Reachability in Concurrent Uninterpreted Programs

Salvatore La Torre
Università degli Studi di Salerno, Italia
https://docenti.unisa.it/salvatore.latorre
slatorre@unisa.it

Madhusudan Parthasarathy
University of Illinois, Urbana-Champaign, USA
http://madhu.cs.illinois.edu/
madhu@illinois.edu

Abstract

We study the safety verification (reachability problem) for concurrent programs with uninterpreted functions/relations. By extending the notion of coherence, recently identified for sequential programs, to concurrent programs, we show that reachability in coherent concurrent programs under various scheduling restrictions is decidable by a reduction to multistack pushdown automata, and establish precise complexity bounds for them. We also prove that the coherence restriction for these various scheduling restrictions is itself a decidable property.

2012 ACM Subject Classification Theory of computation → Logic and verification; Theory of computation → Formal languages and automata theory; Software and its engineering → Formal software verification

Keywords and phrases Verification, uninterpreted programs, concurrent programs, shared memory

Digital Object Identifier 10.4230/LIPIcs.FSTTCS.2019.46

Acknowledgements This material is based upon work supported by the National Science Foundation under Grant CCF-1527395, GNCS 2019 grant and MIUR-FARB 2018-19 grant.

1 Introduction

Verification against assertion violations for sequential programs that have only Boolean variables and have recursive function calls is decidable, as the problem is equivalent to pushdown automata reachability/emptiness. However, generalizations of this result to other settings is hard. First, there are hardly any positive results for verifying recursive programs that work over infinite domains. Second, concurrent recursive program verification is typically undecidable (two stacks suffice to encode the executions of Turing machines) if the interaction of the threads is not restricted in any way.

A recent paper by Mathur et al. introduces a decidable class of sequential programs where the data domain is infinite [27]. The first ingredient for decidability is that the programs compute terms over functions and compare them over relations that are both assumed to be *uninterpreted*. The theory of uninterpreted functions is an important theory. Theoretically, it was the one studied by Gödel for his completeness theorems [15], and practically, the decidability of validity of its quantifier-free fragment is exploited by SMT solvers and is often used (typically in combination with other theories) to solve feasibility of loop-free program snippets, in bounded model-checking, and to validate verification conditions [10]. The second ingredient for decidability is a technical restriction called *coherence*. Coherent programs have two properties – the *memoizing property* (which intuitively says that computed terms once dropped cannot be recomputed) and the *early assume property* (which intuitively says that equality assumptions in program executions happen early, well before their superterms

are computed and dropped). The work by [27] shows that for coherent programs, one can build a streaming congruence closure algorithm with finite memory, and by modeling this algorithm as an automaton, verify programs.

From a practical point of view, uninterpreted abstractions have been considered [18], and a recent paper shows that verifying programs using an uninterpreted abstraction can be effective [12]. Also, extensions of the decidable verification result for uninterpreted programs has found applications in verifying memory-safety for heap-manipulating programs, where heaps are naturally modeled using infinite domains [28].

In this paper, we consider the problem of verifying concurrent uninterpreted programs, with both recursion and shared memory.

Note that concurrent recursive programs even over Boolean domains have an undecidability verification (reachability) problem. Programs with uninterpreted functions/relations are much more complex than programs with Boolean domains – it is easy to see that we can simulate a program with Boolean domains by using a program with no functions or relations, with two special immutable variables for T and F, and only using equality relations in the program. Moreover, this will always yield a coherent program. Consequently, concurrent recursive coherent uninterpreted programs clearly have an undecidable reachability problem.

There has been a rich literature of work that has identified restrictions of concurrent recursive Boolean programs for which reachability is decidable (see [2, 9, 19, 21, 24, 25, 30]). Verification of such programs can be modeled as reachability/emptiness problem in *multistack pushdown systems* and several underapproximations based on restricting the scheduling of threads has yielded decidability. These include bounded context-switching [30], bounded scope executions [24], (k, d)-budgeted executions [2], k-phase executions [20], k-path-tree executions [23], etc. Some generalizations of the above decidability results that show decidability when manipulating multiple stacks in a way that the accesses correspond to bounded tree-width manipulations are also known [13, 26], and some of these restrictions have been applied for finding errors in predicate abstracted programs as well [17, 31, 21]. There has also been a lot of work on studying register automata (both sequential and concurrent) on *data words* where registers can read data from infinite domains but use only equality/disequality checks (note that uninterpreted functions/relations that must satisfy the congruence axioms are not allowed) [6, 7, 14, 8, 29]; see also work extending to programs where variables range over natural numbers, with equality, but with no functions [1].

The goal of this paper is to establish decidability results for concurrent recursive programs working over *infinite* uninterpreted data domains with certain scheduling restrictions that were shown to yield decidable reachability in the setting of Boolean programs. Our main results are that coherent concurrent program safety verification (for a notion of coherence for concurrent programs we define) is decidable for bounded context-switching, bounded scope executions, (k, d)-budgeted executions, k-phase executions, ordered executions, and k-path-tree executions.

There are two primary technical challenges to establish our results. First, we need to define an appropriate extension of coherence for concurrent programs. We propose such a natural extension. We model a concurrent program as working on a data-domain that is shared amongst all processes; we think this is an important design decision as the alternative choice of having local universes is both unnatural (domains of programs in the real world are often common, like integers or other forms of data structures) and prohibits communication of unbounded data between processes. The notion of coherence (memoizing and early assumes) is defined based on the *frontier* of computation, which includes all variables that could ever come into scope. This includes the unbounded copies of local variables stored in each program's stack as well as the shared variables.

The second challenge is to build a multistack automaton that accurately captures the feasibility of coherent runs of the concurrent recursive program. In the automata constructed in [27] for sequential programs, the automata do not store actual elements of the universe in the stack or state, of course, as the universe is infinite. Rather, the automata store *relationships* between variables – more precisely the equalities between variables in the *initial model* defined by the equality assumptions occurring in the program's execution, the disequality constraints implied by it, and certain local function maps between variables. Let us call this the *EDF information* – information on equalities, disequalities, and function maps between variables. The automaton constructed by [27] effects a *streaming congruence closure* algorithm that has finite memory by keeping track of the EDF information on variables currently in scope after any execution.

In concurrent programs, the EDF information is significantly harder to keep track of because it includes relationships between not just the local variables of one process or relationships between local variables and shared variables, but also between local variables of *different* processes. For example, when a call returns in a local thread, it has to recover all EDF information between new local variables in scope and local variables of other threads.

The above complications mean that we *cannot* simply abstract each local thread into its EDF information and then take their concurrent evolution. In fact, our results do *not* extend to parameterized concurrent systems (where there are an unbounded number of processes), even for the cases where Boolean program verification under certain scheduling restrictions are known to be decidable (e.g., *bounded rounds* is known to be decidable [21]).

Our multistack automaton construction instead maintains a complex invariant – an element in the stack for a process p contains several kinds of information: the EDF information on local variables of p and shared variables at the time the push (function call) happened, EDF relationship of local variables in p to local variables in p just below the stack (the caller's variables), and most importantly, EDF information between local variables across processes *at the time this information was pushed onto the stack*. Maintaining this complex invariant at every stage is involved, and gives us the reduction from reachability of concurrent coherent programs to multistack automata reachability.

The reduction to multistack automata reachability is rewarding as we can exploit the fact that reachability of the latter under various scheduling restrictions have been well studied for establishing decidability. Utilizing these results, we show that concurrent coherent program reachability is decidable for the following restrictions: bounded context-switching, bounded scope executions (k, d)-budgeted executions, k-phase executions, ordered executions, and k-path-tree executions. We also show that our decidability results have optimal complexity. In fact we can show that the complexity of verification of coherent concurrent programs is precisely the same as that for Boolean programs under similar scheduling restrictions.

There is another natural related question that arises: Given a concurrent program with a scheduling restriction as above, how can the user determine whether it is coherent? We show that checking whether a concurrent program is coherent under these scheduling restrictions is also decidable and decidable in the same time complexity as the reachability algorithm.

2 Concurrent Uninterpreted Programs

In this section, we introduce the notion of concurrent programs over data domains with uninterpreted functions and relations. We start by recalling some definitions about first order data structures, then we recall the definition of uninterpreted sequential programs and then we extend it to concurrent programs.

A first order *signature* is $(\mathcal{C}, \mathcal{F}, \mathcal{R})$ where $\mathcal{C}$ is a set of constants, $\mathcal{F}$ is a set of function symbols, and $\mathcal{R}$ is a set of relation symbols. Relations and functions have an implicitly assigned arity in $\mathbb{N}_{>0}$. A signature is *algebraic* if $\mathcal{R}$ is empty and in this case we denote it simply as the pair $(\mathcal{C}, \mathcal{F})$. A *data model* for $(\mathcal{C}, \mathcal{F}, \mathcal{R})$ is $\mathcal{M} = (U, \{[\![c]\!] \mid c \in \mathcal{C}\}, \{[\![f]\!] \mid f \in \mathcal{F}\}, \{[\![R]\!] \mid R \in \mathcal{R}\})$ consisting of a universe, and an interpretation on the universe for constants, functions and relations (a data model for an algebraic signature will not have an interpretation for the relations). The set of *terms* is defined inductively as follows: each constant from $\mathcal{C}$ is a term and for any m-ary function f and terms $t_1, \ldots, t_m$, $f(t_1, \ldots, t_m)$ is also a term. An *immediate superterm* of t is $f(t_1, \ldots, t_m)$ where $t \in \{t_1, \ldots, t_m\}$. A *superterm* of t is either an immediate superterm of t or an immediate superterm of a superterm of t. The interpretation of a term t in $\mathcal{M}$ is denoted as $[\![t]\!]_{\mathcal{M}}$.

In the following, for an integer $n > 0$, we denote with $[n]$ the set $\{1, \ldots, n\}$.

2.1 Uninterpreted sequential programs

We consider simple *sequential* programs over uninterpreted functions and relations, and with possibly recursive calls to methods. We fix a finite set of variables V which includes both local and global variables used by the programs to store information during a computation. Values are manipulated by using function and relation symbols from a first order signature $(\mathcal{C}, \mathcal{F}, \mathcal{R})$. We also fix a finite set of method names M. A program is essentially formed of a list of method definitions, one for each method name in M and such that there is a *main* method, i.e., the method from which the execution starts, that we denote m_0. We allow methods to return tuples of values, thus for every method $m \in M$, we fix a tuple of distinct *output* variables $\mathtt{o}_m$. Also for the ease of presentation and without loss of generality, we assume that all methods have the same list of parameters which coincides with a fixed permutation of all the local variables. In the following, we denote such list as *lvars*. Each method body contains assignments, sequencing, conditionals, loops and method calls.

The precise syntax is given by the following grammar:

$$
\begin{array}{lll}
\langle pgm \rangle & ::= & m \Rightarrow \mathtt{o}_m\langle\, stmt\rangle \mid \langle pgm\rangle\ \langle pgm\rangle \\
\langle stmt \rangle & ::= & \langle stmt\rangle\ \mathtt{;}\ \langle stmt\rangle \mid \mathtt{skip} \mid x := y \mid x := f(\mathtt{z}) \mid \mathtt{assume}(\langle cond\rangle) \\
 & & \mid \mathtt{w} := m(lvars) \mid \mathtt{if}\ (\langle cond\rangle)\ \mathtt{then}\ \langle stmt\rangle\ \mathtt{else}\ \langle stmt\rangle \mid \mathtt{while}\ (\langle cond\rangle)\ \langle stmt\rangle \\
\langle cond \rangle & ::= & x = y \mid x = c \mid c = d \mid R(\mathtt{z}) \mid \langle cond\rangle \vee \langle cond\rangle \mid \neg\langle cond\rangle
\end{array}
$$

In the above, $m \in M$ is a method name, $c, d \in \mathcal{C}$ are constants, $f \in \mathcal{F}$ is a function name, $R \in \mathcal{R}$ is a relation name, $x, y \in V$ are variables, $\mathtt{w}$ is a tuple of variables from V, and $\mathtt{z}$ is a tuple of constants from $\mathcal{C}$ and variables from V. Moreover, we allow for standard operators: ':=' is the assignment operator, ';' is the program sequencing operator, `skip` is the "do nothing" statement, `if-then-else` is the usual conditional statement and `while` is the usual loop statement. Method calls are handled as usual with a call stack. Namely, a *configuration* of the program consists of a *stack* which stores the history of positions at which calls were made, along with valuations for local variables, and the top of the stack contains the local and global valuations, and a pointer to the current statement being executed. Note that we do not make use of an explicit `return` statement: a call to module m is returned when there are no more statements of m to execute (the values that need to be returned are assigned to the output variables $\mathtt{o}_m$ before the call ends).

For the ease of presentation, in the rest of the paper we will assume that the programs have only conditionals of the form '$x = y$' and '$x \neq y$', constants will not appear in any of the program expressions and the signature is algebraic, i.e., $\mathcal{R}$ is empty. Note that this is without loss of generality. In fact, any relation R can be captured by a function f_R with the

same arity and a Boolean variable b_R. A Boolean combination of conditions can be modeled using the `if-then-else` construct. Constants can be removed by using instead variables that are not modified in the program.

Fix a set of functions $\mathcal{F}$. An *execution* of a sequential program over a set of variables V and set of methods M is a sequence over the alphabet $\Pi = \{\text{“}x := y\text{”}, \text{“}x := f(\mathtt{z})\text{”}, \text{“}\mathtt{assume}(x = y)\text{”}, \text{“}\mathtt{assume}(x \neq y)\text{”}, \text{“}\mathtt{call}\ m\text{”}, \text{“}\mathtt{w} := \mathtt{return}\text{”} \mid x, y, \mathtt{z}, \mathtt{w} \text{ are in } V, m \in M\}$. In particular, for a program P, denoting $bd(m)$ the body of a method $m \in M$, the set of *complete executions* of P is generated by the following context-free grammar:

$$
\begin{array}{rcl}
X_{\varepsilon} & \to & \varepsilon \\
X_{\mathtt{skip};\, st} & \to & X_{st} \\
X_{x:=y;\, st} & \to & \text{“}x := y\text{”} \cdot X_{st} \\
X_{x:=f(\mathtt{z});\, st} & \to & \text{“}x := f(\mathtt{z})\text{”} \cdot X_{st} \\
X_{\mathtt{assume}(c);\, st} & \to & \text{“}\mathtt{assume}(c)\text{”} \cdot X_{st} \\
X_{\mathtt{if}(c)\ \mathtt{then}\ st_1\ \mathtt{else}\ st_2\ ;\, st} & \to & \text{“}\mathtt{assume}(c)\text{”} \cdot X_{st_1\ ;\, st} \mid \text{“}\mathtt{assume}(\neg c)\text{”} \cdot X_{st_2\ ;\, st} \\
X_{\mathtt{while}(c)\ \{st_1\}\ ;\, st} & \to & \text{“}\mathtt{assume}(c)\text{”} \cdot X_{st_1\ ;\, \mathtt{while}(c)\ \{st_1\}\ ;\, st} \mid \text{“}\mathtt{assume}(\neg c)\text{”} \cdot X_{st} \\
X_{\mathtt{w}:=m(lvars)\ ;\ st} & \to & \text{“}\mathtt{call}\ m\text{”} \cdot X_{bd(m)} \cdot \text{“}\mathtt{w} := \mathtt{return}\text{”} \cdot X_{st}
\end{array}
$$

where X_{st} denotes the nonterminal symbol corresponding to a statement st and $X_{bd(m_0)\,;\,\varepsilon}$ is the start symbol (recall m_0 is the main method). An *execution* is any prefix of a complete execution. Note that not all the executions are feasible.

2.2 Concurrent uninterpreted programs

A *concurrent uninterpreted program* is a finite set of recursive uninterpreted programs running in parallel and sharing a finite set of variables S. The syntax of concurrent programs is defined by extending the syntax of the sequential programs with the following rule:

$$\langle \textit{conc-pgm} \rangle \ ::= \ \langle \textit{pgm} \rangle \ \mid \ \langle \textit{pgm} \rangle \,||\, \langle \textit{conc-pgm} \rangle$$

where each sequential program uses its own set of local and global variables along with the set of the shared variables (for a sequential program the shared variables are as global variables, the only difference is that they can be read and written also by the other sequential threads).

For the rest of the paper, we fix a concurrent uninterpreted program $\mathcal{P}$ that is formed by the sequential programs $P_1, \ldots, P_n$ (where $n > 0$). We refer to programs $P_1, \ldots, P_n$ as the *component programs* of $\mathcal{P}$. We denote with V_i the set of local and global variables of each P_i and assume that the sets $S, V_1, \ldots, V_n$ are pairwise disjoint. Further, we denote $\mathtt{Vars} = S \cup \bigcup_{i=1}^{n} V_i$ the set of all variables used in $\mathcal{P}$. Any (*complete*) *execution* of $\mathcal{P}$ is obtained as an interleaving $\rho_1, \ldots, \rho_n$ where ρ_i is a (complete) execution of P_i for $i \in [n]$.

To distinguish among the different sequential programs in a concurrent execution, we assume pairwise disjoint alphabets and for each program P_i, we will denote the corresponding alphabet Π_i and any symbol of the form “a” as $\langle a \rangle_i$, that is, we let $\Pi_i = \{\langle x := y \rangle_i, \langle x := f(\mathtt{z}) \rangle_i, \langle \mathtt{assume}(x = y) \rangle_i, \langle \mathtt{assume}(x \neq y) \rangle_i, \langle \mathtt{call}\ m \rangle_i, \langle \mathtt{w} := \mathtt{return} \rangle_i \mid x, y, \mathtt{z}, \mathtt{w} \text{ are in } V_i \cup S, m \in M\}$. Thus, the overall alphabet for $\mathcal{P}$ is $\Pi = \bigcup_{i \in [n]} \Pi_i$.

At the beginning of any execution each variable x is set according to an initial interpretation of the data model that we denote with $\mathtt{init}(x)$. Then, the variables are updated according to the intended semantics of the statements.

For any ρ' such that $\rho \cdot \rho' \cdot \rho''$ is an execution of $\mathcal{P}$ and $i \in [n]$, we say that ρ' is *i-matched* if all the calls of program P_i (i.e., of the form $\langle \mathtt{call}\ m \rangle_i$) that occur in ρ' are matched within ρ'. The map `Comp` captures the term associated with program variables at the end of any execution. Denoting with ρ any execution of $\mathcal{P}$, `Comp` is inductively defined as follows:

$$
\begin{array}{rcll}
comp(\varepsilon, x) & = & \texttt{init}(x) & x \in \texttt{Vars} \\
\texttt{Comp}(\rho.\langle x := y\rangle_i, x) & = & \texttt{Comp}(\rho, y) & \\
\texttt{Comp}(\rho.\langle x := y\rangle_i, x') & = & \texttt{Comp}(\rho, x') & x' \in \texttt{Vars} \text{ and } x' \neq x \\
\texttt{Comp}(\rho \cdot \langle x := f(\texttt{z})\rangle_i, x) & = & f(\texttt{Comp}(\rho, z_1), \ldots, \texttt{Comp}(\rho, z_r)) & \text{where } \texttt{z} = (z_1, \ldots, z_r) \\
\texttt{Comp}(\rho \cdot \langle x := f(\texttt{z})\rangle_i, x') & = & \texttt{Comp}(\rho, x') & x \neq x' \\
\texttt{Comp}(\rho \cdot \langle \texttt{assume}(x = y)\rangle_i, x') & = & \texttt{Comp}(\rho, x') & x' \in \texttt{Vars} \\
\texttt{Comp}(\rho \cdot \langle \texttt{assume}(x \neq y)\rangle_i, x') & = & \texttt{Comp}(\rho, x') & x' \in \texttt{Vars} \\
\texttt{Comp}(\rho \cdot \langle \texttt{call } m\rangle_i \cdot \rho' & & & \\
\quad \cdot \langle (w_1, \ldots w_r := \texttt{return}\rangle_i, w_j) & = & \texttt{Comp}(\rho \cdot \langle \texttt{call } m\rangle_i \cdot \rho', \texttt{o}_m[j]) & \rho' \text{ is } i\text{-matched} \\
\texttt{Comp}(\rho \cdot \langle \texttt{call } m\rangle_i \cdot \rho' & & & \rho' \text{ is } i\text{-matched}, \\
\quad \cdot \langle (w_1, \ldots w_r := \texttt{return}\rangle_i, x) & = & \texttt{Comp}(\rho, x) & x \notin \{w_1, \ldots w_r\}
\end{array}
$$

We denote with $\leq$ the prefix relation among executions, i.e., for executions ρ, ρ', with $\rho' \leq \rho$ we mean that ρ' is a prefix of ρ. The set of all the terms computed by an execution ρ is $\texttt{Terms}(\rho) = \bigcup_{\rho' \leq \rho, x \in \texttt{Vars}} \texttt{Comp}(\rho', x)$.

The semantics of uninterpreted programs is defined with respect to a data model that gives a meaning to the elements in the signature, and thus to the computed terms. An execution ρ is said to be *feasible* with respect to a data model if the assumptions it makes are true in that model. To formalize the notion of feasible executions we first define the sets of the equality assumes and the disequality assumes of an execution.

For any execution ρ, the set of *equality assumes* of ρ, denoted $\alpha(\rho)$, is a subset of $\texttt{Terms}(\rho) \times \texttt{Terms}(\rho)$ inductively defined as: $\alpha(\varepsilon) = \emptyset$; and if σ is $\langle \texttt{assume}(x = y)\rangle_i$, $i \in [n]$, then $\alpha(\rho \cdot \sigma) = \alpha(\rho) \cup \{(\texttt{Comp}(\rho, x), \texttt{Comp}(\rho, y))\}$, otherwise $\alpha(\rho \cdot \sigma) = \alpha(\rho)$. Similarly, the set of *disequality assumes* $\beta(\rho)$ can be defined as: $\beta(\varepsilon) = \emptyset$; and if σ is $\langle \texttt{assume}(x \neq y)\rangle_i$, $i \in [n]$, then $\beta(\rho \cdot \sigma) = \beta(\rho) \cup \{(\texttt{Comp}(\rho, x), \texttt{Comp}(\rho, y))\}$, otherwise $\beta(\rho \cdot \sigma) = \beta(\rho)$.

An execution ρ is *feasible* in a data-model $\mathcal{M}$ if $[\![t]\!]_{\mathcal{M}} = [\![t']\!]_{\mathcal{M}}$ for every $(t, t') \in \alpha(\rho)$, and $[\![t]\!]_{\mathcal{M}} \neq [\![t']\!]_{\mathcal{M}}$ for every $(t, t') \in \beta(\rho)$.

We recall that an equivalence relation $\cong \subseteq \texttt{Terms} \times \texttt{Terms}$ 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 $A \subseteq \texttt{Terms} \times \texttt{Terms}$, the *congruence closure* of A, denoted $\cong_A$, is the smallest congruence containing A. We can then show:

▶ **Proposition 1.** *An execution ρ is feasible in some data model if and only if $\cong_{\alpha(\rho)} \cap \beta(\rho) = \emptyset$.*

3 Verification of concurrent uninterpreted programs

The basic verification problem is *reachability* that consists of checking whether a given set of target states is reachable in a program execution. For uninterpreted programs, there is an additional request: the execution must be feasible in some data model. The target set is often captured by a program counter (corresponding to an assertion) and a Boolean combination of equalities over program variables. As also observed in [27], by simple program transformations, the reachability problem for uninterpreted programs can always be reduced to checking the existence of a feasible complete execution (the assertion condition is translated into a block containing `assume` and `if` statements). In such a translation the size of the resulting program is linear in the sizes of the starting program and the assertion conditions. Thus we consider the following reachability problem for concurrent uninterpreted programs:

▶ **Definition 2** (Reachability). *Given a concurrent uninterpreted program $\mathcal{P}$, the reachability problem asks whether there exists a feasible complete execution of $\mathcal{P}$.*

We observe that this decision problem is already undecidable for sequential uninterpreted programs even in absence of recursive method calls, and becomes EXPTIME-complete if the search is restricted to coherent computations [27]. Unfortunately, in the case of concurrent uninterpreted programs, assuming coherence does not suffice to gain decidability. In fact, reachability is undecidable even for concurrent programs with variables ranging over finite domains and with only two component programs. Consequently, we further restrict the executions by adopting some limitations studied in the literature that bound the interaction among the component programs.

In the rest of this section, we define first the notion of coherence, then the above mentioned restrictions for concurrent uninterpreted programs, and then the general bounded reachability problem. Finally, we conclude with a high level description of our approach to decide it.

Coherence. For terms $t_1, t_2 \in \texttt{Terms}$ and congruence $\cong$ on $\texttt{Terms}$, we say that t_2 *is a superterm of* t_1 *modulo* $\cong$ if there are terms $t_1', t_2' \in \texttt{Terms}$ such that $t_1' \cong t_1$, $t_2' \cong t_2$ and t_2' is a superterm of t_1'. For the ease of presentation in the informal descriptions we will identify equivalent terms, and thus we will refer to a term meaning any of its equivalent terms. For example, if we say that a term t is recomputed in an execution, we actually mean that the computed term t is equivalent to a term that was computed earlier in the execution. Also, we call a superterm modulo an equivalence $\cong$ simply a superterm.

The notion of coherent execution introduced in [27] for sequential programs naturally extends to concurrent programs by using the $\texttt{Comp}$ map defined above. Informally, an execution is coherent if it is memoizing and has early assumes. The memoizing property says that if a term t is recomputed there must be a variable that currently evaluates to t. The early assume property instead imposes constraints on when $\langle \texttt{assume}(x = y)\rangle_i$ steps are taken within the execution: it requires that such $\texttt{assume}$ statements appear before the execution reassigns all the variables storing any computed term t that is a superterm of the terms stored in x or y (i.e., before it "drops" all of such superterms).

Formally, we say that a (complete) execution ρ over variables $\texttt{Vars}$ is *coherent* if it satisfies the following two properties:

1. (*Memoizing*) Let $\pi' = \pi \cdot \langle x := f(\texttt{z})\rangle_i$ be a prefix of ρ and let $t = \texttt{Comp}(\pi', x)$. If there is a term $t' \in \texttt{Terms}(\pi)$ such that $t' \cong_{\alpha(\pi)} t$, then there must exist some $y \in V$ such that $\texttt{Comp}(\pi, y) \cong_{\alpha(\pi)} t$.
2. (*Early Assumes*) Let $\pi' = \pi \cdot \langle \texttt{assume}(x{=}y)\rangle$ be a prefix of ρ and let $t_x = \texttt{Comp}(\pi, x)$ and $t_y = \texttt{Comp}(\pi, y)$. If there is a term $t' \in \texttt{Terms}(\pi)$ such that t' is either a superterm of t_x or of t_y modulo $\cong_{\alpha(\pi)}$, then there must exist a variable $z \in V$ such that $\texttt{Comp}(\pi, z) \cong_{\alpha(\pi)} t'$.

A *coherent program* is a program whose executions are all coherent.

In the literature, there is a notion of *freshness* [32, 8] that may remind people of the notion of memoizing above; however, these are not similar, as the memoizing restriction is on freshness of the computed *terms* and not on the underlying semantics of the data values computed (two terms may be different but still correspond to the same element in a particular data model).

Bounding the interaction among the component programs. Denote with $\Pi = \bigcup_{i\in[n]} \Pi_i$ the alphabet over variables $\texttt{V}$ and functions $\mathcal{F}$. Also, for any ρ' such that $\rho \cdot \rho' \cdot \rho''$ is an execution of $\mathcal{P}$ and $i \in [n]$, we say that ρ' is *i-matched* if all the calls of program P_i (i.e., of the form $\langle \texttt{call}\ m\rangle_i$) that occur in ρ' are matched within ρ'. For integers $k, d > 0$, we consider the *bounding conditions* that restrict the search respectively to the following sets of executions:

- a *k-context execution* ρ is the concatenation of k contexts, i.e., $\rho = \rho_1 \dots \rho_k$ where $\rho_i \in \Pi_{j_i}^*$, for $i \in [k]$ and $j_i \in [n]$, is a *context* of P_{j_i} (*bounded context-switching*, CON for short) [30];
- a *k-scoped execution* ρ is such that for each pair of matching call and return from any Π_i, the portion of ρ delimited by them does not contain more than k contexts of P_i, i.e., for any decomposition $\rho = \rho'.\sigma_c.\rho''.\sigma_r.\rho'''$ where for some $i \in [n]$, σ_c is of the form $\langle \texttt{call } m \rangle_i$, σ_r is of the form $\langle \texttt{w} := \texttt{return} \rangle_i$, and ρ'' is i-matched, ρ'' does not contain more than k contexts of P_i (*scope-bounded matching relations*, SCO for short) [24];
- a (k, d)-*budget execution* ρ is such that for each component program P_i and for each ρ'' such that $\rho = \rho'.\rho''.\rho'''$ where the call stack of P_i contains more than d calls, there are at most k contexts of P_i in ρ'' (*budget-bounded context-switching*, BUD for short) [2][1];
- a *k-phase execution* is the concatenation of k phases where a *phase* of component program P_i is a sequence from Π where all the `return` symbols are from alphabet Π_i, i.e., are of the form $\langle \texttt{w} := \texttt{return} \rangle_i$ (*bounded number of phases*, PHA for short) [20];
- an *ordered execution* ρ is such that for each $j \in [n]$ and for each return σ_r from Π_j, all the calls from Π_i, with $i < j$, that occur in ρ before σ_r are matched, i.e., for each decomposition $\rho = \rho'.\sigma_r.\rho''$, ρ' is i-matched for all $i < j$ (*ordered matching relations*, ORD for short) [11];
- a *k-path-tree execution* ρ is such that it can be encoded into a *stack tree*[2] whose nodes can be discovered in the order given by ρ by a walk that starts from the root and visits each node at most k times (*bounded path-trees*, PAT for short) [23].

Bounded reachability. The bounded reachability problem asks to solve reachability by restricting the search within a subset of the coherent program executions that satisfy a given bounding condition. Formally:

▶ **Definition 3** (Bounded Reachability). *Given a concurrent uninterpreted program $\mathcal{P}$ and a bounding condition $\mathcal{B}$ over the executions of $\mathcal{P}$, the $\mathcal{B}$-bounded reachability problem asks whether there exists a feasible and coherent complete execution of $\mathcal{P}$ that satisfies $\mathcal{B}$.*

In the following, we will refer to a $\mathcal{B}$-bounded reachability problem as $\mathcal{B}$-REACH.

Decision algorithm. We reduce the bounded reachability problem to a reachability problem in multistack visibly pushdown automata (MVPA). In particular, we construct an MVPA $\mathcal{A}_{\texttt{Vars}}$ that captures exactly all the coherent and feasible executions over an alphabet Π, and an MVPA $\mathcal{A}_{\mathcal{P}}$ that captures all the executions of $\mathcal{P}$. We then take the intersection of the two MVPA's and check if it accepts an execution that fulfills the bounding condition. In Section 4, we construct $\mathcal{A}_{\texttt{Vars}}$ and prove its correctness, and in Section 5 we give $\mathcal{A}_{\mathcal{P}}$ and discuss the correctness and complexity of the decision algorithm for the considered bounding conditions.

[1] The original definition admits a different value of k and d for each component program however the computational complexity of the reachability problem is the same.

[2] A stack tree is a binary tree obtained by labeling the root with the first symbol of ρ, and then the successor in ρ labels the left child unless it is a matched return, and in this case it labels the right child of the matching call.

4 MVPA capturing coherent and feasible executions

In this section, we construct an MVPA $\mathcal{A}_{\mathtt{Vars}}$ that accepts all the coherent and feasible executions of a concurrent uninterpreted program over the variables $\mathtt{Vars}$ and functions $\mathcal{F}$.

The crux of the construction is to represent and maintain the equality/disequality/functional (EDF) relationship between variables along a concurrent execution. Concurrency poses new challenges on how to maintain this EDF information. Besides the terms that are currently stored in the program variables, we need to account for those in the local variables of unreturned calls (still in a call stack) for each of the component programs. In concurrent programs, a term can flow from a local variable into a shared variable and then to a local variable of another component program thus potentially establishing direct equality/inequality/superterm relations among the terms stored in two local variables of two different component programs. Moreover, as the execution proceeds, these terms can go deep down into their respective call stacks while no other currently used variable stores them (nor terms that are equivalent to them), and still on returning, these relations need to be restored.

We organize this complex (and unbounded) piece of information into *stack* and *shared states*. Stack states are kept into the stacks of the corresponding component programs while shared states are maintained in the control state of the MVPA, and both of them store the relations among the content of all variables. To link the shared state with all the stack states at the top of the stacks and a stack state to the next stack state below into the stack, we use *shadow variables*, i.e., additional variables that are used in our relations as placeholders for actual program variables. For each component program we add a shadow variable for each program variable. The stack state of a component program is then augmented with its shadow variables, and the shared state is augmented with the overall set of shadow variables (we need to link this state to all the stacks). When a method call of component P_i is issued, we push into stack i the stack state of P_i that can be derived from the current shared state, and then update the shared state by setting the shadow variables of P_i equal to the corresponding program variables. Shadow variables stay unchanged in all the other cases, and thus we maintain the invariant that a shadow variable of P_i has the value of the corresponding program variable at the time the current method of P_i was called (the initial value in the case of the main method). This way, in each stack, a stack state l is linked to the state l' below it by having each shadow variable of l to evaluate equal to the corresponding program variable in l', thus forming a chain across the stack values. The same holds for the shared state and the stack states at the top of all the stacks.

We recall that a *multistack visibly pushdown automaton* (MVPA) consists of a finite control along with one or more pushdown stores (stacks) that are driven by the input. We refer the reader to [3, 24] for the details.

In the rest of the section, we first formalize the introduced notions, then we give some details on the construction of the MVPA and argue its correctness.

Shadow variables. Fix $i \in [n]$. For each component program P_i, we consider a shadow variable for each shared variable and for each variable of all the component programs P_j with $j \in [n]$. We denote the first set as S'_i and the second set as $V'_{i,j}$. Further, we denote $\mathtt{Vars}'_i = S'_i \cup \bigcup_{j\in[n]} V'_{i,j}$ (the overall set of shadow variables for P_i), $\mathtt{Vars}' = \bigcup_{i\in[n]} \mathtt{Vars}'_i$ (the set of all the shadow variables), $\mathtt{V} = \mathtt{Vars} \cup \mathtt{Vars}'$ (the overall set of variables), and $\mathtt{V}_i = \mathtt{Vars} \cup \mathtt{Vars}'_i$ (the set of the program variables along with the shadow variables of P_i).

We extend the notation $\mathtt{Comp}$ to capture the described semantics of the shadow variables as follows. For $i \in [n]$, denoting with $x' \in \mathtt{Vars}'_i$ the shadow variable corresponding to $x \in \mathtt{Vars}$, we set: $\mathtt{Comp}(\varepsilon, x') = \mathtt{Comp}(\varepsilon, x)$, $\mathtt{Comp}(\rho.\langle \mathtt{call}\ m\rangle_i, x') = \mathtt{Comp}(\rho, x)$ (i.e., x' stores the value

of x on calling a method), $\texttt{Comp}(\rho.\langle \texttt{call } m\rangle_i.\rho'.\langle \texttt{w} := \texttt{return}\rangle_i, x') = \texttt{Comp}(\rho, x')$ where ρ' is i-matched (i.e., after the call the previous value of x' is restored) and $\texttt{Comp}(\rho.\sigma, x') = \texttt{Comp}(\rho, x')$ for all $\sigma \notin \{\langle \texttt{call } m\rangle_i, \langle \texttt{w} := \texttt{return}\rangle_i \mid m \text{ is a method}\}$.

States and invariants. For an equivalence relation $\sim$ over a set V, we denote with $V/\!\sim$ the quotient set, i.e., $\{[v]_\sim \mid v \in V\}$.

Given a set of variables V and a set of functions $\mathcal{F}$, let (E, D, P, B) be a tuple such that:

- $E \subseteq V \times V$ is an equivalence relation over V;
- $D \subseteq V/E \times V/E$ is a symmetric relation;
- P is a partial interpretation of the functions from $\mathcal{F}$ over the equivalence classes of E (for an r-ary function f, $P(f)$ is a partial map from $(V/E)^r$ to V/E);
- B is such that for an r-ary function f, $B(f)$ is map from $(V/E)^r$ to $\{\bot, \top\}$.

A *shared state* (resp. *stack state* of P_i for $i \in [n]$) is a tuple of the form (E, D, P, B) as above where $V = \texttt{V}$ (resp. $V = \texttt{V}_i$).

In our construction, along any execution, we aim to maintain a shared state (E, D, P, B) such that E tracks the occurred equivalences, D tracks the occurred inequalities, P captures the superterm relation among the currently stored terms, and B signals that some superterms of the currently stored terms have been already computed. Formally, we wish to maintain the following:

Invariants. For an execution ρ, variables $x, y, x_1, \ldots, x_r \in \texttt{Vars}$ and function $f \in \mathcal{F}$,

I1. $(x, y) \in E$ if and only if $\texttt{Comp}(\rho, x) \cong_{\alpha(\rho)} \texttt{Comp}(\rho, y)$;

I2. $([x]_E, [y]_E) \in D$ if and only if there are $t_0, t_1 \in \texttt{Terms}(\rho)$ s.t. $(t_0, t_1) \in \beta(\rho)$, and for $i \in \{0, 1\}$, $t_i \cong_{\alpha(\rho)} \texttt{Comp}(\rho, x)$ and $t_{1-i} \cong_{\alpha(\rho)} \texttt{Comp}(\rho, y)$;

I3. $P(f)([x_1]_E, \ldots, [x_r]_E) = [x]_E$ if and only if $t \cong_{\alpha(\rho)} f(t_1, \ldots, t_r)$ with $t = \texttt{Comp}(\rho, x)$ and $t_i = \texttt{Comp}(\rho, x_i)$ for $i \in [r]$;

I4. $B(f)([x_1]_E, \ldots, [x_r]_E) = \top$ if and only if there is a prefix ρ' of ρ s.t. $t \cong_{\alpha(\rho)} f(t_1, \ldots, t_r)$ with $t = \texttt{Comp}(\rho', z)$ for some $z \in \texttt{Vars}$ and $t_i = \texttt{Comp}(\rho, x_i)$ for $i \in [r]$.

4.1 The Mvpa $\mathcal{A}_{\texttt{Vars}}$

$\mathcal{A}_{\texttt{Vars}}$ uses n stacks, one for each component program P_i. The symbols of stack i are the stack states. The control states are $\bar{q}_{fs}$, $\bar{q}_{mem}$, $\bar{q}_{ea}$ and the shared states. Intuitively, $\bar{q}_{fs}$, $\bar{q}_{mem}$, and $\bar{q}_{ea}$ are entered when respectively the feasibility, memoizing and early assume property is violated by the input execution. The set of initial states is a singleton containing only the shared state $(E_0, \emptyset, P_0, B_0)$ where $E_0 = \{(x, x) \mid x \in \texttt{V}\}$, and for each $f \in \mathcal{F}$, $P_0(f)$ is undefined and $B_0(f)$ is the constant map assigning $\bot$. Let $\bar{Q}_{fs}$ be the set containing $\bar{q}_{fs}$ and all the shared states (E, D, P, B) such that D is not irreflexive. All the control states are accepting except for $\bar{q}_{mem}$, $\bar{q}_{ea}$ and all the states from $\bar{Q}_{fs}$. The transition relation is defined below.

As we go along with the description of the transitions, we also convey a proof by induction of the fulfillment of the invariants I1–I4 at the control states of the form (E, D, P, B) assuming that the concurrent execution that leads to such states is coherent. Indeed, in our proof we show a stronger property. In fact, we show that invariants I1–I4 hold with respect to $\texttt{V}$ (not only $\texttt{Vars}$) and additionally:

I5. for $i \in [n]$ and any execution of the form $\rho = \rho'.\langle \texttt{call } m\rangle_i.\rho''$ where ρ'' is i-matched, the stack state at the top of the stack for program P_i after reading ρ fulfills I1–I4 on ρ'.

The induction is on the length of the input execution. The base case, i.e., when the execution is empty, is a direct consequence of I1–I5 holding at the initial state of $\mathcal{A}_{\mathtt{Vars}}$.

Transitions from non-accepting states. The only transitions going out of the states $\bar{q}_{fs}$, $\bar{q}_{mem}$, and $\bar{q}_{ea}$ are transitions to themselves (sink rejecting states). From all states of the form (E, D, P, B) such that D is not irreflexive, it is only possible to reach $\bar{q}_{fs}$.

Internal transitions from accepting states. We describe the internal transitions from a control state q of the form (E, D, P, B) such that D is irreflexive. We start by analyzing two cases that can take to states that are not of the form (E', D', P', B'). On the input symbol $\langle x := f(x_1, \ldots, x_r)\rangle_i$, $\mathcal{A}_{\mathtt{Vars}}$ enters $\bar{q}_{mem}$ if $B(f)([x_1]_E, \ldots, [x_r]_E) = \top$ and $P(f)([x_1]_E, \ldots, [x_r]_E)$ is undefined (i.e., when we are trying to recompute a term that is not stored in any variables at this point of the execution and thus the memoizing property breaks). On the input symbol $\langle \mathtt{assume}(x = y)\rangle_i$, $\mathcal{A}_{\mathtt{Vars}}$ enters $\bar{q}_{ea}$ if there is a superterm t of either the term stored in x or the one stored in y that is stored in z and there is a function $f \in \mathcal{F}$ s.t. $B(f)([x_1]_E, \ldots, [x_r]_E) = \top$ but $P(f)([x_1]_E, \ldots, [x_r]_E)$ is undefined where z is one of $x_1, \ldots, x_r$ (i.e., we get evidence that no term equivalent to a previously computed superterm of those stored in either x or y is currently stored, and thus the early assume property breaks).

In the remaining cases, the internal transitions take to a state of the form (E', D', P', B') such that on input σ and with $i \in [n]$: if $\sigma = \langle \mathtt{assume}(x = y)\rangle_i$, we merge the equivalence classes of x and y and propagate equality on the stored superterms (by P), then update D, P and B according to the equivalence classes of E'; if $\sigma = \langle \mathtt{assume}(x \neq y)\rangle_i$, we just add $([x]_E, [y]_E)$ and $([y]_E, [x]_E)$ to the set of inequalities D; if $\sigma = \langle x := f(x_1, \ldots, x_r)\rangle_i$, we essentially move x to the equivalence class $P(f)([x_1]_E, \ldots, [x_r]_E)$ if defined and start a new one otherwise, remove the pairs of D involving x if x was the only variable of its class, and update P and B according to E'; the case $\sigma = \langle x := y\rangle_i$ is simpler than the previous one, we just need to remove the pairs of D involving x if x was the only variable of its class, merge the equivalence classes of x and y, and modify P and B accordingly. It is simple to see that the invariants I1–I5 are preserved.

Push and pop transitions. The only push and pop transitions are from control states of the form (E, D, P, B). As for the internal transitions, (E, D, P, B) is updated with the purpose of preserving the wished invariants. Additionally, on a call symbol of the form $\langle \mathtt{call}\ m\rangle_i$, the shadow variables of component program P_i are set to the terms currently stored in the corresponding program variables (by enforcing equality with these variables) and the current stack state (i.e., the restriction of the current control state to the variables $\mathtt{V}_i$) is pushed onto stack i. On a return symbol of the form $\langle \mathtt{w} := \mathtt{return}\rangle_i$, the stack state at the top of stack i is popped and merged with the current control state such that the resulting state relates: the terms at the beginning of the resumed method call (referred by the variables from $\mathtt{Vars}'_i$ in the popped state) to the rest of the terms of the current state (referred by the variables other than $\mathtt{Vars}'_i$ in the current control state). Below, we give the details only for the return transitions.

Let $\sigma = \langle \mathtt{w} := \mathtt{return}\rangle_i$ be the input symbol, $q = (E, D, P, B)$ be the current control state, m be the returned method and $q_\ell = (E_\ell, D_\ell, P_\ell, B_\ell)$ be the top symbol of stack i. From q, by reading σ and popping q_ℓ, $\mathcal{A}_{\mathtt{Vars}}$ moves to a control state $q' = (E', D', P', B')$ that is obtained as follows. We start by renaming in q_ℓ each variable x to $\underline{x}$. Denote $\underline{\mathtt{Vars}} = \{\underline{x} \mid x \in \mathtt{Vars}\}$ and $\underline{\mathtt{Vars}}'_i = \{\underline{x}' \mid x' \in \mathtt{Vars}'_i\}$ (recall that q_ℓ is over the set of variables $\mathtt{V}_i$). Then, we define q'' as the component-wise union of q and q_ℓ (i.e., we retain the equivalences, inequalities and map definitions from these states). Note that q'' is over the variables from $\mathtt{V} \cup \underline{\mathtt{Vars}} \cup \underline{\mathtt{Vars}}'_i$.

Now, we update q'' by assuming the equations $x' = \underline{x}$ for $x \in \mathtt{Vars}$ where $x' \in \mathtt{Vars}'_i$ and $\underline{x} \in \underline{\mathtt{Vars}}$ are the variables corresponding to x in the respective sets. In the resulting state we then drop all the variables from $\mathtt{Vars}'_i \cup \underline{\mathtt{Vars}}$ and rename back each variable $\underline{x}' \in \underline{\mathtt{Vars}}'_i$ to the corresponding variable $x' \in \mathtt{Vars}'_i$. The resulting state q_{mrg} is thus over the variables from $\mathtt{V}$. Finally, we get q' by updating this state according to the assignments $\mathtt{w} := \mathtt{o}_m$.

To argue the induction step in this case, let $\rho = \rho'.\langle \mathtt{call}\ m\rangle_i.\rho''$ where ρ'' is i-matched. From the induction hypothesis, q_ℓ fulfills I1–I4 restricted to $\mathtt{V}_i$ on ρ' (I5) and q fulfills I1–I4 on ρ. Thus, by effect of the equations $x' = \underline{x}$ that we assumed to get q_{mrg} from q'', we relate the valuation of each variable at the end of ρ' to: its valuation at the end of ρ for the variables listed in $\mathtt{w}$ (i.e., before assigning the terms returned by method m), and its valuation at the end of $\rho.\langle \mathtt{w} := \mathtt{return}\rangle_i$ for the remaining ones. Therefore, the state q_{mrg} fulfills I1–I4 on $\rho.\langle \mathtt{w}_m := \mathtt{return}\rangle_i$ except for assuming the valuation $\mathtt{Comp}(\rho, w)$ for w in $\mathtt{w}$. Finally, since q' is obtained from q_{mrg} through the assignments $\mathtt{w} := \mathtt{o}_m$, we get that q' fulfills I1–I4 on $\rho.\langle \mathtt{w}_m := \mathtt{return}\rangle_i$. Further, after the transition is taken, the stack state at the top of the stack must clearly fulfill I5. In fact, since on reading a call of P_i we push onto the corresponding stack the restriction of the control state to $\mathtt{V}_i$, by the inductive hypothesis we get that I1–I4 clearly holds up to that point of the computation.

Correctness. From the above arguments, the following lemma holds:

▶ **Lemma 4.** *Let ρ be a coherent concurrent execution over variables $\mathtt{Vars}$ and functions $\mathcal{F}$. If $\mathcal{A}_{\mathtt{Vars}}$ reaches a control state of the form (E, D, P, B) after reading ρ, then (E, D, P, B) satisfies the invariants I1–I4.*

By building on the results from [27] and the above lemma, we get:

▶ **Lemma 5.** *Let ρ be a coherent concurrent execution over variables $\mathtt{Vars}$ and functions $f \in \mathcal{F}$. For $\sigma \in \Pi$, the following holds:*

1. *ρ is infeasible iff $\mathcal{A}_{\mathtt{Vars}}$ enters a state in $\bar{Q}_{fs}$ on input ρ;*
2. *$\rho.\sigma$ is not memoizing iff $\mathcal{A}_{\mathtt{Vars}}$ enters $\bar{q}_{mem}$ on input $\rho.\sigma$;*
3. *$\rho.\sigma$ does not satisfy the early-assumes property iff $\mathcal{A}_{\mathtt{Vars}}$ enters $\bar{q}_{ea}$ on input $\rho.\sigma$.*

Since the only non-accepting states of $\mathcal{A}_{\mathtt{Vars}}$ are $\bar{q}_{mem}$, $\bar{q}_{ea}$ and all the states from $\bar{Q}_{fs}$, by inductively applying the above lemma we get:

▶ **Theorem 6.** *A concurrent computation ρ is accepted by $\mathcal{A}_{\mathtt{Vars}}$ if and only if ρ is coherent and feasible.*

By assuming that the signature has constant size, the number of different tuples of the form (E, D, P, B) over the set of variables $\mathtt{V}$ is $O(2^{|\mathtt{V}|^{O(1)}})$ where $|\mathtt{V}| = O(n\,|\mathtt{Vars}|)$, and thus, also the size of $\mathcal{A}_{\mathtt{Vars}}$ is $\mathtt{V}$ is $O(2^{|\mathtt{V}|^{O(1)}})$.

5 Checking bounded reachability and coherence

Fix a concurrent uninterpreted program $\mathcal{P}$ with component programs $P_1, \ldots, P_n$.

Reduction to Mvpa reachability. By standard constructions, it is possible to construct an Mvpa $\mathcal{A}_{\mathcal{P}}$ of size exponential in the number of components n that accepts all and only the complete executions of $\mathcal{P}$.

Since the stack operations are visible in the input alphabet, the intersection of two Mvpas is still an Mvpa that can be obtained by the cross product of the starting Mvpas [3, 24]. Denoting $\mathcal{A}_{\mathcal{P},\mathtt{Vars}}$ the Mvpa capturing the intersection of $\mathcal{A}_{\mathcal{P}}$ and $\mathcal{A}_{\mathtt{Vars}}$, the size of $\mathcal{A}_{\mathcal{P},\mathtt{Vars}}$ is $2^{|\mathtt{V}|^{O(1)}}$ (note that $|\mathtt{V}| \geq n$). Thus, by Theorem 6, we have:

▶ **Theorem 7.** *$\mathcal{A}_{\mathcal{P},\texttt{Vars}}$ has size $2^{|\texttt{V}|^{O(1)}}$ and accepts all and only the coherent and feasible executions of $\mathcal{P}$.*

Decidable bounded reachability problems. By restricting the executions with the bounding conditions given in Section 3, we obtain versions of MVPA's that have a decidable reachability problem. This along with Theorem 7 gives the decidability of the bounded reachability problem for concurrent uninterpreted programs under all the considered bounding conditions. Concerning to the computational complexity, we have the following upper-bounds (we denote with A an n-stack MVPA):

- from [30], the MVPA reachability problem within k context-switches can be solved in time $O(k^3|A|^5(n|Q|)^k)$ where Q denotes the set of control states of A, thus by Theorem 7 we get that CON-REACH can be decided in time exponential in the size of $\texttt{V}$ and k;
- from [24] we have that the MVPA reachability problem restricted to k-scoped executions can be decided in $O(2^n|Q|^{2kn+1})$ time, and thus by Theorem 7 we get that SCO-REACH can be decided in time exponential in the size of $\texttt{V}$ and k;
- from [2], we have that the MVPA reachability problem restricted to (k,d)-budget executions can be decided in time exponential in $(|A|+d+k)$; by encoding the stacks up to depth d into the control state, we can give a decision algorithm in the style of that given for the scope-bounded restriction that takes $O(2^n(|Q|+|\Gamma|^{nd})^{2kn+1})$ time, which gives for BUD-REACH an upper-bound that is exponential in size of $\texttt{V}$, d and k;
- from [23], we have that the MVPA reachability problem restricted to k-path-tree executions can be decided in time $2^{O(k(n+\log|A|))}$, thus by Theorem 7 we get that PAT-REACH can be decided in time exponential in the size of $\texttt{V}$ and k;
- from [20], we have that the MVPA reachability problem restricted to k-phase executions can be decided in time $2^{|A|2^{O(k)}}$, thus by Theorem 7 we get that PHA-REACH can be decided in time double exponential in the size of $\texttt{V}$ and k;
- from [4], we have that the MVPA reachability problem restricted to ordered executions can be decided in time $|A|^{2^{O(n)}}$, thus by Theorem 7 we get that ORD-REACH can be decided in time exponential in the size of $\texttt{V}$ and double exponential in n.

Since the reachability problem for sequential uninterpreted programs is EXPTIME-hard [27] and each instance of this problem is also an instance of CON-REACH, SCO-REACH, BUD-REACH and PAT-REACH, we have that all these problems are EXPTIME-complete. Moreover, since the reachability of MVPA restricted to ordered executions and bounded phase executions can be reduced to the respective problems for Boolean programs, and both these problems are 2EXPTIME-hard [4, 20], we have that PHA-REACH and ORD-REACH are 2EXPTIME-complete. Thus, we get the following theorem:

▶ **Theorem 8.** *The problems* CON-REACH, SCO-REACH, BUD-REACH *and* PAT-REACH *are EXPTIME-complete, and the problems* PHA-REACH *and* ORD-REACH *are 2EXPTIME-complete.*

Deciding coherence. Define $\mathcal{A}_{notco}$ as the MVPA obtained from $\mathcal{A}_{\texttt{Vars}}$ by removing the state $\bar{q}_{fs}$ and the transitions involving it, and making $\bar{q}_{mem}$ and $\bar{q}_{ea}$ its only accepting states. From Lemma 5, we get that $\mathcal{A}_{notco}$ accepts a concurrent execution ρ if and only if ρ is incoherent. Thus, to determine whether a program $\mathcal{P}$ is coherent we can just check $\mathcal{A}_{notco} \cap \mathcal{A}_{\mathcal{P}} = \emptyset$. Therefore, by the results on the considered bounding conditions introduced above, we get:

▶ **Theorem 9.** *Deciding coherence is EXPTIME-complete under* CON, SCO, BUD *and* PAT *restrictions and 2EXPTIME-complete under* PHA *and* ORD *restrictions.*

6 Conclusions

In this paper, we have shown the decidability of the reachability problem for concurrent uninterpreted programs under a number of restrictions that have been considered in the literature for the analysis of finite-domain concurrent programs. Our results do not extend directly to parametric uninterpreted programs, i.e., concurrent uninterpreted programs with executions formed of unboundedly many component programs (see [5, 21, 16, 22]). In fact, we crucially use in our reduction to MVPA to distinguish among the local variables of each component program. Also, known results on sequentializations (see [19, 25, 31]), i.e., code-to-code translations into non-deterministic sequential programs which (under certain assumptions) behave equivalently, do not seem to to work for uninterpreted programs as coherence suddenly breaks when rearranging the order of the statements. Both these directions deserve future investigation.

References

1 Parosh Aziz Abdulla, C. Aiswarya, and Mohamed Faouzi Atig. Data Multi-Pushdown Automata. In Roland Meyer and Uwe Nestmann, editors, *28th International Conference on Concurrency Theory, CONCUR 2017, September 5-8, 2017, Berlin, Germany*, volume 85 of *LIPIcs*, pages 38:1–38:17. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017. doi:10.4230/LIPIcs.CONCUR.2017.38.

2 Parosh Aziz Abdulla, Mohamed Faouzi Atig, Othmane Rezine, and Jari Stenman. Budget-bounded model-checking pushdown systems. *Formal Methods in System Design*, 45(2):273–301, 2014. doi:10.1007/s10703-014-0207-y.

3 Rajeev Alur and P. Madhusudan. Adding nesting structure to words. *J. ACM*, 56(3):16:1–16:43, 2009. doi:10.1145/1516512.1516518.

4 Mohamed Faouzi Atig, Benedikt Bollig, and Peter Habermehl. Emptiness of Ordered Multi-Pushdown Automata is 2ETIME-Complete. *Int. J. Found. Comput. Sci.*, 28(8):945–976, 2017. doi:10.1142/S0129054117500332.

5 Mohamed Faouzi Atig, Ahmed Bouajjani, and Shaz Qadeer. Context-Bounded Analysis For Concurrent Programs With Dynamic Creation of Threads. *Logical Methods in Computer Science*, 7(4), 2011. doi:10.2168/LMCS-7(4:4)2011.

6 Mikolaj Bojańczyk, Claire David, Anca Muscholl, Thomas Schwentick, and Luc Segoufin. Two-variable logic on data words. *ACM Trans. Comput. Log.*, 12(4):27:1–27:26, 2011. doi:10.1145/1970398.1970403.

7 Mikolaj Bojańczyk, Anca Muscholl, Thomas Schwentick, and Luc Segoufin. Two-variable logic on data trees and XML reasoning. *J. ACM*, 56(3):13:1–13:48, 2009. doi:10.1145/1516512.1516515.

8 Benedikt Bollig, Aiswarya Cyriac, Paul Gastin, and K. Narayan Kumar. Model Checking Languages of Data Words. In Lars Birkedal, editor, *Foundations of Software Science and Computational Structures - 15th International Conference, FOSSACS 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*, volume 7213 of *Lecture Notes in Computer Science*, pages 391–405. Springer, 2012. doi:10.1007/978-3-642-28729-9_26.

9 Ahmed Bouajjani, Michael Emmi, and Gennaro Parlato. On Sequentializing Concurrent Programs. In Eran Yahav, editor, *Static Analysis - 18th International Symposium, SAS 2011, Venice, Italy, September 14-16, 2011. Proceedings*, volume 6887 of *Lecture Notes in Computer Science*, pages 129–145. Springer, 2011. doi:10.1007/978-3-642-23702-7_13.

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

11 Luca Breveglieri, Alessandra Cherubini, Claudio Citrini, and Stefano Crespi-Reghizzi. Multi-Push-Down Languages and Grammars. *Int. J. Found. Comput. Sci.*, 7(3):253–292, 1996. doi:10.1142/S0129054196000191.

12 Denis Bueno and Karem A. Sakallah. euforia: Complete Software Model Checking with Uninterpreted Functions. In Constantin Enea and Ruzica Piskac, editors, *Verification, Model Checking, and Abstract Interpretation - 20th International Conference, VMCAI 2019, Cascais, Portugal, January 13-15, 2019, Proceedings*, volume 11388 of *Lecture Notes in Computer Science*, pages 363–385. Springer, 2019. doi:10.1007/978-3-030-11245-5_17.

13 Aiswarya Cyriac, Paul Gastin, and K. Narayan Kumar. MSO decidability of multi-pushdown systems via split-width. In Maciej Koutny and Irek Ulidowski, editors, *CONCUR 2012 - Concurrency Theory - 23rd International Conference, CONCUR 2012, Newcastle upon Tyne, UK, September 4-7, 2012. Proceedings*, volume 7454 of *Lecture Notes in Computer Science*, pages 547–561. Springer, 2012. doi:10.1007/978-3-642-32940-1_38.

14 Claire David, Leonid Libkin, and Tony Tan. On the Satisfiability of Two-Variable Logic over Data Words. In Christian G. Fermüller and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning - 17th International Conference, LPAR-17, Yogyakarta, Indonesia, October 10-15, 2010. Proceedings*, volume 6397 of *Lecture Notes in Computer Science*, pages 248–262. Springer, 2010. doi:10.1007/978-3-642-16242-8_18.

15 Martin Davis. Kurt Gödel. Über Die Vollständigkeit des Logikkalküls . Collected Works, Volume I, Publications 1929–1936, by Kurt Gödel, Edited by Solomon Feferman, John W. Dawson Jr., Stephen C. Kleene, Gregory H. Moore, Robert M. Solovay, and Jean van Heijenoort, Clarendon Press, Oxford University Press, New York and Oxford, 1986, Even Pp. 60– 100. - Kurt Gödel. On the Completeness of the Calculus of Logic . English Translation by Stefan Bauer-Mengelberg and Jean van Heijenoort of the Preceding. *Journal of Symbolic Logic*, 55(1):341–342, 1990. doi:10.2307/2274974.

16 Antoine Durand-Gasselin, Javier Esparza, Pierre Ganty, and Rupak Majumdar. Model checking parameterized asynchronous shared-memory systems. *Formal Methods in System Design*, 50(2-3):140–167, 2017. doi:10.1007/s10703-016-0258-3.

17 Michael Emmi, Shaz Qadeer, and Zvonimir Rakamaric. Delay-bounded scheduling. In Thomas Ball and Mooly Sagiv, editors, *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, pages 411–422. ACM, 2011. doi:10.1145/1926385.1926432.

18 Sumit Gulwani and Ashish Tiwari. Assertion Checking Unified. In Byron Cook and Andreas Podelski, editors, *Verification, Model Checking, and Abstract Interpretation, 8th International Conference, VMCAI 2007, Nice, France, January 14-16, 2007, Proceedings*, volume 4349 of *Lecture Notes in Computer Science*, pages 363–377. Springer, 2007. doi:10.1007/978-3-540-69738-1_26.

19 Omar Inverso, Ermenegildo Tomasco, Bernd Fischer, Salvatore La Torre, and Gennaro Parlato. Bounded Model Checking of Multi-threaded C Programs via Lazy Sequentialization. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, volume 8559 of *Lecture Notes in Computer Science*, pages 585–602. Springer, 2014. doi:10.1007/978-3-319-08867-9_39.

20 Salvatore La Torre, P. Madhusudan, and Gennaro Parlato. A Robust Class of Context-Sensitive Languages. In *22nd IEEE Symposium on Logic in Computer Science (LICS 2007), 10-12 July 2007, Wroclaw, Poland, Proceedings*, pages 161–170. IEEE Computer Society, 2007. doi:10.1109/LICS.2007.9.

21 Salvatore La Torre, P. Madhusudan, and Gennaro Parlato. Model-Checking Parameterized Concurrent Programs Using Linear Interfaces. In Tayssir Touili, Byron Cook, and Paul B. Jackson, editors, *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings*, volume 6174 of *Lecture Notes in Computer Science*, pages 629–644. Springer, 2010. doi:10.1007/978-3-642-14295-6_54.

22 Salvatore La Torre, Anca Muscholl, and Igor Walukiewicz. Safety of Parametrized Asynchronous Shared-Memory Systems is Almost Always Decidable. In Luca Aceto and David de Frutos-Escrig, editors, *26th International Conference on Concurrency Theory, CONCUR 2015, Madrid, Spain, September 1.4, 2015*, volume 42 of *LIPIcs*, pages 72–84. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015. doi:10.4230/LIPIcs.CONCUR.2015.72.

23 Salvatore La Torre, Margherita Napoli, and Gennaro Parlato. A Unifying Approach for Multistack Pushdown Automata. In Erzsébet Csuhaj-Varjú, Martin Dietzfelbinger, and Zoltán Ésik, editors, *Mathematical Foundations of Computer Science 2014 - 39th International Symposium, MFCS 2014, Budapest, Hungary, August 25-29, 2014. Proceedings, Part I*, volume 8634 of *Lecture Notes in Computer Science*, pages 377–389. Springer, 2014. doi:10.1007/978-3-662-44522-8_32.

24 Salvatore La Torre, Margherita Napoli, and Gennaro Parlato. Scope-Bounded Pushdown Languages. *Int. J. Found. Comput. Sci.*, 27(2):215–234, 2016. doi:10.1142/S0129054116400074.

25 Akash Lal and Thomas W. Reps. Reducing concurrent analysis under a context bound to sequential analysis. *Formal Methods in System Design*, 35(1):73–97, 2009. doi:10.1007/s10703-009-0078-9.

26 P. Madhusudan and Gennaro Parlato. The tree width of auxiliary storage. In Thomas Ball and Mooly Sagiv, editors, *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, pages 283–294. ACM, 2011. doi:10.1145/1926385.1926419.

27 Umang Mathur, P. Madhusudan, and Mahesh Viswanathan. Decidable verification of uninterpreted programs. *PACMPL*, 3(POPL):46:1–46:29, 2019. URL: https://dl.acm.org/citation.cfm?id=3290359, doi:10.1145/3290359.

28 Umang Mathur, Adithya Murali, Paul Krogmeier, P. Madhusudan, and Mahesh Viswanathan. Deciding Memory Safety for Forest Datastructures. *CoRR*, abs/1907.00298, 2019. arXiv:1907.00298.

29 Andrzej S. Murawski, Steven J. Ramsay, and Nikos Tzevelekos. Reachability in pushdown register automata. *J. Comput. Syst. Sci.*, 87:58–83, 2017. doi:10.1016/j.jcss.2017.02.008.

30 Shaz Qadeer and Jakob Rehof. Context-Bounded Model Checking of Concurrent Software. In Nicolas Halbwachs and Lenore D. Zuck, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 11th International Conference, TACAS 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings*, volume 3440 of *Lecture Notes in Computer Science*, pages 93–107. Springer, 2005. doi:10.1007/978-3-540-31980-1_7.

31 Shaz Qadeer and Dinghao Wu. KISS: keep it simple and sequential. In William Pugh and Craig Chambers, editors, *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation 2004, Washington, DC, USA, June 9-11, 2004*, pages 14–24. ACM, 2004. doi:10.1145/996841.996845.

32 Nikos Tzevelekos. Fresh-register automata. In Thomas Ball and Mooly Sagiv, editors, *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, pages 295–306. ACM, 2011. doi:10.1145/1926385.1926420.