Logical Methods in Computer Science
Vol. 5 (2:12) 2009, pp. 1–30
www.lmcs-online.org

Submitted Apr. 2, 2006
Published May 28, 2009

SIMULATING REACHABILITY USING FIRST-ORDER LOGIC WITH APPLICATIONS TO VERIFICATION OF LINKED DATA STRUCTURES *

TAL LEV-AMI [a], NEIL IMMERMAN [b], THOMAS W. REPS [c], MOOLY SAGIV [d], SIDDHARTH SRIVASTAVA [e], AND GRETA YORSH [f]

[a,d,f] School of Computer Science, Tel Aviv University
e-mail address: tal.levami@cs.tau.ac.il, {msagiv,gretay}@post.tau.ac.il

[b,e] Department of Computer Science, University of Massachusetts, Amherst
e-mail address: {immerman,siddharth}@cs.umass.edu

[c] Computer Science Department, University of Wisconsin, Madison
e-mail address: reps@cs.wisc.edu

ABSTRACT. This paper shows how to harness existing theorem provers for first-order logic to automatically verify safety properties of imperative programs that perform dynamic storage allocation and destructive updating of pointer-valued structure fields. One of the main obstacles is specifying and proving the (absence) of reachability properties among dynamically allocated cells.

The main technical contributions are methods for simulating reachability in a conservative way using first-order formulas—the formulas describe a superset of the set of program states that would be specified if one had a precise way to express reachability. These methods are employed for semi-automatic program verification (i.e., using programmer-supplied loop invariants) on programs such as mark-and-sweep garbage collection and destructive reversal of a singly linked list. (The mark-and-sweep example has been previously reported as being beyond the capabilities of ESC/Java.)

1. INTRODUCTION

This paper explores how to harness existing theorem provers for first-order logic to prove reachability properties of programs that manipulate dynamically allocated data structures. The approach that we use involves simulating reachability in a conservative way using first-order formulas—i.e., the formulas describe a superset of the set of program states that would be specified if one had an accurate way to express reachability.

1998 ACM Subject Classification: F.3.1, F.4.1, F.3.2.

Key words and phrases: First Order Logic, Transitive Closure, Approximation, Program Verification, Program Analysis.

* A preliminary version of this paper appeared in Automated Deduction - CADE-20, 20th International Conference on Automated Deduction, Tallinn, Estonia, July 22-27, 2005.

[a] This research was supported by an Adams Fellowship through the Israel Academy of Sciences and Humanities.

[b,e] Supported by NSF grants CCF-0514621,0541018,0830174.

[c] Supported by ONR under contracts N00014-01-1-{0796,0708}.

[f] Partially supported by the Israeli Academy of Science.

Automatically establishing safety and liveness properties of sequential and concurrent programs that permit dynamic storage allocation and low-level pointer manipulations is challenging. Dynamic allocation causes the state space to be infinite; moreover, a program is permitted to mutate a data structure by destructively updating pointer-valued fields of nodes. These features remain even if a programming language has good capabilities for data abstraction. 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 [Hoa75]. In languages such as Java, concurrency poses yet another challenge: establishing the absence of deadlock requires establishing the absence of any cycle of threads that are waiting for locks held by other threads.

Reachability is crucial 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 cases where reachability is a useful notion include

- Specifying acyclicity of data-structure fragments, i.e., from every element reachable from node n, one cannot reach n
- Specifying the effect of procedure calls when references are passed as arguments: only elements that are reachable from a formal parameter can be modified
- Specifying the absence of deadlocks
- Specifying safety conditions that allow establishing that a data-structure traversal terminates, e.g., there is a path from a node to a sink-node of the data structure.

The verification of such properties presents a challenge. Even simple decidable fragments of first-order logic become undecidable when reachability is added [GME99, IRR$^+$04a]. Moreover, the utility of monadic second-order logic on trees is rather limited because (i) many programs allow non-tree data structures, (ii) expressing the postcondition of a procedure (which is essential for modular reasoning) usually requires referring to the pre-state that holds before the procedure executes, and thus cannot, in general, be expressed in monadic second-order logic on trees—even for procedures that manipulate only singly-linked lists, such as the in-situ list-reversal program shown in Fig. 6, and (iii) the complexity is prohibitive.

While our work was actually motivated by our experience using abstract interpretation – and, in particular, the TVLA system [LAS00, SRW02, RSW04] – to establish properties of programs that manipulate heap-allocated data structures, in this paper, we consider the problem of verifying data-structure operations, assuming that we have user-supplied loop invariants. This is similar to the approach taken in systems like ESC/Java [FLL$^+$02], and Pale [MS01].

The contributions of the paper can be summarized as follows:

Handling FO(TC) **formulas using** FO **theorem provers.** We want to use first-order theorem provers and we need to discuss the transitive closure of certain binary predicates, f. However, first-order theorem provers cannot handle transitive closure. We solve this conundrum by adding a new relation symbol f_{tc} for each such f, together with first-order axioms that assure that f_{tc} is interpreted correctly. The theoretical details of how this is done are presented in Section 3. The fact that we are able to handle transitive closure effectively and reasonably automatically is quite surprising.

As explained in Section 3, the axioms that we add to control the behavior of the added predicates, f_{tc}, must be sound but not necessarily complete. One way to think about this is that we are simulating a formula, χ, in which transitive closure occurs, with a pure first-order formula χ'. If our axioms are not complete then we are allowing χ' to denote more stores than χ does. The study of methods that are sound but potentially incomplete is motivated by the fact that *abstraction* [CC77]

can be an aid in the verification of many properties. In terms of logic, abstraction corresponds to using formulas that describe a superset of the set of program states that can actually arise. A definite answer about whether a property always holds can sometimes be obtained even when information has been lost because of abstraction.

If χ' is proven valid in FO then χ is also valid in FO(TC); however, if we fail to prove that χ' is valid, it is still possible that χ is valid: the failure would be due to the incompleteness of the axioms, or the lack of time or space for the theorem prover to complete the proof.

As we will see in Section 3, it is easy to write a sound axiom, $T_1[f]$, that is "complete" in the very limited sense that every finite, acyclic model satisfying $T_1[f]$ must interpret f_{tc} as the reflexive, transitive closure of its interpretation of f. However, in practice this is not worth much because, as is well-known, finiteness is not expressible in first-order logic. Thus, the properties that we want to prove do not follow from $T_1[f]$. We do prove that $T_1[f]$ is complete for positive transitive-closure properties (Proposition 3.2). The real difficulty lies in proving properties involving the negation of f_{tc}, i.e., that a certain f-path does not exist.

Induction axiom scheme. To solve the above problem, we add an induction axiom scheme. Although in general, there is no complete, recursively-enumerable axiomatization of transitive closure (Proposition 4.1), we have found, on the practical side, that on the examples we have tried, T_1 plus induction allows us to automatically prove all of our desired properties. On the theoretical side, we prove that our axiomatization is complete for word models (Theorem 4.8).

We think of the axioms that we use as aides for the first-order theorem prover that we employ (SPASS [WGR96]) to prove the properties in question. Rather than giving SPASS many instances of the induction scheme, our experience is that it finds the proof faster if we give it several axioms that are simpler to use than induction. As already mentioned, the hard part is to show that certain paths do not exist.

Coloring axiom schemes. In particular, we use three axiom schemes, having to do with partitioning memory into a small set of colors. We call instances of these schemes "coloring axioms". Our coloring axioms are simple, and are ***easily proved using* SPASS *(in under ten seconds) from the induction axioms***. For example, the first coloring axiom scheme, **NoExit**$[A, f]$, says that if no f-edges leave color class, A, then no f-paths leave A. It turns out that the **NoExit** axiom scheme implies – and thus is equivalent to – the induction scheme. However, we have found in practice that explicitly adding other coloring axioms (which are consequences of **NoExit**) enables SPASS to prove properties that it otherwise fails at.

We first assume that the programmer provides the colors by means of first-order formulas with transitive closure. Our initial experience indicates that the generated coloring axioms are useful to SPASS. In particular, it provides the ability to verify programs like the mark phase of a mark-and-sweep garbage collector. This example has been previously reported as being beyond the capabilities of ESC/Java. TVLA also succeeds on this example; however our new approach provides verification methods that can in some instances be more precise than TVLA.

Prototype implementation. Perhaps most exciting, we have implemented the heuristics for selecting colors and their corresponding axioms in a prototype using SPASS. We have used this to automatically choose useful color axioms and then verify a series of small heap-manipulating programs. We believe that the detailed examples presented here give convincing evidence of the promise of our methodology. Of course much further study is needed.

Strengthening Nelson's results. Greg Nelson considered a set of axiom schemes for reasoning about reachability in function graphs, i.e., graphs in which there is at most one f-edge leaving any node [Nel83]. He left open the question of whether his axiom schemes were complete for function

graphs. We show that Nelson's axioms are provable from T_1 plus our induction axioms. We also show that Nelson's axioms are not complete: in fact, they do not imply **NoExit**.

Outline. The remainder of the paper is organized as follows: Section 2 explains our notation and the setting; Section 3 fills in our formal framework, introduces the induction axiom scheme, and presents the coloring axiom schemes; Section 4 provides more detail about TC-completeness including a description of Nelson's axioms, a proof that they are not TC-complete for the functional case, and a proof that our axiomatization is TC-complete for words; Section 5 presents our heuristics including the details of their successful use on a variety of examples; Section 6 describes the applicability of our methodology, relating it to the reasoning done in the TVLA system; Section 7 describes some related work; and Section 8 describes some conclusions and future directions.

2. PRELIMINARIES

This section defines the basic notations used in this paper and the setting.

2.1. **Notation.** *Syntax*: A relational **vocabulary** $\tau = \{p_1, p_2, \ldots, p_k\}$ is a set of relation symbols, each of fixed arity. We use the letters u, v, and w (possibly with numeric subscript) for first-order variables. We write first-order formulas over τ with quantifiers $\forall$ and $\exists$, logical connectives $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$, and $\neg$, where atomic formulas include: equality, $p_i(v_1, v_2, \ldots v_{a_i})$, and $\mathrm{TC}[f](v_1, v_2)$, where $p_i \in \tau$ is of arity a_i and $f \in \tau$ is binary. Here $\mathrm{TC}[f](v_1, v_2)$ denotes the existence of a finite path of 0 or more f edges from v_1 to v_2. A formula without TC is called a **first-order** formula.

We use the following precedence of logical operators: $\neg$ has highest precedence, followed by $\wedge$ and $\vee$, followed by $\rightarrow$ and $\leftrightarrow$, and $\forall$ and $\exists$ have lowest precedence.

Semantics: A **model**, $\mathcal{A}$, of vocabulary τ, consists of a non-empty universe, $|\mathcal{A}|$, and a relation $p^{\mathcal{A}}$ over the universe interpreting each relation symbol $p \in \tau$. We write $\mathcal{A} \models \varphi$ to mean that the formula φ is true in the model $\mathcal{A}$. For Σ a set of formulas, we write $\Sigma \models \varphi$ (Σ semantically implies φ) to mean that all models of Σ satisfy φ.

2.2. **Setting.** We are primarily interested in formulas that arise while proving the correctness of programs. We assume that the programmer specifies pre and post-conditions for procedures and loop invariants using first-order formulas with transitive closure on binary relations. The transformer for a loop body can be produced automatically from the program code.

For instance, to establish the partial correctness with respect to a user-supplied specification of a program that contains a single loop, we need to establish three properties: First, the loop invariant must hold at the beginning of the first iteration; i.e., we must show that the loop invariant follows from the precondition and the code leading to the loop. Second, the loop invariant provided by the user must be maintained; i.e., we must show that if the loop invariant holds at the beginning of an iteration and the loop condition also holds, the transformer causes the loop invariant to hold at the end of the iteration. Finally, the postcondition must follow from the loop invariant and the condition for exiting the loop.

In general, these formulas are of the form

$$\psi_1[\tau] \wedge Tr[\tau, \tau'] \rightarrow \psi_2[\tau']$$

where τ is the vocabulary of the before state, τ' is the vocabulary of the after state,[1] and Tr is the transformer, which may use both the before and after predicates to describe the meaning of the module to be executed. If symbol f denotes the value of a predicate before the operation, then f' denotes the value of the same predicate after the operation.

An interesting special case is the proof of the maintenance formula of a loop invariant. This has the form:

$$LC[\tau] \wedge LI[\tau] \wedge Tr[\tau, \tau'] \rightarrow LI[\tau']$$

Here LC is the condition for entering the loop and LI is the loop invariant. $LI[\tau']$ indicates that the loop invariant remains true after the body of the loop is executed.

The challenge is that the formulas of interest contain transitive closure; thus, the validity of these formulas cannot be directly proven using a theorem prover for first-order logic.

3. Axiomatization of Transitive Closure

The original formula that we want to prove, χ, contains transitive closure, which first-order theorem provers cannot handle. To address this problem, we replace χ by a new formula, χ', where all appearances of $\mathrm{TC}[f]$ have been replaced by the new binary relation symbol, f_{tc}.

We show in this paper that from χ', we can often automatically generate an appropriate first-order axiom, σ, with the following two properties:

(1) if $\sigma \rightarrow \chi'$ is valid in FO, then χ is valid in FO(TC).
(2) A theorem prover successfully proves that $\sigma \rightarrow \chi'$ is valid in FO.

We now explain the theory behind this process. A **TC model**, $\mathcal{A}$, is a model such that if f and f_{tc} are in the vocabulary of $\mathcal{A}$, then $(f_{\mathrm{tc}})^{\mathcal{A}} = (f^{\mathcal{A}})^{\star}$; i.e., $\mathcal{A}$ interprets f_{tc} as the reflexive, transitive closure of its interpretation of f.

A first-order formula φ is **TC valid** iff it is true in all TC models. We say that an axiomatization, Σ, is **TC sound** if every formula that follows from Σ is TC valid. Since first-order reasoning is sound, Σ is TC sound iff every $\sigma \in \Sigma$ is TC valid.

We say that Σ is **TC complete** if for every TC-valid φ, $\Sigma \models \varphi$. If Σ is TC complete and TC sound, then for all first-order φ,

$$\Sigma \models \varphi \quad \Leftrightarrow \quad \varphi \qquad \text{is TC valid}$$

Thus a TC-complete set of axioms proves exactly the first-order formulas, χ', such that the corresponding FO(TC) formula, χ, is valid.

All the axioms that we consider are TC valid. There is no recursively enumerable TC-complete axiom system (Proposition 4.1). However, the axiomatization that we give does allow SPASS to prove all the desired properties on the examples that we have tried. We do prove that our axiomatization is TC complete for word models (Theorem 4.8).

[1] In some cases it is useful for the postcondition formula to refer to the original vocabulary as well. This way the postcondition can summarize some of the behavior of the transformer, e.g., summarize the behavior of an entire procedure.

3.1. **Some TC-Sound Axioms.** We begin with our first TC axiom scheme. For any binary relation symbol, f, let,

$$T_1[f] \quad \equiv \quad \forall u, v \,.\, f_{\rm tc}(u,v) \;\leftrightarrow\; (u = v) \vee \exists w \,.\, f(u,w) \wedge f_{\rm tc}(w,v)$$

We first observe that $T_1[f]$ is "complete" in a very limited way for finite, acyclic graphs, i.e., $T_1[f]$ exactly characterizes the meaning of $f_{\rm tc}$ for all finite, acyclic graphs. The reason that we say this is limited is that it does not give us a complete set of first-order axioms: as is well known, there is no first-order axiomatization of "finite".

Proposition 3.1. *Any finite and acyclic model of $T_1[f]$ is a TC model.*

Proof. Let $\mathcal{A} \models T_1[f]$ where $\mathcal{A}$ is finite and acyclic. Let $a_0, b \in |\mathcal{A}|$. Assume that there is an f-path from a_0 to b. Since $\mathcal{A} \models T_1[f]$, it is easy to see that $\mathcal{A} \models f_{\rm tc}(a_0, b)$. Conversely, suppose that $\mathcal{A} \models f_{\rm tc}(a_0, b)$. If $a_0 = b$, then there is a path of length 0 from a_0 to b. Otherwise, by $T_1[f]$, there exists an $a_1 \in |\mathcal{A}|$ such that $\mathcal{A} \models f(a_0, a_1) \wedge f_{\rm tc}(a_1, b)$. Note that $a_1 \neq a_0$ since $\mathcal{A}$ is acyclic. If $a_1 = b$ then there is an f-path of length 1 from a to b. Otherwise there must exist an $a_2 \in |\mathcal{A}|$ such that $\mathcal{A} \models f(a_1, a_2) \wedge f_{\rm tc}(a_2, b)$ and so on, generating a set $\{a_1, a_2, \ldots\}$. None of the a_i can be equal to a_j, for $j < i$, by acyclicity. Thus, by finiteness, some $a_i = b$. Hence $\mathcal{A}$ is a TC model. □

Let $T_1'[f]$ be the $\leftarrow$ direction of $T_1[f]$:

$$T_1'[f] \quad \equiv \quad \forall u, v \,.\, f_{\rm tc}(u,v) \;\leftarrow\; (u = v) \vee \exists w \,.\, f(u,w) \wedge f_{\rm tc}(w,v)$$

Proposition 3.2. *Let $f_{\rm tc}$ occur only positively in φ. If φ is TC valid, then $T_1'[f] \models \varphi$.*

Proof. Suppose that $T_1'[f] \not\models \varphi$. Let $\mathcal{A} \models T_1'[f] \wedge \neg\varphi$. Note that $f_{\rm tc}$ occurs only negatively in $\neg\varphi$. Furthermore, since $\mathcal{A} \models T_1'[f]$, it is easy to show by induction on the length of the path, that if there is an f-path from a to b in $\mathcal{A}$, then $\mathcal{A} \models f_{\rm tc}(a, b)$. Define $\mathcal{A}'$ to be the model formed from $\mathcal{A}$ by interpreting $f_{\rm tc}$ in $\mathcal{A}'$ as $(f^{\mathcal{A}})^\star$. Thus $\mathcal{A}'$ is a TC model and it only differs from $\mathcal{A}$ by the fact that we have removed zero or more pairs from $(f_{\rm tc})^{\mathcal{A}}$ to form $(f_{\rm tc})^{\mathcal{A}'}$. Because $\mathcal{A} \models \neg\varphi$ and $f_{\rm tc}$ occurs only negatively in $\neg\varphi$, it follows that $\mathcal{A}' \models \neg\varphi$, which contradicts the assumption that φ is TC valid. □

Proposition 3.2 shows that proving positive facts of the form $f_{\rm tc}(u, v)$ is easy; it is the task of proving that paths do not exist that is more subtle.

Proposition 3.1 shows that what we are missing, at least in the acyclic case, is that there is no first-order axiomatization of finiteness. Traditionally, when reasoning about the natural numbers, this problem is mitigated by adding induction axioms. We next introduce an induction scheme that, together with T_1, seems to be sufficient to prove any property we need concerning TC.

Notation: In general, we will use F to denote the set of all binary relation symbols, f, such that $\mathrm{TC}[f]$ occurs in a formula we are considering. If $\varphi[f]$ is a formula in which f occurs, let $\varphi[F] = \bigwedge_{f \in F} \varphi[f]$. Thus, for example, $T_1[F]$ is the conjunction of the axiom $T_1[f]$ for all binary relation symbols, f, under consideration.

Definition 3.3. For any first-order formulas $Z(u), P(u)$, and binary relation symbol, f, let the **induction principle**, **IND**$[Z, P, f]$, be the following first-order formula:

$$\begin{aligned}(\forall w \,.\, Z(w) \rightarrow P(w)) \quad &\wedge \quad (\forall u, v \,.\, P(u) \wedge f(u,v) \rightarrow P(v)) \\ &\rightarrow \quad \forall u, w \,.\, Z(w) \wedge f_{\rm tc}(w,u) \rightarrow P(u)\end{aligned}$$

In order to explain the meaning of **IND** and other axioms it is important to remember that we are trying to write axioms, Σ, that are,

- **TC valid**, i.e., true in all TC models, and
- **useful**, i.e., all models of Σ are sufficiently like TC models that they satisfy the TC-valid properties we want to prove.

To make the meaning of our axioms intuitively clear, in this section we will say, for example, that "y is $f_{\rm tc}$-reachable from x" to mean that $f_{\rm tc}(x, y)$ holds. Later, we will assume that the reader has the idea and just say "reachable" instead of "$f_{\rm tc}$-reachable".

The intuitive meaning of the induction principle is that if every zero point satisfies P, and P is preserved when following f-edges, then every point $f_{\rm tc}$-reachable from a zero point satisfies P. Obviously this principle is TC valid, i.e., it is true for all structures such that $f_{\rm tc} = f^\star$.

As an easy application of the induction principle, consider the following cousin of $T_1[f]$,

$$T_2[f] \quad \equiv \quad \forall u, v \,.\, f_{\rm tc}(u, v) \;\leftrightarrow\; (u = v) \vee \exists w \,.\, f_{\rm tc}(u, w) \wedge f(w, v)$$

The difference between T_1 and T_2 is that T_1 requires that each path represented by $f_{\rm tc}$ starts with an f edge and T_2 requires the path to end with an f edge. It is easy to see that neither of $T_1[f]$, $T_2[f]$ implies the other. However, in the presence of the induction principle they do imply each other. For example, it is easy to prove $T_2[f]$ from $T_1[f]$ using **IND**$[Z, P, f]$ where $Z(v) \equiv v = u$ and $P(v) \equiv u = v \vee \exists w \,.\, f_{\rm tc}(u, w) \wedge f(w, v)$. Here, for each u we use **IND**$[Z, P, f]$ to prove by induction that every v reachable from u satisfies the right-hand side of $T_2[f]$.

Another useful axiom scheme provable from T_1 plus **IND** is the transitivity of reachability:

$$\textbf{Trans}[f] \;\equiv\; \forall u, v, w \,.\, f_{\rm tc}(u, w) \wedge f_{\rm tc}(w, v) \rightarrow f_{\rm tc}(u, v)$$

3.2. **Coloring Axioms.** We next describe three TC-sound axioms schemes that are not implied by $T_1[F] \wedge T_2[F]$, and are provable from the induction principle. We will see in the sequel that these coloring axioms are very useful in proving that paths do not exist, permitting us to verify a variety of algorithms. In Section 5, we will present some heuristics for automatically choosing particular instances of the coloring axiom schemes that enable us to prove our goal formulas.

The first coloring axiom scheme is the NoExit axiom scheme:

$$(\forall u, v \,.\, A(u) \wedge \neg A(v) \rightarrow \neg f(u, v)) \;\rightarrow\; \forall u, v \,.\, A(u) \wedge \neg A(v) \rightarrow \neg f_{\rm tc}(u, v)$$

for any first-order formula $A(u)$, and binary relation symbol, f, **NoExit**$[A, f]$ says that if no f-edge leaves color class A, then no point outside of A is $f_{\rm tc}$-reachable from A.

Observe that although it is very simple, **NoExit**$[A, f]$ does not follow from $T_1[f] \wedge T_2[f]$. Let $G_1 = (V, f, f_{\rm tc}, A)$ be a model consisting of two disjoint cycles: $V = \{1, 2, 3, 4\}$, $f = \{\langle 1, 2\rangle, \langle 2, 1\rangle, \langle 3, 4\rangle, \langle 4, 3\rangle\}$, and $A = \{1, 2\}$. Let $f_{\rm tc}$ have all 16 possible pairs. Thus G_1 satisfies $T_1[f] \wedge T_2[f]$ but violates **NoExit**$[A, f]$. Even for acyclic models, **NoExit**$[A, f]$ does not follow from $T_1[f] \wedge T_2[f]$ because there are infinite models in which the implication does not hold (Proposition 4.7).

NoExit$[A, f]$ follows easily from the induction principle: if no f-edges leave A, then induction tells us that everything $f_{\rm tc}$-reachable from a point in A satisfies A. Similarly, **NoExit**$[A, f]$ implies the induction axiom, **IND**$[Z, A, f]$, for any formula Z.

The second coloring axiom scheme is the GoOut axiom: for any first-order formulas $A(u), B(u)$, and binary relation symbol, f, **GoOut**$[A, B, f]$ says that if the only f-edges leaving color class A are to B, then any $f_{\rm tc}$-path from a point in A to a point not in A must pass through B.

$$\begin{array}{rcl} (\forall u, v \,.\, A(u) \wedge \neg A(v) \wedge f(u, v) \rightarrow B(v)) & \rightarrow & \\ \forall u, v \,.\, A(u) \wedge \neg A(v) \wedge f_{\rm tc}(u, v) & \rightarrow & \exists w \,.\, B(w) \wedge f_{\rm tc}(u, w) \wedge f_{\rm tc}(w, v) \end{array}$$

To see that **GoOut**$[A, B, f]$ follows from the induction principle, assume that the only f-edges out of A enter B. For any fixed u in A, we prove by induction that any point v f_{tc}-reachable from u is either in A or has a predecessor, b in B, that is f_{tc}-reachable from u.

The third coloring axiom scheme is the **NewStart** axiom, which is useful in the context of dynamically changing graphs: for any first-order formula $A(u)$, and binary relation symbols f and g, think of f as the previous edge relation and g as the current edge relation. **NewStart**$[A, f, g]$ says that if there are no new edges between A nodes, then any new path, i.e., g_{tc} but not f_{tc}, from A must leave A to make its change:

$$\begin{array}{rcl}(\forall u, v \,.\, A(u) \wedge A(v) \wedge g(u,v) \rightarrow f(u,v)) & \rightarrow & \\ \forall u, v \,.\, g_{\mathrm{tc}}(u,v) \wedge \neg f_{\mathrm{tc}}(u,v) & \rightarrow & \exists w \,.\, \neg A(w) \wedge g_{\mathrm{tc}}(u,w) \wedge g_{\mathrm{tc}}(w,v)\end{array}$$

NewStart$[A, f, g]$ follows from the induction principle by a proof that is similar to the proof of **GoOut**$[A, B, f]$.

3.2.1. *Linked Lists.* The spirit behind our consideration of the coloring axioms is similar to that found in a paper of Greg Nelson's in which he introduced a set of reachability axioms for a functional predicate, f, i.e., there is at most one f edge leaving any point [Nel83]. Nelson asked whether his axiom schemes are complete for the functional setting. We remark that Nelson's axiom schemes are provable from T_1 plus our induction principle. However, Nelson's axiom schemes are not complete: we constructed a functional graph that satisfies Nelson's axioms but violates **NoExit**$[A, f]$ (Proposition 4.7).

At least one of Nelson's axiom schemes seems orthogonal to our coloring axioms and may be useful in certain proofs. Nelson's fifth axiom scheme states that the points reachable from a given point are linearly ordered. The soundness of the axiom scheme is due to the fact that f is functional. We make use of a simplified version of Nelson's ordering axiom scheme: Let **Func**$[f] \equiv \forall u, v, w \,.\, f(u,v) \wedge f(u,w) \rightarrow v = w$; then,

$$\textbf{Order}[f] \;\equiv\; \textbf{Func}[f] \rightarrow \forall u, v, w \,.\, f_{\mathrm{tc}}(u,v) \wedge f_{\mathrm{tc}}(u,w) \;\rightarrow\; f_{\mathrm{tc}}(v,w) \vee f_{\mathrm{tc}}(w,v)$$

3.2.2. *Trees.* When working with programs manipulating trees, we have a fixed set of selectors Sel and transitive closure is performed on the $down$ relation, defined as

$$\forall v_1, v_2 \,.\, down(v_1, v_2) \;\leftrightarrow\; \bigvee_{s \in Sel} s(v_1, v_2)$$

Trees have no sharing (i.e., the $down$ relation is injective), thus a similar axiom to **Order**$[f]$ is used:

$$\forall u, v, w \,.\, down_{\mathrm{tc}}(v,u) \wedge down_{\mathrm{tc}}(w,u) \;\rightarrow\; down_{\mathrm{tc}}(v,w) \vee down_{\mathrm{tc}}(w,v)$$

Another important property of trees is that the subtrees below distinct children of a node are disjoint. We use the following axioms to capture this, where $s_1 \neq s_2 \in Sel$:

$$\forall v, v_1, v_2, w \,.\, \neg(s_1(v,v_1) \wedge s_2(v,v_2) \wedge down_{\mathrm{tc}}(v_1,w) \wedge down_{\mathrm{tc}}(v_2,w))$$

4. On TC-Completeness

In this section we consider the concept of TC-Completeness in detail. The reader anxious to see how we use our methodology is encouraged to skim or skip this section.

We first show that there is no recursively enumerable TC-complete set of axioms.

Proposition 4.1. *Let Γ be an r.e. set of TC-valid first-order sentences. Then Γ is not TC-complete.*

Proof. By the proof of Corollary 9, page 11 of [IRR$^+$04a], there is a recursive procedure that, given any Turing machine M_n as input, produces a first-order formula φ_n in a vocabulary τ_n such that φ_n is TC-valid iff Turing machine, M_n, on input 0 never halts. The vocabulary τ_n consists of the two binary relation symbols, $E, E_{\rm tc}$, constant symbols, a, d, and some unary relation symbols. It follows that if Γ were TC-complete, then it would prove all true instances of φ_n and thus the halting problem would be solvable. □

Proposition 4.1 shows that even in the presence of only one binary relation symbol, there is no r.e. TC-complete axiomatization.

In [Avr03], Avron gives an elegant finite axiomatization of the natural numbers using transitive closure, a successor relation and the binary function symbol, "+". Furthermore, he shows that multiplication is definable in this language. Since the unique TC-model for Avron's axioms is the standard natural numbers it follows that:

Corollary 4.2. *Let Γ be an arithmetic set of TC-valid first-order sentences over a vocabulary including a binary relation symbol and a binary function symbol (or a ternary relation symbol). Then Γ is not TC-complete.*

In Proposition 3.1 we showed that any finite and acyclic model of $T_1[f]$ is a TC model. This can be strengthened to

Proposition 4.3. *Any finite model of T_1 plus* **IND** *is a TC-model.*

Proof. Let $\mathcal{A}$ be a finite model of T_1 plus **IND**. Let f be a binary relation symbol, and let a, b be elements of the universe of $\mathcal{A}$. Since $\mathcal{A} \models T_1$, if there is an f path from a to b then $\mathcal{A} \models f_{\rm tc}(a, b)$.

Conversely, suppose that there is no f path from a to b. Let R_a be the set of elements of the universe of $\mathcal{A}$ that are reachable from a. Let $k = |R_a|$. Since $\mathcal{A}$ is finite we may use existential quantification to name exactly all the elements of R_a : $x_1, \ldots, x_k$. We can then define the color class: $C(y) \equiv y = x_1 \vee \cdots \vee y = x_k$. Then we can prove using **IND**, or equivalently **NoExit**, that no vertex outside this color class is reachable from a, i.e., $\mathcal{A} \models \neg f_{\rm tc}(a, b)$. Thus, as desired, $\mathcal{A}$ is a TC-model. □

4.1. **More About TC-Completeness.** Even though there is no r.e. set of TC-complete axioms in general, there are TC-complete axiomatizations for certain interesting cases. Let Σ be a set of formulas. We say that ψ is *TC-valid wrt* Σ iff every TC-model of Σ satisfies ψ. Let Γ be TC-sound. We say that Γ is *TC-complete wrt* Σ iff $\Gamma \cup \Sigma \vdash \psi$ for every ψ that is TC-valid wrt Σ. We are interested in whether T_1 plus **IND** is TC-complete with respect to interesting theories, Σ.

Since $\mathrm{TC}[s](a, b)$ asserts the existence of a finite s-path from a to b, we can express that a structure is finite by writing the formula: $\Phi \equiv$ **Func**$[s] \wedge \exists x \forall y \,.\, s_{\rm tc}(x, y)$. Observe that every TC-model that satisfies Φ is finite. Thus, if we are in a setting – as is frequent in logic – where we may add a new binary relation symbol, s, then **finiteness is TC-expressible**.

Proposition 4.4. *Let Σ be a finite set of formulas, and Γ an r.e., TC-complete axiomatization wrt Σ in a language where finiteness is TC-expressible. Then finite TC-validity for Σ is decidable.*

Proof. Let Φ be a formula as above that TC-expresses finiteness. Let ψ be any formula. If ψ is not finite TC-valid wrt Σ, then we can find a finite TC model of Σ where ψ is false. If ψ is finite TC-valid, then $\Gamma \cup \Sigma \vdash \Phi \rightarrow \psi$, and we can find this out by systematically generating all proofs from Γ. □

From Proposition 4.4 we know that we must restrict our search for cases of TC-completeness to those where finite TC-validity is decidable. In particular, since the finite theory of two functional relations is undecidable, e.g., [IRR$^+$04a], we know that,

Corollary 4.5. *There are no r.e. TC-valid axioms for the functional case even if we restrict to at most two binary relation symbols.*

4.2. **Nelson's Axioms.** Our idea of considering transitive-closure axioms is similar in spirit to the approach that Nelson takes [Nel83]. To prove some program properties, he introduces a set of reachability axiom schemes for a functional predicate, f. By "functional" we mean that f is a partial function: $\mathbf{Func}[f] \equiv \forall u, v, w \,.\, f(u, v) \wedge f(u, w) \rightarrow v = w$.

We remark that Nelson's axiom schemes are provable from T_1 plus our induction principle. At least two of his schemes may be useful for us to add in our approach. Nelson asked whether his axioms are complete for the functional setting. It follows from Corollary 4.5 that the answer is no. We prove below that Nelson's axioms do not prove **NoExit**.

Nelson's basic relation symbols are ternary. For example, he writes "$u \xrightarrow[x]{f} v$" to mean that there is an f-path from u to v that follows no edges out of x. We encode this as, $f^x_{\mathrm{tc}}(u, v)$, where, for each parameter x we add a new relation symbol, f^x, together with the assertion: $\forall u, v \,.\, f^x(u, v) \leftrightarrow f(u, v) \wedge (u \neq x)$. Nelson also includes a notation for modifying the partial function f. He writes, $f^{(p)}_q$ for the partial function that agrees with f everywhere except on argument p where it has value q. Nelson's eighth axiom scheme asserts a basic consistency property for this notation. In our translation we simply assert that $f^{(p)}_q(u, v) \leftrightarrow (u \neq p \wedge f(u, v)) \vee (u = p \wedge v = q)$. When we translate Nelson's eighth axiom scheme the result is tautological, so we can safely omit it.

Using our translation, Nelson's axiom schemes are the following.

(N1) $f^x_{\mathrm{tc}}(u, v) \;\leftrightarrow\; (u = v) \vee \exists z \,.\, (f^x(u, z) \wedge f^x_{\mathrm{tc}}(z, v))$

(N2) $f^x_{\mathrm{tc}}(u, v) \wedge f^x_{\mathrm{tc}}(v, w) \rightarrow f^x_{\mathrm{tc}}(u, w)$

(N3) $f^x_{\mathrm{tc}}(u, v) \rightarrow f_{\mathrm{tc}}(u, v)$

(N4) $f^y_{\mathrm{tc}}(u, x) \wedge f^z_{\mathrm{tc}}(u, y) \rightarrow f^z_{\mathrm{tc}}(u, x)$

(N5) $f_{\mathrm{tc}}(u, x) \rightarrow f^y_{\mathrm{tc}}(u, x) \vee f^x_{\mathrm{tc}}(u, y)$

(N6) $f^y_{\mathrm{tc}}(u, x) \wedge f^z_{\mathrm{tc}}(u, y) \rightarrow f^z_{\mathrm{tc}}(x, y)$

(N7) $f(x, u) \wedge f_{\mathrm{tc}}(u, v) \rightarrow f^x_{\mathrm{tc}}(u, v)$

These axiom schemes can be proved using appropriate instances of T_1 and the induction principle. Just as we showed in Proposition 3.1 that any finite and acyclic model of $T_1[f]$ is a TC model, we have that,

Proposition 4.6. *Any finite and functional model of Nelson's axioms is a TC-model.*

Proof. Consider any finite and function model, $\mathcal{M}$. We claim that for each f and $x \in |\mathcal{M}|$, $(f^x_{\mathrm{tc}})^{\mathcal{M}} = ((f^x)^{\mathcal{M}})^\star$. If there is an f^x path from u to v, then it follows from repeated uses of (N1) that f^x_{tc} holds.

If there is no f^x path from u to v and u is not on an f-cycle, then using (N1) we can follow f-edges from u to the end and prove that f^x_{tc} does not hold.

If there is no f^x path from u to v and u is on an f-cycle containing x, then using (N1) we can follow f-edges from u to x to prove that $f^x_{\mathrm{tc}}(u, v)$ does not hold.

Finally, if there is no f path from u to v and u is on an f-cycle, suppose for the sake of a contradiction that $f_{\mathrm{tc}}(u, v)$ holds. Let x be the predecessor of u on the cycle. By N7, $f^x_{\mathrm{tc}}(u, v)$ must hold. However, this contradicts the previous paragraph. □

Axiom schemes (N5) and (N7) may be useful for us to assert when f is functional. (N5) says that the points reachable from u are totally ordered in the sense that if x and y are both reachable from u, then in the path from u either x comes first or y comes first. (N7) says that if there is an edge from x to u and a path from u to v, then there is a path from u to v that does not go through x. This implies the useful property that no vertex not on a cycle is reachable from a vertex on the cycle.

We conclude this section by proving the following,

Proposition 4.7. *Nelson's axioms do not imply* **NoExit**.

Proof. Consider the structure $G = (V, f, f_{\mathrm{tc}}, f^0_{\mathrm{tc}}, f^1_{\mathrm{tc}}, f^2_{\mathrm{tc}}, \ldots, f^\infty_{\mathrm{tc}}, A)$ such that $V = \mathbf{N} \cup \{\infty\}$, the set of natural numbers plus a point at infinity. Let $A = \mathbf{N}$, i.e., the color class A is interpreted as all points except ∞. Define $f = \{\langle u, u+1\rangle \mid u \in \mathbf{N}\}$, i.e., there is an edge from every natural number to its successor, but ∞ is isolated. However, let $f_{\mathrm{tc}} = \{\langle u, v\rangle \mid u \leq v\}$, i.e., G believes that there is a path from each natural number to infinity. Similarly, for each $k \in V$, $f^k_{\mathrm{tc}} = \{\langle u, v\rangle | u \leq v \wedge (k < u \vee v \leq k)\}$.

It is easy to check that G satisfies all of Nelson's axioms.

The problem is that $G \models \neg$**NoExit**$[A, f]$. It follows that Nelson's axioms do not entail **NoExit**$[A, f]$. This is another proof that they are not TC complete. □

4.3. **TC-Completeness for Words.** In this subsection, we prove that T_1 plus **IND** is TC-complete for words.

For any alphabet, Σ, let the vocabulary of words over Σ be $vocab(\Sigma) = \langle 0, max; s^2, s^2_{\mathrm{tc}}, P^1_\sigma : \sigma \in \Sigma\rangle$. The domain of a word model is an ordered set of positions, and the unary relation $P_\sigma(x)$ expresses the presence of symbol σ at position x. s is the successor relation over positions, and s_{tc} is its transitive closure. The constants 0 and max represent the first and last positions in the word. A simple axiomatization of words is $A_{\Sigma w}$, the conjunction of the following four statements:

(A1) $\forall x\,.\,(\neg s(x, 0) \wedge \neg s(max, x) \wedge (x \neq 0 \rightarrow \exists y\,.\,s(y, x)) \wedge (x \neq max \rightarrow \exists y\,.\,s(x, y)))$

(A2) $\forall xyz\,.\,((s(x, y) \wedge s(x, z)) \vee (s(y, x) \wedge s(z, x))) \rightarrow y = z$

(A3) $\forall x\,.\,s_{\mathrm{tc}}(0, x) \wedge s_{\mathrm{tc}}(x, max)$

(A4) $\forall x\,.\,\bigvee_{\sigma\in\Sigma} (P_\sigma(x) \wedge \bigwedge_{\tau\neq\sigma} \neg P_\tau(x))$

In particular, observe that a TC-model of $A_{\Sigma w}$ is exactly a Σ word. Let $\Gamma =$ **IND** $\cup \{T_1\}$. We wish to prove the following:

Theorem 4.8. Γ *is TC-complete wrt* $A_{\Sigma w}$.

We first note that $\Gamma \cup \{A_{\Sigma w}\}$ implies acyclicity: $\forall xy\,.\,s(x,y) \rightarrow \neg s_{\mathrm{tc}}(y,x)$. The proof using induction proceeds as follows: in the base case, there is no loop at 0. Inductively, suppose there is no loop starting at x, $s(x,y)$ holds, but there is a loop at y, i.e., $\exists z\,.\,s(y,z) \wedge s_{\mathrm{tc}}(z,y)$. Then by T_1 and **IND** we know $\exists x'\,.\,s_{\mathrm{tc}}(z,x') \wedge s(x',y)$, and $s_{\mathrm{tc}}(y,x')$. (A2) asserts that the in-degree of s is 1, which means $x' = x$ and we have a contradiction: $s_{\mathrm{tc}}(y,x)$.

In order to prove Theorem 4.8, we need to show that if φ is true in all TC models of $\Gamma \cup \{A_{\Sigma w}\}$, i.e., in all words, then $\Gamma \cup \{A_{\Sigma w}\} \vdash \varphi$. By the completeness of first-order logic it suffices to show that $\Gamma \cup \{A_{\Sigma w}\} \models \varphi$. We prove the contrapositive of this in Lemma 4.10. In order to do so, we first construct a DFA D_φ that has some desirable properties.

Lemma 4.9. *For any $\varphi \in \mathcal{L}(vocab(\Sigma))$ we can build a DFA $D_\varphi = (Q_\varphi, \Sigma, \delta_\varphi, q_1, F_\varphi)$, satisfying the following properties:*

(1) *The states $q_1, q_2, \ldots q_n$ of D_φ are first-order definable as formulas $q_1^1, q_2^1, \ldots q_n^1$, where intuitively $q_i(x)$ will mean that D_φ is in state q_i after reading symbols at word positions $0, 1, \ldots, x$.*

(2) *The transition function δ_φ of D_φ is captured by the first-order definitions of the states. That is, for all $i \leq n$, $\Gamma \cup \mathcal{A}_{\Sigma w}$ semantically implies the following two formulas for every state q_i:*

(a) $q_i(0) \quad \leftrightarrow \bigvee_{\sigma \in \Sigma, \delta_\varphi(q_1,\sigma)=q_i} P_\sigma(0).$

(b) $\forall u, v\,.\,s(u,v) \rightarrow \Big(q_i(v) \quad \leftrightarrow \bigvee_{\sigma \in \Sigma, \delta_\varphi(q_j,\sigma)=q_i} (P_\sigma(v) \wedge q_j(u))\Big).$

(3) $\Gamma \cup \{\mathcal{A}_{\Sigma w}\} \models \varphi \leftrightarrow F(max), \quad$ *where* $F(u) \equiv \bigvee_{q_i \in F_\varphi} q_i(u).$

Proof. We prove properties 1, 2, and 3 while constructing D_φ and the first-order definitions of its states by induction on the length of φ. The reward is that we get a generalized form of the McNaughton-Papert [MP71] construction that works on non-standard models.

Some subformulas of φ may have free variables, e.g., x, y. In the inductive step considering such subformulas, we expand the vocabulary of the automaton to $\Sigma' = \{x, \epsilon\} \times \{y, \epsilon\} \times \Sigma$. We write $P_\sigma(u) \wedge (x = u) \wedge (y \neq u)$ to mean that at position u, symbol σ occurs, as does x, but not y.

Note: Since every structure gives a unique value to each variable, x, we are only interested in strings in which x occurs at exactly one position.

For the following induction, let $\mathcal{B}$ be any model of $\Gamma \cup \{\mathcal{A}_{\Sigma w}\}$. For the intermediate stages of induction where some variables may occur freely, we assume that $\mathcal{B}$ interprets these free variables. We prove that the formulas of properties 2 and 3 must hold in $\mathcal{B}$ at each step of the induction.

Base cases: φ is either $P_\sigma(x)$, $x = y$, $s(x,y)$, or $s_{\mathrm{tc}}(x,y)$.

$\varphi = P_\sigma(x)$: The automaton for $P_\sigma(x)$ and its state definitions are shown in Fig 1.

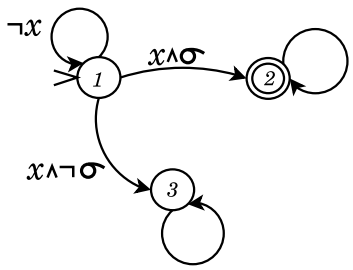

Figure 1: $D_{P_\sigma(x)}$

State predicate	Definition
$q_1(v)$	$\neg s_{\mathrm{tc}}(x,v)$
$q_2(v)$	$s_{\mathrm{tc}}(x,v) \wedge P_\sigma(x)$
$q_3(v)$	$s_{\mathrm{tc}}(x,v) \wedge \neg P_\sigma(x)$

Table 1: $D_{P_\sigma(x)}$

Properties 2 and 3 can be verified as follows:

For property 2b, suppose that $\mathcal{B} \models s(u, v)$. We must show that $\mathcal{B} \models q_2(v)$ iff one of two rules leading to state q_2 holds. These two rules correspond to the edge from q_1 (if $x = v$), and the self loop on q_2 (if $x \neq v$). Suppose $\mathcal{B} \models q_2(v) \wedge (v = x)$. Expanding the definition of q_2, we get $\mathcal{B} \models s_{\mathrm{tc}}(x, v) \wedge P_\sigma(x) \wedge (v = x)$. But this means $\mathcal{B} \models \neg s_{\mathrm{tc}}(x, u)$ since $\mathcal{B} \models \Gamma \cup \{\mathcal{A}_{\Sigma w}\}$ and we have acyclicity. Therefore, we have $\mathcal{B} \models q_1(u)$ by definition of q_1, and we get the desired conclusion, $\mathcal{B} \models q_1(u) \wedge P_\sigma(v)$.

The case corresponding to $x \neq v$ is also easy, and relies on the fact that $\mathcal{B} \models s_{\mathrm{tc}}(x, v) \wedge s(u, v) \wedge (x \neq v) \rightarrow s_{\mathrm{tc}}(x, u)$. In other words, if $q_2(v)$ holds and $x \neq v$, then q_2 holds at v's predecessor too.

This proves one direction of property 2b for state q_2. The other direction for q_2, and the proofs for other states proceed similarly. The proof for 2a is similar.

For property 3, we need to show that $\mathcal{B} \models P_\sigma(x) \leftrightarrow q_2(max)$. This can be verified easily from the definition of q_2.

$\varphi = (x = y)$ or $s(x, y)$: The automata and their state definitions for $\varphi = (x = y)$ and $\varphi = s(x, y)$ are shown in Figs 2 and 3. Properties 2 and 3 can be verified easily for these definitions.

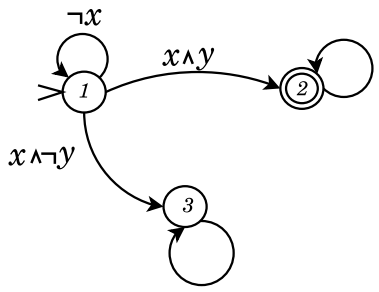

Figure 2: $D_{x=y}$

State predicate	Definition
$q_1(v)$	$\neg s_{\mathrm{tc}}(x, v)$
$q_2(v)$	$(x = y) \wedge s_{\mathrm{tc}}(x, v)$
$q_3(v)$	$(x \neq y) \wedge s_{\mathrm{tc}}(x, v)$

Table 2: $D_{x=y}$

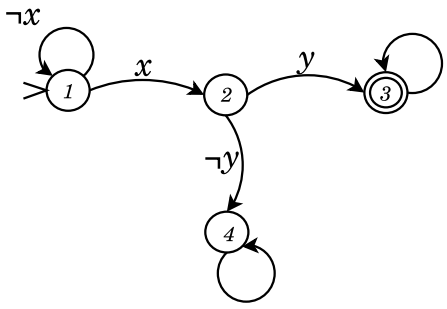

Figure 3: $D_{s(x,y)}$

State predicate	Definition
$q_1(v)$	$\neg s_{\mathrm{tc}}(x, v)$
$q_2(v)$	$x = v$
$q_3(v)$	$s(x, y) \wedge s_{\mathrm{tc}}(y, v)$
$q_4(v)$	$s_{\mathrm{tc}}(x, v) \wedge (x \neq v) \wedge$
	$\neg s(x, y)$

Table 3: $D_{s(x,y)}$

$\varphi = s_{\mathrm{tc}}(x, y)$: The automaton for $\varphi = s_{\mathrm{tc}}(x, y)$, and its state definitions are shown in Fig 4.

We provide a sketch of the proof of property 2b for state q_3. Proofs for other states follow using similar arguments. Suppose $\mathcal{B} \models q_3(v) \wedge s(u, v)$. Expanding the definition of $q_3(v)$, we get $\mathcal{B} \models s_{\mathrm{tc}}(x, y) \wedge s_{\mathrm{tc}}(y, v) \wedge s(u, v)$.

There are two possibilities: $v \neq y$ and $v = y$, corresponding to the loop on state q_3, and the incoming edges from q_2 or q_1. Suppose $v = y$. Now we have two further cases, $x = y$ and $x \neq y$.

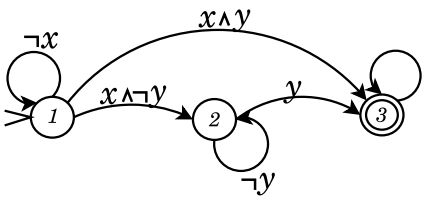

Figure 4: $D_{s_{\mathrm{tc}}(x,y)}$

State predicate	Definition
$q_1(v)$	$\neg s_{\mathrm{tc}}(x,v)$
$q_2(v)$	$s_{\mathrm{tc}}(x,v) \wedge$ $\neg(s_{\mathrm{tc}}(x,y) \wedge s_{\mathrm{tc}}(y,v))$
$q_3(v)$	$s_{\mathrm{tc}}(x,y) \wedge s_{\mathrm{tc}}(y,v)$

Table 4: $D_{s_{\mathrm{tc}}(x,y)}$

If $x = y = v$, we get $\mathcal{B} \models \neg s_{\mathrm{tc}}(x,u)$, or $\mathcal{B} \models q_1(u) \wedge s(u,x) \wedge (x = y = v)$, denoting the appropriate transition from state q_1.

On the other hand, if $\mathcal{B} \models (x \neq y)$, we need to show that q_3 was reached via q_2. Expanding the definition of $q_3(v)$ we have $\mathcal{B} \models s_{\mathrm{tc}}(x,y) \wedge s_{\mathrm{tc}}(y,v)$. Since $y = v$, we get $\mathcal{B} \models s_{\mathrm{tc}}(x,u) \wedge s(u,y)$. But by definition of q_2, this means $\mathcal{B} \models q_2(u)$. Thus, we have $\mathcal{B} \models q_2(u) \wedge s(u,v) \wedge v = y$, the appropriate transition rule for moving from state q_2 to q_3.

For this direction of property 2b, the only remaining case is $y \neq v$. In this case, it is easy to prove that we entered state q_3 at y, and looped thereafter using the appropriate transition for the loop.

For the reverse direction, we need to prove that if a transition rule is applicable at a position then the corresponding next state must hold at the next position. This is easily verified using the state-definitions. Property 2 for other states follows by similar arguments. Property 3 can also be verified easily using the definition of q_3.

Inductive steps: φ is either $\varphi_1 \wedge \varphi_2$, or $\neg\psi$, or $\exists x \,.\, \psi(x)$.

$\varphi = \varphi_1 \wedge \varphi_2$: Inductively we have D_{φ_1} and D_{φ_2} with final state definitions q_{f_1} and q_{f_2} respectively. To construct D_φ, we perform the product construction: let q_i be state definitions of D_{φ_1} and q'_i those of D_{φ_2}. Then the state definitions of D_φ are $q_{\langle i,j\rangle}$, and we have $q_{\langle i,j\rangle}(u) \equiv q_i(u) \wedge q'_j(u)$. The accepting states are

$$F_{\varphi_1 \wedge \varphi_2}(u) \quad \equiv \bigvee_{f_1 \in F_1 \wedge f_2 \in F_2} q_{\langle f_1, f_2\rangle}(u).$$

Property 1 holds because we are still in first-order. Property 2 follows because we are just performing logical transliterations of the standard DFA conjunction operation. Property 3 follows from the fact that we already have $\mathcal{B} \models F_1(max) \leftrightarrow \varphi_1$ and $\mathcal{B} \models F_2(max) \leftrightarrow \varphi_2$, and from the definition of $F_{\varphi_1 \wedge \varphi_2}$.

$\varphi = \neg\psi$: In this case, we take the complement of D_ψ which is easy because our automata are deterministic. Let the final state of D_ψ be F'. D_φ has the same state definitions as ψ, but its final state definition is $F(u) \equiv \neg F'(u)$. It is easy to see that properties 1, 2 and 3 hold in this case.

$\varphi = \exists x \,.\, \psi(x)$:

Inductively we have $D_\psi = (\{q_1, \ldots, q_n\}, \Sigma \times \{x, \epsilon\}, \delta_\psi, q_1, F_\psi)$.

First we transform D_ψ to an NFA $\mathcal{N}_\varphi = (\{p_1, \ldots, p_n, p'_1, \ldots, p'_n\}, \Sigma, \delta, p_1, F)$, where $F = \{p'_i | q_i \in F_\psi\}$ and $\delta(p_i, \sigma) = \{p_j, p'_k | \delta_\psi(q_i, \sigma \wedge \neg x) = q_j, \delta_\psi(q_i, \sigma \wedge x) = q_k\}$.

Thus $\mathcal{N}_\varphi$ no longer sees x's. Instead, it guesses the one place that x might occur, and that is where the transition from p_i to p'_i occurs. (See Fig. 5)

Let $p_i(u) \equiv \exists x \,.\, \neg s_{\mathrm{tc}}(x,u) \wedge q_i(u)$; $\quad p'_i(u) \equiv \exists x \,.\, s_{\mathrm{tc}}(x,u) \wedge q_i(u)$.

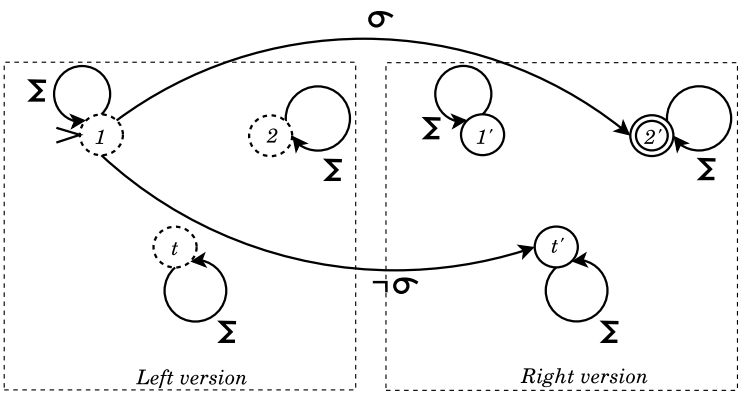

Figure 5: $\mathcal{N}_{\exists x\,.\,P_\sigma(x)}$

Define D_φ to be the DFA equivalent to $\mathcal{N}_\varphi$ using the subset construction. Let $S_0 = \{p_{i_0}, p'_j | j \in J_0\}$, $S_1 = \{p_{i_1}, p'_j | j \in J_1\}$ be two states of D_φ. (Note that each reachable state of D_φ has exactly one element of $\{p_1, \ldots, p_n\}$.)

Observe that in a "run" of $\mathcal{N}_\varphi$ on $\mathcal{B}$, we can be in state p_i at position u iff $\mathcal{B} \models p_i(u)$ and we can be in state p'_i of u iff $\mathcal{B} \models p'_i(u)$. Thus, the first-order formula capturing state S_0 is

$$S_0(u) \quad \equiv \quad p_{i_0} \ \wedge \bigwedge_{j \in J_0} p'_j(u) \ \wedge \bigwedge_{j \notin J_0} \neg p'_j(u)$$

Conditions 2 and 3 for D_φ thus follow by these conditions for D_ψ, which hold by inductive assumption.

For example, if $\delta_\varphi(S_0, \sigma) = S_1$, then $\delta_\psi(p_{i_0}, \sigma \wedge \neg x) = p_{i_1}$, and $j \in J_1$ iff $\delta_\psi(q_{i_0}, \sigma \wedge x) = q_j$ or $\delta_\psi(q_{j_0}, \sigma \wedge \neg x) = q_j$ for some $j_0 \in J_0$.

Thus, we have inductively constructed the D_φ and proved that it satisfies properties 1, 2, and 3. □

Lemma 4.9 tells us that for any model $\mathcal{B}$ of $\Gamma \cup \{A_{\Sigma w}\}$, $\mathcal{B} \models \varphi$ iff $\mathcal{B} \models F_\varphi(max)$. In other words, $\mathcal{B} \models \varphi$ iff $\mathcal{B}$ "believes" that there is a path from the start state to some q_f in F_φ. As a part of the next lemma, we use induction to prove that this implies that there actually must be a path in D_φ from the start state to some q_f in F_φ.

Lemma 4.10. *Suppose $\mathcal{B} \models \Gamma \cup \{A_{\Sigma w}\} \cup \{\varphi\}$. Then, there exists a word, w_0, such that its corresponding word model, $\mathcal{B}_0$, satisfies φ.*

Proof. By Lemma 4.9, we can construct D_φ, and we have $\mathcal{B} \models F_\varphi(max)$. So $\mathcal{B}$ "believes" that there is a path to some $q_f \in F_\varphi$. Suppose there is no such path in D_φ. Let C denote the disjunction of all states that are truly reachable from the start state in D_φ. This situation can be expressed as follows: $\forall u, v\,.\,C(u) \wedge s(u, v) \rightarrow C(v)$. But this is exactly the premise for the axiom scheme **NoExit**, which must hold since $\mathcal{B} \models \Gamma$. Therefore, we have $\mathcal{B} \models \forall u, v\,.\,C(v) \wedge s_{\mathrm{tc}}(u, v) \rightarrow C(v)$. This implies some accepting state q_f should be in C, because $\mathcal{B} \models \forall u\,.\,s_{\mathrm{tc}}(u, max) \wedge F_\varphi(max)$, and we get a contradiction.

Therefore, there has to be a real path from the start state to a final state q_f in D_φ. This implies that the DFA D_φ accepts some standard word, w_0. Let $\mathcal{B}_0$ be the word model corresponding to w_0. Thus $\mathcal{B}_0 \models F_\varphi(max)$, and therefore by Lemma 4.9 $\mathcal{B}_0 \models \varphi$ as desired. □

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

Figure 6: A simple Java-like implementation of the in-place reversal of a singly linked list.

5. Heuristics for Using the Coloring Axioms

This section presents heuristics for using the coloring axioms. Toward that end, it answers the following questions:

- How can the coloring axioms be used by a theorem prover to prove χ? (Section 5.2)
- When should a specific instance of a coloring axiom be given to the theorem prover while trying to prove χ? (Section 5.4)
- What part of the process can be automated? (Section 5.5)

We first present a running example (more examples are described in Section 5.6 and used in later sections to illustrate the heuristics). We then explain how the coloring axioms are useful, describe the search space for useful axioms, give an algorithm for exploring this space, and conclude by discussing a prototype implementation we have developed that proves the example presented and others.

5.1. **Reverse Specification.** The heuristics described in Sections 5.2–5.4 are illustrated on problems that arise in the verification of partial correctness of a list reversal procedure. Other examples proven using this technique can be found in Section 5.6.

The procedure reverse, shown in Fig. 6, performs in-place reversal of a singly linked list, destructively updating the list. The precondition requires that the input list be acyclic and unshared (i.e., each heap node is pointed to by at most one heap node). For simplicity, we assume that there is no garbage. The postcondition ensures that the resulting list is acyclic and unshared. Also, it ensures that the nodes reachable from the formal parameter on entry to reverse are exactly the nodes reachable from the return value of reverse at the exit. Most importantly, it ensures that each edge in the original list is reversed in the returned list.

The specification for reverse is shown in Fig. 7. We use unary predicates to represent program variables and binary predicates to represent data-structure fields. Fig. 7(a) defines some shorthands. To specify that a unary predicate z can point to a single node at a time and that a binary predicate f of a node can point to at most one node (i.e., f is a partial function), we use $unique[z]$ and $func[f]$. To specify that there are no cycles of f-fields in the graph, we use $acyclic[f]$. To specify that the graph does not contain nodes shared by f-fields, (i.e., nodes with 2 or more incoming f-fields), we use $unshared[f]$. To specify that all nodes in the graph are reachable from z_1 or z_2 by following f-fields, we use $total[z_1, z_2, f]$. Another helpful shorthand is $r_{x,f}(v)$ which specifies that v is reachable from the node pointed to by x using f-edges.

The precondition of the reverse procedure is shown in Fig. 7(b). We use the predicates xe and ne to record the values of the variable x and the next field at the beginning of the procedure. The precondition requires that the list pointed to by x be acyclic and unshared. It also requires that $unique[z]$ and $func[f]$ hold for all unary predicates z that represent program variables and all binary predicates f that represent fields, respectively. For simplicity, we assume that there is no garbage, i.e., all nodes are reachable from x.

The post-condition is shown in Fig. 7(c). It ensures that the resulting list is acyclic and unshared. Also, it ensures that the nodes reachable from the formal parameter x on entry to the procedure are exactly the nodes reachable from the return value y at the exit. Most importantly, we wish to show that each edge in the original list is reversed in the returned list (see Eq. (5.9)).

A loop invariant is given in Fig. 7(d). It describes the state of the program at the beginning of each loop iteration. Every node is in one of two disjoint lists pointed to by x and y (Eq. (5.10)). The lists are acyclic and unshared. Every edge in the list pointed to by x is exactly an edge in the original list (Eq. (5.12)). Every edge in the list pointed to by y is the reverse of an edge in the original list (Eq. (5.13)). The only original edge going out of y is to x (Eq. (5.14)).

The transformer is given in Fig. 7(e), using the primed predicates n', x', and y' to describe the values of predicates n, x, and y, respectively, at the end of the iteration.

5.2. **Proving Formulas using the Coloring Axioms.** All the coloring axioms have the form $A \equiv P_A \rightarrow C_A$, where P_A and C_A are closed formulas. We call P_A the axiom's premise and C_A the axiom's conclusion. For an axiom to be useful, the theorem prover will have to prove the premise (as a subgoal) and then use the conclusion in the proof of the goal formula χ. For each of the coloring axioms, we now explain when the premise can be proved, how its conclusion can help, and give an example.

NoExit. The premise $P_{\textbf{NoExit}}[C, f]$ states that there are no f-edges exiting color class C. When C is a unary predicate appearing in the program, the premise is sometimes a direct result of the loop invariant. Another color that will be used heavily throughout this section is reachability from a unary predicate, i.e., unary reachability, formally defined in Eq. (5.6). Let us examine two cases. $P_{\textbf{NoExit}}[r_{x,f}, f]$ is immediate from the definition of $r_{x,f}$ and the transitivity of f_{tc}. $P_{\textbf{NoExit}}[r_{x,f}, f']$ actually states that there is no f-path from x to an edge for which f' holds but f does not, i.e., a change in f' with respect to f. Thus, we use the absence of f-paths to prove the absence of f'-paths. In many cases, the change is an important part of the loop invariant, and paths from and to it are part of the specification.

A sketch of the proof by refutation of $P_{\textbf{NoExit}}[r_{x',n}, n']$ that arises in the reverse example is given in Fig. 8. The numbers in brackets are the stages of the proof.

(1) The negation of the premise expands to:

$$\exists u_1, u_2, u_3 \,.\, x'(u_1) \wedge n_{\mathrm{tc}}(u_1, u_2) \wedge \neg n_{\mathrm{tc}}(u_1, u_3) \wedge n'(u_2, u_3)$$

(2) Since u_2 is reachable from u_1 and u_3 is not, by T_2, we have $\neg n(u_2, u_3)$.
(3) By the definition of n' in the transformer, the only edge in which n differs from n' is out of x (one of the clauses generated from Eq. (5.15) is $\forall v_1, v_2 \,.\, \neg n'(v_1, v_2) \vee n(v_1, v_2) \vee x(v_1)$) . Thus, $x(u_2)$ holds.
(4) By the definition of x' it has an incoming n edge from x. Thus, $n(u_2, u_1)$ holds.

The list pointed to by x must be acyclic, whereas we have a cycle between u_1 and u_2; i.e., we have a contradiction. Thus, $P_{\textbf{NoExit}}[r_{x',n}, n']$ must hold.

$C_{\textbf{NoExit}}[C, f]$ states there are no f paths (f_{tc} edges) exiting C. This is useful because proving the absence of paths is the difficult part of proving formulas with TC.

(a)

$$unique[z] \stackrel{\text{def}}{=} \forall v_1, v_2 . z(v_1) \wedge z(v_2) \rightarrow v_1 = v_2 \quad (5.1)$$

$$func[f] \stackrel{\text{def}}{=} \forall v_1, v_2, v . f(v, v_1) \wedge f(v, v_2) \rightarrow v_1 = v_2 \quad (5.2)$$

$$acyclic[f] \stackrel{\text{def}}{=} \forall v_1, v_2 . \neg f(v_1, v_2) \vee \neg \mathrm{TC}[f](v_2, v_1) \quad (5.3)$$

$$unshared[f] \stackrel{\text{def}}{=} \forall v_1, v_2, v . f(v_1, v) \wedge f(v_2, v) \rightarrow v_1 = v_2 \quad (5.4)$$

$$total[z_1, z_2, f] \stackrel{\text{def}}{=} \forall v . \exists w . (z_1(w) \vee z_2(w)) \wedge \mathrm{TC}[f](w, v) \quad (5.5)$$

$$r_{x,f}(v) \stackrel{\text{def}}{=} \exists w \, . \, x(w) \wedge \mathrm{TC}[f](w, v) \quad (5.6)$$

$$r_{x,\overleftarrow{f}}(v) \stackrel{\text{def}}{=} \exists w \, . \, x(w) \wedge \mathrm{TC}[f](v, w) \quad (5.7)$$

(b)

$$pre \stackrel{\text{def}}{=} total[xe, xe, ne] \wedge acyclic[ne] \wedge unshared[ne] \wedge unique[xe] \wedge func[ne] \quad (5.8)$$

(c)

$$post \stackrel{\text{def}}{=} total[y, y, n] \wedge acyclic[n] \wedge unshared[n] \wedge \forall v_1, v_2 . ne(v_1, v_2) \leftrightarrow n(v_2, v_1) \quad (5.9)$$

(d)

$$LI[x, y, n] \stackrel{\text{def}}{=} total[x, y, n] \wedge \forall v . (\neg r_{x,n}(v) \vee \neg r_{y,n}(v)) \wedge acyclic[n] \wedge unshared[n] \quad (5.10)$$

$$unique[x] \wedge unique[y] \wedge func[n] \wedge \quad (5.11)$$

$$\forall v_1, v_2 . (r_{x,n}(v_1) \rightarrow (ne(v_1, v_2) \leftrightarrow n(v_1, v_2))) \wedge \quad (5.12)$$

$$\forall v_1, v_2 . (r_{y,n}(v_2) \wedge \neg y(v_1) \rightarrow (ne(v_1, v_2) \leftrightarrow n(v_2, v_1))) \wedge \quad (5.13)$$

$$\forall v_1, v_2, v . y(v_1) \rightarrow (x(v_2) \leftrightarrow ne(v_1, v_2)) \quad (5.14)$$

(e)

$$T \stackrel{\text{def}}{=} \forall v . (y'(v) \leftrightarrow x(v)) \wedge \forall v . (x'(v) \leftrightarrow \exists w . x(w) \wedge n(w, v)) \wedge \forall v_1, v_2 . n'(v_1, v_2) \leftrightarrow ((n(v_1, v_2) \wedge \neg x(v_1)) \vee (x(v_1) \wedge y(v_2))) \quad (5.15)$$

Figure 7: Example specification of reverse procedure: (a) shorthands, (b) precondition pre, (c) postcondition $post$, (d) loop invariant $LI[x, y, n]$, (e) transformer T (effect of the loop body).

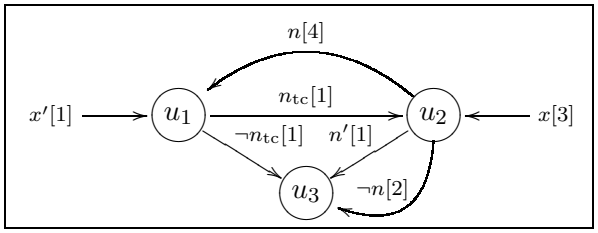

Figure 8: Proving $P_{\textbf{NoExit}}[r_{x,n}, n']$.

GoOut. The premise $P_{\textbf{GoOut}}[A, B, f]$ states that all f edges going out of color class A, go to B. When A and B are unary predicates that appear in the program, again the premise sometimes holds as a direct result of the loop invariant. An interesting special case is when B is defined as

$\exists w\,.\,A(w) \wedge f(w,v)$. In this case the premise is immediate. Note that in this case the conclusion is provable also from T_1. However, from experience, the axiom is very useful for improving performance (2 orders of magnitude when proving the acyclic part of reverse's postcondition).

$C_{\textbf{GoOut}}[A,B,f]$ states that all paths out of A must pass through B. Thus, under the premise $P_{\textbf{GoOut}}[A,B,f]$, if we know that there is a path from A to somewhere outside of A, we know that there is a path to there from B. In case all nodes in B are reachable from all nodes in A, together with the transitivity of f_{tc} this means that the nodes reachable from B are exactly the nodes outside of A that are reachable from A.

For example, $C_{\textbf{GoOut}}[y',y,n']$ allows us to prove that only the original list pointed to by y is reachable from y' (in addition to y' itself).

NewStart. The premise $P_{\textbf{NewStart}}[C,g,h]$ states that all g edges between nodes in C are also h edges. This can mean the iteration has not added edges or has not removed edges according to the selection of h and g. In some cases, the premise holds as a direct result of the definition of C and the loop invariant.

$C_{\textbf{NewStart}}[C,g,h]$ means that every g path that is not an h path must pass outside of C. Together with $C_{\textbf{NoExit}}[C,g]$, it proves there are no new paths within C.

For example, in reverse the **NewStart** scheme can be used as follows. No outgoing edges were added to nodes reachable from y. There are no n or n' edges from nodes reachable from y to nodes not reachable from y. Thus, no paths were added between nodes reachable from y. Since the list pointed to by y is acyclic before the loop body, we can prove that it is acyclic at the end of the loop body.

We can see that **NewStart** allows the theorem prover to reason about paths within a color, and the other axioms allow the theorem prover to reason about paths between colors. Together, given enough colors, the theorem prover can often prove all the facts that it needs about paths and thus prove the formula of interest.

5.3. **The Search Space of Possible Axioms.** To answer the question of when we should use a specific instance of a coloring axiom when attempting to prove the target formula, we first define the search space in which we are looking for such instances. The axioms can be instantiated with the colors defined by an arbitrary unary formula (one free variable) and one or two binary predicates. First, we limit ourselves to binary predicates for which TC was used in the target formula. Now, since it is infeasible to consider all arbitrary unary formulas, we start limiting the set of colors we consider.

The initial set of colors to consider are unary predicates that occur in the formula we want to prove. Interestingly enough, these colors are enough to prove that the postcondition of mark and sweep is implied by the loop invariant, because the only axiom we need is **NoExit**$[marked,f]$.

An immediate extension that is very effective is forward and backward reachability from unary predicates, as defined in Eq. (5.6) and Eq. (5.7), respectively. Instantiating all possible axioms from the unary predicates appearing in the formula and their unary forward reachability predicates, allows us to prove reverse. For a list of the axioms needed to prove reverse, see Fig. 9. Other examples are presented in Section 5.6. Finally, we consider Boolean combinations of the above colors. Though not used in the examples shown in this paper, this is needed, for example, in the presence of sharing or when splicing two lists together.

All the colors above are based on the unary predicates that appear in the original formula. To prove the reverse example, we needed x' as part of the initial colors. Table 5 gives a heuristic for finding the initial colors we need in cases when they cannot be deduced from the formula, and how it applies to reverse.

NoExit$[r_{x',n}, n']$	**GoOut**$[x, x', n]$	**NewStart**$[r_{x',n},\ n, n']$	**NewStart**$[r_{x',n}, n', n]$
NoExit$[r_{x',n'}, n]$	**GoOut**$[x, y, n']$	**NewStart**$[r_{x',n'}, n, n']$	**NewStart**$[r_{x',n'}, n', n]$
NoExit$[r_{y,n}, n']$		**NewStart**$[r_{y,n},\ n, n']$	**NewStart**$[r_{y,n}, n', n]$
NoExit$[r_{y,n'}, n]$		**NewStart**$[r_{y,n'}, n, n']$	**NewStart**$[r_{y,n'}, n', n]$

Figure 9: The instances of coloring axioms used in proving reverse.

Group	**Criteria**
Roots[f]	All changes are reachable from one of the colors using f_{tc}
StartChange[f,g]	All edges for which f and g differ start from a node in these colors
EndChange[f,g]	All edges for which f and g differ end at a node in these colors

(a)

Group	**Colors**
$Roots[n]$	$x(v)$, $y(v)$
$Roots[n']$	$x'(v)$, $y'(v)$
$StartChange[n, n']$	$x(v)$
$EndChange[n, n']$	$y(v)$, $x'(v)$

(b)

Table 5: (a) Heuristic for choosing initial colors. (b) Results of applying the heuristic on reverse.

An interesting observation is that the initial colors we need can, in many cases, be deduced from the program code. As in the previous section, we have a good way for deducing paths between colors and within colors in which the edges have not changed. The program usually manipulates fields using pointers, and can traverse an edge only in one direction. Thus, the unary predicates that represent the program variables (including the temporary variables) are in many cases what we need as initial colors.

5.4. **Exploring the Search Space.** When trying to automate the process of choosing colors, the problem is that the set of possible colors to choose from is doubly-exponential in the number of initial colors; giving all the axioms directly to the theorem prover is infeasible. In this section, we define a heuristic algorithm for exploring a limited number of axioms in a directed way. Pseudocode for this algorithm is shown in Fig. 10. The operator $\vdash$ is implemented as a call to a theorem prover.

Because the coloring axioms have the form $A \equiv P_A \rightarrow C_A$, the theorem prover must prove P_A or the axiom is of no use. Therefore, the pseudocode works iteratively, trying to prove P_A from the current $\psi \wedge \Sigma$, and if successful it adds C_A to Σ.

The algorithm tries colors in increasing levels of complexity. $BC(i, C)$ gives all the Boolean combinations of the predicates in C up to size i. After each iteration we try to prove the goal formula. Sometimes we need the conclusion of one axiom to prove the premise of another. The **NoExit** axioms are particularly useful for proving $P_{\textbf{NewStart}}$. Therefore, we need a way to order instantiations so that axioms useful for proving the premises of other axioms are acquired first. The ordering we chose is based on phases: First, try to instantiate axioms from the axiom scheme **GoOut**. Second, try to instantiate axioms from the axiom scheme **NoExit**. Finally, try to instantiate axioms from the axiom scheme **NewStart**. For **NewStart**$[c, f, g]$ to be useful, we need to be able

```
explore(Init, χ) {
   Let χ = ψ → φ
   Σ := {Trans[f], Order[f] | f ∈ F}
   Σ := Σ ∪ {T1[f], T2[f] | f ∈ F}
   C := {r_{c,f}(v) | c ∈ Init, f ∈ F}
   C := C ∪ Init
   i := 1
   forever {
      C' := BC(i, C)
      // Phase 1
      foreach f ∈ F, c_s ≠ c_e ∈ C'
        if Σ ∧ ψ ⊢ P_GoOut[c_s, c_e, f]
          Σ := Σ ∪ {C_GoOut[c_s, c_e, f]}
      // Phase 2
      foreach f ∈ F, c ∈ C'
        if Σ ∧ ψ ⊢ P_NoExit[c, f]
          Σ := Σ ∪ {C_NoExit[c, f]}
      // Phase 3
      foreach C_NoExit[c, f] ∈ Σ, g ≠ f ∈ F
        if Σ ∧ ψ ⊢ P_NewStart[c, f, g]
          Σ := Σ ∪ {C_NewStart[c, f, g]}
      if Σ ∧ ψ ⊢ φ
         return SUCCESS
      i := i + 1
   }
}
```

Figure 10: An iterative algorithm for instantiating the axiom schemes. Each iteration consists of three phases that augment the axiom set Σ

to show that there are either no incoming f-paths or no outgoing f-paths from c. Thus, we only try to instantiate such an axiom when either $P_{\textbf{NoExit}}[c, f]$ or $P_{\textbf{NoExit}}[\neg c, f]$ has been proven.

5.5. **Implementation.** The algorithm presented here was implemented using a `Perl` script and the SPASS theorem prover [WGR96] and used successfully to verify the example programs of Section 5.1 and Section 5.6.

The method described above can be optimized. For instance, if C_A has already been added to the axioms, we do not try to prove P_A again. These details are important in practice, but have been omitted for brevity.

When trying to prove the different premises, SPASS may fail to terminate if the formula that it is trying to prove is invalid. Thus, we limit the time that SPASS can spend proving each formula. It is possible that we will fail to acquire useful axioms this way.

5.6. **Further Examples.** This section shows the code (Fig. 11) and the complete specification of two additional examples: appending two linked lists, and the mark phase of a simple mark and sweep garbage collector.

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

(a)

```
void mark(NodeSet 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);
         }
      }
  }
```

(b)

Figure 11: A simple Java-like implementation of (a) the concatenation procedure for two singly-linked lists; (b) the mark phase of a mark-and-sweep garbage collector.

5.6.1. *Specification of append.* The specification of append (see Fig. 11(a)) is given in Fig. 12. The specification includes procedure's pre-condition, a transformer of the procedure's body effect, and the procedure's post-condition. The pre-condition (Fig. 12(a)) states that the lists pointed to by x and y are acyclic, unshared and disjoint. It also states there is no garbage. The post condition (Fig. 12(b)) states that after the procedure's execution, the list pointed to by x' is exactly the union of the lists pointed to by x and y. Also, the list is still acyclic and unshared. The transformer is given in Fig. 12(c). The result of the loop in the procedure's body is summarized as a formula defining the $last$ variable. The only change to n is the addition of an edge between $last$ and y.

The coloring axioms needed to prove append are given in Fig. 13.

(a)

$$pre \stackrel{\text{def}}{=} acyclic[n] \wedge unshared[n] \wedge unique[x] \wedge unique[y] \wedge func[n] \wedge (\forall v.\neg r_{x,n}(v) \vee \neg r_{y,n}(v)) \wedge \forall v.r_{x,n}(v) \vee r_{y,n}(v) \quad (5.16)$$

(b)

$$post \stackrel{\text{def}}{=} acyclic[n'] \wedge unshared[n'] \wedge unique[x'] \wedge unique[last] \wedge func[n'] \wedge (\forall v\,.\,r_{x',n'}(v) \;\leftrightarrow\; (r_{x,n}(v) \vee r_{y,n}(v))) \wedge \forall v_1, v_2\,.\,n'(v_1, v_2) \leftrightarrow n(v_1, v_2) \vee (last(v_1) \wedge y(v_2)) \quad (5.17)$$

(c) T is the conjunction of the following formulas:

$$\forall v.x'(v) \;\leftrightarrow\; x(v) \quad (5.18)$$

$$\forall v.last(v) \;\leftrightarrow\; r_{x,n}(v) \wedge \forall u.\neg n(v, u) \quad (5.19)$$

$$\exists v.\; last(v) \quad (5.20)$$

$$\forall v_1, v_2.n'(v_1, v_2) \;\leftrightarrow\; n(v_1, v_2) \vee (last(v_1) \wedge y(v_2)) \quad (5.21)$$

Figure 12: Example specification of append procedure: (a) precondition pre, (b) postcondition $post$, (c) transformer T (effect of the procedure body).

NoExit$[r_{y,n}, n']$	**GoOut**$[last, y, n']$
NewStart$[r_{x,n}, n, n']$	**NewStart**$[r_{x,n}, n', n]$
NewStart$[r_{y,n}, n, n']$	**NewStart**$[r_{y,n}, n', n]$

Figure 13: The instances of coloring axioms used in proving append.

5.6.2. *Specification of the mark phase.* Another example proven is the mark phase of a mark-and-sweep sequential garbage collector, shown in Fig. 11(b). The example goes beyond the reverse example in that it manipulates a general graph and not just a linked list. Furthermore, as far as we know, ESC/Java [FLL+02] was not able prove its correctness because it could not show that unreachable elements were not marked. Note that the axiom needed to prove this property is **NoExit**, which we have shown to be beyond the power of Nelson's axiomatization.

The loop invariant of `mark` is given in Fig. 14(a). The first disjunct of the formula holds only in the first iteration, when only the nodes in root are pending and nothing is marked. The second holds from the second iteration on. Here, the nodes in root are marked or pending (they start as pending, and the only way to stop being pending is to become marked). No node is both marked and pending (because the procedure checks if the node is marked before adding it to pending). All nodes that are marked or pending are reachable from the root set (we start with only the root nodes as pending, and after that only nodes that are neighbors of pending nodes became pending; furthermore, only pending nodes may become marked). There are no edges between marked nodes and nodes that are neither marked nor pending (because when we mark a node we add all its neighbors to pending, unless they are marked already). Our method succeeded in proving the loop invariant in Fig. 14(a) using only the positive axioms.

The post-condition of `mark` is given in Fig. 14(b). To prove it, we had to use the fact that there are no edges between marked and unmarked nodes (i.e, there are no pending nodes at the end

(a)

$$
\begin{array}{rcll}
((\forall v\,.\,root(v) & \leftrightarrow & pending(v)) \wedge & (5.22)\\
(\forall v\,. & \neg & marked(v))) & (5.23)\\
 & \vee & & \\
((\forall v\,.\,root(v) & \rightarrow & marked(v) \vee pending(v)) \wedge & (5.24)\\
(\forall v\,.\,\neg pending(v) & \vee & \neg marked(v)) \wedge & (5.25)\\
(\forall v\,.\,pending(v) & \vee & marked(v) \rightarrow r_{root,f}(v)) \wedge & (5.26)\\
(\forall v_1, v_2\,.\,marked(v_1) & \wedge & \neg marked(v_2) \wedge \neg pending(v_2) & \\
 & \rightarrow & \neg f(v_1, v_2))) & (5.27)
\end{array}
$$

(b)

$$\forall v\,.\,marked(v) \leftrightarrow r_{root,f}(v) \qquad (5.28)$$

Figure 14: Example specification of mark procedure: (a) The loop invariant of mark, (b) The post-condition of mark.

of the loop). Thus, we instantiate the axiom **NoExit**$[marked, f]$, and this is enough to prove the post-condition.

6. Applicability of the Coloring Axioms

The coloring axioms are applicable to a wide variety of verification problems. To demonstrate this, we describe the reasoning done by the TVLA system and how it can be simulated using the coloring axioms. TVLA is based on the theory of abstract interpretation [CC79] and specifically on canonical abstraction [SRW02]. TVLA has been successfully used to analyze a large verity of small but intricate heap manipulating programs (see e.g., [LAS00, BLARS07]), including the verification of several algorithms (see e.g., [LARSW00, LRS06]). Furthermore, the axioms described in this paper have been used to integrate SPASS as the reasoning engine behind the TVLA system. The integrated system is used to perform backward analysis on heap manipulating programs as described in [LASR07].

In [SRW02], logical structures are used to represent the concrete stores of the program, and FO(TC) is used to specify the concrete transformers. This provides great flexibility in what programming-language constructs the method can handle. For the purpose of this section, we assume that the vocabulary used is fixed and always contains equality. Furthermore, we assume that the transformer cannot change the universe of the concrete store. Allocation and deallocation can be easily modeled by using a designated unary predicate that holds for the allocated heap cells. Similarly, we assume that the universe of the concrete store is non-empty. Abstract stores are represented as finite 3-valued logical structures. We shall explain the meaning of a structure S by describing the formula $\widehat{\gamma}(S)$ to which it corresponds.

The individuals of a 3-valued logical structure are called abstract nodes. We use an auxiliary unary predicate for each abstract node to capture the concrete nodes that are mapped to it. For an abstract structure with universe $\{node_1, \ldots, node_n\}$, let $\{a_1, \ldots a_n\}$ be the corresponding unary predicates.

For each k-ary predicate p in the vocabulary, each k-tuple $\langle node_1, \ldots, node_k\rangle$ in the abstract structure (called an abstract tuple) can have one of the following truth values $\{0, 1, \frac{1}{2}\}$ as follows:

- The truth value 1 means that the predicate p universally holds for all of the concrete tuples mapped to this abstract tuple, i.e.,
$$\forall v_1, \ldots, v_k \,.\, a_1(v_1) \wedge \ldots \wedge a_k(v_k) \rightarrow p(v_1, \ldots, v_k) \tag{6.1}$$
- The truth value 0 means that the predicate p universally does not hold, for all of the concrete tuples mapped to this abstract tuple, i.e.,
$$\forall v_1, \ldots, v_k \,.\, a_1(v_1) \wedge \ldots \wedge a_k(v_k) \rightarrow \neg p(v_1, \ldots, v_k) \tag{6.2}$$
- The truth value $\frac{1}{2}$ means that we have no information about this abstract tuple, and thus the value of the predicate p is not restricted.

We use a designated set of unary predicates called *abstraction predicates* to control the distinctions among concrete nodes that can be made in an abstract element, which also places a bound on the size of abstract elements. For each abstract node $node_i$, A_i denotes the set of abstraction predicates for which $node_i$ has the truth value 1, and $\overline{A}_i$ denotes the set of abstraction predicates for which $node_i$ has the truth value 0. Every pair $node_i, node_j$ of different abstract nodes either $A_i \cap \overline{A}_j \neq \emptyset$ or $\overline{A}_i \cap A_j \neq \emptyset$. In addition, we require that the abstract nodes in the structure represent all the concrete nodes, i.e., $\forall v \,.\, \bigvee_i a_i(v)$. Thus, the abstract nodes form a bounded partition of the concrete nodes. Finally, each node must represent at least one concrete node, i.e., $\exists v \,.\, a_i(v)$.

The vocabulary may contain additional predicates called *derived predicates*, which are explicitly defined from other predicates using a formula in FO(TC). These derived predicates help the precision of the analysis by recording correlations not captured by the universal information. Some of the unary derived predicates may also be abstraction predicates, and thus can induce finer-granularity abstract nodes.

We say that $S_1 \sqsubseteq S_2$ if there is a total mapping m between the abstract nodes of S_1 and the abstract nodes of S_2 such that S_2 represents all of the concrete stores that S_1 represents when considering each abstract node of S_2 as a union of the abstract nodes of S_1 mapped to it by m. Formally, $\widehat{\gamma}(S_1) \wedge \psi_m \rightarrow \widehat{\gamma}(S_2)$ where

$$\psi_m = \bigwedge_{\substack{node_i \in S_1 \\ m(node_i) = node'_j}} \forall v \,.\, a_i(v) \rightarrow a'_j(v)$$

The order is extended to sets using the induced Hoare order (i.e., $\mathrm{XS}_1 \sqsubseteq \mathrm{XS}_2$ if for each element $S_1 \in \mathrm{XS}_1$ there exists an element $S_2 \in \mathrm{XS}_2$ such that $S_1 \sqsubseteq S_2$).

In the original TVLA implementation [LAS00] the abstract transformer is computed by a three step process:

- First, a heuristic is used to perform case splits by refining the partition induced by the abstraction predicates. This process is called *Focus*.
- Second, the formulas comprising the concrete transformer are used to conservatively approximate the effect of the concrete transformer on all the represented memory states. Update formulas are either handwritten or derived using finite differencing [RSL03].
- Third, a constraint solver called *Coerce* is used to improve the precision of the abstract element by taking advantage of the inter-dependencies between the predicates dictated by the defining formulas of the derived predicates and constraints of the programming language semantics.

Most of the logical reasoning performed by TVLA is first order in nature. The transitive-closure reasoning is comprised of three parts:

(1) The update formulas for derived predicates based on transitive closure use first-order formulas to update the transitive-closure relation, as explained in Section 6.1.

(2) The Coerce procedure relates the definition of the edge relation with its transitive closure by performing *Kleene evaluation* (see below).
(3) Handwritten axioms are given to Coerce to allow additional transitive-closure reasoning. They are usually written once and for all per data-structure analyzed by the system.

To compare the transitive-closure reasoning of TVLA and the coloring axioms presented in this paper, we concentrate on programs that manipulate singly-linked lists and trees, although the basic argument holds for other data-structures analyzed by TVLA as well. The handwritten axioms used by TVLA for these cases are all covered by the axioms described in Section 3.2. The issue of update formulas is covered in detail in Section 6.1. A detailed description of Kleene evaluation is beyond the scope of this paper and can be found in [SRW02]. Kleene evaluation of transitive closure is equivalent to applying transitivity to infer the existence of paths, and finding a subset of the partition that has no outgoing edges to infer the absence of paths. The latter is equivalent to applying the **NoExit** axiom on the formula that defines the appropriate partition.

6.1. **Precise Update.** Maintenance of transitive closure through updates in the underlying relation is required for the verification of heap-manipulating programs. In general, it is not possible to update transitive closure for arbitrary change using first-order-logic formulas. Instead, we limit the discussion to unit changes (i.e., the addition or removal of a single edge). Work in descriptive dynamic complexity [PI97, Hes03] and database theory [DS95] gives first-order update formulas to unit changes in several classes of graphs, including functional graphs and acyclic graphs.

We demonstrate the applicability of the proposed axiom schemes by showing how they can be used to prove the precise update formula for unit changes in several classes of graphs.

6.1.1. *Edge addition.* We refer to the edge relation before the update by e and the edge relation after the update by e'. Adding an edge from s to t can be formulated as

$$\forall v_1, v_2 \,.\, e'(v_1, v_2) \;\leftrightarrow\; (e(v_1, v_2) \vee (s(v_1) \wedge t(v_2))).$$

The precise update formula for this change is

$$\exists v_s, v_t \,.\, s(v_s) \wedge t(v_t) \wedge \forall v_1, v_2 \,.\, e'_{\mathrm{tc}}(v_1, v_2) \;\leftrightarrow\; (e_{\mathrm{tc}}(v_1, v_2) \vee (e_{\mathrm{tc}}(v_1, v_s) \wedge e_{\mathrm{tc}}(v_t, v_2)))$$

We have used SPASS to prove the validity of this update formula using the color axioms described in this paper. The basic colors needed are $r_{t,e}$, i.e., forward reachability from the target of the new edge, and $r_{s,\overleftarrow{e}}$, i.e., backward reachability from the source of the new edge. The axioms instantiated in the proof are given in Table 6(a).

6.1.2. *Edge removal.* There is no known precise formula for updating the transitive closure of a general graph. For general acyclic graphs, Dong and Su [DS95] give a precise update formula that is beyond the scope of this work. For functional graphs, Hesse [Hes03] gives precise update formulas based on either an auxiliary binary relation, or by using a ternary relation to describe paths in the graph that pass through each node. Without these additions, it is not possible to give precise update formulas in the presence of cyclicity.

When limiting the discussion to acyclic graphs in which between any two nodes there is at most one path (such as acyclic functional graphs and trees) it is possible to give a simple precise update formula. As before, let s be the source of the edge to be removed and t be the target of the edge. The formula for removing an edge is

$$\forall v_1, v_2 \,.\, e'(v_1, v_2) \;\leftrightarrow\; (e(v_1, v_2) \wedge \neg(s(v_1) \wedge t(v_2))).$$

NewStart$[true, e, e']$
NewStart$[r_{t,e} \wedge \neg r_{s,\overleftarrow{e}}, e', e]$
NewStart$[\neg r_{t,e} \wedge r_{s,\overleftarrow{e}}, e', e]$
NewStart$[\neg r_{t,e}, e', e]$
NewStart$[\neg r_{s,\overleftarrow{e}}, e', e]$
NoExit$[\neg r_{s,\overleftarrow{e}}, e']$
NoExit$[r_{t,e}, e']$

(a)

NewStart$[true, e', e]$
NewStart$[r_{t,e}, e, e']$
NewStart$[r_{s,\overleftarrow{e}}, e, e']$
NewStart$[\neg r_{t,e}, e, e']$
NewStart$[\neg r_{s,\overleftarrow{e}}, e, e']$
NoExit$[r_{s,\overleftarrow{e}}, e']$

(b)

NewStart$[true, e', e]$
NewStart$[r_{t,e}, e, e']$
NewStart$[r_{s,\overleftarrow{e}}, e, e']$
NewStart$[\neg r_{t,e}, e, e']$
NewStart$[\neg r_{s,\overleftarrow{e}}, e, e']$
NoExit$[\neg r_{t,e}, e']$

(c)

Table 6: Axioms instantiated for the proof of the precise update formula of: (a) adding an edge to a general graph, (b) removing an edge from an acyclic functional graph, and (c) removing an edge from a tree.

The precise update formula for this change is

$$\exists v_s, v_t \,.\, s(v_s) \wedge t(v_t) \wedge \forall v_1, v_2 \,.\, e'_{\mathrm{tc}}(v_1, v_2) \;\leftrightarrow\; (e_{\mathrm{tc}}(v_1, v_2) \wedge \neg(e_{\mathrm{tc}}(v_1, v_s) \wedge e_{\mathrm{tc}}(v_t, v_2))).$$

We have used SPASS to prove the validity of this update formula for the case of acyclic functional graphs and the case of trees. As in edge addition, $r_{t,e}$ and $r_{s,\overleftarrow{e}}$ are used as the basic colors. The axioms instantiated in the proof are given in Table 6(b) and Table 6(c).

7. RELATED WORK

Shape Analysis. This work was motivated by our experience with TVLA [LAS00, SRW02], which is a generic system for abstract interpretation [CC77]. The TVLA system is more automatic than the methods described in this paper since it does not rely on user-supplied loop invariants. However, the techniques presented in the present paper are potentially more precise due to the use of full first-order reasoning. It can be shown that the **NoExit** scheme allows us to infer reachability at least as precisely as evaluation rules for 3-valued logic with Kleene semantics. In the future, we hope to develop an efficient non-interactive theorem prover that enjoys the benefits of both approaches. An interesting observation is that the colors needed in our examples to prove the formula are the same unary predicates used by TVLA to define its abstraction. This similarity may, in the future, help us find better ways to automatically instantiate the required axioms. In particular, inductive logic programming has recently been used to learn formulas to use in TVLA abstractions [LRS05], which holds out the possibility of applying similar methods to further automate the approach of the present paper.

Decidable Logics. Decidable logics can be employed to define properties of linked data structures: Weak monadic second-order logic has been used in [EMS00, MS01] to define properties of heap-allocated data structures, and to conduct Hoare-style verification using programmer-supplied loop invariants in the PALE system [MS01]. A decidable logic called L_r (for "logic of reachability expressions") was defined in [BRS99]. L_r is rich enough to express the shape descriptors studied in [SRW98] and the path matrices introduced in [Hen90]. More recent decidable logics include Logic of Reachable Patterns [YRS$^+$06] and a decision procedure for linked data structures that can handle singly linked lists [BR06].

The present paper does not develop decision procedures, but instead suggests methods that can be used in conjunction with existing theorem provers. Thus, the techniques are incomplete and the theorem provers need not terminate. However, our initial experience is that the extra flexibility gained by the use of first-order logic with transitive closure is promising. For example, we can prove

the correctness of imperative destructive list-reversal specified in a natural way and the correctness of mark and sweep garbage collectors, which are beyond the scope of Mona and L_τ.

Indeed, in [IRR$^+$04b], we have tried to simulate existing data structures using decidable logics and realized that this can be tricky because the programmer may need to prove a specific simulation invariant for a given program. Giving an inaccurate simulation invariant causes the simulation to be unsound. One of the advantages of the technique described in the present paper is that soundness is guaranteed no matter which axioms are instantiated. Moreover, the simulation requirements are not necessarily expressible in the decidable logic.

Other First-Order Axiomatizations of Linked Data Structures. The closest approach to ours that we are aware of was taken by Nelson as we describe in Section 4. This also has some follow-up work by Leino and Joshi [Lei98]. Our impression from their write-up is that Leino and Joshi's work can be pushed forward by using our coloring axioms.

A more recent work by Lahiri and Qadeer [LQ06] uses first-order axiomatization. This work can be seen as a specialization of ours to the case of (cyclic) singly linked lists.

Dynamic Maintenance of Transitive Closure. Another orthogonal but promising approach to transitive closure is to maintain reachability relations incrementally as we make unit changes in the data structure. It is known that in many cases, reachability can be maintained by first-order formulas [DS95, PI97] and even sometimes by quantifier-free formulas [Hes03]. Furthermore, in these cases, it is often possible to automatically derive the first-order update formulas using finite differencing [RSL03].

8. CONCLUSION

This paper reports on our proposal of a new methodology for using off-the-shelf first-order theorem provers to reason about reachability in programs. We have explored many of the theoretical issues as well as presenting examples that, while still preliminary, suggest that this is indeed a viable approach.

As mentioned earlier, proving the absence of paths is the difficult part of proving formulas with TC. The promise of our approach is that it is able to handle such formulas effectively and reasonably automatically, as shown by the fact that it can successfully handle the programs described in Section 5 and the success of the TVLA system, which uses similar transitive-closure reasoning. Of course, much further work is needed including the following:

- Exploring other heuristics for identifying color classes.
- Exploring variations of the algorithm given in Fig. 10 for instantiating coloring axioms.
- Exploring the use of additional axiom schemes, such as two of the schemes from [Nel83], which are likely to be useful when dealing with predicates that are partial functions. Such predicates arise in programs that manipulate singly-linked or doubly-linked lists—or, more generally, data structures that are acyclic in one or more "dimensions" [HHN92] (i.e., in which the iterated application of a given field selector can never return to a previously visited node).
- Additional work should be done on the theoretical power of $T_1 + \mathbf{IND}$ and related axiomatizations of transitive closure. We conjecture, for example, that $T_1 + \mathbf{IND}$ is TC-complete for trees.

Acknowledgements. Thanks to Aharon Abadi and Roman Manevich for interesting suggestions. Thanks to Viktor Kuncak for useful conversations including his observation and proof of Proposition 4.4.

REFERENCES

[Avr03] A. Avron. Transitive closure and the mechanization of mathematics. In *Thirty Five Years of Automating Mathematics*, pages 149–171. Kluwer Academic Publishers, 2003.

[BLARS07] I. Bogudlov, T. Lev-Ami, T. Reps, and M. Sagiv. Revamping tvla: Making parametric shape analysis competitive. In *CAV*, 2007.

[BR06] J. Bingham and Z. Rakamaric. A logic and decision procedure for predicate abstraction of heap-manipulating programs. In *VMCAI*, pages 207–221, 2006.

[BRS99] 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.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL '77: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252. ACM Press, 1977.

[CC79] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Symp. on Princ. of Prog. Lang.*, pages 269–282, New York, NY, 1979. ACM Press.

[DS95] G. Dong and J. Su. Incremental and decremental evaluation of transitive closure by first-order queries. *Inf. & Comput.*, 120:101–106, 1995.

[EMS00] J. Elgaard, A. Møller, and M.I. Schwartzbach. Compile-time debugging of C programs working on trees. In *European Symp. On Programming*, pages 119–134, 2000.

[FLL+02] C. Flanagan, K.R.M. Leino, M. Lillibridge, G. Nelson, J.B. Saxe, and R. Stata. Extended static checking for java. In *SIGPLAN Conf. on Prog. Lang. Design and Impl.*, 2002.

[GME99] E. Grädel, M.Otto, and E.Rosen. Undecidability results on two-variable logics. *Archive of Math. Logic*, 38:313–354, 1999.

[Hen90] L. Hendren. *Parallelizing Programs with Recursive Data Structures*. PhD thesis, Cornell Univ., Ithaca, NY, Jan 1990.

[Hes03] W. Hesse. *Dynamic Computational Complexity*. PhD thesis, Department of Computer Science, UMass, Amherst, July 2003.

[HHN92] 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.

[Hoa75] C.A.R. Hoare. Recursive data structures. *Int. J. of Comp. and Inf. Sci.*, 4(2):105–132, 1975.

[IRR+04a] N. Immerman, A. Rabinovich, T. Reps, M. Sagiv, and G. Yorsh. The boundary between decidability and undecidability of transitive closure logics. In *CSL'04*, 2004.

[IRR+04b] N. Immerman, A. Rabinovich, T. Reps, M. Sagiv, and G. Yorsh. Verification via structure simulation. In *Proc. Computer-Aided Verif.*, pages 281–294, 2004.

[LARSW00] T. Lev-Ami, T. Reps, M. Sagiv, and R. Wilhelm. Putting static analysis to work for verification: A case study. In *ISSTA 2000: Proc. of the Int. Symp. on Software Testing and Analysis*, pages 26–38, 2000.

[LAS00] T. Lev-Ami and M. Sagiv. TVLA: A system for implementing static analyses. In *Static Analysis Symp.*, pages 280–301, 2000.

[LASR07] T. Lev-Ami, M. Sagiv, and T. Reps. Backward analysis for inferring quantified preconditions. Submitted for publication, 2007.

[Lei98] R. Leino. Recursive object types in a logic of object-oriented programs. *Nordic J. of Computing*, 5:330–360, 1998.

[LQ06] S. K. Lahiri and S. Qadeer. Verifying properties of well-founded linked lists. In *POPL*, pages 115–126, 2006.

[LRS05] A. Loginov, T. Reps, and M. Sagiv. Abstraction refinement via inductive learning. In *Proc. Computer-Aided Verif.*, 2005.

[LRS06] A. Loginov, T. Reps, and M. Sagiv. Automatic verification of the Deutsch-Schorr-Waite tree-traversal algorithm. In *SAS*, 2006.

[MP71] R. McNaughton and S. Papert. *Counter-Free Automata*. MIT Press, 1971.

[MS01] A. Møller and M.I. Schwartzbach. The pointer assertion logic engine. In *SIGPLAN Conf. on Prog. Lang. Design and Impl.*, pages 221–231, 2001.

[Nel83] G. Nelson. Verifying reachability invariants of linked structures. In *Symp. on Princ. of Prog. Lang.*, pages 38–47, 1983.

[PI97] S. Patnaik and N. Immerman. Dyn-FO: A parallel, dynamic complexity class. *Journal of Computer and System Sciences*, 55(2):199–209, October 1997.

[RSL03] T. Reps, M. Sagiv, and A. Loginov. Finite differencing of logical formulas for static analysis. In *European Symp. On Programming*, pages 380–398, 2003.

[RSW04] T. Reps, M. Sagiv, and R. Wilhelm. Static program analysis via 3-valued logic. In *CAV*, pages 15–30, 2004.

[SRW98] M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. *Trans. on Prog. Lang. and Syst.*, 20(1):1–50, January 1998.

[SRW02] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *Trans. on Prog. Lang. and Syst.*, 2002.

[WGR96] Christoph Weidenbach, Bernd Gaede, and Georg Rock. Spass & flotter version 0.42. In *CADE-13: Proceedings of the 13th International Conference on Automated Deduction*, pages 141–145. Springer-Verlag, 1996.

[YRS+06] G. Yorsh, A. Rabinovich, M. Sagiv, A. Meyer, and A. Bouajjani. A logic of reachable patterns in linked data-structures. In *FOSSACS*, 2006.