A Logic of Reachable Patterns in Linked Data-Structures

Greta Yorsh [a,*,1] Alexander Rabinovich [a] Mooly Sagiv [a]
Antoine Meyer [b] Ahmed Bouajjani [b]

[a]*School of Computer Science, Tel-Aviv University, Tel-Aviv, Israel.*
Email addresses: {gretay,rabinoa,msagiv}@post.tau.ac.il

[b]*LIAFA laboratory, University of Paris 7, Paris, France.*
Email addresses: {ameyer,abou}@liafa.jussieu.fr

Abstract

We define a new decidable logic for expressing and checking invariants of programs that manipulate dynamically-allocated objects via pointers and destructive pointer updates. The main feature of this logic is the ability to limit the neighborhood of a node that is reachable via a regular expression from a designated node. The logic is closed under boolean operations (entailment, negation) and has a finite model property. The key technical result is the proof of decidability.

We show how to express preconditions, postconditions, and loop invariants for some interesting programs. It is also possible to express properties such as disjointness of data-structures, and low-level heap mutations. Moreover, our logic can express properties of arbitrary data-structures and of an arbitrary number of pointer fields. The latter provides a way to naturally specify postconditions that relate the fields on the entry of a procedure to the field on the exit of a procedure. Therefore, it is possible to use the logic to automatically prove partial correctness of programs performing low-level heap mutations.

Key words: Program Verification, Shape Analysis, Heap-manipulating programs, Decidable logic with reachability, Reachability, Routing expression, Pattern, Transitive closure logics, Weak monadic second-order logic
1991 MSC: 03B25, 11U05, 03B45, 03B15, 68N15, 68N30, 68N19

[*] Corresponding author
[1] This research was supported by THE ISRAEL SCIENCE FOUNDATION (grant No 304/03).

1 Introduction

The automatic verification of programs with dynamic memory allocation and pointer manipulation is a challenging problem. In fact, due to dynamic memory allocation and destructive updates of pointer-valued fields, the program memory can be of arbitrary size and structure. This requires the ability to reason about a potentially infinite number of memory (graph) structures, even for programming languages that have good capabilities for data abstraction. Usually abstract-datatype operations are implemented using loops, procedure calls, and sequences of low-level pointer manipulations; consequently, it is hard to prove that a data-structure invariant is reestablished once a sequence of operations is finished [27].

To tackle the verification problem of such complex programs, several approaches emerged in the last few years with different expressive powers and levels of automation, including works based on abstract interpretation [35,46,42], logic-based reasoning [31,43], and automata-based techniques [32,37,8]. An important issue is the definition of a formalism that (1) allows us to express relevant properties (invariants) of various kinds of linked data-structures, and (2) has the closure and decidability features needed for automated verification. The aim of this paper is to study such a formalism based on logics over arbitrary graph structures, and to find a balance between expressiveness, decidability and complexity.

Reachability is a crucial notion for reasoning about linked data-structures. For instance, to establish that a memory configuration contains no garbage elements, we must show that every element is reachable from some program variable. Other examples of properties that involve reachability are (1) data-structure invariants, e.g., the tail of a queue is reachable from the head of a queue, (2) the acyclicity of data-structure fragments, i.e., every element reachable from node u cannot reach u, (3) the property that a data-structure traversal terminates, e.g., there is a path from a node to a sink-node of the data-structure, (4) the property that, for programs with procedure calls when references are passed as arguments, elements that are *not* reachable from a formal parameter are not modified.

A natural formalism to specify properties involving reachability is the first-order logic over graph structures with transitive closure. Unfortunately, even simple decidable fragments of first-order logic become undecidable when transitive closure is added [21,29].

In this paper, we propose a logic that can be seen as a fragment of the first-order logic with transitive closure. Our logic (1) is simple and natural to use, (2) is expressive enough to cover important properties of a wide class of arbitrary linked data-structures, and (3) allows for algorithmic modular verification using programmer-specified loop-invariants, preconditions, and postconditions.

Alternatively, our logic can be seen as a propositional logic with atomic propositions (called reachability constraints) modelling reachability between heap objects pointed-to by program variables and other heap objects with certain properties. The properties are specified using *patterns* that limit the neighborhood of an object.

For example, we can specify the property that an object v is an element of a doubly-linked list using a the pattern $inv_{f,b}$, defined by $(v \xrightarrow{f} w) \Rightarrow (w \xrightarrow{b} v)$. This pattern says that if v has an emanating `forward` pointer f that leads to an object w, then w has a `backward` pointer b into v. Using the pattern $inv_{f,b}$, we can describe a doubly-linked list pointed-to by a program variable x by the atomic proposition $x[\xrightarrow{f}{}^*]inv_{f,b}$ in our logic. This reachability constraint says that any object v reachable from an object pointed-to by x using a (possibly empty) sequence of `forward` pointers satisfies the property $inv_{f,b}$. [2]

The design of our logic is guided by the following principles. First, reachability constraints are closed formulas without quantifier alternations. This guarantees that we are dealing with alternation-free formulas. Second, reachability is expressed via the Kleene star operator. We believe that regular expressions yield a more natural notation than the transitive closure operator. Third, decidability is obtained by syntactically restricting the way patterns are formed. In particular, the use of equality is limited. Semantically, the restriction means that a pattern cannot relate two nodes that are distant from one another, unless these nodes are "named". As a result, a pattern can only describe local properties. Global properties can only be described using reachability along regular paths that start from "named" nodes. Therefore, complex properties can be enforced only between "named" nodes. For example, complex sharing patterns can be created around objects pointed-to by program variables; arbitrary sharing is allowed but cannot be enforced deep in the data-structure, because the objects that are deep are indistinguishable and distant nodes cannot be related by a pattern.

The contributions of this paper can be summarized as follows:

- We define the logic $\mathcal{L}_0$ where reachability constraints such as those mentioned above can be used. Patterns in such constraints are defined by quantifier-free first-order formulas over graph structures and sets of access paths are defined by regular expressions.
- We show that $\mathcal{L}_0$ has a finite-model property, i.e., every satisfiable formula has a finite model. Therefore, invalid formulas are always falsified by a finite store.
- We prove that the logic $\mathcal{L}_0$ is undecidable.
- We define restrictions on the patterns which lead to a fragment of $\mathcal{L}_0$ called $\mathcal{L}_1$.
- We prove that the satisfiability (and validity) problem of $\mathcal{L}_1$-formulas is decidable. The fragment $\mathcal{L}_1$ is the main technical result of the paper and the

[2] This and other examples are explained in detail in Section 4.2.

decidability proof is non-trivial. The main idea is to show that every satisfiable $\mathcal{L}_1$ formula is also satisfied by a tree-like graph. Thus, even though $\mathcal{L}_1$ expresses properties of arbitrary data-structures, because the logic is limited enough, a formula that is satisfied on an arbitrary graph is also satisfied on a tree-like graph. Therefore, it is possible to answer satisfiability (and validity) queries for $\mathcal{L}_1$ using a decision procedure for weak monadic second-order logic (MSO) on trees.

- We show that despite the restriction on patterns we introduce, the logic $\mathcal{L}_1$ is still expressive enough for use in program verification: various important data-structures, and loop invariants concerning their manipulation, are in fact definable in $\mathcal{L}_1$.
- We show that the proof of decidability of $\mathcal{L}_1$ holds "as is" for many useful extensions of $\mathcal{L}_1$.

We define *Logic of Reachable Patterns* (*LRP* for short) to be one of the decidable extensions of $\mathcal{L}_1$ (see Section 9 for details). The new logic *LRP* forms a basis of the verification framework for programs with pointer manipulation, which is a subject of an ongoing work. For instance, in contrast to decidable logics that restrict the graphs of interest (such as weak monadic second-order logic on trees), our logic allows arbitrary graphs with an arbitrary number of fields. We show that this is very useful even for verifying programs that manipulate singly-linked lists in order to express postconditions and loop invariants that relate the input and the output state. By restricting the syntax, we guarantee that queries posed over arbitrary graphs can be answered by considering only tree-like graphs. This approach allows us to automate the reasoning about limited but interesting properties of arbitrary graphs. Moreover, our logic strictly generalizes the decidable logic in [4], which inspired our work. Therefore, it can be shown that certain heap abstractions including [24,45] can be expressed using *LRP* formulas.

The rest of the paper is organized as follows: Section 2 defines the syntax and the semantics of $\mathcal{L}_0$, and shows that it has a finite model property; Section 3 shows that $\mathcal{L}_0$ is undecidable; Section 4 defines the fragment $\mathcal{L}_1$, demonstrates the expressiveness of $\mathcal{L}_1$ on several examples, and defines an interesting extension of $\mathcal{L}_1$, called $\mathcal{L}_2$; Section 5 presents the decidability proof for $\mathcal{L}_1$, with a detailed proof of the main theorem given in Section 6; Section 7 sketches the proof of decidability of $\mathcal{L}_2$, which does not immediately follow from that of $\mathcal{L}_1$; Section 8 contains the complexity results for $\mathcal{L}_1$; Section 9 discusses the limitations and the extensions of the new logics; finally, Section 10 discusses related work.

2 The $\mathcal{L}_0$ Logic

In this section, we define the syntax and the semantics of our logic. For simplicity, we explain the material in terms of expressing properties of heaps. However, our logic can actually model properties of arbitrary directed graphs. Still, the logic is powerful enough to express the property that a graph denotes a heap.

2.1 Syntax of $\mathcal{L}_0$

$\mathcal{L}_0$ is a propositional logic over reachability constraints. That is, an $\mathcal{L}_0$ formula is a boolean combination of closed formulas in first-order logic with transitive closure that satisfy certain syntactic restrictions.

Let $\tau = \langle C, U, F \rangle$ denote a vocabulary, where

- C is a finite set of constant symbols usually denoting designated objects in the heap, pointed to by program variables;
- U is a set of unary relation symbols denoting properties, e.g., the color of a node in a Red-Black tree;
- F is a finite set of binary relation symbols (edges) usually denoting pointer fields. [3]

For example, we can describe a doubly-linked list with forward pointer f and backward pointer b, pointed-to by a program variable x, using the vocabulary in which $C = \{x\}$, $U = \{\}$, and $F = \{f, b\}$. We can describe a Red-Black tree pointed-to by a program variable $root$ using the vocabulary in which $C = \{root\}$, $U = \{red, black\}$, and $F = \{right, left\}$.

A **term** t is either a variable or a constant. An **atomic formula** is an equality $t = t'$, a monadic formula $u(t)$ for some $u \in U$, or an edge formula $t \xrightarrow{f} t'$ for some $f \in F$, and terms t, t'. A **quantifier-free formula** $\psi(v_0, \ldots, v_n)$ over τ and variables $v_0, \ldots, v_n$ is an arbitrary boolean combination of atomic formulas. We say that a sub-formula ψ appears positively (negatively) in φ, if ψ appears under an even (odd) number of negations in φ. Let $FV(\psi)$ denote the free variables of the formula ψ.

Definition 2.1 *A* ***neighborhood formula*** $N(v_0, \ldots, v_n)$ *is a conjunction of edge formulas of the form* $v \xrightarrow{f} v'$*, where* $f \in F$ *and* $v, v' \in \{v_0, \ldots, v_n\}$*, and monadic formulas of the form* $u(v)$ *or* $\neg u(v)$*, where* $u \in U$*.*

Definition 2.2 *Let* $N(v_0, \ldots, v_n)$ *be a neighborhood formula. The* ***Gaifman graph*** *of* N*, denoted by* B_N*, is an undirected graph with a vertex for each free variable of* N*. There is an edge between the vertices corresponding to* v_i *and* v_j *in* B_N *if and only if* $(v_i \xrightarrow{f} v_j)$ *appears in* N*, for some* $f \in F$*. The* ***distance*** *between logical variables* v_i *and* v_j *in the formula* N *is the minimal edge distance between the corresponding vertices* v_i *and* v_j *in* B_N*.*

[3] We can also allow auxiliary constants and fields including abstract fields [11].

For example, for the formula $N = (v_0 \xrightarrow{f} v_1) \wedge (v_0 \xrightarrow{f} v_2)$ the distance between v_1 and v_2 in N is 2, and its underlying graph B_N looks like this: v_1 — v_0 — v_2.

Definition 2.3 *A **routing expression** is an extended regular expression, defined as follows:*

$$
\begin{array}{rlll}
R ::= & \emptyset & & \textit{empty set} \\
| & \epsilon & & \textit{empty path} \\
| & \xrightarrow{f} & f \in F & \textit{forward along edge} \\
| & \xleftarrow{f} & f \in F & \textit{backward along edge} \\
| & u & u \in U & \textit{test if u holds} \\
| & \neg u & u \in U & \textit{test if u does not hold} \\
| & c & c \in C & \textit{test if c holds} \\
| & \neg c & c \in C & \textit{test if c does not hold} \\
| & R_1.R_2 & & \textit{concatenation} \\
| & R_1|R_2 & & \textit{union} \\
| & R^* & & \textit{Kleene star}
\end{array}
$$

Intuitively, a routing expression describes a path in the heap.

A routing expression can require that a path traverse some pointer fields backwards. For example, the routing expression $\xrightarrow{f}^*.\xleftarrow{f}^*$ describes a sequence of f-edges that may look like this: $\xrightarrow{f}\xrightarrow{f}\xleftarrow{f}\xleftarrow{f}\xleftarrow{f}$. We use this routing expression in Section 4.2 to describe disjoint data-structures.

A routing expression has the ability to test properties of heap objects along the path. For example, a routing expression $(\xrightarrow{f}.\neg y)^*$ describes a path which does not traverse an object pointed-to by the program variable y. We use this routing expression to describe a path along which some property holds *until* the path reaches the object pointed-to by y (see Section 4.2).

Definition 2.4 (Syntax of $\mathcal{L}_0$) *A **reachability constraint** is a closed formula of the form:*

$$\forall v_0, \ldots, v_n . R(c, v_0) \Rightarrow (N(v_0, \ldots, v_n) \Rightarrow \psi(v_0, \ldots, v_n)) \tag{1}$$

where $c \in C$ is a constant, R is a routing expression, N is a neighborhood formula, and ψ is an arbitrary quantifier-free formula, such that $FV(N) \subseteq \{v_0, \ldots, v_n\}$ and $FV(\psi) \subseteq FV(N) \cup \{v_0\}$. In particular, if the neighborhood formula N is $true$ (the empty conjunction), then ψ is a formula with a single free variable v_0.

*An $\mathcal{L}_0$ **formula** is a boolean combination of reachability constraints.*

The subformula $N(v_0, \ldots, v_n) \Rightarrow \psi(v_0, \ldots, v_n)$ defines a **pattern**, denoted by

$p(v_0)$. Here, the designated variable v_0 denotes the “central” node of the “neighborhood” reachable from c by following an R-path. Intuitively, neighborhood formula N binds the variables $v_0, \ldots, v_n$ to nodes that form a subgraph, and ψ defines more constraints on those nodes. [4]

For example, the pattern $det_f(v_0)$ defined by the formula $(v_0 \xrightarrow{f} v_1) \wedge (v_0 \xrightarrow{f} v_2) \Rightarrow (v_1 = v_2)$ ensures that v_0 has at most one outgoing f-edge. The neighborhood formula $(v_0 \xrightarrow{f} v_1) \wedge (v_0 \xrightarrow{f} v_2)$ contains two edges emanating from the central node v_0. The restriction on the neighborhood is that the edges are in fact the same, because they have the same source, v_0, the same target, $v_1 = v_2$, and the same label f.

We use **let** expressions to specify the scope in which the pattern is declared:

$$\textbf{let}\ p_1(v_0) \stackrel{\text{def}}{=} N_1(v_0, \ldots, v_n) \Rightarrow \psi_1(v_0, \ldots, v_n)\ \textbf{in}\ \varphi$$

This allows us to write more concise formulas via reuse of pattern definitions. For example, we can say that program variables x and y are pointing to (potentially shared) doubly-linked lists:

$$\textbf{let}\ inv_{f,b}(v_0) \stackrel{\text{def}}{=} (v_0 \xrightarrow{f} v_1 \Rightarrow v_1 \xrightarrow{b} v_0)\ \textbf{in}\ x[\xrightarrow{f}]inv_{f,b} \wedge x[\xrightarrow{f}]inv_{f,b}$$

2.1.1 Shorthands

We use $c[R]p$ to denote a reachability constraint (1). Intuitively, the reachability constraint requires that every node that is reachable from c by following an R-path satisfy the pattern p.

We use $c_1[R]\neg c_2$ to denote **let** $p(v_0) \stackrel{\text{def}}{=} (true \Rightarrow \neg(v_0 = c_2))$ **in** $c_1[R]p$. In this simple case, the neighborhood is only the node assigned to v_0. Intuitively, $c_1[R]\neg c_2$ means that the node labeled by constant c_2 is not reachable along an R-path from the node labeled by c_1. We use $c_1\langle R\rangle c_2$ as a shorthand for $\neg(c_1[R]\neg c_2)$. Intuitively, $c_1\langle R\rangle c_2$ means that *there exists* an R-path from c_1 to c_2. We use $c_1 = c_2$ to denote $c_1\langle\epsilon\rangle c_2$, and $c_1 \neq c_2$ to denote $\neg(c_1 = c_2)$.

We use $c[R](p_1 \wedge p_2)$ to denote $(c[R]p_1) \wedge (c[R]p_2)$, when p_1 and p_2 agree on the central node variable. When two patterns are often used together, we introduce a name for their conjunction (instead of naming each one separately): **let** $p(v_0) \stackrel{\text{def}}{=} (N_1 \Rightarrow \psi_1) \wedge (N_2 \Rightarrow \psi_2)$ **in** φ.

For a quantifier-free formula $\psi(v_0)$ with a single free variable v_0, we write $c[R]\psi$ instead of **let** $p(v_0) \stackrel{\text{def}}{=} (true \Rightarrow \psi(v_0))$ **in** $c[R]p$. In particular, for a unary relation symbol u, we use $c[R]u$ to denote **let** $p(v_0) \stackrel{\text{def}}{=} (true \Rightarrow u(v_0))$ **in** $c[R]p$. We use $u(c)$

[4] In all our examples, a neighborhood formula N used in a pattern is such that B_N (the Gaifman graph of N) is connected.

to denote the formula $c\langle\epsilon\rangle u$ (equivalently, $c[\epsilon]u$). We abuse the notations slightly by writing $N \wedge \psi_1 \Rightarrow \psi_2$ instead of $N \Rightarrow (\psi_1 \Rightarrow \psi_2)$.

In routing expressions, we use $\xrightarrow{\Sigma}$ to denote the routing expression $(\xrightarrow{f_1}|\xrightarrow{f_2}|\ldots|\xrightarrow{f_m})$, the union of all the fields in F. Similarly, $\xleftarrow{\Sigma}$ denotes the routing expression $(\xleftarrow{f_1}|\xleftarrow{f_2}|\ldots|\xleftarrow{f_m})$. For example, $c_1[\xrightarrow{\Sigma}^*]\neg c_2$ means that c_2 is not reachable from c_1 by any path. Finally, we sometimes omit the concatenation operator "." in routing expressions.

2.2 Semantics of $\mathcal{L}_0$

$\mathcal{L}_0$ formulas are interpreted over labeled directed graphs. A labeled directed graph G over a vocabulary $\tau = \langle C, U, F\rangle$ is a tuple $\langle V^G, E^G, C^G, U^G\rangle$ where:

- V^G is a set of nodes modelling the heap objects,
- E^G: $F \to \mathcal{P}(V^G \times V^G)$ are labeled edges,
- C^G: $C \to V^G$ provides interpretation of constants as unique labels on the nodes of the graph, and
- U^G: $U \to \mathcal{P}(V^G)$ maps unary relation symbols to the set of nodes in which they hold.

The language $L(R)$ of words accepted by a routing expression R is defined as usual for regular expression. The semantics of $\mathcal{L}_0$ formulas is formally defined as follows.

Definition 2.5 *Consider a routing expression R and $w \in L(R)$. We say that* ***there is a path labeled by w from a node s_1 to a node s_2*** *in G if one of the following conditions holds:*

- $s_1 = s_2$ *and* $w = \epsilon$,
- $s_1 = s_2$, $w = u$ *for a unary relation symbol* u *and* $s_1 \in U^G(u)$,
- $s_1 = s_2$, $w = \neg u$ *for a unary relation symbol* u *and* $s_1 \notin U^G(u)$,
- $s_1 = s_2$, $w = c$ *for a constant* c *and* $C^G(c) = s_1$,
- $s_1 = s_2$, $w = \neg c$ *for a constant* c *and* $C^G(c) \neq s_1$,
- $w = \xrightarrow{f}$ *for an edge* $f \in F$ *and* $\langle s_1, s_2\rangle \in E^G(f)$,
- $w = \xleftarrow{f}$ *for an edge* $f \in F$ *and* $\langle s_2, s_1\rangle \in E^G(f)$,
- $w = w_1.w_2$ *and there exists a node* s_3 *such that there is a path labeled by* w_1 *from* s_1 *to* s_3 *and there exists a path labeled by* w_2 *from* s_3 *to* s_2 .

A node tuple in G satisfies a pattern p if it satisfies the quantifier-free formula that defines p, according to the usual semantics of the first-order logic over graph structures.

The satisfaction relation $\models$ between a graph G and $\mathcal{L}_0$ formulas is defined similarly to the usual semantics the first-order logic with transitive closure over graphs. A graph G satisfies a formula $c[R]p$ (and we write $G \models c[R]p$) if and only if for every

$w \in L(R)$ *and for every node tuple* $s_0, \dots, s_n$ *in* G*, if there is a path labeled by* w *from* c *to* s_0*, then the tuple* $s_0, \dots, s_n$*, satisfies* p *with* s_0 *used as the central node for* p*. The meaning of Boolean connectives is defined in a standard way.*

We say that *node* $s \in G$ *is labeled with* σ if $\sigma \in C$ and $s = C^G(\sigma)$ or $\sigma \in U$ and $s \in U^G(\sigma)$. For an edge $\langle s_1, s_2 \rangle \in G$ and $f \in F$, we say that $\langle s_1, s_2 \rangle$ *is labeled with* f, if $\langle s_1, s_2 \rangle \in E^G(f)$. In the rest of the paper, *graph* denotes a directed labeled graph, in which nodes are labeled by constant and unary relation symbols, and edges are labeled by binary relation symbols, as defined above.

Remark. The translation from $\mathcal{L}_0$ to MSO in Section 5.1 provides an alternative definition for the semantics of $\mathcal{L}_0$.

2.3 Finite Model Property

We are interested in checking validity (and satisfiability) of $\mathcal{L}_0$ formulas only over finite graphs. The graphs are finite because they represent data-structures allocated by a program. (However, the graphs may be unbounded, due to dynamic allocation of memory.) In general, a finite validity problem is considered more difficult than a validity problem. For example, in first-order logic, the validity problem is recursively enumerable while the finite validity problem is not. In a logic with the finite model property, the notions of validity and *finite* validity coincide. Thus, the finite model property is desirable.

$\mathcal{L}_0$ with arbitrary patterns has a finite model property. If formula $\varphi \in \mathcal{L}_0$ has an infinite model, each reachability constraint in φ that is satisfied by this model has a finite witness.

Theorem 2.6 (Finite model property) *Every satisfiable* $\mathcal{L}_0$ *formula is satisfiable by a finite graph.*

Sketch of Proof: We show that $\mathcal{L}_0$ can be translated into a fragment of an infinitary logic that has a finite model property. Observe that $c[R]p$ is equivalent to an infinite conjunction of universal first-order sentences. Therefore, if G is a model of $c[R]p$ then every subgraph of G is also its model. Dually, $\neg c[R]p$ is equivalent to an infinite disjunction of existential first-order sentences. Therefore, if G is a model of $\neg c[R]p$, then G has a finite subgraph G' such that every subgraph of G that contains G' is a model of $\neg c[R]p$. It follows that every satisfiable boolean combination of formulas of the form $c[R]p$ has a finite model. Thus, $\mathcal{L}_0$ has a finite model property.

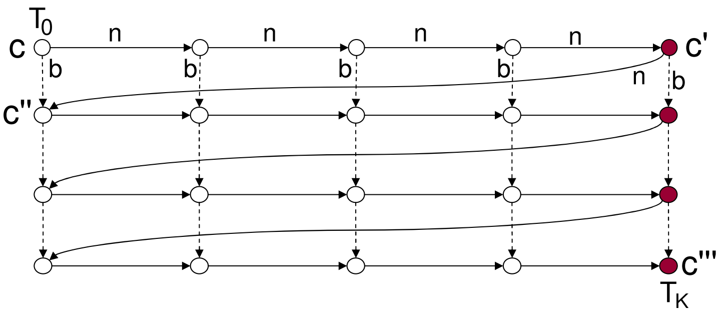


Fig. 1. A sketch of a grid model for a tiling problem $\mathcal{T}$. The n-edges are depicted with solid lines, the b-edges are depicted with dashed lines. The filled circles denote nodes labeled with "red".

3 Undecidability of $\mathcal{L}_0$

The satisfiability and the validity problems of $\mathcal{L}_0$ formulas are undecidable. Since $\mathcal{L}_0$ is closed under negation, it is sufficient to show that its satisfiability problem is undecidable. The proof uses a reduction from the tiling problem.

Definition 3.1 *Define a* ***tiling problem****, $\mathcal{T} = \langle T, R, D\rangle$, to consist of a finite list of tile types, $T = [t_0, \ldots t_k]$, together with horizontal and vertical adjacency relations, $R, D \subseteq T^2$. Here $R(a, b)$ means that tiles of type b fit immediately to the right of tiles of type a, and $D(a, b)$ means that tiles of type b fit one step down from those of type a. A* solution *to a tiling problem is an arrangement of instances of the tiles in a rectangular grid such that a t_0 tile occurs in the top left node of the grid, and a t_k tile occurs in the bottom right node of the grid, and all adjacency relationships are respected.*

It is well-known that tiling problems of this flavor are undecidable. Therefore, if a logic can express tilings, its satisfiability problem is also undecidable. Given a tiling problem $\mathcal{T}$, we construct a formula $\varphi_{\mathcal{T}}$, such that $\varphi_{\mathcal{T}}$ is satisfiable if and only if there exists a solution to $\mathcal{T}$.

The idea is that each node in the graph that satisfies $\varphi_{\mathcal{T}}$ describes a tile, with unary relation symbols $T_0, \ldots, T_k$ encoding the tile types $t_0, \ldots t_k$. There is a b-edge between every two nodes that are vertically adjacent in the grid. There is an n-edge between every two nodes that are horizontally adjacent in the grid, and from the last node of every row to the first node in the subsequent row. The constant c labels the top left node of the grid, the constant c' labels the top right node of the grid, the constant c'' labels the first node of the second row of the grid, and the constant c''' labels the bottom right node of the grid (see sketch in Fig. 1). The unary relation red labels the nodes of the last column of the grid.

The most interesting part of the formula $\varphi_{\mathcal{T}}$ ensures that all graphs that satisfy $\varphi_{\mathcal{T}}$ have a grid-like form. It states that for every node v that is n-reachable from c, if there is a b-edge from v to u, then there is a b-edge from the n-successor of v to the

n-successor of u:

$$\textbf{let } p(v) \overset{\text{def}}{=} (v \xrightarrow{b} u) \wedge (v \xrightarrow{n} v_1) \wedge (u \xrightarrow{n} u_1) \Rightarrow (v_1 \xrightarrow{b} u_1) \textbf{ in } c[(\xrightarrow{n})^*]p \tag{2}$$

Theorem 3.2 (Undecidability) *The satisfiability problem of $\mathcal{L}_0$ formulas is undecidable.*

Proof: Given a tiling problem $\mathcal{T} = \langle T, R, D\rangle$, we construct an $\mathcal{L}_0$ formula $\varphi_{\mathcal{T}}$ as a conjunction of the following formulas:

(1) There is n-path from c to c': $c\langle(\xrightarrow{n})^*\rangle c'$
(2) There is n-edge from c' to c'': $c'\langle\xrightarrow{n}\rangle c''$
(3) There is n-path from c'' to c''': $c''\langle(\xrightarrow{n})^*\rangle c'''$
(4) There is b-edge from c to c'' : $c\langle\xrightarrow{b}\rangle c''$.
(5) No n-edge exits t: $c'''[\xrightarrow{n}]false$.
(6) For every node v that is n-reachable from s, if there is a b-edge from v to u, then there is a b-edge from the n-successor of v to the n-successor of u: **let** $p(v) \overset{\text{def}}{=} (v \xrightarrow{b} u) \wedge (v \xrightarrow{n} v_1) \wedge (u \xrightarrow{n} u_1) \Rightarrow (v_1 \xrightarrow{b} u_1)$ **in** $c[(\xrightarrow{n})^*]p$.
(7) The n-edges and the b-edges reachable from s are deterministic: **let** $det_n(v) \overset{\text{def}}{=} (v \xrightarrow{n} v') \wedge (v \xrightarrow{n} v'') \Rightarrow (v' = v'')$ **in** $s[(\xrightarrow{n})^*]det_n$, similarly, for b-edges.
(8) The top left node of the grid has a t_0 tile type, and the bottom right node of the grid has a t_k tile type: $T_0(c) \wedge T_k(c''')$.
(9) Each node in the grid has exactly one tile type:

$$c[(\xrightarrow{n})^*]\left(\bigwedge_{0\le i<j\le k} \neg(T_i \wedge T_j)\right) \wedge \left(\bigvee_{0\le i\le k} T_i\right)$$

(10) Every node in the last column of the grid is labeled with red: $c'[(\xrightarrow{b})^*]red$.
(11) To express that only nodes in the last column of the grid are labeled with red, we say that the first row is not labeled with red, except its last node, and if a node is labeled with red, then its b-predecessor is labeled:

$$c[(\xrightarrow{n}.\neg c')^*]\neg red \wedge \textbf{let } p(v) \overset{\text{def}}{=} (w \xrightarrow{b} v) \wedge red(v) \Rightarrow red(w) \textbf{ in } c[(\xrightarrow{n})^*]p$$

(12) Two horizontally adjacent tiles are compatible according to R:

$$\textbf{let } p(v) \overset{\text{def}}{=} (v \xrightarrow{n} w) \wedge \neg red(v) \Rightarrow \left(\bigvee_{R(t_i,t_j)} (T_i(v) \wedge T_j(w))\right) \textbf{ in } c[(\xrightarrow{n})^*]p$$

(13) Two vertically adjacent tiles are compatible according to D:

$$\textbf{let } p(v) \overset{\text{def}}{=} (v \xrightarrow{b} w) \Rightarrow \bigvee_{D(t_i,t_j)} (T_i(v) \wedge T_j(w)) \textbf{ in } c[(\xrightarrow{n})^*]p$$

Remark. The reduction uses only two binary relation symbols and a fixed number of unary relation symbols. It can be modified to show that the logic with three binary relation symbols (and no unary relations) is undecidable.

4 Decidable and Useful Fragments of $\mathcal{L}_0$

In this section we define two fragments of $\mathcal{L}_0$ and show their usefulness. In the next section, we show that these fragments are decidable.

First, we define the $\mathcal{L}_1$ fragment of $\mathcal{L}_0$, by syntactically restricting the patterns. We show that $\mathcal{L}_1$ naturally describes some commonly-used data-structures, and express verification conditions. Second, we define $\mathcal{L}_2$ by extending $\mathcal{L}_1$ with constants in patterns, and show that this extension allows us to describe more complex data-structures.

4.1 The $\mathcal{L}_1$ Fragment

The $\mathcal{L}_1$ fragment is defined by syntactically restricting the patterns which can be used. The fragment $\mathcal{L}_1$ permits arbitrary boolean combinations in patterns, but it restricts the distance between variables and forbids the use of constants in positive occurrences of equality and edge formulas.

Definition 4.1 (The syntax of $\mathcal{L}_1$) *In every reachability constraint $c[R]p$ that appears in an $\mathcal{L}_1$ formula, the pattern $p(v_0) \stackrel{\text{def}}{=} N(v_0, \ldots, v_n) \Rightarrow \psi(v_0, \ldots, v_n)$ satisfies the following restrictions on ψ:*

- ***(equality restriction)** If ψ contains a positive occurrence of an equality between variables $v_i = v_j$, then the distance between v_i and v_j in N is at most 2 (distance is defined in Def. 2.2).*
- ***(edge restriction)** If ψ contains a positive occurrence of an edge formula of the form $v_i \xrightarrow{f} v_j$, then the distance between v_i and v_j in N is at most 1.*
- ***(constant restriction)** Positive occurrences of formulas of the form $v \xrightarrow{f} c$, $c \xrightarrow{f} v$, and $v = c$ in ψ are not allowed.*

Remark. Note that formula (2), which is used in the proof of undecidability in Theorem 3.2, is not in $\mathcal{L}_1$, because p contains a positive $v_1 \xrightarrow{b} u_1$ with distance 3 between v_1 and u_1, while $\mathcal{L}_1$ allows edge patterns with distance at most 1.

Pattern Name	**Pattern Definition**	**Meaning**
$det_f(v_0)$	$(v_0 \xrightarrow{f} v_1) \wedge (v_0 \xrightarrow{f} v_2) \Rightarrow (v_1 = v_2)$	at most one outgoing f-edge from v_0
$uns_f(v_0)$	$(v_1 \xrightarrow{f} v_0) \wedge (v_2 \xrightarrow{f} v_0) \Rightarrow (v_1 = v_2)$	v_0 has at most one incoming f-edge
$uns_{f,g}(v_0)$	$(v_1 \xrightarrow{f} v_0) \wedge (v_2 \xrightarrow{g} v_0) \Rightarrow false$	v_0 is not heap-shared by f-edge and g-edge
$inv_{f,b}(v_0)$	$(v_0 \xrightarrow{f} v_1 \Rightarrow v_1 \xrightarrow{b} v_0)$	every f-edge from v_0 to v_1 has a b-edge in the opposite direction.
$same_{f,g}(v_0)$	$(v_0 \xrightarrow{f} v_1 \Rightarrow v_0 \xrightarrow{g} v_1)$ $\wedge$ $(v_0 \xrightarrow{g} v_1 \Rightarrow v_0 \xrightarrow{f} v_1)$	edges f and g emanating from v_0 are parallel

Table 1
Useful pattern definitions ($f, b, g \in F$ are edge labels).

4.2 Describing Linked Data-Structures in $\mathcal{L}_1$

In this section, we show that $\mathcal{L}_1$ can express properties of data-structures. Table 1 lists some useful patterns and their meanings. For example, the first pattern det_f means that there is at most one outgoing f-edge from a node. Another important pattern uns_f means that a node has at most one incoming f-edge. We use the subscript f to emphasize that this definition is parametric in f.

Well-formed heaps We assume that C (the set of constant symbols) contains a constant for each pointer variable in the program (denoted by x, y in our examples). Also, C contains a designated constant $null$ that represents `null` values. Throughout the rest of the paper we assume that all the graphs denote well-formed heaps, i.e., the fields of all objects reachable from constants are deterministic, and dereferencing NULL yields $null$. In $\mathcal{L}_1$ this is expressed by the formula:

$$(\bigwedge_{c \in C} \bigwedge_{f \in F} c[\Sigma^*]det_f) \wedge (\bigwedge_{f \in F} null\langle \xrightarrow{f} \rangle null) \qquad (3)$$

Using the patterns in Table 1, Table 2 defines some interesting properties of data-structures using $\mathcal{L}_1$. The formula $reach_{x,f,y}$ means that the object pointed-to by the program variable y is reachable from the object pointed-to by the program variable x by following an access path of f field pointers. We can also use it with $null$ in the place of y. For example, the formula $reach_{x,f,null}$ describes a (possibly empty) linked-list pointed-to by x. Note that $reach_{x,f,null}$ implies that the list is acyclic, because $null$ is always a "sink" node in a well-formed heap. We can also express that there are no incoming f-edges into the list pointed to by x, by conjoining the previous formula with $unshared_{x,f}$. Alternatively, we can specify the fact that x is located on a cycle of f-edges: $cyclic_{x,f}$. Disjointness can be expressed by the formula $disjoint_{x,f,y,g}$ that uses both forward and backward traversal of edges in the

Name	**Formula**
$reach_{x,f,y}$	$x\langle(\xrightarrow{f})^*\rangle y$ the heap object pointed-to by y is reachable from the heap object pointed-to by x.
$cyclic_{x,f}$	$x\langle(\xrightarrow{f})^+\rangle x$ cyclicity: the heap object pointed-to by x is located on a cycle.
$unshared_{x,f}$	$x[(\xrightarrow{f})^*]uns_f$ every heap object reachable from x by an f-path has at most one incoming f-edge.
$disjoint_{x,f,y,g}$	$x[(\xrightarrow{f})^*(\xleftarrow{g})^*]\neg y$ disjointness: there is no heap object that is reachable from x by an f-path and also reachable from y by a g-path.
$same_{x,f,g}$	$x[(\xrightarrow{f}\vert\xrightarrow{g})^*]same_{f,g}$ the f-path and the g-path from x are parallel, and traverse the same objects.
$inverse_{x,f,b,y}$	$reach_{x,f,y} \wedge x[(\xrightarrow{f}.\neg y)^*]inv_{f,b}$ doubly-linked lists between two variables x and y with f and b as forward and backward edges.
$tree_{root,r,l}$	$root[(\xrightarrow{l}\vert\xrightarrow{r})^*](uns_{l,r} \wedge uns_l \wedge uns_r) \wedge \neg(root\langle(\xrightarrow{l}\vert\xrightarrow{r})^+\rangle root)$ tree rooted at $root$.
$tree_{root,r,l,b}$	$tree_{root,r,l} \wedge root[(\xrightarrow{l}\vert\xrightarrow{r})^*]inv_{l,b} \wedge inv_{r,b}$ tree rooted at $root$ with parent pointers b from every tree node to its parent.

Table 2
Properties of data-structures expressed in $\mathcal{L}_1$.

routing expression. Disjointness of data-structures is important for parallelization (e.g., see [25]). For example, we can express that the linked list pointed to by x is disjoint from the linked-list pointed to by y, using the formula $disjoint_{x,f,y,f}$. This formula guarantees that every node v that is reachable from the node pointed-to by x using an f-path must *not* be reachable from y using an f-path. However, v may be reachable from y using other edges, or v maybe a part of another data-structure which shares elements with y.

The last three examples in Table 2 specify data-structures with multiple fields. The formula $inverse_{x,f,b,y}$ describes a doubly-linked list with variables x and y pointing to the head and the tail of the list, respectively. First, it guarantees the existence of an f-path. Next, it uses the pattern $inv_{f,b}$ to express that if there is an f-edge from one node to another, then there is a b-edge in the opposite direction. This pattern is applied to all nodes on the f-path that starts from x and that does not visit y, expressed using the test "$\neg y$" in the routing expression.

```
Node reverse(Node x){
  [0] Node y = null;
  [1] while (x != null){
  [2]    Node t = x.n;
  [3]    x.n = y;
  [4]    y = x;
  [5]    x = t;
  [6] }
  [7] return y;
}
```

Fig. 2. The `reverse` procedure performs in-place reversal of a singly-linked list

The formula $tree_{root,r,l}$ describes a binary tree. The first part requires that the nodes reachable from the root (by following any path of l and r fields) not be heap-shared. The second part prevents edges from pointing back to the root of the tree by forbidding the root to participate in a cycle. The formula $tree_{root,r,l,b}$ describes a binary tree rooted at $root$ with parent pointers b from every tree node to its parent.

The ability to express properties like $tree_{root,r,l}$ is non-trivial, because we are operating on general graphs, and not just trees. Operating on general graphs allows us to verify that the data-structure invariant is reestablished after a sequence of low-level mutations that temporarily violate the invariant data-structure.

4.3 Expressing Verification Conditions in $\mathcal{L}_1$

4.3.1 The Reverse Procedure

The `reverse` procedure shown in Fig. 2 performs in-place reversal of a singly-linked list. This procedure is interesting because it destructively updates the list and the natural specification of its partial correctness requires reasoning about two fields. Moreover, it manipulates linked lists in which each list node can be pointed-to from the outside. We show that the verification conditions for the procedure `reverse` can be expressed in $\mathcal{L}_1$. If the verification conditions are valid, then the program is partially correct with respect to the specification. The validity of the verification conditions can be checked automatically because the logic $\mathcal{L}_1$ is decidable, as shown in the next section. In [49], we show how to automatically generate verification conditions in $\mathcal{L}_1$ for arbitrary procedures that are annotated with preconditions, postconditions, and loop invariants in $\mathcal{L}_1$.

Notice that in this section we assume that all graphs denote valid stores, i.e., satisfy (3). The precondition requires that x point to an acyclic list, on entry to the procedure. We use the symbols x^0 and n^0 to record the values of the variable x and

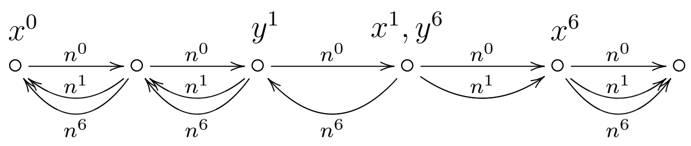


Fig. 3. An example graph that satisfies the VC_{loop} formula for `reverse`.

the n-field on entry to the procedure.

$$pre_{reverse} \overset{\text{def}}{=} x^0\langle(\xrightarrow{n^0})^*\rangle null$$

The postcondition ensures that the result is an acyclic list pointed-to by y. Most importantly, it ensures that each edge of the original list is reversed in the returned list, which is expressed in a similar way to a doubly-linked list, using $inverse$ formula. We use the relation symbols y^7 and n^7 to refer to the values on exit.

$$post_{reverse} \overset{\text{def}}{=} y^7\langle(\xrightarrow{n^7})^*\rangle null \land inverse_{x^0,n^0,n^7,y^7}$$

The loop invariant φ shown below relates the heap on entry to the procedure to the heap at the beginning of each loop iteration (label `L1`). First, we require that the part of the list reachable from x be the same as it was on entry to `reverse`. Second, the list reachable from y is reversed from its initial state. Finally, the only original edge outgoing of y is to x.

$$\varphi \overset{\text{def}}{=} same_{x^1,n^0,n^1} \land inverse_{x^0,n^0,n^1,y^1} \land y^1\langle\xrightarrow{n^0}\rangle x^1$$

Note that the postcondition uses two binary relations, n^0 and n^7, and also the loop invariant uses two binary relations, n^0 and n^1. This illustrates that reasoning about singly-linked lists requires more than one binary relation.

The verification condition of `reverse` consists of two parts, VC_{loop} and VC, explained below.

The formula VC_{loop} expresses the fact that φ is indeed a loop invariant. To express it in our logic, we use several copies of the vocabulary, one for each program point. Different copies of the relation symbol n in the graph model values of the field n at different program points. Similarly, for constants. For example, Fig. 3 shows a graph that satisfies the formula VC_{loop} below. It models the heap at the end of some loop iteration of `reverse`. The superscripts of the symbol names denote the corresponding program points.

To show that the loop invariant φ is maintained after executing the loop body, we assume that the loop condition and the loop invariant hold at the beginning of the iteration, and show that the loop body was executed without performing a null-

```
  Node append(Node x, Node y) {
    [0] Node t = x;
    [1] if (t == null)
    [2]    return y;
    [3] while (t.n != null) {
    [4]    t = t.n;
    [5] }
    [6] t.n = y;
    [7] return x;
}
```

Fig. 4. The `append` procedure concatenates two singly-linked lists.

dereference, and the loop invariant holds at the end of the loop body:

$$\begin{aligned}
VC_{loop} \overset{\text{def}}{=}\ & (x^1 \neq null) && \text{loop is entered}\\
& \wedge\varphi && \text{loop invariant holds on loop head}\\
& \wedge(y^6 = x^1) \wedge x^1\langle n^1\rangle x^6 \wedge x^1\langle n^6\rangle y^1 && \text{loop body}\\
& \wedge same_{y^1,n^1,n^6} \wedge same_{x^6,n^1,n^6} && \text{rest of the heap remains unchanged}\\
\Rightarrow\ & (x^1 \neq null) && \text{no null-derefernce in the body}\\
& \wedge\varphi^6 && \text{loop invariant after executing loop body}
\end{aligned}$$

Here, φ^6 denotes the loop-invariant formula φ after executing the loop body (label `L6`), i.e., replacing all occurrences of x^1, y^1 and n^1 in φ by x^6, y^6 and n^6, respectively. The formula VC_{loop} defines a relation between three states: on entry to the procedure, at the beginning of a loop iteration and at the end of a loop iteration.

The formula VC expresses the fact that if the precondition holds and the execution reaches the exit of the procedure (i.e., the loop is not entered because the loop condition does not hold), the postcondition holds on exit: $VC \overset{\text{def}}{=} pre \wedge (x^1 = null) \Rightarrow post$.

4.3.2 The Append Procedure

The `append` procedure given in Fig. 4 concatenates two singly-linked lists.

To describe the effect of a procedure on the heap, we sometimes use *auxiliary* relations and constants, whose interpretation is constrained in the precondition, and used in the postconditions. It allows us to relate the values after a call to a procedure returns to the values before the corresponding call. Note that the auxiliary constant does not have an index, because it is not part of the program. In this example, we use the auxiliary constant $last$ to label the last node of the first list.

The precondition for append requires that x and y point to acyclic and disjoint lists, and defines the meaning of the new constant $last$:

$$pre_{append} = x^0\langle \xrightarrow{n^0}{}^* \rangle null \land y^0\langle \xrightarrow{n^0}{}^* \rangle null \land x^0[\xrightarrow{n^0}{}^* . \xleftarrow{n^0}{}^*]\neg y^0 \land$$
$$x^0\langle (\xrightarrow{n^0} . \neg null)^* \rangle last \land last\langle \xrightarrow{n^0} \rangle null$$

The postcondition for append uses x^7 to denote the return value, which points to an acyclic list. It uses the constant $last$ to identify the object whose $next$ field was modified by the procedure.

$$post_{append} = x^7\langle \xrightarrow{n^7}{}^* \rangle null \land x^7 = x^0 \land last\langle \xrightarrow{n^7} \rangle y^0 \land$$
$$x^0[(\xrightarrow{n^0} . \neg last)^*]same_{n^0,n^7} \land y^0[\xrightarrow{n^0}{}^*]same_{n^0,n^7}$$

Unary relations symbols can be used to describe data values from a limited domain, and their interaction with the structural properties of the heap. For example, for a Red-Black tree we can specify that both children of every red node are black:

let $rb(v_0) \stackrel{\text{def}}{=} (red(v_0) \land (v_0 \xrightarrow{l} v_1) \Rightarrow black(v_1)) \land (red(v_0) \land (v_0 \xrightarrow{r} v_1) \Rightarrow black(v_1))$
in $root[(\xrightarrow{l} | \xrightarrow{r})^*]rb$

Moreover, unary information can be used to describe states of objects, and sets of objects, as shown by the following example.

4.3.3 The Mark Procedure

The `mark` procedure shown in Fig. 5 implements the mark phase of a Mark&Sweep garbage collector. [5]

The procedure operates on a general graph, with a node pointed by $root$. Therefore, the precondition is simply $true$. There is an f-edge from v_1 to v_2 if either $v_2 = v_1.car$ or $v_2 = v_1.cdr$. Note that unlike the previous examples, f is not deterministic. As an optimization, we do not create copies of f and $root$ for each label of the `mark` procedure, because the procedure does not modify f and $root$. We use unary relations p and m to denote objects in the $pending$ and $marked$ sets of nodes, respectively.

The postcondition for `mark` states that a node is marked if and only if it is reachable from $root$: $post_{mark} \stackrel{\text{def}}{=} post^{if}_{mark} \land post^{only-if}_{mark}$. The "if" part can be easily expressed

[5] This version is simplified because it assumes a single root object; a set of roots can be handled as shown in Section 9.

```
void mark(Node root, NodeSet marked) {
  [0] Node x;
  [1] if(!root.isEmpty()){
  [2]    NodeSet pending = new NodeSet();
  [3]    pending.addAll(root);
  [4]    marked.clear();
  [5]    while (!pending.isEmpty()) {
  [6]       x = pending.selectAndRemove();
  [7]       marked.add(x);
  [8]       if (x.car != null &&
  [9]            !marked.contains(x.car))
  [10]         pending.add(x.car);
  [11]      if (x.cdr != null &&
  [12]           !marked.contains(x.cdr))
  [13]         pending.add(x.cdr);
  [14]   }
      }
  }
```

Fig. 5. The `mark` phase of a Mark&Sweep garbage collector.

using the positive monadic formula $m^{14}(v_0)$ in the pattern, allowed in $\mathcal{L}_1$:

$$post_{mark}^{if} \overset{\text{def}}{=} root[\xrightarrow{f}^*](\neg null \Rightarrow m^{14})$$

The "only if" part requires reasoning about nodes that are not necessarily reachable from $root$. Moreover, it requires reasoning about nodes that need not be reachable from any program variable. To address it, we introduce a new constant c_m which represents an arbitrary node, because it is not restricted in the precondition, and write the postcondition and the loop invariant in terms of c_m. Intuitively, when checking validity of these formulas, the constant c_m can be treated as a universally quantified variable. In the postcondition, we require that if c_m is marked, then it is reachable from $root$:

$$post_{mark}^{only-if} \overset{\text{def}}{=} m^{14}(c_m) \Rightarrow root\langle\xrightarrow{f}^*\rangle c_m$$

The loop invariant for `mark` consists of two parts. First, before the loop at label `[5]` is entered for the first time, the only pending node is $root$, and no nodes are marked. In particular, $root$ and c_m are not marked. Second, after the loop was executed at least some number of times, (i) $root$ remains either marked or pending, (ii) a node cannot be both marked and pending, and (iii) most importantly, if a node is marked then its f-successor is either marked or pending. It means that the "frontier" of the exploration consists of pending nodes: there is no edge from a $mark$ node to a node that is neither marked nor pending. Finally, if a node is marked or pending, then it is reachable from $root$, which implies the postcondition, because the loop terminates when there are no pending nodes.

$$
\begin{aligned}
& p^5(root) \wedge (c_m \neq root \Rightarrow \neg p^5(c_m)) \wedge (root[\xrightarrow{f}^*]\neg m^5) \wedge \neg m^5(c_m) \\
\bigvee & \\
& (m^5(root) \vee p^5(root)) \\
\wedge\ & root[\xrightarrow{f}^*](\neg p^5 \vee \neg m^5) \\
\wedge\ & (\neg m^5(c_m) \vee \neg p^5(c_m)) \\
\wedge\ & \textbf{let } t(v) \overset{\text{def}}{=} f(v,v') \wedge m^5(v) \Rightarrow (m^5(v') \vee p^5(v')) \textbf{ in } root[\xrightarrow{f}^*]t \\
\wedge\ & m^5(c_m) \Rightarrow c_m[\xrightarrow{f}](p^5 \vee m^5) \\
\wedge\ & p^5(c_m) \vee m^5(c_m) \Rightarrow root\langle\xrightarrow{f}^*\rangle c_m
\end{aligned}
$$

4.4 The $\mathcal{L}_2$ Fragment

The fragment $\mathcal{L}_2$ extends $\mathcal{L}_1$ by allowing constants to be freely used in patterns, removing the last restriction of Def. 4.1. For example, the property that a general graph is a tree in which each node has a pointer b back to the root is expressible in $\mathcal{L}_2$, using the pattern $true \Rightarrow b(v_0, root)$, but this pattern is not in $\mathcal{L}_1$. It can be shown that the property cannot be expressed in $\mathcal{L}_1$, using the same arguments as in Section 7.

5 Decidability of $\mathcal{L}_1$

In this section, we show that $\mathcal{L}_1$ is decidable for validity and satisfiability. Since $\mathcal{L}_1$ is closed under negation, it is sufficient to show that it is decidable for satisfiability. The proof proceeds as follows:

(1) Translate an $\mathcal{L}_0$ formula into an equivalent MSO formula (Lemma 5.2).
(2) Define a class of simple graphs $\mathcal{A}_k$, for which the Gaifman graph (Def. 5.4) is a tree with at most k additional edges (Def. 5.5).
(3) Show that the satisfiability of MSO logic over $\mathcal{A}_k$ is decidable, by reduction to MSO on trees [41] (Lemma 5.6). We could have also shown decidability using the fact that the tree width of all graphs in $\mathcal{A}_k$ is bounded by k, and that MSO over graphs with bounded tree width is decidable [15,2,48].
(4) Every formula $\varphi \in \mathcal{L}_1$ can be effectively translated into an equi-satisfiable normal-form formula that is a disjunction of formulas in $C\mathcal{L}_1$ (Def. 5.9 and Theorem 5.12). It is sufficient to show that the satisfiability of $C\mathcal{L}_1$ is decidable.

(5) Show that if formula $\varphi \in C\mathcal{L}_1$ has a model, φ has a model in $\mathcal{A}_k$, where k is proportional to the size of the formula φ (Theorem 5.14). This is the main part of the proof, given in detail in Section 6.

In Section 7, we extend this proof to show decidability of $\mathcal{L}_2$.

5.1 Translation from $\mathcal{L}_0$ to MSO

Every regular expression R can be effectively translated into an MSO formula $\varphi_R(x, y)$, that describes the paths from x to y labeled with w, for every word w in R. To encode the Kleene star expression, we use a least fixpoint operation, expressible in MSO.

Lemma 5.1 *Every routing expression R can be translated into an MSO formula $tr(R)(v_1, v_2)$ with two (first-order) free variables v_1 and v_2 such that for every graph S and nodes $a, b \in S$, there is an R-path from a to b if and only if $S, a, b \models tr(R)(v_1, v_2)$.*

Sketch of Proof: For atomic regular expressions and concatenation, we define $tr(R)(v_1, v_2)$ as follows:

$$tr(R)(v_1, v_2) \overset{\text{def}}{=} \begin{cases} f(v_1, v_2) & \text{if } R \text{ is } \xrightarrow{f} \\ f(v_2, v_1) & \text{if } R \text{ is } \xleftarrow{f} \\ \neg(c = v_1) \wedge (v_1 = v_2) & \text{if } R \text{ is } \neg c \\ u(v_1) \wedge (v_1 = v_2) & \text{if } R \text{ is } u \\ \neg u(v_1) \wedge (v_1 = v_2) & \text{if } R \text{ is } \neg u \end{cases}$$

$$tr(R_1.R_2)(v_1, v_2) \overset{\text{def}}{=} \exists v_3.tr(R_1)(v_1, v_3) \wedge tr(R_2)(v_3, v_2)$$

The formula $tr(R^*)(v_1, v_2)$ holds when the minimal set Y that contains v_1 and is closed under R, contains v_2. Formally, we define

$$tr(R^*)(v_1, v_2) \overset{\text{def}}{=} \exists Y.(v_2 \in Y) \wedge Q(v_1, Y) \wedge \forall Y'.Q(v_1, Y') \Rightarrow Y \subseteq Y'$$

where $Q(v_1, Z)$ is $(v_1 \in Z) \wedge \forall v_1', v_2'.(v_1' \in Z) \wedge \varphi_R(v_1', v_2') \Rightarrow (v_2' \in Z)$.

For example, the routing expression $R \overset{\text{def}}{=} (\xrightarrow{n}.\neg y)^*$ is translated into the MSO formula $tr(R)(x, v) \overset{\text{def}}{=} \exists Y.(v \in Y) \wedge Q(x, Y) \wedge \forall Y'.Q(x, Y') \Rightarrow Y \subseteq Y'$, where $Q(x, Z)$ is $(x \in Z) \wedge \forall v_1', v_2'.(v_1' \in Z) \wedge \exists v_3'.(f(v_1', v_3') \wedge \neg(x = v_3') \wedge (v_3' = v_2')) \Rightarrow (v_2' \in Z)$.

Using the translation of regular expressions as defined above, it is easy to translate a general $\mathcal{L}_0$ formula to an *equivalent* MSO formula. For $\varphi \in \mathcal{L}_0$ over τ, $TR_2(\varphi)$ is an MSO formula over the same vocabulary τ. The translation TR_2 is defined inductively:

$$\begin{aligned} TR_2(c[R]p) &\overset{\text{def}}{=} \forall v_0, v_1, \ldots, v_n . \varphi_R(c, v_0) \Rightarrow p(v_0, \ldots, v_n) \\ TR_2(\varphi_1 \wedge \varphi_2) &\overset{\text{def}}{=} TR_2(\varphi_1) \wedge TR_2(\varphi_2) \\ TR_2(\neg\varphi_1) &\overset{\text{def}}{=} \neg TR_2(\varphi_1) \end{aligned}$$

For example, the $\mathcal{L}_0$ formula $\varphi \overset{\text{def}}{=} x\langle \xrightarrow{n}^* \rangle y \wedge x[(\xrightarrow{n}.\neg y)^*]inv_{n,n'}$ which is part of a loop invariant of the reverse procedure (Section 4.3.1), is translated into the MSO formula

$$TR_2(\varphi) = tr(\xrightarrow{n}^*)(x, y) \wedge \forall v_0, v_1 . tr((\xrightarrow{n}.\neg y)^*)(x, v_0) \Rightarrow (n(v_0, v_1) \Rightarrow n'(v_1, v_0))$$

where $tr(\xrightarrow{n}^*)$ and $tr((\xrightarrow{n}.\neg y)^*)$ are defined as above.

Lemma 5.2 *For all $\varphi \in \mathcal{L}_0$ and all graphs S, $S \models \varphi$ iff $S \models TR_2(\varphi)$.*

5.2 Decidability of MSO on Ayah Graphs

We define a set T^k of undirected graphs, each of which is a tree[6] with at most k extra edges.

Definition 5.3 *An* undirected *graph B is in T^k if removing self loops and at most k additional edges from B results in an acyclic (undirected) graph.*

For a directed graph we define the corresponding undirected graph:

Definition 5.4 *Let $\mathcal{G}(S)$ denote the **Gaifman graph** of the graph S, i.e., an undirected graph obtained from S by removing node labels, edge labels, and edge directions (and parallel edges).*

We define a notion of simple tree-like (directed) graphs, called *Ayah* graphs.

Definition 5.5 (Ayah Graphs) *For $k \geq 0$, an Ayah graph of k is a graph S whose Gaifman graph is in T^k: $\mathcal{A}_k = \{S | \mathcal{G}(S) \in T^k\}$.*

Examples of graphs in $\mathcal{A}_0$, $\mathcal{A}_1$, and $\mathcal{A}_2$ are shown in Fig. 6. For $j = \{0, 1, 2\}$, a structure $S_j \in \mathcal{A}_j$ is shown in the left column, and the corresponding Gaifman

[6] In this paper, we use the term “tree” instead of the term “forest” to refer to an acyclic graph, possibly undirected.

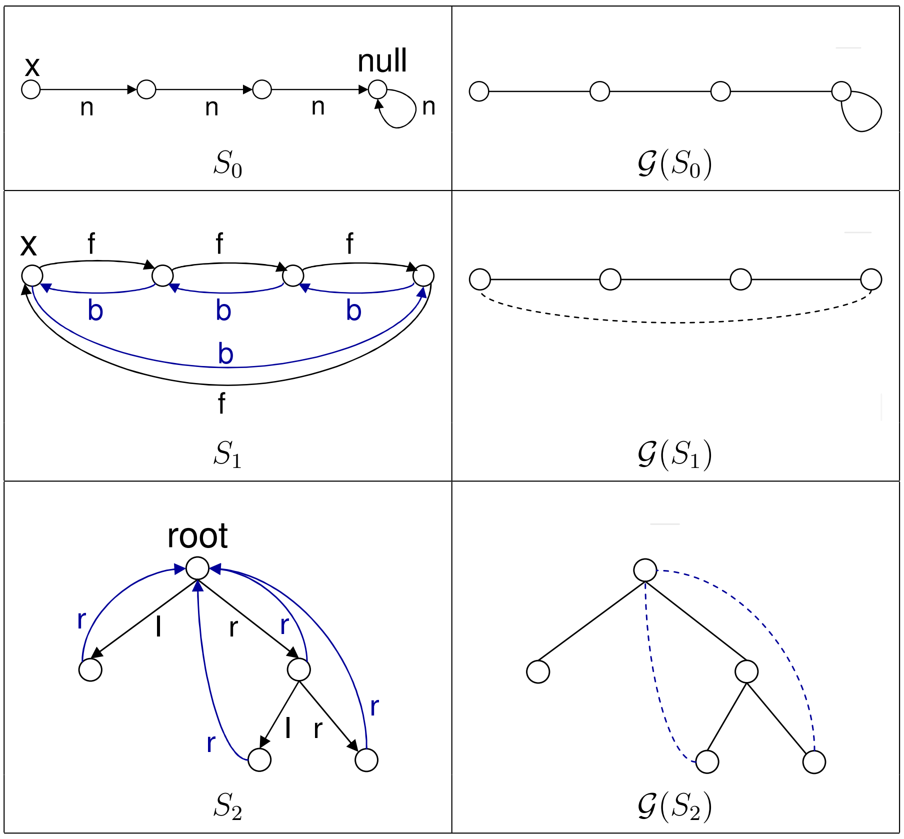


Fig. 6. Examples of graphs in $\mathcal{A}_0$, $\mathcal{A}_1$, and $\mathcal{A}_2$. For $j \in \{0, 1, 2\}$, $S_j \in \mathcal{A}_j$ (left column) and $\mathcal{G}(S_j) \in T^j$ (right column). Dashed edges denote extra edges removing which results in a tree.

graph $\mathcal{G}(S_j) \in T^j$ is shown in the right column; with j dashed edges. Removing the dashed edges from $\mathcal{G}(S_j)$ yields a tree.

The graph S_0 describes an acyclic singly-linked list pointed-to by x. The node labeled with $null$ does *not* represent an element of the list: it is a "sink" node which models the `null` value, as explained in Section 4.2. In $\mathcal{G}(S_0)$, the self-loop is not dotted because Def. 5.3 ignores self-loops. (As we show later, self-loops can be easily handled, while larger cycles require a more complex treatment.) The graph S_1 describes a cyclic doubly-linked list. In $\mathcal{G}(S_1)$, a single edge represents the parallel edges of S_1 with different directions and different labels. The graph S_2 describes a tree with pointers from every tree node to the root. In $\mathcal{G}(S_2)$, removing a single edge cannot break both cycles, thus the graph S_2 is in $\mathcal{A}_2$, but not in $\mathcal{A}_1$.

Remark. For every graph S in $\mathcal{A}_k$, the tree width [44,16] of $\mathcal{G}(S)$ is at most $k+1$, but can it can be strictly less than that. For example, a graph which consists of 17 simple disjoint cycles is in $\mathcal{A}_{17}$, but its tree width is 2.

The satisfiability problem of MSO logic on Ayah graphs can be reduced to the satisfiability problem of MSO logic on trees. The latter is decidable, due to the classical result by Rabin [41]. This reduction provides a constructive way to check satisfiability of $\mathcal{L}_1$ formulas, using an existing decision procedure for MSO on trees, MONA [26].

The reduction consists of two satisfiability-preserving translations: The first is a translation TR_3 from MSO on Ayah graphs to MSO on Σ-labeled trees, defined below. The second is a translation TR_4 from MSO on Σ-labeled trees to MSO on (infinite) binary trees.

Lemma 5.6 *There are translations TR_3 and TR_4 between MSO-formulas such that for every MSO-formula φ, there exists a graph $S \in \mathcal{A}_k$ that satisfies φ if and only if there exists a binary tree S' such that $S' \models (TR_3 \circ TR_4)(\varphi)$.*

In this paper, we describe only the translation TR_3, and omit the (standard) translation, TR_4.

5.2.1 *Encoding $\mathcal{A}_k$ Graphs as Σ-Labeled Trees*

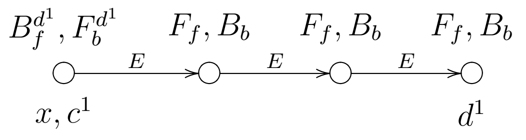
Given the vocabulary $\tau = \langle C, U, F\rangle$ and a number k we define a new vocabulary $\tau' = \langle C', U', \{E\}\rangle$, where E is the only binary relation, $C' = C \cup \{c^1, \ldots, c^k\} \cup \{d^1, \ldots, d^k\}$, and $U' = \{F_f, B_f, L_f, F_f^{d^i}, B_f^{d^i} | f \in F, i = 1, \ldots, k\})$.

Let $\Sigma = \mathcal{P}(C' \cup U')$ be the set of all possible node labels from τ'. A Σ-*labeled tree* is a graph S over τ' that satisfies the following:

(1) The E-edges form a directed forest: each node in S has at most one incoming E edge. An E-edge from node u_1 to node u_2 means that u_2 is a child of u_1 in the tree.
(2) If a node has no incoming E-edge, then it must not be labeled by F_f, B_f, for any $f \in F$.

We use T_Σ to denote the set of all Σ-labeled trees.

Every graph in $\mathcal{A}_k$ can be represented by a Σ-labeled tree. For example, consider the cyclic doubly-linked list S_1 from Fig. 6, defined over the vocabulary τ with $C = \{x\}$, $U = \{\}$, and $F = \{f, b\}$. The new vocabulary τ' consists of $C' = \{x, c^1, d^1\}$, $U' = \{F_f, F_b, F_f^{d^1}, F_b^{d^1}, B_f^{d^1}, B_b^{d^1}\}$, and $F' = \{E\}$. The graph S_1 can be represented by the following Σ-labeled tree (actually, it is a list in this example):



The graph S represented by a Σ-labeled tree has the same set of nodes as the tree. The labels of S are defined as follows. A graph node is labeled with the constants and unary relation symbols that hold for the corresponding node in the tree. An edge in the tree from node v to v' represents edges between the corresponding nodes v and v' in the graph. Additional labels on tree nodes represent the direction and the labels of the graph edges adjacent to the corresponding nodes in the graph, as follows.

For each binary relation symbol $f \in F$, we introduce two unary relation symbols F_f and B_f, denoting forward and backward f-edge. If there is an edge from v to v' in the tree, and v' is labeled with F_f in the tree, then there is an f-edge from v to v' in S. Similarly, if there is an edge from v' to v in the tree, and v is labeled with B_f in the tree, then there is an f-edge from v to v' in S. There is a self-loop of f on a node v in S if the node v in the tree is labeled with L^f. Also, each of the k pairs of constants c^i and d^i in a tree represents edges between the nodes corresponding to c^i and d^i in the graph. If v is labeled with c^i and $F_f^{d^i}$ in the tree, then there is an f-edge from v to the node labeled with d^i in S. If v is labeled with c^i and $B_f^{d^i}$ in the tree, then there is an f-edge from the node labeled with d^i to v in S.

For an MSO formula φ over τ, $TR_3(\varphi)$ is an MSO formula over the vocabulary τ'. The translation TR_3 is defined inductively on φ, where the only interesting part is the translation of a binary relation formula $f \in F$:

$$\begin{aligned} TR_3(f(v_1, v_2)) = {} & (E(v_1, v_2) \wedge F_f(v_2)) \\ & \vee (E(v_2, v_1) \wedge B_f(v_1)) \\ & \vee (E(v_1, v_2) \wedge v_1 = v_2 \wedge L_f(v_1)) \\ \textstyle\bigvee_{i=1}^{k} & ((c^i = v_1 \wedge d^i = v_2 \wedge F_f^{d^i}(v_1)) \vee (c^i = v_2 \wedge d^i = v_1 \wedge B_f^{d^i}(v_2))) \end{aligned}$$

Lemma 5.7 *Let φ be an MSO formula. There is a graph $S \in \mathcal{A}_k$ such that $S \models \varphi$ if and only if there is a Σ-labeled tree $T \in T_\Sigma$ such that $T \models TR_3(\varphi)$.*

Proof: Given a graph S in $\mathcal{A}_k$, we can encode it as a Σ-labeled tree T as follows. First, remove all self loops and at most k additional edges from the Gaifman graph of S to obtain an acyclic undirected graph, U. It is easy to transform the undirected graph U into a directed forest T, by choosing one node in every connected component of U as a root, and directing all edges from it downwards. Then, we can set the labels of T uniquely from the labels of the corresponding nodes in S. To encode that an edge in S is labeled with f, we identify the corresponding edge in T, and label the target of the edge with a unary relation to remember the label f.

Given $T \in T_\Sigma$, we can uniquely reconstruct the graph $S \in \mathcal{A}_k$ that corresponds to it. Every node in T that is labeled with F_f has exactly one incoming edge, which defines the corresponding edge in S, labeled with f. For each $F_f^{d^i}$, at most one edge can be created in S, because TR_3 guarantees that in T the source is labeled with c^i, and the target is labeled with d^i, which are constants.

Theorem 5.8 *The satisfiability problem of MSO formulas is decidable on $\mathcal{A}_k$.*

Proof: Follows from Lemma 5.6 and [41].

5.3 Normal Form of $\mathcal{L}_0$ Formulas

We define a normal-form formula to be a disjunction of conjunctions of formulas of the form $c\langle R\rangle c'$ and $c[R]p$.

Definition 5.9 (Normal-form formulas) *A formula in $C\mathcal{L}_0$ is of the form*

$$\bigwedge_i \neg(c_i[R_i]\neg c_i') \wedge \bigwedge_j c_j[R_j]p_j$$

A normal-form formula is a disjunction of $C\mathcal{L}_0$ formulas.

A formula φ is in $C\mathcal{L}_1$ if and only if $\varphi \in C\mathcal{L}_0$ and $\varphi \in \mathcal{L}_1$, i.e., all the patterns that appear in φ satisfy the requirement of Def. 4.1.

For a formula $\varphi \in C\mathcal{L}_0$, we use $\varphi_\diamond$ to denote the first part of φ, namely $\bigwedge_i \neg c_i[R_i]\neg c_i'$, and $\varphi_\Box$ to denote the second part of φ, namely $\bigwedge_j c_j[R_j]p_j$. We use $|\varphi_\diamond|$ to denote the number of conjuncts in the formula $\varphi_\diamond$.

Note that while $\mathcal{L}_0$ is closed under negation, $C\mathcal{L}_0$ is not. The following theorem shows that every $\mathcal{L}_0$-formula can be effectively translated into an equi-satisfiable normal-form formula. The main difficulty is to translate a formula of the form $\neg c[R]p$, where p is an arbitrary pattern, into a formula in which negation appears only in front of constraints of the form $c'[R]\neg c''$.

Definition 5.10 *Let θ be the formula $\neg c[R]p$ over τ, where $p(v_0) = N(v_0, \ldots, v_n) \Rightarrow \psi(v_0, \ldots, v_n)$. We introduce new constant symbols $c_0, \ldots, c_n$, and define $\tau' = \tau \cup \{c_0, \ldots, c_n\}$. We define $tr(\theta)$ as follows:*

- *Translate $\neg\psi$ into an equivalent negated normal form formula ψ',*
- *Let θ' be $c\langle R\rangle c_0 \wedge N(c_0, \ldots, c_n) \wedge \psi'(c_0, \ldots, c_n)$, where every edge formula $v_i \xrightarrow{f} v_j$ that appears in N or ψ' is replaced by $c_i\langle \xrightarrow{f}\rangle c_j$.* [7]
- *If $\neg c\langle R\rangle c'$ appears in θ', replace it with $c[R]\neg c'$, to obtain θ''.*
- *Transform θ'' into an equivalent disjunctive normal form formula θ'''.*
- *Let $tr(\theta)$ be θ'''.*

The formula $tr(\theta)$ is a normal-form formula by Def. 5.9, because it is a disjunction of $C\mathcal{L}_0$-formulas. In fact, $tr(\theta)$ is a very simple formula: all the patterns in it are of the form $true \Rightarrow c \neq v_0$. Thus, negation can appear only in front of reachability constraints of the form $c[R]\neg c'$ where R does not contain the Kleene star operator.

[7] Recall from Section 2.1.1 that $c\langle R\rangle c'$ is a shorthand for $\neg c[R]\neg c'$.

Lemma 5.11 *For a graph S over τ, if S satisfies θ, then there exists an expansion of S to τ', that satisfies $tr(\theta)$. For a graph S' over τ', if $S' \models tr(\theta)$ then the restriction S of S' to τ satisfies φ.*

Theorem 5.12 *There is a computable translation TR_1 from $\mathcal{L}_0$ to a disjunction of formulas in $C\mathcal{L}_0$ that preserves satisfiability.*

Sketch of Proof: For every formula $\varphi \in \mathcal{L}_0$ over τ, the formula $TR_1(\varphi)$ is a disjunction of formulas in $C\mathcal{L}_0$ over τ' such that φ is satisfiable if and only if $TR_1(\varphi)$ is satisfiable. The vocabulary τ' is an extension of τ with new constant symbols. The translation $TR_1(\varphi)$ is defined as follows:

(1) Translate φ into an equivalent formula φ' in negated normal form using de-Morgan rules to push negations inwards.
(2) Replace every sub-formula $\neg c[R]p$ that appears in φ' with $tr(\neg c[R]p)$, as in Def. 5.10. The resulting formula φ'' is satisfiable if and only if φ' is satisfiable, by Lemma 5.11. Note that this translation only preserves satisfiability (not equivalence).
(3) Translate φ'' into an equivalent disjunctive normal form formula φ'''. All atomic formulas are of the form $c[R]\neg c'$.
    The result of $TR_1(\varphi)$ is φ'''.

The translation is applicable to the full $\mathcal{L}_0$ logic, in which case the reachability constraints in $\varphi_\Box$ can contain arbitrary patterns.

The translation TR_1 may introduce only patterns of the form $true \Rightarrow c_2 \neq v_0$ beyond those patterns that appear in the input formula. This observation yields the following corollary:

Corollary 5.13 *For $\varphi \in \mathcal{L}_1$, the translation TR_1 returns a disjunction of formulas in $C\mathcal{L}_1$ (and preserves satisfiability).*

5.4 Decidability of $\mathcal{L}_1$

The following theorem states that $C\mathcal{L}_1$ has an Ayah-model property, i.e., every satisfiable $C\mathcal{L}_1$ formula φ has a model in $\mathcal{A}_{f(\varphi)}$ where $f(\varphi)$ is defined by

$$f(\varphi) \overset{\text{def}}{=} 2 \times n \times |C| \times |\varphi_\Diamond| \tag{4}$$

Here, we assume that for every routing expression that appears in $\varphi_\Diamond$ there is an equivalent automaton with at most n states.

Theorem 5.14 (Ayah model property of $\mathcal{L}_1$) *If $\varphi \in C\mathcal{L}_1$ is satisfiable, then φ is satisfiable by a graph in $\mathcal{A}_{f(\varphi)}$, where f is defined in (4).*

A non-trivial proof of this theorem is presented in Section 6.

Theorem 5.15 *The satisfiability problem of $\mathcal{L}_1$ is decidable.*

Proof: Follows from combining the results of Theorem 5.12, Theorem 5.14, Lemma 5.2, Theorem 5.8.

6 Ayah Model Property of $\mathcal{L}_1$

In this section we provide a detailed proof of the main technical theorem of the paper, Theorem 5.14. Before diving into the details, we explain the main proof at a high-level.

Given a normal-form formula $\varphi \in C\mathcal{L}_1$ and a graph S such that $S \models \varphi$, we construct a graph S' and show that $S' \models \varphi$ and $S' \in \mathcal{A}_k$.

The construction operates as follows. We construct a pre-model S_0 of S and φ, which satisfies all constraints of the form $c\langle R\rangle c'$ in φ. The idea is to extract from S a witness path for each constraint of the form $c\langle R\rangle c'$ in φ, and define S_0 to be the union of these witness paths (Section 6.5).

The pre-model S_0 may violate some of the constraints of the form $c[R]p$ in φ. Consider the case when the pattern p contains a positive occurrence of edge formula or equality formula. If a graph G violates a constraint $c[R]p$, then there is an enabled merge operation or edge-addition operation, depending on the pattern p (Section 6.3).

For example, if p is of the form $N(v_0, v_1, v_2) \Rightarrow v_1 = v_2$, it defines a merge operation. We say that this merge operation is enabled in a graph G (by $c[R]p$) when G contains a node w_0 reachable by an R-path from c and *distinct* nodes w_1 and w_2 forming the neighborhood $N(w_0, w_1, w_2)$. Applying this operation means merging the nodes w_1 and w_2. After merging w_1 and w_2, other merge operations may still be enabled in G by $c[R]p$. If there are no more enabled operations in G, then $G \models c[R]p$. Similarly, if p is of the form $N(v_0, v_1, v_2) \Rightarrow v_1 \xrightarrow{f} v_2$, it defines an edge-addition operation. Applying this operation means adding an f-edge.

Given a pre-model S_0, we apply all enabled operations in any order, producing a sequence of distinct graphs $S_0, S_1, \ldots$ until the last graph S' has no enabled operations. Thus, S' satisfies all constraints of the form $c[R]p$ where p contains a positive occurrence of edge formula or equality formula. We show that applying any enabled operation preserves witness paths for the constraints of the form $c\langle R\rangle c'$. Thus, S' also satisfies all constraints of the form $c\langle R\rangle c'$. This construction also guarantees that S' satisfies all the constraints of the form $c[R]p$ where p is a negative formula. To show this formally, we use homomorphisms (Section 6.4) which preserves existence of edges and both existence and absence of labels on nodes (preserving absence of labels is non-standard).

Finally, the fact that S' is in $\mathcal{A}_k$ is proved by induction. By construction, S_0 is in $\mathcal{A}_k$ (Lemma 6.11), and $\mathcal{A}_k$ is closed under operations enabled by $\mathcal{L}_1$ formulas (Lemma 6.5). The proof of closure properties of $\mathcal{A}_k$ is based on closure properties for a class of undirected graphs, T^k (Lemma 6.1).

The rest of the section describes the building blocks of the proof of Theorem 5.14: closure properties of T^k (Section 6.1), closure properties of $\mathcal{A}_k$ (Section 6.2), the definition of operations enabled by $\mathcal{L}_1$ formulas (Section 6.3), the definition of homomorphism relation and its properties (Section 6.4), and the definition of witness splitting and properties of a pre-model (Section 6.5). The proof of Theorem 5.14 concludes the section.

6.1 Trees with Extra Edges

Recall from Def. 5.3 that T^k is a set of undirected graphs that are trees with k extra edges. In this section we prove that T^k is closed under merging of vertices at distance at most 2.

The *distance* between the vertices v_1 and v_2 in an undirected graph B is the number of edges on the shortest path between v_1 and v_2 in B.

Merging two vertices in an undirected graph is defined in the usual way, by gluing these vertices. Formally, let the undirected graph B' denote the result of merging nodes v_1 and v_2 in B. The set of vertices of B' is $V^{B'} \stackrel{\text{def}}{=} (V^B \setminus \{v_1, v_2\}) \cup \{v_{12}\}$, where v_{12} is a new vertex. Let $m\colon V^B \to V^{B'}$ be defined as follows:

$$m(v) = \begin{cases} v_{12} & \text{if } v = v_1 \text{ or } v = v_2 \\ v & \text{otherwise} \end{cases}$$

If there is an edge e between the vertices v_1 and v_2 in B then there is an edge $m(e)$ between $m(v_1)$ and $m(v_2)$ in B. If there is an edge e between v_1' and v_2' in B' then there exist vertices v_1 and v_2 in B such that $m(v_1) = v_1'$, $m(v_2) = v_2'$, and there is an edge between v_1 and v_2 in B.

Lemma 6.1 *Assume that B is in T^k and vertices v_1 and v_2 are at distance at most two in B. The graph B' obtained from B by merging v_1 and v_2 in B is also in T^k.*

Proof: By definition of T^k, there exists a set of edges $D \subseteq E$ such that $B \setminus D$, denoted by T, is acyclic and $|D| \leq k$. We show how to transform D into $D' \subseteq E'$ such that $B' \setminus D'$, denoted by T', is acyclic and $|D'| \leq k$. We consider only the case when v_1 and v_2 are at distance of exactly two in B, i.e., there is a vertex v_0 distinct form v_1 and v_2, an edge e_1 between v_1 and v_0, and an edge e_2 between v_0 and v_2. We consider three cases, depicted in Fig. 7.

- If $e_1, e_2 \notin D$, let $D' = \{m(e) | e \in D\}$.
- Assume that $e_1 \notin D$ and $e_2 \in D$. If v_2 is not reachable from v_1 in T, let $D' = \{m(e) | e \in D\}$, thus $|D'| \leq k$.

  If v_2 is reachable from v_1 in T, there is at most one (simple) path from v_1 to v_2 in T, because T is acyclic. If the path contains e_1, we define D' as before: $D' = \{m(e) | e \in D\}$.

  If the path from v_1 to v_2 does not contain e_1, let e_3 be the first edge on the path from v_1 to v_2 (see the second case in Fig. 7).[8] To obtain D' from D, we remove e_2 and add e_3: $D' = (\{m(e) | e \in D\} \setminus \{m(e_2)\}) \cup \{m(e_3)\}$. The size of D' is the same as the size of D, because $e_2 \in D$.
- Assume that $e_1, e_2 \in D$. If v_2 is not reachable from v_1, we can use the simple construction $D' = \{m(e) | e \in D\}$. It follows that $|D'| = |D| - 1$, because both e_1 and e_2 are mapped to the same edge $e' = m(e_1) = m(e_2)$, and no multiple edges are allowed.

  If v_2 is reachable from v_1, let e_3 be the first edge on the path. We define $D' = \{m(e) | e \in D\} \cup \{m(e_3)\}$ (see the third case in Fig. 7). Same construction applies when v_1 or v_2 are reachable from v_0.

6.2 *Ayah Graphs*

In this section we prove that $\mathcal{A}_k$ is closed under edge-addition operations at distance at most one (Lemma 6.2), and under merge operations at distance at most 2 (Lemma 6.3).

The *distance* between nodes v_1 and v_2 in a graph S is the distance between v_1 and v_2 in $\mathcal{G}(S)$, i.e., the number of edges on the shortest path between v_1 and v_2 in $\mathcal{G}(S)$.

It is easy to see that $\mathcal{A}_k$ is closed under edge-addition operations at distance at most one, which means adding an edge in parallel to an existing one (distance one) or adding a self-loop (distance zero).

Lemma 6.2 (Adding edges at distance ≤ 1 in $\mathcal{A}_k$) *Assume that the graph S' is obtained from S by adding an edge from v_1 to v_2 in S. If S is in $\mathcal{A}_k$ and nodes v_1 and v_2 are at distance at most 1 in S, then S' is in $\mathcal{A}_k$.*

[8] Note that we cannot use the simple D' definition as before, because merging v_1 and v_2 in T to obtain T' creates a cycle that does not involve e_1. We observe that, in this case, the subgraph reachable from v_1 through e_1 in T remains acyclic after the merge operation, because it is disjoint from the subtree of v_2. Thus, e_1 need not be removed from T.

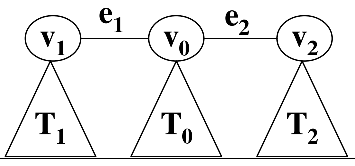
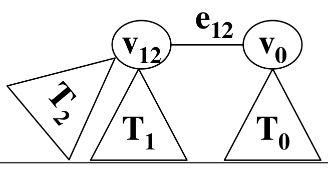
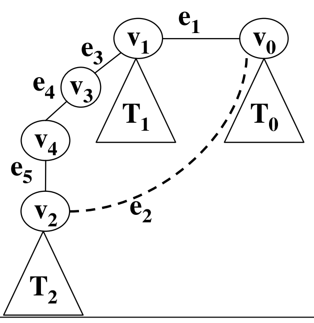
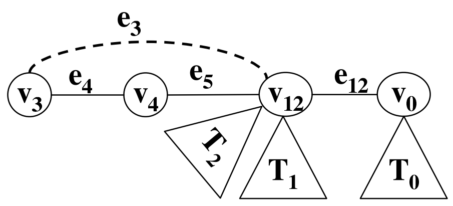
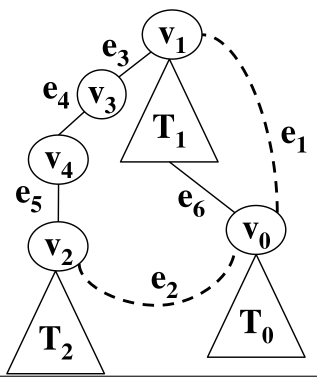
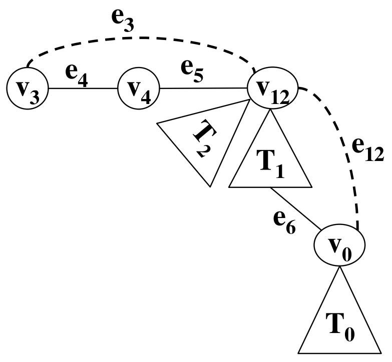
	T	T'
$e_1, e_2 \notin D$		
$e_1 \notin D, e_2 \in D$		
$e_1, e_2 \in D$		

Fig. 7. Merge operation on T^k-graphs. Dotted lines represent additional edges, i.e., edges of a T^k-graph that do not belong to the tree. The vertex v_{12} and the edge e_{12} in T' result from merging the vertices v_1 and v_2, and the edges e_1 and e_2 in T.

Proof: Distance at most 1 between v_1 and v_2 means that there is already an edge between v_1 and v_2. Addition of edges to S in parallel to existing edges does not affect the $\mathcal{G}(S)$, and self-loops do not affect T^k.

Merging two nodes in a graph is defined in the usual way by gluing these nodes. Formally, let S' be the result of merging the nodes v_1 and v_2 in S. The set of nodes of S' is $V^{S'} \overset{\text{def}}{=} (V^S \setminus \{v_1, v_2\}) \cup \{v_{12}\}$, where v_{12} is a new node. We define $m\colon V^S \to V^{S'}$ as follows:

$$m(v) = \begin{cases} v_{12} & \text{if } v = v_1 \text{ or } v = v_2 \\ v & \text{otherwise} \end{cases}$$

The interpretation of constant and relation symbols in S' is defined as follows:

(1) For every constant symbol $c \in \tau$, and for every node $v \in S$, v is labeled with c in S if and only if $m(v)$ is labeled with c in S'.
(2) For every unary relation symbol $\sigma \in \tau$, and for every node $v \in S$, if v is labeled with σ in S then $m(v)$ is labeled with σ in S'.
(3) For every unary relation symbol $\sigma \in \tau$, and for every node $v' \in S'$, if v' is labeled with σ in S' then there exists a node v in S such that $m(v) = v'$ and v is labeled with σ in S.
(4) For every binary relation symbol $\sigma \in \tau$, and every pair of nodes $w_1, w_2 \in S$, if there is an edge from w_1 to w_2 labeled with σ then there is an edge from $m(w_1)$ to $m(w_2)$ in S' labeled with σ.
(5) for every binary relation symbol $\sigma \in \tau$, and every pair of nodes $w'_1, w'_2 \in S'$, if there is an edge from w'_1 to w'_2 labeled with σ in S' then there are nodes w_1 and w_2 in S such that $m(w_1) = w'_1$, $m(w_2) = w'_2$, and there is an edge from w_1 to w_2 in S labeled with σ.

Later, we guarantee that merge operations are applied only to those nodes which are labeled by the same unary relations and constants.

The proof that $\mathcal{A}_k$ is closed under merge operations at distance at most two is based on the result of Lemma 6.1 from the previous section.

Lemma 6.3 (Merging nodes at distance ≤ 2 in $\mathcal{A}_k$) *Assume that the graph S' is obtained from S by merging v_1 and v_2 in S. If S is in $\mathcal{A}_k$ and nodes v_1 and v_2 are at distance at most 2 in S, then S' is in $\mathcal{A}_k$.*

Proof: To show that $S' \in \mathcal{A}_k$, it is sufficient to show that $\mathcal{G}(S') \in T^k$. We use the definitions of a Gaifman graph and a merging operation. First, merging the nodes of $\mathcal{G}(S)$ that correspond to v_1 and v_2 in $\mathcal{G}(S)$, results in $\mathcal{G}(S')$. Second, the distance between v_1 and v_2 in $\mathcal{G}(S)$ is at most 2 because the distance between the corresponding nodes in S is at most 2. Third, $\mathcal{G}(S) \in T^k$, because $S \in \mathcal{A}_k$. Thus, using Lemma 6.1, we get that $\mathcal{G}(S') \in T^k$.

6.3 Graph Operations Enabled by $\mathcal{L}_1$ Formulas

The notion of enabled operations defined in this section is used for defining the construction in the proof of Theorem 5.14.

Let $p(v_0) \stackrel{\text{def}}{=} N(v_0, \ldots, v_n) \Rightarrow \psi(v_0, \ldots, v_n)$ be an $\mathcal{L}_1$ pattern. Let S be a graph, and w_1, w_2 nodes in S.

We say that *merge operation of w_1 and w_2 is enabled* (by $c[R]p$) when (a) the equality between variables $(v_1 = v_2)$ appears positively in ψ, (b) we can assign nodes $w_0, \ldots, w_n$ to $v_0, \ldots, v_n$, respectively, such that there is an R-path from c to w_0, $N(w_0, \ldots, w_n)$ holds but $\psi(w_0, \ldots, w_n)$ does not hold, and (b) w_1 and w_2 are

distinct nodes. Merging the nodes w_1 and w_2 disables this merge operation (other merge operations may still be enabled after merging w_1 and w_2).

We say that *edge-addition between w_1 and w_2 is enabled* (by $c[R]p$) when (a) the edge formula $(v_1 \xrightarrow{f} v_2)$ appears positively in ψ, (b) we can assign nodes $w_0, \ldots, w_n$ to $v_0, \ldots, v_n$, respectively, such that there is an R-path from c to w_0, $N(w_0, \ldots, w_n)$ holds but $\psi(w_0, \ldots, w_n)$ does not hold, and (c) there is **no** f-edge from w_1 to w_2. We can add an f-edge from w_1 and w_2 to discharge this assignment.

Lemma 6.4 *Let $N(v_0, \ldots, v_n)$ be a neighborhood formula, and S be a graph with an assignment to $v_0, \ldots, v_n$ that satisfies N. If the variables v_1 and v_2 are at distance at most k in N, then the nodes assigned to v_1 and v_2 are at distance at most k in S.*

Proof: Follows from the definition of neighborhood as a conjunction of edges (Def. 2.2).

The following lemma is the key observation of the proof.

Lemma 6.5 *Let $p(v_0) \stackrel{\text{def}}{=} N(v_0, v_1, \ldots, v_n) \Rightarrow \psi(v_0, \ldots, v_n)$ be an $\mathcal{L}_1$ pattern. Let S be a graph, and w_1, w_2 nodes in S. Assume that a merge (an edge-addition) operation is enabled in a graph S between nodes w_1 and w_2 by a reachability constraint $c[R]p$. If $S \in \mathcal{A}_k$, then the result of merging (adding an edge) between w_1 and w_2 is a graph in $\mathcal{A}_k$.*

Proof: Suppose that a merge operation is enabled in S between nodes w_1 and w_2. It is possible to assign nodes $w_0, \ldots, w_n$ to the variables $v_0, \ldots, v_n$, such that N holds. In particular, w_1 is assigned to v_1 and w_2 is assigned to v_2, and the equality $v_1 = v_2$ appears positively in ψ. According to the equality restriction on $\mathcal{L}_1$ patterns, v_1 and v_2 are at distance at most 2 in N. By Lemma 6.4, w_1 and w_2 are at distance at most 2 in S. Thus, by Lemma 6.3 we get that the result of merging w_1 and w_2 is a graph in $\mathcal{A}_k$, because S is in $\mathcal{A}_k$. The proof for edge-addition is similar, using Lemma 6.2.

6.4 Homomorphism Preservation

In this section, we give a slightly non-standard definition of homomorphism between graphs. It preserves existence of edges and both existence and absence of labels on nodes (preserving absence of labels is non-standard). The homomorphism relation is preserved by $C\mathcal{L}_1$ formulas, and also by merging operations.

Definition 6.6 (Homomorphism) *Let S_1 and S_2 be graphs over the same vocabulary τ. A homomorphism from S_1 to S_2 is a mapping $h \colon V^{S_1} \to V^{S_2}$ such that*

(1) for every constant symbol and unary relation symbol $\sigma \in \tau$, and for every

$v \in S_1$, v is labeled with σ in S_1 if and only if $h(v)$ is labeled with σ in S_2.

(2) for every binary relation symbol $\sigma \in \tau$, and every pair of nodes $v_1, v_2 \in S_1$, if there is an edge from v_1 to v_2 in S_1 labeled with σ, then there is an edge from $h(v_1)$ to $h(v_2)$ in S_2 labeled with σ.

Lemma 6.7 *Let $h\colon S_1 \to S_2$ be a homomorphism. If $S_1 \models c_1\langle R\rangle c_2$ then $S_2 \models c_1\langle R\rangle c_2$. Dually, if $S_2 \models c[R]p$, and p does not contain positive occurrences of edge formulas or equality formulas, then $S_1 \models c[R]p$.*

Sketch of Proof: If $S_1 \models c_1\langle R\rangle c_2$, there exists an R-path from c_1 to c_2. By definition of homomorphism from S_1 to S_2, the same path exists in S_2. Thus, $S_2 \models c_1\langle R\rangle c_2$.

For the sake of contradiction, assume that $S_2 \models c[R]p$ but $S_1 \not\models c[R]p$. That is, there exists an R-path from c to some node v in S_1 and v does not satisfy the pattern p. The same path exists in S_2, due to the homomorphism from S_1 to S_2. To obtain a contradiction, we show that $h(v)$ does not satisfy the pattern p in S_2. The formula p is of the form $N \Rightarrow \psi$, where N contains only positive occurrences of edge formulas. By assumption, we get that ψ does not contain positive occurrences of edge formulas or equality formulas. Thus, the formula p does not contain positive occurrences of edge formulas and equality formulas. If S_1 does not satisfy p, there exists a subgraph in S_2 which satisfies $\neg p$. This subgraph exists in S_2 as well, due to homomorphism. [9] Thus, S_2 satisfies $\neg p$, and a contradiction is obtained.

Lemma 6.8 *Assume that f is a homomorphism from S_1 to S, and S_2 is obtained by merging the nodes v_1 and v_2 in S_1. If $f(v_1) = f(v_2)$ then there is a homomorphism from S_2 to S.*

6.5 Witness Splitting

A *witness W for $c_1\langle R\rangle c_2$ in a graph S*, is a path in S, labeled with a word $w \in L(R)$, from the node labeled with c_1 to the node labeled with c_2. Note that the nodes and edges on a witness path for R need not be distinct. S contains a witness for $c_1\langle R\rangle c_2$ if and only if $S \models c_1\langle R\rangle c_2$.

Using a witness W for $c_1\langle R\rangle c_2$ in S, we construct a graph W' that consists of a path, also labeled with w, that starts at the node labeled by c_1 and ends at the node labeled by c_2. Intuitively, we create W' by duplicating a node of S each time the witness path W traverses it, unless the node is labeled with a constant. The nodes in W' are named $t_{v,l}$ where v is a node in S and $l \geq 0$ is an integer. For $l > 0$, a node $t_{v,l}$ in W' corresponds to the l-th occurrence of v on the witness path W, if a node v in S is not labeled with a constant. If v is labeled with a constant, we create for it

[9] Note that $\neg p$ may contain negative occurrences of unary formulas, but these are also preserved under the (non-standard) homomorphism relation we are using.

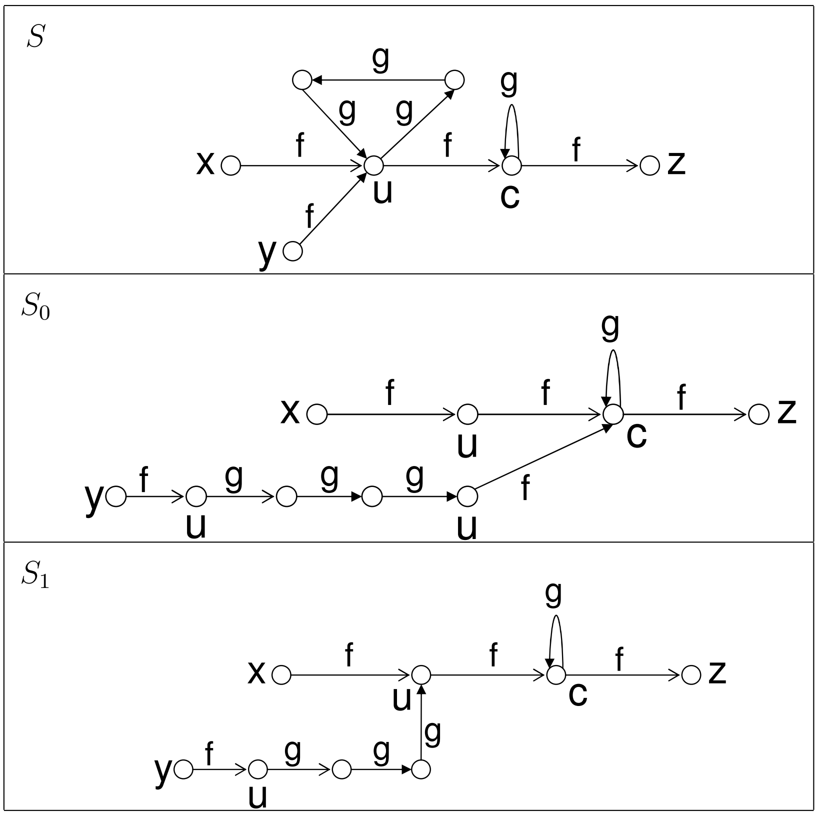


Fig. 8. The graph S satisfies the formula in (5), and $S \in \mathcal{A}_1$. A pre-model of S is S_0. Note that $S_0 \in \mathcal{A}_0$. The graph S_1 is the result of applying a merge operation to S_0. Note that S_1 satisfies the formula in (5), and $S_1 \in \mathcal{A}_0$. The graph S_1 is the final result of the construction used in the proof of Theorem 5.14.

a unique node $t_{v,0}$ in W' even if v is traversed several times by W. As a result, all shared nodes in W' are labeled with constants. Also, every cycle contains a node labeled with a constant. By construction, W' satisfies $c_1\langle R\rangle c_2$.

For example, consider the formula

$$\varphi \stackrel{\text{def}}{=} x\langle \xrightarrow{f}^{*}\rangle z \wedge y\langle \xrightarrow{f}.(\xrightarrow{g}^{+}.(c|u).\xrightarrow{f})^{*}\rangle z \wedge c[\epsilon]uns_f \tag{5}$$

where u is a unary relation symbol and c is a constant symbol. Fig. 8 shows a graph S which satisfies φ. The shortest witness path for $x\langle \xrightarrow{f}^{*}\rangle z$ is labeled with the word $\xrightarrow{f}.\xrightarrow{f}.\xrightarrow{f}$. The shortest witness path for $y\langle \xrightarrow{f}.(\xrightarrow{g}^{+}.(c|u).\xrightarrow{f})^{*}\rangle z$ is labeled with the word $\xrightarrow{f}.\xrightarrow{g}.\xrightarrow{g}.\xrightarrow{g}.u.\xrightarrow{f}.\xrightarrow{g}.c.\xrightarrow{f}$. Note that this witness traverses each of the nodes labeled by u and by c twice. To split this witness, the node marked by u is duplicated, while the node marked by c is not duplicated, because c is a constant. After splitting the witnesses, we construct a pre-model of S, denoted by S_0, by taking the union of both witness paths and merging the nodes of the different witness paths which are labeled with the same constant.

Formally, the witness path W is a sequence of nodes from S: $t_1, t_2, \ldots, t_r$, where $t_i \in S$. Let $C(t_i)$ denote the set of constant symbols that label the node t: $C(t_i) \stackrel{\text{def}}{=}$

$\{\sigma \in C | C^S(\sigma) = t_i\}$. We define a mapping $d(t_i)$ as follows:

$$d(t_i) \overset{\text{def}}{=} \begin{cases} t_{v,0} & \text{if } C(t_i) \neq \emptyset \text{ and } t_i \text{ is the node } v \\ t_{v,l} & \text{if } t_i \text{ is the } l\text{-th occurrence of the node } v \in S \text{ on the path } W \end{cases}$$

W' is a graph with nodes $\{d(t_1), \ldots, d(t_r)\}$. If the witness path W goes from t_i to t_{i+1} through an edge labeled with $f_i \in F$, then there is an edge in W' labeled with f_i from $d(t_i)$ to $d(t_{i+1})$. Note that W' contains only edges traversed by the witness path. For every unary relation and constant symbol $\sigma \in C \cup U$ and node $t_i \in W$, $d(t_i)$ is labeled with σ in W' if and only if t_i is labeled with σ in S.

We say that W' is the result of *splitting* the witness W. We say that W is the *shortest witness* for $c_1\langle R\rangle c_2$ if any other witness path for $c_1\langle R\rangle c_2$ is at least as long as W.

For a formula $\varphi \in C\mathcal{L}_1$ and a graph S such that $S \models \varphi$, we define a *pre-model of a S and φ* to be the graph S_0 constructed as follows.

- Let W_i denote a shortest witness in S for every $c_i\langle R\rangle c_i'$ in $\varphi_\diamond$.
- Let W_i' be the result of splitting the witness W_i. Let $t^i_{v,l}$ be the names the nodes of W_i'.
- Let S_0' be a disjoint union of all W_i's.
- For every $c \in C$, if S_0' does not contain any node labeled with c, add a new node $t^0_{v,0}$ to S_0', where v is the node in S labeled with c. For all $\sigma \in C \cup U$, $t^0_{v,0}$ is labeled with σ in S_0' if and only if v is labeled with σ in S.
- The graph S_0 is the result of merging all nodes that are labeled with the same constants, i.e., nodes $t^i_{v,0}$ for all i are merged and the new node named $t^0_{v,0}$.

Note that S_0' cannot be used as a legal interpretation for $\mathcal{L}_0$ formulas over τ, because it may contain several nodes labeled with the same constant, or no interpretation for some constants. These problems are addressed by the last two steps of the construction.

By construction, S_0 contains a witness for each $c_1\langle R\rangle c_2$ in $\varphi_\diamond$.

Lemma 6.9 *If $S \models \varphi$ and S_0 is a pre-model of S and φ, then $S_0 \models \varphi_\diamond$.*

Lemma 6.10 *Let S_0 be a pre-model of S and φ. There is a homomorphism $h_0 \colon S_0 \to S$ defined by $h_0(t^i_{v,l}) = v$.*

Proof: We define $h_0' \colon S_0' \to S$ by $h_0'(t^i_{v,l}) = v$. The mapping h_0' preserves existence of edges and the presence and absence of node labels between S_0' and S because it is preserved for every W' separately, by definition of witness splitting, and S_0' is a *disjoint* union of W_i's. Thus, h_0' is a homomorphism.

Because S_0 is obtained from S_0' by merging nodes that are mapped by h_0' to the same node in S, the mapping h_0 is also a homomorphism, by Lemma 6.8.

Lemma 6.11 *For $\varphi \in C\mathcal{L}_1$, if S_0 is a pre-model of S and φ, then $S_0 \in \mathcal{A}_{f(\varphi)}$, where f is defined in (4).*

Proof:

Recall that for every routing expression that appears in $\varphi_\diamond$ there is an equivalent automaton with at most n states. If a node is visited more than once in the same state of the automaton, the path can be shortened by removing the part traversed between the two visits. Thus, a shortest witness visits a node at most n times. In the worst case, each time a shortest witness visits a node, it enters and exits the node with a different edge. Because S_0 consists of $|\varphi_\diamond|$ shortest witnesses, there are at most $2 \times n \times |\varphi_\diamond|$ edges adjacent to any node.

In fact, by construction of S_0, only nodes labeled by constants in S_0 can have more than two adjacent edges. Thus, every (simple) cycle in S_0 must go through a constant. To break all cycles in S_0 (and, thus, in its Gaifman graph), it is sufficient to remove all the edges adjacent to nodes labeled with constants, i.e., at most $k = 2 \times n \times |\varphi_\diamond| \times |C|$ edges. It follows that $S_0 \in \mathcal{A}_k$. [10]

6.6 $\mathcal{A}_k$-Model Property of $\mathcal{L}_1$

Theorem 5.14(Ayah model property of $\mathcal{L}_1$) *If $\varphi \in C\mathcal{L}_1$ is satisfiable, then φ is satisfiable by a graph in $\mathcal{A}_{f(\varphi)}$, where f is defined in (4).*

Proof: Given a graph S such that $S \models \varphi$, we construct a graph S' and show that $S' \in \mathcal{A}_k$ and $S' \models \varphi$.

First, we construct a pre-model S_0 of S and φ, and define the mapping $h_0 \colon S_0 \to S$ according to Lemma 6.10. Then, we apply all enabled merge operations and all enabled edge-addition operations in any order, producing a sequence of distinct graphs $S_0, S_1, \ldots, S_r$, until S_r has no enabled operations. The result $S' = S_r$.

Formally, for every $c[R]p \in \varphi$ and ever pair of nodes $w_1, w_2 \in S_j$,

- If a merge operation is enabled, and $h_j(w_1) = h_j(w_2)$ in S_j then construct S_{j+1} by merging w_1 and w_2, and define $h_{j+1} \colon S_{j+1} \to S$ to be $h_{j+1}(w) = h_j(w_1)$ if w is the result of merging w_1 and w_2, otherwise $h_{j+1}(w) = h_j(w)$.
- If an edge-addition operation is enabled for $f \in F$, and there is an f-edge from $h_j(w_1)$ to $h_j(w_2)$ in S then construct S_{j+1} by adding an f-edge from w_1 to w_2, and define $h_{j+1} \colon S_{j+1} \to S$ to be the same as h_j.

[10] This bound is not tight.

For example, the pre-model S_0 shown in Fig. 8 does not satisfy the constraint $c[\epsilon]uns_f$ from (5), which requires that the node labeled with c have at most one incoming f-edge. The result of applying the corresponding merge operation is the structure S_1, also shown in Fig. 8.

An enabled merge operation is not applied to S_j if the corresponding nodes in the original model S are distinct. Similarly, an enabled edge-addition is not applied, unless the corresponding edge is present in S. This allows us to deal with disjunctions in patterns. For example,

$$\textbf{let } p(v_0) \stackrel{\text{def}}{=} (v_0 \xrightarrow{f} v_1) \Rightarrow (v_0 = v_1 \vee (v_0 \xrightarrow{g} v_1) \vee (v_0 \xrightarrow{g'} v_1)) \textbf{ in}$$
$$c\langle \xrightarrow{f}^* \rangle c' \wedge c[\xrightarrow{f}^*]p \wedge (c \neq c')$$

Suppose that S_0 looks like this: $$\begin{tikzcd} w_1 \arrow[r, "f"] & w_2 \\ c & c' \end{tikzcd}$$ The nodes w_1 and w_2 are labeled with the constants c and c', respectively. Both merge and edge-addition operations are enabled in S_0 by $c[\xrightarrow{f}^*]p$. Had we applied the merge operation, we would have immediately obtained a contradiction with $c \neq c'$. However, if we consult the original model, we find out that the corresponding nodes are distinct,[11] but there is a g-edge between them. Therefore, adding a g-edge to S_0 would not lead to a contradiction.

Remark. Even when we consult with S whether to apply an enabled operation or not, we do not merge more than necessary, or add more edges than necessary. In the previous example, after adding g the formula holds, i.e., the edge-addition operation of g' is not enabled any more. However, a different order of application of the enable operations may produce different graphs at the end. Fortunately, it does not affect the size of $\mathcal{A}_k$, or the decidability.

The process described above terminates after a finite number of steps, because in each step either the number of nodes in the graph is decreased (by merge operations) or the number of edges is increased (by edge-addition operations). For a fixed vocabulary and a fixed number of nodes, the number of edges that can be added to the graph is bounded, because a pair of nodes in a graph can have at most one f edge in each direction, for every $f \in F$.

To show that $S' \in \mathcal{A}_k$, we prove a stronger claim that for all j, $S_j \in \mathcal{A}_k$. In particular, it follows that $S' \in \mathcal{A}_k$. Recall that all operations applied in the process above are enabled by $\mathcal{L}_1$ patterns. The key observation of the proof is that $\mathcal{A}_k$ is closed under all operations enabled by $\mathcal{L}_1$ patterns (Lemma 6.5). This is the only place in our proof where we use the distance restriction of $\mathcal{L}_1$ patterns. The proof proceeds by induction on the process described above. Initially, S_0 is in $\mathcal{A}_k$, by

[11] The nodes $h_0(w_1)$ and $h_0(w_2)$ in S are distinct, because our construction of pre-model S_0 does not split nodes labeled by constants.

$$\begin{tikzcd}
 & & & S & & \\
S_0 \arrow[urrr, "h_0"] \arrow[r, "t_1"] \arrow[r, bend right, "f_1"'] \arrow[rr, bend right, "f_2"'] \arrow[rrrr, bend right, "f_{m-1}"'] \arrow[rrrrr, bend right, "f_m"'] & S_1 \arrow[urr, "h_1"] \arrow[r, "t_2"] & S_2 \arrow[ur, "h_2"] & \bullet\bullet\bullet & S_{m-1} \arrow[ul, "h_{m-1}"'] \arrow[r, "t_m"] & S_m \arrow[ull, "h_m"']
\end{tikzcd}$$

Fig. 9. Construction and homomorphisms in the proof of decidability.

Lemma 6.11. By inductive hypothesis, $S_j \in \mathcal{A}_k$. Because S_{j+1} is obtained from S_j by an operation that is enabled by an $\mathcal{L}_1$ pattern, we get that $S_{j+1} \in \mathcal{A}_k$, using Lemma 6.5.

To show that $S' \models \varphi$, we observe that the graphs generated by the process above are related to each other by different homomorphism relations (Def. 6.6), as depicted in Fig. 9.

First, each step of the process can be seen as a transformation t_j from S_{j-1} to S_j, which is defined by an operation applied at step j. That is, t_j is either a merge operation or an edge-addition operation. It is easy to see that both operations are homomorphisms. Therefore, each t_j is a homomorphism, for all j.

Second, we define a mapping f_j from S_0 to S_j as a composition $t_j \circ \ldots \circ t_0$; the mapping f_j is a homomorphism, because it is a composition of homomorphisms. Initially, $S_0 \models \varphi_\diamond$, according to Lemma 6.9. For all S_j, from the existence of a homomorphism f_j from S_0 to S_j we get that $S_j \models \varphi_\diamond$, by Lemma 6.7. In particular, $S' \models \varphi_\diamond$.

Third, we show that for all j, h_j defined by the process above is a homomorphism. Initially, $h_0 \colon S_0 \to S$ is a homomorphism, according to Lemma 6.10. If t_j is a merge operation of w_1 and w_2, then the process applies this operation only if $h_j(w_1) = h_j(w_2)$. From the inductive hypothesis that h_j is a homomorphism, we get that h_{j+1} is a homomorphism, by Lemma 6.8.

For every $c[R]p \in \varphi_\Box$, if p does not contain positive occurrences of edge formulas or equality formulas, then by Lemma 6.7 and the existence of a homomorphism h_r from S' to S, $S' \models c[R]p$, because $S \models c[R]p$.

For the sake of contradiction, assume that the process terminates, but $S' \not\models c[R]p$, where $p(v_0) \overset{\text{def}}{=} N(v_0, \ldots, v_n) \Rightarrow \psi(v_0, \ldots, v_n)$. That is, we can assign nodes $w_0, \ldots, w_n$ to $v_0, \ldots, v_n$, respectively, such that there is an R-path from c to w_0, $N(w_0, \ldots, w_n)$ holds but $\psi(w_0, \ldots, w_n)$ does not hold. Consider the assignment $h_r(w_0), \ldots, h_r(w_n)$ in S. Because homomorphism preserves existences of paths and edges, there is an R-path from c to $h_r(w_0)$, and $N(h_r(w_0), \ldots, h_r(w_n))$ holds. Because $S \models c[R]p$, we know that $\psi(w_0, \ldots, w_n)$ holds. Therefore, there is an

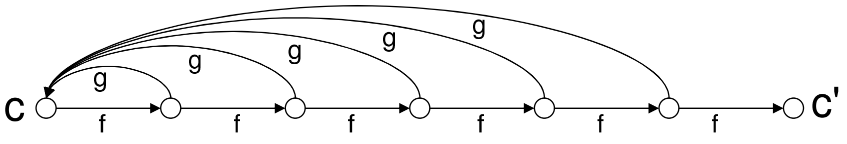


Fig. 10. The graph G_4.

atomic formula θ that appears positively in ψ and evaluates to $false$ in S' and to $true$ in S.

If θ is an equality formula $v_1 = v_2$, then the merge operation of w_1 and w_2 in S' is enabled (because θ is $false$ in S'), and $h(w_1) = h(w_2)$ in S (because θ is $true$ in S), contradiction to the assumption that the process terminated. Similarly, if θ is an edge formula $v_1 \xrightarrow{f} v_2$, then the edge-addition operation of w_1 and w_2 in S' is enabled (because θ is $false$ in S'), and there is an f-edge from $h(w_1)$ to $h(w_2)$ in S (because θ is $true$ in S), contradiction to the assumption that the process terminated. Thus, $S' \models \varphi_\Box$.

7 Decidability of $\mathcal{L}_2$

In this section, we show how to modify the proof of decidability of $\mathcal{L}_1$, to prove the decidability of $\mathcal{L}_2$.

We start by explaining why the proof of Theorem 5.14 does not go through for $\mathcal{L}_2$. Recall that if a graph is in $\mathcal{A}_k$, and an operation that is enabled by an $\mathcal{L}_1$ reachability constraint is applied, then the result is in $\mathcal{A}_k$, due to the distance restrictions in $\mathcal{L}_1$ patterns (see Lemma 6.5). In $\mathcal{L}_2$, this nice property no longer holds.

For example, consider the $\mathcal{L}_2$ constraint

$$\textbf{let } p(v_0) \overset{\text{def}}{=} (v_0 \xrightarrow{f} v_1) \Rightarrow (v_1 \xrightarrow{g} c) \textbf{ in } c[\xrightarrow{f}^*]p$$

Given k, we construct a graph G_k that consists of an f-path of $k+3$ disjoint nodes, but only $k+1$ nodes on the path have a g-edge back to c. Fig. 10 shows G_4. The graph G_k is in $\mathcal{A}_k$, but violates the reachability constraint above. Thus, it has an edge-addition operation enabled for adding a g-edge between the first and the last nodes. It is easy to see that after adding the edge, we get a graph G'_k that is not in $\mathcal{A}_k$.[12]

If the construction of Theorem 5.14 is applied to an $\mathcal{L}_2$ formula, it might generate a graph in which the number of extra edges is proportional to the number of nodes, due to the use of constants in patterns, and not bounded by the size of the formula. The good news is that the extra edges have one of the endpoints labeled with a constant, except, possibly a small number of them. The proof of decidability of $\mathcal{L}_2$

[12] The tree width of $\mathcal{G}(G_k)$ is k and the tree width of $\mathcal{G}(G'_k)$ is $k+1$.

is based on the fact that each extra edge has one of its endpoints labeled with a constant.

We define a graph operation rem that removes all edges to and from nodes labeled with constants. Formally, the result of $rem(S)$ is a graph S' with the same set of nodes as S, such that there is an f-edge from v_1 to v_2 in S' if and only if there is an f-edge from v_1 to v_2 in S and the nodes v_1 and v_2 are not labeled by any constants in S. $\mathcal{A}_k^{rem}$ is the set of graphs on which rem yields a graph in $\mathcal{A}_k$, i.e., $\mathcal{A}_k^{rem} \overset{\text{def}}{=} \{S \mid rem(S) \in \mathcal{A}_k\}$.

7.1 $\mathcal{A}_k^{rem}$-Model Property of $\mathcal{L}_2$

We define graph operations enabled by $\mathcal{L}_2$ formulas (similarly to Section 6.3), and prove that $\mathcal{A}_k^{rem}$ is closed under those operations (similarly to Lemma 6.5).

Let $p(v_0) \overset{\text{def}}{=} N(v_0, \ldots, v_n) \Rightarrow \psi(v_0, \ldots, v_n)$ be an $\mathcal{L}_2$ pattern. Let S be a graph, w_1 be a node in S, and $c_2 \in C$.

We say that *edge-addition between w_1 and c_2 is enabled* (by c[R]p) when (a) $(v_1 \xrightarrow{f} c_2)$ (resp. $(c_2 \xrightarrow{f} v_1)$) appears positively in ψ, (b) we can assign nodes $w_0, \ldots, w_n$ to $v_0, \ldots, v_n$, respectively, such that there is an R-path from c to w_0, $N(w_0, \ldots, w_n)$ holds, but $\psi(w_0, \ldots, w_n)$ does not hold, and (c) there is **no** f-edge from w_1 to the node labeled with c_2 in S (resp. to w_1 from the node labeled with c_2).

Lemma 7.1 *Assume that a graph operation is enabled in a graph S by an $\mathcal{L}_2$ reachability constraint. If $S \in \mathcal{A}_k^{rem}$ then the result of applying the operation is a graph $S' \in \mathcal{A}_k^{rem}$.*

Proof: For graph operations that do not involve constants, the result follows directly from Lemma 6.5.

Assume that $S \in \mathcal{A}_k^{rem}$. Suppose that an edge-addition operation between a node w_1 and c_2 is enabled in a graph S. The graph S' is the result of adding the edge between w_1 and the constant c. In this case, $rem(S)$ and $remS'$ is the same graph. Thus, $S' \in \mathcal{A}_k^{rem}$.

Remark. We can show that $\mathcal{A}_k^{rem}$ is closed under merge operations enabled by a pattern with $v_1 = c$. However, this situation never occurs in the construction used in Theorem 5.14, because we do not split nodes that are labeled with constants, when we create a pre-model.

The following theorem shows that $\mathcal{L}_2$ has $\mathcal{A}_k^{rem}$-property, i.e., every satisfiable $\mathcal{L}_2$ formula has a model in $\mathcal{A}_k^{rem}$. The proof is similar to the proof of Theorem 5.14, except the use of Lemma 7.1 to show that the result $S' \in \mathcal{A}_k^{rem}$.

Theorem 7.2 **($\mathcal{A}_k^{rem}$-Model Property)** *If $\varphi \in \mathcal{L}_2$ is satisfiable, then there exists a graph S such that $S \models \varphi$ and $S \in \mathcal{A}_k^{rem}$, where $k = f(\varphi)$ and f is defined in (4).*

7.2 MSO is decidable on $\mathcal{A}_k^{rem}$

In this section, we show a reduction from the satisfiability problem of MSO logic on $\mathcal{A}_k^{rem}$ to the satisfiability of MSO on $\mathcal{A}_k$, which is decidable by Theorem 5.8. This reduction completes the proof of decidability of $\mathcal{L}_2$.

Lemma 7.3 *There is a translation TR_5 between MSO-formulas such that for every MSO-formula φ, there exists a graph $S \in \mathcal{A}_k^{rem}$ such that $S \models \varphi$ if and only if there exists a graph $S' \in \mathcal{A}_k$ such that $S' \models TR_5(\varphi)$.*

Given the vocabulary $\tau = \langle C, U, F\rangle$ and a number k we define a new vocabulary $\tau' = \langle C, U', F\rangle$, where $U' = U \cup \{F_f^c, B_f^c | f \in F, c \in C\}$.

For an MSO formula φ over τ, $TR_5(\varphi)$ is an MSO formula over the vocabulary τ'. The translation TR_5 is defined inductively on φ, as usual. For a binary relation formula $f \in F$, we define:

$$TR_5(f(v_1, v_2)) = (E(v_1, v_2) \wedge F_f(v_2)) \vee (E(v_2, v_1) \wedge B_f(v_1))$$
$$\bigvee_{c \in C \cup \{d^1, \ldots, d^k\}} (c = v_1 \wedge F_f^c(v_2)) \vee (c = v_2 \wedge B_f^c(v_1))$$

Intuitively, a tree node v is labeled with F_f^c if and only if there is an f-edge from v to the node labeled by c in the corresponding Ayah graph. A tree node v is labeled with B_f^c if and only if there is an f-edge to v from the node labeled by c in the corresponding Ayah graph. This allows us to encode both the direction and the label of the extra edges.

Remark. We have chosen a simple encoding that is not parsimonious in the number of additional unary relations. For example, if an edge has two constants on its adjacent nodes, it can be encoded in more than one way. This ambiguity can be resolved using ordering between constants, but we ignore it here, to simplify the presentation.

Theorem 7.4 *The satisfiability problem of MSO formulas is decidable on $\mathcal{A}_k^{rem}$.*

Proof: Follows from Lemma 7.3 and Theorem 5.8.

Theorem 7.5 *The satisfiability problem of $\mathcal{L}_2$ is decidable.*

Proof: Follows from combining Theorem 5.12, Theorem 7.2, Lemma 5.2, and Theorem 7.4.

8 Complexity

In Section 5, we proved decidability by reduction to MSO on trees, which allows us to check satisfiability of $\mathcal{L}_1$ formulas using MONA decision procedure [26]. Alternatively, we can directly construct a tree automaton from an $\mathcal{L}_1$ formula, and can then check emptiness of the automaton, which yields a double-exponential procedure. [13]

However, a naive translation of $\mathcal{L}_1$ formulas to automata does not yield a practical decision procedure. First, the size of the automaton is exponential in the input vocabulary, regardless of the complexity of the input formula. Second, a naive translation produces *two-way alternating* tree automata. To the best of our knowledge, there are no tools that can check emptiness of such automata. A translation from two-way alternating tree automata to tree automata that can be handled by existing tools, such as MONA [26], Timbuk [18], or H1 [39], is at least exponential.

We are investigating tableaux-based techniques to implement a decision procedure for validity, satisfiability, and model generation for $\mathcal{L}_1$. A tableaux-based decision procedure can be adaptive to specific formulas, and the formulas that come up in practice are quite simple.

The worst case complexity of the satisfiability problem of $\mathcal{L}_1$ formulas is at least NEXPTIME (Section 8.1), but it remains elementary (in contrast to MSO on trees, which is non-elementary [36]). The complexity depends on the bound k of $\mathcal{A}_k$ models, according to Theorem 5.14.

Bounded-Model Property of $\mathcal{L}_1$ We can show that $\mathcal{L}_1$ has a bounded model property: every satisfiable $\mathcal{L}_1$ formula has a model whose size is a (elementary) function of the size of the formula. The translation of $\mathcal{L}_1$ formulas to automata and the finite-model property (Theorem 2.6) yield a double-exponential bound on the size of a model. We believe that it can be improved. Bounded-model property is important for example for guaranteeing termination of tableaux-based decision procedures.

Bounded Branching of $\mathcal{L}_1$ Lemma 6.11 implies that an upper bound on the branching of a node in a Σ-labeled tree is $r = 2 \times n \times \varphi_\diamond \times |C|$. If a node is not labeled with a constant, we can improve the bound to be $2 \times n \times \varphi_\diamond$. The branching does not increase as a result of merging and edge additions enabled by $\mathcal{L}_1$ patterns. Thus, for checking satisfiability of $\mathcal{L}_1$ it is sufficient to consider only Σ-labeled trees with a branching bounded by r.

The Use of Constants in Routing Expressions If the routing expressions do not contain positive occurrences of constant symbols, then the bound k for $\mathcal{L}_1$ does not

[13] The proof is not included in the paper, because we are investigating tighter upper and lower bounds.

depend on the routing expressions:

Theorem 8.1 *Assume that $\varphi \in \mathcal{L}_1$ is satisfiable, and that the routing expressions that appear in φ do not contain positive occurrences of constant symbols. Then, there exists a graph $S \in \mathcal{A}_k$ where $k = |\varphi_\diamond|$, and $S \models \varphi$.*

Sketch of Proof: To prove this, we modify the proof of Theorem 5.14. The main observation is that we cannot force a path to visit a node labeled with a constant, except at the endpoints of a path. (a) when creating a pre-model, duplicate nodes with constants, (b) witness splitting results in a pre-model with at most $|\varphi_\diamond|$ extra edges, (c) use homomorphism which only preserves existence of constants, not their absence, and (d) merge operation enabled by $\mathcal{L}_1$ preserve homomorphism, because they do not require merging a node with a constant, because a pattern may not contain a positive occurrence of equality between a variable and a constant (unlike $\mathcal{L}_2$).

Constant symbols can be eliminated from routing expressions, but the complexity of this operation is prohibitive. The $\mathcal{L}_1$ formulas that come up in practice are well-structured, and we hope to achieve a reasonable performance.

8.1 $\mathcal{L}_1$ is NEXPTIME-hard

The proof in this section is an adapted version of the NEXPTIME-hardness proof from [29, Theorem 5]. [29, Theorem 5] uses universal quantification over nodes, which is not available in $\mathcal{L}_0$. Instead, the proof in this section use reachability constraints and patterns.

Let $\mathcal{T}$ be a tiling problem as in Def. 3.1, and let n be a natural number. It is an NEXPTIME-complete problem to test on input $(\mathcal{T}, 1^n)$ whether there is a $\mathcal{T}$-tiling of a square grid of size 2^n by 2^n [40].

Theorem 8.2 *The satisfiability of $\mathcal{L}_1$ formulas is NEXPTIME-hard.*

Proof: Let $\mathcal{T}$ be a tiling problem as in Def. 3.1, and let n be a natural number. We define a formula φ_n that exactly expresses a solution to the tiling problem. When φ_n is satisfiable, it has a minimal model of size $2^{\Omega(n)}$.

We use two constants: s, denoting the top left node of the grid, and t, denoting the bottom right node of the grid. The desired model will consist of 2^{2n} tiles:

$$\begin{array}{lll} s = [1,1,t_0] & \cdots & [1,2^n,t] \\ \quad\;\; [2,1,t'] & \cdots & [2,2^n,t''] \\ \qquad \vdots & & \quad \vdots \\ \quad\;\; [2^n,1,t'''] & \cdots & [2^n,2^n,t_k] = t \end{array}$$

The binary relation n holds between each pair of consecutive tiles, including, for example, $[1,2^n,t]$ and $[2,1,t']$. We include the following unary relation symbols: $H_1,\ldots H_n$, indicating the horizontal position as an n-bit number; $V_1,\ldots V_n$, indicating the vertical position; and $T_0,\ldots T_k$, indicating the tile type.

The formula φ_n is the conjunction of the following assertions.

There is a path from s to t:

$$s\langle \xrightarrow{n}^{*}\rangle t \tag{6}$$

All E edges reachable from s are deterministic and unshared:

$$s[\xrightarrow{n}^{*}]det_n \wedge uns_n \tag{7}$$

The node labeled with s is the first tile, has tile type t_0, and the node labeled with t is the last tile and has tile type t_k:

$$T_0(s) \wedge \bigwedge_{i=1}^{n} (\neg H_i(s) \wedge \neg V_i(s)) \wedge T_k(t) \wedge \bigwedge_{i=1}^{n} (H_i(t) \wedge V_i(t)) \tag{8}$$

We have chosen for simplicity to encode the tile types in unary so we need to say that tile types are mutually exclusive and every node has a tile:

$$s[\xrightarrow{n}^{*}]\left(\bigwedge_{0\le i<j\le k} \neg(T_i \wedge T_j)\right) \wedge \left(\bigvee_{0\le i\le k} T_i\right) \tag{9}$$

The arrangement of tiles honors $\mathcal{T}$'s horizontal and vertical adjacency requirements:

$$\textbf{let}\ p(v) \overset{\text{def}}{=} \text{Next}_h(v,v') \Rightarrow \text{Hor}(v,v')\ \textbf{in}\ s[\xrightarrow{n}^{*}]p \tag{10}$$

$$\textbf{let}\ p(v) \overset{\text{def}}{=} \text{Next}_v(v,v') \Rightarrow \text{Vert}(v,v')\ \textbf{in}\ s[\xrightarrow{n}^{*}]p \tag{11}$$

The abbreviation Next_v, Next_h, Vert, Horz, and Next denote formulas which contain only unary relation symbols and variables, and no equality. We rely on the fact that a neighborhood of a pattern need not be connected.

The abbreviation Next$_h(x,y)$ means that x and y have the same vertical position and y's horizontal position is one more than that of x. Next$_v(x,y)$ means that x and y have the same horizontal position and y's vertical position is one more than that of x.

$$\begin{aligned}\text{Next}_h(x,y) &\equiv \left(\bigwedge_{i=1}^{n} V_i(x) \leftrightarrow V_i(y)\right) \wedge \text{PlusOne}_h(x,y) \\ \text{Next}_v(x,y) &\equiv \left(\bigwedge_{i=1}^{n} H_i(x) \leftrightarrow H_i(y)\right) \wedge \text{PlusOne}_v(x,y)\end{aligned}$$

The abbreviations PlusOne$_h(x,y)$ and PlusOne$_v(x,y)$ are nearly identical. Thus, we restrict our attention to PlusOne$_h(x,y)$, which means that the horizontal position of y is one greater than the horizontal position of x. (Our convention is that the bit positions are numbered 1 to n, with 1 being the high-order bit, and n the low-order bit.) PlusOne$_h(x,y)$ can be written as follows:

$$\begin{aligned}\text{PlusOne}_h(x,y) \quad \equiv \quad & \textstyle\bigvee_{i=1}^{n}[\bigwedge_{j>i}(H_j(x) \wedge \neg H_j(y)) \quad \wedge \quad (\neg H_i(x) \wedge H_i(y)) \\ & \qquad\qquad \wedge \quad \textstyle\bigwedge_{j<i}(H_j(x) \leftrightarrow H_j(y))]\end{aligned}$$

The length of the formula PlusOne$_h(x,y)$ is $O(n^2)$.

The abbreviation Hor(x,y) (resp. Vert(x,y)) is a disjunction over the tile types asserting that the tiles in positions x and y are horizontally (resp. vertically), compatible. For example,

$$\text{Hor}(x,y) \equiv \bigvee_{R(t_i,t_j)} (T_i(x) \wedge T_j(y)) \tag{12}$$

The abbreviation Next(x,y) means Next$_h(x,y)$ or x has horizontal position 2^n, y has horizontal position 1, and y's vertical position is one more than that of x:

$$\begin{aligned}\text{Next}(x,y) \equiv \ & \text{Next}_h(x,y) \bigvee \\ & \Big((\textstyle\bigwedge_{i=1}^{n} H_i(x)) \wedge (\bigwedge_{i=1}^{n-1} \neg H_i(y)) \wedge H_n(y) \wedge \text{PlusOne}_v(x,y)\Big)\end{aligned}$$

Finally, if there is an edge from x to y, then there $Next(x,y)$ holds:

$$\textbf{let } p(v) \overset{\text{def}}{=} \left(v \xrightarrow{n} v' \Rightarrow \text{Next}(v,v')\right) \textbf{ in } s[\xrightarrow{n}{}^{*}]p \tag{13}$$

Remark. The length of the formula φ_n described above is $O(n^2)$. The only difficulty in keeping φ_n to total size $O(n)$ is in writing the formulas PlusOne$_h(x,y)$ and

PlusOne$_v(x, y)$. We can decrease the size by keeping track of the position i using $2n$ addition unary relation symbols, similarly to the proof of [29, Lemma 14].

9 Limitations and Further Extensions

Despite the fact that $\mathcal{L}_2$ is useful, there are interesting program properties that cannot be expressed directly. For example, transitivity of a binary relation, that can be used, e.g., to express partial orders, is naturally expressible in $\mathcal{L}_0$, but not in $\mathcal{L}_2$. There are of course interesting properties that are beyond $\mathcal{L}_0$, such as the property that a general graph is a tree in which every leaf has a pointer to the root of a tree.

In the future, we plan to generalize $\mathcal{L}_2$ while maintaining decidability, perhaps beyond $\mathcal{L}_0$ (i.e., to capture properties that are not expressible in $\mathcal{L}_0$). We are encouraged by the fact that the proof of decidability in Section 5 holds "as is" for many useful extensions. For example, more complex patterns can be used, as long as they do not violate the $\mathcal{A}_k$-model property.

9.1 The Logic $\mathcal{L}_3$

In the $\mathcal{L}_0$ logic, reachability constraints describe paths that start from nodes labeled by some constant. The requirement that a path start with a constant is not necessary for decidability. We define $\mathcal{L}_3$ that generalizes $\mathcal{L}_0$ with paths that start from any node that satisfies a quantifier-free *positive* formula θ:

$$\theta[R]p \overset{\text{def}}{=} \forall w_0, \ldots, w_m, v_0, \ldots, v_n . R(w_0, v_0) \land \theta(w_0, \ldots, w_m) \Rightarrow p(v_0, \ldots, v_n)$$

A simple and very useful fragment of $\mathcal{L}_3$ is $\mathcal{L}_4$ in which θ is fixed to be $true$. We use $[R]p$ to denote $true[R]p$. For example, we can specify that all f-edges in the graph are deterministic, and not only those reachable from some constant: $[\epsilon]det_f$.

The fragment $\mathcal{L}_3$ provides several ways to express the same property; this flexibility can be useful when writing specifications manually. For example, the formula $(x \lor y)[R]p$ in $\mathcal{L}_3$ is equivalent to $x[R]p \lor y[R]p$ in $\mathcal{L}_1$, and to $[x + y.R]p$ in $\mathcal{L}_4$. The formula $(x \land y)[R]p$ in $\mathcal{L}_3$ is equivalent to $(x = y) \Rightarrow x[R]p$ in $\mathcal{L}_1$ and to $[x.y.R]p$ in $\mathcal{L}_4$.

We can translate every $\mathcal{L}_0$ formula to $\mathcal{L}_4$ using constants in routing expressions: $x[R]p \in \mathcal{L}_0$ is translated into $[x.R]p$. We can show that $\mathcal{L}_3$ has a finite model property. The logic *LRP* that results from $\mathcal{L}_3$ by restricting it to $\mathcal{L}_2$ patterns is decidable.

For example, recall the `mark` procedure from Section 4. We can modify it to scan the heap from a set of roots, instead of a single root. To write specifications for the modified version of `mark`, we can model the set of root objects using a unary relation $root$, instead of the constant symbol with the same name, which is used in Section 4. The rest of the specification remains unchanged. The resulting formulas are in *LRP*.

9.2 The Logic $U\mathcal{L}_1$

We can extend $\mathcal{L}_1$ with (a possibly restricted use of) quantifiers, going beyond the proposition logic $\mathcal{L}_0$. This extension provides a more general way to write specifications. In fact, the auxiliary constants used in the specification of `append` and `mark` procedures in Section 4, can be thought of as universally quantified variables.

We extend $\mathcal{L}_1$ with universal quantification over constants, as follows. For a vocabulary τ, a formula in $U\mathcal{L}_1$ over τ is a positive boolean combination of formulas of the form $\forall c_1, \ldots, c_n.\varphi'$, where φ' is in $\mathcal{L}_1$ over the vocabulary $\tau' = \tau \cup \{c_1, \ldots, c_n\}$). The semantics of the universal quantifiers is defined as usual. The problem of validity of $U\mathcal{L}_1$-formulas is decidable by reduction to validity in $\mathcal{L}_1$.

Lemma 9.1 *Let $\varphi \in U\mathcal{L}_1$ be of the form $\forall c_1, \ldots, c_n.\varphi'$. The formula φ is valid if and only if φ' is valid.*

Note that $U\mathcal{L}_1$ is *not* closed under negation (whereas $\mathcal{L}_1$ is closed under negation).

It is possible to add quantification over sets and relations, while preserving decidability, as long as there are no quantifier alternations. Quantification of binary relations can be useful for writing modular specifications, and analysis that does not violate abstraction layers. For example, if a procedure's formal parameter x is a pointer to an abstract data-type, we can specify that the field of objects that implement the abstract data-type are not modified by the procedure, without exposing the implementation: $\forall\Sigma.\forall f, f'.x[\xrightarrow{\Sigma}^*]same_{f,f'}$.

10 Related Work

There are several works on logic-based frameworks for reasoning about graph/heap structures. We mention here the ones which are, as far as we know, the closest to ours.

The logic $\mathcal{L}_0$ can be seen as a fragment of the first-order logic over graph structures with transitive closure (TC logic [28]). It is well known that TC is undecidable, and that this fact holds even when transitive closure is added to simple fragments of FO such as the decidable fragment L^2 of formulas with two variables [38,23,21].

It can be seen that our logics $\mathcal{L}_0$ and $\mathcal{L}_1$ are both uncomparable with L^2 + TC. Indeed, in $\mathcal{L}_0$ no alternation between universal and existential quantification is allowed. On the other hand, $\mathcal{L}_1$ allows us to express patterns (e.g., heap sharing) that require more than two variables (see Table 1, Section 4).

In [4], decidable logic L_r (which can also be seen as a fragment of TC) is introduced. The logics $\mathcal{L}_0$ and $\mathcal{L}_1$ generalize L_r, which is in fact the fragment of these logics where only two fixed patterns are allowed: equality to a program variable and heap sharing.

In [29,3,34,5] other decidable logics are defined, but their expressive power is rather limited w.r.t. $\mathcal{L}_1$ since they allow at most one binary relation symbol (modelling linked data-structures with 1-selector). For instance, the logic of [29] does not allow us to express the reversal of a list. Concerning the class of 1-selector linked data-structures, [9] provides a decision procedure for a logic with reachability constraints and arithmetical constraints on lengths of segments in the structure. It is not clear how the proposed techniques can be generalized to larger classes of graphs. Other decidable logics [10,33] are restricted in the sharing patterns and the reachability they can describe.

Other works in the literature consider extensions of the first-order logic with fixpoint operators. Such an extension is again undecidable in general but the introduction of the notion of (loosely) guarded quantification allows one to obtain decidable fragments such as μGF (or μLGF) (Guarded Fragment with least and greater fixpoint operators) [22,20]. Similarly to our logics, the logic μGF (and also μLGF) has the tree model property: every satisfiable formula has a model of bounded tree width. However, guarded fixpoint logics are incomparable with $\mathcal{L}_0$ and $\mathcal{L}_1$. For instance, the $\mathcal{L}_1$ pattern det_f that requires determinism of f-field, is not a (loosely) guarded formula.

The PALE system [37] uses an extension of the weak monadic second order logic on trees as a specification language. The considered linked data structures are those that can be defined as *graph types* [32]. Basically, they are graphs that can be defined as trees augmented by a set of edges defined using routing expressions (regular expressions) defining paths in the (undirected structure of the) tree. $\mathcal{L}_1$ allows us to reason naturally about arbitrary graphs without limitation to tree-like structures. By restricting the syntax, we guarantee that satisfiability queries posed over arbitrary graphs can be answered precisely by considering only tree-like graphs. This approach allows us to automate the reasoning about limited but interesting properties of *arbitrary* graphs.

Moreover, as we show in Section 4, our logical framework allows us to express postconditions and loop invariants that relate the input and the output state. For instance, even in the case of singly-linked lists, our framework allows us to express properties that cannot be expressed in the PALE framework: in the list reversal example of Section 4, we show that the output list is precisely the reversed input list, by expressing the relationships between fields before and after the procedure, whereas in the PALE approach, a postcondition can only express that the output is a list that is a permutation of the input list. In particular, a postcondition that relates fields before and after the procedure involves two binary relations with arbitrary interpretation. This can be easily done in $\mathcal{L}_0$ which supports an arbitrary number of binary relations. This is not supported by PALE, which allows two binary relations with a specific interpretation as tree edges. In the PALE approach, a postcondition can only express that the output is a list that is a permutation of the input list.

In [30], we tried to employ a decision procedure for MSO on trees to reason about reachability. However, this places a heavy burden on the specifier to prove that the data-structures in the program can be simulated using trees. The current paper eliminated this burden by defining syntactic restrictions on the formulas and showing a general reduction theorem.

Other approaches in the literature use undecidable formalisms such as [25], which provides a natural and expressive language, but does not allow for automatic property checking.

Separation logic has been introduced recently as a formalism for reasoning about heap structures [43]. The general logic is undecidable [13] but there are few works showing decidable fragments [13,5]. One of the fragments is propositional separation logic where quantification is forbidden [13,12] and therefore seems to be incomparable with our logic. The fragment defined in [5] allows one to reason only about singly-linked lists with explicit sharing. In fact, the fragment considered in [5] can be translated to $\mathcal{L}_1$, and therefore, entailment problems as stated in [5] can be reduced to validity of implications in $\mathcal{L}_1$.

The logic $\mathcal{L}_0$ integrates features of such prominent formalisms as the modal logics, the classical first-order logic, and the regular expressions. The hybrid logics [1] also combine features of modal and classical logics. The most relevant is the hybrid μ-calculus [47] which extends the μ-calculus with the following features: (i) nominals, that correspond to constants in $\mathcal{L}_1$, (ii) universal program, that corresponds to the fragment $\mathcal{L}_4$, and (iii) the ability to reasoning about the past, that corresponds to the use of backward edges in routing expressions. The hybrid μ-calculus is incomparable in its expressive power to $\mathcal{L}_1$: on one hand, it supports a more general reachability via the least and greatest fixpoint operators; on the other hand, the equality is restricted to nominals. For example, it cannot express that a graph is a tree. Unlike $\mathcal{L}_0$, the hybrid μ-calculus does not have a finite model property. Every satisfiable formula in hybrid μ-calculus has a tree-like model. The complexity of hybrid μ-calculus is EXPTIME-complete, but currently, there is no decision procedure available. Reportedly, a tableaux-based decision procedure for the alternation-free fragment of hybrid μ-calculus is being developed.

$\mathcal{L}_0$ shares some common features with description logics [17], which is traditionally used for knowledge representation, databases, semantic web, with the notable exception of [19], which shows the description logics can be used for reasoning about data-structures. The basic notions of Description Logics are concepts, that correspond to unary relations in $\mathcal{L}_0$, and roles, that correspond to binary relations in $\mathcal{L}_1$. In addition, expressive Description Logics support (iii) nominals, that correspond to constants in $\mathcal{L}_0$; quantified role restrictions, that can encode determinism; and inverse roles, that correspond to backward edges in routing expressions. The combination of quantified role restrictions and inverse roles provides a way to express sharing. The need for transitivity and fixpoints arises in many contexts [14], including, service description logics [6]. It has been shown that a description logic which combines with nominals, inverse roles, determinism, and least fixpoints is undecidable [7]. In light of the negative results, it is interesting to investigate the usefulness of $\mathcal{L}_1$ for specifying web services. There are a variety of efficient reasoning tools for description logics, both tableaux-based and resolution-based, which provide some support for expressive features, such as nominals and inverse roles, e.g., FaCT, Racer. To the best of our knowledge, none of the existing tools supports transitive closure of roles or fixpoints.

11 Conclusions

Defining decidable fragments of first order logic with transitive closure that are useful for program verification is a difficult task (e.g., [29]). In this paper, we demonstrated that this is possible by combining three principles: (i) allowing arbitrary boolean combinations of the reachability constraints, which are closed formulas without quantifier alternations, (ii) defining reachability using regular expressions denoting pointer access paths (not) reaching a certain pattern, and (iii) syntactically limiting the way patterns are formed. Extensions of the patterns that allow larger distances between nodes in the pattern either break our proof of decidability or are directly undecidable.

The decidability result presented in this paper improves the state-of-the-art significantly. In contrast to [29,3,34,5], *LRP* allows several binary relations. This provides a natural way to (i) specify invariants for data-structures with multiple fields (e.g., trees, doubly-linked lists), (ii) specify post-condition for procedures that mutate pointer fields of data-structures, by expressing the relationships between fields before and after the procedure (e.g., list reversal, which is beyond the scope of PALE), (iii) express verification conditions using a copy of the vocabulary for each program location. Operating on general graphs allows us to verify that the data-structure invariant is reestablished after a sequence of low-level mutations that temporarily violate the invariant data-structure.

We are encouraged by the expressiveness of this simple logic and plan to explore its usage for program verification and abstract interpretation.

References

[1] C. Areces, P. Blackburn, and M. Marx. Hybrid logics: characterization, interpolation and complexity. *The Journal of Symbolic Logic*, 66(3):977–1010, 2001.

[2] S. Arnborg, J. Lagergren, and D. Seese. Easy problems for tree-decomposable graphs. *J. Algorithms*, 12(2):308–340, 1991.

[3] I. Balaban, A. Pnueli, and L. D. Zuck. Shape analysis by predicate abstraction. In *VMCAI*, pages 164–180, 2005.

[4] M. Benedikt, T. Reps, and M. Sagiv. A decidable logic for describing linked data structures. In *European Symp. On Programming*, pages 2–19, March 1999.

[5] J. Berdine, C. Calcagno, and P. O'Hearn. A Decidable Fragment of Separation Logic. In *FSTTCS'04*. LNCS 3328, 2004.

[6] P. A. Bonatti. Towards service description logics. In *JELIA*, pages 74–85, London, UK, 2002. Springer-Verlag.

[7] P.A. Bonatti and A. Peron. On the undecidability of logics with converse, nominals, recursion and counting. *Artificial Intelligence*, 158(1):75–96, 2004.

[8] A. Bouajjani, P. Habermehl, P.Moro, and T. Vojnar. Verifying Programs with Dynamic 1-Selector-Linked Structures in Regular Model Checking. In *Proc. of TACAS '05*, volume 3440 of *LNCS*. Springer, 2005.

[9] M. Bozga and R. Iosif. Quantitative Verification of Programs with Lists. In *VISSAS intern. workshop*. IOS Press, 2005.

[10] M. Bozga, R. Iosif, and Y. Lakhnech. On logics of aliasing. In *Static Analysis Symp.*, pages 344–360, 2004.

[11] L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of jml tools and applications. *Int. J. on Software Tools for Technology Transfer*, 7(3):212–232, 2005.

[12] C. Calcagno, P. Gardner, and M. Hague. From Separation Logic to First-Order Logic. In *FOSSACS'05*. LNCS 3441, 2005.

[13] C. Calcagno, H. Yang, and P. O'Hearn. Computability and Complexity Results for a Spatial Assertion Language for Data Structures. In *FSTTCS'01*. LNCS 2245, 2001.

[14] D. Calvanese, G. De Giacomo, and M. Lenzerini. Reasoning in expressive description logics with fixpoints based on automata on infinite trees. In *IJCAI*, pages 84–89, 1999.

[15] B. Courcelle. The monadic second-order logic of graphs, ii: Infinite graphs of bounded width. *Mathematical Systems Theory*, 21(4):187–221, 1989.

[16] Reinhard Diestel. *Graph Theory*. Springer-Verlag, 2000. Electronic Edition.

[17] F. Baader et al., editor. *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press, 2003.

[18] T. Genet and V. Tong. Reachability analysis of term rewriting systems with timbuk. In *LPAR*, pages 695–706, 2001.

[19] L. Georgieva and P. Maier. Description logics for shape analysis. In *SEFM*, pages 321–331, 2005.

[20] E. Grädel. Guarded fixed point logic and the monadic theory of trees. *Theoretical Computer Science*, 288:129–152, 2002.

[21] E. Grädel, M.Otto, and E.Rosen. Undecidability results on two-variable logics. *Archive of Math. Logic*, 38:313–354, 1999.

[22] E. Grädel and I. Walukiewicz. Guarded Fixed Point Logic. In *LICS'99*. IEEE, 1999.

[23] E. Graedel, P. Kolaitis, and M. Vardi. On the decision problem for two variable logic. *Bulletin of Symbolic Logic*, 1997.

[24] L. Hendren. *Parallelizing Programs with Recursive Data Structures*. PhD thesis, Cornell Univ., Ithaca, NY, Jan 1990.

[25] L. Hendren, J. Hummel, and A. Nicolau. Abstractions for recursive pointer data structures: Improving the analysis and the transformation of imperative programs. In *SIGPLAN Conf. on Prog. Lang. Design and Impl.*, pages 249–260, New York, NY, June 1992. ACM Press.

[26] J.G. Henriksen, J. Jensen, M. Jørgensen, N. Klarlund, B. Paige, T. Rauhe, and A. Sandholm. Mona: Monadic second-order logic in practice. In *TACAS*, 1995.

[27] C.A.R. Hoare. Recursive data structures. *Int. J. of Comp. and Inf. Sci.*, 4(2):105–132, 1975.

[28] N. Immerman. Languages that capture complexity classes. *SIAM Journal of Computing*, 16:760–778, 1987.

[29] N. Immerman, A. Rabinovich, T. Reps, M. Sagiv, and G. Yorsh. The boundery between decidability and undecidability of transitive closure logics. In *CSL*, 2004.

[30] N. Immerman, A. Rabinovich, T. Reps, M. Sagiv, and G. Yorsh. Verification via structure simulation. In *CAV*, 2004.

[31] S. S. Ishtiaq and P. W. O'Hearn. Bi as an assertion language for mutable data structures. In *POPL*, pages 14–26, 2001.

[32] N. Klarlund and M. I. Schwartzbach. Graph Types. In *POPL'93*. ACM, 1993.

[33] V. Kuncak and M. Rinard. Generalized records and spatial conjunction in role logic. In *Static Analysis Symp.*, Verona, Italy, August 26–28 2004.

[34] S. K. Lahiri and S. Qadeer. Verifying properties of well-founded linked lists. In *Symp. on Princ. of Prog. Lang.*, 2006.

[35] T. Lev-Ami and M. Sagiv. TVLA: A system for implementing static analyses. In *Static Analysis Symp.*, pages 280–301, 2000.

[36] Albert R. Meyer. Weak monadic second-order theory of successor is not elementary recursive. In *Logic Colloquium (Proc. Symposium on Logic, Boston, 1972)*, volume 453, pages 132–154, 1975.

[37] A. Møller and M.I. Schwartzbach. The pointer assertion logic engine. In *SIGPLAN Conf. on Prog. Lang. Design and Impl.*, pages 221–231, 2001.

[38] M. Mortimer. On languages with two variables. *Zeitschrift für Mathematische Logik und Grundlagen der Mathematik*, 21:135–140, 1975.

[39] F. Nielson, H. Riis Nielson, and H. Seidl. Normalizable horn clauses, strongly recognizable relations, and spi. In *SAS*, pages 20–35, 2002.

[40] C. M. Papadimitriou. Addison-Wesley, 1994.

[41] M. Rabin. Decidability of second-order theories and automata on infinite trees. *Trans. Amer. Math. Soc.*, 141:1–35, 1969.

[42] T. Reps, M. Sagiv, and R. Wilhelm. Static program analysis via 3-valued logic. In *CAV*, pages 15–30, 2004.

[43] J. C. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *LICS'02*. IEEE, 2002.

[44] N. Robertson and P. D. Seymour. Graph minors. ii. algorithmic aspects of tree-width. *J. Algorithms*, 7(3):309–322, 1986.

[45] M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM Transactions on Programming Languages and Systems*, 20(1):1–50, January 1998.

[46] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems*, 2002.

[47] U. Sattler and M. Y. Vardi. The hybrid -calculus. In *IJCAR*, pages 76–91, 2001.

[48] D. Seese. Interpretability and tree automata: A simple way to solve algorithmic problems on graphs closely related to trees. In *Tree Automata and Languages*, pages 83–114. 1992.

[49] G. Yorsh, M. Sagiv, A. Rabinovich, A. Bouajjani, and A. Meyer. Verification framework based on the logic of reachable patterns. In preparation, 2005.