CHC-Based Verification of Programs Through Graph Decompositions

Marco Faella[1], Giulio Garbi[2,3], Salvatore La Torre[3], Gennaro Parlato[2*]

[1]Dipartimento di Ingegneria Elettrica e delle Tecnologie dell'Informazione, Università degli Studi di Napoli "Federico II", Via Claudio 21, Napoli, 80125, Italy.
[2]Department, Università degli Studi del Molise, C.da Fonte Lappone, Pesche, 86090, Italy.
[3]Dipartimento di Informatica, Università degli Studi di Salerno, Via Giovanni Paolo II 132, Fisciano, 84084, Italy.

*Corresponding author(s). E-mail(s): gennaro.parlato@unimol.it;
Contributing authors: m.faella@unina.it; giulio.garbi@gmail.com;
slatorre@unisa.it;

Abstract

We present a novel methodology for automated program analysis that leverages graph encodings of computations. The crux of our approach lies in restructuring the program behavior graphs through tree decompositions of bounded width. To achieve this, we introduce a notion of labeled multigraph, called *nested-word shape*, that is used as a summary for portions of program behavior graphs. Such multigraphs are used within the construction of a symbolic data-tree automaton (SDTA), a recently introduced notion of automaton designed to accept tree data structures. We use SDTAs to capture the tree decompositions of the program behavior graphs of a given program. Verification of the original program is then accomplished by checking the emptiness of the data-tree language accepted by these SDTAs, which can be effectively reduced to the satisfiability of constrained Horn clauses (CHC). Our approach results in an under-approximate analysis parameterized by the width $\boldsymbol{k}$ of the tree decomposition used for the analysis, and thus provides a complete method for the classes of programs whose behavior graphs have bounded treewidth. We detail our methodology for recursive sequential programs, which enjoy the bounded treewidth property, and subsequently extend it to concurrent programs. Notably, our approach shows promise across

an even broader spectrum of program classes, including distributed systems and concurrent programs operating under weak memory models.

Keywords: Program verification, tree decompositions, treewidth, tree automata

1 Introduction

In this paper, we present a new methodology for automatically verifying properties of programs that encompass a range of complex features in a cohesive manner. Program computations are frequently represented as sequences of transitions in a flat transition system, possibly with an infinite number of states. Each state encapsulates the current program instruction being executed and the values of all statically allocated variables. Depending on the program type, supplementary information may be stored, including heap structures in case of dynamic memory allocation, call stacks for recursive calls, and various data structures for managing concurrency (such as multiple call stacks, FIFO channels, etc.). The incorporation of these additional features often results in configurations of unbounded size. This unboundedness, in various forms, promptly leads to undecidability when attempting to ascertain properties about programs. Furthermore, the challenge escalates when attempting to reason about the collective interplay of these features, making the process exceedingly difficult, especially in the context of creating automated methods for program analysis.

The methodology introduced in this paper relies on a graph-based representation of computations, specifically termed *behavior graphs*. Within these graphs, vertices serve as representations of basic (finite) information within a given state. Various types of edges are employed to encapsulate state transitions, connections between states, and additional data structures used in the computation. To provide a visual example, envision a stack, which can be depicted by establishing a link between the pair of states associated with a push and its corresponding pop operation (see Fig. 3 for an illustration). Similarly, a queue can be captured by connecting an enqueue operation to its corresponding dequeue operation. These instances showcase the flexibility of behavior graphs in representing diverse computational structures, and further variations are readily conceivable. In conclusion, such graph representations of program behaviors provide a unified framework that empowers us to analyze programs with diverse features using the same reasoning engine and techniques developed for sequential recursive programs, as discussed later in this section.

The concept of representing executions using labeled graphs has been previously explored in the context of automata that employ auxiliary storage, and here, we extend this idea to programs. *Nested-words* effectively capture executions of pushdown automata, as demonstrated in [1]. Similarly, multiply nested-words encode runs of multistack pushdown automata. Furthermore, stack-queue graphs offer a versatile framework for modeling distributed automata, where each process is a pushdown automaton employing queues and stacks, as highlighted in [2]. In this paper, we adopt this concept from the realm of automata and elevate this representation to programs, where vertices are not labeled with symbols from a finite alphabet but rather with

tuples of numbers representing the evaluations of scalar variables in a given point of the execution.

Our approach enables the automatic analysis of programs whose computations can be represented as graphs of bounded *treewidth*. A class of graphs has treewidth k if each graph within it can be structured with a *tree decomposition* of *width* at most $k+1$. This means that the graph can be rearranged into a tree, assigning to each node a set (called *bag*) containing at most $k+1$ graph vertices. This arrangement must cover all vertices and edges, and any vertex appearing in the bags of two different nodes must also belong to all the bags along the path connecting those nodes. In essence, for a behavior graph G, a tree decomposition T of width k ensures that we can verify the consistency of the computation described by G by processing the information stored at each node of its tree decomposition and its neighbors. To ensure consistency across the edges of G, it is sufficient to examine one bag at a time. Furthermore, to guarantee that each vertex retains the same information in any bag where it appears, we only need to compare the bag at a node with those at its children, for each node in the tree.

The primary technical contribution of this paper is the development of a mechanism for generating binary trees that encode all tree decompositions of the behavior graphs under analysis. To achieve this, we introduce the concept of the *shape* of a behavior graph. A shape represents a small subgraph of a behavior graph or condenses explored portions while retaining the other nodes and edges. Shapes are themselves graphs, and we use them instead of bags in our tree decompositions. At an internal node, the shape condenses the information of the shapes labeling its children. This process involves initially a *merge* operation that glues the shapes of the two children, followed by a *contraction* operation that condenses vertices that have been fully explored. Essentially, in constructing a tree decomposition, we use these shapes to summarize the information of the portion of the behavior graph covered by the nodes of a subtree. In particular, we show that the trees of shapes obtained through this method, called *merge-and-contract trees*, can exactly represent the tree decompositions of nested-words.

We encode merge-and-contract trees for a given program P into a *symbolic data-tree automaton* that accepts all and only the behavior graphs of P for the given parameter k. Symbolic data-tree automata, or SDTAs, are computational models designed for symbolic processing of tree-structured data, introduced in [3]. In contrast to traditional tree automata that operate on concrete data values taken from a finite alphabet, SDTAs manipulate symbols and employ logical constraints expressed with formulas of quantifier-free first-order theories to recognize and process complex patterns within tree structures. For the encoding of merge-and-contract trees, each time an original edge of the encoded behavior graph is encountered in a shape (located in the shapes associated with the leaves of the merge-and-contract tree), the automaton enforces the constraints imposed by the program as part of its transition function. For internal nodes of the tree, we utilize the triples formed by the two shapes at the children and the one labeling the parent, where the vertices of the shape carry the evaluation of the variables associated with them. Here, we ensure, with constraints injected into the transition function of the SDTA, that the valuation of the variables in the children shapes representing the same vertex in the parent shape, and hence the final behavior

graph, agrees on the assigned values. When the root is labeled with a shape representing a complete behavior graph *failing* a program assertion, the tree is accepted. Thus, checking the emptiness of the constructed SDTA enables us to verify the fulfillment of the assertions for a program. Given that this reduction is amenable to automation, our proposed approach establishes an automatic methodology for program analysis.

The emptiness problem for SDTAs is undecidable in general, but it can be reduced to the satisfiability problem of *constrained Horn clauses* (CHCs), offering several advantages. Many algorithms have been developed for solving systems of CHCs, often building upon or generalizing techniques from the field of automatic program verification [4–6]. Consequently, CHCs are extensively used as an intermediate representation in various verification and synthesis tools [7–14]. Our reduction approach provides several advantages. Firstly, it enables a separation of concerns, allowing us to focus solely on aspects related to the specific class of programs under consideration. Simultaneously, it provides CHC solver developers with a clean framework that can be instantiated using various model-checking algorithms and specialized decision procedures. Our reduction to data tree automata allows us to reason about programs using approaches and techniques akin to those employed when reasoning about finite-state automata with auxiliary storage. In fact, we do not need to handle data directly and can concentrate solely on structural aspects. Furthermore, by expressing CHCs in the standard SMT-LIB language enables the use of different CHC engines and capitalize on their consistent year-over-year performance improvements, as demonstrated by the annual *competition on constrained Horn clauses* CHC-COMP [15].

In this paper, we meticulously elaborate on the intricacies of our approach for recursive sequential programs. Our aim is to present the concepts in a thorough manner, offering comprehensive insights while minimizing the use of notation. Specifically, the behavior graphs in this context take the form of *nested-words*, and we refer to their corresponding shapes as *nested-word shapes*. Since nested-words admit a tree decomposition of width two, we establish that our approach empower us to prove the full correctness of the programs. We then extend these foundational concepts to concurrent programs, elevating all relevant ideas and constructions to develop verification approaches for this more complex class of programs. Unlike nested-words, the class of behavior graphs for concurrent programs lacks a fixed bound on their treewidth. Consequently, our approach in this scenario can only verify correctness up to a given fixed bound. However, if it is known that the behavior graphs of the program under analysis is bounded, our approach remains effective in proving correctness. In cases where the behavior graph is unbounded, our method can still serve as an under-approximation technique.

This paper builds upon our previous work [16], expanding and refining the concepts therein to lay the groundwork for the contributions presented here. We offer a more comprehensive overview of the approach, providing formal definitions of the various constructions and formal correctness proofs. A key addition to our methodology is the utilization of symbolic data-tree automata (SDTAs), which allows us to streamline the analysis and ultimately reduce it to the satisfiability of constrained Horn clauses.

$\langle prgm \rangle$ ::= 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. 1 BNF grammar of (sequential) programs.

Organization of the paper: The rest of the paper is structured as follows. In Section 2, we formally define the recursive sequential programs, and related verification problems. In Section 3, we review basic notions of multigraphs, with a specific emphasis on nested-words and tree decomposition of multigraphs. Furthermore, we introduce a novel form of tree decomposition integral to out methodology. Moving to Section 4, we introduce the notion of nested-word shape, as a means to succinctly represent portions of nested-words. We define two essential operations on nested-word shapes, namely *merge* and *contraction*, and establish key properties of these operations pivotal for the development of our verification methodology. Section 5 provides a characterization of tree decompositions of nested-words through merge-and-contract trees, leveraging the insights gained in Section 4. The methodology itself is detailed in Section 6, beginning with an extension of nested-words through the annotation of nodes with program data to faithfully capture program executions. We show that this approach can be factorized by leveraging decompositions provided by merge-and-contract trees. Finally, we outline the encoding of these extended trees into symbolic data tree automata. In Section 7, we discuss the extension of our methodology to concurrent programs by lifting up our methodology to handle multiple nested-words. We delve into related work in Section 8, and Section 9 provides concluding remarks, extensions, and a discussion on future directions.

2 Programs with recursive procedure calls

To simplify our discussion while maintaining general applicability, we focus on sequential programs with recursive procedure calls, excluding local variables and procedure parameters. This choice ensures consistent access to all variables across procedures, achieved through the sole use of global variables. Henceforth, we will refer to these as simply "variables" throughout the paper.

Syntax. The syntax of our programs is defined by the BNF grammar presented in Fig. 1. Every program starts with a finite set of global variables, denoted by Var, accessible to all procedures within the program. Each variable belongs to a specific data type, which often has an infinite data domain. Common types include integers, floating-point rationals and real numbers, booleans ($\mathbb{B}$), and fixed-length bit vectors.

Additionally, we assume a language of expressions $\langle expr \rangle$, and a language of predicates $\langle pred \rangle$ over the variables.

After the variable declarations, programs consists of one or more *procedures*, including a special procedure called `main` that is executed first to start the program execution. Each procedure contains a non-empty sequence of labeled statements of the form $pc : \langle stmt \rangle$ where pc (*program counter* or *program location*) is a unique label that identifies a specific statement within the procedure, and $\langle stmt \rangle$ represents a statement in a C-like language. A statement *stmt* can be an assignment, a `skip`-statement, an `assume`- or `assert`-statement, a conditional statement, a `while`-loop, a call to a procedure, or a `return`-statement. We assume that each procedure has `return` as its last statement. Furthermore, `return` is never the first statement of a procedure.

For a given program P, we use the notation PC_P (and similarly, $Call_P$, Ret_P) to represent the set of all program counters pc for which $pc : stmt$ (resp., $pc :$ `call` p, $pc :$ `return`) is a labeled statement within the program P. Additionally, for each $pc \in Call_P$ we employ $afterCall_P(pc)$ to denote the (unique) program counter pc' such that $pc' : stmt$ is the statement that is executed after the return of 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. Consequently, a state is formally defined as a *configuration* of the form $\langle \nu, pc, St \rangle$ where ν represents the valuation of the variables, $pc \in PC_P$ is the program counter, and St captures the content of the call stack (i.e., the control locations of the pending procedure calls). An *initial* configuration $C = \langle \nu, pc, St \rangle$ is characterized by specific conditions: the program counter pc corresponds to the first statement of procedure `main`, and the call stack St is empty.

The *transition relation* between configurations, denoted by $\hookrightarrow$, is defined as usual. Assignment statements not only advance the program counter but also update the variable evaluation. The `skip`-statement simply updates the program counter to the next one. The control-flow statements, on the other hand, update the program counter, possibly based on a predicate (condition). Specifically, `assume`- and `assert`-statements both advance the program counter to the next one if the condition holds true, and halt the computation (i.e., no subsequent configuration is allowed) otherwise. The distinction between these two statements is evident when considering correctness: `assume`-statements are used to build more accurate program models by filtering out spurious computations, while `assert`-statements are used to specify correctness requirements, with a failed assertion signaling an error. Typically, a failing assertion leads to an error state, although we do not introduce a specific configuration for it here.

During a procedure call, the current location of the caller (pc) is pushed onto the stack, and the control shifts 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 $afterCall_P(pc)$.

A *computation* of a program is a sequence of configurations $C_0 C_1 \dots C_n$, where C_0 is initial, and $C_{i-1} \hookrightarrow C_i$ for every $i \in [1, n]$. An *error computation* is a computation $C_0 C_1 \dots C_n$ that *fails an assertion*. In other words, it is characterized by

```
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. 2 A sample program.

$C_n = \langle \nu, pc, St \rangle$, $pc : assert(cond)$ and, upon assigning the program variables with ν, $cond$ evaluates to *false*.

Verification problem. We address the *assertion checking problem*, which involves determining, for a given program P, whether there exists an error computation of P.

Example 1 Throughout the remainder of the paper, we utilize the program P illustrated in Fig. 2 as a **running example**. P is a simple program with three possible behaviours contingent on the initial value of the variable x, namely, 1, 2, or any other value. In the latter scenario, the condition of the `assume` statement fails, leading to an immediate halt in the computation. In the remaining cases, the procedures *boo* and *foo* get recursively invoked until the `assert` statement at program counter 5 is reached. Subsequently, a computation with $x = 2$ violates the assertion, thereby reaching an *error* state. Conversely, a computation with $x = 1$ progresses uninterrupted, proceeding through the entirety of procedure `main`.

2.1 Formulas for programs

In our methodology, we shall employ template formulas that succinctly capture the semantics of a program.

Data signatures. Data signatures are similar to structured data types in programming languages. A *data signature* $\mathcal{S}$ is defined as a finite collection of pairs $\{id_i : type_i\}_{i \in [n]}$, where each id_i represents a field name, and $type_i$ denotes the corresponding data type. Common types include integers, floating-point rationals, and real numbers, as well as the Boolean type $\mathbb{B}$ and fixed-length bit vectors. An *evaluation* ν of $\mathcal{S}$ is a mapping that associates each field name id in $\mathcal{S}$ with a value of the corresponding type, denoted by $\nu.id$. We denote by $\mathcal{E}(\mathcal{S})$ the set of all evaluations of $\mathcal{S}$.

Data logic. We use formulas expressed in quantifier-free first-order logic with equality, following standard syntax and semantics [17]. To accommodate various data types of program variables, we adopt many-sorted signatures. Specifically, we employ a many-sorted first-order logic $\mathcal{D}$ with sorts $data_1, \ldots, data_n$. Each $data_i$ has a corresponding logic $\mathcal{D}_{data_i}$ permitting function symbols of type $data_i^h \mapsto data_i$ and relation symbols of type $data_i^h \mapsto \mathbb{B}$, with arity h. These logics encompass a range of features, including but not limited to integer or real arithmetic, arrays, and more. As a result, we assume that $\mathcal{D}$ is well-equipped to handle diverse data types and program variables, ensuring adequacy for a variety of scenarios.

Program template formulas. For a program P, we define the data signature $\mathcal{S}_P^{state}$, which incorporates distinct fields for each program variable, including its type within

the program. Additionally, we define $\mathcal{S}_P$ as the data signature for P, consisting of two fields: one named *state* of type $\mathcal{S}_P^{state}$, and another field called *pc* representing the program counter (type PC_P). In the following, it is assumed that variables ν, ν_1, and ν_2 are all of type $\mathcal{S}_P$.

To articulate the semantics of program P, we introduce the following template formulas:

Initial states: $Init_P(\nu)$ is a formula that holds true if and only if $\nu.pc$ is the program counter of the first statement of procedure `main`.

State update: $Trans_P(\nu_1, \nu_2)$ imposes constraints on program variables and counters in consecutive program configurations. For instance, in an assignment, it computes the variables in the next state $\nu_2.state$ based on those in the current state $\nu_1.state$ and updates the program counter. In the event of a procedure call, such as `call` p, it ensures that the valuations of the fields in $\nu_1.state$ and $\nu_2.state$ corresponding to program variables align, while $\nu_2.pc$ is set to the program counter of the initial statement in p. Finally, if ν_1 corresponds to a return statement, $Trans_P(\nu_1, \nu_2)$ only copies the values of ν_1 to ν_2 field by field, leaving $\nu_2.pc$ unconstrained. The handling of $\nu_2.pc$ is deferred to the subsequent template formula.

Matching call-return: $CallRet_P(\nu_1, \nu_2)$ is a formula that evaluates to *true* if and only if $\nu_1.pc \in Call_P$, and $\nu_2.pc = afterCall_P(\nu_1.pc)$.

Error state: The formula $AssertionFail_P(\nu)$ holds *true* if and only if $\nu.pc$ is the program counter of an assertion statement whose expression evaluates to *false* on the program variable valuation $\nu.state$.

3 Multigraphs and Decompositions

In this section, we review fundamental notions related to multigraphs and recall the concept of nested-words as a subclass of multigraphs. In our framework, nested-words will serve as the underlying structure for our representation of program executions. Additionally, we revisit the concept of graph decomposition, refining it by also mapping the edges to bags. This will better align with our specific objectives.

Multigraphs. A *multigraph* is defined as a structure $G = (V, E_1, \ldots, E_n)$, where V is a finite set of vertices, and each $E_i \subseteq (V \times V)$ represents a set of directed edges. In this context, an edge $(u, v) \in E_i$ is also denoted as uE_iv. Furthermore, for two vertices $u, v \in V$, we use the notation uE_i^*v to indicate that either $u = v$, or there exists a node $z \in V$ such that uE_i^*z and zE_iv. As usual, we use uE_i^+v to denote that uE_i^*v and there is at least an edge connecting u to v.

Given a multigraph $G = (V, E_1, \ldots, E_n)$, a *sub-multigraph* of G is a multigraph $G' = (V', E_1', \ldots, E_n')$ such that the vertices of G' are a subset of the vertices of G, i.e., $V' \subseteq V$, and the edge sets of G' are each contained within the corresponding edge set of G, i.e., for each index $i \in [1, n]$, $E_i' \subseteq E_i$.

A *graph* is a multigraph $G = (V, E)$ with a single set of edges. A *line graph* is a graph in which, for a total order of all the vertices $v_0, v_1, \ldots, v_m$, the set of edges E is defined as $E = \{v_{j-1}Ev_j \mid j \in [m]\}$.

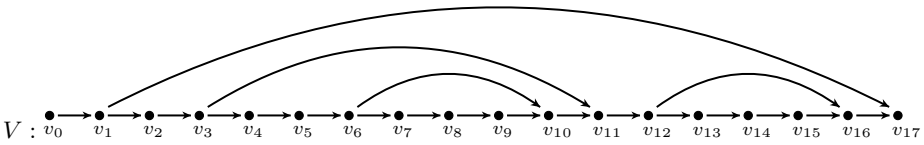

Fig. 3 Illustration of a nested-word (multigraph).

Given two multigraphs $G^i = (V^i, E^i_1, \ldots, E^i_n)$ for $i \in [1,2]$, the *union* of G_1 and G_2 is defined as $(V^1 \cup V^2, E^1_1 \cup E^2_1, \ldots, E^i_n \cup E^2_n)$. G^1 and G^2 are *edge-disjoint* if $E^1_i \cap E^2_i = \emptyset$ for $i \in [1,n]$.

Matching relations. For a line graph $(V, \rightsquigarrow)$, we say that $\curvearrowright \subseteq V \times V$ is a *matching relation* w.r.t. $\rightsquigarrow$ (resp. *loose matching relation*) if for every $u, v, x, y \in V$:

- if $u \curvearrowright v$, then $u \rightsquigarrow^+ v$ (resp. $u \rightsquigarrow^* v$);
- if $u \curvearrowright v$, $u' \curvearrowright v'$, and $u \rightsquigarrow^+ u'$ (resp. $u \rightsquigarrow^* u'$), then either $v \rightsquigarrow^+ u'$ or $v' \rightsquigarrow^+ v$ (resp. $v' \rightsquigarrow^* v$).

Definition 1 (Nested-Words) A *nested-word*[1] is defined as a multigraph $(V, \rightarrow, \curvearrowright)$, subject to the following two conditions:

1. $(V, \rightarrow)$ is a line graph, and
2. $\curvearrowright$ is a matching relation w.r.t. $\rightarrow$;

The edges in $\rightarrow$ and $\curvearrowright$ are referred to as *linear edges* and *matching edges*, respectively. A *sub-nested-word* is any sub-multigraph of a nested-word. □

In Fig. 7, we illustrate an instance of a nested-word, featuring vertices v_0 through v_{17}. As suggested by our notation, the $\curvearrowright$-edges are depicted as curved arrows, while the $\rightarrow$-edges are represented as straight arrows.

Tree Decompositions and Treewidth of Multigraphs. Our methodology employs tree decompositions of nested-words to analyze the computations of the given program. Initially, we revisit the concepts of tree decomposition and treewidth.

A *binary tree* T is formally represented as a multigraph $T = (V, L, R)$. To facilitate technical considerations, we define V as a finite and prefix-closed subset of $\{0,1\}^*$. Within this framework, the elements of T are referred to as *nodes*, and the node identified by ϵ is called the *root* of T. The *edge relation* is implicitly defined as follows: xLy if and only if $y = x0$, and xRy if and only if $y = x1$. When xLy (respectively, xRy), we say that y is the *left* (respectively, *right*) *child* of x, and x is the *parent* of y. A *leaf* is characterized as a node with no children, while an *internal node* is defined as a node that is not a leaf. T is *fully binary* if every internal node u of T has exactly two children.

Informally, a *tree decomposition* of a graph G is a binary tree characterized by the following properties. The nodes of the tree are labeled with sets of vertices from G, referred to as *bags*. Each edge or vertex in G is encompassed by at least one bag within

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

the decomposition – an edge is considered covered if both of its endpoints are contained within a same bag. Moreover, if a vertex v belongs to the bags of two distinct nodes, then all bags along the path connecting those nodes also include vertex v. Formally, a tree decomposition is defined as follows.

Definition 2 (Tree Decomposition and Treewidth) A *tree decomposition* of a multigraph $G = (V, E_1, \ldots, E_n)$ is defined as a pair $D = (T, \{bag_t\}_{t \in N})$, where T is a binary tree with a set of nodes N, and $bag_t \subseteq V$ satisfies the following properties:

- For every $v \in V$, there exists at least one node $t \in N$ such that $v \in bag_t$.
- For every $i \in [1, n]$ 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 node t'' of T lying 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, formally, $\max_{t \in N}\{|bag_t|\} - 1$. The *treewidth* of a multigraph is the *minimum* width over all its tree decompositions. □

Example 2 Fig. 4(a) provides a visual representation of a tree decomposition of the nested-word illustrated in Fig. 3. As required by the definition, each vertex within $v_0, \ldots, v_{17}$ belongs to at least one bag, and labels all the nodes along a path connecting two nodes labeled by it. For instance, vertex v_{11} is part of the bags of nodes n_0, n_1, n_2, n_4 and n_5. If we remove v_{11} from n_1, the resulting tree ceases to be a tree decomposition.

The explicit coverage of the edges is not presented in Fig. 4(a). However, a witness of this coverage is given in Fig. 4(b). This augmented version of the tree decomposition will be discussed in more detail later in this section.

For the tree decomposition from Fig. 4(a), the maximum size over all the bags is 6 which is the size of n_3 and n_4. Therefore, the width of this decomposition is 5. Note that this width falls short of being optimal; in fact for nested-words it is always possible to determine a tree decomposition of width at most 2, as shown by the following theorem. □

Theorem 1 ([2]) *Any nested-word has treewidth at most* 2.

Augmented tree decompositions. In this context, we refine the previously defined tree decomposition concept to better suit our specific objectives. We augment traditional 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. It is worth noting that such a labeling is always feasible, as by definition, each edge is covered by at least one bag. To provide a concrete illustration, consider the tree decomposition depicted in Fig. 4(b). We augment this decomposition by representing the covered edges and vertices through corresponding fragments of the nested-word. Formally:

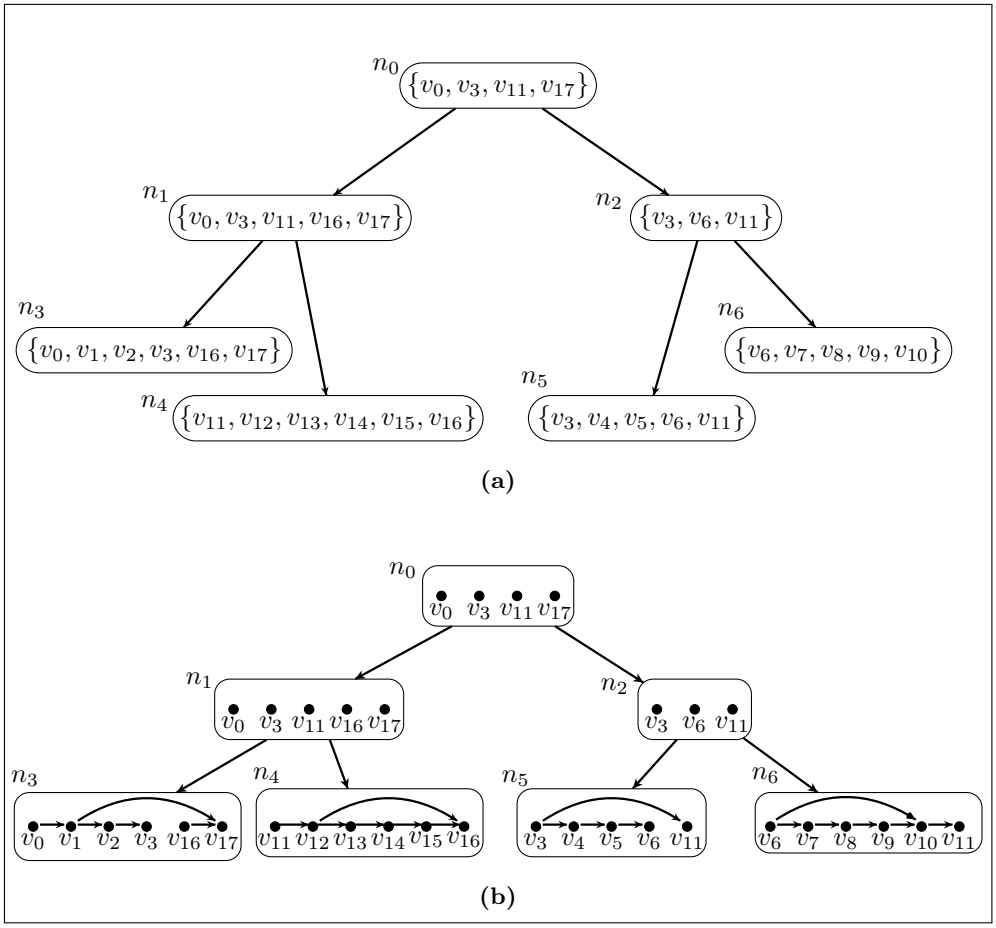

Fig. 4 Example of: **(a)** a tree decomposition, and **(b)** a strict augmented tree decomposition of the nested-word illustrated in Fig. 3.

Definition 3 (Augmented Tree Decomposition) An *augmented tree decomposition* of a multigraph $G = (V, E_1, \ldots, E_n)$ is formally defined as a pair $D = (T, \{G^t\}_{t \in N})$, where the components are specified as follows:

- T is a fully binary tree with a set of nodes N.
- $G^t = (V^t, E_1^t, \ldots, E_n^t)$ is a sub-multigraph of G.
- $(T, \{V^t\}_{t \in N})$ is a tree decomposition of G.
- For every $i \in [1, n]$, every edge aE_ib is an edge of exactly one multigraph G^t with $t \in T$, and moreover, $\bigcup_{t \in T} E_i^t = E_i$.

The width of an augmented tree decomposition D is the *width* of the underlying tree decomposition $(T, \{V^t\}_{t \in N})$. □

For nested-words, we are interested in augmented tree decompositions with two additional properties: edges are mapped to the leaves of the tree decomposition and

the bag labeling the root must always contain the first and the last vertex in the nested-word. The motivation behind these the two properties stems from their role in our methodology. Formally:

Definition 4 (Strict Augmented Tree Decomposition of Nested-Words) A *strict* augmented tree decomposition of a nested-word $\omega = (V, \rightarrow, \curvearrowright)$ is a pair $D = (T, \{G^t\}_{t \in N})$, where D is an augmented tree decomposition and the following conditions hold:

1. for any internal node t of T, G^t has an empty set of edges, formally expressed as $\rightarrow^t = \curvearrowright^t = \emptyset$;
2. the set V^ε, representing the set of vertices of the multigraph labeling the root of T, includes the first and the last vertices in the total order induced by $\rightarrow$. □

The augmented tree decomposition given in Fig. 4(b) is a strict one. In fact, the internal nodes are only labeled with vertices, and v_0 and v_{17}, i.e., the first and last vertices of the nested-word ω given in Fig. 3, label the root.

Notably, the two additional requirements of strict augmented tree decompositions do not impose any limitations on the decomposability of nested-words: we demonstrate that for any tree decomposition of width k, there exists a corresponding strict augmented tree decomposition of the same width, and viceversa.

Theorem 2 *A nested-word ω admits a tree decomposition of width k if and only if it admits a (strict) augmented tree decomposition of width* $\max(2, k)$. *Moreover, any nested word admits a strict augmented tree decomposition of width at most* 2.

Proof The converse direction of the theorem is straightforward, as the width of an augmented tree decomposition precisely aligns with the width of the underlying tree decomposition.

To complete the proof, we need to show that for any tree decomposition of a nested-word, there exists a corresponding strict augmented tree decomposition with the same width.

It is simple to obtain an augmented tree decomposition starting from a tree decomposition. In fact, since every edge is covered by at least one bag we choose exactly one node and include the edge in it.

To adhere to condition 1 of Definition 4, we propose an iterative transformation procedure that can be applied to a given augmented tree decomposition. For each internal node t labeled with edges, the procedure unfolds as follows:

1. insert a fresh internal node t_p as child of t's parent (if t is the root make this new node the new root), make t its left child and add a leaf t_f as right child of t_p;
2. label t_p and t with a multigraph that retains only the vertices form the multigraph G^t labeling t, and label t_f only with the edges (along with all their endpoints) of G^t.

Clearly, the augmented tree decomposition resulting from this process maintains the same width as the initial one, and the asserted property holds.

Now, starting from an augmented tree decomposition of width k satisfying property 1 of strict augmented tree decompositions, we use induction to prove that we can construct an augmented tree decomposition of width at most k where also property 2 holds. We show this property for any n-prefix of a nested-word. Here, for n-prefix we refer to the sub-nested

word composed by the first n vertices in the ordering induced by the linear edges and their incident edges. Notably, a prefix may contain unmatched calls but no unmatched returns, and any nested word can be considered as a prefix of another nested word. The base case is straightforward. For a 1-prefix of a nested-word we can construct a strict augmented tree decomposition using a single node labeled with the entire nested-word. We now proceed to the induction step. Assume that the claim holds true for any n-prefix with at most $n \geq 1$ vertices. Let ω denote an $(n+1)$-prefix of a nested-word with its vertices listed as $v_1 \ldots, v_{n+1}$ according to the linear ordering induced by the edges of ω. We consider two cases.

1. v_{n+1} *is not the right endpoint of a matching edge.* In this case, there is only a linear edge incident to v_{n+1}. Thus, by applying the induction hypothesis to the prefix of ω formed by the first n vertices (and the edges whose endpoints are incident on $v_1 \ldots, v_n$), we get for it a strict augmented tree decomposition D_n of width at most k. We construct the desired tree decomposition D_{n+1} for ω as follows. Create a new root node, and make D_n its left subtree, and add another fresh node as its right subtree. We label the root with only vertices v_1, v_n and v_{n+1}. Then, label its right child (the leaf) with the multigraph formed by only the linear edge connecting v_n to v_{n+1}; this is clearly a strict augmented tree decomposition and its width is at most k.
2. v_{n+1} *is the right endpoint of a matching edge.* Let v_h denote the other endpoint of this edge. We can thus partition the $(n+1)$-prefix into a prefix up to v_h, and a nested-word from v_h to v_{n+1} (note that no edges connect vertices from these two multigraphs except for v_h). The strict augmented three decomposition is then obtained by taking a new node and making it the root of a tree where the left subtree is a strict augmented tree decomposition D_L of the h-prefix and the right subtree is that of the nested word from v_h to v_{n+1}, say D_R. By the induction hypothesis these two tree decompositions exist and have width at most k. Now to conclude the construction we just label the root with v_1, v_h and v_{n+1}. The resulting tree is clearly a tree decomposition since by the inductive hypothesis, v_1 and v_h must label the root of D_L and v_h and v_{n+1} the root of D_R. Moreover, the root is not labeled with edges and the width of the constructed tree decomposition is the maximum over k and 2.

The second part of the theorem is a direct consequence of the first part and Theorem 1. □

4 Nested-Word Shapes

In this section, we introduce the notion of *nested-word shape* (*nw-shape*), a formalism crafted to encode sub-nested-words such that some parts are expressed explicitly and others retain only the important parts needed for composition with other nw-shapes. The idea is to use them in our methodology as an intermediate representation to generate tree decompositions of nested-words progressively in steps.

Technically, a nw-shape is a labeled multigraph that is essentially a sub-nested-word with the addition of a total ordering over the vertices and a labeling of the vertices, and with some minor deviations.

For sub-nested-words, a total ordering can be inherited by the corresponding nested-word. However, they can be formed of separate multigraph components. Therefore, one needs to explicitly mention such ordering when the nested-word is not known. Similarly, also nw-shapes may be formed of separate multigraph components and, therefore, in our definition we fix a total ordering among all the vertices of a nw-shape. This total ordering is given as an edge relation that forms a line graph when coupled

with the vertices of the nw-shape and is such that the linear edges of the nw-shape are a subset of the edges of such line graph (i.e., the total ordering must not contradict the partial ordering defined by the linear edges).

Then, we equip nw-shape with two kinds of labeling. The first labeling, which we usually denote with τ, gives a type to each vertex, distinguishing between starting endpoints (`call`) and ending endpoints (`ret`) of a matching edge, and any other vertex (`int` for *internal*). The second labeling, which we usually denote with ℓ, is used to annotate a vertex with the information whether the incident left (`l`) and right (`r`) linear edges, and the possibly incident matching edge (`m`) are present in the multigraph . For the left and the right linear edges this information is only a matter of convenience, since that information could be computed from the set of linear edges. However, for the matching edges instead it is needed since we allow for the contraction of portions of the multigraph that do not contain both the endpoints of a matching edge. In fact, when such a portion of graph is contracted, we substitute a removed endpoint with one of the surviving vertices according to a search strategy that will be detailed later in the paper. Thus, we use the marking to distinguish for the vertices of type either `call` or `ret` between the cases when the incident matching edge is the expected one or not.

A second difference with the sub-nested-words concerns with the matching edges. For nw-shapes, we relax the constraint we have imposed on the nested words and only require that the matching edges just define a loose matching relation. This is required in order to allow the contraction of portions that do not contain both the endpoints of a matching edge.

The third and last difference with the sub-nested-words is that we do not allow for isolated vertices in nw-shapes. This restriction is only introduced to simplify our notation mostly in relation to the merging of nw-shapes. It could be avoided by slightly modifying the conditions in our results and deal with isolated vertices as a separate corner case.

Fig. 5 gives some examples of labeled multigraphs. In the figure, the total ordering is given implicitly by listing the vertices from left to right according to the intended ordering (that matches also the ordering given by increasing indices). The τ labeling is shown by denoting the `call` vertices as white ellipses, the `ret` vertices as grey ellipses, and the `int` vertices as black circles. For the ℓ labeling, markings `l` and `r` are captured by the incidence of the linear edges, and we only mark explicitly the vertices when they hold the label `m`.

The multigraph in Fig. 5(a) is an nw-shape. In fact, the matching edges form a loose matching relation (it is not a matching relation since the two curved edges share the same starting endpoint), the linear edges conform to a total ordering of the vertices and labeling conforms to the intended meaning of the labels: only vertices that are either `call` or `ret` are marked with `m`. Note that, the mark `m` on vertex v_3 denotes that it is the `call` endpoint of the matching edge from v_3 to v_{11}, and is a placeholder for the `call` endpoint of the matching edge that ends onto v_{17}. There is no ambiguity since when contracting portions of the multigraph we move the canceled endpoints inside (i.e., below) the curved edge up to the first surviving vertex. In particular, this nw-shape can be obtained by a contraction from an nw-shape corresponding to a portion

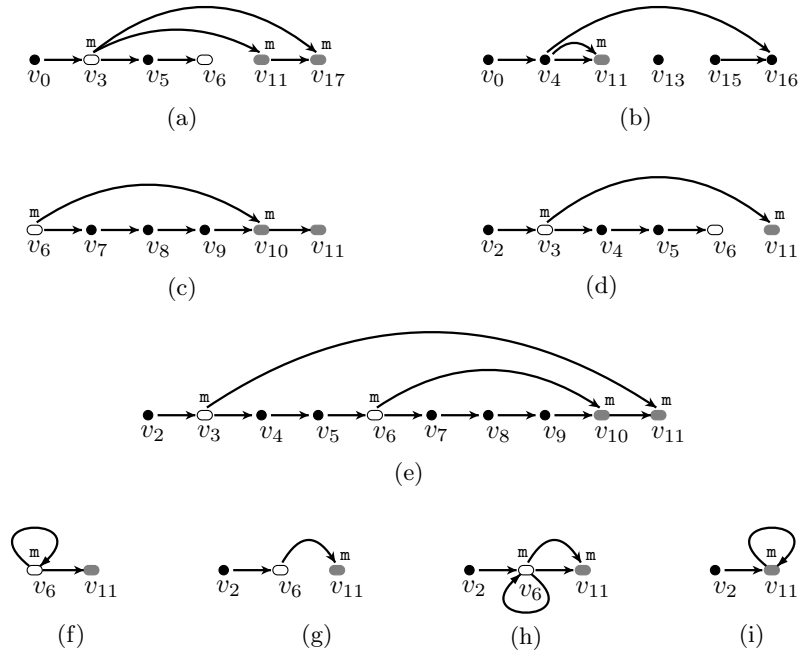

Fig. 5 Examples of multigraphs: (a) sample of nw-shape; (b) labeled multigraph which is not a nw-shape; (c) nw-shape S_1 corresponding to a sub-nested-word of the nested-word in Fig. 3; (d) nw-shape S_2 corresponding to a sub-nested-word of the nested-word in Fig. 3; (e) $S_3 = \mathit{merge}(S_1, S_2, \rightarrow)$ where $\rightarrow$ induces an ordering of the vertices by increasing indices, S_3 is fully expanded; (f) $S_4 = \mathit{contract}(S_1, \{v_6, v_{11}\})$; (g) $S_5 = \mathit{contract}(S_2, \{v_2, v_6, v_{11}\})$; (h) $S_6 = \mathit{merge}(S_4, S_5, \rightarrow_{\{v_2, v_6, v_{11}\}})$ which can also be obtained as $\mathit{contract}(S_3, \{v_2, v_6, v_{11}\})$; (i) $S_7 = \mathit{contract}(S_6, \{v_2, v_{11}\})$ which can also be obtained directly as $\mathit{contract}(S_3, \{v_2, v_{11}\})$.

of the nested-word given in Fig. 3 and this curved edge is representative of the curved edge from v_1 to v_{17} there.

The multigraph in Fig. 5(b) is not an nw-shape since it has an isolated vertex, besides this the rest of the graph conforms to our notion of nw-shape.

Formally, nested-word shapes are defined as follows.

Definition 5 (Nested-Word Shapes) A *nested-word shape* (nw-shape) is a tuple $S = (\sigma, \rightsquigarrow, \ell, \tau)$, where $\sigma = (V, \rightarrow, \curvearrowright)$ is a multigraph such that the following holds:

1. $(V, \rightsquigarrow)$ is a line graph, and $\rightarrow$ is a subset of $\rightsquigarrow$;
2. $\ell : V \mapsto 2^{\{\mathtt{l},\mathtt{r},\mathtt{m}\}}$;
3. $\tau : V \mapsto \{\mathtt{int}, \mathtt{call}, \mathtt{ret}\}$ gives the type of each vertex;
4. $\curvearrowright$ is a loose matching relation w.r.t. $\rightsquigarrow$;
5. the labeling is *consistent*, that is:
   a. for $u \rightarrow v$, $\mathtt{r} \in \ell(u)$ and $\mathtt{l} \in \ell(v)$ must hold;
   b. if $\mathtt{r} \in \ell(u)$, then $u \rightarrow v$ for some v;
   c. if $\mathtt{l} \in \ell(v)$, then $u \rightarrow v$ for some u;
   d. if $\tau(u) = \mathtt{call}$ and $\mathtt{m} \in \ell(u)$, then $u \curvearrowright v$ for some v;

e. if $\tau(v) = \texttt{ret}$ and $\texttt{m} \in \ell(v)$, then $u \curvearrowright v$ for some u;
f. if $\texttt{m} \in \ell(u)$, then either $\tau(u) = \texttt{call}$ or $\tau(u) = \texttt{ret}$;
g. $\ell(u) \neq \emptyset$ for each $u \in V$.

S is *ground* if $\curvearrowright$ is also a matching relation, and for any pair $u \curvearrowright v$, it must hold that $\texttt{m} \in \ell(u)$ and $\texttt{m} \in \ell(v)$.
The *size* of S is $|V|$. □

If the edge relation $\curvearrowright$ of a nw-shape is a matching one, then the underlying multigraph is indeed a sub-nested-word. Thus we get:

Lemma 1 *If $S = (\sigma, \rightsquigarrow, \ell, \tau)$ is a ground nw-shape, then σ is a sub-nested-word.*

The three multigraphs from Fig. 5(c–e) (as already informally observed before) are all nw-shapes according to the above definition, moreover since the curved edges form their matching relations (there is no overlap of such edges not even at their endpoints) they are also ground, and is simple to verify that the underlying multigraphs (obtained by removing the labeling) are indeed sub-nested-words of the nested-word given in Fig. 3.

Definition 6 (Fully Expanded Nested-Word Shapes) Let $S = (\sigma, \rightsquigarrow, \ell, \tau)$ be an nw-shape with set of vertices V. A vertex $u \in V$ is *fully expanded* whenever the following conditions hold:

1. If u is the first vertex in the line graph $(V, \rightsquigarrow)$, then
   - if $\tau(u) \in \{\texttt{call}\}$ then $\ell(u) = \{\texttt{r}, \texttt{m}\}$; otherwise, $\ell(u) = \{\texttt{r}\}$.
2. If u is the last vertex in the line graph $(V, \rightsquigarrow)$, then
   - if $\tau(u) \in \{\texttt{ret}\}$ then $\ell(u) = \{\texttt{l}, \texttt{m}\}$; otherwise, $\ell(u) = \{\texttt{l}\}$.
3. If u is an internal vertex in the line graph $(V, \rightsquigarrow)$, then
   - if $\tau(u) \in \{\texttt{call}, \texttt{ret}\}$ then $\ell(u) = \{\texttt{l}, \texttt{r}, \texttt{m}\}$; otherwise, $\ell(u) = \{\texttt{l}, \texttt{r}\}$

S is a *fully expanded nw-shape* if all of its vertices are fully expanded. □

The nw-shape from Fig. 5(c), even if it is totally connected, is not fully expanded since the `ret` vertex v_{11} is not matched by a curved edge. The nw-shape from Fig. 5(e) is instead fully expanded.

Every ground and fully expanded nw-shape has a nested-word as its underlying multigraph. This result follows directly from two properties of a ground and fully expanded nw-shape:

1. *Matching relation*: the edge relation $\curvearrowright$ of a ground nw-shape is a matching relation (refer to Definition 5), and
2. *Line graph*: the fulfillment of the condition of being fully expanded results in the $\rightarrow$ edge relation forming a line graph with the vertices of the nw-shape (see Definition 6).

These two characteristics precisely define a nested-word. This assertion is explicitly formalized in the following lemma.

Lemma 2 *If $S = (\sigma, \rightsquigarrow, \ell, \tau)$ is a ground and fully expanded nw-shape, then σ is a nested-word.*

The multigraph from Fig. 5(e) is ground and fully expanded as observed before. It simple to verify that its underlying multigraph indeed forms a nested-word.

4.1 Operations on shapes

In this section, we lay the groundwork for our verification methodology by introducing two essential operations on nested-word shapes: *merge* and *contraction*. We also establish key properties of these operations, playing a critical role in proving the soundness of our approach.

Contraction

The contraction operation essentially compresses linearly connected portions of a nw-shape into single $\rightarrow$-edge. This process preserves the labeling of the surviving vertices. Furthermore, in a contraction some $\curvearrowright$-edges may have their endpoints adjusted if they are part of a compressed portion, or they may just disappear if the edge lays entirely within one such portion. As a result, in the underlying multigraph, the property of being linearly connected is both preserved in and not a direct consequence of any contraction.

The multigraph in Fig. 5(f) is the contraction of the one in Fig. 5(c) w.r.t. the set of vertices $\{v_6, v_{11}\}$. Note that the entire sequence of linear edges from v_6 to v_{11} has collapsed into a single linear edge. Additionally, since vertex v_{10} has been abstracted away, the incident matching edge has been redirected to v_6, which is the closest vertex to v_{11} among the surviving ones that are below the original edge. Each surviving vertex retains the original labeling. Clearly, since v_6 is of type `call`, the graph carries also the information that the `ret` endpoint of the original matching edge has been abstracted away (along with all the underlying portion of the starting nw-shape). Also, the multigraph in Fig. 5(g) is obtained by contraction, this time w.r.t. the set of vertices $\{v_2, v_6, v_{11}\}$ and from the nw-shape in Fig. 5(d). Observe that here, the left endpoint of the matching edge from v_3 to v_{11} has been moved to v_6 which is not labeled with `m` correctly meaning that this is not the matching edge that starts from v_6 but one coming from an outer `call` vertex.

Notably, both the multigraphs in Fig. 5(f–g) are also nw-shapes. This is not a coincidence but a property of our notion of contraction as we will state in a lemma after giving the formal definition of contraction.

Before getting to the formal definition of contraction of a nested-word shape, we leverage the notion of *edge contraction*. Given a generic edge relation $\rightarrow$ on a set of vertices V, and a subset of vertices $U \subseteq V$, we define the *contraction of $\rightarrow$ over U*, denoted $\rightarrow_U \in (U \times U)$, as a new relation on U. This relation connects vertices of U that are either directly linked by $\rightarrow$ or indirectly connected through a $\rightarrow$-path involving only vertices *outside* U. Formally, for any $u, v \in U$, $u \rightarrow_U v$ holds if and only if:

- $u \rightarrow v$, or

- there exists a sequence of vertices $u_1, \ldots, u_m \in (V \setminus U)$ such that $u \to u_1 \to \ldots \to u_m \to v$.

Definition 7 (Contraction) Let $S = (\sigma, \rightsquigarrow, \ell, \tau)$ be an nw-shape with $\sigma = (V, \to, \curvearrowright)$, and $V' \subseteq V$.

Contractability: S is *contractable* w.r.t. V' if all the vertices of S that are not fully expanded belong to V'.

Contraction operation: If S is contractable w.r.t. V', the *contraction* of S w.r.t. V', denoted by $contract(S, V')$, is the nw-shape $(\sigma', \rightsquigarrow', \ell', \tau')$ with $\sigma' = (V', \to', \curvearrowright')$:

- $\rightsquigarrow'$ is the contraction of $\rightsquigarrow$ over V', i.e., $\rightsquigarrow' = \rightsquigarrow_{V'}$,
- $\to'$ is the contraction of $\to$ over V', i.e., $\to' = \to_{V'}$;
- $(x, y) \in \curvearrowright'$ if there exists $u \curvearrowright v$ such that
  - $u \rightsquigarrow^* x \rightsquigarrow^* y \rightsquigarrow^* v$;
  - if $u \in V'$ then $x = u$, otherwise $u \to_U x$ where $U = V' \cup \{u\}$;
  - if $v \in V'$ then $y = v$, otherwise $y \to_U v$ where $U = V' \cup \{v\}$.

Well-definedness: The contraction operation $contract(S, V')$ is *well-defined* if S is contractable w.r.t. V'. □

In the following, we give some key properties of the contraction operation.

The following two lemmas are a direct consequence of the definitions and the informal observations given at the beginning of this section.

Lemma 3 *Any contraction of an nw-shape is also an nw-shape.*

Lemma 4 *Let S be an nw-shape, and V be a set of vertices such that S is contractable w.r.t. V. It holds that S is fully expanded if and only if $contract(S, V)$ is fully expanded.*

Another relevant property of nw-shapes is the following. The nw-shape S' resulting from a sequential application of two contractions, first w.r.t. a set V_1 and subsequently w.r.t. V_2, applied to an initial shape S, is exclusively determined by S and V_2. For example, the nw-shape in Fig. 5(i) can be obtained either as $contract(contract(S_3, \{v_2, v_6, v_{11}\}), \{v_2, v_{11}\})$ or directly as $contract(S_3, \{v_2, v_{11}\})$, where S_3 is the nw-shape given in Fig. 5(e). This property is formally stated in the following lemma.

Lemma 5 *Let S be an nw-shape, and V_1, V_2 be two sets of vertices such that S is contractable w.r.t. V_1, and $contract(S, V_1)$ is contractable w.r.t. V_2. It holds that S is contractable w.r.t. V_2, and $contract(contract(S, V_1), V_2) = contract(S, V_2)$.*

Proof Define $S_1 = contract(S, V_1)$ and $S_2 = contract(S_1, V_2)$. The condition $V_2 \subseteq V_1$ is a prerequisite for S to be contractable w.r.t. V_1 and S_1 to be contractable w.r.t. V_2. Furthermore, given that contraction maintains the labeling of surviving vertices unchanged, it follows that

S is also contractable with respect to V_2. Directly applying the definition of the contraction operation, we deduce that $S_2 = \mathit{contract}(S, V_2)$. □

Merge

As its name suggests, the *merge operation* seamlessly integrates two nested nw-shapes into a single nw-shape. This unification entails combining the vertices, $\rightarrow$-edges, and $\curvearrowright$-edges of the two nw-shapes through set union. The vertices' labels are adjusted in accordance with the merged edge relations. Notably, not every merge of nw-shapes yields a valid nw-shapes. To address this limitation, we introduce a definition of *mergeable* that precisely captures this condition. We also present fundamental properties that are essential for the verification methodology outlined in this paper.

Definition 8 (MERGE) Let $S_i = (\sigma_i, \rightsquigarrow_i, \ell_i, \tau_i)$ with $\sigma_i = (V_i, \rightarrow_i, \curvearrowright_i)$ for $i \in \{1, 2\}$ be two nw-shapes, and $\rightsquigarrow$ be such that $(V_1 \cup V_2, \rightsquigarrow)$ is a line graph.

Mergeability: S_1 and S_2 are *mergeable* with respect to $\rightsquigarrow$ if:

1. $\rightsquigarrow_1 = \rightsquigarrow_{V_1}$, and $\rightsquigarrow_2 = \rightsquigarrow_{V_2}$;
2. $\rightarrow_1, \rightarrow_2 \subseteq \rightsquigarrow$;
3. $(\rightarrow_1 \cap \rightarrow_2) = \emptyset$;
4. $\curvearrowright_1 \cup \curvearrowright_2$ is a loose matching relation w.r.t. $\rightsquigarrow$;
5. for each $u \in (V_1 \cap V_2)$, $\tau_1(u) = \tau_2(u)$ and $(\ell_1(u) \cap \ell_2(u)) = \emptyset$.

Merge operation: If S_1 and S_2 are mergeable w.r.t. $\rightsquigarrow$ then the *merge* of S_1 and S_2, denoted $\mathit{merge}(S_1, S_2, \rightsquigarrow)$, is $(\sigma, \rightsquigarrow, \ell, \tau)$ where $\sigma = (V, \rightarrow, \curvearrowright)$ and:

- σ is the union of σ_1 and σ_2;
- $\tau(u) = \tau_1(u)$ if $u \in V_1$, and $\tau(u) = \tau_2(u)$ otherwise;
- $\ell(u) = \ell_1(u)$ if $u \in (V_1 \setminus V_2)$, $\ell(u) = \ell_2(u)$ if $u \in (V_2 \setminus V_1)$, and $\ell(u) = (\ell_1(u) \cup \ell_2(u))$ otherwise.

Well-definedness: The merge operation $\mathit{merge}(S_1, S_2, \rightsquigarrow)$ is *well-defined* if S_1 and S_2 are mergeable w.r.t. $\rightsquigarrow$. □

The multigraph given in Fig. 5(e) is the merge of the two nw-shapes given in 5(c–d) w.r.t. the edge relation given by the union of their respective linear edges. It is simple to verify that the two starting nw-shapes satisfy the mergeability condition w.r.t. the mentioned edge relation.

In the following, we highlight some key properties of the merge operation also in combination with the contraction operation.

If the underlying multigraphs of two nw-shapes are edge-disjoint sub-nested-words of a given nested-word, then all the criteria of the mergeability definition with respect to the edge relation inherited by the nested-word are straightforwardly met (see Definition 8). Therefore, the following lemma holds.

Lemma 6 *For $i \in \{1, 2\}$, let S_i be a ground nw-shape where V_i is the set of vertices of its underlying multigraph σ_i, and ω be a nested word with set of linear edges $\rightarrow$.*

If σ_1 and σ_2 are edge-disjoint sub-nested-words of ω, then $\mathit{merge}(S_1, S_2, \rightarrow_{V_1 \cup V_2})$ is well-defined.

Note that, the multigraphs underlying the two nw-shapes given in 5(c–d) are as observed before in this paper are sub-nested-words and additionally they are also edge disjoint. Thus, mergeability in this case could be shown by the using the above lemma.

The next lemma naturally follows from Definition 8 and can be seen again comparing the three nw-shapes from Fig. 5(c-e).

Lemma 7 *Let $S_i = (\sigma_i, \rightsquigarrow_i, \ell_i, \tau_i)$ for $i \in \{1, 2\}$ be two nw-shapes with set of vertices V_i and $\rightsquigarrow$ be such that $(V_1 \cup V_2, \rightsquigarrow)$ is a line graph.*

If $merge(S_1, S_2, \rightsquigarrow)$ is well-defined, then denoting $merge(S_1, S_2, \rightsquigarrow) = (\sigma, \rightsquigarrow, \ell, \tau)$, σ is the union of σ_1 and σ_2.

The following lemma states a fundamental property of the merge operation: the merge of two mergeable nw-shapes result in another nw-shape, and this operation preserves the property of being ground. Again, this lemma can be seen by the three nw-shapes from Fig. 5(c-e).

Lemma 8 *Given two nw-shapes S_1 and S_2, if they are mergeable w.r.t. $\rightsquigarrow$, then $merge(S_1, S_2, \rightsquigarrow)$ is an nw-shape. Furthermore, if S_1 and S_2 are ground, then also $merge(S_1, S_2, \rightsquigarrow)$ is ground.*

Proof Let $S = merge(S_1, S_2, \rightsquigarrow)$. Denote $S_i = (\sigma_i, \rightsquigarrow_i, \ell_i, \tau_i)$ with $\sigma_i = (V_i, \rightarrow_i, \frown_i)$ for $i \in [1, 2]$. Additionally, let $S = (\sigma, \rightsquigarrow, \ell, \tau)$, where $\sigma = (V, \rightarrow, \frown)$. We start showing the first claim of the lemma, namely, S is an nw-shape.

From Definition 8, we deduce that $V = (V_1 \cup V_2)$, $(V, \rightsquigarrow)$ is a line graph, and $\rightarrow = (\rightarrow_1 \cup \rightarrow_2)$. As S_1 and S_2 are mergeable, it follows that $\rightarrow_1, \rightarrow_2 \subseteq \rightsquigarrow$ (as per Part 2 of the mergeability definition in Definition 8). Consequently, $\rightarrow \subseteq \rightsquigarrow$. Therefore, Part 1 of Definition 5, which defines nw-shapes, holds for S.

Parts 2 and 3 of Definition 5 clearly follow from the construction of S.

Part 4 of Definition 5 is also satisfied for S. This is a direct consequence of $\frown = \frown_1 \cup \frown_2$ (as dictated by the definition of merge) and Part 4 of Definition 8.

From Part 5 of Definition 8, we observe that $\tau_1(u) = \tau_2(u)$ and $\ell_1(u) \cap \ell_2(u) = \emptyset$ for each $u \in V_1 \cap V_2$. Additionally, by definition, τ agrees with τ_1 and τ_2. Furthermore, ℓ agrees with ℓ_1 and ℓ_2 on the vertices that are not in common between S_1 and S_2, and join them on the remaining ones. Consequently, the consistency of the labeling of S (as defined in Part 5 of Definition 5) is ensured by the consistency of the labelings of S_1 and S_2. This concludes the proof that indeed S is an nw-shape.

To conclude the proof of the lemma, we now prove the second assertion of the lemma statement, namely, if S_1 and S_2 are ground, then so is S. To achieve this, we need to prove two conditions:

1. $\frown$ is a matching relation, and
2. if $u \frown v$, then $\mathtt{m} \in \ell(u) \cap \ell(v)$ (see Definition 5).

Since we have already shown that S is an nw-shape, we know that $\frown$ is a loose matching relation. Thus, our focus is on demonstrating that the edges in $\frown$ neither share endpoints nor connect a vertex to itself. As $\frown = (\frown_1 \cup \frown_2)$, and by hypothesis $\frown_i$ is a matching relation for $i \in [1, 2]$, the second property naturally holds. The first property could only be violated

by two edges, one coming from $\curvearrowright_1$, and the other from $\curvearrowright_2$. However, such a violation is not possible due to Part 5 of the mergeability definition (Definition 8). According to this part, vertices common to both S_1 and S_2 must have disjoint labeling through ℓ_1 and ℓ_2. Consequently, a vertex serving as an endpoint for an edge in $\curvearrowright_1$ and another in $\curvearrowright_2$ can be marked with $\mathtt{m}$ only by one of ℓ_1 and ℓ_2. This would either contradict part 5.d/e of Definition 5 or the assumption that both S_1 and S_2 are ground. Therefore, S must be ground, and this concludes the proof of the lemma. □

The merge operation is associative in the following sense. By fixing a relation $\rightsquigarrow$ of the line graph of the resulting nw-shape, we obtain this nw-shape regardless of the order in which we merge the initial nw-shapes provided that the merging is carried out w.r.t. relations that are contractions of $\rightsquigarrow$. This clearly holds as mergeability necessitates that the nw-shapes have disjoint edge relations and possess compatible line graphs and vertex labelings.

Lemma 9 *For each $i \in [1,3]$, let S_i be an nw-shape with set of vertices V_i. Furthermore, let $\rightsquigarrow$ be the edge relation such that $(V_1 \cup V_2 \cup V_3, \rightsquigarrow)$ is a line graph. The following equality holds:*

$$\mathit{merge}\left(\mathit{merge}\big(S_1, S_2, \rightsquigarrow_{(V_1 \cup V_2)}\big), S_3, \rightsquigarrow\right) \;=\; \mathit{merge}\left(S_1, \mathit{merge}\big(S_2, S_3, \rightsquigarrow_{(V_2 \cup V_3)}\big), \rightsquigarrow\right),$$

when all the merge operations involved, on at least one side of the equality, are well-defined.

Proof We only examine the scenario in which all the merge operations on the left-hand side of the equality are all well-defined. A proof for the converse direction can be achieved by using similar arguments and is consequently omitted for brevity. We start proving that

1. both the merge operations on the right-hand side of the equality are well-defined,
2. subsequently, we show that the nw-shape S_R defined by the expression on the right-end side of the equality coincides with S_L, i.e., the one defined on the left-hand side.

For the proof of the first claim, we focus only on the mergeability of S_2 and S_3, as the same arguments apply to both the merge operations. Denote $S_i = (\sigma_i, \rightsquigarrow_i, \ell_i, \tau_i)$ with $\sigma_i = (V_i, \rightarrow_i, \curvearrowright_i)$ for $i \in [1,3]$. Additionally, consider $S_{12} = \mathit{merge}\left(S_1, S_2, \rightsquigarrow_{(V_1 \cup V_2)}\right)$, where $S_{12} = (\sigma_{12}, \rightsquigarrow_{12}, \ell_{12}, \tau_{12})$ and $\sigma_{12} = (V_{12}, \rightarrow_{12}, \curvearrowright_{12})$. Under the assumption that the merge operations on the left-hand side are well-defined, we have the following properties:

A. $\rightsquigarrow_1, \rightsquigarrow_2, \rightsquigarrow_3 \subseteq \rightsquigarrow$;

B. the linear edges of S_{12} and S_3 are disjoint, i.e., $(\rightarrow_{12} \cap \rightarrow_3) = \emptyset$;

C. the union $\curvearrowright_{12} \cup \curvearrowright_3$ of the loose matching relations of S_{12} and S_3 is a loose matching relation w.r.t. $\rightsquigarrow$;

D. the labelings of S_{12} and S_3 are compatible, i.e., $\tau_{12}(u) = \tau_3(u)$ and $\ell_{12}(u) \cap \ell_3(u) = \emptyset$ for each $u \in V_{12} \cap V_3$.

We now provide a rationale for each of the parts of the definition of mergeability (refer to Definition 8), demonstrating that they hold for S_2 and S_3 w.r.t. $\rightsquigarrow$. This will conclude the proof of the first claim. Part 1 of the definition of mergeability directly follows from the aforementioned Property A. From Property B above, and the fact that the linear edges of S_{12} are defined as $\rightarrow_1 \cup \rightarrow_2$ (refer to Definition 8), it follows that $\rightarrow_2$ and $\rightarrow_3$ must be disjoint. Consequently, Part 2 of the mergeability definition also holds. Given that $\curvearrowright_{12}=$

$(\curvearrowright_1 \cup \curvearrowright_2)$ by definition (see Definition 8), it follows that $(\curvearrowright_{12} \cup \curvearrowright_3)$ is $(\curvearrowright_1 \cup \curvearrowright_2 \cup \curvearrowright_3)$. Considering the fact that any subset of a loose matching relation is itself a loose matching relation, Property C implies that Part 3 of the mergeability definition holds. Furthermore, as $V_{12} = V_1 \cup V_2$ by definition (see Definition 8), we get that $V_2 \cap V_3 \subseteq V_{12} \cap V_3$. Consequently, leveraging Property D, we conclude that $\tau_2(u) = \tau_3(u)$ and $\ell_2(u) \cap \ell_3(u) = \emptyset$ for each $u \in V_2 \cap V_3$. This, in turn, establishes the validity of Part 4 of the mergeability definition, thereby completing the proof of the first claim.

Now, we proceed to establish the second claim, demonstrating that S_L and S_R coincide. To begin, we note that by Lemma 8, S_L and S_R are nw-shapes. Furthermore, the underlying multigraphs are obtained by performing set unions on the vertices and the edges of the multigraphs underlying each nw-shape S_i. Consequently, by the associativity of the set union, we immediately have that those multigraphs coincide. As the labeling is solely determined by the resulting multigraphs, labeling consistency is also guaranteed between S_L and S_R. Lastly, since we use the same final relation $\rightsquigarrow$ on both sides and we have shown that the first claim holds, we thus get that $S_L = S_R$, thereby concluding the proof of the lemma. □

The following lemma states that in an expression where we compose nw-shapes by the operations of contraction and merge we can always move the merge operations outside and achieve an equivalent expression. For example, the nw-shape in Fig. 5(h) is the result of merging the contraction of the two nw-shapes in Fig. 5(c-d) or by contracting the merge of the same two nw-shapes.

The intuition behind this lemma is twofold: (1) a merge operation glues two nw-shapes at vertices that are not fully expanded and (2) a contract operation condenses connected portions of the nw-shape by abstracting away edges and vertices that are fully expanded. These characteristics ensure that the mergeability of two nw-shapes is preserved on the contracted nw-shapes.

Lemma 10 *For each $i \in [1, 2]$, let S_i be an nw-shape with set of vertices V_i, and let $U_i \subseteq V_i$ be such that S_i is contractable w.r.t. U_i. Furthermore, let $\rightsquigarrow$ be an edge relation such S_1 and S_2 are mergeable w.r.t. $\rightsquigarrow$. It holds that:*

$$\mathit{merge}\Big(\mathit{contract}(S_1, U_1), \mathit{contract}(S_2, U_2), \rightsquigarrow_{(U_1 \cup U_2)}\Big) = \mathit{contract}\big(\mathit{merge}(S_1, S_2, \rightsquigarrow), U_1 \cup U_2\big),$$

and all the merge and contraction operations involved are well-defined.

Proof We concentrate specifically on the scenario where all merge and contract operations on the left-hand side of the equation are well-defined. The proof for the reverse direction can be established using analogous arguments and we omit it for brevity.

We start examining the expression on the left-hand side of the equality. It is important to emphasize that U_i must contain all the vertices of S_i that have yet to be fully expanded (see Definition 6).

We recall that the contraction of S_i w.r.t. U_i absorbs all the vertices of S_i not included in U_i, which by definition must be fully expanded. In particular, the operation compresses linearly connected portions of S_i into single $\rightarrow$-edges without modifying the labeling of the surviving vertices. During this process, some $\curvearrowright$-edges may have their endpoints reassigned if they are part of a compressed portion, or they may vanish if the edge lies entirely within one such portion.

Turning our attention to the expression on the right-hand side of the equality, we merge S_1 and S_2 into an nw-shape, say S_{12}, through set union operations on its vertices and edges, and subsequently adjust the vertex labeling accordingly. Notably, fully expanded vertices of S_1 and S_2 remain so in S_{12}. However, more vertices may become fully expanded during the merge. Nevertheless, those vertices must be contained in $U_1 \cup U_2$ by contractability, and therefore these newly fully expanded vertices would not be absorbed in the subsequent contract operation applied to S_{12}.

To complete the proof, we first list three facts easily derived from the definitions provided earlier in this section. Assuming that V_i is the set of vertices of the sub-nested-word underlying S_i, we have the following:

- Vertices absorbed in the contraction $\mathit{contract}(S_i, U_i)$ are all and only those in $(V_i \setminus U_i)$, all of which need to be fully expanded in S_i.
- Vertices that are common to S_1 and S_2 must be in $(U_1 \cap U_2)$. Otherwise S_1 and S_2 would not be mergeable w.r.t $\rightsquigarrow$, or either one of them would not be an nw-shape by containing an isolated vertex.
- A fully expanded vertex in S_{12} is either fully expanded in S_1 or S_2 (but never in both), or it belongs to $(U_1 \cap U_2)$ (this is the case when it is not fully expanded in both S_1 and S_2 and in the merging all the expected incident edges are present either in S_1 or S_2).

As a consequence of the listed facts, and considering that the vertices in $(U_1 \cup U_2)$ are not deleted in the contraction of S_{12} w.r.t. $(U_1 \cup U_2)$, the portions of S_{12} that are contracted are all and only those contracted in the separate contractions of S_i w.r.t. U_i for $i \in [1, 2]$. This justifies that performing the merge first and a contraction afterward would not change the final result. □

5 Trees of Nested-Word Shapes

In this section, we define *merge-and-contract trees*, full binary labeled trees where every node carries a label corresponding to a nested-word shape. The name comes from the property that the label of each node is related with those of its children by merge and contract operations. The entire tree serves as a representation for a complete nested-word or a fragment thereof. We also demonstrate that each merge-and-contract tree encodes not only the multigraph itself but also a tree decomposition for it.

Merge-and-contract trees assume a central role in the next section, as we encode them within symbolic tree automata. This encoding can then be seamlessly translated into fixed-point algorithms.

We define merge-and-contract trees by applying local constraints to each triple formed by the nw-shapes labeling a node and those labeling its two children. The set of all these triples that satisfy these constraints is characterized as follows.

Definition 9 (Merge-and-Contract Triples) For a positive natural number k, a triple (S_1, S_2, S_3) of nw-shapes is a *k merge-and-contract triple* if each of S_1, S_2 and S_3 has at most k vertices, and there exists an edge relation $\rightsquigarrow$ such that:

$$S_3 = \mathit{contract}\big(\mathit{merge}(S_1, S_2, \rightsquigarrow), V_3\big),$$

where V_3 is the set of vertices of S_3 and the merge and the contraction operations are well-defined. □

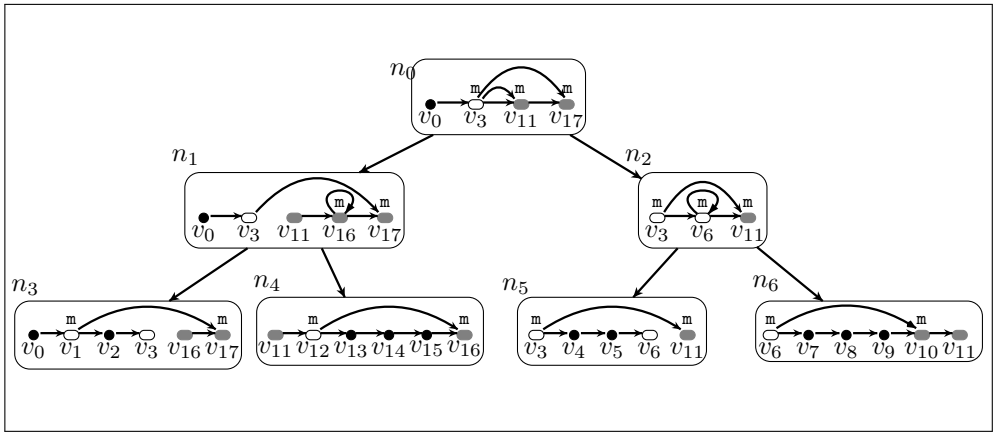

Fig. 6 Example of a merge-and-contract tree for the nested-word illustrated in Fig. 3.

A merge-and-contract tree is formally defined as follows.

Definition 10 (Merge-and-Contract Trees) For a positive natural number k, a k *merge-and-contract tree* is a full binary Γ-tree $\mathcal{T} = (T, \lambda)$, where Γ is the set of all nw-shapes with at most k vertices, such that the following conditions hold:

- if t is a leaf, then $\lambda(t)$ is ground;
- if t is an internal node, then $(\lambda(t0), \lambda(t1), \lambda(t))$ is a k merge-and-contract triple.

Furthermore, $\mathcal{T}$ is *fully expanded* if $\lambda(\epsilon)$ is a fully expanded nw-shape. □

Example 3 Fig. 6 gives a 6 merge-and-contract tree $\mathcal{T}_{ex}$. In fact, all the multigraphs labeling the leaves are ground nw-shapes. The underlying multigraphs are indeed fragments of the nested-word given in Fig. 3. The multigraphs labeling the internal nodes can be obtained by merging and then contracting the nw-shapes of the two children. The merge operations are all w.r.t. the vertex ordering given by increasing indices, and the contractions are w.r.t. the set of vertices of the considered internal node. Note that the nw-shape labeling the root is fully expanded. Moreover, if we remove the edges from the multigraphs labeling the internal nodes and the vertex labeling from all the multigraphs, the resulting tree is exactly the strict augmented tree decomposition of the nested-word from Fig. 3 that we have illustrated in Fig. 4. As we prove later in this section, we can use merge-and-contract trees to generate tree decompositions of any nested-word. □

For the rest of this section, we fix a merge-and-contract tree denoted as $\mathcal{T} = (T, \lambda)$, where each node t is associated with $\lambda(t) = (\sigma_t, \rightsquigarrow_t, \ell_t \tau_t)$ where $\sigma_t = (V_t, \rightarrow_t, \curvearrowright_t)$. Moreover, we assume that the vertices over all the nw-shapes labeling its nodes have the same name if and only if they actually represent the same vertex: for each $t_1, t_2 \in T$, $v_1 \in V_{t_1}$ and $v_2 \in V_{t_2}$, $v_1 \neq v_2$ unless there is a t' that is a common ancestor of t_1 and t_2 such that v_1 (and thus v_2) belongs to V^t for each node t along the paths from t_1 to

t' and from t_2 to t'. Note that this assumption is without loss of generality since we can always impose it by a simple vertex renaming.

Definition 11 (Merge Trees) For a merge-and-contract tree $\mathcal{T} = (T, \lambda)$ and a vertex t of T, we define $\lambda^{merge}(t)$ as the nw-shape obtained only by merging the nw-shapes labeling its children, that is:

- if t is a leaf, then $\lambda^{merge}(t) = \lambda(t)$;
- otherwise, $\lambda^{merge}(t) = merge\big(\lambda^{merge}(t0), \lambda^{merge}(t1), \rightsquigarrow\big)$, where $\rightsquigarrow$ is such that $\rightsquigarrow_{V_{t0} \cup V_{t1}}$ is the edge relation used in the merge of the merge-and-contract triple $(\lambda(t0), \lambda(t1), \lambda(t))$.

We define $G(\mathcal{T})$ as the multigraph labeling the root of $\mathcal{T}$, i.e., $\lambda^{merge}(\epsilon)$. □

Note that λ^{merge} is always defined since mergeability is guaranteed by the fact that the only vertices belonging to both $\lambda^{merge}(t0)$ and $\lambda^{merge}(t1)$ also belong to $\lambda(t0)$ and $\lambda(t1)$.

For the merge-and-contract tree $\mathcal{T}_{ex}$ given in Fig. 6, denoting by ω the nested-word from Fig. 3, λ^{merge} labels the internal nodes as follows: the nw-shape labeling node n_2 corresponds to the sub-nested-word of ω from v_3 through v_{11}, the nw-shape labeling n_1 corresponds to the remaining part of ω, and the one labeling the root n_0 corresponds to the entire ω. In particular, $G(\mathcal{T}_{ex})$ is precisely ω.

The following lemma establishes a connection through the contraction operation between the labeling of the merge-and-contract trees and λ^{merge}. It is essentially a consequence of the properties of the merge and contraction operations we have established in Section 4.

Lemma 11 *Given a merge-and-contract tree $\mathcal{T} = (T, \lambda)$, for each internal node $t \in T$, $\lambda(t) = contract(\lambda^{merge}(t), V_t)$ where V_t is the set of vertices of $\lambda(t)$.*

Proof To prove the lemma, we proceed by structural induction in a bottom-up fashion on the tree. The base case arises when t is a leaf of T. By definition, $\lambda(t)$ equals $\lambda^{merge}(t)$ and thus we need to prove that $\lambda^{merge}(t) = contract(\lambda^{merge}(t), V_t)$. Observe that both $\lambda^{merge}(t)$ and $\lambda(t)$ share identical vertices, namely V_t. Consequently, the application of contraction has no effect, leaving the entire shape unchanged. Thus, the equality between $\lambda^{merge}(t)$ and $contract(\lambda^{merge}(t))$ holds.

For the induction step, we need to prove the equality $\lambda(t.i) = contract(\lambda^{merge}(t.i), V_{t.i})$. The formal steps are given below, followed by explanatory comments:

$$
\begin{aligned}
\lambda(t) &= contract\left(merge(\lambda(t0), \lambda(t1), \rightsquigarrow_t), V_t\right) && \text{(A)}\\
&= contract\left(merge\left(contract(\lambda^{merge}(t0), V_{t0}), contract(\lambda^{merge}(t1), V_{t1}), \rightsquigarrow_t\right), V_t\right) && \text{(B)}\\
&= contract\left(contract\left(merge(\lambda^{merge}(t0), \lambda^{merge}(t1), \rightsquigarrow_t), V_{t0} \cup V_{t1}\right), V_t\right) && \text{(C)}\\
&= contract\left(contract\left(\lambda^{merge}(t), V_{t0} \cup V_{t1}\right), V_t\right) && \text{(D)}\\
&= contract\left(\lambda^{merge}(t), V_t\right). && \text{(E)}
\end{aligned}
$$

We provide a rationale for the derivation of the above steps:

- Equation (A) by the definition of λ.
- Equation (B) by the induction hypothesis.
- Equation (C) by Lemma 10.
- Equation (D) by definition of λ^{merge}.
- Equation (E) by Lemma 5.

Equation (E) provides the desired result, allowing us to conclude the proof. □

As an example of application of the above lemma, consider again the merge-and-contract tree $\mathcal{T}_{ex}$ from Fig. 6. It is simple to verify that the nw-shape S labeling node n_2 is the contraction of the nw-shape $\lambda^{merge}(n_2)$ over the set of vertices of S.

Earlier in this section, we have already observed that $G(\mathcal{T}_{ex})$ is a nested-word and that the nw-shape labeling the root of $\mathcal{T}_{ex}$ is fully expanded. This indeed is not just a coincidence as stated by the following lemma.

Lemma 12 *Given a merge-and-contract tree $\mathcal{T} = (T, \lambda)$, $G(\mathcal{T})$ is a sub-nested-word. Moreover, if $\lambda(\epsilon)$ is a fully expanded nested-word shape, then $G(\mathcal{T})$ is a nested-word.*

Proof By inductively applying Lemma 8, bottom up from the leaves, we get that $\lambda^{merge}(\epsilon)$ is a ground nw-shape. Therefore, directly from the definition of $G(\mathcal{T})$ and Lemma 1 we have that $G(\mathcal{T})$ is a sub-nested-word.

From Lemma 4 and Lemma 11, we get that $\lambda^{merge}(\epsilon)$ is a fully expanded nw-shape. From the proof of the first part of this lemma, we know that $\lambda^{merge}(\epsilon)$ is also ground. Thus, from the definition of $G(\mathcal{T})$ and Lemma 2, we get that $G(\mathcal{T})$ is a nested-word. □

As noted at the end of Example 3, the merge-and-contract tree from Fig 6 corresponds to the strict augmented tree decomposition given in Fig 4(b). In general, it is always possible to construct a merge-and-contract tree corresponding to a given strict augmented tree decomposition as stated in the following lemma.

Lemma 13 *For any augmented strict tree decomposition of a nested-word $\omega = (V, \rightarrow, \curvearrowright)$ of width k, there exists a $k+1$ merge-and-contract tree $\mathcal{T} = (T, \lambda)$ such that the graph $G(\mathcal{T})$ is ω.*

Proof Let $D = (T, \{\omega_t\}_{t \in T})$ be an augmented strict tree decomposition of ω of width k and denote $\omega_t = (V_t^\omega, \rightarrow_t^\omega, \curvearrowright_t^\omega)$.

We define $\mathcal{T}$ as the Γ-tree (T, λ), where Γ is the set of all nw-shapes with at most $k+1$ vertices. The definition of λ is as follows:

- if t is a leaf of T, then $\lambda(t) = (\sigma_t, \rightsquigarrow_t, \ell_t, \tau_t)$ where:
  - $\sigma_t = \omega_t$ and $\rightsquigarrow_t = \rightarrow_t^\omega$,
  - τ_t is such that for every $u \in V_t^\omega$:
    - $\tau_t(u) =$ `call` if there exists a vertex v such that $u \curvearrowright v$;
    - $\tau_t(u) =$ `ret` if there exists a vertex v such that $v \curvearrowright u$;
    - $\tau_t(u) =$ `int` in all the other cases.
  - for every $u \in V_t^\omega$, $\ell_t(u)$ is the smallest set obeying to the following conditions:

* $\mathtt{r} \in \ell_t(u)$ if there exists a vertex v such that $u \to_t^\omega v$;
* $\mathtt{l} \in \ell_t(u)$ if there exists a vertex v such that $v \to_t^\omega u$;
* $\mathtt{m} \in \ell_t(u)$ if there exists a vertex v such that $u \curvearrowright_t^\omega v$ or $v \curvearrowright_t^\omega u$.

- if t is an internal node of T, then $\lambda(t) = \mathit{contract}(\mathit{merge}(\lambda(t0), \lambda(t1), \to_{V_{t0} \cup V_{t1}}), V_t^\omega)$ (recall that for $t \in T$, V_t denotes the set of vertices of $\lambda(t)$).

We now prove that $\mathcal{T}$ is well-defined. To see this we show by structural induction the following stronger claim: for each $t \in T$,

1. the subtree T_t of T rooted at t is a merge-and-contract tree, and
2. the multigraph underlying $\lambda^{merge}(t)$ is the sub-nested-word of ω obtained by the union of the sub-nested-words labeling the leaves of T_t.

The base case, i.e., when t is a leaf, is straightforward: a single node labeled with a ground nw-shape is a merge-and-contract tree by definition, and λ labels leaves with sub-nested-words of ω.

For the induction step, let $t \in T$ be an internal node. Since by definition of strict augmented tree decomposition, $\mathcal{T}$ is a full binary tree, then $t0, t1 \in T$.

By part (2) of the induction hypothesis, $\lambda^{merge}(t.i)$ is the union of the sub-nested-words labeling the leaves of the subtree rooted at $t.i$, for $i \in [1, 2]$. Thus, $\lambda^{merge}(t0)$ and $\lambda^{merge}(t1)$ are edge-disjoint. Thus by Lemma 6, denoting $U_t = \cup_{t' \in T_t} V_{t'}^\omega$, i.e., the union of the set of vertices of all the sub-nested-words labeling the nodes in the subtree T_t, we get that $\lambda'(t) = \mathit{merge}(\lambda^{merge}(t0), \lambda^{merge}(t1), \to_{U_t})$ is well-defined and by Lemma 7, it is the union of the sub-nested-words labeling the leaves of the subtree T_t.

For $i \in [1, 2]$, by part (1) of the induction hypothesis we get that the tree rooted at $t.i$ is a merge-and-contract tree. Thus, by Lemma 11, $\mathit{contract}(\lambda^{merge}(ti), V_{ti}^\omega)$ is well-defined. Therefore, since we have argued above that $\mathit{merge}(\lambda^{merge}(t0), \lambda^{merge}(t1), \to_{U_t})$ is well-defined and $\to_{V_{t0}^\omega \cup V_{t1}^\omega} \subseteq \to_{U_t}$ (recall that $V_{t0}^\omega \cup V_{t1}^\omega \subseteq U_t$ by definition of U_t), by Lemma 10, we get

$$\begin{aligned}&\mathit{contract}(\mathit{merge}(\lambda^{merge}(t0), \lambda^{merge}(t1), \to_{U_t}), V_{t0}^\omega \cup V_{t1}^\omega)\\ &\quad = \mathit{merge}(\mathit{contract}(\lambda^{merge}(t0), V_{t0}^\omega), \mathit{contract}(\lambda^{merge}(t1), V_{t1}^\omega), \to_{V_{t0}^\omega \cup V_{t1}^\omega}).\end{aligned}$$

Again, by Lemma 11, we also get that $\lambda(t.i) = \mathit{contract}(\lambda^{merge}(t), V_{t.i}^\omega)$ for $i \in [1, 2]$, and then by applying also the definition of $\lambda'(t)$, from the above equality we get:

$$\mathit{contract}(\lambda'(t), V_{t0}^\omega \cup V_{t1}^\omega) = \mathit{merge}(\lambda(t0), \lambda(t1), \to_{V_{t0}^\omega \cup V_{t1}^\omega}). \tag{1}$$

Now we observe that V_t^ω contains all the vertices of $(V_{t0}^\omega \cup V_{t1}^\omega)$ that are not fully expanded in $\lambda'(t)$. In fact, assume by contradiction the existence of a not fully expanded vertex v belonging to $(V_{t0}^\omega \cup V_{t1}^\omega)$ that does not belong to V_t^ω. Since v is not fully expanded, there must be an edge of ω that is not covered by any of the multigraphs labeling the leaves of the subtree rooted at t. Thus, this edge must be covered by a multigraph of one of the remaining leaves, and vertex v must be in the bag of such a leaf. However, this would contradict the property of tree decompositions that a vertex must be in all the bags along any path connecting two nodes labeled with two bags containing it. Hence, V_t^ω must indeed contain all the vertices of U_t that are not fully expanded. Therefore, $\lambda'(t)$ is contractable w.r.t. V_t^ω.

Observe that since D is strict, we get that $V_t^\omega \subseteq V_{t0}^\omega \cup V_{t1}^\omega$ must hold. By Lemma 5, we have:

$$\mathit{contract}(\lambda'(t), V_t^\omega) = \mathit{contract}(\mathit{contract}(\lambda'(t), V_{t0}^\omega \cup V_{t1}^\omega), V_t^\omega).$$

Therefore, from equation 1 we also get that $\mathit{contract}(\mathit{merge}(\lambda(t0), \lambda(t1), \to_{V_{t0}^\omega \cup V_{t1}^\omega}), V_t^\omega)$ is well-defined, but this is exactly $\lambda(t)$. Therefore, this conclude the induction for part (1) of our claim, that is, that the subtree rooted at t is a merge-and-contract tree.

To conclude the proof of our claim, we just observe that since we have just shown that the subtree rooted at t is a merge-and-contract tree, we get that λ^{merge} coincides with λ'

and thus the induction step of part (2) of our claim holds, and therefore $\mathcal{T}$ is well-defined and is a $k+1$ merge-and-contract tree (by definition, for each node of T, the set of vertices of the nw-shape associated in $\mathcal{T}$ and of the multigraph associated in D coincide).

By definition, $G(\mathcal{T})$ is $\lambda^{merge}(\epsilon)$. Thus, by part (2) of the above claim, $G(\mathcal{T})$ is exactly ω, which concludes the proof of the lemma. □

From Theorem 2, for each nested word there is a strict augmented tree decomposition of width 2. From Lemma 13, each nested word is encoded by at least a 3 merge-and-contract tree. Conversely, from Lemma 12 each merge-and-contract tree whose root is labeled with a fully expanded nw-shape encodes a nested word. Thus, the following theorem holds:

Theorem 3 *For each nested-word ω and $k \geq 3$, there exists a k fully expanded merge-and-contract tree $\mathcal{T} = (T, \lambda)$ such that $G(\mathcal{T})$ is ω.*

6 Program Verification Methodology

In this section, we delineate our verification methodology for (sequential) programs, with a particular focus on tackling the assertion checking problem introduced in Section 2.

In Section 6.1, we give an extension of multigraphs to represent program computations. Specifically, we utilize *program* nested-words that are nested-words where nodes are labeled with a valuation of program variables and program counters. The linear edges of these nested-words signify consecutive program configurations, providing insights into the interrelation of data annotated at vertices. Meanwhile, the call stack is conventionally captured by nesting edges. We conclude the section by formally asserting that solving the assertion-checking problem for the given class of programs is tantamount to verifying the existence of a program nested-word ending with a vertex whose label unequivocally confirms the failure of a program assertion.

In Section 6.2, we revisit a recently introduced type of tree automata known as symbolic data-tree automata. In Section 6.3, this class proves instrumental in encoding program nested-words, employing a decomposition based on merge-and-contract trees as introduced in Section 5. The overall reduction allows us to reduce the assertion checking problem to checking the emptiness problem for symbolic data-tree automata, that in turn, can be reduced to the satisfiability problem of constrained Horn clauses, for which well-performing tools exist.

6.1 Multigraph Data Structures for Program Computations

In our methodology, we employ an extended form of multigraphs to represent program computations. Each node within the multigraph is annotated with an evaluation of a data signature. Specifically, in the context of modeling the executions of a program P, we consider nested-words with vertices labeled with an evaluation of the data signature $\mathcal{S}_P$.

Data-Multigraphs. A *data-multigraph* with data signature $\mathcal{S}$, called an *$\mathcal{S}$-multigraph*, is formally defined as a pair (G, λ), where G is a multigraph with a set of

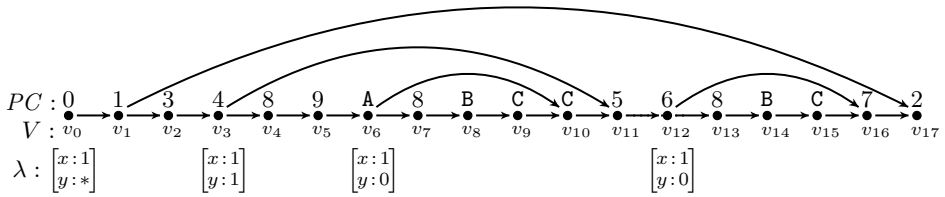

Fig. 7 Program nested-word of a run of the program in Fig. 2.

vertices V, and $\lambda : V \mapsto \mathcal{E}(\mathcal{S})$ is a labeling function that assigns a valuation of $\mathcal{S}$ to each node $v \in V$. To simplify the notation, we denote the value of a field *id* associated with a multigraph vertex v by $v.id$, when the function λ is clear in the context. Moreover, when (G, λ) takes the form of a binary tree, we call it a *data-tree* or $\mathcal{S}$*-tree*.

Program nested-words

We aim to examine program computations by utilizing their behavior graphs—finite graphs that intricately represent the control-flow structure through carefully defined edges. Specifically, we adopt nested-words as our behavior graphs, augmenting them with annotations for the program counter (*pc*) and variable valuations in each state, i.e., an evaluation of $\mathcal{S}_P$. These augmented nested-words are denoted as *program nested-words*. They serve as a comprehensive representation, capturing both the control flow and variable states within the program computations.

Example 4 Fig. 7 gives a visual representation, as a behavior graph, of the computation of the program illustrated in Fig. 2 under the initial condition $x = 1$. The graph is essentially a nested-word where the vertices $v_0, \ldots, v_{17}$ are labeled with their respective program counters (*pc*) derived from the program. Additionally, the vertices of the graph are labeled with variable valuations at the beginning and at each program counter following an assignment. The $\rightarrow$-edges represent the linear progression of the program states along the computation, thereby capturing the *transitions* within the computation. Meanwhile, the $\curvearrowright$-edges connect a vertex corresponding to a procedure call to the vertex corresponding to the statement that follows the call in the current procedure – i.e., the point of execution after returning from the call (*return location*). Consider for example $v_1 \curvearrowright v_{17}$: v_1 corresponds to the state preceding the call to the *boo* procedure from the `main` function (with *pc* 1), and v_{17} corresponds to the state after returning from this call (with *pc* 2). Additionally, there is the transition $v_1 \rightarrow v_2$, where v_2 corresponds to the beginning of the first activation of *boo* (with *pc* 3). □

Below, we give a logical characterization of program nested-words, by using the program template formulas introduced in Section 2.1. For clarity of presentation, we make the assumption that all the procedure calls in the computations are returned. It is worth noting that this assumption is without loss of generality, as we can always append a possibly empty sequence of transitions. These additional transitions, being non-actual program transitions, can be identified as such. This ensures the matching of all pending calls in the call stack.

Formally, we have:

Definition 12 (Program Nested-Words) A *program nested-word* of a program P is a pair (ω, λ) where:

- $\omega = (V, \rightarrow, \curvearrowright)$ is a nested-word, with $V = \{v_0, \ldots, v_n\}$ and $v_{i-1} \rightarrow v_i$ for each $i \in [1, n]$;
- $\lambda : V \mapsto \mathcal{E}(\mathcal{S}_P)$ is a labeling function assigning a valuation of $\mathcal{S}$ to each vertex in V, where $Init_P(\lambda(v_0))$ holds *true*, and for every $u, v, x \in V$:
  - if $u \rightarrow v$, then $Trans_P(\lambda(u), \lambda(v))$ holds *true*;
  - if $u \curvearrowright v$, then $CallRet_P(\lambda(u), \lambda(v))$ holds *true*;
  - if $\lambda(u).pc \in Call_P$, then there exists $y \in V$ such that $u \curvearrowright y$;
  - if $\lambda(v).pc \in Ret$, then there is $y \in V$ such that $y \curvearrowright v$. □

Bridging Executions and Program Nested-Words: Consider a computation $\pi = C_0 C_1 \ldots C_n$ of program P, where $C_i = \langle \nu_i, pc_i, St_i \rangle$ for every $i \in [0, n]$. For each $i \in [0, n]$ with $pc_i \in Call_P$, we define a notion of *matching* between indices. Specifically, we say that index i *matches* index j in π if j is the smallest index greater than i such that $St_j = St_i$. We define $NW(\pi)$ as the pair (ω, λ) where $\omega = (\{v_0, \ldots, v_n\}, \rightarrow, \curvearrowright)$ is a nested-word satisfying the following conditions:

1. $v_{i-1} \rightarrow v_i$ for $i \in [1, n]$,
2. $v_i \curvearrowright v_j$ if and only if i matches j in π, and
3. $\lambda(v_i).state = \nu_i$ and $\lambda(v_i).pc = pc_i$, for every $i \in [0, n]$.

It is straightforward to demonstrate that $NW(\pi)$ is indeed a program nested-word for program P.

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

1. if $\lambda(v_{i-1}).pc \in Call_P$, then $St_i = (\lambda(v_{i-1}).pc) \cdot St_{i-1}$ (procedure call);
2. if $v_j \curvearrowright v_i$ for some j, then $St_{i-1} = (\lambda(v_j).pc) \cdot St_i$ (return from a call);
3. otherwise, $St_i = St_{i-1}$ (internal move).

In this case as well, it is evident that $RUN(pnw)$ constitutes a valid computation of program P.

Thus the following holds:

Theorem 4 *Given a program P, there exists a one-to-one mapping between its computations and the program nested-words of P.*

The above theorem enables us to explore program nested-words for determining the existence of an error computation. Specifically, for a program nested-word $pnw = (\omega, \lambda)$ of P, we say that it *witnesses the failure of an assertion* if its corresponding computation of P is an error one. Thus, as corollary of the above theorem we get:

Theorem 5 *For a given a program P, P has an error computation if and only if there exists a program nested-word of P that witnesses the failure of an assertion.*

6.2 Symbolic Data-Tree Automata

In this section, we define a new class of tree automata called *symbolic data-tree automata*. They generalize traditional bottom-up finite tree automata as they work with data trees. Furthermore, they are symbolic because the alphabet and set of states are defined using evaluations of data signatures, and its transition function is defined through constraints involving states and alphabet.

Definition 13 A **symbolic data-tree automaton** equipped with the data logic $\mathcal{D}$, denoted as SDTA, $\mathcal{A}$ is a quadruple $(\mathcal{S}^{\Sigma}, \mathcal{S}^{Q}, \psi, \psi^{F})$ where:

- $\mathcal{S}^{\Sigma}$ is the *alphabet data signature* defining the alphabet $\Sigma = \mathcal{E}(\mathcal{S}^{\Sigma})$ labeling the input trees;
- $\mathcal{S}^{Q}$ is the *state data signature* defining the set of states $Q = \mathcal{E}(\mathcal{S}^{Q})$; Q is extended to include **nil** for missing child nodes, forming $\widehat{Q} = Q \cup \{\textbf{nil}\}$.
- $\psi(q_l, q_r, val, q)$ denotes a $\mathcal{D}$-formula involving the free variables q_l, q_r of type $\widehat{Q}$, q of type Q, and val of type $\mathcal{S}^{\Sigma}$. This formula represents the *transition function* of $\mathcal{A}$;
- $\psi^{F}(q)$ is a $\mathcal{D}$-formula on the free variable q of type $\mathcal{S}^{Q}$defining the set of *final states* $F \subseteq Q$, i.e., the set consisting of all elements $q \in Q$ such that $\psi^{F}(q)$ holds to *true*.

A binary $\mathcal{S}^{\Sigma}$-tree (T, λ) is *accepted* by $\mathcal{A}$ if there exists a total function $\pi : T \mapsto Q$, known as a *run*, such that the following conditions hold:

- $\psi^{F}\big(\pi(\epsilon)\big)$ holds; and
- for all nodes $t \in T$, $\psi\big(q_1, q_2, \lambda(t), \pi(t)\big)$ holds, where for all $j \in [1, 2]$: q_j is $\pi(t.j)$ if $t.j \in T$, otherwise **nil**.

The *language* of $\mathcal{A}$, denoted as $L(\mathcal{A})$, is the set of all accepted $\mathcal{S}^{\Sigma}$-trees. We recover standard tree automata when both data signatures $\mathcal{S}^{\Sigma}$ and $\mathcal{S}^{Q}$ are enumerations. In that case, we call $\mathcal{A}$ an *enumeration tree automaton* and we denote it as (Σ, Q, F, Δ), where $\Sigma = L(\mathcal{S}^{\Sigma})$, $Q = L(\mathcal{S}^{Q})$, and so on. □

The emptiness problem for SDTAs asks to determine whether the language $L(\mathcal{A})$ of a given SDTA $\mathcal{A}$ is empty. The following results is a simple generalization of the analogous result for binary SDTAs presented in [3].

Theorem 6 (EMPTINESS PROBLEM) *The emptiness problem for* SDTA*s is undecidable and can be effectively reduced in linear time to the satisfiability problem of a system of constrained Horn clauses.*

6.3 Symbolic Data-Tree Automata for Nested-Word Decompositions

The previous sections provide the foundational elements for our methodology of program analysis through the tree decomposition of (program) nested-words. Here, we refine the verification approach described in Section 6.1. Rather than conducting an exhaustive examination of each nested-word against its entire underlying structure, a process demanding unbounded memory, we propose a more efficient method, that

ultimately uses CHC solvers to perform the analysis. Our proposed approach involves conducting verification by examining distinct parts of the computation incrementally, leveraging the merge-and-contract tree associated with an augmented strict tree decomposition of the underlying nested-word. This offers a key advantage: we keep track of a bounded number of program variable evaluations at each step. This reduces the memory footprint required by our methodology, facilitating a more efficient and streamlined program analysis.

To accomplish this, we employ the symbolic data-tree automata (SDTAs) introduced in Section 6.2. SDTAs serve as a powerful tool for systematically verifying all merge-and-contract trees extended to program nested-words that define the language of program nested-words failing any program assertion. Checking the emptiness of the constructed SDTA gives a way to validate the correctness of the program under consideration via CHC satisfiability. Given that the chain of reductions is amenable to automation, our proposed approach establishes an automatic methodology for the analysis of sequential recursive programs.

We now give a reduction from the assertion checking problem to the emptiness problem for SDTAs.

A variant of merge-and-contract triples

We adopt a variant of the merge-and-contract triples to define our encoding into SDTAs. When defining merge-and-contract trees, we assigned the same names to vertices in the nw-shapes as their counterparts in the resulting multigraph $G(\mathcal{T})$. While this facilitated identifying identical vertices across different nw-shapes, particularly during merge operations, it incidentally creates an unbounded set of merge-and-contract triples despite a constant parameter k.

To address this issue, we propose decoupling vertices from their names. Each nw-shape now uses the same fixed set of k names, but we introduce mappings γ_i specifically for the first two shapes in each triple. These mappings indicate, for each vertex a in the first two nw-shapes, which vertex it corresponds to in the third nw-shape. Crucially, this approach ensures the set of valid triples to be bounded regardless of vertex names.

Formally, a triple is represented as $((S_1, \gamma_1), (S_2, \gamma_2), S_3)$, where:

- each nw-shape S_i has vertex set $V_i \subseteq \{v_1, v_2, \ldots, v_k\}$, for $i \in [1, 3]$;
- $\gamma_i : V_i \mapsto V_3$, is a partial map indicating the corresponding vertex $\gamma_i(v)$ in S_3 for vertex v in S_i, for every $i \in [1, 2]$.

By applying vertex renaming aligned with the γ_i maps and adhering to merge and contract definitions, we define the set of all valid triples as MCT_k.

The reduction to the SDTA emptiness

In this section, given a program P and a bound $k > 0$, we construct an SDTA $\mathcal{A}_{P,k}$ that accepts merge-and-contract trees which correspond to executions of P ending into a configuration that violates an assertion. According to the results we have shown, this amounts to reducing the assertion violation problem for sequential programs to checking the emptiness for $\mathcal{A}_{P,k}$ whenever $k \geq 3$ (due to Theorem 1).

The SDTA $\mathcal{A}_{P,k}$ is $(\mathcal{S}^{\Sigma}_{P,k}, \mathcal{S}^{Q}_{P,k}, \psi_{P,k}, \psi^{F}_{P,k})$ whose components are defined as follows.

$\mathcal{A}_{P,k}$ accepts $\mathcal{S}^{\Sigma}_{P,k}$-labeled trees. Denote by $V_k = \{v_1, v_2, \ldots, v_k\}$ a set of k vertices. The fields of $\mathcal{S}^{\Sigma}_{P,k}$ are:

Shape: An enumerated field *shape* ranging over Shapes_k, the set of all possible nw-shapes over a subset of vertices from V_k.

Program state tracker: For each $v_i \in V_k$, a field state_i of type $\mathcal{S}_P$ (see Section 2) to track the valuation of program variables and the program counter associated with the vertex v_i.

The state data signature $\mathcal{S}^{Q}_{P,k}$ coincides with the alphabet data signature $\mathcal{S}^{\Sigma}_{P,k}$. Indeed, the state of the automaton just stores the label of the current input node.

In order to define $\psi_{P,k}$, we first introduce the following auxiliary formula $\mathit{VarsConsistent}(S, \mathit{state}_1, \ldots, \mathit{state}_k)$, where $S = (V', \rightarrow, \curvearrowright)$ is an nw-shape in Shapes_k and hence $V' \subseteq V_k$:

$$\bigwedge_{v_i \rightarrow v_j} \mathit{Trans}_P(\mathit{state}_i, \mathit{state}_j) \wedge \bigwedge_{v_i \curvearrowright v_j} \mathit{CallRet}_P(\mathit{state}_i, \mathit{state}_j).$$

The transition predicate $\psi_{P,k}$ can now be defined as:

$$\psi_{P,k}(q_l, q_r, \mathit{val}, q) \stackrel{\text{def}}{=} \Big((q = \mathit{val}) \wedge \Big(\psi_{\mathit{leaf}}(q_l, q_r, \mathit{val}, q) \vee \psi_{\mathit{int}}(q_l, q_r, \mathit{val}, q) \Big) \Big)$$

where:

$\psi_{\mathit{leaf}}(q_l, q_r, \mathit{val}, q)$ is a formula that evaluates to *true* on all leaves, i.e., nodes where both of its children are **nil**, and if its associated nw-shape is ground and has size at most k while being data consistent, as defined below:

$$(q_l = \mathbf{nil}) \wedge (q_r = \mathbf{nil})$$
$$\wedge \left(\bigvee_{S \in \mathit{Ground}_k} \big(q.\mathit{shape} = S \wedge \mathit{VarsConsistent}(S, q.\mathit{state}_1, \ldots, q.\mathit{state}_k) \big) \right)$$

where Ground_k is the subset of all the ground nw-shapes from Shapes_k.

$\psi_{int}(q_l, q_r, val, q)$ is a formula designed for internal nodes, wherein the nw-shape of its label, combined with those extracted from its children, constitute merge-and-contract triples. Furthermore, it ensures that vertices shared among the involved nw-shapes have identical associated data, as specified below:

$$
\begin{aligned}
&(q_l \neq \textbf{nil}) \wedge (q_r \neq \textbf{nil}) \\
&\wedge \Bigg(\bigvee_{((S_1,\gamma_1),(S_2,\gamma_2),S_3) \in MCT_k} \Big(q_l.shape = S_1 \wedge q_r.shape = S_2 \wedge q.shape = S_3 \\
&\qquad\qquad \wedge \bigwedge_{v_j = \gamma_1(v_i)} (q_l.state_i = q.state_j) \\
&\qquad\qquad \wedge \bigwedge_{v_j = \gamma_2(v_i)} (q_r.state_i = q.state_j) \qquad\qquad \Big) \Bigg)
\end{aligned}
$$

Finally, let $\psi^F_{P,k}(q)$ be defined as $ErrorShape(q)$, where $ErrorShape(q)$ yields *true* if and only if:

- $q.shape$ represents a fully expanded nested-word shape, and
- if v_{i_1} and v_{i_2} correspond to the initial and final vertices of $q.shape$ according to the linear ordering induced by the $\rightarrow$-edges of $q.shape$, then both $Init_P(q.state_{i_1})$ and $AssertionFail_P(q.state_{i_2})$ must evaluate to *true*.

We now conclude the section with the main result. By induction, it is direct to verify that $\mathcal{A}_{P,k}$ essentially checks the following:

- the tree obtained from an accepted tree by retaining only the nw-shapes of its labeling is indeed a merge-and-contract tree;
- the nw-shape labeling the root is fully expanded (checked by the acceptance condition $\psi^F_{P,k}$), and therefore the tree defines a nested-word (by Lemma 12);
- the labeling given by the program state tracker field along with the above nested-word forms a program nested-word of P;
- such a program nested-word indeed witnesses the failure of an assertion (checked by the acceptance condition $\psi^F_{P,k}$).

The size of $\mathcal{A}_{P,k}$ depends essentially on the length of the formulas $\psi_{P,k}$ and $\psi^F_{P,k}$. The first formula is linear in the size of P and in $2^{O(k \log k)}$, where the logarithmic term arises from the enumeration of the mappings γ_1, γ_2 within the merge-and-contract triples. The second one is linear in the size of P and in k. Thus, overall the size of $\mathcal{A}_{P,k}$ is linear in the size of P and exponential in $k \log k$. We recall that for sequential programs it suffices to pick $k = 3$. Therefore, by Theorem 5 we get the following:

Theorem 7 *Let P be a program and $k \geq 3$. It holds that: $L(\mathcal{A}_{P,k}) \neq \emptyset$ if and only if there is a computation of P that violates an assertion. Moreover, the size of $\mathcal{A}_{P,k}$ is linear in the size of P and exponential in $k \log k$.*

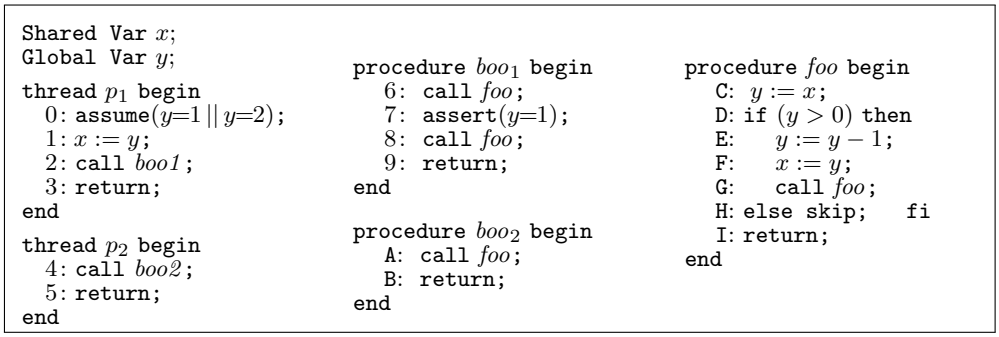

Fig. 8 A sample concurrent program.

7 Extension to Concurrent Programs

The verification methodology we have introduced in the previous sections, built on the principle of exploring program computations through behavioral graph tree decompositions, is well-suited for sequential programs with recursive procedure calls. In this section, we demonstrate how this foundational principle can be exploited also for the verification of concurrent programs by extending our methodology to this class of programs. We provide sufficient detail to guide further elaboration.

We begin by presenting the class of concurrent programs and its formal semantics. Next, we introduce the class of behaviour graphs specifically designed for this program category. We then delve into the appropriate shape characteristics for these behaviour graphs, analyzing the concepts of merge and contraction operations on these shapes. Notably, these definitions share significant overlap with those established for sequential programs. The remaining parts follow a similar structure as those for sequential programs and will be discussed briefly.

Concurrent programs

Concurrent programs consist of a finite number of threads, each defined by a sequential program. These threads execute in parallel, potentially overlapping in their execution, and communicate through a finite number of shared variables according to the Sequential Consistency memory model (SC) [18]. It is assumed that the global variables of each thread have names distinct from the shared variables. In addition to the statements presented in Fig. 1, each thread incorporates two new types of statements, expressed as:

$$g := s \mid s := g.$$

The first statement represents the assignment of a shared variable s into a global variable g, referred to as a *read operation*. The second statement, is a *write operation*, and involves the assignment of a global variable g into a shared variable s. These statements are unique in that they are the only ones involving both global and shared variables. The semantics of concurrent programs can thus be obtained by that of the sequential threads, and the notion of computation is obtained by interleaving the computations of the component threads. Each maximal sequence of consecutive statements of a single thread is called a *context*.

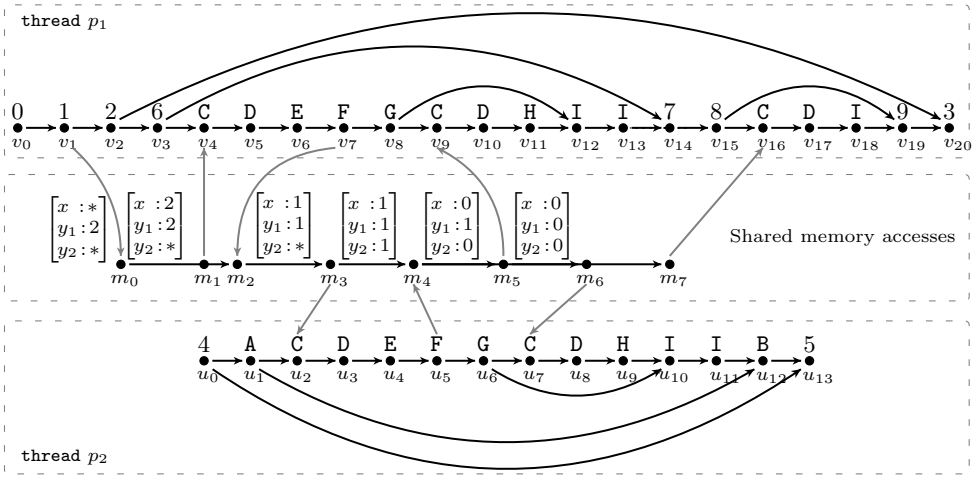

Fig. 9 The concurrent nested-word corresponding to a run of the program in Fig. 8.

Fig. 8 illustrates an example of a concurrent program with two threads p_1 and p_2. The threads share a variable x and each of them has a private copy of the global variable y, denoted by y_1 and y_2 respectively. The main procedure of each thread is denoted by the keyword `thread`. A possible execution π starts in a state where $y_1 = 2$ holds (in the execution that we are going to describe, the starting value of the other two variables is irrelevant). Then, thread p_1 executes its statements up to the `assert` statement at program counter 7. Note that the conditions of the `assume` statement at program counter 0 and of the `assert` statement at program counter 7 are fulfilled. Moreover, at the end of this first context, variables x and y_1 both evaluate to 1. Then, the control passes to thread p_2 that starts executing its statements up to the assignment of the shared variable at program counter `F`. Note that y_2 is assigned with x that evaluates 1 at program counter `C` and therefore the condition at the next `if` statement holds. Moreover, y_2 gets decremented and the resulting value is assigned back to x, which thus evaluates to 0 at the end of this context. The rest of the computation contains one more context for each thread. In particular, thread p_1 resumes from the call to *foo* at program counter G. The call starts by assigning y_1 with x and thus the condition at the following `if` statement this time does not hold, which causes exiting immediately from this call, from the previous call to the same procedure, and from the call to boo_1, going back up to the `return` statement of the main procedure. Similarly, thread p_2 also concludes all the pending calls up to the `return` statement of the main procedure.

Behavior graphs for concurrent programs

A natural graph representation for modeling the computations of concurrent programs is as follows. The behavior of each thread is modeled with a nested-word, while the shared memory's behavior 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 conventionally labeled with a program counter and a valuation of the global variables. In contrast, memory vertices are labeled with a valuation of the shared variables. A vertex of a nested-word whose local transition involves "reading" a shared variable is connected through a *memory edge* from 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 v corresponds to exactly one memory operation, v has exactly one memory edge incident on it. Importantly, memory edges do not cross w.r.t. temporal events (as we assume SC). We call these behavior graphs *concurrent nested-words* (*cnw*).

An example of a concurrent nested-word is shown in Fig. 9. Note that this is the concurrent nested word corresponding to the computation π described above in this section. We extend the notation used for nested-words by adding gray arrows to denote memory edges. For convenience, the evaluations of all the variables are given all together and label the vertices of the memory portion of the graph. As usual, we omit them when none of the variables changes its value.

Concurrent nested-word shapes along with merge and contract operations

Concurrent nested-word shapes provide a mechanism for creating natural summaries that effectively capture the inherent composition of nw-shapes. A *concurrent nw-shape* (*cnw-shape* for short) is formed by a distinct nw-shape for each nested-word component, along with an additional *memory-shape* that is an nw-shape without matching edges. To delve deeper into the specifics of cnw-shapes, we enhance the labeling map $\tau : V \mapsto \{\texttt{int}, \texttt{call}, \texttt{ret}\}$ by incorporating two supplementary labels, `read` and `write`, specifically designed for annotating nested-word vertices involved in memory operations. The mapping $\ell : V \mapsto 2^{\{\texttt{l},\texttt{r},\texttt{m}\}}$ remains unchanged. However, we extend the use of `m` to encompass memory nodes as well. In the integration of these components, we impose a key constraint: memory edges must not cross.

A *contraction operation*, as applied to nw-shapes, has the capability to absorb only fully expanded vertices. The contraction operation for cnw-shape vertices works similarly to the contraction operation for nw-shape in isolation, with the added nuance that a memory edge incident on a vertex being absorbed is inherited by the first vertex not contracted as we traverse backward along the chain formed by $\rightarrow$-edges, originating from the absorbed node. The contraction of a memory vertex operates similarly, with the main difference being the absence of matching edges in this context. The treatment of memory edges during the contraction operation aids us in determining, during a merge operation, whether there would be a crossing memory operation in the original multigraph represented in a merge-and-contract tree.

As in the case of nw-shapes, the *merge* operation is defined to achieve a multigraph union. *Mergeability* is defined as expected: (1) the nw-shapes corresponding to the same thread must be mergeable, (2) the memory shapes must be meargeable, and

(3) additionally, we need to ensure that by merging the edges we do not give rise to crossing memory edges.

By defining *cnw-shape merge-and-contract trees* using the same combination of operations to define (nw-shape) merge-and-contract trees, we can establish an analogous of Theorem 3, as follows:

Theorem 8 *For every concurrent nested-word ω and $k \geq 3$, there exists a k fully expanded cnw-shape merge-and-contract tree $\mathcal{T} = (T, \lambda)$ such that $G(\mathcal{T})$ is (isomorphic to) ω.*

In a similar vein, we can define *program concurrent nested-words* by employing two labeling functions, one for vertices of the nested-word components and another for the vertices of the memory components. These elements serve as the foundation for defining an encoding into an SDTA $\mathcal{A}_{P,k}$, enabling us to establish a result akin to Theorem 7, as follows:

Theorem 9 *Let P be a concurrent program, and consider $k \geq 3$. It holds that: $L(\mathcal{A}_{P,k}) \neq \emptyset$ if and only if the assertion checking problem for P admits a positive answer, taking into account all and only the executions of P whose underlying concurrent nested-word has treewidth at most k.*

It is important to note that, for concurrent programs, we can only capture computations up to a specified treewidth, limiting our ability to claim correctness in general. Nevertheless, for programs exhibiting behavior leading to bounded treewidth graphs, we can assert correctness, as demonstrated in specific instances (see, for example, [19, 20]). Even in cases where complete correctness cannot be proven, we can confidently state that our approach explores a broader range of computations compared to other syntactic restrictions imposed on program behavior, such as *bounded context-switches* [21–23], delay-bound [24], *bounded-scope* [25–28], *bounded-phase* [2, 19], and other restrictions such as [20, 29].

8 Related work

Graph representations for executions and Courcelle's theorem

The representation of computations through graphs was initially introduced by Alur and Madhusudan within the realm of visibly pushdown automata [1]. Building upon this foundation, Madhusudan and Parlato [2] extended the concept to a broader class of automata, including multistack automata and distributed automata, where each process is modeled as a pushdown automaton. Notably, in this scenario, the labeling process involves a finite alphabet. In contrast to conventional approaches, our methodology involves associating each vertex with a tuple of elements capable of drawing values from infinite sets. Our approach draws inspiration from the proof of Courcelle's theorem given in [30] and is anchored in the concept of shapes. Significantly, our method utilizes the symbolic version of tree automata known as SDTA, and ultimately CHC solvers [3].

Treewidth of behavior graphs

The verification of concurrent programs/pushdown automata is undecidable, as evidenced by the fact that encoding Turing machines only requires two threads. However, over the past two decades, there has been a considerable body of literature dedicated to studying concurrent automata. Researchers have explored ways to impose some limitations to make this problem decidable and therefore more manageable. This line of research started from the intuition, later supported by an empirical study [31], that concurrency bugs often manifest within few context-switches [32]. This has triggered a series of interesting results both of theoretical and practical interest. Among these very effective from a practical point of view and often challenging from a theretical point of view have been the approaches based on sequentializations (see [21–23, 33] and references therein). From a theoretical standpoint, the primary challenge has been to identify more expressive yet still decidable bounding parameters. We refer to [2, 19, 20, 24, 27, 28, 34, 35] (and references therein) as a sample of this research. Our work aligns with this pursuit as we introduce the width of the tree decomposition as a bounding parameter. Importantly, our methodology has the versatility to encompass other existing approaches, as, to the best of our knowledge, they generate behavior graphs of bounded tree-width. Notably, a comprehensive result in the realm of decidability is the resolution of MSO (Monadic Second-Order Logic) for all MSO-definable classes of graphs with bounded treewidth [2, 36, 37], a generalization of Courcelle/Seese's theorem [38, 39].

Using Constrained Horn Clauses for verification

Our innovative methodology for program verification introduces a distinctive translation of the verification problem, leveraging the satisfiability of constrained Horn clauses (CHCs). In the evolving landscape of verification tool design and implementation, our approach aligns with a notable research stream where CHCs serve as a key intermediate representation [4, 10, 12, 13, 40]. This strategy not only enhances the efficacy of verification tools but also contributes to the broader goal of advancing formal verification techniques. Furthermore, our methodology represents a broadening or extension of the work presented in [41] to encompass programs with data types beyond the Boolean type.

The growing success of CHC-based verification methodologies is supported by the constant improvement of CHC solvers and at the same time fosters future enhancements. Indeed, a number of solvers have been developed and figure in the annual CHC competition [15], including Spacer [5] (currently integrated into Z3 [42]), Golem [43], and Eldarica [44]. The approach introduced in the present paper could be easily adapted to work with any of these solvers as a CHC-solving backend.

Using Constrained Horn Clauses in AI

CHCs have also been applied in the realm of multi-agent reasoning, specifically to verify safety properties of behavior trees [45] and to solve infinite-state games [46]. In the latter paper, the authors show that a significant class of games can be effectively reduced to CHCs, and that such reduction leads to more efficient solutions compared to the approaches that only employ the quantifier-free capabilities of SMT-solvers.

9 Conclusions

We have proposed a novel methodology for automated analysis that works for several classes of programs, leveraging graph representations of their computations. The core concept revolves around restructuring program behavior graphs via tree decompositions of a given width. To achieve this, we have introduced nested-word shapes, which are essentially labeled multigraphs that serve as concise summaries of program behaviors for efficiently computing tree decompositions of nested words. These multigraphs play a crucial role in defining symbolic data-tree automata [3]. These automata capture all possible tree decompositions of program behaviour graphs. Verification of the original program is thus performed by checking the emptiness of the data tree language accepted by these SDTAs. This last task can be efficiently solved through CHC satisfiability. This translates the problem of program verification into an equivalent CHC satisfiability problem, resulting in an effective reduction.

Our approach leads to an under-approximate analysis, meaning it may miss some potential bugs due to the parameterization by the tree decomposition width k. However, increasing k improves accuracy but requires more computational resources. We have presented our methodology for recursive sequential programs in full details, and subsequently extended it to concurrent programs. Notably, our approach has the potential to work for a broader class of programs, including distributed programs, and concurrent programs with weak memory models, e.g., encoding approaches such as [47, 48]. The constrained component of CHCs not only is suitable for handling tree decompositions but also manages various characteristic features related to data handling and types provided by programming languages [5].

We believe that his research can open avenues for enhanced automated program analysis and verification techniques, offering opportunities for further exploration and enhancement of our methodology. In particular, we see the following directions for future work. Our framework is developed for programs with statically allocated memory, handling variables, arrays and structured variables formed by combining these basic data types. Consequently, features of common programming languages related to the static usage of memory — such as casting in the C language — can be seamlessly integrated into our approach through the application of the theory of bitvectors. More complex aspects, such as pointers and dynamic memory allocation, require further investigation. For this, array theory is often used (e.g., [49, 50]), however it can result in complex CHCs. Also, it would be important to investigate the practical impact of our methodology giving particular attention to scalability on large scale programs. Furthermore, another intriguing future research concerns with the development of a more intricate specification language based on the SDTA framework. Extending the expressiveness of the specification language can empower users to express more complex correctness properties (see for example [51]). Investigating how to encode and verify specifications involving intricate temporal constrains possibly with with quantitative measures can contribute to the applicability of our methodology.

Declarations

Competing Interests

On behalf of all authors, the corresponding author states that there is no conflict of interest.

Funding Information

This work was partially supported by INDAM-GNCS 2022-24, AWS 2021 Amazon Research Awards, the MUR project SOP (Securing sOftware Platforms - CUP: H73C22000890001) as part of the SERICS project (Security and Rights in CyberSpace - n. PE00000014 - CUP: B43C22000750006), *Verifica di proprietà di sicurezza nello sviluppo del software* under the Start-up 2022 program funded by the Computer Science Division UNIMOL, the MUR project Future AI Research (FAIR) Spoke 3, and FARB 2022–24 grants of Università degli Studi di Salerno.

Authors' contribution

All the authors contributed equally to this work.

Data Availability Statement

Not applicable.

Research Involving Human and /or Animals

This article does not contain any studies with human participants or animals performed by any of the authors.

Informed Consent

Not applicable.

References

[1] Alur, R., Madhusudan, P.: Adding nesting structure to words. In: Ibarra, O.H., Dang, Z. (eds.) Developments in Language Theory, 10th International Conference, DLT 2006, Santa Barbara, CA, USA, June 26-29, 2006, Proceedings. Lecture Notes in Computer Science, vol. 4036, pp. 1–13. Springer, (2006). https://doi.org/10.1007/11779148_1 . https://doi.org/10.1007/11779148_1

[2] 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, January 26-28, 2011, pp. 283–294. ACM, (2011). https://doi.org/10.1145/1926385.1926419 . https://doi.org/10.1145/1926385.1926419

[3] Faella, M., Parlato, G.: Reasoning about data trees using CHCs. In: Shoham, S., Vizel, Y. (eds.) Computer Aided Verification - 34th International Conference, CAV 2022, Haifa, Israel, August 7-10, 2022, Proceedings, Part II. Lecture Notes in Computer Science, vol. 13372, pp. 249–271. Springer, (2022). https://doi.org/10.1007/978-3-031-13188-2_13

[4] 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 '12, Beijing, China - June 11 - 16, 2012, pp. 405–416. ACM, (2012). https://doi.org/10.1145/2254064.2254112

[5] Gurfinkel, A., Bjørner, N.: The science, art, and magic of constrained Horn clauses. In: 21st International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, SYNASC 2019, Timisoara, Romania, September 4-7, 2019, pp. 6–10. IEEE, (2019). https://doi.org/10.1109/SYNASC49474.2019.00010

[6] Bjørner, N., Gurfinkel, A., McMillan, K.L., Rybalchenko, A.: Horn clause solvers for program verification. In: Beklemishev, L.D., Blass, A., Dershowitz, N., Finkbeiner, B., Schulte, W. (eds.) Fields of Logic and Computation II - Essays Dedicated to Yuri Gurevich on the Occasion of His 75th Birthday. Lecture Notes in Computer Science, vol. 9300, pp. 24–51. Springer, (2015). https://doi.org/10.1007/978-3-319-23534-9_2

[7] Champion, A., Chiba, T., Kobayashi, N., Sato, R.: Ice-based refinement type discovery for higher-order functional programs. J. Autom. Reason. **64**(7), 1393–1418 (2020) https://doi.org/10.1007/s10817-020-09571-y

[8] Fedyukovich, G., Ahmad, M.B.S., Bodík, R.: Gradual synthesis for static parallelization of single-pass array-processing programs. In: Cohen, A., Vechev, M.T. (eds.) Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017, pp. 572–585. ACM, (2017). https://doi.org/10.1145/3062341.3062382

[9] Garoche, P., Kahsai, T., Thirioux, X.: Hierarchical state machines as modular Horn clauses. In: Gallagher, J.P., Rümmer, P. (eds.) Proceedings 3rd Workshop on Horn Clauses for Verification and Synthesis, HCVS@ETAPS 2016, Eindhoven, The Netherlands, 3rd April 2016. EPTCS, vol. 219, pp. 15–28 (2016). https://doi.org/10.4204/EPTCS.219.2

[10] Gurfinkel, A., Kahsai, T., Komuravelli, A., Navas, J.A.: The seahorn verification framework. In: Kroening, D., Pasareanu, C.S. (eds.) Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I. Lecture Notes in Computer Science, vol. 9206, pp. 343–361. Springer, (2015). https://doi.org/10.1007/978-3-319-21690-4_20

[11] Hojjat, H., Konecný, F., Garnier, F., Iosif, R., Kuncak, V., Rümmer, P.: A verification toolkit for numerical transition systems - tool paper. In: Giannakopoulou, D., Méry, D. (eds.) FM 2012: Formal Methods - 18th International Symposium, Paris, France, August 27-31, 2012. Proceedings. Lecture Notes in Computer Science, vol. 7436, pp. 247–251. Springer, (2012). https://doi.org/10.1007/978-3-642-32759-9_21

[12] Kahsai, T., Rümmer, P., Sanchez, H., Schäf, M.: Jayhorn: A framework for verifying java programs. In: Chaudhuri, S., Farzan, A. (eds.) Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part I. Lecture Notes in Computer Science, vol. 9779, pp. 352–358. Springer, (2016). https://doi.org/10.1007/978-3-319-41528-4_19

[13] Kobayashi, N., Sato, R., Unno, H.: Predicate abstraction and CEGAR for higher-order model checking. In: Hall, M.W., Padua, D.A. (eds.) Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011, pp. 222–233. ACM, (2011). https://doi.org/10.1145/1993498.1993525

[14] Matsushita, Y., Tsukada, T., Kobayashi, N.: Rusthorn: Chc-based verification for rust programs. In: Müller, P. (ed.) Programming Languages and Systems - 29th European Symposium on Programming, ESOP 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings. Lecture Notes in Computer Science, vol. 12075, pp. 484–514. Springer, (2020). https://doi.org/10.1007/978-3-030-44914-8_18

[15] De Angelis, E., K., H.G.V.: CHC-COMP 2022: Competition report. In: Hamilton, G.W., Kahsai, T., Proietti, M. (eds.) Proceedings 9th Workshop on Horn Clauses for Verification and Synthesis and 10th International Workshop on Verification and Program Transformation, HCVS/VPT@ETAPS 2022, and 10th International Workshop on Verification and Program TransformationMunich, Germany, 3rd April 2022. EPTCS, vol. 373, pp. 44–62 (2022). https://doi.org/10.4204/EPTCS.373.5 . https://doi.org/10.4204/EPTCS.373.5

[16] Inverso, O., La Torre, S., Parlato, G., Tomasco, E.: Verifying programs by bounded tree-width behavior graphs. In: Malvone, V., Murano, A. (eds.) Multi-Agent Systems - 20th European Conference, EUMAS 2023, Naples, Italy, September 14-15, 2023, Proceedings. Lecture Notes in Computer Science, vol. 14282, pp. 116–132. Springer, (2023). https://doi.org/10.1007/978-3-031-43264-4_8

[17] Manna, Z., Zarba, C.G.: Combining decision procedures. In: Formal Methods at the Crossroads. From Panacea to Foundational Support, 10th Anniversary Colloquium of UNU/IIST, the International Institute for Software Technology

of The United Nations University, Lisbon, Portugal, March 18-20, 2002, Revised Papers. LNCS, vol. 2757, pp. 381–422. Springer, (2002)

[18] Lamport, L.: How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE Trans. Computers **28**(9), 690–691 (1979) https://doi.org/10.1109/TC.1979.1675439

[19] La Torre, S., Madhusudan, P., Parlato, G.: A robust class of context-sensitive languages. In: 22nd IEEE Symposium on Logic in Computer Science (LICS 2007), 10-12 July 2007, Wroclaw, Poland, Proceedings, pp. 161–170. IEEE Computer Society, (2007). https://doi.org/10.1109/LICS.2007.9

[20] Atig, M.F., Bollig, B., Habermehl, P.: Emptiness of ordered multi-pushdown automata is 2etime-complete. Int. J. Found. Comput. Sci. **28**(8), 945–976 (2017) https://doi.org/10.1142/S0129054117500332

[21] 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, June 9-11, 2004, pp. 14–24. ACM, (2004). https://doi.org/10.1145/996841.996845 . https://doi.org/10.1145/996841.996845

[22] 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

[23] La Torre, S., Madhusudan, P., Parlato, G.: Reducing context-bounded concurrent reachability to sequential reachability. In: Bouajjani, A., Maler, O. (eds.) Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings. Lecture Notes in Computer Science, vol. 5643, pp. 477–492. Springer, (2009). https://doi.org/10.1007/978-3-642-02658-4_36 . https://doi.org/10.1007/978-3-642-02658-4_36

[24] 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, January 26-28, 2011, pp. 411–422. ACM, (2011). https://doi.org/10.1145/1926385.1926432 . https://doi.org/10.1145/1926385.1926432

[25] La Torre, S., Napoli, M.: Reachability of multistack pushdown systems with scope-bounded matching relations. In: Katoen, J., König, B. (eds.) CONCUR 2011 - Concurrency Theory - 22nd International Conference, CONCUR 2011, Aachen, Germany, September 6-9, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6901, pp. 203–218. Springer, (2011). https://doi.org/10.1007/978-3-642-23217-6_14 . https://doi.org/10.1007/978-3-642-23217-6_14

[26] 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, December 15-17, 2012, Hyderabad, India. LIPIcs, vol. 18, pp. 173–184. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, (2012). https://doi.org/10.4230/LIPICS.FSTTCS.2012.173 . https://doi.org/10.4230/LIPIcs.FSTTCS.2012.173

[27] La Torre, S., Napoli, M., Parlato, G.: Scope-bounded pushdown languages. Int. J. Found. Comput. Sci. **27**(2), 215–234 (2016) https://doi.org/10.1142/S0129054116400074

[28] 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

[29] Bouajjani, A., Emmi, M., Parlato, G.: On sequentializing concurrent programs. In: Yahav, E. (ed.) Static Analysis - 18th International Symposium, SAS 2011, Venice, Italy, September 14-16, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6887, pp. 129–145. Springer, (2011). https://doi.org/10.1007/978-3-642-23702-7_13

[30] Flum, J., Grohe, M.: Parameterized Complexity Theory. Texts in Theoretical Computer Science. An EATCS Series. Springer, (2006). https://doi.org/10.1007/3-540-29953-X

[31] Musuvathi, M., Qadeer, S.: Iterative context bounding for systematic testing of multithreaded programs. In: Ferrante, J., McKinley, K.S. (eds.) Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007, pp. 446–455. ACM, (2007). https://doi.org/10.1145/1250734.1250785

[32] Qadeer, S., Rehof, J.: Context-bounded model checking of concurrent software. In: Halbwachs, N., Zuck, L.D. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, 11th International Conference, TACAS 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings. Lecture Notes in Computer Science, vol. 3440, pp. 93–107. Springer, (2005). https://doi.org/10.1007/978-3-540-31980-1_7

[33] Inverso, O., Tomasco, E., Fischer, B., La Torre, S., Parlato, G.: Bounded verification of multi-threaded programs via lazy sequentialization. ACM Trans. Program. Lang. Syst. **44**(1), 1–1150 (2022) https://doi.org/10.1145/3478536

[34] La Torre, S., Napoli, M., Parlato, G.: A unifying approach for multistack pushdown automata. In: Csuhaj-Varjú, E., Dietzfelbinger, M., Ésik, Z. (eds.)

Mathematical Foundations of Computer Science 2014 - 39th International Symposium, MFCS 2014, Budapest, Hungary, August 25-29, 2014. Proceedings, Part I. Lecture Notes in Computer Science, vol. 8634, pp. 377–389. Springer, (2014). https://doi.org/10.1007/978-3-662-44522-8_32

[35] Tomasco, E., Inverso, O., Fischer, B., La Torre, S., Parlato, G.: Verifying concurrent programs by memory unwinding. In: Baier, C., Tinelli, C. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings. Lecture Notes in Computer Science, vol. 9035, pp. 551–565. Springer, (2015). https://doi.org/10.1007/978-3-662-46681-0_52 . https://doi.org/10.1007/978-3-662-46681-0_52

[36] 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, Verona, Italy, September 10-12, 2014. EPTCS, vol. 161, pp. 74–87 (2014). https://doi.org/10.4204/EPTCS.161.9

[37] 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

[38] 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

[39] 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

[40] Gurfinkel, A.: Program verification with constrained Horn clauses (invited paper). In: Shoham, S., Vizel, Y. (eds.) Computer Aided Verification - 34th International Conference, CAV 2022, Haifa, Israel, August 7-10, 2022, Proceedings, Part I. Lecture Notes in Computer Science, vol. 13371, pp. 19–29. Springer, (2022). https://doi.org/10.1007/978-3-031-13185-1_2

[41] 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, June 15-21, 2009, pp. 211–222. ACM, (2009). https://doi.org/10.1145/1542476.1542500 . https://doi.org/10.1145/1542476.1542500

[42] Moura, L.M., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the

Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings. Lecture Notes in Computer Science, vol. 4963, pp. 337–340. Springer, (2008). https://doi.org/10.1007/978-3-540-78800-3_24 . https://doi.org/10.1007/978-3-540-78800-3_24

[43] Blicha, M., Britikov, K., Sharygina, N.: The Golem Horn solver. In: Enea, C., Lal, A. (eds.) Computer Aided Verification - 35th International Conference, CAV 2023, Paris, France, July 17-22, 2023, Proceedings, Part II. Lecture Notes in Computer Science, vol. 13965, pp. 209–223. Springer, (2023). https://doi.org/10.1007/978-3-031-37703-7_10 . https://doi.org/10.1007/978-3-031-37703-7_10

[44] Hojjat, H., Rümmer, P.: The ELDARICA Horn solver. In: Bjørner, N.S., Gurfinkel, A. (eds.) 2018 Formal Methods in Computer Aided Design, FMCAD 2018, Austin, TX, USA, October 30 - November 2, 2018, pp. 1–7. IEEE, (2018). https://doi.org/10.23919/FMCAD.2018.8603013 . https://doi.org/10.23919/FMCAD.2018.8603013

[45] Henn, T., Völker, M., Kowalewski, S., Trinh, M., Petrovic, O., Brecher, C.: Verification of behavior trees using linear constrained horn clauses. In: Groote, J.F., Huisman, M. (eds.) Formal Methods for Industrial Critical Systems - 27th International Conference, FMICS 2022, Warsaw, Poland, September 14-15, 2022, Proceedings. Lecture Notes in Computer Science, vol. 13487, pp. 211–225. Springer, (2022). https://doi.org/10.1007/978-3-031-15008-1_14 . https://doi.org/10.1007/978-3-031-15008-1_14

[46] Faella, M., Parlato, G.: Reachability games modulo theories with a bounded safety player. In: Williams, B., Chen, Y., Neville, J. (eds.) Thirty-Seventh AAAI Conference on Artificial Intelligence, AAAI 2023, Thirty-Fifth Conference on Innovative Applications of Artificial Intelligence, IAAI 2023, Thirteenth Symposium on Educational Advances in Artificial Intelligence, EAAI 2023, Washington, DC, USA, February 7-14, 2023, pp. 6330–6337. AAAI Press, (2023). https://doi.org/10.1609/AAAI.V37I5.25779 . https://doi.org/10.1609/aaai.v37i5.25779

[47] Tomasco, E., Nguyen, T.L., Inverso, O., Fischer, B., La Torre, S., Parlato, G.: Lazy sequentialization for TSO and PSO via shared memory abstractions. In: Piskac, R., Talupur, M. (eds.) 2016 Formal Methods in Computer-Aided Design, FMCAD 2016, Mountain View, CA, USA, October 3-6, 2016, pp. 193–200. IEEE, (2016). https://doi.org/10.1109/FMCAD.2016.7886679 . https://doi.org/10.1109/FMCAD.2016.7886679

[48] Tomasco, E., Nguyen, T.L., Fischer, B., La Torre, S., Parlato, G.: Using shared memory abstractions to design eager sequentializations for weak memory models. In: Cimatti, A., Sirjani, M. (eds.) Software Engineering and Formal Methods - 15th International Conference, SEFM 2017, Trento, Italy, September 4-8, 2017, Proceedings. Lecture Notes in Computer Science, vol. 10469, pp. 185–202. Springer, (2017). https://doi.org/10.1007/978-3-319-66197-1_12 . https://doi.

org/10.1007/978-3-319-66197-1_12

[49] Komuravelli, A., Bjørner, N.S., Gurfinkel, A., McMillan, K.L.: Compositional verification of procedural programs using horn clauses over integers and arrays. In: Kaivola, R., Wahl, T. (eds.) Formal Methods in Computer-Aided Design, FMCAD 2015, Austin, Texas, USA, September 27-30, 2015, pp. 89–96. IEEE, (2015). https://doi.org/10.1109/FMCAD.2015.7542257 . https://doi.org/10.1109/FMCAD.2015.7542257

[50] De Angelis, E., Fioravanti, F., Pettorossi, A., Proietti, M.: Program verification using constraint handling rules and array constraint generalizations. Fundam. Informaticae **150**(1), 73–117 (2017) https://doi.org/10.3233/FI-2017-1461

[51] Faella, M., Parlato, G.: A unified automata-theoretic approach to LTLf modulo theories. In: ECAI 2024 - 27th European Conference on Artificial Intelligence, 19-24 October 2024, Santiago de Compostela, Spain. Frontiers in Artificial Intelligence and Applications. IOS Press, (2024)