Decidable Verification of Uninterpreted Programs

UMANG MATHUR, University of Illinois, Urbana Champaign, USA
P. MADHUSUDAN, University of Illinois, Urbana Champaign, USA
MAHESH VISWANATHAN, University of Illinois, Urbana Champaign, USA

We study the problem of completely automatically verifying uninterpreted programs—programs that work over arbitrary data models that provide an interpretation for the constants, functions and relations the program uses. The verification problem asks whether a given program satisfies a postcondition written using quantifier-free formulas with equality on the final state, with no loop invariants, contracts, etc. being provided. We show that this problem is undecidable in general. The main contribution of this paper is a subclass of programs, called *coherent programs* that admits decidable verification, and can be decided in PSPACE. We then extend this class of programs to classes of programs that are k-coherent, where $k \in \mathbb{N}$, obtained by (automatically) adding k ghost variables and assignments that make them coherent. We also extend the decidability result to programs with recursive function calls and prove several undecidability results that show why our restrictions to obtain decidability seem necessary.

CCS Concepts: • **Theory of computation → Logic and verification**; *Automated reasoning*;

Additional Key Words and Phrases: Uninterpreted Programs, Coherence, Program Verification, Decidability, Streaming Congruence Closure

ACM Reference Format:
Umang Mathur, P. Madhusudan, and Mahesh Viswanathan. 2019. Decidable Verification of Uninterpreted Programs. *Proc. ACM Program. Lang.* 3, POPL, Article 46 (January 2019), 29 pages. https://doi.org/10.1145/3290359

1 INTRODUCTION

Completely automatic verification of programs is almost always undecidable. The class of sequential programs, with and without recursive functions, admits a decidable verification problem when the state-space of variables/configurations is *finite*, and this has been the cornerstone on which several fully automated verification techniques have been based, including predicate abstraction, and model-checking. However, when variables range over infinite domains, verification almost inevitably is undecidable. For example, even for programs manipulating natural numbers with increment and decrement operators, and checks for equality, program verification is undecidable.

In this paper, we investigate classes of programs over *uninterpreted* functions and relations over infinite domains that admit, surprisingly, a decidable verification problem (with no user help whatsoever, not even in terms of inductive loop invariants or pre/post conditions).

A program can be viewed as working over a data-domain that consists of constants, functions and relations. For example, a program manipulating integers works on a data-model that provides

Authors' addresses: Umang Mathur, Department of Computer Science, University of Illinois, Urbana Champaign, USA, umathur3@illinois.edu; P. Madhusudan, Department of Computer Science, University of Illinois, Urbana Champaign, USA, madhu@illinois.edu; Mahesh Viswanathan, Department of Computer Science, University of Illinois, Urbana Champaign, USA, vmahesh@illinois.edu.

This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2019 Copyright held by the owner/author(s).
2475-1421/2019/1-ART46
https://doi.org/10.1145/3290359

constants like 0, 1, functions like $+$, $-$, and relations like $\leq$, where there is an implicit assumption on the meaning of these constants, functions, and relations. Programs over uninterpreted data models work over *arbitrary* data-models, where the interpretation of functions and relations are not restricted in any way, except of course that equality is a congruence with respect to their interpretations (e.g., if $x = y$, then $f(x) = f(y)$, no matter what the interpretation of f is). A program satisfies its assertions over uninterpreted data models if it satisfies the assertions when working over *all* data-models.

The theory of uninterpreted functions is a theory that only has the congruence axioms, and is an important theory both from a theoretical and practical standpoint. Results in classical logic such as Gödel's (weak) completeness theorem are formulated for such theories. And in verification, when inductive loop invariants are given, verification conditions are often formulated as formulas in SMT theories, where the theory of uninterpreted functions is an important theory used to model memory, pointers in heaps, arrays, and mathematical specifications. In particular, the quantifier-free logic of uninterpreted functions is decidable and amenable to Nelson-Oppen combination with other theories, making it a prime theory in SMT solvers [Bradley and Manna 2007].

We show, perhaps unsurprisingly, that verification of uninterpreted programs is undecidable. The main contribution of this paper is to identify a class of programs, called *coherent* programs, for which verification is decidable.

Program executions can be viewed abstractly as computing *terms* conditioned on *assumptions over terms*. Assignments apply functions of the underlying data-domain and hence the value of a variable at any point in an execution can be seen as computing a term in the underlying data-model. Conditional checks executed by the program can be seen as assumptions the program makes regarding the relations that hold between terms in the data-model. For example, after an execution of the statements $x := y; x := x + 1; \textbf{assume}(x > 0); z := x * y$, the program variable z corresponds to the term $(\widehat{y} + 1) * \widehat{y}$, and the execution makes the assumption that $\widehat{y} + 1 > 0$, where $\widehat{y}$ is the value of variable y at the start of the execution. A coherent program has only executions where the following two properties hold. The first is the *memoizing property* that says that when a term is *recomputed* by the execution, then some variable of the program already has the same term (or perhaps, a different term that is equivalent to it, modulo the assumptions seen so far in the execution). The second property, called *early assumes* says, intuitively, that when an assumption of equality between variables x and y is made by a program, superterms of the terms stored in variables x and y computed by the program must be stored in one of the current variables.

We show that the notion of coherence effectively skirts undecidability of program verification. Both notions of memoizing and early-assumes require variables to store certain computed terms in the current set of variables. This notion in fact is closely related to *bounded path-width* of the computational graph of terms computed by the program; bounded path-width and bounded tree-width are graph-theoretic properties exploited by many decision procedures of graphs for decidability of MSO and for efficient algorithms [Courcelle and Engelfriet 2012; Seese 1991], and there have been several recent works where they have been useful in finding decidable problems in verification [Chatterjee et al. 2016, 2015; Madhusudan and Parlato 2011].

Our decidability procedure is automata-theoretic. We show that coherent programs generate *regular* sets of coherent executions, and we show how to construct automata that check whether an execution satisfies a post-condition assertion written in quantifier-free theory of equality. The automaton works by computing the *congruence closure* of terms defined by equality assumptions in the execution, checking that the disequality assumptions are met, while maintaining this information only on the bounded window of terms corresponding to the current valuation of variables of the program. In fact, the automaton can be viewed as a *streaming congruence closure algorithm* that computes the congruence closure on the moving window of terms computed by the program. The

assumption of coherence is what allows us to build such a streaming algorithm. We show that if either the *memoizing* assumption or the *early-assumes* assumption is relaxed, the verification problem becomes undecidable, arguing for the necessity of these assumptions.

The second contribution of this paper is a decidability result that extends the first result to a larger class of programs — those uninterpreted programs that can be *converted* to coherent ones. A program may not be coherent because of an execution that either recomputes a term when no variable holds that term, or makes an assumption on a term whose superterm has been previously computed but later over-written. However, if the program was given access to more variables, it could keep the required term in an auxiliary variable to meet the coherence requirement. We define a natural notion of k-coherent executions — executions that can be made coherent by adding k *ghost variables* that are write-only and assigned at appropriate times. We show that programs that generate k-coherent executions also admit a decidable verification problem. The notion of k-coherence is again related to path-width — instead of demanding that executions have path-width $|V|$, we allow them to have path-width $|V| + k$, where V is the set of program variables.

We also show that k-coherence is a decidable property. Given a program and $k \in \mathbb{N}$, we can decide if its executions can be made k-coherent. (Notice that when $k = 0$, k-coherence is simply coherence, and so these results imply the decidability of coherence as well.) And if they can, we can automatically build a regular collection of coherent executions that automatically add the ghost variable assignments. This result enables us to verify programs by simply providing a program and a budget k (perhaps iteratively increased), and automatically check whether the program's executions can be made k-coherent, and if so, perform automatic verification for it.

The third contribution of this paper is an extension of the above results to programs with recursive function calls. We show that we can build *visibly pushdown automata* (VPA) that read coherent executions of programs with function calls, and compute congruence closure effectively. Intersecting such a VPA with the VPA accepting the program executions and checking emptiness of the resulting VPA gives the decidability result. We also provide the extension of verification to k-coherent recursive programs.

To the best of our knowledge, the results here present the first interesting class of sequential programs over infinite data-domains for which verification is decidable[1].

The main contributions of this paper are:

- We show verification of uninterpreted programs (without function calls) is undecidable.
- We introduce a notion of coherent programs and show verification of coherent uninterpreted programs (without function calls) is decidable and is PSPACE-complete.
- We introduce a notion of k-coherent programs, for any k. We show that given a program (without function calls) and a constant k, we can decide if it is k-coherent; and if it is, decide verification for it.
- We prove the above results for programs with (recursive) function calls, showing decidability and EXPTIME-completeness.

The paper is structured as follows. Section 2 introduces uninterpreted programs and their verification problem, and summarizes the main results of the paper. Section 3 contains our main technical result and is devoted to coherent programs and the decision procedure for verifying them, as well as the decision procedure for recognizing coherent programs. In Section 4, we show our undecidability results for general progams as well as programs that satisfy only one of the conditions of the coherence definition. Section 5 consists of our decidability results for k-coherent

[1]There are some automata-theoretic results that can be interpreted as decidability results for sequential programs; but these are typically programs reading streaming data or programs that allow very restricted ways of manipulating counters or are very coarse abstractions, which are not natural classes for software verification. See Section 7 for a detailed discussion.

programs. In Section 6 we extend our results to recursive programs. Related work discussion can be found in Section 7 and concluding remarks in Section 8 where we also discuss possible extensions and applications of our results. We refer the reader to our companion technical report [Mathur et al. 2018] for detailed proofs of the results presented.

2 THE VERIFICATION PROBLEM AND SUMMARY OF RESULTS

In this paper we investigate the verification problem for imperative programs, where expressions in assignments and conditions involve uninterpreted functions and relations. We, therefore, begin by defining the syntax and semantics of the class of programs we study, and then conclude the section by giving an overview of our main results; the details of these results will be presented in subsequent sections.

Let us begin by recalling some classical definitions about first order structures. A (finite) first order *signature* Σ is a tuple $(\mathcal{C}, \mathcal{F}, \mathcal{R})$, where $\mathcal{C}$, $\mathcal{F}$, and $\mathcal{R}$ are finite sets of constants, function symbols, and relation symbols, respectively. Each function symbol and relation symbol is implicitly associated with an arity in $\mathbb{N}_{>0}$. A first order signature is *algebraic* if there are no relation symbols, i.e., $\mathcal{R} = \varnothing$. We will denote an algebraic signature as $\Sigma = (\mathcal{C}, \mathcal{F})$ instead of $\Sigma = (\mathcal{C}, \mathcal{F}, \varnothing)$. An *algebra* or *data model* for an algebraic signature $\Sigma = (\mathcal{C}, \mathcal{F})$, is $\mathcal{M} = (U, \{[\![c]\!] \mid c \in \mathcal{C}\}, \{[\![f]\!] \mid f \in \mathcal{F}\})$ which consists of a universe U and an interpretation for each constant and function symbol in the signature. The set of (ground) *terms* are those that can be built using the constants in $\mathcal{C}$ and the function symbols in $\mathcal{F}$; inductively, it is the set containing $\mathcal{C}$, and if f is an m-ary function symbol, and $t_1, \dots t_m$ are terms, then $f(t_1, \dots t_m)$ is a term. We will denote the set of terms as Terms_Σ or simply Terms, since the signature Σ will often be clear from the context. Given a term t and a data model $\mathcal{M}$, the *interpretation* of t (or the *value* that t evaluates to) in $\mathcal{M}$ will be denoted by $[\![t]\!]_\mathcal{M}$.

2.1 Programs

Our imperative programs will use a finite set of variables to store information during a computation. Let us fix $V = \{v_1, \dots v_r\}$ to be this finite set of variables. These programs will use function symbols and relation symbols from a first order signature $\Sigma = (\mathcal{C}, \mathcal{F}, \mathcal{R})$ to manipulate values stored in the variables. We will assume, without loss of generality, that the first order signature has constant symbols that correspond to the *initial values* of each variable at the begining of the computation. More precisely, let $\widehat{V} = \{\widehat{x} \mid x \in V\} \subseteq \mathcal{C}$ represent the initial values for each variable of the program. The syntax of programs is given by the following grammar.

$$\begin{aligned}\langle stmt\rangle ::= &\ \textbf{skip} \mid x := c \mid x := y \mid x := f(\mathbf{z}) \mid \textbf{assume}\,(\langle cond\rangle) \mid \langle stmt\rangle\,;\,\langle stmt\rangle \\ &\mid \textbf{if}\,(\langle cond\rangle)\,\textbf{then}\,\langle stmt\rangle\,\textbf{else}\,\langle stmt\rangle \mid \textbf{while}\,(\langle cond\rangle)\,\langle stmt\rangle\end{aligned}$$

$$\langle cond\rangle ::= x = y \mid x = c \mid c = d \mid R(\mathbf{z}) \mid \langle cond\rangle \vee \langle cond\rangle \mid \neg\langle cond\rangle$$

Here, $f \in \mathcal{F}$, $R \in \mathcal{R}$, $c, d \in \mathcal{C}$, $x, y \in V$, and $\mathbf{z}$ is a tuple of variables in V and constants in $\mathcal{C}$.

The constructs above define a simple class of programs with conditionals and loops. Here, ':=' denotes the assignment operator, ';' stands for sequencing of programs, **skip** is a "do nothing" statement, **if** − **then** − **else** is a construct for conditional statements and **while** is our looping construct. We will also use the shorthand '**if** $(\langle cond\rangle)$ **then** $\langle stmt\rangle$' as syntactic sugar for '**if** $(\langle cond\rangle)$ **then** $\langle stmt\rangle$ **else skip**'. The conditionals can be equality (=) atoms, predicates defined by relations ($R(\cdot)$), and boolean combinations ($\vee, \neg$) of other conditionals. Formally, the semantics of the program depends on an underlying data model that provides a universe, and meaning for functions, relations, and constants; we will define this precisely in Section 2.3.

The conditionals in the above syntax involve Boolean combinations of equalities as well as relations over variables and constants. However, for technical simplicity and without loss of

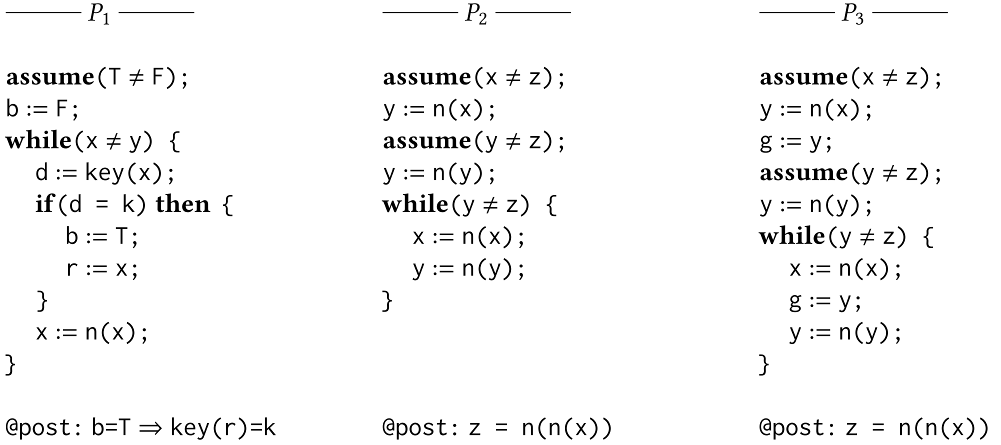

Fig. 1. Examples of Uninterpreted Programs; P_1 and P_3 are coherent, P_2 is not coherent

generality, we disallow relations entirely. Note that a relation R of arity m can be modeled by fixing a new constant $\top$ and introducing a new function f_R of arity m and a variable b_R. Then, each time we see $R(\mathbf{z})$, we add the assignment statement $b_R := f_R(\mathbf{z})$ and replace the occurrence of $R(\mathbf{z})$ by the conditional '$b_R = \top$'. Also, Boolean combinations of conditions can be modeled using the **if** – **then** – **else** construct. Constant symbols used in conditionals and assignments can also be removed simply by using a variable in the program that is not modified in any way by the program. Hence we will avoid the use of constant symbols as well in the program syntax. Henceforth, without loss generality, we can assume that our first order signature Σ is algebraic ($\mathcal{R} = \emptyset$), constant symbols do not appear in any of the program expressions, and our programs have conditionals only of the form $x = y$ or $x \neq y$.

Example 1. Consider the uninterpreted program P_1 in Figure 1. The program works on any first-order model that has an interpretation for the unary functions n and key, and an initial interpretation of the variables T, F, x, y and k. The program is similar to a program that searches whether a list segment from x to y contains a key k. However, in the program above, the functions n and key are uninterpreted, and we allow all possible models on which the program can work. Note that if and when the program terminates, we know that if b = T, then there is an element reachable from x before reaching y such that key applied to that node is equal to k. Note that we are modeling T and F, which are Boolean constants, as variables in the program (assuming that they are different elements in the model).

Programs P_2 and P_3 in Figure 1 are also uninterpreted programs, and resemble programs that given a linked list segment from x to z, finds the node that is two nodes before the node z (i.e., find the node u such that n(n(u)) = z).

2.2 Executions

Definition 1 (Executions)**.** An execution over a finite set of variables V is a word over the alphabet $\Pi = \{$"$x := y$", "$x := f(\mathbf{z})$", "**assume**$(x = y)$", "**assume**$(x \neq y)$" $\mid x, y, \mathbf{z}$ *are in* $V\}$.

We use quotes around letters for readability, and may sometimes skip them.

Definition 2 (Complete and Partial Executions of a program)**.** *Complete executions* of programs that manipulate a set of variables V are executions over V defined formally as follows:

$$
\begin{array}{rcl}
\mathsf{Exec}(\mathbf{skip}) & = & \epsilon \\
\mathsf{Exec}(x := y) & = & \text{“}x := y\text{”} \\
\mathsf{Exec}(x := f(\mathbf{z})) & = & \text{“}x := f(\mathbf{z})\text{”} \\
\mathsf{Exec}(\mathbf{assume}(c)) & = & \text{“}\mathbf{assume}(c)\text{”} \\
\mathsf{Exec}(\mathbf{if}\, c\, \mathbf{then}\, s_1\, \mathbf{else}\, s_2\,) & = & \text{“}\mathbf{assume}(c)\text{”} \cdot \mathsf{Exec}(s_1) \cup \text{“}\mathbf{assume}(\neg c)\text{”} \cdot \mathsf{Exec}(s_2) \\
\mathsf{Exec}(\, s_1; s_2\,) & = & \mathsf{Exec}(s_1) \cdot \mathsf{Exec}(s_2) \\
\mathsf{Exec}(\mathbf{while}\, c\, \{s\}\,) & = & [\text{“}\mathbf{assume}(c)\text{”} \cdot \mathsf{Exec}(s_1)]^* \cdot \text{“}\mathbf{assume}(\neg c)\text{”}
\end{array}
$$

Here, c is a conditional of the form $x = y$ or $x \neq y$, where $x, y \in V$.

The set of *partial executions*, denoted by $\mathsf{PExec}(s)$, is the set of prefixes of complete executions in $\mathsf{Exec}(s)$.

Example 2. For the example program P_1 in Figure 1, the following word ρ

$$
\begin{aligned}
\rho \triangleq\; & \mathbf{assume}(\mathsf{T} \neq \mathsf{F}) \cdot \mathsf{b} := \mathsf{F} \cdot \mathbf{assume}(\mathsf{x} \neq \mathsf{y}) \cdot \mathsf{d} := \mathsf{key}(\mathsf{x}) \cdot \mathbf{assume}(\mathsf{d} \neq \mathsf{k}) \cdot \mathsf{x} := \mathsf{n}(\mathsf{x}) \\
& \cdot \mathbf{assume}(\mathsf{x} \neq \mathsf{y}) \cdot \mathsf{d} := \mathsf{key}(\mathsf{x}) \cdot \mathbf{assume}(\mathsf{d} = \mathsf{k}) \cdot \mathsf{b} := \mathsf{T} \cdot \mathsf{r} := \mathsf{x} \cdot \mathsf{x} := \mathsf{n}(\mathsf{x})
\end{aligned}
$$

is a partial execution of P_1 and the word $\rho_1 = \rho \cdot \mathbf{assume}(\mathsf{x} = \mathsf{y})$ is a complete execution.

Our notion of executions is more syntactic than semantic. In other words, we do not insist that executions are *feasible* over any data model. For example, the word $\mathbf{assume}(x = y) \cdot \mathbf{assume}(x \neq y) \cdot x := f(x)$ is an execution though it is not feasible over any data model. Note also that the complete executions of a program capture (syntactically) *terminating* computations, i.e., computations that run through the entire program.

It is easy to see that an NFA accepting $\mathsf{Exec}(s)$ (as well as for $\mathsf{PExec}(s)$) of size linear in s, for any program s, can be easily constructed in polynomial time from s, using the definitions above and a standard translation of regular expressions to NFAs.

Example 3. For the program P_1 in Figure 1, its set of executions is given by the following regular expression

$$\mathbf{assume}(\mathsf{T} \neq \mathsf{F}) \cdot \mathsf{b} := \mathsf{F} \cdot R \cdot \mathbf{assume}(\mathsf{x} = \mathsf{y})$$

where R is the regular expression

$$[\mathbf{assume}(\mathsf{x} \neq \mathsf{y}) \cdot \mathsf{d} := \mathsf{key}(\mathsf{x}) \cdot (\mathbf{assume}(\mathsf{d} \neq \mathsf{k}) + \mathbf{assume}(\mathsf{d} = \mathsf{k}) \cdot \mathsf{b} := \mathsf{T} \cdot \mathsf{r} := \mathsf{x}) \cdot \mathsf{x} := \mathsf{n}(\mathsf{x})]^*$$

2.3 Semantics of Programs and The Verification Problem

2.3.1 Terms Computed by an Execution. We now define the set of terms computed by executions of a program over variables V. The idea is to capture the term computed for each variable at the end of an execution. Recall that $\widehat{V} = \{\widehat{x} \mid x \in V\}$ is the set of constant symbols that denote the initial values of the variables in V when the execution starts, i.e., $\widehat{x}$ denotes the initial value of variable x, etc. Recall that Terms are the set of all terms over signature Σ. Let $\Pi = \{$“$x := y$”, “$x := f(\mathbf{z})$”, “$\mathbf{assume}(x = y)$”, “$\mathbf{assume}(x \neq y)$” $\mid x, y, \mathbf{z}$ *are in* $V\}$ be the alphabet of executions.

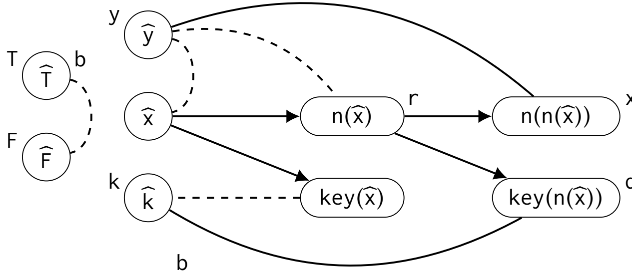

Fig. 2. Computation Graph of ρ_1. Nodes represent terms computed in ρ_1. Directed edges (⟶) represent immediate subterm relation. Nodes are labelled by variables that correspond to the terms denoted by nodes. Undirected solid lines (——) denote equalities and dashed lines (- - - -) denote disequalities seen in ρ_1

Definition 3. The term assigned to a variable x after some partial execution is captured using the function Comp : $\Pi^* \times V \to$ Terms defined inductively as follows.

$$\begin{array}{rcll}
\mathsf{Comp}(\epsilon, x) & = & \widehat{x} & \text{for each } x \in V \\
\mathsf{Comp}(\rho \cdot \text{“}x := y\text{”}, x) & = & \mathsf{Comp}(\rho, y) & \\
\mathsf{Comp}(\rho \cdot \text{“}x := y\text{”}, x') & = & \mathsf{Comp}(\rho, x') & x' \neq x \\
\mathsf{Comp}(\rho \cdot \text{“}x := f(\mathbf{z})\text{”}, x) & = & f(\mathsf{Comp}(\rho, z_1), \ldots, \mathsf{Comp}(\rho, z_r)) & \text{where } \mathbf{z} = (z_1, \ldots, z_r) \\
\mathsf{Comp}(\rho \cdot \text{“}x := f(\mathbf{z})\text{”}, x') & = & \mathsf{Comp}(\rho, x') & x' \neq x \\
\mathsf{Comp}(\rho \cdot \text{“}\mathbf{assume}(y = z)\text{”}, x) & = & \mathsf{Comp}(\rho, x) & \text{for each } x \in V \\
\mathsf{Comp}(\rho \cdot \text{“}\mathbf{assume}(y \neq z)\text{”}, x) & = & \mathsf{Comp}(\rho, x) & \text{for each } x \in V
\end{array}$$

The set of *terms computed* by an execution ρ is $\mathsf{Terms}(\rho) = \bigcup_{\substack{\rho' \text{ is a prefix of } \rho, \\ x \in V}} \mathsf{Comp}(\rho', x)$.

Notice that the terms computed by an execution are independent of the **assume** statements in the execution and depend only on the assignment statements.

Example 4. Consider the execution (from Example 2) below

$$\begin{aligned}
\rho_1 \triangleq\ & \mathbf{assume}(\mathsf{T} \neq \mathsf{F}) \cdot \mathsf{b} := \mathsf{F} \cdot \mathbf{assume}(\mathsf{x} \neq \mathsf{y}) \cdot \mathsf{d} := \mathsf{key}(\mathsf{x}) \cdot \mathbf{assume}(\mathsf{d} \neq \mathsf{k}) \cdot \mathsf{x} := \mathsf{n}(\mathsf{x}) \\
& \cdot \mathbf{assume}(\mathsf{x} \neq \mathsf{y}) \cdot \mathsf{d} := \mathsf{key}(\mathsf{x}) \cdot \mathbf{assume}(\mathsf{d} = \mathsf{k}) \cdot \mathsf{b} := \mathsf{T} \cdot \mathsf{r} := \mathsf{x} \cdot \mathsf{x} := \mathsf{n}(\mathsf{x}) \cdot \mathbf{assume}(x = y)
\end{aligned}$$

For this execution, the set of terms computed can be visualized by the computation graph in Figure 2. Here, the nodes represent the various terms computed by the program, the solid directed edges represent the immediate subterm relation, the solid lines represent the assumptions of equality made in the execution on terms, and the dashed lines represent the assumptions of dis-equality made by the execution. The labels on nodes represent the variables that evaluate to the terms at the end of the execution.

Hence, we have $\mathsf{Comp}(\rho_1, \mathsf{x}) = \mathsf{n}(\mathsf{n}(\widehat{\mathsf{x}}))$, $\mathsf{Comp}(\rho_1, \mathsf{d}) = \mathsf{key}(\mathsf{n}(\widehat{\mathsf{x}}))$, $\mathsf{Comp}(\rho_1, \mathsf{b}) = \widehat{\mathsf{T}}$, $\mathsf{Comp}(\rho_1, \mathsf{y}) = \widehat{\mathsf{y}}$, $\mathsf{Comp}(\rho_1, \mathsf{k}) = \widehat{\mathsf{k}}$, $\mathsf{Comp}(\rho_1, \mathsf{T}) = \widehat{\mathsf{T}}$, $\mathsf{Comp}(\rho_1, \mathsf{F}) = \widehat{\mathsf{F}}$, and $\mathsf{Comp}(\rho_1, \mathsf{r}) = \mathsf{n}(\widehat{\mathsf{x}})$.

2.3.2 Equality and Disequality Assumptions of an Execution. Though the **assume** statements in an execution do not influence the terms that are assigned to any variable, they play a role in defining the semantics of the program. The equalities and disequalities appearing in **assume** statements must hold in a given data model, for the execution to be feasible. We, therefore, identify what these are.

For an (partial) execution ρ, let us first define the set of *equality assumes* that ρ makes on terms. Formally, for any execution ρ, the set of equality assumes defined by ρ, called $\alpha(\rho)$, is a subset of $\mathsf{Terms}(\rho) \times \mathsf{Terms}(\rho)$ defined as follows.

$$\alpha(\epsilon) = \varnothing$$
$$\alpha(\rho \cdot a) = \begin{cases} \alpha(\rho) \cup \{(\mathsf{Comp}(\rho, x), \mathsf{Comp}(\rho, y))\} & \text{if } a \text{ is "}\textbf{assume}(x = y)\text{"} \\ \alpha(\rho) & \text{otherwise} \end{cases}$$

The set of disequality assumes, $\beta(\rho)$, can be similarly defined inductively.

$$\beta(\epsilon) = \varnothing$$
$$\beta(\rho \cdot a) = \begin{cases} \beta(\rho) \cup \{(\mathsf{Comp}(\rho, x), \mathsf{Comp}(\rho, y))\} & \text{if } a \text{ is "}\textbf{assume}(x \neq y)\text{"} \\ \beta(\rho) & \text{otherwise} \end{cases}$$

Example 5. Consider the execution (from Example 2) below

$$\rho_1 \triangleq \textbf{assume}(\mathsf{T} \neq \mathsf{F}) \cdot \mathsf{b} := \mathsf{F} \cdot \textbf{assume}(\mathsf{x} \neq \mathsf{y}) \cdot \mathsf{d} := \mathsf{key(x)} \cdot \textbf{assume}(\mathsf{d} \neq \mathsf{k}) \cdot \mathsf{x} := \mathsf{n(x)}$$
$$\cdot\, \textbf{assume}(\mathsf{x} \neq \mathsf{y}) \cdot \mathsf{d} := \mathsf{key(x)} \cdot \textbf{assume}(\mathsf{d} = \mathsf{k}) \cdot \mathsf{b} := \mathsf{T} \cdot \mathsf{r} := \mathsf{x} \cdot \mathsf{x} := \mathsf{n(x)} \cdot \textbf{assume}(x = y)$$

We have $\alpha(\rho_1) = \{(\mathsf{key}(\mathsf{n}(\widehat{\mathsf{x}})), \widehat{\mathsf{k}}), (\mathsf{n}(\mathsf{n}(\widehat{\mathsf{x}})), \widehat{\mathsf{y}})\}$ and $\beta(\rho_1) = \{(\widehat{\mathsf{T}}, \widehat{\mathsf{F}}), (\widehat{\mathsf{x}}, \widehat{\mathsf{y}}), (\mathsf{key}(\widehat{\mathsf{x}}), \widehat{\mathsf{k}}), (\mathsf{n}(\widehat{\mathsf{x}}), \widehat{\mathsf{y}})\}$.

2.3.3 *Semantics of Programs.* We define the semantics of a program with respect to an algebra or data model that gives interpretations to all the constants and function symbols in the signature. An execution ρ is said to be *feasible* with respect to a data model if, informally, the set of assumptions it makes are true in that model. More precisely, for an execution ρ, recall that $\alpha(\rho)$ and $\beta(\rho)$ are the set of equality assumes and disequality assumes over terms computed in ρ. An execution ρ is feasible in a data-model $\mathcal{M}$, if for every $(t, t') \in \alpha(\rho)$, $[\![t]\!]_{\mathcal{M}} = [\![t']\!]_{\mathcal{M}}$, and for every $(t, t') \in \beta(\rho)$, $[\![t]\!]_{\mathcal{M}} \neq [\![t']\!]_{\mathcal{M}}$.

2.3.4 *The Verification Problem.* Let us now define the logic for postconditions, which are quantifier-free formulas Boolean combination of equality constraints on variables. Given a finite set of variables V, the syntax for postconditions is defined by the following logic $\mathcal{L}_=$.

$$\mathcal{L}_= : \ \varphi ::= x = y \ \mid \ \varphi \vee \varphi \ \mid \ \neg\varphi$$

where above, $x, y \in V$.

Note that a more complex post-condition in the form of a quantifier-free formulae using the functions/relations/constants of the underlying data domain and the current variables can be incorporated by inserting code at the end of the program that computes the relevant terms, leaving the actual postcondition to check only properties of equality over variables.

We can now define the verification problem for uninterpreted programs.

Definition 4 (The Verification Problem for Uninterpreted Programs)**.** Given a $\mathcal{L}_=$ formula φ over a set of variables V, and a program s over V, determine, for every data-model $\mathcal{M}$ and every execution $\rho \in \mathsf{Exec}(s)$ that is feasible in $\mathcal{M}$, if $\mathcal{M}$ satisfies the formula φ under the interpretation that maps every variable $x \in V$ to $[\![\mathsf{Comp}(\rho, x)]\!]_{\mathcal{M}}$. □

It is useful to observe that the verification problem for a program s with postcondition φ in $\mathcal{L}_=$ can be reduced to the verification of a program s' with additional **assume**s and **if** − **then** − **else** statements and postcondition $\bot$. Thus, without loss of generality, we may assume that the postcondition is fixed to be $\bot$. Observe that in this situation, the verification problem essentially reduces to determining the existence of an execution that is feasible in some data model. If there is a feasible execution (in some data model) then the program violates its postcondition; otherwise the program is correct.

2.4 Main Results

In this paper, we investigate the decidability of the verification problem for uninterpreted programs. Our first result is that this problem is, in general, undecidable. We discuss this in detail in Section 4.

Result #1: *The verification problem for uninterpreted programs is undecidable.*

Since the general verification problem is undecidable, we identify a special class of programs for which the verification problem is decidable. In order to describe what these special programs are, we need to introduce a notion of *coherent executions.* Observe that as an execution proceeds, more (structurally) complex terms get computed and assigned to variables, and more **assume** statements identify constraints that narrow the collection of data models in which the execution is feasible. Coherent executions satisfy two properties. The first property, that we call *memoizing*, requires that if a term t is computed (through an assignment) and either t or something "equivalent" (w.r.t. to the equality assumes in the execution) to t was computed before in the execution, then it must be currently stored in one of the variables. This is best illustrated through an example. Consider the partial execution

$$\pi \triangleq \textbf{assume}(\mathsf{x} \neq \mathsf{z}) \cdot \mathsf{y} := \mathsf{n}(\mathsf{x}) \cdot \mathsf{g} := \mathsf{y} \cdot \textbf{assume}(\mathsf{y} \neq \mathsf{z}) \cdot \mathsf{y} := \mathsf{n}(\mathsf{y}) \cdot \textbf{assume}(\mathsf{y} \neq \mathsf{z}) \cdot \mathsf{x} := \mathsf{n}(\mathsf{x})$$

of program P_3 (in Figure 1). The term $\mathsf{n}(\widehat{\mathsf{x}})$ is re-computed in the last step, but it is currently stored in the variable g. On the other hand, a similar partial execution

$$\pi' \triangleq \textbf{assume}(\mathsf{x} \neq \mathsf{z}) \cdot \mathsf{y} := \mathsf{n}(\mathsf{x}) \cdot \textbf{assume}(\mathsf{y} \neq \mathsf{z}) \cdot \mathsf{y} := \mathsf{n}(\mathsf{y}) \cdot \textbf{assume}(\mathsf{y} \neq \mathsf{z}) \cdot \mathsf{x} := \mathsf{n}(\mathsf{x})$$

of P_2 is not memoizing since when $\mathsf{n}(\widehat{\mathsf{x}})$ is recomputed in the last step, it is not stored in any variable; the contents of variable y, which stored $\mathsf{n}(\widehat{\mathsf{x}})$ when it was first computed, have been over-written at this point, or, in other words, the term $\mathsf{n}(\widehat{\mathsf{x}})$ was "dropped" by the execution before it was recomputed. The second property that coherent executions must satisfy is that any step of the form "**assume**$(x = y)$" in the execution comes "early". That is, any superterms of the terms stored in x and y computed by the execution upto this point, are still stored in the program variables and have not been overwritten. The formal definition of coherent executions will be presented later in Section 3 . Finally, a program is *coherent* if all its executions are coherent. The most technically involved result of this paper is that the verification problem for coherent uninterpreted programs is decidable.

Result #2: *The verification problem for coherent uninterpreted programs is decidable.*

The notion of coherence is inspired by the notion of bounded pathwidth, but is admittedly technical. However, we show that determining if a given program is coherent is decidable; hence users of the verification result need not ensure that the program are coherent manually.

Result #3: *Given a program, the problem of checking whether it is coherent is decidable.*

The notion of coherence has two properties, namely, that executions are memoizing and have early-assumes. Both these properties seem to be important for our decidability result. The verification problem for programs all of whose executions satisfy only one of these two conditions turns out to be undecidable.

Result #4: *The verification problem for uninterpreted programs whose executions are memoizing is undecidable. The verification problem for uninterpreted programs whose executions have early assumes is undecidable.*

The memoizing and early-assume requirements of coherence may not be satisfied by even simple programs. For example, program P_2 in Figure 1 does not satisfy the memoizing requirement as demonstrated by the partial execution π' above. However, many of these programs can be made

coherent by adding a finite number of *ghost variables*. These ghost variables are only written and never read, and therefore, play no role in the actual computation. They merely remember terms that have been previously computed and can help meet the memoizing and early-assume requirements. We show that given a budget of k variables, we can *automatically* check whether a corresponding coherent program with k additional ghost variables exists, and in fact compute a regular automaton for its executions, and verify the resulting coherent program. The notation and terminology for k-coherent programs is more complex and we delay defining them formally to Section 5, where they are considered.

Result #5: *Given a program P and $k \in \mathbb{N}$, we can decide whether executions of P can be augmented with k ghost variables and assignments so that they are coherent (i.e., check whether P is k-coherent). Furthermore, if such a coherent program exists, we can construct it and verify it against specifications.*

Finally, in Section 6, we consider programs with recursive function calls, and extend our results to them. In particular, we show the following two results.

Result #6: *The verification problem for coherent uninterpreted programs with recursive function calls is decidable.*

Result #7 *Given a program P, with recursive function calls, and $k \in \mathbb{N}$, we can decide whether executions of P can be augmented with k local ghost variables (for each function) and interleaved ghost assignments that results in a coherent program. Furthermore, if such a coherent program exists, we can construct it and verify if against specifications.*

3 VERIFICATION OF COHERENT UNINTERPRETED PROGRAMS

The verification problem for uninterpreted programs is undecidable; we will establish this result in Section 4. In this section, we establish our main technical results, where we identify a class of programs for which the verification problem is decidable. We call this class of programs *coherent*. We begin by formally defining this class of programs. We then present our algorithm to verify coherent programs. Finally, we conclude this section by showing that the problem of determining if a given program is coherent is also decidable.

Before presenting the main technical content of this section, let us recall that an equivalence relation $\cong \subseteq \text{Terms} \times \text{Terms}$ is said to be a *congruence* if whenever $t_1 \cong t'_1, t_2 \cong t'_2, \ldots t_m \cong t'_m$ and f is an m-ary function then $f(t_1, \ldots t_m) \cong f(t'_1, \ldots t'_m)$. Given a binary relation $A \subseteq \text{Terms} \times \text{Terms}$, the *congruence closure* of A, denoted $\cong_A$, is the smallest congruence containing A.

For a congruence $\cong$ on Terms, the equivalence class of a term t will be denoted by $[t]_\cong$; when $\cong = \cong_A$, we will write this as $[t]_A$ instead of $[t]_{\cong_A}$. For terms $t_1, t_2 \in \text{Terms}$ and congruence $\cong$ on Terms, we say that t_2 *is a superterm of* t_1 *modulo* $\cong$ if there are terms $t'_1, t'_2 \in \text{Terms}$ such that $t'_1 \cong t_1$, $t'_2 \cong t_2$ and t'_2 is a superterm of t'_1.

3.1 Coherent Programs

Coherence is a key property we exploit in our decidability results, and is inspired by the concept of bounded pathwidth. In order to define coherent programs we first need to define the notion of coherence for executions. Recall that, for a partial execution ρ, $\alpha(\rho)$ denotes the set of equality assumes made in ρ.

Definition 5 (Coherent executions)**.** We say that a (partial or complete) execution ρ over variables V is *coherent* if it satisfies the following two properties.

Memoizing. Let $\sigma' = \sigma \cdot$ "$x := f(\mathrm{z})$" be a prefix of ρ and let $t = \text{Comp}(\sigma', x)$. If there is a term $t' \in \text{Terms}(\sigma)$ such that $t' \cong_{\alpha(\sigma)} t$, then there must exist some $y \in V$ such that $\text{Comp}(\sigma, y) \cong_{\alpha(\sigma)} t$.

Early Assumes. Let $\sigma' = \sigma \cdot$ "**assume**$(x = y)$" be a prefix of ρ and let $t_x = \mathsf{Comp}(\sigma, x)$ and $t_y = \mathsf{Comp}(\sigma, y)$. If there is a term $t' \in \mathsf{Terms}(\sigma)$ such that t' is either a superterm of t_x or of t_y modulo $\cong_{\alpha(\sigma)}$, then there must exist a variable $z \in V$ such that $\mathsf{Comp}(\sigma, z) \cong_{\alpha(\sigma)} t'$.

Formally, the memoizing property says that whenever a term t is recomputed (modulo the congruence enforced by the equality assumptions until then), there must be a variable that currently corresponds to t. In the above definition, the assignment $x := f(\mathbf{z})$ is computing a term t, and if t has already been computed (there is a term t' computed by the prefix σ that is equivalent to t), then we demand that there is a variable y which after σ, holds a term that is equivalent to t.

The second requirement of early assumes imposes constraints on when "**assume**$(x = y)$" steps are taken within the execution. We require that such **assume** statements appear before the execution "drops" any computed term t that is a superterm of the terms corresponding to x and y, i.e., before the execution reassigns the variables storing such superterms; notice that $\mathsf{Terms}(\sigma)$ also includes those terms that have been computed along the execution σ and might have been dropped. Formally, we demand that whenever an **assume** statement is executed equating variables x and y, if there is a superterm (t') of either the term stored in x or y modulo the congruence so far, then there must be a variable (z) storing a term equivalent to t'.

Finally, we come to the main concept of this section, namely, that of coherent programs.

Definition 6 (Coherent programs)**.** A coherent program is a program all of whose executions are coherent.

Example 6. Consider the partial execution π' of P_2 (Figure 1) that we considered in Section 2.4.

$\pi' \triangleq$ **assume**$(\mathsf{x} \neq \mathsf{z}) \cdot \mathsf{y} := \mathsf{n}(\mathsf{x}) \cdot$ **assume**$(\mathsf{y} \neq \mathsf{z}) \cdot \mathsf{y} := \mathsf{n}(\mathsf{y}) \cdot$ **assume**$(\mathsf{y} \neq \mathsf{z}) \cdot \mathsf{x} := \mathsf{n}(\mathsf{x})$

Any extension of π' to a complete execution of P_2, will not be coherent. This is because ρ' is not memoizing — when $\mathsf{n}(\widehat{\mathsf{x}})$ is recomputed in the last step, it is not stored in any variable; the contents of variable y, which stored $\mathsf{n}(\widehat{\mathsf{x}})$ when it was first computed, have been over-written at this point.

On the other hand, the following execution over variables $\{\mathsf{x}, \mathsf{y}, \mathsf{z}\}$

$\sigma \triangleq \mathsf{z} := \mathsf{f}(\mathsf{x}) \cdot \mathsf{z} := \mathsf{f}(\mathsf{z}) \cdot$ **assume**$(\mathsf{x} = \mathsf{y})$

is also not coherent because "**assume**$(\mathsf{x} = \mathsf{y})$" is not early. Observe that $\mathsf{Comp}(\sigma, \mathsf{x}) = \widehat{\mathsf{x}}$, $\mathsf{Comp}(\sigma, \mathsf{y}) = \widehat{\mathsf{y}}$, and $\mathsf{Comp}(\sigma, \mathsf{z}) = \mathsf{f}(\mathsf{f}(\widehat{\mathsf{x}}))$. Now $\mathsf{f}(\widehat{\mathsf{x}}) \in \mathsf{Terms}(\sigma)$, is a superterm of $\mathsf{Comp}(\sigma, \mathsf{x})$ but is not stored in any variable.

Consider the programs in Figure 1. P_2 is not coherent because of partial execution π' above. On the other hand, program P_1 is coherent. This is because whenever an execution encounters "**assume**$(\mathsf{d} = \mathsf{k})$", both d and k have no superterms computed in the execution seen so far. The same holds for the "**assume**$(\mathsf{x} = \mathsf{y})$" at the end of an execution due to the **while** loop. Further, whenever a term gets *dropped*, or over-written, it never gets computed again, even modulo the congruence induced by the assume equations. Similar reasoning establishes that P_3 is also coherent.

3.2 Verifying Coherent Programs

We are now ready to prove that the verification problem for coherent programs is decidable. Recall that, without loss of generality, we may assume that the postcondition is $\bot$. Observe that, when the postcondition is $\bot$, a program violates the postcondtion, if there is an execution ρ and a data model $\mathcal{M}$ such that ρ is feasible in $\mathcal{M}$, i.e., every equality and disequality assumption of ρ holds in $\mathcal{M}$. On the face of it, this seems to require evaluating executions in all possible data models. But in fact, one needs to consider only one class of data models. We begin by recalling the notion of an initial model.

Given a binary relation $A \subseteq \mathsf{Terms} \times \mathsf{Terms}$ of equalities, $\mathcal{M}$ is said to *satisfy* A (or A *holds in* $\mathcal{M}$) if for every pair $(t, t') \in A$, $[\![t]\!]_{\mathcal{M}} = [\![t']\!]_{\mathcal{M}}$. For a relation A, there is a canonical model in which A holds.

Definition 7. The *initial term model* for $A \subseteq \mathsf{Terms}\times\mathsf{Terms}$ over an algebraic signature $\Sigma = (\mathcal{C}, \mathcal{F})$ is $\mathcal{T}(A) = (U, \{[\![c]\!] \mid c \in \mathcal{C}\}, \{[\![f]\!] \mid f \in \mathcal{F}\})$ where

- $U = \mathsf{Terms}/\cong_A$,
- $[\![c]\!] = [c]_A$ for any $c \in \mathcal{C}$, and
- $[\![f]\!]([t_1]_A, \ldots [t_m]_A) = [f(t_1, \ldots t_m)]_A$ for any m-ary function symbol $f \in \mathcal{F}$ and terms $t_1, \ldots t_m \in \mathsf{Terms}$.

An important property of the initial term model is the following.

Proposition 1. Let A be a binary relation on terms, and $\mathcal{M}$ be any model satisfying A. For any pair of terms t, t', if $[\![t]\!]_{\mathcal{T}(A)} = [\![t']\!]_{\mathcal{T}(A)}$ then $[\![t]\!]_{\mathcal{M}} = [\![t']\!]_{\mathcal{M}}$.

Proof. Any model $\mathcal{M}$ defines an equivalence on terms $\equiv_{\mathcal{M}}$ as follows: $t \equiv_{\mathcal{M}} t'$ iff $[\![t]\!]_{\mathcal{M}} = [\![t']\!]_{\mathcal{M}}$. Observe that $\equiv_{\mathcal{M}}$ is a congruence, and if $\mathcal{M}$ satisfies A, then $A \subseteq \equiv_{\mathcal{M}}$. Thus, $\cong_A \subseteq \equiv_{\mathcal{M}}$. Next, observe that for the term model $\mathcal{T}(A)$, $\equiv_{\mathcal{T}(A)} = \cong_A$. The proposition follows from these observations. □

One consequence of the above proposition is the following. Let A be a set of equalities, t_1, t_2 be terms, and $\mathcal{M}$ be a data model satisfying A. If $[\![t_1]\!]_{\mathcal{M}} \neq [\![t_2]\!]_{\mathcal{M}}$ then $[\![t_1]\!]_{\mathcal{T}(A)} \neq [\![t_2]\!]_{\mathcal{T}(A)}$. This means that to check the feasibility of an execution ρ, it suffices to check its feasibility in $\mathcal{T}(\alpha(\rho))$.

Corollary 2. Let ρ be any execution. There is a data model $\mathcal{M}$ such that ρ is feasible in $\mathcal{M}$ if and only if ρ is feasible in $\mathcal{T}(\alpha(\rho))$.

That is, an execution ρ is feasible iff $\cong_{\alpha(\rho)} \cap \beta(\rho) = \varnothing$.

Let us return to the problem of verifying if a program satisfies the postcondition $\perp$. This requires us to check that no execution of the program is feasible in any data model. Let us now focus on the simpler problem of *execution verification* — given an execution ρ check if there is some data model in which ρ is feasible. If we can solve the execution verification problem, then we could potentially solve the program verification problem; since the set of executions of a program are regular, we could run the execution verification algorithm synchronously with the NFA representing the set of all program executions to see if any of them are feasible.

Corollary 2 has an important consequence for execution verification — to check if ρ is feasible, evaluate ρ in the data model $\mathcal{T}(\alpha(\rho))$. If the execution verification algorithm is to be lifted to verify all executions of a program, then the algorithm must evaluate the execution as the symbols come in. It cannot assume to have the entire execution. This poses challenges that must be overcome. First the term model $\mathcal{T}(\alpha(\rho))$ is typically infinite and cannot be constructed explicitly. Second, since equality assumptions come in as the execution unfolds, $\alpha(\rho)$ is not known at the beginning and therefore, neither is the exact term model on which ρ must be evaluated known. In fact, in general, we cannot evaluate an arbitrary execution ρ in a term model $\mathcal{T}(\alpha(\rho))$ in an incremental fashion. The main result of this section shows that we can exploit properties of coherent executions to overcome these challenges.

To explain the intuition behind our algorithm, let us begin by considering a naïve algorithm that evaluates an execution in a data model. Suppose the data model $\mathcal{M}$ is completely known. One algorithm to evaluate an execution ρ in $\mathcal{M}$, would keep track of the values of each program variable with respect to model $\mathcal{M}$, and for each **assume** step, check if the equality or disequality assumption holds in $\mathcal{M}$. When $\mathcal{M}$ is the term model, the value that variable x takes after (partial) execution ρ', is the equivalence class of $\mathsf{Comp}(\rho', x)$ with respect to congruence defined by all the equality assumptions in the *complete* execution ρ.

Our algorithm to verify an execution, will follow the basic template of the naïve algorithm above, with important modifications. First, after a prefix ρ' of the execution ρ, we have only seen a partial set of equality assumptions and not the entire set. Therefore, the value of variable x that the algorithm tracks will be $[\mathsf{Comp}(\rho', x)]_{\alpha(\rho')}$ and not $[\mathsf{Comp}(\rho', x)]_{\alpha(\rho)}$. Now, when a new equality assumption $\mathbf{assume}(y = z)$ is seen, we will need to update the values of each variable to be that in the term model that also satisfies this new equation. This requires updating the congruence class of the terms corresponding to each variable as new equalities come in. In addition, the algorithm needs to ensure that if a previously seen disequality assumption is violated because of the new equation, it can be determined, eventhough the disequality assumption maybe between two terms that are no longer stored in any of the program variables. Second, our algorithm will also track the interpretation of the function symbols when applied to the values stored for variables in the program. Thus, after a prefix ρ', the algorithm constructs part of the model $\mathcal{T}(\alpha(\rho'))$ when restricted to the variable values. This partial model helps the algorithm update the values of variables when a new equality assume is read. The third wrinkle concerns how $[\mathsf{Comp}(\rho', x)]_{\alpha(\rho')}$ is stored. We could store a representative term from this equivalence class. This would result in an algorithm whose memory requirements grow with the execution. Instead the algorithm only maintains, for every pair of variables x, y, whether their values in $\mathcal{T}(\alpha(\rho'))$ are equal or not. This means that the memory requirements of the algorithm do not grow with the length of the execution being analyzed. Thus, we will in fact show, that the collection of all feasible partial executions is a regular language.

In order to be able to carry out the above analysis incrementally, our algorithm crucially exploits the coherence properties of the execution. To illustrate one reason why the above approach would not work for non-coherent executions, consider a prefix ρ' such that $\mathsf{Comp}(\rho', x) = \widehat{x}$, $\mathsf{Comp}(\rho', y) = \widehat{y}$, $\mathsf{Comp}(\rho', u) = f^{100}(\widehat{x})$, and $\mathsf{Comp}(\rho', v) = f^{100}(\widehat{y})$. Let us assume that $\alpha(\rho') = \varnothing$. Suppose we now encounter $\mathbf{assume}(x = y)$. This means that in the term model that satisfies this equality, the values of variables u and v are the same. However, this is possible only if the algorithm somehow maintains the information that u and v are the result of hundred applications of f to the values in x and y. This cannot be done using bounded memory. Notice, however, that in this case $\rho' \cdot$ "$\mathbf{assume}(x = y)$" is not a coherent execution because the assume at the end is not early. Early assumes ensure that the effect of an new equality assumption can be fully determined on the current values to the variables.

To understand the importance of the memoizing property in the decision procedure, consider the execution $\rho' \triangleq x := y \cdot \underbrace{y := f(y) \cdots y := f(y)}_{\text{n times}} \cdot \underbrace{x := f(x) \cdots x := f(x)}_{\text{n times}}$. This execution trivially satisfies the "early assumes" criterion. However it is not memoizing since the terms $\widehat{y}, f(\widehat{y}), \ldots, f^{n-1}(\widehat{y})$ have been re-computed after they have been dropped. Now, suppose that $\rho \triangleq \rho' \cdot \mathbf{assume}(x \neq y)$ is a complete extension of ρ'. Notice that ρ is not memoizing but still satisfies the "early assumes" criterion (as it has no equality assumptions). Now, in order for the algorithm to correctly determine that this execution is infeasible, it needs to correctly maintain the information that $\mathsf{Comp}(\rho', y) = \mathsf{Comp}(\rho', x) = f^n(\widehat{y})$. This again, is not possible using bounded memory.

3.2.1 Preliminaries for Streaming Congruence Closure. We will now flesh out the intuitions laid out above. We will introduce concepts and properties that will be used in the formal construction and its correctness proof.

Recall that our algorithm will track the values of the program variables in a term model. When we have a coherent execution $\sigma' = \sigma \cdot \mathbf{assume}(x = y)$, the terms corresponding to program variables obey a special relationship with the set of terms $\mathsf{Terms}(\sigma')$ constructed anytime during

the execution and with the equality assumptions seen in σ. We capture this through the following definition which has been motivated by the condition of early assumes.

Definition 8 (Superterm closedness modulo congruence)**.** Let T be a subterm closed set of terms. Let $E \subseteq T \times T$ be a set of equations on T and $\cong_E$ be its congruence closure. Let $W \subseteq T$ and let $t_1, t_2 \in W$. Then, W is said to be *closed under superterms with respect to* T*,* E *and* (t_1, t_2) if for any term $s \in T$ such that s is a superterm of either t_1 or t_2 modulo $\cong_E$, there is term $s' \in W$ such that $s' \cong_E s$.

Coherent executions ensure that the set of values of variables (or equivalently the set of terms corresponding to the variables) is superterm closed modulo congruence with respect to a newly encountered equality assumption; observe that for any partial execution ρ, $\mathsf{Terms}(\rho)$ is subterm closed.

Lemma 3. Let σ be a coherent execution over variables V and let $\rho' = \rho \cdot$ "$\mathbf{assume}(x = y)$" be any prefix of σ. Then $W = \{\mathsf{Comp}(\rho, v) \mid v \in V\}$ is closed under superterms with respect to $\mathsf{Terms}(\rho)$, $\alpha(\rho)$ and $(\mathsf{Comp}(\rho, x), \mathsf{Comp}(\rho, y))$.

As pointed out in the overview, our algorithm will not explicitly track the terms stored in program variable, but instead track the equivalence between these terms in the term model. In addition, it also tracks the interpretations of function symbols on the stored terms in the term model. Finally, it will store the pairs of terms (stored currently in program variables) that have been assumed to be not equal in the execution. The following definition captures when such an algorithm state is consistent with a set of terms, equalities, and disequalities. In the definition below, the reader may think of W as the set of terms corresponding to each program variable, E as the set equality assumptions, and D as the set of disequality assumptions after a prefix of the execution.

Definition 9 (Consistency)**.** Let W be a set of terms, E a set of equations on terms, and D be a set of disequalities on terms. Let $\equiv_W$ be an equivalence relation on W, $D_W \subseteq W/\equiv_W \times W/\equiv_W$ be a symmetric relation, and P be a partial interpretation of function symbols, i.e., for any k-ary function symbol f, $P(f)$ is a partial function mapping k-tuples in $(W/\equiv_W)^k$ to $W/\equiv_W$. We will say $(\equiv_W, D_W, P)$ is *consistent* with respect to (W, E, D) iff the following hold.

(a) For $t_1, t_2 \in W$, $t_1 \equiv_W t_2$ if and only if $[\![t_1]\!]_{\mathcal{T}(E)} = [\![t_2]\!]_{\mathcal{T}(E)}$, i.e., t_1 and t_2 evaluate to the same value in $\mathcal{T}(E)$,

(b) $([t_1]_{\equiv_W}, [t_2]_{\equiv_W}) \in D_W$ iff there are terms t'_1, t'_2 such that $t'_1 \cong_E t_1$, and $t'_2 \cong_E t_2$ and $\{(t'_1, t'_2), (t'_2, t'_1)\} \cap D \neq \varnothing$.

(c) $P(f)([t_1]_{\equiv_W}, \ldots, [t_k]_{\equiv_W}) = \begin{cases} [t]_{\equiv_W} & \text{if } f(t_1, \ldots t_k) \cong_E t \\ \mathsf{undef} & \text{otherwise} \end{cases}$

There are two crucial properties about a set W that is superterm closed (Definition 8). When we have a state that is consistent (as per Definition 9), we can correctly update it when we add an equation by doing a "local" congruence closure of the terms in W. This is the content of Lemma 4 and its detailed proof can be found in [Mathur et al. 2018].

Lemma 4. Let T be a set of subterm-closed set of terms, $E \subseteq T \times T$ be a set of equalities on T, and $D \subseteq T \times T$ be a set of disequalities. Let $W \subseteq T$ be a set closed under superterms with respect to T, E and some pair $(s_1, s_2) \in W \times W$. Let $(\equiv_W, D_W, P)$ be consistent with (W, E, D). Define $\sim_{s,s'}$ to be the smallest equivalence relation on W such that

- $\equiv_W \cup \{(s, s')\} \subseteq \sim_{s,s'}$
- for every k-ary function symbol f and terms $t_1, t'_1, t_2, t'_2, \ldots t_k, t'_k, t, t' \in W$ such that $t \in P(f)([t_1]_{\equiv_W}, \ldots [t_k]_{\equiv_W})$, $t' \in P(f)([t'_1]_{\equiv_W}, \ldots [t'_k]_{\equiv_W})$, and $(t_i, t'_i) \in \sim_{s,s'}$ for each i, we have $(t, t') \in \sim_{s,s'}$.

In addition, take $D'_W = \{([t_1]_{\sim_{s,s'}}, [t_2]_{\sim_{s,s'}}) \mid ([t_1]_{\equiv_W}, [t_2]_{\equiv_W}) \in D_W\}$ and

$$P'(f)([t_1]_{\sim_{s,s'}}, \ldots [t_k]_{\sim_{s,s'}}) = \begin{cases} [t]_{\sim_{s,s'}} & \text{if } P(f)([t_1]_{\equiv_W}, \ldots [t_k]_{\equiv_W}) = [t]_{\equiv_W} \\ \text{undef} & \text{otherwise} \end{cases}$$

Then $(\sim_{s,s'}, D'_W, P')$ is consistent with $(W, E \cup \{(s, s')\}, D)$.

The second important property about W being superterm closed is that feasibility of executions can be checked easily. Recall that, our previous observations indicate that an execution is feasible, if all the disequality assumptions hold in the term model, i.e., if E is a set of equality assumptions and D is a set of disequality assumptions, feasibility requires checking that $\cong_E \cap D = \varnothing$. Now, we show that when W is superterm closed, then checking this condition when a new equation is added to E can be done by just looking at W; notice that D may have disequalities involving terms that are not in W, and so the observation is not trivial.

Lemma 5. Let T be a set of subterm-closed set of terms, $E \subseteq T \times T$ be a set of equalities on T, and $D \subseteq T \times T$ be a set of disequalities such that $D \cap \cong_E = \varnothing$. Let $W \subseteq T$ and let $t_1, t_2 \in W$ be such that W is closed under superterms with respect to T, E and (t_1, t_2). Let $(\equiv_W, D_W, P)$ be consistent with (W, E, D). Then, $\cong_{E \cup \{(t_1,t_2)\}} \cap D \neq \varnothing$ iff there are terms $t'_1, t'_2 \in W$ such that $([t'_1]_{\equiv_W}, [t'_2]_{\equiv_W}) \in D_W$ and $t'_1 \sim_{t_1,t_2} t'_2$, where $\sim_{t_1,t_2}$ is the equivalence relation on W defined in Lemma 4.

Further, the notion of consistency allows us to correctly check for feasibility when a disequality assumption is seen, and this is formalized below.

Lemma 6. Let T be a set of subterm-closed set of terms, $W \subseteq T$, $E \subseteq T \times T$ be a set of equalities on T, and $D \subseteq T \times T$ be a set of disequalities such that $D \cap \cong_E = \varnothing$. Let $(\equiv_W, D_W, P)$ be consistent with (W, E, D) and let $t_1, t_2 \in W$. Then, $\cong_E \cap (D \cup \{(t_1, t_2)\} \neq \varnothing$ iff $(t_1, t_2) \in \equiv_W$.

Lemmas 3, 4, 5 and 6 suggest that when an execution is coherent, the equivalence between terms stored in program variables can be tracked and feasibility of the execution can be checked, as equality and disequality assumptions are seen. Next, we will exploit these observations to give the construction of an automaton that accepts exactly those coherent executions that are feasible.

3.2.2 Streaming Congruence Closure: Automaton for Feasibility of Coherent Executions. Having given a broad overview of our approach, and defined various concepts and properties, we are ready to present the main result of this section, which says that the collection of feasible partial executions forms a regular language. Let us fix V to be the set of program variables and $\Sigma = (\mathcal{C}, \mathcal{F})$ to be the signature of operations. Recall that Π denotes the alphabet over which executions are defined. We now formally define the automaton $\mathcal{A}_{\text{fs}}$ whose language is the collection of all partial executions that are feasible in some data model.

States. The states in our automaton are either the special state q_{reject} or tuples of the form $(\equiv, d, P)$ where $\equiv \subseteq V \times V$ is an equivalence relation, $d \subseteq V/\equiv \times V/\equiv$ is symmetric and *irreflexive*, and P be a partial interpretation of the function symbols in $\mathcal{F}$, i.e., for any k-ary function symbol f, $P(f)$ is a partial function from $(V/\equiv)^k$ to $V/\equiv$. Intuitively, in a state of the form $(\equiv, d, P)$, $\equiv$ captures the equivalence amongst the terms stored in the variables that hold in the term model satisfying the equalities seen so far, d are the disequality assumptions seen so far restricted to the terms stored

in the program variables, and P summarizes the interpretations of the function symbols in the relevant term model.

Initial state. The initial state $q_0 = (\equiv_0, d_0, P_0)$, where $\equiv_0 = \{(x,x) \mid x \in V\}$, $d_0 = \varnothing$, and for every k-ary function symbol f, and any $x_1, \ldots x_k \in V$, $P(f)([x_1]_{\equiv_0}, \ldots [x_k]_{\equiv_0}) = \text{undef}$ (undefined).

Accepting states. All states except q_{reject} are accepting.

Transitions. The q_{reject} state is absorbing, i.e., for every $a \in \Pi$, $\delta_{\text{fs}}(q_{\text{reject}}, a) = q_{\text{reject}}$. For the other states, the transitions are more involved. Let $q = (\equiv, d, P)$ and let $q' = (\equiv', d', P')$. There is a transtion $\delta_{\text{fs}}(q, a) = q'$ if one of the following conditions holds. If in any of the cases d' is *not irreflexive* then $\delta_{\text{fs}}(q, a) = q_{\text{reject}}$ in each case. For a subset $V' \subseteq V$, we will use the notation $\equiv\downarrow_{V'}$ to denote the relation $\Big(\equiv \cap V' \times V' \Big)$.

$a =$ "$x := y$" .

In this case if y and x are the same variables, then $\equiv' = \equiv$, $d' = d$ and $P' = P$. Otherwise, the variable x gets updated to be in the equivalence class of the variable y, and d' and P' are updated in the most natural way. Formally,

- $\equiv' = \equiv\downarrow_{V \setminus \{x\}} \cup \{(x, y'), (y', x) \mid y' \equiv y\} \cup \{(x,x)\}$.
- $d' = \{([x_1]_{\equiv'}, [x_2]_{\equiv'}) \mid x_1, x_2 \in V \setminus \{x\}, ([x_1]_\equiv, [x_2]_\equiv) \in d\}$
- P' is such that for every r-ary function h,

$$P'(h)([x_1]_{\equiv'}, \ldots [x_r]_{\equiv'}) = \begin{cases} [u]_{\equiv'} & x \notin \{u, x_1, \ldots x_r\} \text{ and} \\ & [u]_\equiv = P(h)([x_1]_\equiv, \ldots [x_r]_\equiv) \\ \text{undef} & \text{otherwise} \end{cases}$$

$a =$ "$x := f(z_1, \ldots z_k)$" .

There are two cases to consider.

(1) **Case** $P(f)([z_1]_\equiv, \ldots [z_k]_\equiv)$ **is defined**.
Let $P(f)([z_1]_\equiv, \ldots [z_k]_\equiv) = [v]_\equiv$. This case is similar to the case when a is "$x := y$". That is, when $x \in [v]_\equiv$, then $\equiv' = \equiv$, $d' = d$ and $P' = P$. Otherwise, we have
   - $\equiv' = \equiv\downarrow_{V \setminus \{x\}} \cup \{(x, v'), (v', x) \mid v' \equiv v\} \cup \{(x,x)\}$
   - $d' = \{([x_1]_{\equiv'}, [x_2]_{\equiv'}) \mid x_1, x_2 \in V \setminus \{x\}, ([x_1]_\equiv, [x_2]_\equiv) \in d\}$
   - P' is such that for every r-ary function h,

$$P'(h)([x_1]_{\equiv'}, \ldots [x_r]_{\equiv'}) = \begin{cases} [u]_{\equiv'} & x \notin \{u, x_1, \ldots x_r\} \text{ and} \\ & [u]_\equiv = P(h)([x_1]_\equiv, \ldots [x_r]_\equiv) \\ \text{undef} & \text{otherwise} \end{cases}$$

(2) **Case** $P(f)([z_1]_\equiv, \ldots [z_k]_\equiv$ **is undefined.**
In this case, we remove x from its older equivalence class and make a new class that only contains the variable x. We update P to P' so that the function f maps the tuple $([z_1]_{\equiv'}, \ldots, [z_k]_{\equiv'})$ (if each of them is a valid/non-empty equivalence class) to the class $[x]_{\equiv'}$. The set d' follows easily from the new $\equiv'$ and the older set d. Thus,
   - $\equiv' = \equiv\downarrow_{V \setminus \{x\}} \cup \{(x,x)\}$
   - $d' = \{([x_1]_{\equiv'}, [x_2]_{\equiv'}) \mid x_1, x_2 \in V \setminus \{x\}, ([x_1]_\equiv, [x_2]_\equiv) \in d\}$
   - P' behaves similar to P for every function different from f.
     - For every r-ary function $h \neq f$,

$$P'(h)([x_1]_{\equiv'}, \ldots, [x_r]_{\equiv'}) = \begin{cases} [u]_{\equiv'} & \text{if } x \notin \{u, x_1, \ldots x_k\} \text{ and} \\ & [u]_\equiv = P(h)([x_1]_\equiv, \ldots [x_r]_\equiv) \\ \text{undef} & \text{otherwise} \end{cases}$$

– For the function f, we have the following.

$$P'(f)([x_1]_{\equiv'},\ldots,[x_k]_{\equiv'}) = \begin{cases} [x]_{\equiv'} & \text{if } x_i = z_i\ \forall i \text{ and } x \notin \{x_1,\ldots x_k\} \\ [u]_{\equiv'} & \text{if } x \notin \{u, x_1,\ldots x_k\} \text{ and} \\ & [u]_{\equiv} = P(f)([x_1]_{\equiv},\ldots [x_k]_{\equiv}) \\ \text{undef} & \text{otherwise} \end{cases}$$

a = “**assume**$(x = y)$” .

Here, we essentially merge the equivalence classes in which x and y belong and perform the “local congruence closure” (as in Lemma 4). In addition, d' and P' are also updated as in Lemma 4.

- $\equiv'$ is the smallest equivalence relation on V such that (a) $\equiv \cup\{(x,y)\} \subseteq\equiv'$, and (b) for every k-ary function symbol f and variables $x_1, x'_1, x_2, x'_2, \ldots x_k, x'_k, z, z' \in V$ such that $[z]_{\equiv} = P(f)([x_1]_{\equiv},\ldots [x_k]_{\equiv})$, $[z']_{\equiv} = P(f)([x'_1]_{\equiv},\ldots [x'_k]_{\equiv})$, and $(x_i, x'_i) \in\equiv'$ for each i, we have $(z,z') \in\equiv'$.
- $d' = \{([x_1]_{\equiv'}, [x_2]_{\equiv'}) \mid ([x_1]_{\equiv}, [x_2]_{\equiv}) \in d\}$
- P' is such that for every r-ary function h,

$$P'(h)([x_1]_{\equiv'},\ldots [x_r]_{\equiv'}) = \begin{cases} [u]_{\equiv'} & \text{if } [u]_{\equiv} = P(h)([x_1]_{\equiv},\ldots [x_r]_{\equiv}) \\ \text{undef} & \text{otherwise} \end{cases}$$

a = “**assume**$(x \neq y)$” .

In this case,

- $\equiv'=\equiv$
- $d' = d \cup \{([x]_{\equiv'}, [y]_{\equiv'}), ([y]_{\equiv'}, [x]_{\equiv'})\}$
- $P' = P$

The formal description of automaton $\mathcal{A}_{\text{fs}}$ is complete. We begin formally stating the invariant maintained by the automaton during its computation.

Lemma 7. Let ρ be a coherent partial execution. Let q_ρ be the state reached by $\mathcal{A}_{\text{fs}}$ after reading ρ. The following properties are true.

(1) If ρ is infeasible then $q_\rho = q_{\text{reject}}$.
(2) If ρ is feasible in some data model, then q_ρ is of the form $(\equiv, d, P)$ such that $(\equiv, d, P)$ is consistent with $(\{\text{Comp}(\rho, x) \mid x \in V\}, \alpha(\rho), \beta(\rho))$; see Definition 9 for the notion of consistency.

Assuming that the signature Σ is of constant size, the automaton $\mathcal{A}_{\text{fs}}$ has $O(2^{|V|^{O(1)}})$ states.

3.2.3 Verifying a Coherent Program. We have so far described a finite memory, streaming algorithm that given a coherent execution can determine if it is feasible in any data model by computing congruence closure. We can use that algorithm to verify coherent, uninterpreted programs.

Theorem 8. *Given a coherent program s with postcondition $\perp$, the problem of verifying s is* PSPACE-*complete.*

Proof Sketch. Observe that a coherent program with postcondition $\perp$ is correct if it has no feasible executions. If $\mathcal{A}_s$ is the NFA accepting precisely the executions of s, the goal is to determine if $L(\mathcal{A}_s) \cap L(\mathcal{A}_{\text{fs}}) \neq \emptyset$. This can be done taking the cross product of the two automata and searching for an accepting path. Notice that storing the states of $\mathcal{A}_{\text{fs}}$ uses space that is polynomial in the number of variables, and the cross product automaton can be constructed on the fly. This gives us a PSPACE upper bound for the verification problem.

The lower bound is obtained through a reduction from Boolean program verification. Recall that Boolean programs are imperative programs with while loops and conditionals, where all program

variables take on Boolean values. The verification problem for such programs is to determine if there is an execution of the program that reaches a special **halt** statement. This problem is known to be PSPACE-hard. The Boolean program verification problem can be reduced to the verification problem for uninterpreted programs — the uninterpreted program corresponding to a Boolean program will have no function symbols in its signature, have constants for true and false, and have two program variables that are never modified which store true and false respectively. Since in such a program the variables never store terms other than constants, the executions are trivially coherent. □

3.3 Decidability of Coherence

Manually checking if a program is coherent is difficult. However, in this section we prove that, given a program s, checking if s is coherent can be done in PSPACE. The crux of this result is an observation that the collection of coherent executions form a regular language. This is the first result we will prove, and we will use this observation to give a decision procedure for checking if a program is coherent. Let us recall that executions of programs over variables V are words over the alphabet Π.

Theorem 9. *The language $L_{cc} = \{\rho \in \Pi^* \mid \rho$ is coherent$\}$ is regular. More precisely, there is a DFA $\mathcal{A}_{cc}$ of size $O(2^{V^{O(1)}})$ such that $L(\mathcal{A}_{cc}) = L_{cc}$.*

Proof. Observe that an execution ρ is coherent if and only if the execution $\rho|_{\Pi\setminus\{\mathbf{assume}(x\neq y)\mid x,y\in V\}}$ is coherent; here $\rho|_{\Pi\setminus\{\mathbf{assume}(x\neq y)\mid x,y\in V\}}$ is the execution obtained by dropping all the disequality assumes from ρ. Hence, the automaton $\mathcal{A}_{cc}$ will ignore all the disequality assumes and only process the other steps.

The automaton $\mathcal{A}_{cc}$ heavily uses the automaton $\mathcal{A}_{fs}$, constructed in Section 3.2. Intuitively, given a word ρ, our algorithm inductively checks whether prefixes of ρ are coherent. Hence, when examining a prefix $\sigma \cdot a$, we can assume that σ is coherent and use the properties of the state of $\mathcal{A}_{fs}$ obtained on σ.

The broad outline of how $\mathcal{A}_{cc}$ works is as follows.

- We keep track of the state of the automaton $\mathcal{A}_{fs}$. Recall that its state is of the form $(\equiv, d, P)$, where $\equiv$ defines an equivalence relation over the variables V, d is a set of disequalities, and P is a partial interpretation of the function symbols in the signature, restricted to contents of the program variables. Now, since we will drop all the disequality assumes, d in this context is always $\varnothing$ and so the state of $\mathcal{A}_{fs}$ will never be q_{reject}.
- In addition, the state of $\mathcal{A}_{cc}$ will have a function E that associates with each function symbol f of arity k, a function from $(V/\equiv)^k$ to $\{\top, \bot\}$. Intuitively, E tracks, for each function symbol f and each tuple of variables, whether f has ever been computed on that tuple at any point in the execution — $E(f)(\mathbf{z}) = \top$, if $f(\mathbf{z})$ has been computed, and is $\bot$ if it has not. Note that, if $P(f)(\mathbf{z})$ is defined (here P is the component that is part of the state of $\mathcal{A}_{fs}$) then $E(f)(\mathbf{z})$ will definitely be $\top$. The role of $E(f)$, however, is to remember in addition whether f has been computed on terms but the image has been "dropped" by the program.

The update of the extra information $E(f)$ is not hard. Whenever $\mathcal{A}_{cc}$ reads "$x := f(z_1, \dots z_k)$", we set $E(f)([z_1]_\equiv, \dots [z_k]_\equiv)$ to $\top$. If $E(f)([z_1]_\equiv, \dots [z_k]_\equiv) = \top$, then we know it was computed, and hence check if $P(f)([z_1]_\equiv, \dots [z_k]_\equiv)$ is defined. If it is *not*, then we reject the word as it is not memoizing and therefore, not coherent.

An **assume**$(x = y)$ statement is dealt with as follows. Let us assume that the state of $\mathcal{A}_{cc}$ is $q = (\equiv, \varnothing, P, E)$ when it processes **assume**$(x = y)$. First $\mathcal{A}_{cc}$ checks if the equality is early as follows. We will say that a variable v is an immediate superterm of u in q, if there is a function f

and arguments $\mathbf{z}$ such that $P(f)(\mathbf{z}) = [v]_\equiv$ and $[u]_\equiv \in \mathbf{z}$. More generally, v is a superterm of u in q if either $[v]_\equiv = [u]_\equiv$, or there is a variable w such that w is an immediate superterm of u in q and v (inductively) is a superterm of w in q. To check that **assume**$(x = y)$ is *not early*, we will first check if there is a superterm u of either x or y in q and a tuple $\mathbf{z}$ such that $[u]_\equiv \in \mathbf{z}$ such that $P(f)(\mathbf{z})$ is undefined but $E(f)(\mathbf{z}) = \top$. If **assume**$(x = y)$ is early then $\mathcal{A}_{cc}$ will merge several equivalence classes of variables as it performs congruence closure locally. Whenever two equivalence classes C and C' merge, we set $E(f)(\mathbf{z})$ to $\top$ and appropriately define $P(f)(\mathbf{z})$ (similar to the transition in $\mathcal{A}_{fs}$) if there were any equivalent variables that were to $\top$ by $E(f)$.

After reading each letter, if the word is not rejected, we are guaranteed that the word is coherent, and hence the meaning of the state of automaton $\mathcal{A}_{fs}$ is correct when reading the next letter. Hence the above automaton precisely accepts the set of coherent words. □

Observe that since $\mathcal{A}_{cc}$ constructed in Theorem 9 is deterministic, it can be modified without blowup to accept only non-coherent words as well.

Using $\mathcal{A}_{cc}$ we can get a PSPACE algorithm to check if a program is coherent.

We can now compute, given a program s, the NFA for $\mathsf{Exec}(s)$, and check whether the intersection of $\mathsf{Exec}(s)$ and the above automaton constructed for accepting non-coherent words is empty. Hence

Theorem 10. *Given a program s, one can determine if s is coherent in* PSPACE.

Proof. Let $\mathcal{A}_s$ be the NFA accepting the set of execution of program s. Let $\overline{\mathcal{A}_{cc}}$ be the automaton accepting the collection of all non-coherent executions. Notice that s is coherent iff $L(\mathcal{A}_s) \cap L(\overline{\mathcal{A}_{cc}}) = \varnothing$. The PSPACE algorithm will construct the product of $\mathcal{A}_s$ and $\overline{\mathcal{A}_{cc}}$ on the fly, while it searches for an accepting computation. □

4 UNDECIDABILITY OF VERIFICATION OF UNINTERPRETED PROGRAMS

We show that verifying uninterpreted programs is undecidable by reducing the halting problem for 2-counter machines.

A 2-counter machine is a finite-state machine (with Q as the set of states) augmented with two counters C_1 and C_2 that take values in $\mathbb{N}$. At every step, the machine moves to a new state and performs one of the following operations on one of the counters: check for zero, increment by 1, or decrement by 1. We can reduce the halting problem for 2-counter machines to verification of uninterpreted programs to show the following result (detailed proof in [Mathur et al. 2018]; we found just before this paper went to print that this result seems to be known in the literature [Müller-Olm et al. 2005]).

Theorem 11. *The verification problem for uninterpreted programs is undecidable.*

The reduction in the above proof proceeds by encoding configurations using primarily three variables, a variable x_{curr} modeling the current state, and two variables y_1 and y_2 modeling the two counters, along with other variables $\{x_q\}_{q \in Q}$ modeling constants for the set of states and the constant $0 \in \mathbb{N}$ and few other auxiliary variables.

The key idea is to model a counter value i using the term $f^i(0)$, and to ensure the data model has two functions f and g, modeling increment and decrement functions respectively, that are inverses of each other on terms representing counter values. We refer the reader to the details of the proof, but note here that this reduction in fact creates programs that are not memoizing nor have early assumes, and in fact cannot be made coherent with any bounded number of ghost variables.

In the following, we present undecidability results (Theorem 12 and Theorem 13) that argue that both our restrictions are required for decidability.

Theorem 12. *The verification problem for uninterpreted programs all of whose executions are memoizing is undecidable.*

Theorem 13. *The verification problem for uninterpreted programs all of whose executions have early-assumes is undecidable.*

The proofs for the theorems above also give reductions from the halting problem for 2-counter machines; one ensures that executions satisfy the memoizing property (while sacrificing the early-assumes criterion) and the other that executions satisfy early-assumes (but not memoizing). Note that the above results do not mean that these two conditions themselves cannot be weakened while preserving decidability. They just argue that neither condition can be simply dropped.

5 k-COHERENT UNINTERPRETED PROGRAMS

In this section, we will generalize the decidability results of Section 3 to apply to a larger class of programs. Programs may sometimes not be coherent because they either violate the memoizing property, i.e., they have executions where a term currently not stored in any of the program variables, is recomputed, or, they violate the early assumes criterion, i.e., have executions where an assume is seen after some superterms have been dropped entirely. However, some of these programs could be *made* to coherent, if they were given access to additional, auxiliary variables that store the terms that need to be recomputed in the future or are needed until a relevant **assume** statement is seen in the future. For example, we observed that program P_2 in Figure 1 is not coherent (Example 6), but program P_3, which is identical to P_2 except for the use of auxiliary variable g is coherent. These auxiliary variables, which we call *ghost variables*, can only be written to. They are never read from, and so do not affect the control flow in the program. We show that the verification problem for *k-coherent programs* — programs that can be made coherent by adding k ghost variables — is decidable. In addition, we show that determining if a program is k-coherent is also decidable.

Let us fix the set of program variables to be $V = \{v_1, v_2, \ldots v_r\}$ and the signature to be $\Sigma = (C, \mathcal{F})$ such that $\widehat{V} \subseteq C$ is the set of initial values of the program variables. Recall that executions of such programs are over the alphabet $\Pi = \{$"$x := y$", "$x := f(\mathbf{z})$", "$\textbf{assume}(x = y)$", "$\textbf{assume}(x \neq y)$" $\mid$ $x, y, \mathbf{z}$ *are in* $V\}$.

Let us define executions that use program variables V and *ghost variables* $G = \{g_1, g_2, \ldots g_l\}$. These will be words over the alphabet $\Pi(G) = \Pi \cup \{$"$g := x$" $\mid g \in G$ and $x \in V\}$. Notice, that the only additional step allowed in such executions is one where a ghost variable is assigned the value stored in a program variable. Before presenting the semantics of such executions, we will introduce some notation. For an execution $\rho \in \Pi(G)^*$, $\rho|_\Pi$ denotes its *projection* onto alphabet Π, i.e., it is the sequence obtained by dropping all ghost variable assignment steps.

To define the semantics of such executions, we once again need to associate terms with variables at each point in the execution. The (partial) function $\mathsf{Comp} : \Pi(G)^* \times (V \cup G) \to \mathsf{Terms}$ will be defined in the same manner as in Definition 3. The main difference will be that Comp now is a partial function, since we will assume that ghost variables are undefined before their first assignment; we do not have constants in our signature Σ corresponding to initial values of ghost variables. For variables $x \in V$, we define $\mathsf{Comp}(\rho, x) = \mathsf{Comp}(\rho|_\Pi, x)$, where the function Comp on the right hand side is given in Definition 3. For ghost variables, Comp is defined inductively as follows.

$$
\begin{array}{rcll}
\mathsf{Comp}(\epsilon, g) & = & \mathsf{undef} & \text{for each } g \in G \\
\mathsf{Comp}(\rho \cdot \text{“}g := x\text{”}, g) & = & \mathsf{Comp}(\rho, x) & \\
\mathsf{Comp}(\rho \cdot \text{“}g := x\text{”}, g') & = & \mathsf{Comp}(\rho, g') & \text{for } g' \neq g \\
\mathsf{Comp}(\rho \cdot a, g) & = & \mathsf{Comp}(\rho, g) & \text{for any } g \in G, \text{ and } a \in \Pi
\end{array}
$$

The set of equality and disequality assumes for executions with ghost variables can be defined in the same way as it was defined for regular executions. More precisely, for any execution $\rho \in \Pi(G)^*$, the set of equality assumes is $\alpha(\rho) = \alpha(\rho|_{\Pi})$ and the set of disequality assumes is $\beta(\rho) = \beta(\rho|_{\Pi})$.

An execution ρ with ghost variables is *coherent* if it satisfies the memoization and early assumes conditions given in Definition 5, except that we also allow variables in G to hold superterms modulo congruence. We are now ready to define the notion of k-*coherence* for executions and programs.

Definition 10. An execution $\sigma \in \Pi^*$ over variables V is said to be k-*coherent* if there is an execution $\rho \in \Pi(G)$ over V and k ghost variables $G = \{g_1, \ldots g_k\}$ such that (a) ρ is coherent, and (b) $\rho|_{\Pi} = \sigma$.

A program s over variables V is said to be k-coherent if every execution $\sigma \in \mathsf{Exec}(s)$ is k-coherent.

Like coherent executions, the collection of k-coherent executions are regular.

Proposition 14. The collection of k-coherent executions over program variables V is regular.

Proof. Theorem 9 establishes the regularity of the collection of all coherent executions. The automaton $\mathcal{A}_{\mathrm{cc}}$ constructed in its proof in Section 3.3 essentially also recognizes the collection of all coherent executions over the set $V \cup G$. Now, observe that the collection of k-coherent executions over the set V is $L(\mathcal{A}_{\mathrm{cc}})|_{\Pi}$, and therefore the proposition follows from the fact that regular languages are closed under projections. □

Given a program, one can decide if it is k-coherent.

Theorem 15. *Given an uninterpreted program s over variables V, one can determine if s is k-coherent in space that is linear in $|s|$ and exponential in $|V|$.*

Proof. Let $\mathcal{A}_s$ be the NFA that accepts the executions of program s, and $\mathcal{A}_{\mathrm{cc}}$ be the automaton recognizing the set of all coherent executions over V and G. Let $\mathcal{A}_{kcc}$ be the automaton recognizing $L(\mathcal{A}_{\mathrm{cc}})|_{\Pi}$, which is the collection of all k-coherent executions. Observe that s is k-coherent if $L(\mathcal{A}_s) \subseteq L(\mathcal{A}_{kcc})$. $\mathcal{A}_{kcc}$ is a nondeterministic automaton with $O(2^{|V|^{O(1)}})$ states, and the proposition therefore follows. □

Finally, the verification problem for uninterpreted k-coherent programs is PSPACE-complete.

Theorem 16. *Given a k-coherent program s with postcondition $\bot$, the problem of verifying s is* PSPACE-*complete.*

Proof. Since every coherent program is also k-coherent (for any k), the lower bound follows from Theorem 8. Let $\mathcal{A}_s$ be the automaton accepting executions of s, $\mathcal{A}_{\mathrm{cc}}$ the automaton accepting coherent executions over V and G, and $\mathcal{A}_{\mathrm{fs}}$ the automaton checking feasibility of executions. Observe that the automaton $\mathcal{A}$ recognizing $(L(\mathcal{A}_{\mathrm{cc}}) \cap L(\mathcal{A}_{\mathrm{fs}}))|_{\Pi}$ accepts the collection of k-coherent executions that are feasible. The verification problem requires one to determine that $L(\mathcal{A}) \cap L(\mathcal{A}_s) = \varnothing$. Using an argument similar to the proof of Theorem 8, this can be accomplished in PSPACE because $\mathcal{A}$ has exponentially many states. □

6 VERIFICATION OF COHERENT AND k-COHERENT RECURSIVE PROGRAMS

In this section, we extend the decidability results to recursive programs. In particular, we define the notions of coherence and k-coherence for recursive programs, and show that the following problems are decidable: (a) verifying recursive coherent programs; (b) determining if a recursive program is coherent; (c) verifying recursive k-coherent programs; and (d) determining if a program is k-coherent.

This section will extend the automata-theoretic constructions to *visibly pushdown automata* [Alur and Madhusudan 2004], and use the fact that they are closed under intersection; we assume the

reader is familiar with these automata as well as with *visibly context-free languages* [Alur and Madhusudan 2009].

6.1 Recursive Programs

Let us fix a finite set of variables $V = \{v_1, \ldots, v_r\}$ and a finite set of method names/identifiers M. Let $m_0 \in M$ be a designated "main" method. Let us also fix a permutation $\langle v_1, \ldots, v_r\rangle$ of variables and denote it by $\langle V\rangle$.

For technical simplicity and without loss of generality, we will assume that for each method $m \in M$, the set of local variables is exactly the set V; methods can, however, ignore certain variables if they use fewer local variables. We will also assume that each method always gets called with *all* the r variables, thereby initializing all local variables. This also does not lead to any loss in generality—if a constant c is used[2] in some function call, the caller can pass this constant to the called function, which will use the passed parameter to initialize its local copy of the variable reserved for c. Finally, we will assume that when a method m is invoked, the order of the parameters is fixed to be $\langle V\rangle$. This again does not lead to a loss of generality — the caller can rearrange the variables to the right order (by swapping) and then reassign them after m returns. These conventions simplify the exposition considerably. We will, however, allow functions to return multiple values back, and allow the caller to assign these to local variables on return.

For every method m, let us fix a tuple $\mathbf{o}_m$ of *output variables* over V. We require the output variables in $\mathbf{o}_m$ to be distinct (in order to avoid implicit aliasing that can be caused when variables are repeated).

The syntax of recursive programs now has method definitions, where the body of the methods can also include recursive calls, besides the usual assignment, sequencing, conditionals and loops.

$$
\begin{aligned}
\langle pgm\rangle &::= m \Rightarrow \mathbf{o}_m\, \langle stmt\rangle \mid \langle pgm\rangle\, \langle pgm\rangle \\
\langle stmt\rangle &::= \textbf{skip} \mid x := y \mid x := f(\mathbf{z}) \mid \textbf{assume}\,(\langle cond\rangle) \mid \langle stmt\rangle\, ;\, \langle stmt\rangle \\
&\quad \mid \textbf{if}\,(\langle cond\rangle)\,\textbf{then}\,\langle stmt\rangle\,\textbf{else}\,\langle stmt\rangle \mid \textbf{while}\,(\langle cond\rangle)\,\langle stmt\rangle \mid \mathbf{w} := m(\langle V\rangle) \\
\langle cond\rangle &::= x = y \mid x \neq y
\end{aligned}
$$

Here, m is a method in M, and the variables $x, y, \mathbf{z}, \mathbf{w}$ belong to V. The length of the vector $\mathbf{w}$ must of course match the length of the vector $\mathbf{o}_m$ of output parameters of the called method m. A program consists of a definition for each method $m \in M$, and we assume each method is defined exactly once.

Example 7. The example in Figure 3 illustrates a recursive program with a single method m_0. This program checks whether any node reachable from `x` using `left` and `right` pointers (which defines a directed acyclic graph) contains a node with key `k`. The method returns a single value—value of the variable `b` upon return. The variables `x` and `k` are the true parameters, but we additionally augment the other variables `b,d,y,T,F` for simplifying notations and have the method rewrite those variables as described before. Here `T` and `F` are variables storing the constants `true` and `false`, respectively. Notice that, it is hard to find an iterative program with a bounded number of variables and without recursive functions that achieves the same functionality.

The semantics of recursive programs given by the grammar $\langle pgm\rangle$ is a standard call-by-value semantics. We next define the formal semantics using terms over a data model.

[2] Recall that, we model constants as initial values of certain variables.

```
m0 ⇒ ⟨b⟩ {
  assume(T ≠ F);
  d := key(x);
  if(d = k) then {
    b := T;
  }
    else {
     y := x;
     x := left(x);
     b := m0(x, k, b, d, y, T, F);
     if(b = F) then {
       x := right(x);
       b := m0(x, k, b, d, y, T, F);
     }
  }
}
```

Fig. 3. Example of uninterpreted recursive program.

6.2 Executions

Executions of recursive programs over the finite set of variables V and the finite set of methods M are sequences over the alphabet $\Pi_M = \{$"$x := y$", "$x := f(\mathbf{z})$", "$\mathbf{assume}(x = y)$", "$\mathbf{assume}(x \neq y)$", "$\mathbf{call}\ m$", "$\mathbf{w} := \mathbf{return}$" $\mid x, y, \mathbf{z}, \mathbf{w}$ are in $V, m \in M\}$.

We will, in fact, treat the above alphabet as partitioned into three kinds: a *call-alphabet*, a *return-alphabet*, and an *internal-alphabet*. The letters of the form "**call** m" belong to the call-alphabet, the letters of the form "$\mathbf{z}$:= **return**" belong to the return alphabet, and the remaining letters belong to the internal alphabet.

The collection of all executions, denoted Exec, is given by the following context-free grammar with start variable E.

$$E \rightarrow \text{“}x := y\text{”} \mid \text{“}x := f(\mathbf{z})\text{”} \mid \text{“}\mathbf{assume}(x = y)\text{”} \mid \text{“}\mathbf{assume}(x \neq y)\text{”}$$
$$\mid \text{“}\mathbf{call}\ m\text{”} \cdot E \cdot \text{“}\mathbf{w} := \mathbf{return}\text{”} \mid E \cdot E$$

In the above rule, m ranges over M. Furthermore, with respect to the call-return-internal alphabet defined above, the above defines a *visibly pushdown language*.

Definition 11 (Complete and Partial Executions of a recursive program)**.** *Complete executions* of recursive programs that manipulate a set of variables V are sequences over Π_M and are defined as follows. Let P be a recursive program. For each method $m \in M$, we denote by $s(m)$ the body (written over the grammar $\langle stmt \rangle$) in the definition of m.

Consider a grammar where we have nonterminals of the form S_s, for various statements $s \in \langle stmt \rangle$, where the rules of the grammar are as follows.

$$
\begin{array}{rcl}
S_{\epsilon} & \rightarrow & \epsilon \\
S_{\textbf{skip}\,;\,s} & \rightarrow & S_s \\
S_{x:=y\,;\,s} & \rightarrow & \text{“}x := y\text{”} \cdot S_s \\
S_{x:=f(\mathbf{z})\,;\,s} & \rightarrow & \text{“}x := f(\mathbf{z})\text{”} \cdot S_s \\
S_{\textbf{assume}(c)\,;\,s} & \rightarrow & \text{“}\textbf{assume}(c)\text{”} \cdot S_s \\
S_{\textbf{if }(c)\textbf{ then } s_1 \textbf{ else } s_2\,;\,s} & \rightarrow & \text{“}\textbf{assume}(c)\text{”} \cdot S_{s_1\,;\,s} \quad | \quad \text{“}\textbf{assume}(\neg c)\text{”} \cdot S_{s_2\,;\,s} \\
S_{\textbf{while }(c)\{s_1\}\,;\,s} & \rightarrow & \text{“}\textbf{assume}(c)\text{”} \cdot S_{s_1\,;\,\textbf{while }(c)\{s_1\}\,;\,s} \quad | \quad \text{“}\textbf{assume}(\neg c)\text{”} \cdot S_s \\
S_{\mathbf{w}:=m(\langle V\rangle)\,;\,s} & \rightarrow & \text{“}\textbf{call}\ m\text{”} \cdot S_{s(m)} \cdot \text{“}\mathbf{z} := \textbf{return}\text{”} \cdot S_s
\end{array}
$$

The set of executions of a program $P \in \langle pgm\rangle$, $\mathsf{Exec}(P)$ are those accepted by the above grammar with start symbol $S_{s(m_0)\,;\,\epsilon}$, where $s(m_0)$ is the body of the “main” method m_0. The set of *partial executions*, denoted by $\mathsf{PExec}(P)$, is the set of prefixes of complete executions in $\mathsf{Exec}(P)$.

In the above definition, the grammar for the language $\mathsf{Exec}(P)$ is taken to be the one that can be defined by using the minimal set of nonterminals for the definitions $S_{s(m)\,;\,\epsilon}$, where $m \in M$. It is easy to see that this is a finite set of nonterminals, and hence the above grammar is a context-free grammar. In fact, all productions rules except the one involving method calls (i.e., production rules for nonterminals of the form $S_{\mathbf{w}:=m(\langle V\rangle)\,;\,s}$) are right-regular grammar productions. Further, the production rules for method calls have a call-letter and return-letter guarding the first nonterminal. Therefore, it is easy to see that the above defines a visibly pushdown language [Alur and Madhusudan 2009]. A visibly pushdown automaton (VPA) that is at most quadratic in the size of the program accepts this language as follows. This VPA will have states of the form S^m_s and mimic the right-regular grammar productions using internal transitions generating the associated terminal, and the rule for method calls by pushing the nonterminal to execute after return onto the stack, and recovering it in its state after the pop when simulating the return from the method call. This construction is fairly standard and simple, and we omit formal definitions.

6.3 Semantics of Recursive Programs and The Verification Problem

6.3.1 Terms Computed by an Execution. Let us now define the term computed for any (local) variable at any point in the computation. We say a subword σ of an execution is *matched* if σ has an equal number of call-letters and return-letters.

We now define the terms that correspond to local variables in scope after a partial execution ρ.

Definition 12. We define $\mathsf{Comp} : \mathsf{Exec} \times V \rightarrow \mathsf{Terms}$ inductively as follows

$$
\begin{array}{rcll}
\mathsf{Comp}(\epsilon, x) & = & \widehat{x} & \\
\mathsf{Comp}(\rho \cdot \text{“}x := y\text{”}, x) & = & \mathsf{Comp}(\rho, y) & \\
\mathsf{Comp}(\rho \cdot \text{“}x := y\text{”}, x') & = & \mathsf{Comp}(\rho, x') & x' \neq x \\
\mathsf{Comp}(\rho \cdot \text{“}x := f(\mathbf{z})\text{”}, x) & = & f(\mathsf{Comp}(\rho, z_1), \ldots, \mathsf{Comp}(\rho, z_r)) & \mathbf{z} = (z_1, \ldots, z_r) \\
\mathsf{Comp}(\rho \cdot \text{“}x := f(\mathbf{z})\text{”}, x') & = & \mathsf{Comp}(\rho, x') & x' \neq x \\
\mathsf{Comp}(\rho \cdot \text{“}\textbf{assume}(R(\mathbf{z}))\text{”}, x) & = & \mathsf{Comp}(\rho, x) & \\
\mathsf{Comp}(\rho \cdot \text{“}\textbf{call}\ m\text{”}, x) & = & \mathsf{Comp}(\rho, x) & \\
\mathsf{Comp}(\rho \cdot \text{“}\textbf{call}\ m\text{”} \cdot \rho' \cdot \text{“}\langle w_1, \ldots w_r\rangle := \textbf{return}\text{”}, w_i) & = & \mathsf{Comp}(\rho \cdot \text{“}\textbf{call}\ m\text{”} \cdot \rho', \mathbf{o}_m[i]) & \rho' \text{ is matched} \\
\mathsf{Comp}(\rho \cdot \text{“}\textbf{call}\ m\text{”} \cdot \rho' \cdot \text{“}\langle w_1, \ldots w_r\rangle := \textbf{return}\text{”}, x) & = & \mathsf{Comp}(\rho, x) & x \notin \{w_1, \ldots w_r\},\ \rho' \text{ is matched}
\end{array}
$$

The set of *terms computed* by an execution ρ is $\mathsf{Terms}(\rho) = \bigcup_{\substack{\rho' \text{ is a prefix of } \rho,\\ v \in V}} \mathsf{Comp}(\rho', v)$.

6.3.2 Semantics of Recursive Programs. Semantics of programs are, again, with respect to a data-model. We define the maps α and β that collect the assumptions of equality and disequality on terms, as we did for programs without function calls; we skip the formal definition as it is the natural extension to executions of recursive programs, ignoring the call and return letters. An execution is feasible on a data model if these assumptions on terms are satisfied in the model.

6.3.3 Verification Problem. As before, without loss of generality we can assume that the post condition is $\perp$ (false). The *verification problem* is then: given a program P, determine if there is some (complete) execution ρ and data model $\mathcal{M}$ such that ρ is feasible on $\mathcal{M}$.

6.3.4 Coherence. The notion of coherence is similar to the one for non-recursive programs and their executions. In fact, it is precisely the definition of coherence for regular programs (Definition 5), except for the fact that it uses the new definitions of Comp, and α for recursive programs. We skip repeating the definition. Note that, in this case, the *memoizing* condition and the *early assumes* condition are based on the set $\mathsf{Terms}(\sigma)$ (where σ is a partial execution), which also includes all terms computed before, including those by other methods. A recursive program is said to be *coherent* if all its executions are coherent.

We can now state our main theorems for recursive programs.

Theorem 17. *The verification problem for coherent recursive programs is decidable, and is* EXPTIME-*complete.* □

The proof of the above result proceeds by constructing a VPA $\mathcal{A}_P$ that accepts the executions of the program P and a VPA $\mathcal{A}_{\mathsf{rfs}}$ that accepts feasible executions of recursive programs, and checking if $L(\mathcal{A}_P \cap L(\mathcal{A}_{\mathsf{rfs}}))$ is empty, the latter being a decidable problem.

The automaton $\mathcal{A}_{\mathsf{rfs}}$ is designed similar to the automaton $\mathcal{A}_{\mathsf{fs}}$ constructed in Section 3.2. Below, we sketch the primary ideas for handling the extension to recursive program executions.

PROOF SKETCH. Recall that the automaton for nonrecursive program executions keeps track of (a) an equivalence relation $\equiv$ over the variables V, (b) partial maps for each k-ary function that map from $(V/\equiv)^k$ to $V/\equiv$, and (c) a set of disequalities over the equivalence classes of $\equiv$. The automaton $\mathcal{A}_{\mathsf{rfs}}$ will keep a similar state, except that it would keep this over *double* the number of variables $V \cup V'$, where $V' = \{v' \mid v \in V\}$. The variables in V' correspond to terms in the *caller* at the time of the call, and these variables do not get reassigned till the current method returns.

When the automaton sees a symbol of the form "**call** m", it pushes the current state of the automaton on the stack, and moves to a state that has only the equivalence classes of the current V variables (along with the partial functions and disequalities restricted to them). It also makes each v' equivalent to v. When processing assignments, assumes, etc. in the called method, the variables V' will never be reassigned, and hence the terms corresponding to them will not be dropped. At the end of the method, when we return to the caller reading a symbol of the form "$\mathbf{w}$:= **return**", we pop the state from the stack and *merge* it with the current state.

This merging essentially recovers the equivalence classes on variables that were not changed across the call and sets up relationship (equivalence, partial f-maps, etc.) to the variables $\mathbf{w}$ assigned by the return. This is done as follows. Let the state popped be s' and the current state be s. Let us relabel variables in s', relabeling each $v \in V$ to $\underline{v}$ and each $v' \in V$ to $\underline{v'}$. Let $\underline{V} = \{\underline{v} \mid v \in V\}$ and $\underline{V'} = \{\underline{v'} \mid v' \in V'\}$. Now let us take the *union* of the two states s and s' (inheriting equivalence classes, partial function maps, disequalities), to get $s \oplus s'$ over variables $\underline{V'} \cup \underline{V} \cup V' \cup V$. In this structure, we merge (identify) each node $\underline{v}$ with v', retaining its label as v'. Merging can cause equivalence classes to merge, thereby also updating partial function interpretations and disequalities. We now *drop* the variables V', dropping the equivalence classes if they become empty.

The new state is over $\underline{V'} \cup V$, and we relabel the variables $\underline{v'}$ to v' to obtain the actual state we transition to. The f-maps and set of disequalities get updated across these manipulations.

The resulting VPA has exponentially many states in $|V|$ and taking its intersection with the automaton $\mathcal{A}_P$ and checking emptiness clearly can be done in exponential time. The lower bound follows from the fact that checking reachability in recursive Boolean programs is already Exptime-hard and the fact that we can emulate any recursive Boolean program using a recursive uninterpreted program (even with an empty signature of functions). ◻

We can also extend the notion of k-coherence to executions of recursive programs; here we allow executions to have ghost assignments at any point to local (write-only) ghost variables in scope, in order to make an execution coherent. We can build an automaton, again a VPA, that accepts *all* k-coherent executions that are semantically feasible. Then, given a program P and a $k \in \mathbb{N}$, we can build an automaton that accepts all coherent extensions of executions of P, and also check whether every execution of P has at least one equivalent coherent execution. If this is true, then P is k-coherent, and we can check whether the automaton accepts any word to verify P.

Theorem 18. *The problem of checking, given a program P and $k \in \mathbb{N}$, whether P is k-coherent is decidable. And if P is found to be k-coherent, verification of P is decidable.*

7 RELATED WORK

The class of programs (with and without recursion) over a finite set of Boolean variables admits a decidable verification problem [Alur et al. 2005; Esparza et al. 2000; Esparza and Knoop 1999; Godefroid and Yannakakis 2013; Schwoon 2002]. As mentioned in the introduction, we believe that our work is the first natural class of programs that work over infinite data domains and yet admit decidable verification, without severely restricting the structure of programs.

There are several automata-theoretic decidability results that could be interpreted as decidability results for programs—for example, coverability and reachability in (unsafe) Petri nets are decidable [Karp and Miller 1969; Kosaraju 1982; Mayr 1981], and this can be interpreted as a class of programs with counters with increments, decrements, and checks for positivity (but *no* checks for zero), which is arguably not a very natural class of programs. The work in [Godoy and Tiwari 2009] establishes decidability for uninterpreted *Sloopy* programs with restricted control flow—such programs only allow non-deterministic guards, disallow the use of conditionals, loops and recursive calls inside other loops and only support checking of equality assertions (see also [Gulwani and Tiwari 2007; Müller-Olm et al. 2005] for verification of uninterpreted programs with other kinds of restrictions).

Complete automatic verification can be seen as doing both the task of finding inductive invariants and validating verification conditions corresponding to the various iteration/recursion-free snippets. In this light, there is classical work for certain domains like *affine programs*, where certain static analyses techniques promise to always find an invariant, if there exists one that can be expressed in a particular logic [Granger 1991; Karr 1976; Müller-Olm and Seidl 2004; Müller-Olm and Seidl 2005]. However, these results do not imply decidable verification for these programs, as there are programs in these classes that are correct but do not have inductive invariants that fall in the fragment of logic considered.

There is a line of work that takes an automata-theoretic flavor to verification [Farzan et al. 2014, 2015; Heizmann et al. 2010, 2013], which rely on building automata accepting infeasible program traces, obtained by generalizing counterexample traces that can be proved infeasible through SMT solving. The method succeeds when it can prove that the set of traces of the program that are erroneous, are contained in the constructed set of infeasible traces. The technique can handle several background theories, but of course tackles an undecidable problem. Our work relates to this

line of work and can be interpreted as a technique for providing, directly and precisely, the set of infeasible coherent traces as a regular/visibly-pushdown language, and thereby providing decidable verification for programs with coherent traces. Combining our techniques for uninterpreted traces with the techniques above for other theories seems a promising future direction.

The theory of uninterpreted functions is a fragment of first order logic with decidable quantifier free fragment [Bradley and Manna 2007], and has been used popularly in abstract domains in program analysis [Alpern et al. 1988; Gulwani and Necula 2004a,b], verification of hardware [Bryant et al. 2001; Burch and Dill 1994] and software [Gulwani and Tiwari 2006; Lopes and Monteiro 2016].

The notion of memoizing executions, which is an integral part of our coherence and k-coherence definitions, is closely related to *bounded path-width* [Robertson and Seymour 1983]. We can think of a computation of the program as sweeping the initial model using a window of terms defined by the set of program variables. The memoizing condition essentially says that the set of windows that contain a term must occur *contiguously* along an execution i.e., a term computed should not be "dropped" if it gets recomputed. The notion of bounded path-width and the related notion of bounded tree-width have been exploited recently in many papers to provide decidability results in verification [Chatterjee et al. 2016, 2015; Madhusudan and Parlato 2011].

8 CONCLUSIONS

We have proved that the class of coherent programs and k-coherent programs (for any $k \in \mathbb{N}$) admit decidable verification problems. Checking if programs are coherent or k-coherent for a given k is also decidable. Moreover, the decision procedure is not very expensive, and in fact matches the complexity of verifying the weaker class of Boolean programs over the same number of variables.

Our results lay foundational theorems for decidable reasoning of uninterpreted programs, and open up a research direction exploring problems that can be tackled using uninterpreted functions/relations. There are several avenues for applications that we foresee. One is reasoning about programs using uninterpreted abstractions, as in the work on reasoning with containers [Dillig et al. 2011] and modeling pointers in heap manipulating programs[Löding et al. 2017; Pek et al. 2014; Qiu et al. 2013]. Such applications will likely call for an extension of our results to handle axioms that restrict the uninterpreted functions (such as associativity and commutativity of certain functions) or to incorporate first order theories such as arithmetic and sets. Specifications for heap manipulating programs often involve recursive definitions, and this may require enriching our results to incorporate such definitions. We also conjecture that our results can be useful in domains such as verification of compiler transformations (such as instruction reordering), when proofs of correctness of transformations rely only on a few assumptions on the semantics of operations and library functions. Trace abstraction based verification approaches [Farzan et al. 2014, 2015; Heizmann et al. 2009, 2010] build automata that capture infeasible traces incompletely using a counter-example guided approach. In this context, our results would enrich such automata—we can accept precisely the set of infeasible traces that become infeasible when making functions uninterpreted. This is a possible future direction to combine our results with other background theories in order to tackle verification applications.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant Nos. 1422798, 1329991, and 1527395.

REFERENCES

B. Alpern, M. N. Wegman, and F. K. Zadeck. 1988. Detecting Equality of Variables in Programs. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '88)*. ACM, New York, NY, USA, 1–11.

https://doi.org/10.1145/73560.73561

Rajeev Alur, Michael Benedikt, Kousha Etessami, Patrice Godefroid, Thomas Reps, and Mihalis Yannakakis. 2005. Analysis of Recursive State Machines. *ACM Trans. Program. Lang. Syst.* 27, 4 (July 2005), 786–818. https://doi.org/10.1145/1075382.1075387

Rajeev Alur and P. Madhusudan. 2004. Visibly Pushdown Languages. In *Proceedings of the Thirty-sixth Annual ACM Symposium on Theory of Computing (STOC '04)*. ACM, New York, NY, USA, 202–211. https://doi.org/10.1145/1007352.1007390

Rajeev Alur and P. Madhusudan. 2009. Adding Nesting Structure to Words. *J. ACM* 56, 3, Article 16 (May 2009), 43 pages. https://doi.org/10.1145/1516512.1516518

Aaron R. Bradley and Zohar Manna. 2007. *The Calculus of Computation: Decision Procedures with Applications to Verification*. Springer-Verlag, Berlin, Heidelberg.

Randal E. Bryant, Steven German, and Miroslav N. Velev. 2001. Processor Verification Using Efficient Reductions of the Logic of Uninterpreted Functions to Propositional Logic. *ACM Trans. Comput. Logic* 2, 1 (Jan. 2001), 93–134. https://doi.org/10.1145/371282.371364

Jerry R. Burch and David L. Dill. 1994. Automatic Verification of Pipelined Microprocessor Control. In *Proceedings of the 6th International Conference on Computer Aided Verification (CAV '94)*. Springer-Verlag, London, UK, UK, 68–80. http://dl.acm.org/citation.cfm?id=647763.735662

Krishnendu Chatterjee, Amir Kafshdar Goharshady, Rasmus Ibsen-Jensen, and Andreas Pavlogiannis. 2016. Algorithms for Algebraic Path Properties in Concurrent Systems of Constant Treewidth Components. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16)*. ACM, New York, NY, USA, 733–747. https://doi.org/10.1145/2837614.2837624

Krishnendu Chatterjee, Rasmus Ibsen-Jensen, Andreas Pavlogiannis, and Prateesh Goyal. 2015. Faster Algorithms for Algebraic Path Properties in Recursive State Machines with Constant Treewidth. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '15)*. ACM, New York, NY, USA, 97–109. https://doi.org/10.1145/2676726.2676979

Professor Bruno Courcelle and Dr Joost Engelfriet. 2012. *Graph Structure and Monadic Second-Order Logic: A Language-Theoretic Approach* (1st ed.). Cambridge University Press, New York, NY, USA.

Isil Dillig, Thomas Dillig, and Alex Aiken. 2011. Precise Reasoning for Programs Using Containers. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '11)*. ACM, New York, NY, USA, 187–200. https://doi.org/10.1145/1926385.1926407

Javier Esparza, David Hansel, Peter Rossmanith, and Stefan Schwoon. 2000. Efficient Algorithms for Model Checking Pushdown Systems. In *Proceedings of the 12th International Conference on Computer Aided Verification (CAV '00)*. Springer-Verlag, London, UK, UK, 232–247. http://dl.acm.org/citation.cfm?id=647769.734087

Javier Esparza and Jens Knoop. 1999. An Automata-Theoretic Approach to Interprocedural Data-Flow Analysis. In *Foundations of Software Science and Computation Structures*, Wolfgang Thomas (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 14–30.

Azadeh Farzan, Zachary Kincaid, and Andreas Podelski. 2014. Proofs That Count. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '14)*. ACM, New York, NY, USA, 151–164. https://doi.org/10.1145/2535838.2535885

Azadeh Farzan, Zachary Kincaid, and Andreas Podelski. 2015. Proof Spaces for Unbounded Parallelism. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '15)*. ACM, New York, NY, USA, 407–420. https://doi.org/10.1145/2676726.2677012

Patrice Godefroid and Mihalis Yannakakis. 2013. Analysis of Boolean Programs. In *Proceedings of the 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'13)*. Springer-Verlag, Berlin, Heidelberg, 214–229. https://doi.org/10.1007/978-3-642-36742-7_16

Guillem Godoy and Ashish Tiwari. 2009. Invariant Checking for Programs with Procedure Calls. In *Proceedings of the 16th International Symposium on Static Analysis (SAS '09)*. Springer-Verlag, Berlin, Heidelberg, 326–342. https://doi.org/10.1007/978-3-642-03237-0_22

Philippe Granger. 1991. Static Analysis of Linear Congruence Equalities Among Variables of a Program. In *Proceedings of the International Joint Conference on Theory and Practice of Software Development on Colloquium on Trees in Algebra and Programming (CAAP '91): Vol 1 (TAPSOFT '91)*. Springer-Verlag New York, Inc., New York, NY, USA, 169–192. http://dl.acm.org/citation.cfm?id=111310.111320

Sumit Gulwani and George C. Necula. 2004a. Global Value Numbering Using Random Interpretation. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '04)*. ACM, New York, NY, USA, 342–352. https://doi.org/10.1145/964001.964030

Sumit Gulwani and George C Necula. 2004b. A polynomial-time algorithm for global value numbering. In *International Static Analysis Symposium*. Springer, 212–227.

Sumit Gulwani and Ashish Tiwari. 2006. Assertion Checking over Combined Abstraction of Linear Arithmetic and Uninterpreted Functions. In *Proceedings of the 15th European Conference on Programming Languages and Systems (ESOP'06)*. Springer-Verlag, Berlin, Heidelberg, 279–293. https://doi.org/10.1007/11693024_19

Sumit Gulwani and Ashish Tiwari. 2007. Assertion Checking Unified. In *Proceedings of the 8th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'07)*. Springer-Verlag, Berlin, Heidelberg, 363–377. http://dl.acm.org/citation.cfm?id=1763048.1763086

Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. 2009. Refinement of Trace Abstraction. In *Proceedings of the 16th International Symposium on Static Analysis (SAS '09)*. Springer-Verlag, Berlin, Heidelberg, 69–85. https://doi.org/10.1007/978-3-642-03237-0_7

Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. 2010. Nested Interpolants. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '10)*. ACM, New York, NY, USA, 471–482. https://doi.org/10.1145/1706299.1706353

Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. 2013. Software Model Checking for People Who Love Automata. In *Computer Aided Verification*, Natasha Sharygina and Helmut Veith (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 36–52.

Richard M. Karp and Raymond E. Miller. 1969. Parallel Program Schemata. *J. Comput. Syst. Sci.* 3, 2 (May 1969), 147–195. https://doi.org/10.1016/S0022-0000(69)80011-5

Michael Karr. 1976. Affine Relationships Among Variables of a Program. *Acta Inf.* 6, 2 (June 1976), 133–151. https://doi.org/10.1007/BF00268497

S. Rao Kosaraju. 1982. Decidability of Reachability in Vector Addition Systems (Preliminary Version). In *Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing (STOC '82)*. ACM, New York, NY, USA, 267–281. https://doi.org/10.1145/800070.802201

Christof Löding, P. Madhusudan, and Lucas Peña. 2017. Foundations for Natural Proofs and Quantifier Instantiation. *Proc. ACM Program. Lang.* 2, POPL, Article 10 (Dec. 2017), 30 pages. https://doi.org/10.1145/3158098

Nuno P. Lopes and José Monteiro. 2016. Automatic equivalence checking of programs with uninterpreted functions and integer arithmetic. *International Journal on Software Tools for Technology Transfer* 18, 4 (01 Aug 2016), 359–374. https://doi.org/10.1007/s10009-015-0366-1

P. Madhusudan and Gennaro Parlato. 2011. The Tree Width of Auxiliary Storage. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '11)*. ACM, New York, NY, USA, 283–294. https://doi.org/10.1145/1926385.1926419

Umang Mathur, P. Madhusudan, and Mahesh Viswanathan. 2018. Decidable Verification of Uninterpreted Programs. *CoRR* abs/1811.00192 (2018). http://arxiv.org/abs/1811.00192

Ernst W. Mayr. 1981. An Algorithm for the General Petri Net Reachability Problem. In *Proceedings of the Thirteenth Annual ACM Symposium on Theory of Computing (STOC '81)*. ACM, New York, NY, USA, 238–246. https://doi.org/10.1145/800076.802477

Markus Müller-Olm, Oliver Rüthing, and Helmut Seidl. 2005. Checking Herbrand Equalities and Beyond. In *Proceedings of the 6th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'05)*. Springer-Verlag, Berlin, Heidelberg, 79–96. https://doi.org/10.1007/978-3-540-30579-8_6

Markus Müller-Olm and Helmut Seidl. 2004. A Note on Karr's Algorithm. In *Automata, Languages and Programming*, Josep Díaz, Juhani Karhumäki, Arto Lepistö, and Donald Sannella (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1016–1028.

Markus Müller-Olm and Helmut Seidl. 2005. A Generic Framework for Interprocedural Analysis of Numerical Properties. In *Proceedings of the 12th International Conference on Static Analysis (SAS'05)*. Springer-Verlag, Berlin, Heidelberg, 235–250. https://doi.org/10.1007/11547662_17

Edgar Pek, Xiaokang Qiu, and P. Madhusudan. 2014. Natural Proofs for Data Structure Manipulation in C Using Separation Logic. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. ACM, New York, NY, USA, 440–451. https://doi.org/10.1145/2594291.2594325

Xiaokang Qiu, Pranav Garg, Andrei Ştefănescu, and Parthasarathy Madhusudan. 2013. Natural Proofs for Structure, Data, and Separation. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*. ACM, New York, NY, USA, 231–242. https://doi.org/10.1145/2491956.2462169

Neil Robertson and Paul D Seymour. 1983. Graph minors. I. Excluding a forest. *Journal of Combinatorial Theory, Series B* 35, 1 (1983), 39–61.

Stefan Schwoon. 2002. *Model-Checking Pushdown Systems.* Ph.D. Thesis. Technische Universität München. http://www.lsv.ens-cachan.fr/Publis/PAPERS/PDF/schwoon-phd02.pdf

D. Seese. 1991. The structure of the models of decidable monadic theories of graphs. *Annals of Pure and Applied Logic* 53, 2 (1991), 169 – 195. https://doi.org/10.1016/0168-0072(91)90054-P