

  For every $t, z \in T$, $\overset{t,z}{\curvearrowright}'$ is the minimal set containing all pairs (x, y) such that there exist $u, v \in V$ where (1) $u \overset{p,s}{\curvearrowright} v$ for some p, s, (2) $\mathit{contraction}_{\curvearrowright}(u, v) = (x, y)$, (3) $t = \blacktriangledown$ iff $u = x$ and $p = \blacktriangledown$, and (4) $z = \blacktriangledown$ iff $v = y$ and $s = \blacktriangledown$.

S is the *merge* of two nw-shapes S_1 and S_2, denoted $S = \mathit{merge}(S_1, S_2)$, if S is an nw-shape and the following holds:

– $V = V_1 \cup V_2$;
– $\Rightarrow_1, \Rightarrow_2 \subseteq \Rightarrow$;
– $(\rightarrow_1 \cap \rightarrow_2) = \emptyset$, and $\rightarrow = (\rightarrow_1 \cup \rightarrow_2)$;
– For every $t, z \in T$, $\overset{t,z}{\curvearrowright} = (\overset{t,z}{\curvearrowright}_1 \cup \overset{t,z}{\curvearrowright}_2)$.

Tree Decompositions via nw-Shapes. In an nw-shape tree $\mathcal{T}$, the vertices of the nw-shape are typed as either left ($\mathcal{L}$) or right ($\mathcal{R}$) endpoint of a matching edge, or none of them, with the meaning that $u \in \mathcal{L}$, resp. $v \in \mathcal{R}$, iff there is a ground nw-shape of $\mathcal{T}$ that contains an edge $u \overset{\blacktriangledown,\blacktriangledown}{\curvearrowright} v$.

Definition 3 (NW-SHAPE TREE)**.** *An* nw-shape tree $\mathcal{T}$ *is a triple* (T, sh, gsh) *where* T *is a binary tree with set of nodes* N*, and* sh *and* gsh *label each node of* T *with respectively an nw-shape and a ground nw-shape such that for each* $n \in N$ *the following holds:*

– *if* n *is a leaf, then* $sh(n) = gsh(n)$*;*
– *if* n *is an internal node, denoting with* n_1 *and* n_2 *its left and right children and with* V_n *the set of the vertices of* $gsh(n)$*:*
  - $sh(n) = \mathit{contraction}(S(n), V_n)$ *where* $S(n)$ *is the merge of* $sh(n_1)$*,* $sh(n_2)$ *and* $gsh(n)$*;*
  - *denoting with* $\mathcal{LR}$ *the set of all vertices from* $\mathcal{L} \cup \mathcal{R}$ *that are not concrete endpoints of* $\curvearrowright$*-edges of* $sh(n_1)$ *and* $sh(n_2)$*,* V_n *contains* $\mathcal{LR}$ *and all the linear borders of* $sh(n_1)$ *and* $sh(n_2)$*;*
  - *if* n *is the root, then additionally* $sh(n)$ *is also linearly connected and* fully matched*, that is, all its vertices from* $\mathcal{L} \cup \mathcal{R}$ *are concrete endpoints of some* $\curvearrowright$*-edge.*

We denote with $G(\mathcal{T})$ *the graph* $\bigcup_{n \in N} gsh(n)$*.* □

Note that in the above definition for each $n \in N$, $gsh(n)$ and $sh(n)$ have the same vertices and each edge of $gsh(n)$ is also an edge of $sh(n)$. By structural induction, it is possible to show that the graph obtained by the union of the ground nw-shapes of the leaves of a subtree is a ground nw-shape corresponding to a fragment of a nested word. When the nw-shape associated with the root of the subtree is also linearly connected and fully matched, then the resulting ground nw-shape corresponds to a nested word.

Lemma 1. *For any nw-shape tree $\mathcal{T}$, $G(\mathcal{T})$ is a nested word.*

For a nested word w denote with $\bullet\text{-}w\text{-}\bullet$ the nested word obtained from w by adding two new vertices v_L and v_R along with the edges $v_L \to v$ and $v' \to v_R$ where v denotes the first vertex and v' the last vertex of w according to $\to^*$.

From an nw-shape tree $\mathcal{T} = (T, sh, gsh)$ such that $G(\mathcal{T}) = \bullet\text{-}w\text{-}\bullet$, define $D(\mathcal{T}) = (T, \{V_n\}_{n\in N})$ where N is the set of nodes of T and V_n is the set of all the vertices of $sh(n)$ but v_L and v_R. By the definitions of nw-shape tree and tree decomposition, we get that $D(\mathcal{T})$ is a tree decomposition of w.

Vice versa, consider a tree decomposition $D = (T, \{B_n\}_{n\in N})$ of a nested word $w = (V, \to, \curvearrowright)$. Add to each bag B_n the vertices v_L and v_R and define:

- $\{\to_n\}_{n\in N}$ such that each edge $u \to v$ of w, belongs to exactly one $\to_n$ such that $u, v \in B_n$, and each of $v_L \to v$ and $v' \to v_R$ belongs to exactly one $\to_n$;
- $\{\curvearrowright_n\}_{n\in N}$ such that for each edge $u \curvearrowright v$ of w, $u \overset{\blacktriangledown,\blacktriangledown}{\curvearrowright} v$ belongs to exactly one $\curvearrowright_n$ such that $u, v \in B_n$;
- each $\Rightarrow_n$ is the total order on B_n induced by $\to^*$;
- $gsh(n) = (B_n, \Rightarrow_n, \to_n, \curvearrowright_n)$, $\mathcal{L} = \{u \mid u \curvearrowright v\}$ and $\mathcal{R} = \{v \mid u \curvearrowright v\}$.

Starting from the parents of the leaves of T we compute for each node n, $sh(n)$ as the contraction on V_n of the merge of $gsh(n)$, $sh(n_1)$ and $sh(n_2)$ where n_1 and n_2 are the children of n. By definition, we can show that $\mathcal{T} = (T, sh, gsh)$ is nw-shapeand $G(\mathcal{T}) = \bullet\text{-}w\text{-}\bullet$. Therefore, the following theorem holds.

Theorem 3. *For any nested word w, there exists a tree decomposition D of width k iff there exists an nw-shape tree $\mathcal{T} = (T, sh, gsh)$ such that $\bullet\text{-}w\text{-}\bullet = G(\mathcal{T})$ and each $gsh(n)$ has at most k vertices of w.*

Note that the additional vertices v_L and v_R do not correspond to any state of the program and are not really needed to have the above theorem. In fact, it would be sufficient to modify the definition of nw-shape tree such that a left (respectively right) linear border can be abstracted away as soon as a prefix (resp. a suffix) of the nested word has been explored.

Shapes and Shape Trees for Programs. We augment nw-shape and nw-shape trees with program counters and variable valuations. In particular, we define a *pnw-shape* inductively from portions of program nested words and with merging and contraction operations. The merging requires that a same vertex is labeled with the same program counter and the same variable valuation. Analogously to nw-shape trees, we define the *pnw-shape tree* with respect to the notion of pnw-shape.

For a mapping $f : A \to B$ and $C \subseteq A$, we denote with $f_{|C}$ the restriction of f to C. For mappings $f_i : A_i \to B_i$ $i = 1, 2$, we denote with $f_1 \cup f_2$ the mapping defined as $f_1(x)$ for each $x \in A_1$ and $f_2(x)$ for each $x \in A_2 \setminus A_1$.

Fix a program P. A *ground pnw-shape* $\mathcal{S}$ of P is a triple $(S, Val, \overline{pc})$ such that S is a ground nw-shape and there exists a program nested word $(nw, Val', \overline{pc}')$ of P such that S is a subgraph of nw, $Val = Val'_{|V}$ and $\overline{pc} = \overline{pc}'_{|V}$.

A *pnw-shape* $\mathcal{S}$ of P is either a ground pnw-shape or $\mathcal{S} = (S, Val, \overline{pc})$ is:

- the *contraction* of a pnw-shape, that is, there is pnw-shape $\mathcal{S}' = (S', Val', \overline{pc}')$ and denoting with V the set of vertices of S, $S = contraction(S', V)$, $Val = Val'_{|V}$ to V and $\overline{pc} = \overline{pc}'_{|V}$, or
- the *merging* of two pnw-shapes, that is, there are two pnw-shapes $\mathcal{S}_1 = (S_1, Val_1, \overline{pc}_1)$ and $\mathcal{S}_2 = (S_2, Val_2, \overline{pc}_2)$ for which, denoting with V the intersection of the sets of vertices of S_1 and S_2, $Val_1(v) = Val_2(v)$ and $\overline{pc}_1(v) = \overline{pc}_2(v)$ for every $v \in V$, then $S = merge(S_1, S_2)$, $Val = Val_1 \cup Val_2$ and $\overline{pc} = \overline{pc}_1 \cup \overline{pc}_2$.

Analogously to nw-shape tree, we define the *pnw-shape tree* with respect to the notion of pnw-shape. The definition is the same except that the merging and contraction operations apply to pnw-shape, and thus we omit further details.

6 Implementation

In this section, we briefly illustrate an implementation of the outlined approach, that is targeted to use a verifier of sequential programs (with recursive procedure calls), though also fixed-point translations in the style of [7,8,11] are possible.

The input program P is transformed into a program P' that is essentially composed of the `main` and five more procedures: `contraction(S1,S2)`, `check(S)`, `ShapeTree()`, `merge(S1, S2)` and `CreateGroundShape(k)`. All the procedures except for `ShapeTree()` do not contain recursive calls.

Procedure `main` nondeterministically computes a pnw-shape `S` by calling `ShapeTree` and then calls `check` on it.

Procedure `check` verifies that `S` is indeed a pnw-shape that can label the root of a pnw-shape tree, i.e., it is linearly connected and fully matched. (Observe that fully matched within a program nw-shape-tree refers to all the vertices that are marked with a program counter from *Call* or correspond to the return states after a call.) If this is the case and the last vertex (according to the linear order) corresponds to an error state, then a statement `assert`(0) is reachable (which defines the error state in P').

Procedure `ShapeTree` (Fig. 5) computes the pnw-shape at the nodes of a possible pnw-shape tree. When invoked from `main`, it starts from the root. At each node, it guesses a ground nw-shape `S` by invoking `CreateGroundShape`. Then, nondeterministically, it decides whether the current node is a leaf or is internal. In the first case, `S` is returned, otherwise two recursive calls to `ShapeTree` are done (one for the left child and the other for the right one). The pnw-shapes returned by these calls are meant to label the two children, then according to the definition of pnw-shape tree these are contracted and merged by respectively invoking procedures `contraction` and `merge`, thus obtaining the pnw-shape for the current node, that is returned.

```
shape ShapeTree() {
  shape S;
  S = CreateGroundShape(k);
  if (nondet()) {
    S1 = contraction(ShapeTree(),S);
    S2 = contraction(ShapeTree(),S);
    S  = merge(S, merge(S1,S2));
  }
  return S;
}
```

Fig. 5. Procedure `ShapeTree`.

Observe that `ShapeTree` exactly implements the properties of Definition 3. To minimize memory usage, we employ contraction of the children's nw-shapebefore merging them, eliminating the need for constructing an intermediate nw-shape containing 2`k` nodes.

The procedure `contraction` also ensures that `S2` has as vertices all the linear borders of `S1` along with the vertices corresponding to calls and return states that have not yet been matched with the $\curvearrowright$-edges (which is required by Definition 3). Procedure `CreateGroundShape` nondeterministically generates a ground pnw-shapewith k vertices and for each edge of the nw-shape it ensures that the program counters of its endpoints conform to the meaning of the edge in the program (that is, a transition or the matching of a call and a return state). Furthermore, if an edge represents a transition, it guarantees the consistency of variable values at its endpoints with the transition's semantics.

7 Discussion

In this paper, we have presented a new methodology to perform software analysis. The main idea is to transform the input programs so that the exploration of the computations is guided by the tree decompositions of their behavior graphs. We have developed in detail our methodology for sequential programs with recursive procedure calls and without dynamic data structures.

In this section, we discuss how to extend our approach to concurrent programs and how it relates to sequentialization of concurrent programs. We then conclude with some remarks and future work.

Concurrent Programs. Concurrent programs consist of a finite number of threads where each of them is defined by a sequential program. All threads run in parallel and communicate through a finite number of shared variables according to the sequential consistency memory model (SC). A natural graph encoding for the computations of concurrent programs is the following. The behavior of each thread is modeled with a nested word. Further, the behaviour of the shared memory is represented by a line graph capturing the sequence of memory operations, where each vertex represents a unique read or write operation. Vertices of the nested words are labeled, as usual, with a program counter and a valuation of the global variables, while memory vertices are labeled with a valuation of the shared variables. A vertex u of a nested word that "reads" a shared variable for executing the local transition, it is linked through a *memory edge* to the memory vertex representing that operation. The direction of this edge is reversed if the vertex "writes" to a shared variable. Since each memory vertex u represents exactly one memory operation, u has exactly one memory edge incident on it. Of course, memory edges will never cross w.r.t. temporal events (as we assume SC). Let us call these behavior graphs *concurrent nested words* (*cnw*).

Concurrent nested words admit natural summaries that reflect their composition. A *concurrent nw-shape* (*cnw-shape* for short) is formed by a distinct nw-shape for each component nested word, and an additional *memory-shape* that is a nw-shape without matching edges. Additional care should be taken

for the memory edges to avoid crossing. We have worked out the details of this representation. For example, a *contraction* operation on a cnw-shape can be accomplished by executing a contraction on each component nw-shape and the memory-shape. Furthermore, memory edges are contracted similarly to matching edges of nw-shapes. The *merge* operation is defined exactly as for nw-shapes. By defining *cnw-shape trees* using the same combination of operations seen for nw-shape trees, we can show an equivalent of Theorem 3 for the concurrent setting. In addition, a code-to-code translation for concurrent programs is again possible (it is similar to that described in Sect. 6). An essential point to note is that verification tools designed for sequential programs can be effectively reused for analyzing concurrent programs.

Here we convey the idea that our approach actually leads to a sort of *sequentialization* when applied to concurrent programs and implemented as a code-to-code translation to sequential programs. A sequentialization translates a concurrent program P into a nondeterministic sequential program P' that (under certain assumptions) behaves equivalently [9,16,18]. To make the approach effective, P' should not track the whole state space of the concurrent program, as in the cross product of the thread states. All sequentializations that have been proposed in literature only track one local state at a time and use k copies of the shared variables, for a given parameter k. Under these restrictions, such approaches can only cover a strict subset of computations in which each thread can at most interact k times with the other threads. These features are indeed desirable as we get a parameterized analysis technique that aims at exploiting as much as possible by tuning k for the underlying sequential verification tool. By increasing the parameter k, we can capture more computations, but this of course comes with a cost in terms of computational resources.

Our analysis schema inherits the favorable features of sequentializations while extending its coverage to a broader range of computations for the parameter k. By considering cnw-shapes with at most k nodes, we also track k copies of the variables (either global or shared), but cover all cnw of tree width k vs. existing sequentializations being only able to intercept a very small subset of them. Moreover, a different sequentialization must be designed to capture new classes of behavior (parameterized programs [10], thread creation [2,4], scope bounded [12,14,15], path bounded [13], etc.), while our schema is uniform for all of them.

Future Work. We believe that obtaining scalable solutions for sequential programs based on our approach will pave the way to lift such results to the concurrent settings. On the theoretical side, it would be interesting to study how computations of concurrent programs running under weak memory models can be modeled with behaviour graphs. Similarly, for distributed programs where the communication among threads happens through FIFO channels (see [17] for behaviour graphs of these programs). Further, we believe that our approach could be useful to reason about programs manipulating heaps. The intuition is that concurrent and distributed programs can be seen as sequential programs

that use stacks for recursion and queues to simulate FIFO channels. We thus wonder whether our approach can be lifted to more general data structures.

References

1. Alur, R., Madhusudan, P.: Adding nesting structure to words. In: Ibarra, O.H., Dang, Z. (eds.) DLT 2006. LNCS, vol. 4036, pp. 1–13. Springer, Heidelberg (2006). https://doi.org/10.1007/11779148_1
2. Bouajjani, A., Emmi, M., Parlato, G.: On sequentializing concurrent programs. In: Yahav, E. (ed.) SAS 2011. LNCS, vol. 6887, pp. 129–145. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-23702-7_13
3. Courcelle, B.: The monadic second-order logic of graphs. I. Recognizable sets of finite graphs. Inf. Comput. **85**(1), 12–75 (1990). https://doi.org/10.1016/0890-5401(90)90043-H
4. Emmi, M., Qadeer, S., Rakamaric, Z.: Delay-bounded scheduling. In: Ball, T., Sagiv, M. (eds.) Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, 26–28 January 2011, pp. 411–422. ACM (2011). https://doi.org/10.1145/1926385.1926432
5. Enea, C., Habermehl, P., Inverso, O., Parlato, G.: On the path-width of integer linear programming. In: Peron, A., Piazza, C. (eds.) Proceedings Fifth International Symposium on Games, Automata, Logics and Formal Verification, GandALF 2014. EPTCS, Verona, Italy, 10–12 September 2014, vol. 161, pp. 74–87 (2014). https://doi.org/10.4204/EPTCS.161.9
6. Enea, C., Habermehl, P., Inverso, O., Parlato, G.: On the path-width of integer linear programming. Inf. Comput. **253**, 257–271 (2017). https://doi.org/10.1016/j.ic.2016.07.010
7. Grebenshchikov, S., Lopes, N.P., Popeea, C., Rybalchenko, A.: Synthesizing software verifiers from proof rules. In: Vitek, J., Lin, H., Tip, F. (eds.) ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2012, Beijing, China, 11–16 June 2012, pp. 405–416. ACM (2012). https://doi.org/10.1145/2254064.2254112
8. Hoder, K., Bjørner, N., de Moura, L.: μZ – an efficient engine for fixed points with constraints. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 457–462. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_36
9. La Torre, S., Madhusudan, P., Parlato, G.: Reducing context-bounded concurrent reachability to sequential reachability. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 477–492. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02658-4_36
10. La Torre, S., Madhusudan, P., Parlato, G.: Sequentializing parameterized programs. In: Bauer, S.S., Raclet, J. (eds.) Proceedings Fourth Workshop on Foundations of Interface Technologies, FIT 2012. EPTCS, Tallinn, Estonia, 25th March 2012, vol. 87, pp. 34–47 (2012). https://doi.org/10.4204/EPTCS.87.4
11. La Torre, S., Madhusudan, P., Parlato, G.: Analyzing recursive programs using a fixed-point calculus. In: Hind, M., Diwan, A. (eds.) Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, 15–21 June 2009, pp. 211–222. ACM (2009). https://doi.org/10.1145/1542476.1542500

12. La Torre, S., Napoli, M.: Reachability of multistack pushdown systems with scope-bounded matching relations. In: Katoen, J.-P., König, B. (eds.) CONCUR 2011. LNCS, vol. 6901, pp. 203–218. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-23217-6_14
13. La Torre, S., Napoli, M., Parlato, G.: A unifying approach for multistack pushdown automata. In: Csuhaj-Varjú, E., Dietzfelbinger, M., Ésik, Z. (eds.) MFCS 2014. LNCS, vol. 8634, pp. 377–389. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-44522-8_32
14. La Torre, S., Napoli, M., Parlato, G.: Reachability of scope-bounded multistack pushdown systems. Inf. Comput. **275**, 104588 (2020). https://doi.org/10.1016/j.ic.2020.104588
15. La Torre, S., Parlato, G.: Scope-bounded multistack pushdown systems: fixed-point, sequentialization, and tree-width. In: D'Souza, D., Kavitha, T., Radhakrishnan, J. (eds.) IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2012. LIPIcs, Hyderabad, India, 15–17 December 2012, vol. 18, pp. 173–184. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2012). https://doi.org/10.4230/LIPIcs.FSTTCS.2012.173
16. Lal, A., Reps, T.W.: Reducing concurrent analysis under a context bound to sequential analysis. Formal Methods Syst. Des. **35**(1), 73–97 (2009). https://doi.org/10.1007/s10703-009-0078-9
17. Madhusudan, P., Parlato, G.: The tree width of auxiliary storage. In: Ball, T., Sagiv, M. (eds.) Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, 26–28 January 2011, pp. 283–294. ACM (2011). https://doi.org/10.1145/1926385.1926419
18. Qadeer, S., Wu, D.: KISS: keep it simple and sequential. In: Pugh, W.W., Chambers, C. (eds.) Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation 2004, Washington, DC, USA, 9–11 June 2004, pp. 14–24. ACM (2004). https://doi.org/10.1145/996841.996845
19. Seese, D.: The structure of models of decidable monadic theories of graphs. Ann. Pure Appl. Log. **53**(2), 169–195 (1991). https://doi.org/10.1016/0168-0072(91)90054-P