A Logic of Reachable Patterns in Linked Data-Structures

Greta Yorsh, Alexander Rabinovich, Mooly Sagiv, Antoine Meyer, Ahmed Bouajjani

To cite this version:

Greta Yorsh, Alexander Rabinovich, Mooly Sagiv, Antoine Meyer, Ahmed Bouajjani. A Logic of Reachable Patterns in Linked Data-Structures. FOSSACS 2006, Mar 2006, Vienne, Austria. p. 94-110, 10.1007/11690634_7. hal-00149120

HAL Id: hal-00149120
https://hal.archives-ouvertes.fr/hal-00149120
Submitted on 24 May 2007

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
A Logic of Reachable Patterns
in Linked Data-Structures

Greta Yorsh\textsuperscript{1*}, Alexander Rabinovich\textsuperscript{1}, Mooly Sagiv\textsuperscript{1},
Antoine Meyer\textsuperscript{2}, and Ahmed Bouajjani\textsuperscript{2}

\textsuperscript{1} Tel Aviv Univ., Israel. \{gretay,rabinoa,msagiv\}@post.tau.ac.il
\textsuperscript{2} Liafa, Univ. of Paris 7, France. \{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 precondition, 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 entry to a procedure to the fields on exit. Therefore, it is possible to use the logic to automatically prove partial correctness of programs performing low-level heap mutations.

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 [19].

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 [27, 34, 31], logic-based reasoning [23, 32], and automata-based techniques [24, 28, 5]. 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

\* This research was supported by THE ISRAEL SCIENCE FOUNDATION (grant No 304/03).
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) the acyclicity of data-structure fragments, i.e., every element reachable from node \( u \) cannot reach \( u \), (2) 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, (3) 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 [13, 21].

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

Alternatively, our logic can be seen as a propositional logic with atomic proposition 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, in a doubly linked list, a pattern says that if an object \( v \) has an emanating forward pointer that leads to an object \( w \), then \( w \) has a backward pointer into \( v \).

The contributions of this paper can be summarized as follows:

- We define the Logic of Reachable Patterns (LRP) 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 LRP 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 LRP is, unfortunately, undecidable.
- We define a suitable restriction on the patterns leading to a fragment of LRP called LRP₂.
- We prove that the satisfiability (and validity) problem is decidable. The fragment LRP₂ is the main technical result of the paper and the decidability proof is non-trivial. The main idea is to show that every satisfiable LRP₂ formula is also satisfied by a tree-like graph. Thus, even though LRP₂ 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 LRP₂ using a decision procedure for monadic second-order logic (MSO) on trees.
We show that despite the restriction on patterns we introduce, the logic LRP$_2$ is still expressive enough for use in program verification: various important data-structures, and loop invariants concerning their manipulation, are in fact definable in LRP$_2$.

The new logic LRP$_2$ forms a basis of the verification framework for programs with pointer manipulation [37], which has important advantages w.r.t. existing ones. For instance, in contrast to decidable logics that restrict the graphs of interest (such as 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 postcondition and loop invariants that relate the input and the output state. Moreover, our logic strictly generalizes the decidable logic in [3], which inspired our work. Therefore, it can be shown that certain heap abstractions including [16, 33] can be expressed using LRP$_2$ formulas.

The rest of the paper is organized as follows: Section 2 defines the syntax and the semantics of LRP, and shows that it has a finite model property, and that LRP is undecidable; Section 3 defines the fragment LRP$_2$, and demonstrates the expressiveness of LRP$_2$ on several examples; Section 4 describes the main ideas of the decidability proof for LRP$_2$; Section 5 discusses the limitations and the extensions of the new logics; finally, Section 6 discusses the related work. The full version of the paper [36] contains the formal definition of the semantics of LRP and proofs.

## 2 The LRP 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 LRP

LRP is a propositional logic over reachability constraints. That is, an LRP 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 (i) $C$ is a finite set of constant symbols usually denoting designated objects in the heap, pointed to by program variables; (ii) $U$ is a set of unary relation symbols denoting properties, e.g., color of a node in a Red-Black tree; (ii) $F$ is a finite set of binary relation symbols (edges) usually denoting pointer fields.$^3$

A term $t$ is either a variable or a constant $c \in C$. An atomic formula is an equality $t = t'$, a unary relation $u(t)$, or an edge formula $t \xrightarrow{f} t'$, where $f \in F$, and $t, t'$ are terms. A quantifier-free formula $\psi(v_0, \ldots, v_n)$ over $\tau$ and variables $v_0, \ldots, v_n$ is an arbitrary boolean combination of atomic formulas. Let $\text{FV}(\psi)$ denote the free variables of the formula $\psi$.

$^3$ We can also allow auxiliary constants and fields including abstract fields [8].
Definition 1. Let $\psi$ be a conjunction of edge formulas of the form $v_i \xrightarrow{f} v_j$, where $f \in F$ and $0 \leq i, j \leq n$. The Gaifman graph of $\psi$, denoted by $B_\psi$, is an undirected graph with a vertex for each free variable of $\psi$. There is an arc between the vertices corresponding to $v_i$ and $v_j$ in $B_\psi$ if and only if $(v_i \xrightarrow{f} v_j)$ appears in $\psi$, for some $f \in F$. The distance between logical variables $v_i$ and $v_j$ in the formula $\psi$ is the minimal edge distance between the corresponding vertices $v_i$ and $v_j$ in $B_\psi$.

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

Definition 2. (Syntax of LRP) A neighborhood formula $N(v_0, \ldots, v_n)$ is a conjunction of edge formulas of the form $v_i \xrightarrow{f} v_j$, where $f \in F$ and $0 \leq i, j \leq n$.

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

$$R ::= \emptyset \quad \text{empty set}$$

$$\varepsilon \quad \text{empty path}$$

$$f \rightarrow \quad f \in F \quad \text{forward along edge}$$

$$f \leftarrow \quad f \in F \quad \text{backward along edge}$$

$$u \quad u \in U \quad \text{test if } u \text{ holds}$$

$$\neg u \quad u \in U \quad \text{test if } u \text{ does not hold}$$

$$c \quad c \in C \quad \text{test if } c \text{ holds}$$

$$\neg c \quad c \in C \quad \text{test if } c \text{ does not hold}$$

$$R_1.R_2 \quad \text{concatenation}$$

$$R_1 | R_2 \quad \text{union}$$

$$R^* \quad \text{Kleene star}$$

A routing expression can require that a path traverse some edges backwards. A routing expression has the ability to test presence and absence of certain unary relations and constants along the path.

A reachability constraint is a closed formula of the form:

$$\forall v_0, \ldots, v_n. R(c, v_0) \Rightarrow (N(v_0, \ldots, v_n) \Rightarrow \psi(v_0, \ldots, v_n))$$

where $c \in C$ is a constant, $R$ is a routing expression, $N$ is a neighborhood formula, and $\psi$ 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 $\psi$ is a formula with a single free variable $v_0$.

An LRP 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 a “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 $\psi$ defines more constraints on those nodes. $^4$

$^4$ In all our examples, a neighborhood formula $N$ used in a pattern is such that $B_N$ (the Gaifman graph of $N$) is connected.
We use let expressions to specify the scope in which the pattern is declared:

\[
\text{let } p_1(v_0) \equiv N_1(v_0, v_1, \ldots, v_n) \Rightarrow \psi_1(v_0, \ldots, v_n) \text{ in } \varphi
\]

This allows us to write more concise formulas via sharing of patterns.

**Shorthands** We use \(c[R]p\) to denote a reachability constraint. 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 \(p(v_0) \equiv (\text{true} \Rightarrow (v_0 = c_2))\) 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 labelled by constant \(c_2\) is not reachable along an \(R\)-path from the node labelled by \(c_1\). We use \(c_1[R]c_2\) as a shorthand for \((\neg (c_1[R]) \neg c_2)\). Intuitively, \(c_1[R]c_2\) means that there exists an \(R\)-path from \(c_1\) to \(c_2\). We use \(c_1 \neq c_2\) to denote \(c_1(c) \land c_2\) to denote \(\neg (c_1 = c_2)\). We use \(c[R](p_1 \land p_2)\) to denote \((c[R]p_1) \land (c[R]p_2)\), 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 \(p(v_0) \equiv (N_1 \Rightarrow \psi_1) \land (N_2 \Rightarrow \psi_2)\) in \(\varphi\).

In routing expressions, we use \(\Sigma\) to denote \((f_1 | f_2 | \ldots | f_m)\), the union of all the fields in \(F\). For example, \(c_1[\Sigma^*]c_2\) means that \(c_2\) is not reachable from \(c_1\) by any path. Finally, we sometimes omit the concatenation operator “.” in routing expressions.

**Semantics** An interpretation for an LRP formula over \(\tau = (C, U, F)\) is a labelled directed graph \(G = (V^G, E^G, C^G, U^G)\) where: (i) \(V^G\) is a set of nodes modelling the heap objects, (ii) \(E^G : F \rightarrow \mathcal{P}(V^G \times V^G)\) are labelled edges, (iii) \(C^G : C \rightarrow V^G\) provides interpretation of constants as unique labels on the nodes of the graph, and (iv) \(U^G : U \rightarrow \mathcal{P}(V^G)\) maps unary relation symbols to the set of nodes in which they hold.

We say that node \(v \in G\) is labelled with \(\sigma\) if \(\sigma \in C\) and \(v = C^G(\sigma)\) or \(\sigma \in U\) and \(v \in U^G(\sigma)\). In the rest of the paper, graph denotes a directed labelled graph, in which nodes are labelled by constant and unary relation symbols, and edges are labelled by binary relation symbols, as defined above.

We define a satisfaction relation \(\models\) between a graph \(G\) and LRP formula \((G \models \varphi)\) similarly to the usual semantics the first-order logic with transitive closure over graphs (see [36]).

### 2.2 Properties of LRP

LRP with arbitrary patterns has a finite model property. If formula \(\varphi \in \text{LRP}\) has an infinite model, each reachability constraint in \(\varphi\) that is satisfied by this model has a finite witness.

**Theorem 1. (Finite Model Property)** Every satisfiable LRP formula is satisfiable by a finite graph.

**Sketch of Proof:** We show that LRP 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
The satisfiability problem of LRP formulas is undecidable. The proof uses a reduction from the halting problem of a Turing machine.

**Theorem 2. (Undecidability)** The satisfiability problem of LRP formulas is undecidable.

**Sketch of Proof:** Given a Turing machine $M$, we construct a formula $ϕ_M$ such that $ϕ_M$ is satisfiable if and only if the execution of $M$ eventually halts.

The idea is that each node in the graph that satisfies $ϕ_M$ describes a cell of a tape in some configuration, with unary relation symbols encoding the symbol in each cell, the location of the head and the current state. The $n$-edges describe the sequence of cells in a configuration and a sequence of configurations. The $b$-edges describe how the cell is changed from one configuration to the next. The constant $c_1$ marks the node that describes the first cell of the tape in the first configuration, the constant $c_2$ marks the node that describes the first cell in the second configuration, and the constant $c_3$ marks the node that describes the last cell in the last configuration (see sketch in Fig. 1).

![Fig. 1. sketch of a model.](image)

The most interesting part of the formula $ϕ_M$ ensures that all graphs that satisfy $ϕ_M$ have a grid-like form. It states that for every node $v$ that is $n$-reachable from $c_1$, 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 $u$:

$$
\text{let } p(v) \equiv (v \xrightarrow{b} u) \land (v \xrightarrow{n} v_1) \land (u \xrightarrow{n} u_1) \Rightarrow (v_1 \xrightarrow{b} u_1) \text{ in } c_1[(\leq n)^*]p \quad (1)
$$

**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.

## 3 The $LRP_2$ Fragment and its Usefulness

In this section we define the $LRP_2$ fragment of $LRP$, by syntactically restricting the patterns. The main idea is to limit the distance between the nodes in the pattern in certain situations.
Definition 3. A formula is in \( LRP_2 \) if in every reachability constraint \( e[R]p \), with a pattern \( p(v_0) = N(v_0, \ldots, v_n) \Rightarrow \psi(v_0, \ldots, v_n) \), \( \psi \) has one of the following forms:

- **(equality pattern)** \( \psi \) is an equality between variables \( v_i = v_j \), where \( 0 \leq i, j \leq n \), and the distance between \( v_i \) and \( v_j \) in \( N \) is at most 2 (distance is defined in Def. 1).

- **(edge pattern)** \( \psi \) is of the form \( v_i \rightarrow v_j \) where \( f \in F \) and \( 0 \leq i, j \leq n \), and the distance between \( v_i \) and \( v_j \) in \( N \) is at most 1.

- **(negative pattern)** atomic formulas appear only negatively in \( \psi \).

Remark. Note that formula (1), which is used in the proof of undecidability in Theorem 2, is not in \( LRP_2 \), because \( p \) is an edge pattern with distance 3 between \( v_1 \) and \( u_1 \), while \( LRP_2 \) allows edge patterns with distance at most 1.

3.1 Describing Linked Data-Structures

In this section, we show that \( LRP_2 \) 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 \( f \).

<table>
<thead>
<tr>
<th>Pattern Name</th>
<th>Pattern Definition</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>( det_f(v_0) )</td>
<td>( (v_0 \xrightarrow{f} v_1) \land (v_0 \xrightarrow{f} v_2) \Rightarrow (v_1 = v_2) )</td>
<td>( f )-edge from ( v_0 ) is deterministic</td>
</tr>
<tr>
<td>( uns_f(v_0) )</td>
<td>( (v_1 \xrightarrow{f} v_0) \land (v_2 \xrightarrow{f} v_0) \Rightarrow (v_1 = v_2) )</td>
<td>( v_0 ) is not heap-shared by ( f )-edges</td>
</tr>
<tr>
<td>( uns_{f,g}(v_0) )</td>
<td>( (v_1 \xrightarrow{f} v_0) \land (v_2 \xrightarrow{g} v_0) \Rightarrow false )</td>
<td>( v_0 ) is not heap-shared by ( f )-edge and ( g )-edge</td>
</tr>
<tr>
<td>( inv_{f,b}(v_0) )</td>
<td>( (v_0 \xrightarrow{f} v_1) \Rightarrow v_1 \xrightarrow{b} v_0 ) \land ( (v_0 \xrightarrow{b} v_1) \Rightarrow v_1 \xrightarrow{f} v_0 )</td>
<td>edges ( f ) and ( b ) form a doubly-linked list between ( v_0 ) and ( v_1 )</td>
</tr>
<tr>
<td>( same_{f,g}(v_0) )</td>
<td>( (v_0 \xrightarrow{f} v_1) \Rightarrow v_0 \xrightarrow{g} v_1 ) \land ( (v_0 \xrightarrow{g} v_1) \Rightarrow v_0 \xrightarrow{f} v_1 )</td>
<td>edges ( f ) and ( g ) emanating from ( v_0 ) are parallel</td>
</tr>
</tbody>
</table>

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

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 \( LRP_2 \) this is expressed by the formula:

\[
(\bigwedge_{c \in C} \bigwedge_{f \in F} c[\Sigma^*]det_f) \land (\bigwedge_{f \in F} null(\xrightarrow{f} null))
\]

(2)
Using the patterns in Table 1, Table 2 defines some interesting properties of data-structures using LRP.2. The formula \( \text{reach}_{x,f,y} \) 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 \( \text{null} \) in the place of \( y \). For example, the formula \( \text{reach}_{x,f,\text{null}} \) describes a (possibly empty) linked-list pointed-to by \( x \). Note that it implies that the list is acyclic, because \( \text{null} \) 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 \( \text{unshared}_{x,f} \). Alternatively, we can specify that \( x \) is located on a cycle of \( f \)-edges: \( \text{cyclic}_{x,f} \). Disjointness can be expressed by the formula \( \text{disjoint}_{x,f,y,g} \) that uses both forward and backward traversal of edges in the routing expression. 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 \( \text{disjoint}_{x,f,y} \). Disjointness of data-structures is important for parallelization (e.g., see [17]).

The last two examples in Table 2 specify data-structures with multiple fields. The formula \( \text{inverse}_{x,f,b,y} \) describes a doubly-linked 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 \( \text{inv}_{f,b} \) 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. The formula \( \text{tree}_{root,r,l} \) describes a binary tree. The first part requires that the nodes reachable from the root (by following any path of \( l \) and
r fields) be not 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.

### 3.2 Expressing Verification Conditions

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 requires two fields to express partial correctness. Moreover, it manipulates linked lists in which each list node can be pointed-to from the outside. In this section, we show that the verification conditions for the procedure reverse can be expressed in LRP. 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 LRP is decidable, as shown in the next section. In [37], we show how to automatically generate verification conditions in LRP for arbitrary procedures that are annotated with preconditions, postconditions, and loop invariants in LRP.

```plaintext
Node reverse(Node x) {
    L0: Node y = NULL;
    L1: while (x != NULL) {
        L2: Node t = x->n;
        L3: x->n = y;
        L4: y = x;
        L5: x = t;
        L6: }
    L7: return y;
}
```

Fig. 2. Reverse.

Notice that in this section we assume that all graphs denote valid stores, i.e., satisfy (2). 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 the $n$-field on entry to the procedure.

$$pre \overset{\text{at}}{=} x^0((n^0)^*)null^0$$

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.

$$post \overset{\text{at}}{=} y^7((n^7)^*)null^7 \wedge \text{inverse}_{x^0,n^0,n^7,y^7}$$
The loop invariant $\varphi$ 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 $x$.

$$\varphi \overset{df}{=} \text{same}_{x^1,n^0,n^1} \land \text{inverse}_{x^0,n^0,n^1,y^1} \land x^0 \langle n^0 \rangle y^1$$

Note that the postcondition uses two binary relations, $n^0$ and $n^1$, 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 $\varphi$ 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.

$$x^0 \xrightarrow{n^0} n^1 \xrightarrow{n^0} y^1 \xrightarrow{n^0} x^1 \xrightarrow{n^0} y^6 \xrightarrow{n^0} x^6 \xrightarrow{n^0} n^0$$

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

To show that the loop invariant $\varphi$ 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-dereference, and the loop invariant holds at the end of the loop body:

$$VC_{loop} \overset{df}{=} (x \neq \text{null}) \land \varphi \land (y^6 = x^1) \land x^1 \langle n^1 \rangle x^6 \land x^1 \langle n^6 \rangle y^1 \land \text{same}_{y^1,n^1,n^6} \land \text{same}_{x^1,n^1,n^6} \Rightarrow (x^1 \neq \text{null}) \land \varphi^6$$

Here, $\varphi^6$ denotes the loop-invariant formula $\varphi$ after executing the loop body (label $L6$), i.e., replacing all occurrences of $x^1$, $y^1$ and $n^1$ in $\varphi$ 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 procedure’s exit (i.e., the loop is not entered because the loop condition does not hold), the postcondition holds on exit: $VC \overset{\text{def}}{=} \text{pre} \land (x^1 = \text{null}) \Rightarrow \text{post}$.

4 Decidability of $LRP_2$

In this section, we show that $LRP_2$ is decidable for validity and satisfiability. Since $LRP_2$ is closed under negation, it is sufficient to show that it is decidable for satisfiability.

The satisfiability problem for $LRP_2$ is decidable. The proof proceeds as follows:

1. Every formula $\varphi \in LRP_2$ can be translated into an equi-satisfiable normal-form formula that is a disjunction of formulas in $CLR_2$ (Def. 4 and Theorem 3). It is sufficient to show that the satisfiability of $CLR_2$ is decidable.
2. Define a class of simple graphs $A_k$, for which the Gaifman graph is a tree with at most $k$ additional edges (Def. 5).
3. Show that if formula $\varphi \in CLR_2$ has a model, $\varphi$ has a model in $A_k$, where $k$ is linear in the size of the formula $\varphi$ (Theorem 4). This is the main part of the proof.
4. Translate formula $\varphi \in CLR_2$ into an equivalent MSO formula.
5. Show that the satisfiability of MSO logic over $A_k$ is decidable, by reduction to MSO on trees [30]. We could have also shown decidability using the fact that the tree width of all graphs in $A_k$ is bounded by $k$, and that MSO over graphs with bounded tree width is decidable [11, 1, 35].

Definition 4. (Normal-Form Formulas) A formula in $CLR_2$ is a conjunction of reachability constraints of the form $c_1 \langle R \rangle c_2$ and $c[R]p$, where $p$ is one of the patterns allowed in $LRP_2$ (Def. 3). A normal-form formula is a disjunction of $CLR_2$ formulas.

Theorem 3. There is a computable translation from $LRP_2$ to a disjunction of formulas in $CLR_2$ that preserves satisfiability.

Ayah Graphs We define a notion of a simple tree-like directed graph, called Ayah graph.

Let $G(S)$ 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). The distance between nodes $v_1$ and $v_2$ in $S$ is the number of edges on the shortest path between $v_1$ and $v_2$ in $G(S)$. An undirected graph $B$ is in $T^k$ if removing self loops and at most $k$ additional edges from $B$ results in an acyclic graph.

Definition 5. For $k \geq 0$, an Ayah graph of $k$ is a graph $S$ for which the Gaifman graph is in $T^k$: $A_k = \{ S \mid G(S) \in T^k \}$.

Let $\varphi \in CLR_2$ be of the form $\varphi_0 \land \varphi_2 \land \varphi_{=} \land \varphi_{-}$, where $\varphi_0$ is a conjunction of constraints of the form $c_1 \langle R \rangle c_2$, $\varphi_2$ is a conjunction of reachability constraints with negative patterns, $\varphi_{=}$ is a conjunction of reachability constraints with equality patterns, and $\varphi_{-}$ is a conjunction of reachability constraints with edge patterns.
Theorem 4. If \( \varphi \in \text{CLRP}_2 \) is satisfiable, then \( \varphi \) is satisfiable by a graph in \( A_k \), where 
\[ k = 2 \times n \times |C| \times m, \]
\( m \) is the number of constraints in \( \varphi \), \( |C| \) is the number of constants in the vocabulary, and for every regular expression that appears in \( \varphi \) there is an equivalent automaton with at most \( n \) states.

**Sketch of Proof:** Let \( S \) be a model of \( \varphi : S \models \varphi \). We construct a graph \( S' \) from \( S \) and show that \( S' \models \varphi \) and \( S' \in A_k \). The construction uses the following operations on graphs.

Witness Splitting A witness \( W \) for a formula \( c_1(R)c_2 \) in \( \text{CLRP}_2 \) in a graph \( S \) is a path in \( S \), labelled with a word \( w \in L(R) \), from the node labelled with \( c_1 \) to the node labelled with \( c_2 \). Note that the nodes and edges on a witness path for \( R \) need not be distinct. Using \( W \), we construct a graph \( W' \) that consists of a path, labelled with \( w \), that starts at the node labelled by \( c_1 \) and ends at the node labelled by \( c_2 \). Intuitively, we duplicate a node of \( W \) each time the witness path for \( R \) traverses it, unless the node is marked with a constant. As a result, all shared nodes in \( W' \) are labelled with constants. Also, every cycle contains a node labelled with a constant. By construction, we get that \( W' \models c_1(R)c_2 \). We say that \( W' \) is the result of splitting the witness \( W \).

Finally, we say that \( W \) is the shortest witness for \( c_1(R)c_2 \) if any other witness path for \( c_1(R)c_2 \) is at least as long as \( W \). The result of splitting the shortest witness is a graph in \( A_k \), where \( k = 2 \times n \times |C| \): to break all cycles it is sufficient to remove all the edges adjacent to nodes labelled with constants, and a node labelled with a constant is visited at most \( n \) times. (If a node is visited more than once in the same state of the automaton, the path can be shortened.)

Merge Operation Merging two nodes in a graph is defined in the usual way by gluing these nodes. Let \( p(v_0) \equiv N(v_0, v_1, v_2) \Rightarrow (v_1 = v_2) \) be an equality pattern. If a graph violates a reachability constraint \( c[R]p \), we can assign nodes \( n_0, n_1, \) and \( n_2 \) to \( v_0, v_1, \) and \( v_2, \) respectively, such that there is a \( R \)-path from \( c \) to \( v_0 \), \( N(n_0, n_1, n_2) \) holds, and \( n_1 \) and \( n_2 \) are distinct nodes. In this case, we say that merge operation of \( n_1 \) and \( n_2 \) is enabled (by \( c[R]p \)). The nodes \( n_1 \) and \( n_2 \) can be merge to discharge this assignment (other merge operations might still be enabled after merging \( n_1 \) and \( n_2 \)).

Edge-Addition Operation Let \( p(v_0) \equiv N(v_0, v_1, v_2) \Rightarrow v_1 \overset{f}{\rightarrow} v_2 \) be an edge pattern. If a graph violates a reachability constraint \( c[R]p \), we can assign nodes \( n_0, n_1, \) and \( n_2 \) to \( v_0, v_1, \) and \( v_2, \) respectively, such that there is a \( R \)-path from \( c \) to \( v_0 \), \( N(n_0, n_1, n_2) \) holds, and there is no \( f \)-edge from \( n_1 \) to \( n_2 \). In this case, we say that edge-operation is enabled (by \( c[R]p \)). We can add an \( f \)-edge from \( n_1 \) and \( n_2 \) to discharge this assignment.

The following lemma is the key observation of this proof.

**Lemma 1.** The class of \( A_k \) graphs is closed under merge operations of nodes in distance at most two and edge-addition operations at distance one.

**Sketch of Proof:** If an edge is added in parallel to an existing one (distance one), it does not affect the Gaifman graph, thus \( A_k \) is closed under edge-addition. The proof that \( A_k \) is closed under merge operations is more subtle [36].
In particular, the class $A_k$ is closed under the merge and edge-addition operations forced by LRP$_2$ formulas. This is the only place in our proof where we use the distance restriction of LRP$_2$ patterns.

Given a graph $S$ that satisfies $\varphi$, we construct the graph $S'$ as follows:

1. For each constraint $i$ in $\varphi$, identify the shortest witness $W_i$ in $S$. Let $W'_i$ be the result of splitting the witness $W_i$.
2. The graph $S_0$ is a union of all $W'_i$'s, in which the nodes labelled with the (syntactically) same constants are merged.
3. 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 $S_m$ has no enabled operations.
4. The result $S' = S_r$.

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). The proof proceeds by induction on the process described above. Initially, $S_0$ is in $A_k$. By Lemma 1, all $S_i$ created in the third step of the construction above are in $A_k$; in particular, $S' \in A_k$.

By construction of $S_0$, it contains a witness for each constraint in $\varphi_\Box$, and merge and edge-addition operations preserve the witnesses, thus $S'$ satisfies $\varphi_\Box$. Moreover, $S_0$ satisfies all constraints in $\varphi_\Box$. We show that merge and edge-addition operations applied in the construction preserve $\varphi_\Box$ constraints, thus $S'$ satisfies $\varphi_\Box$. The process above terminates when no merge and edge-addition operations are enabled, that is, $S'$ satisfies $\varphi$. Thus, $S'$ satisfies $\varphi$.

The full proof is available at [36].

4.1 Complexity

We proved decidability by reduction to MSO on trees, which allows us to decide LRP$_2$ formulas using MONA decision procedure [18]. Alternatively, a decision procedure for LRP$_2$ can directly construct a tree automaton from a normal-form formula, and can then check emptiness of the automaton. The worst case complexity of the satisfiability problem of LRP$_2$ formulas is at least doubly-exponential, but it remains elementary (in contrast to MSO on trees, which is non-elementary); we are investigating tighter upper and lower bounds. The complexity depends on the bound $k$ of $A_k$ models, according to Theorem 4. If the routing expressions do not contain constant symbols, then the bound $k$ does not depend on the routing expressions: it depends only on the number of reachability constraints of the form $c_1(R)c_2$. The LRP$_2$ formulas that come up in practice are well-structured, and we hope to achieve a reasonable performance.

5 Limitations and Further Extensions

Despite the fact that LRP$_2$ is useful, there are interesting program properties that cannot be expressed. For example, transitivity of a binary relation, that can be used, e.g., to express partial orders, is naturally expressible in LRP, but not in LRP$_2$. Also, the property
that a general graph is a tree in which each node has a pointer back to the root is expressible in LRP, but not in LRP₂. Notice that the property 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.

There are of course interesting properties that are beyond LRP, 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 LRP₂ while maintaining decidability, perhaps beyond LRP. We are encouraged by the fact that the proof of decidability in Section 4 holds “as is” for many useful extensions. For example, we can generalize the patterns to allow neighborhood formulas with disjunctions and negations of unary relations. In fact, more complex patterns can be used, as long as they do not violate the $A_k$ property. For example, we can define trees rooted at $x$ with parent pointer $b$ from every tree node to its parent by $\text{tree}_{x,x,t,b} \land \text{let } p(v_0) \equiv ((v_1 \xrightarrow{l} v_0) \lor (v_1 \xrightarrow{r} v_0)) \Rightarrow (v_0 \xrightarrow{b} v_1)\text{in } x[(\rightarrow | \rightarrow^*)](\text{det}_b \land p)$. The extended logic remains decidable, because the pattern $p$ adds edges only in parallel to the existing ones.

Currently, reachability constraints describe paths that start from nodes labelled by constants. We can show that the logic remains decidable when reachability constraints are generalized to describe paths that start from any node that satisfies a quantifier-free positive formula $\theta$: $\forall v, w_0, \ldots, w_m, v_0, \ldots, v_n. R(v, v_0) \land \theta(v, w_0, \ldots, w_m) \Rightarrow (N(v_0, \ldots, v_n) \Rightarrow \psi(v_0, \ldots, v_n))$.

6 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 LRP can be seen as a fragment of the first-order logic over graph structures with transitive closure (TC logic [20]). 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 [29, 15, 13].

It can be seen that our logics LRP and LRP₂ are both uncomparable with $L^2 + TC$. Indeed, in LRP no alternation between universal and existential quantification is allowed. On the other hand, LRP₂ allows us to express patterns (e.g., heap sharing) that require more than two variables (see Table 1, Section 3).

In [3], decidable logic $L_r$ (which can also be seen as a fragment of TC) is introduced. The logics LRP and LRP₂ 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 [21, 2, 26, 4] other decidable logics are defined, but their expressive power is rather limited w.r.t. LRP₂ since they allow at most one binary relation symbol (modelling linked data-structures with 1-selector). For instance, the logic of [21] does not allow us to express the reversal of a list. Concerning the class of 1-selector linked data-structures, [6] 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 [7, 25] 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 fix-
point operators. Such an extension is again undecidable in general but the introduction
of the notion of (loosely) guarded quantification allows one to obtain decidable frag-
ments such as $\mu GF$ (or $\mu LGF$) (Guarded Fragment with least and greater fix-
point operators) [14, 12]. Similarly to our logics, the logic $\mu GF$ (and also $\mu LGF$) has the tree
model property: every satisfiable formula has a model of bounded tree width. However,
guarded fixpoint logics are incomparable with $LRP$ and $LRP_2$. For instance, the $LRP_2$
pattern $det_f$ that requires determinism of $f$-field, is not a (loosely) guarded formula.

The PALE system [28] uses an extension of the monadic second order logic on
trees as a specification language. The considered linked data structures are those that
can be defined as graph types [24]. Basically, they are graphs that can be defined as
trees augmented by a set of edges defined using routing expressions (regular expres-
sions) defining paths in the (undirected structure of the) tree. $LRP_2$ allows us to reason
naturally about arbitrary graphs without limitation to tree-like structures. Moreover, as
we show in Section 3, 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 ex-
pressed in the PALE framework: in the list reversal example of Section 3, we show that
the output list is precisely the reversed input list, whereas in the PALE approach, one
can only establish that the output is a list that is the permutation of the input list.

In [22], 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 [17], 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 [32]. The general logic is undecidable [10] but there are few works
showing decidable fragments [10, 4]. One of the fragments is propositional separation
logic where quantification is forbidden [10, 9] and therefore seems to be incomparable
with our logic. The fragment defined in [4] allows one to reason only about singly-
linked lists with explicit sharing. In fact, the fragment considered in [4] can be translated
to $LRP_2$, and therefore, entailment problems as stated in [4] can be solved as implication
problems in $LRP_2$.

7 Conclusions

Defining decidable fragments of first order logic with transitive closure that are useful
for program verification is a difficult task (e.g., [21]). In this paper, we demonstrated
that this is possible by combining three principles: (i) allowing arbitrary boolean com-
binations 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 [21, 2, 26, 4], 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.

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

References