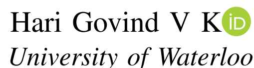
Formal Methods in Computer-Aided Design 2021

Logical Characterization of Coherent Uninterpreted Programs

Hari Govind V K 
University of Waterloo

Sharon Shoham
Tel-Aviv University

Arie Gurfinkel
University of Waterloo

***Abstract*—An uninterpreted program (UP) is a program whose semantics is defined over the theory of uninterpreted functions. This is a common abstraction used in equivalence checking, compiler optimization, and program verification. While simple, the model is sufficiently powerful to encode counter automata, and, hence, undecidable. Recently, a class of UP programs, called coherent, has been proposed and shown to be decidable. We provide an alternative, logical characterization, of this result. Specifically, we show that every coherent program is bisimilar to a finite state system. Moreover, an inductive invariant of a coherent program is representable by a formula whose terms are of depth at most 1. We also show that the original proof, via automata, only applies to programs over unary uninterpreted functions. While this work is purely theoretical, it suggests a novel abstraction that is complete for coherent programs but can be soundly used on *arbitrary* uninterpreted (and partially interpreted) programs.**

I. Introduction

The theory of Equality with Uninterpreted Functions (EUF) is an important fragment of First Order Logic, defined by a set of functions, equality axioms, and congruence axioms. Its satisfiability problem is decidable. It is a core theory of most SMT solvers, used as a glue (or abstraction) for more complex theories. A closely related notion is that of Uninterpreted Programs (UP), where all basic operations are defined by uninterpreted functions. Feasibility of a UP computation is characterized by satisfiability of its path condition in EUF. UPs provide a natural abstraction layer for reasoning about software. They have been used (sometimes without explicitly being named), in equivalence checking of pipelined micro-procesors [1], and equivalence checking of C programs [17]. They also provide the foundations of Global Value Numbering (GVN) optimization in many modern compilers [6], [8], [12].

Unlike EUF, reachability in UP is undecidable. That is, in the *lingua franca* of SMT, the satisfiability of Constrained Horn Clauses over EUF is undecidable. Recently, Mathur et al. [9], have proposed a variant of UPs, called *coherent uninterpreted program* (CUPs). The precise definition of coherence is rather technical (see Def. 3), but intuitively the program is restricted from depending on arbitrarily deep terms. The key result of [9] is to show that both reachability of CUPs and deciding whether an UP is coherent are decidable. This makes CUP an interesting infinite state abstraction with a *decidable* reachability problem.

Unfortunately, as shown by our counterexample in Fig. 4 (and described in Sec. VI), the key construction in [9] is incorrect. More precisely, the proofs of [9] hold only of CUPs restricted to unary functions. In this paper, we address this bug. We provide an alternative (in our view simpler) proof of decidability and extend the results from reachability to arbitrary model checking. The case of non-unary CUPS is much more complex than unary. This is not surprising, since similar complications arise in related results on Uniform Interpolation [4] and Cover [5] for EUF.

Our key result is a logical characterization of CUP. We show that the set of reachable states (i.e., the strongest inductive invariant) of a CUP is definable by an EUF formula, over program variables, with terms of depth at most 1. That is, the most complex term that can appear in the invariant is of the form $v \approx f(\vec{w})$, where v and $\vec{w}$ are program variables, and f a function.

This characterization has several important consequences since the number of such bounded depth formulas is finite. Decidability of reachability, for example, follows trivially by enumerating all possible candidate inductive invariants. More importantly from a practical perspective, it leads to an efficient analysis of *arbitrary* UPs. Take a UP P, and check whether it has a safe inductive invariant of bounded terms. Since the number of terms is finite, this can be done by implicit predicate abstraction [3]. If no invariant is found, and the counterexample is not feasible, then P is not a CUP. At this point, the process either terminates, or another verification round is done with predicates over deeper terms. Crucially, this does not require knowing whether P is a CUP apriori – a problem that itself is shown in [9] to be at least PSPACE.

We extend the results further and show that CUPs are bisimilar to a finite state system, showing, in particular, that arbitrary model checking for CUP (not just reachability) is decidable.

Our proofs are structured around a series of abstractions, illustrated in a commuting diagram in Fig. 1. Our key abstraction is the base abstraction α_b. It forgets terms deeper than depth 1, while maintaining all their consequences (by using additional fresh variables). We show that α_b is sound and complete (i.e., preserves all properties) for CUPs (while, sound, but not complete for UP). It is combined with a cover abstraction $\alpha_{\mathbb{C}}$, that we borrow from [5]. The cover abstraction ensures that reachable states are always expressible over program variables. It serves the purpose of existential quantifier elimination, that is not available for EUF. Finally, a renaming abstraction α_r is a technical tool to bound the occurrences of constants in abstract reachable states.

https://doi.org/10.34727/2021/isbn.978-3-85448-046-4_16

This article is licensed under a Creative Commons Attribution 4.0 International License

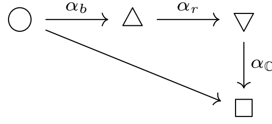


Fig. 1: Sequence of abstractions used in our proofs.

The rest of the paper is structured as follows. We review the necessary background on EUF in Sec. II. We introduce our formalization of UPs and CUPs in Sec. III. Sec. IV presents bisimulation inducing abstractions for UP. Sec. V presents our base abstraction and shows that it induces a bisimulation for CUPs. Sec. VI develops logical characterization for CUPs, presents our decidability results, and shows that a finite state abstraction of CUPs is computable. We conclude the paper in Sec. VII with summary of results and a discussion of open challenges and future work.

II. Background

We assume that the reader is familiar with the basics of First Order Logic (FOL), and the theory of Equality and Uninterpreted Functions (EUF). We use $\Sigma = (\mathcal{C}, \mathcal{F}, \{\approx, \not\approx\})$ to denote a FOL signature with constants $\mathcal{C}$, functions $\mathcal{F}$, and predicates $\{\approx, \not\approx\}$, representing equality and disequality, respectively. A term is a constant or (well-formed) application of a function to terms. A literal is either $x \approx y$ or $x \not\approx y$, where x and y are terms. A formula is a Boolean combination of literals. We assume that all formulas are quantifier free unless stated otherwise. We further assume that all formulas are in Negation Normal Form (NNF), so negation is defined as a shorthand: $\neg(x \approx y) \triangleq x \not\approx y$, and $\neg(x \not\approx y) \triangleq x \approx y$. Throughout the paper, we use $\bowtie$ to indicate a predicate in $\{\approx, \not\approx\}$. For example, $\{x \bowtie y\}$ means $\{x \approx y, x \not\approx y\}$. We write $\bot$ for false, and $\top$ for true. We do not differentiate between sets of literals Γ and their conjunction $(\bigwedge \Gamma)$. We write $\mathrm{depth}(t)$ for the maximal depth of function applications in a term t. We write $\mathcal{T}(\varphi)$, $\mathcal{C}(\varphi)$, and $\mathcal{F}(\varphi)$ for the set of all terms, constants, and functions, in φ, respectively, where φ is either a formula or a collection of formulas. Finally, we write $t[x]$ to mean that the term t contains x as a subterm.

For a formula φ, we write $\Gamma \models \varphi$ if Γ *entails* φ, that is every model of Γ is also a model of φ. For any literal ℓ, we write $\Gamma \vdash \ell$, pronounced ℓ is *derived* from Γ, if ℓ is derivable from Γ by the usual EUF proof system $\mathcal{P}_{EUF}$.[1] By refutational completeness of $\mathcal{P}_{EUF}$, Γ is unsatisfiable iff $\Gamma \vdash \bot$.

Given two EUF formulas φ_1 and φ_2 and a set of constants $V \subseteq \mathcal{C}$, we say that the formulas are V-equivalent, denoted $\varphi_1 \equiv_V \varphi_2$, if, for all quantifier free EUF formulas ψ such that $\mathcal{C}(\psi) \subseteq V$, $(\varphi_1 \wedge \psi) \models \bot$ if and only if $(\varphi_2 \wedge \psi) \models \bot$.

Example 1 Let $\varphi_1 = \{x_1 \approx f(a_0, x_0), y_1 \approx f(b_0, y_0), x_0 \approx y_0\}$, $\varphi_2 = \{x_1 \approx f(a_0, w), y_1 \approx f(b_0, w)\}$, $\varphi_3 = \{x_1 \approx f(a_0, x_0), y_1 \approx f(b_0, y_0)\}$, and $V = \{x_1, y_1, a_0, b_0\}$. Then, $\varphi_1 \equiv_V \varphi_2$ but $\varphi_1 \not\equiv_V \varphi_3$. $\square$

[1]Presented in our companion technical report [7].

$$
\begin{aligned}
\langle stmt \rangle &::= \textbf{skip} \mid \langle var \rangle := \langle var \rangle \mid \langle var \rangle := f(\langle \vec{var} \rangle) \mid \\
&\quad \textbf{assume}\ (\langle cond \rangle) \mid \langle stmt \rangle\ ;\ \langle stmt \rangle \mid \\
&\quad \textbf{if}\ (\langle cond \rangle)\ \textbf{then}\ \langle stmt \rangle\ \textbf{else}\ \langle stmt \rangle \mid \\
&\quad \textbf{while}\ (\langle cond \rangle)\ \langle stmt \rangle \\
\langle cond \rangle &::= \langle var \rangle = \langle var \rangle \mid \langle var \rangle \neq \langle var \rangle \\
\langle var \rangle &::= \mathsf{x} \mid \mathsf{y} \mid \cdots
\end{aligned}
$$

Fig. 2: Syntax of the programming language UPL.

While EUF does not admit quantifier elimination, it does admit elimination of constants while preserving quantifier free consequences. Formally, a *cover* [2], [4], [5] of an EUF formula φ w.r.t. a set of constants V is an EUF formula ψ such that $\mathcal{C}(\psi) \subseteq \mathcal{C}(\varphi) \setminus V$ and $\varphi \equiv_{\mathcal{C}(\varphi)\setminus V} \psi$. By [5], such ψ exists and is unique up to equivalence; we denote it by $\mathbb{C}V \cdot \varphi$.

III. Uninterpreted Programs

An *uninterpreted program (UP)* is a program in the *uninterpreted programming language (UPL)*. The *syntax* of UPL is shown in Fig. 2. Let V denote a fixed set of program variables. We use lower case letters in a special font: x, y, etc. to denote individual variables in V. We write $\vec{\mathsf{y}}$ for a list of program variables. Function symbols are taken from a fixed set $\mathcal{F}$. As in [9], w.l.o.g., UPL does not allow for Boolean combination of conditionals and relational symbols.

The small step symbolic operational semantics of UPL is defined with respect to a FOL signature $\Sigma = (\mathcal{C}, \mathcal{F}, \{\approx, \not\approx\})$ by the rules shown in Fig. 3. A program *configuration* is a triple $\langle s, q, pc \rangle$, where s, called a statement, is a UP being executed, $q : \mathsf{V} \to \mathcal{C}$ is a *state* mapping program variables to constants in $\mathcal{C}$, and pc, called the *path condition*, is a EUF formula over Σ. We use $\mathcal{C}(q) \triangleq \{c \mid \exists \mathsf{v} \cdot q(\mathsf{v}) = c\}$ to denote the set of all constants that represent current variable assignments in q. With abuse of notation, we use $\mathcal{C}(q)$ and q interchangebly. We write $\equiv_q$ to mean $\equiv_{\mathcal{C}(q)}$.

For a state q, we write $q[\mathsf{x} \mapsto x']$ for a state q' that is identical to q, except that it maps x to x'. We write $\langle e, q \rangle \Downarrow v$ to denote that v is the value of the expression e in state q, i.e., the result of substituting each program variable x in e with $q(\mathsf{x})$, and replacing functions and predicates with their FOL counterparts. The value of e is an FOL term or an FOL formula over Σ. For example, $\langle \mathsf{x} = \mathsf{y}, [\mathsf{x} \mapsto x, \mathsf{y} \mapsto y] \rangle \Downarrow x \approx y$.

Given two configurations c and c', we write $c \to c'$ if c reduces to c' using one of the rules in Fig. 3. Note that there is no rule for **skip** – the program terminates once it gets into a configuration $\langle \textbf{skip}, q, pc \rangle$.

Let $\mathcal{C}_0 = \{v_0 \mid \mathsf{v} \in \mathsf{V}\} \subseteq \mathcal{C}$ be a set of initial constants. In the initial state q_0 of a program, every variable is mapped to the corresponding initial constant, i.e., $q_0(\mathsf{v}) = v_0$.

The operational semantics induces, for an UP P, a transition system $\mathcal{S}_P = \langle C, c_0, \mathcal{R} \rangle$, where C is the set of configurations, $c_0 \triangleq \langle P, q_0, \top \rangle$ is the initial configuration, and $\mathcal{R} \triangleq \{(c, c') \mid c \to c'\}$. A configuration c of P is *reachable*

$$\langle \textbf{skip} ; s, q, pc \rangle \to \langle s, q, pc \rangle$$

$$\frac{\langle s_1, q, pc \rangle \to \langle s_1', q', pc' \rangle}{\langle s_1 ; s_2, q, pc \rangle \to \langle s_1' ; s_2, q', pc' \rangle}$$

$$\frac{\langle c, q \rangle \Downarrow v \qquad (pc \wedge v) \not\models \bot}{\langle \textbf{assume}(c), q, pc \rangle \to \langle \textbf{skip}, q, pc \wedge v \rangle}$$

$$\frac{\langle e, q \rangle \Downarrow v \qquad x' \in \mathcal{C}(\Sigma) \text{ is fresh in } pc}{\langle \mathsf{x} := e, q, pc \rangle \to \langle \textbf{skip}, q[\mathsf{x} \mapsto x'], pc \wedge x' = v \rangle}$$

$$\langle \textbf{if } (c) \textbf{ then } s_1 \textbf{ else } s_2, q, pc \rangle \to \langle \textbf{assume}(c) ; s_1, q, pc \rangle$$

$$\langle \textbf{if } (c) \textbf{ then } s_1 \textbf{ else } s_2, q, pc \rangle \to \langle \textbf{assume}(\neg c) ; s_2, q, pc \rangle$$

$$\langle \textbf{while } (c) \; s, q, pc \rangle \to$$
$$\langle \textbf{if } (c) \textbf{ then } (s ; \textbf{while } (c) \; s) \textbf{ else skip}, q, pc \rangle$$

Fig. 3: Small step symbolic operational semantics of UPL, where $\neg c$ denotes $\mathsf{x} \neq \mathsf{y}$ when c is $\mathsf{x} = \mathsf{y}$, and $\mathsf{x} = \mathsf{y}$ when c is $\mathsf{x} \neq \mathsf{y}$.

if c is reachable from c_0 in $\mathcal{S}_P$. We denote the set of all reachable configurations in $\mathcal{S}_P$ using $Reach(\mathcal{S}_P)$. The set of all statements in the semantics of P, including the intermediate statements, are called *locations* of P, and are denoted by $\mathcal{L}(P)$. We often use P and $\mathcal{S}_P$ interchangeably.

Our semantics of UPL differs in some respects from the one in [9]. First, we follow a more traditional small-step operational semantics presentation, by providing semantics rules and the corresponding transition system. However, this does not change the semantics conceptually. More importantly, we ensure that the path condition remains satisfiable in all reachable configurations (by only allowing an assume statement to execute when it results in a satisfiable path condition). We believe this is a more natural choice that is also consistent with what is typically used in other symbolic semantics. UP reachability under our semantics coincides with the definition of [9].

Definition 1 (UP Reachability) Given an UP P, determine whether there exists a state q and a path condition pc s.t., the configuration $\langle \textbf{skip}, q, pc \rangle$ is reachable in P. $\square$

A certificate for unreachability of location s, is an inductive assertion map η (or an inductive invariant) s.t. $\eta(s) = \bot$.

Definition 2 (Inductive Assertion Map) Let $\Sigma_0 \triangleq (\mathcal{C}_0, \mathcal{F}, \{\approx, \not\approx\})$, be restriction of Σ to $\mathcal{C}_0$. An *inductive assertion map* of an UP P, is a map $\eta : \mathcal{L}(P) \to EUF(\Sigma_0)$ s.t. (a) $\eta(P) = \top$, and (b) if $\langle s, q_0, \eta(s) \rangle \to \langle s', q', pc' \rangle$, then $pc' \models (\eta(s')[v_0 \mapsto q'(\mathsf{v}) \mid \mathsf{v} \in \mathsf{V}])$. $\square$

In [9], a special sub-class of UPs has been introduced with a decidable reachability problem.

Definition 3 (Coherent Uninterpreted Program [9]) An UP P is *coherent* (CUP) if all of the reachable configurations

	Program	Assertion
1	`x := t;`	$x_0 \approx t_0$
2	`y := t;`	$x_0 \approx t_0 \wedge y_0 \approx t_0$
3	`while (c != d) {`	$x_0 \approx y_0$
4	`  x := n(x);`	$x_0 \approx n(y_0) \wedge c_0 \not\approx d_0$
5	`  y := n(y);`	$x_0 \approx y_0 \wedge c_0 \not\approx d_0$
6	`  c := n(c);`	$x_0 \approx y_0$
7	`};`	
8	`x := f(a, x);`	$x_0 \approx f(a_0, y_0) \wedge c_0 \approx d_0$
9	`y := f(b, y);`	$(a_0 \approx b_0 \Rightarrow x_0 \approx y_0) \wedge c_0 \approx d_0$
10	`assume(a == b);`	$a_0 \approx b_0 \wedge x_0 \approx y_0 \wedge c_0 \approx d_0$
11	`assume(x != y);`	$\bot$

Fig. 4: An example CUP program and its inductive assertions.

of P satisfy the following two properties:

Memoizing for any configuration $\langle \mathsf{x} := f(\vec{\mathsf{y}}), q, pc \rangle$, if there is a term $t \in \mathcal{T}(pc)$ s.t. $pc \models t \approx f(q(\vec{\mathsf{y}}))$, then there is $\mathsf{v} \in \mathsf{V}$ s.t. $pc \models q(\mathsf{v}) \approx t$.

Early assume for any configuration $\langle \textbf{assume}(\mathsf{x} = \mathsf{y}), q, pc \rangle$, if there is a term $t \in \mathcal{T}(pc)$ s.t. $pc \models t \approx s$ where s is a superterm of either $q(\mathsf{x})$ or $q(\mathsf{y})$, then, there is $\mathsf{v} \in \mathsf{V}$ s.t. $pc \models q(\mathsf{v}) \approx t$. $\square$

Intuitively, memoization ensures that if a term is recomputed, then it is already stored in a program variable; early assumes ensures that whenever an equality between variables is assumed, any of their superterms that was ever computed is still stored in a program variable. Note that unlike the original definition of CUP in [9], we do not require the notion of an *execution*. The path condition accumulates the history of the execution in a configuration, which is sufficient.

Example 2 An example of a CUP is shown in Fig. 4. Some reachable states in the first iteration of the loop are shown below, where line numbers are used as locations, and pc_i stands for the path condition at line i:

$$\langle 2, q_0[\mathsf{x} \mapsto x_1, \mathsf{y} \mapsto y_1], x_1 \approx t_0 \wedge y_1 \approx t_0 \rangle$$
$$\langle 6, q_0[\mathsf{x} \mapsto x_2, \mathsf{y} \mapsto y_2, \mathsf{c} \mapsto c_1], pc_2 \wedge$$
$$c_0 \not\approx d_0 \wedge x_2 \approx n(x_1) \wedge y_2 \approx n(y_1) \wedge c_1 \approx n(c_0) \rangle$$
$$\langle 9, q_0[\mathsf{x} \mapsto x_3, \mathsf{y} \mapsto y_3, \mathsf{c} \mapsto c_1] \rangle, pc_6 \wedge$$
$$c_1 \approx d_0 \wedge x_3 \approx f(a_0, x_2) \wedge y_3 \approx f(b_0, y_2) \rangle$$

The program is coherent because (a) no term is recomputed; (b) for the assume at line 10, the only superterms of a_0 and b_0 are $f(a_0, x_n)$ and $f(b_0, y_n)$, and they are stored in x and y, respectively; and (c) for the assume $(c_n = d_0)$ introduced by the exit condition of the while loop, no superterms of c_n, d_0 are ever computed. The program does not reduce to **skip** (i.e., it does not reach a final configuration). Its inductive assertion map is shown in Fig. 4 (right). $\square$

Note that UP are closely related, but are not equivalent, to the Herbrand programs of [12]. While Herbrand programs use the syntax of UPL, they are interpreted over a fixed universe of Herbrand terms. In particular, in Herbrand programs $f(x) \approx g(x)$ is always false (since $f(x)$ and $g(x)$ have different top-level functions), while in UP, it is satisfiable.

IV. ABSTRACTION AND BISIMULATION FOR UP

In this section, we review abstractions for transition systems. We then define two abstraction for UP: cover and renaming, and show that they induce bisimulation. That is, for UP, these abstractions preserve all properties. Finally, we show a simple logical characterization result for UP to set the stage for our main results in the following sections.

Definition 4 Given a transition system $\mathcal{S} = (C, c_0, \mathcal{R})$ and a (possibly partial) abstraction function $\sharp : C \to C$, the induced *abstract transition system* is $\sharp(\mathcal{S}) = (C, c_0^\sharp, \mathcal{R}^\sharp)$, where

$$c_0^\sharp \triangleq \sharp(c_0)$$
$$\mathcal{R}^\sharp \triangleq \{(c_\sharp, c'_\sharp) \mid \exists c, c'.\ c \to c' \ \wedge\ c_\sharp = \sharp(c) \ \wedge\ c'_\sharp = \sharp(c')\}$$

We write $c \to^\sharp c'$ when $(c, c') \in \mathcal{R}^\sharp$. Note that $\sharp$ must be defined for c_0. □

Throughout the paper, we construct several abstract transition systems. All transition systems considered are *attentive*. Intuitively, this means that their transitions do not distinguish between configurations that have q-equivalent path conditions. We say that two configurations $c_1 = \langle s, q, pc_1\rangle$ and $c_2 = \langle s, q, pc_2\rangle$ are equivalent, denoted $c_1 \equiv c_2$ if $pc_1 \equiv_q pc_2$.

Definition 5 (Attentive TS) A transition system $\mathcal{S} = (C, c_0, \mathcal{R})$ is *attentive* if for any two configurations $c_1, c_2 \in C$ s.t. $c_1 \equiv c_2$, if there exists $c'_1 \in C$ s.t. $(c_1, c'_1) \in \mathcal{R}$, then there exists $c'_2 \in C$, s.t. $(c_2, c'_2) \in \mathcal{R}$ and $c'_1 \equiv c'_2$ and vice versa. □

Weak, respectively strong, preservation of properties between the abstract and the concrete transition systems are ensured by the notions of *simulation*, respectively *bisimulation*.

Definition 6 ([11]) Let $\mathcal{S} = (C, c_0, \mathcal{R})$ and $\sharp(\mathcal{S}) = (C, c_0^\sharp, \mathcal{R}^\sharp)$ be transition systems. A relation $\rho \subseteq C \times C$ is a *simulation* from $\mathcal{S}$ to $\sharp(\mathcal{S})$, if for every $(c, c_\sharp) \in \rho$:

- if $c \to c'$ then there exists $c'_\sharp$ such that $c_\sharp \to^\sharp c'_\sharp$ and $(c', c'_\sharp) \in \rho$.

$\rho \subseteq C \times C$ is a *bisimulation* from $\mathcal{S}$ to $\sharp(\mathcal{S})$ if ρ is a simulation from $\mathcal{S}$ to $\sharp(\mathcal{S})$ and $\rho^{-1} \triangleq \{(c_\sharp, c) \mid (c, c_\sharp) \in \rho\}$ is a simulation from $\sharp(\mathcal{S})$ to $\mathcal{S}$. We say that $\sharp(\mathcal{S})$ *simulates*, respectively *is bisimilar to*, $\mathcal{S}$ if there exists a simulation, respectively, a bisimulation, ρ from $\mathcal{S}$ to $\sharp(\mathcal{S})$ such that $(c_0, c_0^\sharp) \in \rho$. □

We say that a bisimulation $\rho \subseteq C \times C$ is *finite* if its range, $\{\rho(c) \mid c \in C\}$, is finite. A finite bisimulation relates a (possibly infinite) transition system with a finite one.

Next, we define two abstractions for UP programs and show that they result in bisimilar abstract transition systems. The first abstraction eliminates all constants that are not assigned to program variables from the path condition, using the cover operation. The second abstraction renames the constants assigned to program variables back to the initial constants $\mathcal{C}_0$. Both abstractions together ensure that all reachable configurations in the abstract transition system are defined over Σ_0 (i.e., the only constants that appear in states, as well as in path conditions, are from $\mathcal{C}_0$). There may still be infinitely many such configurations since the depth of terms may be unbounded. We show that whenever the obtained abstract transition system has finitely many reachable configurations, the concrete one has an inductive assertion map that characterizes the set of reachable configurations.

Definition 7 (Cover abstraction) The cover abstraction function $\alpha_{\mathbb{C}} : C \to C$ is defined by

$$\alpha_{\mathbb{C}}(\langle s, q, pc\rangle) \triangleq \langle s, q, \mathbb{C}(\mathcal{C} \setminus \mathcal{C}(q)) \cdot pc\rangle$$ □

Since $pc \equiv_q \mathbb{C}(\mathcal{C} \setminus \mathcal{C}(q)) \cdot pc$, the cover abstraction also results in a bisimilar abstract transition system.

Theorem 1 *For any attentive transition system $\mathcal{S} = (C, c_0, \mathcal{R})$, the relation $\rho = \{(c, \alpha_{\mathbb{C}}(c)) \mid c \in Reach(\mathcal{S})\}$ is a bisimulation from $\mathcal{S}$ to $\alpha_{\mathbb{C}}(\mathcal{S})$.* □

To introduce the renaming abstraction, we need some notation. Given a quantifier free formula φ, constants $a, b \in \mathcal{C}(\varphi)$ such that $a \neq b$, let $\varphi[a \rightarrowtail b]$ denote $\varphi[b \mapsto x][a \mapsto b]$, where x is a constant not in $\mathcal{C}(\varphi)$. For example, if $\varphi = (a \approx c \wedge b \approx d)$, $\varphi[a \rightarrowtail b] = (b \approx c \wedge x \approx d)$.

Given a path condition pc and a state q, let $r_0(pc, q)$ denote the formula obtained by renaming all constants in $\mathcal{C}(q)$ using their initial values. $r_0(pc, q) = pc[q(\mathsf{v}) \rightarrowtail v_0]$ for all $\mathsf{v} \in \mathsf{V}$ such that $q(\mathsf{v}) \neq v_0$.

Definition 8 (Renaming abstraction) The renaming abstraction function $\alpha_r : C \to C$ is defined by

$$\alpha_r(\langle s, q, pc\rangle) \triangleq \langle s, q_0, r_0(pc, q)\rangle$$ □

Theorem 2 *For any attentive transition system $\mathcal{S} = (C, c_0, \mathcal{R})$, the relation $\rho = \{(c, \alpha_r(c)) \mid c \in Reach(\mathcal{S})\}$ is a bisimulation from $\mathcal{S}$ to $\alpha_r(\mathcal{S})$.* □

Finally, we denote by $\alpha_{\mathbb{C},r}$ the composition of the renaming and cover abstractions: $\alpha_{\mathbb{C},r} \triangleq \alpha_{\mathbb{C}} \circ \alpha_r$ (i.e., $\alpha_{\mathbb{C},r}(c) = \alpha_r(\alpha_{\mathbb{C}}(c))$). Since the composition of bisimulation relations is also a bisimulation, $\alpha_{\mathbb{C},r}(\mathcal{S})$ is bisimilar to $\mathcal{S}$.

Theorem 3 (Logical Characterization of UP) *If $\alpha_{\mathbb{C},r}$ induces a finite bisimulation on an UP P, then, there exists an inductive assertion map η for P that characterizes the reachable configurations of P.* □

PROOF Define $\eta(s) \triangleq \bigvee\{pc \mid \langle s, q, pc\rangle \in Reach(\alpha_{\mathbb{C},r}(P))\}$. Then, $\eta(s)$ is such an inductive assertion map. ■

Intuitively, Thm. 3 says that inductive invariant of UP, whenever it exists, can be described using EUF formulas over program variables. That is, any extra variables that are added to the path condition during program execution can be abstracted away (specifically, using the cover abstraction). There are, of course, infinitely many such invariants since the depth of terms is not bounded (only constants occurring in them). In the sequel, we systematically construct a similar result for CUP.

V. Bismulation of CUP

The first step in extending Thm. 3 to CUP is to design an abstraction function that bounds the depth of terms that appear in any reachable (abstract) state. It is easy to design such a function while maintaining soundness – simply forget literals that have terms that are too deep. However, we want to maintain precision as well. That is, we want the abstract transition system to be bisimilar to the concrete one. Just like cover abstraction, the base abstraction function also eliminates all constants that are not assigned to program variables. Unlike cover abstraction, the base abstraction does not maintain $\mathcal{C}(q)$-equivalence of the path conditions, but, rather, forgets most literals that cannot be expressed over program variables.

In this section, we focus on the definition of the base abstraction and prove that it induces bisimulation for CUP. This result is used in Sec. VI, to logically characterize CUPs.

Intuitively, the base abstraction "truncates" the congruence graph induced by a path condition in nodes that have no representative in the set of constants assigned to the program variables (V in the following definition), and assigns to the truncated nodes fresh constants (from W in the following definition).

Congruence closure procedures for EUF use a *congruence graph* to concisely represent the deductive closure of a set of EUF literals [15], [16]. Here, we use a logical characterization of a congruence graph, called a V*-basis*. Let Γ be a set of EUF literals. A triple $\langle W, \beta, \delta\rangle$ is a V-basis of Γ relative to a set of constants V, written $\langle W, \beta, \delta\rangle \in \mathrm{base}(\Gamma, V)$, iff (a) W is a set of fresh constants not in $\mathcal{C}(\Gamma)$, and β and δ are conjunctions of EUF literals; (b) $(\exists W \cdot \beta \wedge \delta) \equiv \Gamma$; (c) $\beta \triangleq \beta_{\approx} \cup \beta_{\not\approx} \cup \beta_{\mathcal{F}}$ and $\delta \triangleq \delta_{\approx} \cup \delta_{\not\approx} \cup \delta_{\mathcal{F}}$, where

$$\beta_{\approx} \subseteq \{u \approx v \mid u, v \in V\} \qquad \beta_{\not\approx} \subseteq \{u \not\approx v \mid u, v \in V\}$$
$$\beta_{\mathcal{F}} \subseteq \{v \approx f(\vec{w}) \mid v \in V, \vec{w} \subseteq V \cup W, \vec{w} \cap V \neq \emptyset\}$$
$$\delta_{\approx} \subseteq \{w \approx u \mid w \in V \cup W, u \notin V \cup W\}$$
$$\delta_{\not\approx} \subseteq \{u \not\approx w \mid u \in W, w \in W \cup V\}$$
$$\delta_{\mathcal{F}} \subseteq \{v \approx f(\vec{w}) \mid v, \vec{w} \subseteq V \cup W, v \in V \Rightarrow \vec{w} \subseteq W\}$$

(d) $\beta \wedge \delta \nvdash v \approx w$ for any $v \in V$, $w \in W$; and (e) $\beta \wedge \delta \nvdash w_1 \approx w_2$ for any $w_1, w_2 \in W$ s.t. $w_1 \neq w_2$.

Note that we represent both equalities and disequalities in the V-basis as common in implementations (but not in the theoretical presentations) of the congruence closure algorithm. Intuitively, V are constants in $\mathcal{C}(\Gamma)$ that represent equivalence classes in Γ, and W are constants added to represent equivalence classes that do not have a representative in V. A V-basis, of any satisfiable set Γ, is unique up to renaming of constants in W and ordering of equalities between constants in V.

Example 3 Let $\Gamma = \{x \approx f(a, v_1), y \approx f(b, v_2), v_1 \approx v_2\}$ and $V = \{a, b, x, y\}$. A V-basis of Γ is $\langle W, \beta, \delta\rangle$, where $W = \{w\}$, $\beta = \{x \approx f(a, w), y \approx f(b, w)\}$, $\delta = \{w \approx v_1, w \approx v_2\}$. Renaming w to w' is a different V-basis: $\langle W', \beta', \delta'\rangle \in \mathrm{base}(\Gamma, V)$ where $W' = \{w'\}$, $\beta' = \beta[w \mapsto w']$ and $\delta' = \delta[w \mapsto w']$.

As another example, consider $\Gamma = \{x \approx f(a, p), x \approx f(a, n(p)), y = f(b, p), y = f(c, n(p))\}$ and $V = \{x, y, a, b, c\}$. A V-basis of Γ is $\langle W, \beta, \delta\rangle$, where $W = \{w_0, w_1\}$, $\delta_2 = \{w_0 \approx p, w_1 \approx n(w_0)\}$, and

$$\beta_2 = \begin{Bmatrix} x \approx f(a, w_0) & x \approx f(a, w_1) \\ y \approx f(b, w_0) & y \approx f(c, w_1) \end{Bmatrix}$$ □

While a basis maintains all consequences of Γ (since $(\exists W \cdot \beta \wedge \delta) \equiv \Gamma$), the V-base abstraction of Γ, defined next, is weaker. It preserves consequences of β only:

Definition 9 (V-base abstraction) The V-base abstraction α_V for a set of constants V, is a function between sets of literals s.t. for any sets of literals Γ and Γ':

(1) $\alpha_V(\Gamma) \triangleq \beta$, where $\langle W, \beta, \delta\rangle \in \mathrm{base}(\varphi, V)$,
(2) if there exists a β s.t. $\langle W_1, \beta, \delta_1\rangle \in \mathrm{base}(\Gamma, V)$ and $\langle W_2, \beta, \delta_2\rangle \in \mathrm{base}(\Gamma', V)$, then $\alpha_V(\Gamma) = \alpha_V(\Gamma')$. □

The second requirement of Def. 9 ensures that two formulas that have the same V-consequences, have the same V-abstraction. For example, for a set of constants $V = \{u, v\}$, the formulas $\varphi_1 = \{v \approx f(u, x)\}$ and $\varphi_2 = \{v \approx f(u, y)\}$, have the same V-base abstraction: $v \approx f(u, w)$. Note that at this point, we only require that α_V is well defined (for example, it does not have to be computable.)

We now extend V-base abstraction to program configuration, calling it simply *base abstraction*, since the set of preserved constants is determined by the configuration:

Definition 10 (Base abstraction) The base abstraction $\alpha_b : C \to C$ is defined for configurations $\langle s, q, pc\rangle \in C$, where pc is a *conjunction* of literals: $\alpha_b(\langle s, q, pc\rangle) \triangleq \langle s, q, \alpha_{\mathcal{C}(q)}(pc)\rangle$.□

Namely, the base abstraction $\alpha_{\mathcal{C}(q)}$ applied to the path condition is determined by the state q in the configuration. We often write $\alpha_q(\varphi)$ as a shorthand for $\alpha_{\mathcal{C}(q)}(\varphi)$.

We are now in position to state the main result of this section. Given a CUP P, the abstract transition system $\alpha_b(\mathcal{S}_P) = (C, c_0^{\alpha_b}, \mathcal{R}^{\alpha_b})$ is bisimilar to the concrete transition system $\mathcal{S}_P = (C, c_0, \mathcal{R})$. Note that at this point, we do not claim that $\alpha_b(\mathcal{S}_P)$ is finite, or that it is computable. We focus only on the fact that the literals that are forgotten by the base abstraction do not matter for any future transitions. The key technical step is summarized in the following theorem:

Theorem 4 *Let $\langle s, q, pc\rangle$ be a reachable configuration of a CUP P. Then,*

(1) $\langle s, q, pc\rangle \to \langle s', q', pc \wedge pc'\rangle$ iff $\langle s, q, \alpha_q(pc)\rangle \to \langle s', q', \alpha_q(pc) \wedge pc'\rangle$, and
(2) $\alpha_{q'}(pc \wedge pc') = \alpha_{q'}(\alpha_q(pc) \wedge pc')$. □

The proof of Thm. 4 is not complicated, but it is tedious and technical. It depends on many basic properties of EUF. We summarize the key results that we require in the following lemmas. The proofs of the lemmas are provided in our companion technical report [7].

We begin by defining a *purifier* – a set of constants sufficient to represent a set of EUF literals with terms of depth one.

Definition 11 (Purifier) We say that a set of constants V is a *purifier* of a constant a in a set of literals Γ, if $a \in V$ and for every term $t \in \mathcal{T}(\Gamma)$ s.t. $\Gamma \vdash t \approx s[a]$, $\exists v \in V$ s.t. $\Gamma \vdash v \approx t$.□

For example, if $\Gamma = \{c \approx f(a), d \approx f(b), d \not\approx e\}$. Then, $V = \{a, b, c\}$ is a purifier for a, but not a purifier for b, even though $b \in V$.

In all the following lemmas, Γ, φ_1, φ_2 are sets of literals; V a set constants; $a, b \in \mathcal{C}(\Gamma)$; $u, v, x, y \in V$; V is a purifier for $\{x, y\}$ in Γ, φ_1, and in φ_2; $\beta = \alpha_V(\Gamma)$; and $\alpha_V(\varphi_1) = \alpha_V(\varphi_2)$.

Lemma 1 says that anything newly derivable from Γ and a new equality $a \approx b$ is derivable using superterms of a and b:

Lemma 1 *Let t_1 and t_2 be two terms in $\mathcal{T}(\Sigma)$ s.t. $\Gamma \not\vdash (t_1 \approx t_2)$. Then, $(\Gamma \wedge a \approx b) \vdash (t_1 \approx t_2)$, for some constants a and b in $\mathcal{C}(\Gamma)$, iff there are two superterms, $s_1[a]$ and $s_2[b]$, of a and b, respectively, s.t. (i) $\Gamma \vdash (t_1 \approx s_1[a])$, (ii) $\Gamma \vdash (t_2 \approx s_2[b])$, and (iii) $(\Gamma \wedge a \approx b) \vdash (s_1[a] \approx s_2[b])$.*

Lemma 2 and Lemma 3 say that all consequences of Γ that are relevant to V are present in $\beta = \alpha_V(\Gamma)$ as well.

Lemma 2 $(\Gamma \wedge x \approx y \vdash u \approx v) \iff (\beta \wedge x \approx y \vdash u \approx v)$.

Lemma 3 $(\Gamma \wedge x \approx y \vdash u \not\approx v) \iff (\beta \wedge x \approx y \vdash u \not\approx v)$.

Lemma 4 says that $\beta = \alpha_V(\Gamma)$ can be described using terms of depth one using constants in V.

Lemma 4 *V is a purifier for $x \in V$ in β.*

Lemma 5 says that α_V is idempotent.

Lemma 5 $\alpha_V(\Gamma) = \alpha_V(\alpha_V(\Gamma))$.

Lemma 6 and Lemma 7 say that α_V preserves addition of new literals and dropping of constants.

Lemma 6 $\alpha_V(\varphi_1 \wedge x \approx y) = \alpha_V(\varphi_2 \wedge x \approx y)$.

Lemma 7 *If $U \subseteq V$, then*

$$(\alpha_V(\varphi_1) = \alpha_V(\varphi_2)) \Rightarrow (\alpha_U(\varphi_1) = \alpha_U(\varphi_2))$$

Lemma 8 extends the preservation results to disequalities. V is a set of constants, $x, y \in V$. V is not required to be a purifier (as it was in the previous lemmas).

Lemma 8 $\alpha_V(\varphi_1 \wedge x \not\approx y) = \alpha_V(\varphi_2 \wedge x \not\approx y)$.

Lemma 9 extends the preservation results for equalities involving a fresh constant x' s.t. $x' \notin \mathcal{C}(\varphi_1) \cup \mathcal{C}(\varphi_2)$. $\vec{y} \subseteq V$, $V' = V \cup \{x'\}$, and $f(\vec{y})$ be a term s.t there does not exists a term $t \in \mathcal{T}(\varphi_1) \cup \mathcal{T}(\varphi_2)$ s.t. $\varphi_1 \vdash t \approx f(\vec{y})$ or $\varphi_2 \vdash t \approx f(\vec{y})$.

Lemma 9

$$\alpha_{V'}(\varphi_1 \wedge x' \approx y) = \alpha_{V'}(\varphi_2 \wedge x' \approx y) \quad (1)$$

$$\alpha_{V'}(\varphi_1 \wedge x' \approx f(\vec{y})) = \alpha_{V'}(\varphi_2 \wedge x' \approx f(\vec{y})) \quad (2)$$

We are now ready to present the proof of Thm. 4:

PROOF (THEOREM 4) In the proof, we use $x = q(\mathsf{x})$, and $y = q(\mathsf{y})$. For part (1), we only show the proof for $s = \textbf{assume}(\mathsf{x} \bowtie \mathsf{y})$ since the other cases are trivial.

The only-if direction follows since $\alpha_q(pc)$ is weaker than pc. For the if direction, $pc \not\vdash \bot$ since it is part of a reachable configuration. Then, there are two cases:

- case $s = \textbf{assume}(\mathsf{x} = \mathsf{y})$. Assume $(pc \wedge x \approx y) \models \bot$. Then, $(pc \wedge x \approx y) \vdash t_1 \approx t_2$ and $pc \vdash t_1 \not\approx t_2$ for some $t_1, t_2 \in \mathcal{T}(pc)$. By Lemma 1, in any new equality $(t_1 \approx t_2)$ that is implied by $pc \wedge (x \approx y)$ (but not by pc), t_1 and t_2 are equivalent (in pc) to superterms of x or y. By the early assume property of CUP, $\mathcal{C}(q)$ purifies $\{x, y\}$ in pc. Therefore, every superterm of x or y is equivalent (in pc) to some constant in $\mathcal{C}(q)$. Thus, $(pc \wedge x \approx y) \vdash u \approx v$ and $(pc \wedge x \approx y) \vdash u \not\approx v$ for some $u, v \in \mathcal{C}(q)$. By Lemma 2, $(\alpha_q(pc) \wedge x \approx y) \vdash u \approx v$. By Lemma 3, $(\alpha_q(pc) \wedge x \approx y) \vdash u \not\approx v$. Thus, $(\alpha_q(pc) \wedge x \approx y) \models \bot$.
- case $s = \textbf{assume}(\mathsf{x} \neq \mathsf{y})$. $(pc \wedge x \not\approx y) \models \bot$ if and only if $pc \vdash x \approx y$. Since $x, y \in \mathcal{C}(q)$, $\alpha_q(pc) \vdash x \approx y$. ■

For part (2), we only show the cases for assume and assignment statements, the other cases are trivial.

- case $s = \textbf{assume}(\mathsf{x} = \mathsf{y})$, Since $q' = q$, we need to show that $\alpha_q(pc \wedge x \approx y) = \alpha_q(\alpha_q(pc) \wedge x \approx y)$. From the early assumes property, $\mathcal{C}(q)$ purifies $\{x, y\}$ in pc. By Lemma 4, $\mathcal{C}(q)$ purifies $\{x, y\}$ in $\alpha_q(pc)$ as well. By Lemma 5, $\alpha_q(pc) = \alpha_q(\alpha_q(pc))$. By Lemma 6, $\alpha_q(pc \wedge x \approx y) = \alpha_q(\alpha_q(pc) \wedge x \approx y)$.
- case $s = \textbf{assume}(\mathsf{x} \neq \mathsf{y})$, Since $q' = q$, we need to show that $\alpha_q(pc \wedge x \not\approx y) = \alpha_q(\alpha_q(pc) \wedge x \not\approx y)$. By Lemma 5, $\alpha_q(pc) = \alpha_q(\alpha_q(pc))$. By Lemma 8, $\alpha_q(pc \wedge x \not\approx y) = \alpha_q(\alpha_q(pc) \wedge x \not\approx y)$.
- case $s = \mathsf{x}{:=}\mathsf{y}$. W.l.o.g., assume $q' = q[\mathsf{x} \mapsto x']$, for some constant $x' \notin \mathcal{C}(pc)$. By Lemma 5, $\alpha_q(pc) = \alpha_q(\alpha_q(pc))$. By Lemma 9 (case 1), $\alpha_{\mathcal{C}(q) \cup \{x'\}}(pc \wedge x' \approx y) = \alpha_{\mathcal{C}(q) \cup \{x'\}}(\alpha_q(pc) \wedge x' \approx y)$. By Lemma 7, $\alpha_{q'}(pc \wedge x' \approx y) = \alpha_{q'}(\alpha_q(pc) \wedge x' \approx y)$, since $\mathcal{C}(q') \subseteq (\mathcal{C}(q) \cup \{x'\})$.
- case $s = \mathsf{x} := f(\vec{y})$. W.l.o.g., $q' = q[\mathsf{x} \mapsto x']$ for some constant $x' \notin \mathcal{C}(pc)$. There are two cases: (a) there is a term $t \in \mathcal{T}(pc)$ s.t. $pc \vdash t \approx f(\vec{y})$, (b) there is no such term t.

(a) By the memoizing property of CUP, there is a program variable z s.t. $q(\mathsf{z}) = z$ and $pc \vdash z \approx f(\vec{y})$. Therefore, by definition of α_q, $\alpha_q(pc) \vdash z \approx f(\vec{y})$. The rest of the proof is identical to the case of $s = \mathsf{x} := \mathsf{z}$.

(b) Since there is no term $t \in \mathcal{T}(pc)$ s.t. $pc \vdash t \approx f(\vec{y})$, there is also no such term in $\mathcal{T}(\alpha_q(pc))$ as well. By Lemma 5, $\alpha_q(pc) = \alpha_q(\alpha_q(pc))$. By Lemma 9 (case 2), $\alpha_{\mathcal{C}(q) \cup \{x'\}}(pc \wedge x \approx f(\vec{y})) = \alpha_{\mathcal{C}(q) \cup \{x'\}}(\alpha_q(pc) \wedge x \approx f(\vec{y}))$. By Lemma 7, $\alpha_{q'}(pc \wedge x \approx f(\vec{y})) = \alpha_{q'}(\alpha_q(pc) \wedge x \approx f(\vec{y}))$ since $\mathcal{C}(q') \subseteq (\mathcal{C}(q) \cup \{x'\})$. ■

Corollary 1 *For a CUP P, the relation $\rho \triangleq \{(c, \alpha_b(c)) \mid c \in Reach(\mathcal{S}_P)\}$ is a bisimulation from $\mathcal{S}_P$ to $\alpha_b(\mathcal{S}_P)$.* □

Note that for an arbitrary UP, α_b induces a simulation (since α_b only weakens path conditions).

By construction, for any configuration in an abstract system constructed using α_b, the path condition will be at most depth-1. In Sec. VI, we use this property to build a logical characterization of CUP and show that reachability of CUP programs is decidable.

VI. Logical Characterization of CUP

In this section, we show that for any CUP program P, all reachable configurations of P can be characterized using formulas in EUF, whose size is bounded by the number of program variables in P.

Theorem 5 (Logical Characterization of CUP) *For any CUP P, there exists an inductive assertion map η, ranging over EUF formulas of depth at most 1, that characterizes the reachable configurations of P.* □

The first step in the proof is to compose the renaming abstraction (Def. 8) with the base abstraction (Def. 10). We denote the composition with $\alpha_{b,r}$, i.e., $\alpha_{b,r} \triangleq \alpha_b \circ \alpha_r$. Cor. 1 and Thm. 2 ensures that $\alpha_{b,r}$ is sound and complete for CUP. We split the rest of the proof into two cases: CUPs restricted to unary functions, called 1-CUP, followed by arbitrary CUPs.

Proof (Thm. 5, 1-CUP) Let Σ^1 be a signature containing function symbols of arity atmost 1, $\Sigma^1 \triangleq (\mathcal{C}, \mathcal{F}^1, \{\approx, \not\approx\})$. Let Γ be a set of literals in Σ^1 and V be a set of constants. By the definition of V-base abstraction (Def. 9), $\alpha_V(\Gamma) = \beta_{\approx} \wedge \beta_{\not\approx} \wedge \beta_{\mathcal{F}}$. $\beta_{\approx}$ and $\beta_{\not\approx}$ are over constants in V. $\beta_{\mathcal{F}}$ contains two types of literals: $\beta_{\mathcal{F}_V}$ and $\beta_{\mathcal{F}_W}$. $\beta_{\mathcal{F}_V}$ are 1 depth literals over constants in V. $\beta_{\mathcal{F}_W}$ are literals of the form $v \approx f(\vec{w})$ where $v \in V$ and $\vec{w}$ is a list of constants, at least one of which is in V: $\vec{w} \cap V \neq \emptyset$ and $\vec{w} \not\subseteq V$. Since Γ can only have unary functions, $\beta_{\mathcal{F}_W} = \emptyset$. Therefore, all literals in $\alpha_V(\Gamma)$ are of depth at most 1 and only contain constants from V. Hence, there are only finitely many configurations in $\alpha_{b,r}(\mathcal{S}_P)$. Therefore,

$$\eta(s) \triangleq \bigvee \{pc \mid \langle s, q_0, pc\rangle \in \mathit{Reach}(\alpha_{b,r}(\mathcal{S}_P))\}$$

is an inductive assertion map, ranging over formulas for depth at most 1, that characterizes the reachable configurations of P. Moreover, the size of each disjunct in $\eta(s)$ is polynomial in the number of program variables and functions in P. ■

An interesting consequence of the above proof is that, for 1-CUPs, α_b is efficiently computable (since, $\beta_{\mathcal{F}_W} = \emptyset$). Thus, the transition system $\alpha_{b,r}(\mathcal{S}_P)$ is finite, and can be constructed on-the-fly. Hence, reachability of 1-CUP is in PSPACE.

Proof (Thm. 5, general case) In general, CUP programs can contain unary and non-unary functions. Therefore, the V-base abstraction (Def. 9) may introduce fresh constants. We use the cover abstraction (Def. 7) to eliminate these fresh constants. By Thm. 1, $\alpha_{\mathbb{C}}(\alpha_{b,r}(\mathcal{S}_P))$ is bisimilar to $\alpha_{b,r}(\mathcal{S}_P)$. Notice that all the fresh constants introduced by the V-base abstraction are arguments to function applications. Therefore, all consequences of eliminating the fresh constants are Horn clauses of the form $\bigwedge_i (x_i \approx y_i) \Rightarrow x \approx y$, where $x_i, y_i, x, y \in \mathcal{C}_0$. Since V-basis is of depth at most 1, cover of the V-basis is also of depth at most 1. Since there are only finitely many formulas of depth at most 1 over $\mathcal{C}_0$, $\alpha_{\mathbb{C}}(\alpha_{b,r}(\mathcal{S}_P))$ has only finitely many configurations. Hence,

$$\eta(s) \triangleq \bigvee \{pc \mid \langle s, q_0, pc\rangle \in \mathit{Reach}(\alpha_{\mathbb{C}}(\alpha_{b,r}(\mathcal{S}_P))\}$$

is an inductive assertion map that characterizes the reachable configurations of P and ranges over depth-1 formulas. ■

Consider the CUP shown in Fig. 4. At line 9, the $\alpha_{b,r}$ abstraction produces the following abstract pc: $x_0 \approx f(a_0, w) \wedge y_0 \approx f(b_0, w) \wedge c_0 \approx d_0$. Using cover to eliminate the constant w gives us $\mathbb{C}w \cdot pc = (a_0 \approx b_0 \Rightarrow x_0 \approx y_0) \wedge c_0 \approx d_0$, which is exactly the invariant assertion mapping $\eta(9)$ at line 9.

We have seen that all CUP programs have an inductive assertion map that characterizes their reachable configurations and ranges over a finite set of formulas. Therefore,

Corollary 2 *CUP reachability is decidable.* □

A. Relationship to [9]

In [9], Cor. 2 is proven by constructing a deterministic finite automaton that accepts all *feasible* coherent executions.[2] However, the construction fails for the executions of the CUP in Fig. 4: the execution that reaches a terminal configuration is infeasible, but it is (wrongfully) accepted by the automaton. Intuitively, the reason is that the automaton is deterministic and its states are not sufficiently expressive. The states of the automaton keep track of equalities between program variables (which correspond to $\beta_{\approx}$ in our abstraction), disequalities between them ($\beta_{\not\approx}$ in our case), and partial function interpretations ($\beta_{\mathcal{F}}$). However, the partial function interpretations are restricted to $\beta_{\mathcal{F}_V}$, i.e., do not allow auxiliary constants that are not assigned to program variables. Thus, they are unable to keep track of $x_0 \approx f(a_0, w) \wedge y_0 \approx f(b_0, w) \wedge c_0 \approx d_0$ in line 9, which is essential for showing infeasibility of the execution. Eliminating the auxiliary constants, as we do in the cover abstraction, does not remedy the situation since it introduces a disjunction $(a_0 \not\approx b_0 \wedge c_0 \approx d_0) \vee (x_0 \approx y_0 \wedge c_0 \approx d_0)$, which the deterministic automaton does not capture.

B. Computing a Finite Abstraction

We have shown that CUP programs are bisimilar to finite state systems. However, all our proofs depend on α_b, which was not assumed to be computable. In this section, we show how to implement α_b, and, thereby, show how to compute a finite state system that is bisimilar to a CUP program. Note that our prior results are independent of this section.

The main difficulty is in naming the fresh constants, which we always refer to as W, that are introduced by the base abstraction. Since we require that base abstraction is canonical, the naming has to be unique. Furthermore, we have to show that the number of such W constants is bounded. We solve both of these problems by proposing a deterministic naming scheme. The scheme is determined by a normalization function n_V that replaces all the fresh constants in a V-basis with canonical constants.

Let β be a V-basis. We denote the auxiliary constants in β ($\mathcal{C}(\beta) \setminus V$) by $W = \{w_0, w_1, \ldots\}$, and by '?' some unused constant that we call a *hole*. Recall that constants from W may only appear in literals of the form $v \approx f(\vec{w})$. We define

[2] In our setting, feasible coherent executions correspond to paths in the transition system of any CUP.

the set of W-templates as the set of all terms $f(\vec{a})$, where each element in $\vec{a}$ is either a hole or a constant in W. A term t *matches* a template $f(\vec{a})$ if $t = f(\vec{b})$, and $\vec{a}$ and $\vec{b}$ agree on all constants in W. For example, let ξ be the template $f(?, w_1, ?, w_2)$. The term $f(a, w_1, b, w_2)$ matches ξ, but $f(w_0, w_1, b, w_2)$ does not, because one of the holes is filled with $w_0 \in W$. We say that a literal $v \approx f(\vec{b})$ matches a template ξ if $f(\vec{b})$ matches ξ. The W-context of a W-template ξ in a set of literals L, denoted $Z_L(\xi)$, is the set $Z_L(\xi) \triangleq \{\ell[W \mapsto ?] \mid \ell \in L \land \ell \text{ matches } \xi\}$, where $\ell[W \mapsto ?]$ means that all occurrences of constants in W are replaced with a hole. For example, let $\xi = f(?, w_1, w_2, ?)$ and $L = \{v \approx f(a, w_1, w_2, b), u \approx f(c, w_1, w_2, a), w \approx f(x, w_1, w_2, b), x \approx g(x, w_1, w_2, b))\}$ then $Z_L(\xi) = \{v \approx f(a, ?, ?, b), u \approx f(c, ?, ?, a), w \approx f(x, ?, ?, b)\}$.

Since V and $\mathcal{F}$ are finite, the number of W-contexts is finite, independent of W. Let w_Z be a fresh constant for context Z.

Definition 12 (Normalization Function) The normalization function $n_V(\beta)$ is defined as follows:

(1) for each $t \in \mathcal{T}(\Gamma)$ s.t. $\mathcal{C}(t) \cap W \neq \emptyset$, create a template ξ by dropping all constants not in W. Let Ξ denote the set of templates so obtained.
(2) Let $Ctx \triangleq \{Z_\Gamma(\xi) \mid \xi \in \Xi\}$.
(3) For each $\ell \in \Gamma$, if $\ell[W \mapsto ?] \in Z$ for some $Z \in Ctx$, then replace all occurrences of W in ℓ with w_Z. □

The normalization preserves V-equivalence of β because it renames local constants, while maintaining all consequences that are derivable through them. That is, $n_V(\beta) \equiv_V \beta$. Furthermore, $n_V(\beta)$ is cannonical.

Therefore, given a set of literals Γ, we use $n_V(\beta)$ as a computable implementation of the V-base abstraction, α_V (Def. 9). That is, $\alpha_V(\Gamma) \triangleq n_V(\beta)$ where $\langle W, \beta, \delta\rangle \in \text{base}(\Gamma, V)$. Even though $n_V(\beta)$ may not be a part of a V-basis for Γ, it satisfies all the properties used in the proof of Thm. 4.

We define the normalizing abstraction in the usual way:

Definition 13 (Normalizing abstraction) The normalizing abstraction function $\alpha_n : C \to C$ is defined by

$$\alpha_n(\langle s, q_0, pc\rangle) \triangleq \langle s, q_0, n(pc)\rangle$$
□

Let $\alpha_{b,r,n} \triangleq \alpha_b \circ \alpha_r \circ \alpha_n$ be the composition of normalization abstraction with renaming and base abstraction where α_b is implemented using normalization. Notice that, for any state $c = \langle s, q, pc\rangle$, $\alpha_{b,r,n}(c)$ is computed by first computing *any* V-basis of pc, applying n_q, renaming all $\mathcal{C}(q)$ constants to q_0, and applying n_{q_0}. The second normalization is required to ensure that the fresh constants are canonical with respect to q_0. By definition $\alpha_{b,r,n}$ is computable. Hence, it can be used to compute the finite abstraction of any CUP.

Theorem 6 *For a CUP P, the finite abstract transition system $\alpha_{b',r,n}(\mathcal{S}_P)$ is bisimilar to P and is computable.* □

Thm. 6 implies that any property that is decidable over a finite transition system is also decidable over CUPs. In particular, temporal logic model checking is decidable.

VII. CONCLUSION

In this paper, we study theoretical properties of Coherent Uninterpreted Programs (CUPs) that have been recently proposed by Mathur et al. [9]. We identify a bug in the original paper, and provide an alternative proof of decidability of the reachability problem for CUP. More significantly, we provide a logical characterization of CUP. First, we show that inductive invariant of CUP is describable by shallow formulas. Hence, the set of all candidate invariants can be effectively enumerated. Second, we show that CUPs are bisimilar to finite transition systems. Thus, while they are formally infinite state, they are not any more expressive than a finite state system. Third, we propose an algorithm to compute a finite transition system of a CUP. This lifts all existing results on finite state model checking to CUPs.

In the paper, we have focused on the core result of Mathur et al, and have left out several interesting extensions. In [9], the notion of CUP is extended with k-coherence – a UP P is k-coherent if it is possible to transform P into a CUP $\hat{P}$ by adding k *ghost* variables to P. This is an interesting extension since it makes potentially many more programs amenable to decidable verification. We observe that addition of *ghost* variables is a form of abstraction. Thus, invariants of $\hat{P}$ can be translated to invariants of P using techniques of Namjoshi et al. [13], [14]. This essentially amounts to existentially eliminating ghost variables from the invariant of $\hat{P}$. Such elimination increases the depth of terms in the invariant at most by one for each variable eliminated. Thus, we conjecture that k-coherent programs are characterized by invariants with terms of depth at most k.

Mathur et al. [9] extend their results to recursive UP programs (i.e., UP programs with recursive procedures). We believe our logical characterization results extend to this setting as well. In this case, both the invariants and procedure summaries (i.e., procedure pre- and post-conditions) are described using terms of depth at most 1.

Our results also hold when CUPs are extended with simple axiom schemes, as in [10], while for most non-trivial axiom schemes CUPs become undecidable.

Perhaps most interestingly, our results suggest efficient verification algorithms for CUPs and interesting abstraction for UPs. Since the space of invariant candidates is finite, it can be enumerated, for example, using implicit predicate abstraction. For CUPs, this is a complete verification method. For UPs it is an abstraction. Most importantly, it does not require prior knowledge to whether an UP is a CUP!

Acknowledgment: The research leading to these results has received funding from the European Research Council under the European Union's Horizon 2020 research and innovation programme (grant agreement No [759102-SVIS]). This research was partially supported by the United States-Israel Binational Science Foundation (BSF) grant No. 2016260, and the Israeli Science Foundation (ISF) grant No. 1810/18. We also acknowledge the support of the Natural Sciences and Engineering Research Council of Canada (NSERC).

REFERENCES

[1] J. R. Burch and D. L. Dill, “Automatic verification of pipelined microprocessor control,” in *Computer Aided Verification, 6th International Conference, CAV ’94, Stanford, California, USA, June 21-23, 1994, Proceedings*, ser. Lecture Notes in Computer Science, D. L. Dill, Ed., vol. 818. Springer, 1994, pp. 68–80. [Online]. Available: https://doi.org/10.1007/3-540-58179-0_44

[2] D. Calvanese, S. Ghilardi, A. Gianola, M. Montali, and A. Rivkin, “Model completeness, covers and superposition,” in *Automated Deduction - CADE 27 - 27th International Conference on Automated Deduction, Natal, Brazil, August 27-30, 2019, Proceedings*, ser. Lecture Notes in Computer Science, P. Fontaine, Ed., vol. 11716. Springer, 2019, pp. 142–160. [Online]. Available: https://doi.org/10.1007/978-3-030-29436-6_9

[3] A. Cimatti, A. Griggio, S. Mover, and S. Tonetta, “IC3 modulo theories via implicit predicate abstraction,” in *Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings*, ser. Lecture Notes in Computer Science, E. Ábrahám and K. Havelund, Eds., vol. 8413. Springer, 2014, pp. 46–61. [Online]. Available: https://doi.org/10.1007/978-3-642-54862-8_4

[4] S. Ghilardi, A. Gianola, and D. Kapur, “Computing uniform interpolants for EUF via (conditional) dag-based compact representations,” in *Proceedings of the 35th Italian Conference on Computational Logic - CILC 2020, Rende, Italy, October 13-15, 2020*, ser. CEUR Workshop Proceedings, F. Calimeri, S. Perri, and E. Zumpano, Eds., vol. 2710. CEUR-WS.org, 2020, pp. 67–81. [Online]. Available: http://ceur-ws.org/Vol-2710/paper5.pdf

[5] S. Gulwani and M. Musuvathi, “Cover algorithms and their combination,” in *Programming Languages and Systems, 17th European Symposium on Programming, ESOP 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, ser. Lecture Notes in Computer Science, S. Drossopoulou, Ed., vol. 4960. Springer, 2008, pp. 193–207. [Online]. Available: https://doi.org/10.1007/978-3-540-78739-6_16

[6] S. Gulwani and G. C. Necula, “A polynomial-time algorithm for global value numbering,” in *Static Analysis, 11th International Symposium, SAS 2004, Verona, Italy, August 26-28, 2004, Proceedings*, ser. Lecture Notes in Computer Science, R. Giacobazzi, Ed., vol. 3148. Springer, 2004, pp. 212–227. [Online]. Available: https://doi.org/10.1007/978-3-540-27864-1_17

[7] H. G. V. K., S. Shoham, and A. Gurfinkel, “Logical characterization of coherent uninterpreted programs,” *CoRR*, vol. abs/2107.12902, 2021. [Online]. Available: https://arxiv.org/abs/2107.12902

[8] G. A. Kildall, “A unified approach to global program optimization,” in *Conference Record of the ACM Symposium on Principles of Programming Languages, Boston, Massachusetts, USA, October 1973*, P. C. Fischer and J. D. Ullman, Eds. ACM Press, 1973, pp. 194–206. [Online]. Available: https://doi.org/10.1145/512927.512945

[9] U. Mathur, P. Madhusudan, and M. Viswanathan, “Decidable verification of uninterpreted programs,” *Proc. ACM Program. Lang.*, vol. 3, no. POPL, pp. 46:1–46:29, 2019. [Online]. Available: https://doi.org/10.1145/3290359

[10] ——, “What’s decidable about program verification modulo axioms?” in *Tools and Algorithms for the Construction and Analysis of Systems - 26th International Conference, TACAS 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings, Part II*, ser. Lecture Notes in Computer Science, A. Biere and D. Parker, Eds., vol. 12079. Springer, 2020, pp. 158–177. [Online]. Available: https://doi.org/10.1007/978-3-030-45237-7_10

[11] R. Milner, *Communication and concurrency*, ser. PHI Series in computer science. Prentice Hall, 1989.

[12] M. Müller-Olm, O. Rüthing, and H. Seidl, “Checking herbrand equalities and beyond,” in *Verification, Model Checking, and Abstract Interpretation, 6th International Conference, VMCAI 2005, Paris, France, January 17-19, 2005, Proceedings*, ser. Lecture Notes in Computer Science, R. Cousot, Ed., vol. 3385. Springer, 2005, pp. 79–96. [Online]. Available: https://doi.org/10.1007/978-3-540-30579-8_6

[13] K. S. Namjoshi, “Lifting temporal proofs through abstractions,” in *Verification, Model Checking, and Abstract Interpretation, 4th International Conference, VMCAI 2003, New York, NY, USA, January 9-11, 2002, Proceedings*, ser. Lecture Notes in Computer Science, L. D. Zuck, P. C. Attie, A. Cortesi, and S. Mukhopadhyay, Eds., vol. 2575. Springer, 2003, pp. 174–188. [Online]. Available: https://doi.org/10.1007/3-540-36384-X_16

[14] K. S. Namjoshi and L. D. Zuck, “Witnessing program transformations,” in *Static Analysis - 20th International Symposium, SAS 2013, Seattle, WA, USA, June 20-22, 2013. Proceedings*, ser. Lecture Notes in Computer Science, F. Logozzo and M. Fähndrich, Eds., vol. 7935. Springer, 2013, pp. 304–323. [Online]. Available: https://doi.org/10.1007/978-3-642-38856-9_17

[15] G. Nelson and D. C. Oppen, “Fast decision procedures based on congruence closure,” *J. ACM*, vol. 27, no. 2, pp. 356–364, 1980. [Online]. Available: https://doi.org/10.1145/322186.322198

[16] R. Nieuwenhuis and A. Oliveras, “Proof-producing congruence closure,” in *Term Rewriting and Applications, 16th International Conference, RTA 2005, Nara, Japan, April 19-21, 2005, Proceedings*, ser. Lecture Notes in Computer Science, J. Giesl, Ed., vol. 3467. Springer, 2005, pp. 453–468. [Online]. Available: https://doi.org/10.1007/978-3-540-32033-3_33

[17] O. Strichman and B. Godlin, “Regression verification - A practical way to verify programs,” in *Verified Software: Theories, Tools, Experiments, First IFIP TC 2/WG 2.3 Conference, VSTTE 2005, Zurich, Switzerland, October 10-13, 2005, Revised Selected Papers and Discussions*, ser. Lecture Notes in Computer Science, B. Meyer and J. Woodcock, Eds., vol. 4171. Springer, 2005, pp. 496–501. [Online]. Available: https://doi.org/10.1007/978-3-540-69149-5_54