A Fixpoint Calculus for Local and Global Program Flows

Rajeev Alur  
*University of Pennsylvania, alur@cis.upenn.edu*

Swarat Chaudhuri  
*University of Pennsylvania*

P. Madhusudan  
*University of Illinois at Urbana-Champaign*

Follow this and additional works at: [http://repository.upenn.edu/cis_papers](http://repository.upenn.edu/cis_papers)

Recommended Citation  
[http://dx.doi.org/10.1145/1111037.1111051](http://dx.doi.org/10.1145/1111037.1111051)
A Fixpoint Calculus for Local and Global Program Flows

Abstract
We define a new fixpoint modal logic, the visibly pushdown $\mu$-calculus ($VP-\mu$), as an extension of the modal $\mu$-calculus. The models of this logic are execution trees of structured programs where the procedure calls and returns are made visible. This new logic can express pushdown specifications on the model that its classical counterpart cannot, and is motivated by recent work on visibly pushdown languages [4]. We show that our logic naturally captures several interesting program specifications in program verification and dataflow analysis. This includes a variety of program specifications such as computing combinations of local and global program flows, pre/post conditions of procedures, security properties involving the context stack, and interprocedural dataflow analysis properties. The logic can capture flow-sensitive and inter-procedural analysis, and it has constructs that allow skipping procedure calls so that local flows in a procedure can also be tracked. The logic generalizes the semantics of the modal $\mu$-calculus by considering summaries instead of nodes as first-class objects, with appropriate constructs for concatenating summaries, and naturally captures the way in which pushdown models are model-checked. The main result of the paper is that the model-checking problem for VP-$\mu$ is effectively solvable against pushdown models with no more effort than that required for weaker logics such as CTL. We also investigate the expressive power of the logic VP-$\mu$: we show that it encompasses all properties expressed by a corresponding pushdown temporal logic on linear structures (caret [2]) as well as by the classical $\mu$-calculus. This makes VP-$\mu$ the most expressive known program logic for which algorithmic software model checking is feasible. In fact, the decidability of most known program logics ($\mu$-calculus, temporal logics LTL and CTL, caret, etc.) can be understood by their interpretation in the monadic second-order logic over trees. This is not true for the logic VP-$\mu$, making it a new powerful tractable program logic.

Disciplines
Computer Sciences
A Fixpoint Calculus for Local and Global Program Flows

Rajeev Alur
University of Pennsylvania
alur@cis.upenn.edu

Swarat Chaudhuri
University of Pennsylvania
swarat@cis.upenn.edu

P. Madhusudan
University of Illinois, Urbana-Champaign
madhu@cs.uiuc.edu

Abstract

We define a new fixpoint modal logic, the visibly pushdown \( \mu \)-calculus (VP-\( \mu \)), as an extension of the modal \( \mu \)-calculus. The models of this logic are execution trees of structured programs where the procedure calls and returns are made visible. This new logic can express pushdown specifications on the model that its classical counterpart cannot, and is motivated by recent work on visibly pushdown languages [4]. We show that our logic naturally captures several interesting program specifications in program verification and dataflow analysis. This includes a variety of program specifications such as computing combinations of local and global program flows, pre/post conditions of procedures, security properties involving the context stack, and interprocedural dataflow analysis properties. The logic can capture flow-sensitive and interprocedural analysis, and it has constructs that allow skipping procedure calls so that local flows in a procedure can also be tracked. The logic generalizes the semantics of the modal \( \mu \)-calculus by considering summaries instead of nodes as first-class objects, with appropriate constructs for concatenating summaries, and naturally captures the way in which pushdown models are model-checked. The main result of the paper is that the model-checking problem for VP-\( \mu \) is effectively solvable against pushdown models with no more effort than that required for weaker logics such as CTL. We also investigate the expressive power of the logic VP-\( \mu \): we show that it encompasses all properties expressed by a corresponding pushdown temporal logic on linear structures (CARET [2]) as well as by the classical \( \mu \)-calculus. This makes VP-\( \mu \) the most expressive known program logic for which algorithmic software model checking is feasible. In fact, the decidability of most known program logics (\( \mu \)-calculus, temporal logics LTL and CTL, CARET, etc.) can be understood by their interpretation in the monadic second-order logic over trees. This is not true for the logic VP-\( \mu \), making it a new powerful tractable program logic.

Categories and Subject Descriptors D.2.4 [Software Engineering]: Software/Program Verification—Model checking; F.3.1 [Theory of Computation]: Specifying and Verifying and Reasoning about Programs; F.4.1 [Theory of Computation]: Mathematical Logic—Temporal logic

General Terms Algorithms, Theory, Verification

Keywords Logic, specification, verification, \( \mu \)-calculus, infinite-state, model-checking, games, pushdown systems

1. Introduction

The \( \mu \)-calculus [20, 16] is a modal logic with fixpoints interpreted over labeled transition systems, or equivalently, over their tree unfoldings. It is an extensively studied specification formalism with applications to program analysis, computer-aided verification, and database query languages [13, 25]. From a theoretical perspective, its status as the canonical temporal logic for regular requirements is due to the fact that its expressiveness exceeds that of all commonly used temporal logics such as LTL, CTL, and CTL*, and equals that of alternating parity tree automata or the bisimulation-closed fragment of monadic second-order theory over trees [14, 18]. From a practical standpoint, iterative computation of fixpoints naturally suggests symbolic evaluation, and symbolic model checkers such as SMV check CTL properties of finite-state models by compiling them into \( \mu \)-calculus formulas [8, 21].

In this paper, we focus on the role of \( \mu \)-calculus to specify properties of labeled transition systems corresponding to pushdown automata, or equivalently, Boolean programs [5] or recursive state machines (RSMs) [3, 7]. Such pushdown models can capture the control flow in typical sequential imperative programming languages with recursive procedure calls, and are central to interprocedural dataflow analysis [22] and software model checking [6, 17]. While algorithmic verification of \( \mu \)-calculus properties of such models is possible [26, 10], classical \( \mu \)-calculus cannot express pushdown specifications that require inspection of the stack or matching of calls and returns. Even though the general problem of checking pushdown properties of pushdown automata is undecidable, algorithmic solutions have been proposed for checking many different kinds of non-regular properties [19, 12, 15, 11, 2, 4]. These include access control requirements such as “a module \( A \) should be invoked only if the module \( B \) belongs to the call-stack,” bounds on stack size such as “after any point where \( p \) holds, the number of interrupt-handlers in the call-stack should never exceed \( S \)” and the classical Hoare-style correctness requirements of program modules with pre- and post-conditions, such as “if \( p \) holds when a module is invoked, the module must return, and \( q \) must hold on return”.

In the program analysis literature, it has been argued that data flow analysis, such as the computation of live variables and very busy expressions, can be viewed as evaluating \( \mu \)-calculus formulas over abstractions of programs [24, 23]. This correspondence does not hold when we need to account for local data flow paths. For instance, for an expression \( e \) that involves a variable local to a procedure \( P \), the set of control points within \( P \) at which \( e \) is very
can express such pushdown requirements and yet has a decidable
mation. The goal of this paper is to identify a fixpoint calculus that
though interprocedural dataflow analysis can compute this infor-
busy (that is, $e$ is guaranteed to be used before any of its variables
get modified), cannot be specified using a $\mu$-calculus formula even
tough interprocedural dataflow analysis can compute this infor-
properties \[4\]. In this variation of pushdown automata over words,
sresulting class of languages is closed under union, intersection, and
complementation, and problems such as inclusion that are unde-
cidable for context-free languages are decidable for visibly push-
down automata. This implies that checking pushdown properties of
pushdown models is feasible as long as the calls and returns are
made \textit{visible} allowing the stacks of the property and the model to
synchronize. This visibility requirement seems only natural while
writing requirements about pre/post conditions or for interprocedu-
flow properties. The linear-time temporal logic CARET is based
on the same principle: its formulas are interpreted over sequences
tagged with calls and returns, and its syntax includes for each tem-
poral modality, besides its classical global version, a local version
that jumps from a call-state to the matching return-state, and thus,
can express non-local properties, without causing undecidability.

In order to develop a \textit{visibly pushdown branching-time logic}, we
consider \textit{structured trees} as models. In a structured tree, nodes are
labeled with atomic propositions as in Kripke models, and edges are
tagged as \textit{call}, \textit{return}, or \textit{local}. To associate a structured tree
with a program (or its abstraction), we must choose the set of ob-
servable atomic state properties, tag edges corresponding to calls
and returns from program blocks appropriately, and then take the
tree unfolding of this abstract program model. The abstract model
can be an abstraction of the program at any level of abstraction:
from the skeletal control-flow graph to boolean predicate abstrac-
tions of programs.

We define the \textit{visibly pushdown $\mu$-calculus} (VP-$\mu$) over struc-
tured trees. The variables of the calculus evaluate not over sets of
states, but rather over sets of subtrees that capture summaries of
computations in the “current” program block. The tippoint oper-
ators in the logic then compute fixpoints of summaries. For a given
state $s$ of a structured tree, consider the subtree rooted at $s$ such
that the leaves correspond to exits from the current block: differ-
ent paths in the subtree correspond to different computations of
the program, and the first unmatched return edge along a path leads to a
leaf (some paths may be infinite corresponding to cycles that never
return in the abstracted program). In order to be able to relate paths in
this subtree to the trees rooted at the leaves, we allow marking of the
leaves: a 1-ary summary is specified by the root $s$ and a subset
$U$ of the leaves of the subtree rooted at $s$. Each formula of the logic
is evaluated over such a summary. The central construct of the logic
responds to concatenation of call trees: the formula $(\text{call})\varphi_1(\psi_1)$(\text{call})\varphi_1(\psi_1)$ holds at a summary $(s, U)$ if the state $s$ has a call-edge to a state $t$
and there exists a summary $(t, V)$ satisfying $\varphi$ and for each leaf $v$
that belongs to $V$, the subtree $(v, U)$ satisfies $\psi$.

Our logic is best explained using the specification of local reach-
ability: let us identify the set of all summaries $(s, U)$ such that there
is a \textit{local} path from $s$ to some node in $U$ (i.e. all calls from the
initial procedure must have returned before reaching $U$). In our
logic, this is written as the formula $\varphi = \mu X. (r \in R_1 \land \langle \text{loc} \rangle X \land (\text{call})X \land X)$. The above means that $X$ is the smallest set of sum-
maries of the form $(s, U)$ such that (1) there is a $r$-labeled edge from
$s$ to some node in $U$, (2) there is a $l$-labeled edge from $s$ to $t$
and there is a summary $(t, U)$ in $X$, or (3) there is a call-labeled
edge from $s$ to $t$ and a summary $(t, V)$ in $X$ such that from each
$v \in V$, $(v, U)$ is a summary in $X$. Notice that the above formula
identifies the summaries in the natural way it will be \textit{computed}
on a pushdown system: compute the local summaries of each proce-
dure, and update the reachability relation using the call-to-return
summaries found in the procedures called.

Using the above formula, we can state local reachability of a state satisfying $p$ as: $\mu Y. (p \lor (\langle \text{loc} \rangle Y \lor (\text{call})Y))$ which intuitively states that $Y$ is the set of summaries $(s, U)$ where there
is a local path from $s$ to $U$ that goes through a state satisfying $p$. The
initial summary (involving the initial state of the program) satisfies the
formula only if a $p$-labeled state is reachable in the top-most
context, which cannot be stated in the standard $\mu$-calculus. This
example also illustrates how local flows in the context of dataflow
analysis can be captured using our logic.

In general, we allow markings of the leaves with $k$ colors: a
$k$-colored summary rooted at a node consists of $k$ subsets of the
leaves of the subtree rooted at this node. The $k$-ary concatenation
formula $(\text{call})_k\varphi_1(\psi_1, \ldots, \psi_k)$ says that the called procedure should satisfy $\varphi$, and the subtrees at the return nodes labeled with color $i$ should satisfy the requirement $\psi_i$. While the concatenation op-
eration is a powerful recursive construct that allows the logic to
express pushdown properties, multiple colors allows expression of
branching-time properties that can propagate between the called
and the calling contexts.

The main result of this paper is that the logic VP-$\mu$ can be
model-checked effectively. Given a model of a program as a re-
cursive state machine \[3, 7\], or equivalently a pushdown system, and a VP-$\mu$ formula $\varphi$, we show that we can model-check whether the
tree unfolding of the model satisfies $\varphi$ in exponential time (the
procedure is exponential in both the formula and the model). For a
fixed formula $\varphi$, however, the model-checking problem is only
polynomial in the number of states in the model and exponential
in the number of control locations where a procedure in the model
may return. The model-checking algorithm works by computing
fix-points of the summary sets inductively, and illustrates how the
semantics of the logic naturally suggests a model-checking algo-

The complexity of model-checking VP-$\mu$ is \textit{EXPTIME}-
complete, which matches the complexity of model-checking the
standard $\mu$-calculus on pushdown systems (in fact, model-checking
alternating reachability properties is already \textit{EXPTIME} hard \[26\]).

Finally, we study some expressiveness issues for the logic VP-
$\mu$. We first show that VP-$\mu$ captures the temporal logic CARET,
which is a linear-time temporal logic over visibly pushdown words
that can capture several interesting pushdown specification proper-
ties. This shows that our branching-time logic captures the relevant
counterpart logic over linear models, much the same way as the
standard $\mu$-calculus captures the temporal logic LTL. This makes
VP-$\mu$ the most expressive known specification logic of programs
with a decidable model checking problem with respect to Boolean
programs.

We also show that the notion of $k$-colors in the logic is important
by proving a \textit{hierarchy theorem}: formulas of VP-$\mu$ that use $k$ colors
are strictly weaker than formulas that use $(k + 1)$ colors. Finally,
we show that the satisfiability problem for VP-$\mu$ is undecidable.
Note that this is not an issue as we are really only interested in the
model-checking problem; in fact the result serves to illustrate how
powerful the logic VP-$\mu$ is.

The paper is organized as follows. Section 2 introduces struc-
tured trees and summaries and Section 3 defines the logic VP-$\mu$.
In Section 4 we present various properties that can be expressed
using VP-$\mu$, including reachability, local reachability, expressions for
various temporal modalities like \textit{eventually} and \textit{until}, security
properties that involve inspection of stack, stack overflow proper-
ties, properties describing pre and post-conditions for procedures,
properties of access control and some data-flow analysis properties
such as *very busy expressions*. Section 5 shows how recursive state machine models of programs can be model-checked against VP-$\mu$ formulas, Section 6 contains results on expressiveness and undecidability of satisfiability, and we conclude with some discussion in Section 7.

2. Structured trees

Let $AP$ be a finite set of atomic propositions, and $I = \{\text{call, ret, loc}\}$ a fixed set of tags. We are interested in trees whose nodes and edges are respectively labeled by propositions and tags, and model abstract states and statements in sequential, structured, possibly recursive programs. Formally, an $(AP, I)$-labeled tree is a tuple $S = (S, s_0, E, \lambda, \eta)$, where $(S, s_0, E)$ is a tree with node set $S$, root node $s_0$ and edge relation $E$, the node-labeling function $\lambda : S \rightarrow 2^AP$ labels nodes with sets of propositions they satisfy, and the transition-labeling function $\eta : E \rightarrow I$ tags transitions as procedure calls (labeled by $\text{call}$), procedure returns (ret), or local statements within procedures ($\text{loc}$). For $a \in I$, we write $s \xrightarrow{a} s'$ as shorthand for $"(s, s') \in E \text{ and } \eta((s, s')) = a."$

A finite path in an $(AP, I)$-labeled tree is a sequence $\pi = s_1 s_2 \ldots s_n$ over $S$ such that $(s_i, s_{i+1}) \in E$ for all $1 \leq i \leq n$. We will extend $\eta$ to paths in $S$ as follows. Let $e_i$ represent the transition $(s_i, s_{i+1})$ in the above path $\pi$. Then $\eta(\pi)$ is the word $\eta(e_1) \eta(e_2) \ldots \eta(e_{n-1})$ over the alphabet $I$.

Such a labeling lets us mark certain paths in $S$ as matched. A path $\pi$ in $S$ is called matched if and only if $w = \eta(\pi)$ is of the form $w := \text{loc} \lor \text{call} \lor \text{ret} \lor \text{vw}$.

Given nodes $s$ and $s'$ in $S$, we call $s'$ a matching return of $s$ if and only if there is a matched path $\pi = s \circ s_1 \circ \ldots \circ s_n$ such that $s_n \xrightarrow{\text{ret}} s'$. Intuitively, $s'$ models the first state that the underlying program reaches on popping the context of $s$ off its stack frame. The set of matching returns of $s$ is written as $MR(s)$. Then:

**Definition 1.** A structured tree over $AP$ is an $(AP, I)$-labeled tree with root $s_0$ such that $MR(s_0) = \emptyset$.

*Figure 1.* (a) A structured tree (b) A 2-colored summary

Intuitively, paths from the root in structured trees do not have “extra” returns that do not match any call—a structured tree models the branching behavior of a program from a state $s$ to, at most, the end of the procedural context where $s$ lies. Also observe that the maximal subtree rooted at an arbitrary node in a structured tree is not, in general, structured. Fig. 1-a shows a structured tree, with nodes $s_1, \ldots, s_{15}$ and transitions labeled $\text{call, ret and loc}$. Some of the nodes are labeled by propositions $p$ and $q$. Note particularly the matching return relation; for instance, the nodes $s_{10}, s_{11}, s_{12}$, and $s_{13}$ are matching returns for the node $s_2$. Also, $MR(s_1) = \emptyset$.

2.1 Summaries

We are interested in subtrees of structured trees wholly contained within “procedural” contexts; such a subtree models the branching behavior of a program from a state $s$ to each return point of its context. Each such subtree rooted at $s$ has a *summarizing* comprising (1) the node $s$, and (2) the set of all nodes that are reached on return from its context, i.e., $MR(s)$. Also, in order to demand different temporal requirements at different returns for a context, we introduce a *coloring of nodes* in $MR(s)$—intuitively, a return gets color $i$ if it is to satisfy the $i$-th requirement. Note that such colored summaries are defined for all $s$ and that, in particular, we do not require $s$ to be an “entry” node of a procedure. Sets of such summaries define the semantics of formulas in VP-$\mu$.

Formally, for a non-negative integer $k$, a *$k$-colored summary* $s$ is a tuple $(s, U_1, U_2, \ldots, U_k)$, where $s \in S$ and $U_1, U_2, \ldots, U_k \subseteq MR(s)$. For example, in Fig. 1-a, $(s_1)$ is a valid 0-colored summary, and $(s_2, \{s_{11}, s_{12}\}, \{s_{10}, s_{12}\})$ and $(s_3, \{s_6\})$ are valid 2-colored summaries. The set of all summaries in $S$, each $k$-colored for some $k$, is denoted by $S$.

Observe how each summary describes a subtree along with a coloring of some of its leaves. For instance, the summary $s = (s_2, \{s_{11}, s_{12}\}, \{s_{10}, s_{12}\})$ marks the subtree in Fig. 1-b. Such a tree may be constructed by taking the subtree of $S$ rooted at node $s_2$, and chopping off the subtrees rooted at $MR(s_2)$. Note that because of unmatched infinite paths from the root, such a tree may in general be infinite. Now, nodes $s_{11}$ and $s_{12}$ are assigned the color 1, and nodes $s_{10}$ and $s_{12}$ are colored 2. The node $s_{12}$ is not colored.

Also, note that in the linear-time setting, a pair $(s, s')$, where $s' \in MR(s)$, would suffice as a summary, and that this is the way in which traditional summarization-based decision procedures have defined summaries. On the other hand, for branching-time reasoning, such a simple definition is not enough.

3. A fixpoint calculus of calls and returns

3.1 Syntax

In addition to being interpreted over summaries, the logic VP-$\mu$ differs from classical calculi like the modal $\mu$-calculus [20] in a crucial way: its syntax and semantics explicitly recognize the procedural structure of programs via modalities $\text{call, ret and loc}$. A distinction is made between $\text{call}$-edges, along which a program pushes frames on its stack, $\text{ret}$-edges, which require a pop from the stack, and $\text{loc}$-edges, which change the program counter and local and global store without modifying the stack. Also, in order to enforce different “return conditions” at differently colored returns in a summary, it can pass formulas as “parameters” to $\text{call}$ modalities. Formally, let $AP$ be a finite set of atomic propositions, $Var$ be a finite set of variables, and $\{R_1, R_2, \ldots\}$ be a set of markers. Then, for $p \in AP$ and $X \in Var$, formulas $\varphi$ of VP-$\mu$ are defined by:

$$\varphi ::= p \mid \neg p \mid X \lor \varphi \lor \varphi \land \varphi \mid \mu X.\varphi \lor \nu X.\varphi \mid \langle \text{call} \rangle \varphi(\psi_1, \psi_2, \ldots, \psi_k) \mid \langle \text{call} \rangle \varphi(\psi_1, \psi_2, \ldots, \psi_k) \mid \langle \text{loc} \rangle \varphi \mid [\langle \text{loc} \rangle \varphi \mid [\langle \text{ret} \rangle R_1] \mid [\langle \text{ret} \rangle R_i]$$

where $k \geq 0$ and $i \geq 1$. Let us define the syntactic shorthands $tt = \varphi \lor \neg p$ and $ff = \varphi \land \neg p$ for some $p \in AP$. Also, let the arity of a VP-$\mu$ formula $\varphi$ be the maximum $k$ such that $\varphi$ has a subformula of the form $(\langle \text{call} \rangle \varphi(\psi_1, \ldots, \psi_k))$ or $[\langle \text{call} \rangle \varphi(\psi_1, \ldots, \psi_k)]$.

Intuitively, the markers $R_i$ in a formula are bound by $(\langle \text{call} \rangle$ modalities, and variables $X$ are bound by fixpoint quantifiers $\mu X$ and $\nu X$. We require our $\varphi$-formulas to bind all the markers in their scope. Formally, let the maximum marker index $\text{ind}(\varphi)$ of a formula $\varphi$ be defined inductively as: $\text{ind}(\varphi_1 \lor \varphi_2) = \max\{\text{ind}(\varphi_1), \text{ind}(\varphi_2)\}$; $\text{ind}(\langle \text{loc} \rangle \varphi) = \text{ind}(\langle \text{loc} \rangle \varphi) = \text{ind}(\mu X.\varphi) = \text{ind}(\nu X.\varphi) = \text{ind}(\varphi)$; and
To understand the semantics of local ($S_f$), we could have formulas of the form $\varphi = \langle \text{call} \varphi' \rangle \langle \psi_1, \psi_2, \ldots, \psi_k \rangle$, where $\psi_i$ are arbitrary VP-$\mu$ formulas. To see what this means, look at the summaries $r_1 = \langle s_6, \emptyset, \{s_12\} \rangle$ and $r_2 = \langle s_8, \{s_11\}, \emptyset \rangle$, which capture flow (under the assumed coloring of $MR(s_2)$) from $s_6$ and $s_8$ to the end of the context they are in. To see if $\varphi$ is satisfied, we will need to consider a summary $s''$ rooted at $s_3$ where the color $i$ is assigned to nodes $s_6$ and $s_8$ precisely when $r_1$ and $r_2$ respectively satisfy $\psi_i$. Now, we require $s''$ to satisfy $\varphi'$.

So far as the structures of these trees go, we find that the above requires a split of the tree $S_f$ for summary $s''$ in the way shown in Fig. 2-b. The root of this tree must have a call-edge to the root of the tree for $s'$, which must satisfy $\varphi$. At each leaf of $S_{f,c}$ colored $i$, we must be able to concatenate a summary tree $S_f$ satisfying $\psi_i$ such that (1) every leaf in $S_f$ is a leaf of $S_f$, and (2) each such leaf gets the same color set in $S_f$ and $S_{f,c}$.

As for the return modalities, we use them to assert that we return at a point colored $i$. Because the binding of these colors to temporal requirements was fixed at a context that called the current context, the ret-modalities let us relate a path in the latter with the continuation of a path in the former. For instance, in Fig. 2-c, where the rectangle abstracts the part of a program unfolding within the body of a procedure $\foo$, the marking of return points $s_1$ and $s_2$ by colors 1 and 2 is visible inside $\foo$ as well as at the call site of $\foo$. This lets us match paths $P_1$ and $P_2$ inside $\foo$ respectively with paths $P'_{1}$ and $P'_{2}$ in the calling procedure. This lets VP-$\mu$ capture the pushdown structure of branching-time runs of a procedural program.

Let us now describe the semantics of VP-$\mu$ formally. A VP-$\mu$ formula $\varphi$ is interpreted in an environment $E$ that interprets variables in $Free(\varphi)$ as sets of summaries in a structured tree $S$. Formally, an environment is a map $E : Free(\varphi) \rightarrow 2^S$. Let us write $[\varphi]^S_E$ to denote the set of summaries in $S$ satisfying $\varphi$ in environment $E$ (usually $S$ will be understood from the context, and we will simply write $[\varphi]^E$). For a summary $s = \langle s, U_1, U_2, \ldots, U_k \rangle$, where $s \in S$ and $U_i \subseteq MR(s)$ for all $i$, $s$ satisfies $\varphi$, i.e., $s \in [\varphi]^E$, if and only if one of the following holds:

- $\varphi = p \in AP$ and $p \in \lambda(s)$
- $\varphi = \neg p$ for some $p \in AP$, and $p \notin \lambda(s)$
- $\varphi = X$, and $s \in E(X)$
- $\varphi = \varphi_1 \land \varphi_2$ such that $s \in [\varphi_1]^E$ or $s \in [\varphi_2]^E$
- $\varphi = \forall X.\varphi$ such that $s \in [\varphi_1]^E$ and $s \in [\varphi_2]^E$
- $\varphi = \langle call \varphi' \rangle \langle \psi_1, \psi_2, \ldots, \psi_k \rangle$, and there is a $t \in S$ such that $s \xrightarrow{\text{call}} t$, and (2) the summary $t = \langle t, V_1, V_2, \ldots, V_m \rangle$, where for all $1 \leq i \leq m$, $V_i = MR(t) \cap \{s' : \langle s', U_1 \cap MR(s'), \ldots, U_k \cap MR(s') \rangle \in [\psi_i]^E\}$, is such that $t \in [\varphi]^E$
- $\varphi = \langle call \varphi' \rangle \langle \psi_1, \psi_2, \ldots, \psi_k \rangle$, and for all $t \in S$ such that $s \xrightarrow{\text{call}} t$, the summary $t = \langle t, V_1, V_2, \ldots, V_m \rangle$, where for all $1 \leq i \leq m$, $V_i = MR(t) \cap \{s' : \langle s', U_1 \cap MR(s'), \ldots, U_k \cap MR(s') \rangle \in [\psi_i]^E\}$, is such that $t \in [\varphi]^E$
- $\varphi = \langle loc \varphi' \rangle$, and there is a $t \in S$ such that $s \xrightarrow{\text{loc}} t$, and the summary $t = \langle t, V_1, V_2, \ldots, V_k \rangle$, where $V_i = MR(t) \cap U_i$, is such that $t \in [\varphi]^E$
\( \varphi = \langle \text{ret} \rangle R_t \), and there is a \( t \in S \) such that \( s \xrightarrow{\text{ret}} t \) and \( t \in U_i \\
\varphi = \langle \text{ret} \rangle R_t \), and for all \( t \in S \) such that \( s \xrightarrow{\text{ret}} t \), we have \( t \in U_i \\
\varphi = \mu X.\varphi', \) and \( s \in S \) for all \( S \subseteq S \) satisfying \( \{\varphi'\}^{E[X \mapsto S]} \subseteq S \\
\varphi = \nu X.\varphi', \) and there is some \( S \subseteq S \) such that \( \{\varphi'\}^{E[X \mapsto S]} \subseteq S \) and \( \{\varphi\}^{E[X \mapsto S]} \subseteq S \\
\varphi = \text{Neg}(\varphi), \) \( s \in \text{Neg}(\emptyset) = \emptyset \) \( \text{Neg}(\varphi) = \text{Neg}((\varphi_1 \lor \varphi_2)) = \text{Neg}(\varphi_1) \lor \text{Neg}(\varphi_2) \\
\varphi = \text{Neg}(\varphi_1 \lor \varphi_2) = \text{Neg}(\varphi_1) \lor \text{Neg}(\varphi_2) \\
\varphi = \text{Neg}(\text{call}(v_1, v_2, \ldots, v_k)), \text{ then} \\
\text{Neg}(\varphi) = \text{call}((\text{Neg}(\varphi_1), \text{Neg}(\varphi_2), \ldots, \text{Neg}(\varphi_k))) \\
\varphi = \text{call}(v_1, v_2, \ldots, v_k), \text{ then} \\
\text{Neg}(\varphi) = \text{call}(\text{Neg}(\varphi_1), \text{Neg}(\varphi_2), \ldots, \text{Neg}(\varphi_k)) \\
\text{Neg}(\text{loc}(\varphi)) = \text{loc}(\text{Neg}(\varphi)), \text{ and} \text{Neg}(\text{loc}(\varphi')) = \text{loc}(\text{Neg}(\varphi')) \\
\text{Neg}(\text{ret}(R)) = \text{ret}(R), \text{ and} \text{Neg}(\text{ret}(R')) = \text{ret}(R) \\
\text{Neg}(\mu X.\varphi) = \nu X.\text{Neg}(\varphi), \text{ and} \text{Neg}(\nu X.\varphi) = \mu X.\text{Neg}(\varphi) \\
\begin{align*}
\varphi(X) \colon 2^s \to 2^s & \text{ defined as follows: for all environments } E \text{ and all summary sets } S \subseteq S, \\
\varphi(X)(S) = \{\varphi(X)\}^{E[X \mapsto S]} \subseteq S \text{. It is not hard to verify that this map is monotonic, and that therefore, by the Tarski-Knaster theorem, its least and greatest fixed points exist. The formulas } \mu X.\varphi(X) \text{ and } \nu X.\varphi(X) \text{ respectively evaluate to these two sets. From Tarski-Knaster, we also know that for a VP-\( \mu \) formula } \varphi \text{ with one free variable } X, \text{ the set } \{\mu X.\varphi\} \subseteq \text{ lies in the sequence of summary sets } 0, \varphi(0), \varphi(\varphi(0)), \ldots, \text{ and that } \nu X.\varphi \text{ is a member of the sequence } S, \varphi(S), \varphi(\varphi(S)), \ldots, \\
\text{Fourth, a VP-\( \mu \) formula } \varphi \text{ may also be viewed as a map } \varphi : (U_1, U_2, \ldots, U_k) \to S', \text{ where } S' \text{ is the set of all nodes } s \text{ such that } U_1, U_2, \ldots, U_k \subseteq MR(s) \text{ and the summary } \{s, U_1, U_2, \ldots, U_k\} \text{ satisfies } \varphi. \text{ Naturally, } S' = \emptyset \text{ if no such } s \text{ exists. Now, while a VP-} \mu \text{ formula can demand that a color of the return from the current context is } i, \text{ it cannot assert that the color of a return must not be } i \text{ (i.e., there is no formula of the form, say, } \langle \text{ret} \rangle \neg R_t) \text{. It follows that the output of the above map will stay the same if we grow any of the sets } U_i \text{ of matching returns provided as input. Formally, let } s = \langle s, U_1, \ldots, U_k \rangle \text{ and } s' = \langle s, U'_1, \ldots, U'_k \rangle \text{ be two summaries such that } U_i \subseteq U'_i \text{ for all } i. \text{ Then for every environment } E \text{ and every VP-} \mu \text{ formula } \varphi, s \in \{\varphi\}_E \text{ if } s \in \{\varphi\}_E.
\end{align*}
variables to subsets of \( S \) (the union of the sets of summaries of the disjoint structures). Such an environment is said to be bisimulation-closed if for every variable \( X \), and for every pair of bisimilar summaries \( s \sim t, s \in E(X) \) precisely when \( t \in E(X) \).

**Lemma 1.** If \( E \) is a bisimulation-closed environment and \( \varphi \) is a VP-\( \mu \)-formula, \( \models_E \varphi \) is bisimulation-closed.

**Proof:** The proof is by induction on the structure of the formula \( \varphi \). Consider two bisimulation-closed bisimilar summaries \( s = (s, U_1, \ldots, U_k) \) and \( t = (t, V_1, \ldots, V_k) \), and a bisimulation-closed environment \( E \). We want to show that \( s \models_E \varphi \) if \( t \in \models_E \varphi \).

If \( \varphi \) is a proposition or negated proposition, the claim follows from bisimilarity of nodes \( s \) and \( t \). When \( \varphi \) is a variable, the claim follows from bisimulation closure of \( E \). We consider a few interesting cases.

Suppose \( \varphi = \langle \text{call} \rangle \varphi \) holds. \( s \) satisfies \( \varphi \) precisely when \( s \) has a return-edge to some node \( s' \) in \( U_1 \). Since \( s \) and \( t \) are bisimilar, this can happen precisely when \( t \) has a return edge to a node \( t' \) bisimilar to \( s' \), and from definition of bisimilar summaries, \( t' \) must be in \( V_1 \), and thus \( t \) must satisfy \( \varphi \).

Suppose \( \varphi = \langle \text{call} \rangle \varphi \). Suppose \( s \) satisfies \( \varphi \). Then there is a call-successor \( s' \) of \( s \) such that \( \langle s' \rangle \varphi \) holds, where \( \varphi' \) is \( \varphi' \). Since \( s \) and \( t \) are bisimilar, this can happen precisely when \( t \) has a return edge to a node \( t' \) bisimilar to \( s' \). For each \( i \leq m \), let \( V_i = \{ v \in MR(t') \mid \exists u \in U_i, v \sim u \} \). Verify that the summaries \( \langle s', U_1', \ldots, U_m' \rangle \) and \( \langle t', V_1', \ldots, V_m' \rangle \) are bisimilar. By induction hypothesis, \( \langle t', V_1', \ldots, V_m' \rangle \) satisfies \( \varphi' \).

**Theorem 2.** Two nodes \( s \) and \( t \) satisfy the same set of closed VP-\( \mu \)-formulas precisely when \( S_s \sim S_t \).

### 4. Specifying requirements

In this section, we explore how to use VP-\( \mu \) as a specification language. On one hand, we will see how VP-\( \mu \) and classical temporal logics differ fundamentally in style of expression; on the other, we will express properties not expressible in logics like the \( \mu \)-calculus.

The C program in Fig. 4 will be used to illustrate some of our specifications. Also, because fixpoint formulas are typically hard to read, we will define some syntactic sugar for VP-\( \mu \) using CTL-like temporal operators.

**Reachability.** Let us express in VP-\( \mu \) the reachability property \( \text{Reach} \) that says: “a node \( t \) satisfying proposition \( p \) can be reached from the current node \( s \) before the current context ends.” As a program starts with an empty stack frame, we may omit the restriction about the current context if \( s \) models the initial program state.

Now consider a nontrivial witness \( \pi \) for \( \text{Reach} \) that starts with an edge \( e \). There are two possibilities: (1) a node \( s \) is reached in the new context or a context called transitively from it, and (2) a matching return \( s' \) of \( s \) is reached, and at \( s' \), \( \text{Reach} \) is once again satisfied.

To deal with case (2), we mark a matching return that leads to \( p \) by color 1. Let \( X \) store the set of summaries of form \( \langle s' \rangle \), where \( s' \) satisfies \( \text{Reach} \). Then we want the summary \( \langle s, MR(s) \rangle \) to satisfy \( \langle \text{call} \rangle \varphi \). Where \( \varphi \) states that \( s' \) can reach one of its matching returns of color 1. In case (1), there is no return requirement (we do not need the original call to return), and we simply assert \( \langle \text{call} \rangle \{ \} \).

Before we get to \( \varphi' \), note that the formula \( \langle \text{loc} \rangle X \) captures the case when \( \pi \) starts with a local transition. Combining the two cases and using CTL-style notation, the formula we want is

\[
\text{EF} p = \mu X (p \lor \langle \text{loc} \rangle X \lor \langle \text{call} \rangle X \lor \langle \text{call} \rangle \varphi (X))
\]

Now observe that \( \varphi' \) also expresses reachability, except (1) its target needs to satisfy \( \langle \text{ret} \rangle R_1 \), and (2) this target needs to lie in the same procedural context as \( s' \). In other words, we want to express what we call local reachability of \( \langle \text{ret} \rangle R_1 \). It is easy to verify that

\[
\varphi' = \mu Y (\langle \text{ret} \rangle R_1 \lor \langle \text{loc} \rangle Y \lor \langle \text{call} \rangle Y (Y))
\]

We cannot merely substitute \( p \) for \( \langle \text{ret} \rangle R_1 \) in \( \varphi' \) to express local reachability of \( p \). However, a formula \( EF p \) for this property is
easily obtained by restricting the formula $EF p$:

$$EF^i p = \mu X. (p \lor (\text{loc})X \lor \langle \text{call} \rangle \varphi' (X))$$

For example, consider the structured tree in Fig. 4 that models the unfolding of the C program in the same figure. The transitions in the tree are labeled by line numbers, and some of the nodes are labeled by propositions. Suppose we have a proposition $\text{free}(x)$ that is true immediately after a line where $x$ is freed, $EF^i \text{free}(x)$ holds at the entry point of procedure $\text{foo}(\text{node } s_1)$.

Generalizing, we will allow $p$ to be any $VP-\mu$ formula that keeps $EF^i p$ and $EF^{i+1} p$ closed.

It is easy to verify that the formula $AF p$, which states that “along all paths from the current node, a node satisfying $p$ is reached before the current context terminates,” is given by

$$AF p = \mu X. (p \lor (\text{loc})X \land \langle \text{call} \rangle \varphi'(X))$$

where $\varphi'$ demands that a matching return colored 1 be reached along all local paths:

$$\varphi'' = \mu Y. (p \lor (\text{ret})R_1 \land (\text{loc})Y \land \langle \text{call} \rangle Y \langle Y \rangle).$$

As in the previous case, we can define a corresponding operator $AF^i p$ that asserts local reachability along all paths. For instance, in Fig. 4, $AF^i \text{free}(x)$ does not hold at node $s_1$.

Note that the highlight of this approach to specification is the way we split a program unfolding along procedure boundaries, specify these “pieces” modularly, and plug the summary specifications obtained into their call sites. This “interprocedural” reasoning distinguishes it from logics such as the $\mu$-calculus that would reason only about global runs of the program.

Also, there is a significant difference in the way fixpoints are computed in $VP-\mu$ and the $\mu$-calculus. Consider the fixpoint computation for the $\mu$-calculus formula $\mu X. (p \lor (\text{loc})X)$ that expresses reachability of a node satisfying $p$. The semantics of this formula is given by a set $S_X$ of nodes which is computed iteratively. At the end of the $i$-th step, $S_X$ comprises nodes that have a path with at most $(i-1)$ transitions to a node satisfying $p$. Contrast this with the evaluation of the outer fixpoint in the $VP-\mu$ formula $EF^i p$. Assume that $\varphi'$ (intuitively, the set of “jumps” from calls to returns”) has already been evaluated, and consider the set $S_X$ of summaries for $EF^i p$. At the end of the $i$-th phase, this set contains all $s \models (s')$ such that $s$ has a path consisting of $(i-1)$ call and loc-transitions to a node satisfying $p$. However, because of the sub-formula $\langle \text{call} \rangle \varphi'(X)$, it also includes all $s$ where $s$ reaches $p$ via a path of at most $(i-1)$ local and “jump” transitions. Note how return edges are considered only as part of summaries plugged into the computation.

**Invariance and until**

Now consider the invariance property “on some path from the current node, property $p$ holds everywhere till the end of the current context.” A $VP-\mu$ formula $EG p$ for this is obtained from the identity $EG p = \neg \neg (AF \neg \neg (p))$. The formula $AG p$, which asserts that $p$ holds on each path on each run from the current node, can be written similarly.

Other classic branching-time temporal properties like the existential weak until (written as $E(p_1 W p_2)$) and the existential until ($E(p_1 U p_2)$) are also expressible. The former holds if there is a path $\pi$ from the current node such that $p_1$ holds at every point on $\pi$ till it reaches the end of the current context or a node satisfying $p_2$ (if $\pi$ doesn’t reach either, $p_1$ must hold all along on it). The latter, in addition, requires $p_2$ to hold at some point on $\pi$. The for-all-paths analogs of these properties ($A(p_1 W p_2)$ and $A(p_1 U p_2)$) aren’t hard to write either.

Neither is it difficult to express local or same-context versions of these properties. Consider the maximal subsequence $\pi'$ of a program path $\pi$ from $s$ such that each node of $\pi'$ belongs to the same procedural context as $s$. A $VP-\mu$ formula $EG' p$ for existential local invariance demands that $p$ holds on some such $\pi'$, while $AG' p$ asserts the same for all $\pi'$. Similarly, we can define existential and universal local until properties, and corresponding $VP-\mu$ formulas $E(p_1 U' p_2)$ and $A(p_1 U' p_2)$. For instance, in Fig. 4, $E(\neg \text{free}(g) U' \text{free}(x))$ holds at node $s_1$ (whereas $E(\neg \text{free}(g) U \text{free}(x))$ does not). “Weak” versions of these formulas are also written with ease. For instance, it is easy to verify that we can write generic existential, local, weak until properties as $E(p_1 W' p_2) = \nu X. (p_1 \lor p_2) \land (p_2 \lor (\text{loc})X \lor \langle \text{call} \rangle \varphi'(X))$, where $\varphi'$ asserts local reachability of $\text{ret} R_1$ as before.

**Interprocedural dataflow analysis**

It is well-known that many classic dataflow analysis problems can be reduced to temporal logic model-checking over program abstractions [24, 23]. For example, consider the problem of finding very busy expressions in a program that arises in compiler optimization. An expression $e$ is said to be very busy at a program point $s$ if every path from $s$ must evaluate $e$ before any variable in $e$ is redefined. Let us first assume that all variables are in scope all the time along every path from $s$. Now label every node in the program’s unfolding right after a statement evaluating $e$ by a proposition $use(e)$, and every node reached via redefinition of a variable in $e$ by $\text{mod}(e)$ (see Fig. 4). Because of loops in the flow graph, we would not expect every path from $s$ to eventually satisfy $use(e)$; however, we can demand that each point in such a loop will have a path to a loop exit from where a use of $e$ would be reachable. Then a $VP-\mu$ formula that demands that $e$ is very busy at $s$ is

$$A((EF use(e) \land \neg \text{mod}(e)) W use(e)).$$

Note that this property uses the property of $VP-\mu$ to reason about branching time.

However, complications arise if we are considering interprocedural dataflow and $e$ has local as well as global variables. Suppose in Fig 4, the global variable $a$ and the local variable $b$ are two observables, and we want to check if the expression $e = (a^2 + b^3)$, used

```plaintext
1 int a, *g;
2 void foo ()
3 {
4   int *x, b=1;
5   x = ALLOC(int);
6   g = x;
7   bar ();
8   free (x);
9   b = a*a + b*b;
10  return;
11 }
12 void bar ()
13 {
14   int y;
15   a++;
16   if (y==0)
17     free(g);
18   else
19     foo ();
20   return;
21 }
```

**Figure 4.** A C example
in line 9, is very busy at line 6. We would, as before, track changes before a local use of text should not be of interest of us. On the other hand, the global variable this new instance of current context, the observable the local

Local temporal properties come of use in covering such properties. Let us define two propositions \( \text{mod}_g(e) \) and \( \text{mod}_i(e) \) that are true at points where, respectively, a global or a local variable in \( e \) is modified. The VP-\( \mu \) property we assert at \( s \) is

\[
\nu X.((\text{EF}^1 \text{use}(e)) \land \neg\text{mod}_g(e) \land \neg\text{mod}_i(e)) \lor \text{use}(e) \land (\text{use}(e) \lor ([\text{loc}]X \land [\text{call}]\psi(X, tt)))
\]

where the formula \( \psi \) tracks global variables like a in new contexts:

\[
\psi = \mu Y.((\neg\text{mod}_g(e) \land (\langle [\text{ret}]R_1 \land \langle \text{ret}\rangle R_2) \lor ([\text{call}]Y(Y, tt) \land [\text{loc}]Y(Y))
\]

Note the use of the formula \( \langle \text{ret}\rangle R_2 \) to ensure that \( \text{[ret]}R_1 \) is not vacuously true.

**Pushdown specifications** The domain where VP-\( \mu \) stands out most clearly from previously studied fixpoint calculi is that of pushdown specifications, i.e., specifications involving the program stack. We have already introduced a class of such specifications expressible in VP-\( \mu \)-that of local temporal properties. For instance, the formula \( \text{EF}^1\text{p} \) needs to track the program stack to know whether a reachable node satisfying \( p \) is indeed in the initial calling context. Some such specifications have previously been discussed in context of the temporal logic CARET. On the other hand, it is well-known that the modal \( \mu \)-calculus is a regular specification language (i.e., it is equivalent in expressiveness to a class of finite-state tree automata), and cannot reason about the stack in this way. We have already seen an application of these richer specifications in program analysis. In the rest of this section, we will see more of them.

**Nested formulas and stack inspection** Interestingly, we can express certain properties of the stack just by nesting VP-\( \mu \) formulas for (non-local) reachability and invariance. To understand why, recall that VP-\( \mu \) formulas for reachability and invariance only reason about nodes appearing before the end of the context where they were asserted. Now let us try to express a stack inspection property such as “if procedure \( \text{foo} \) is called, procedure \( \text{bar} \) must not be on the call stack.” Specifications like this have previously been used in research on software security [19, 15], and are not expressible by regular specifications like the \( \mu \)-calculus. While the temporal logic CARET can express such properties, it requires a past-time operator called \( \text{call}\) to do so. To express this property in VP-\( \mu \), we define propositions \( \text{cfoo} \) and \( \text{cabar} \) that respectively hold at every call site for \( \text{foo} \) and \( \text{bar} \). Now, assuming control starts in \( \text{foo} \), consider the formula

\[\varphi = \text{EF}\langle\text{call}\rangle(\text{EF}\langle\text{cfoo}\rangle)\]

This formula demands a program path where, first, \( \text{bar} \) is called (there is no return requirement), and then, before that context is popped off the stack, a call site for \( \text{foo} \) is reached. It follows that the property we are seeking is \( \text{Neg}(\varphi) \).

Other stack inspection properties expressible in VP-\( \mu \) include “when procedure \( \text{foo} \) is called, all procedures on the stack must have the necessary privilege.” Combining reasoning about the program stack with reasoning about the global evolution of the program, VP-\( \mu \) can even specify dynamic security constraints where privileges of procedures change dynamically depending on the privileges used so far.

**Stack overflow** One of the hazards of using recursive calls in a C-like language is that stack overflow, caused by unbounded recursion, is a serious security vulnerability. VP-\( \mu \) can specify requirements that safeguard against such errors. Once again, nested modalities come handy. Suppose we assert \( AG((\langle call\rangle f f)(\}) \) for every context reached through \( k \) calls in succession without intervening returns (this can be kept track of using a \( k \)-length chain of \( \langle call\rangle \) modalities). This will disallow further calls, bounding the stack to height \( k \).

Other specifications for stack boundedness include: “every call in every program execution eventually returns.” This property requires the program stack to be empty infinitely often. Though this requirement does not say how large the stack may get—even if a call returns, it may still overflow the stack at some point. Further, in certain cases, a call may not return because of cycles introduced by abstraction. However, it does rule out infinite recursive loops in many cases; for instance, the program in Fig. 4 will fail this property because of a real recursive cycle. We capture it by asserting \( AG\text{Termin} \) at the initial program point, where

\[\text{Termin} = [\text{call}](\text{AF}^1(\langle \text{ret}\rangle R_1))(tt)\]

**Preconditions and postconditions** For a program state \( s \), let us consider the set \( \text{jump}(s) \) of nodes to which a call from \( s \) may return. Then the requirement: “property \( p \) holds at some node in \( \text{jump}(s) \)” is captured by the VP-\( \mu \) formula \( \langle \text{jump}\rangle p = \langle \text{call}\rangle(\text{EF}^1(\langle \text{ret}\rangle R_1))(p) \). The dual formula \( \langle \text{jump}\rangle p \), which requires \( p \) to hold at all such jump targets, is also easily constructed. An immediate application of this is to encode the partial and total correctness requirements popular in formalisms like Hoare logic and JML [9]. A partial correctness requirement for a procedure \( A \) asserts that if precondition \( \text{Pre} \) is satisfied when \( A \) is called, then if \( A \) terminates, postcondition \( \text{Post} \) holds upon return. Total correctness, additionally, requires \( A \) to terminate. These requirements cannot be expressed using regular specifications. In VP-\( \mu \), let us say that at every call site to procedure \( A \), proposition \( cA \) holds. Then a formula for partial correctness, asserted at the initial program state, is

\[AG((\text{Pre} \land cA) \Rightarrow \langle \text{jump}\rangle \text{Post})\]

Total correctness is expressed as

\[AG((\text{Pre} \land cA) \Rightarrow \langle \text{Termin} \land \langle \text{jump}\rangle \text{Post})\]

**Access control** The ability of VP-\( \mu \) to handle local and global variables simultaneously is useful in other domains, e.g., access control. Consider a procedure \( A \) that can be called with a high or low privilege, and suppose we have a rule that \( A \) can access a database (proposition \( \text{access} \) is true when it does) only if it is called with a high privilege (\( \text{priv} \) holds when it is). It is tempting to write a property \( \varphi = \text{~priv} \Rightarrow AG\langle \text{access}\rangle \) to express this requirement. However, a context where \( A \) has low privilege may lead to another where \( A \) has high privilege via a recursive invocation, and \( \varphi \) will not let \( A \) access the database even in this new context. The formula we are looking for is really \( \varphi' = \text{~priv} \Rightarrow AG\langle \text{~access}\rangle \), asserted at every call site for \( A \).

**Multiple return conditions** As we shall see in Section 6.2, the theoretical expressiveness of VP-\( \mu \) depends on the fact that we can pass multiple return conditions as “parameters” to VP-\( \mu \) call.
formulas. We can also use these parameters to remember events that happen within the scope of a call and take actions accordingly on return. To see how, we go back to Figure 4; now we interested in the properties of the pointer variables x and g. Suppose control starts at foo and moves on to bar; also, let us ignore the recursion in line 19 and assume the call to bar in line 7 returns. Before this call, x and g point to the same memory location. Now consider two scenarios once this call returns: (1) the global g was freed in the new context before the return, so that x now points to a freed location, (2) g was not freed, so that x still points to allocated memory. Suppose our requirements for the next program point in the two cases are: (1) x must not be freed in foo, (2) x should be freed to avoid memory leak. We express these requirements by asserting the VP-μ formula \( \varphi \) at the program point calling bar:

\[
\varphi = (\text{call}) \psi' \{(\text{loc} \not\rightarrow \text{free}(x)), (\text{loc} \not\rightarrow \text{free}(x))\},
\]

where \( \psi' \) is a fixed-point property that states that: each path in the new context must (1) see free(g) at some point and then reach (\text{ret})R₁, or (2) satisfy \( \neg \text{free}(g) \) until (\text{ret})R₂ holds. We omit the details for want of space.

5. Model-checking

In this section, we introduce the problem of model-checking VP-μ over unfoldings of recursive state machines. Our primary result is an iterative, symbolic decision procedure to solve this problem. Appealingly, this algorithm follows directly from the operational semantics of VP-μ and has the same complexity as the best algorithms for model-checking μ-calculus over similar abstractions. We also show a matching lower bound.

5.1 Recursive state machines

Recursive state machines (RSMs) are program abstractions that model interprocedural control flow in recursive programs [3]. While expressively equivalent to pushdown systems, RSMs are more visual and tightly coupled to program control flow. For this reason, we will use them as our system model.

Syntax. A recursive state machine (RSM) M over a set of propositions \( AP \) is a tuple \( (M_1, M_2, \ldots, M_m, \text{start}) \), where each \( M_i \) is a procedure of the form \( (L_i, B_i, Y_i, E_i, \delta_i, \kappa_i) \). The meaning of the components of \( M_i \) is summarized in the following:

- \( L_i \) is a finite set of control locations, and \( B_i \) is a finite set of boxes.
- \( Y_i : B_i \rightarrow \{1, 2, \ldots, m\} \) is a map that assigns a procedure to every box.
- \( E_i \subseteq L_i \times B_i \) is a non-empty set of entry locations, and \( E_i \subseteq L_i \) is a non-empty set of exit locations.
- \( \text{Calls}_i = \{(b, \text{en}) : b \in B_i, \text{en} \in E_n(b_i)\} \) denote the set of calls in \( M_i \), and let \( \text{Retns}_i = \{(b, \text{ex}) : b \in B_i, \text{ex} \in E_{\text{ex}}(b_i)\} \) denote the set of returns in \( M_i \). Then \( \delta_i \subseteq (L_i \cup \text{Calls}_i) \times (L_i \cup \text{Calls}_i) \) defines the set of RSM edges.
- \( \kappa_i \) is a labeling function \( \kappa_i : (L_i \cup \text{Calls}_i \cup \text{Retns}_i) \rightarrow 2^{AP} \) that associates a set of propositions to each control location, call and return.

A control location \( \text{start} \in \bigcup_i L_i \) in one of the components is chosen as the initial location. We assume that for every distinct \( i \) and \( j, L_i, \text{Calls}_i, \text{Retns}_i, N_i, \text{Calls}_j, \text{Retns}_j \) are pairwise disjoint. We refer to arbitrary calls, returns and control locations in \( M \) as vertices. The set all vertices is given by \( V = \bigcup_i (L_i \cup \text{Calls}_i \cup \text{Retns}_i) \), and the set of vertices in procedure \( j \) is denoted by \( V_j \). We also write \( B = \bigcup_i B_i \) to denote the collection of all boxes in \( M \). Finally, the extensions of all functions \( \delta_i, \kappa_i \) and \( Y_i \) are denoted respectively by \( \delta : V \rightarrow V, \kappa : V \rightarrow 2^{AP} \), and \( Y : B \rightarrow \{1, 2, \ldots, m\} \).

Fig. 5 depicts an RSM extracted from the C program in Fig. 4. Here we are interested in the behavior of the pointer variable g, and variables and statements not relevant to this behavior are abstracted out. We use two propositions \( g_0 \) and \( g_1 \) that are true respectively when g points to free and allocated memory. The procedures and vertices in this RSM correspond to procedures and control states in Fig. 4; transitions correspond to lines of C code and are labeled by line numbers.

Each procedure has two entry and exit points corresponding to the two possible abstract values of g. Pointer assignments and calls to free and alloca changes the values of these propositions in the natural way. Note in particular that we cannot tell without a global side-effect analysis whether \( x \) and \( g \) point to the same location before line 8. We model this uncertainty using nondeterminism.

Semantics. The semantics of an RSM \( M \) are defined by an infinite graph \( G(M) = (C, c_0, E_C, \lambda_C, \eta_C) \), known as its configuration graph. Here, \( C \) is a set of configurations, \( c_0 \) is the initial configuration, \( E_C \subseteq C \times C \) is a transition relation, and functions \( \lambda_C : C \rightarrow 2^{AP} \) and \( \eta_C : E_C \rightarrow \{\text{call, ret, loc}\} \) respectively label configurations and transitions. Stealing notation for structured trees, we write \( c \xrightarrow{u} c' \) if \( (c, c') \in E_C \) and \( \eta_C((c, c')) = a \).

The set \( C \) of configurations in \( G(M) \) comprise all elements \((\gamma, u) \in B^* \times V \) such that either:

- \( \gamma = e \) and \( u \in V \), or
- \( \gamma = b_1 \ldots b_n \) (with \( n \geq 1 \)) and (1) \( u \in V_Y(b_n) \), and (2) for all \( i \in \{1, \ldots, n-1\} \), \( b_{i+1} \in V_Y(b_i) \).

The initial configuration is \( c_0 = (e, \text{start}) \). The configuration-labeling function \( \lambda_C \) is defined as: \( \lambda_C((\gamma, u)) = \kappa(u) \) for all \((\gamma, u) \in B^* \times V \). Now we can define the transition relation \( E_C \) and the transition-labeling function \( \eta_C \) in \( G \). For \( c = (\gamma, u) \), \( c' = (\gamma', u') \) and \( a \in \{\text{call, ret, loc}\} \), we have a transition \( c \xrightarrow{a} c' \) if and only if one of the following holds:

- Local move: \( u \in (L_i \cup \text{Retns}_i) \setminus E_{\text{ex}}, (u, u') \in \delta_i, \gamma' = \gamma, \) and \( a = \text{loc}; \)
- Procedure call: \( u = (b, \text{en}) \in \text{Calls}_i, u' = \text{en}, \gamma' = \gamma, b, \) and \( a = \text{call}; \)
- Return from a call: \( u \in E_{\text{ex}}, \gamma = \gamma', b, u' = (b, u), \) and \( a = \text{ret}. \)

5.1.1 Configuration trees

We will evaluate VP-μ formulas on configuration trees of RSMs, which are unfoldings of configuration graphs of RSMs. Consider an RSM \( M \) with configuration graph \( G(M) = (C, c_0, E_C, \lambda_C, \eta_C) \). The configuration tree of \( M \) is a structured tree \( \mathcal{C} \mathcal{F}(M) = (S, s_0, E, \lambda, \eta) \), whose set of nodes \( S \subseteq C^* \) and set of transitions \( E \subseteq S \times S \) are the least sets constructed by the following rules:

1. \( c_0 \in S. \)
The node-labeling function \( \eta \) in Conf(\( M \)).
The node-labeling function \( \lambda \) is defined by: for each node \( s = s.c \),
\( \lambda(s) = \lambda_C(s) \). The initial node is \( s_0 = c_0 \). Finally, we define a map \( \text{Curr} : S \rightarrow C \) that gives us the current configuration for any node in Conf(\( M \)) as follows: for all \( s \in C^* \) and all \( c \in C \),
\( s.c.c' \in S \) and \( s.c \rightarrow s.c' \).

Summaries in Conf(\( M \)) are now defined as in Section 2.1.
We identify the summary \( s_0 = (s_0) \) as the initial summary in Conf(\( M \)).
We say that the RSM \( M \) satisfies a closed VP-\( \mu \) formula \( \phi \) if and only if \( s_0 \in \phi \).

Note that each node in Conf(\( M \)) captures, along with the current configuration, the history of an execution till this point.
However, it is easy to see that if \( \text{Curr}(s) = \text{Curr}(s') \) for two nodes \( s \) and \( s' \) in Conf(\( M \)), then \( s \) and \( s' \) are bisimilar. Then by Theorem 2, the difference between the histories of \( s \) and \( s' \) is irrelevant so far as VP-\( \mu \) formulas are concerned.

5.2 Model-checking VP-\( \mu \) over RSMs
For a closed VP-\( \mu \) formula \( \phi \) and an RSM \( M \), the model-checking problem of \( \phi \) over \( M \) is to determine if \( M \) satisfies \( \phi \).
Recall that configurations of \( M \) are of the form \( (\gamma, u) \), where \( \gamma \) is a stack of boxes and \( u \) is a vertex in \( M \). Clearly, the set \( M.R(s) \) for the current node \( s \) and the set \( \phi_u \) of ret-subformulas that hold at the current summary depend on the current stack \( \gamma \). However, we observe that both these sets refer only to the box to which control returns from the current context, and not to boxes further down the box stack. In other words, so far as satisfaction of VP-\( \mu \) formulas go, we are only interested in the top of \( \gamma \).
To formalize this intuition, let us define a map \( \text{Erase} : (\gamma, u) \rightarrow u \) that erases the stack of a given configuration of \( M \). We can extend this map to sets of nodes in the usual way. Now consider two \( k \)-colored summaries \( s = (s, U_1, U_2, \ldots, U_k) \) and \( s' = (s', U_1, U_2, \ldots, U_k) \), where \( \text{Curr}(s) = (s, b, u) \) and \( \text{Curr}(s') = (s', b, u) \). We call \( s \) and \( s' \) top-equivalent if and only if for all \( i \), \( \text{Erase}(U_i) = \text{Erase}(U'_i) \). Then we have:

**Lemma 2.** Let \( s_1 \) and \( s_2 \) be two top-equivalent \( k \)-colored summaries in the configuration tree of an RSM \( M \). Then for any closed VP-\( \mu \) formula \( \phi \), \( s_1 \) satisfies \( \phi \) iff \( s_2 \) satisfies \( \phi \).

It turns out that we can restrict our attention to bounded-size summaries that only keep track of the top of the box stack while doing VP-\( \mu \) model-checking. We call these summaries stackless. In order to define stackless summaries formally, we will need to define reachability between nodes in an RSM. Consider any vertex \( u \) in an RSM \( M \). A vertex \( v \) is said to be \( empty-box reachable \) from \( u \) if \( u \) is a path in Conf(\( M \)) from \( (e, u) \) to \( (e, v) \). It is well-known that for any \( u \), the set \( \text{Reach}_u(\gamma) \) of vertices empty-box reachable from \( u \) can be computed in time polynomial in the size of \( M \) [3].
We also need to define the set of possible returns from an RSM vertex \( u \). Suppose \( u \in V \) is in procedure \( M \). The set \( \text{Ret}_k(u) \) of possible returns from \( u \) to a box \( b \) with \( Y(b) = l \) consists of all \( (b, ex) \) such that \( ex \in \text{Reach}_u(\gamma) \) and \( \epsilon \). Clearly, for any \( u \) and \( b \), the set \( \text{Ret}_k(u) \) can be computed in time polynomial in \( M \). Also, we will use the notation \( \text{Ret}(u) = \cup_k \text{Ret}_k(u) \).

Now, let \( n \) be the arity of the formula \( \phi \). A stackless summary is a tuple \( (u, \text{Ret}_1, \text{Ret}_2, \ldots, \text{Ret}_k) \), where \( 0 \leq k \leq n \), and for some \( b \), \( \text{Ret}_k \subseteq \text{Ret}_k(u) \) for all \( j \). The set of all stackless summaries in \( M \) is denoted by SLS.

Let \( \mathcal{E}_{SL} : \phi \rightarrow 2^{SLS} \) be an environment mapping free variables in \( \phi \) to sets of stackless summaries, and let \( \bot \) denote the empty environment. We define a function \( \text{Eval}(\phi, \mathcal{E}_{SL}) \) that assigns a set of stackless summaries to a VP-\( \mu \) formula \( \phi \):

- If \( \phi = \varphi \), for \( \varphi \in AP \), then \( \text{Eval}(\varphi, \mathcal{E}_{SL}) \) consists of all \( (v, \text{Ret}_1, \text{Ret}_2, \ldots, \text{Ret}_k) \) such that \( p \in k(v) \) and \( k \leq n \).
- If \( \phi = \neg \varphi \), for \( \varphi \in AP \), then \( \text{Eval}(\varphi, \mathcal{E}_{SL}) \) consists of all \( (v, \text{Ret}_1, \text{Ret}_2, \ldots, \text{Ret}_k) \) such that \( p \notin k(v) \) and \( k \leq n \).
- If \( \phi = \chi \), for \( \chi \in \text{Var} \), then \( \text{Eval}(\chi, \mathcal{E}_{SL}) = \mathcal{E}_{SL}(X) \).
- If \( \phi = \varphi_1 \land \varphi_2 \), then \( \text{Eval}(\varphi_1, \mathcal{E}_{SL}) \cap \text{Eval}(\varphi_2, \mathcal{E}_{SL}) \).
- If \( \phi = \chi \), then \( \text{Eval}(\chi, \mathcal{E}_{SL}) = \text{Eval}(\chi, \mathcal{E}_{SL}) \).

Figure 6. Fixpoint computation for VP-\( \mu \).

The following theorem is easily proved:

**Theorem 3.** For an RSM \( M \) and a closed VP-\( \mu \) formula \( \phi \), Conf(\( M \)) satisfies \( \phi \) if and only if \( s_0 \in \text{Eval}(\phi, \bot) \). Furthermore, \( \text{Eval}(\phi, \bot) \) is computable.
Note that our decision procedure is symbolic in nature, and that one could represent sets of summaries using BDD-like data structures. Also, it directly implements the operational semantics of VP-µ formulas over stackless summaries. In this regard VP-µ resembles the modal µ-calculus, whose formulas encode fixpoint computations over sets; to model-check µ-calculus formulas, we merely need to perform these computations. Unsurprisingly, our procedure is very similar to classical symbolic model-checking for the µ-calculus.

5.3 Complexity

In an RSM M, let θ be an upper bound on the number of possible returns for a vertex, and let N_V be the total number of vertices. Let n be the arity of the formula in question. Then the total number of stackless summaries in M that we need to consider is bounded by \( N = nN_V 2^n \). Let us now assume that union or intersection of two sets of summaries, as well as membership queries on such sets, take linear time. It is easy to see that the time needed to evaluate a non-fixpoint formula \( \varphi \) of arity \( k \) is bounded by \( O(N^3\theta(\varphi)) \) (the most expensive modality is \( \langle \text{call} \rangle \varphi \{\psi_1, \ldots, \psi_n\} \), where we have to match an “inner” summary satisfying \( \varphi \) as well as \( n \) “outer” summaries satisfying the \( \psi_i \)-s. For a fixpoint formula \( \varphi \) with one fixpoint variable, we may need \( N \) such evaluations, so that the total time required to evaluate \( \text{Eval}(\varphi, \lambda) \) is \( O(N^3\theta(\varphi)) \). For a formula \( \varphi \) of alternation depth \( \delta \), this evaluation takes time \( O(N^{3\delta}\theta^d(\varphi)) \).

It is known that model-checking alternating reachability specifications on an RSM M is EXPTIME-hard [26]. Following constructions similar to those in Section 4, we can generate a VP-µ formula \( \varphi \) from a µ-calculus formula \( f \) expressing such a property such that (1) the size of \( \varphi \) is linear in the size of \( f \), and (2) \( M \) satisfies \( \varphi \) if and only if \( M \) satisfies \( f \). It follows that model-checking a closed VP-µ formula \( \varphi \) on an RSM M is EXPTIME-hard.

Combining all of the above, we have:

**Theorem 4.** Model-checking a VP-µ formula \( \varphi \) on an RSM M is EXPTIME-complete.

6. Expressiveness

Now we present a few results concerning the expressiveness of VP-µ.

6.1 VP-µ and the temporal logic CARET

We will now establish that the temporal logic CARET is contained in VP-µ.

A visibly pushdown automaton \( A \) is a pushdown automaton with an added restriction: its input alphabet \( \Sigma \) is partitioned into a call alphabet \( \Sigma_c \), a local alphabet \( \Sigma_l \), and a return alphabet \( \Sigma_r \). On reading a call or return symbol, \( A \) must respectively push and pop precisely one symbol on/from the stack; on a local input symbol, \( A \) must change state without modifying the stack. A run of such an automaton on an input word \( w \) is simply a sequence of moves it makes while reading \( w \).

From the results implicit in [2, 4], it follows that for any CARET formula \( \varphi \) over the set of propositions \( P \), there exists \( A_\varphi \) over infinite words with a Büchi acceptance condition, over the alphabet \( \Sigma_{\infty} = \{(\text{loc}, v) \mid v \in 2^P\} \), \( \Sigma_c = \{(\text{call}, v) \mid v \in 2^P\} \) and \( \Sigma_l = \{(\text{ret}, v) \mid v \in 2^P\} \) that accepts precisely the models that satisfy \( \varphi \). We can show that:

**Theorem 5.** For any nondeterministic Büchi visibly pushdown automaton \( A \) over the alphabets described above, there is a VP-µ formula \( \varphi_A \) that holds in a structured tree \( t \) if and only if there is some path in the tree that is accepted by \( A \).

It follows as a corollary that CARET model checking is reducible to VP-µ model-checking: we can take the negation of any CARET formula \( \varphi \), find the corresponding VP-µ formula, and check whether it holds in a program model. The model satisfies this VP-µ formula if and only if it is the case that all runs of the model satisfy the CARET formula.

We will now sketch the idea behind the proof of Theorem 5. For brevity, however, we will consider acceptance by final state rather than Büchi acceptance. More precisely, we will have a special final state \( q_f \), and a run will be considered accepting if and only if it reaches \( q_f \) somewhere along the run. This simplification will take out many of the details of our translation while retaining its basic flavor.

Now, \( \varphi_A \) will be a 1-ary formula, and will have variables \( X_q \) and \( S_q,q',b \), where \( q,q' \) in \( A \) and \( b \in B \) (the stack alphabet). Intuitively, a 1-ary summary \( (s, U) \) will be in \( X_q \) if there is a run starting from \( s \) and state \( q \) along some path that reaches \( q_f \) before it meets any matching return of \( s \). The summary \( (s, U) \) will be in \( S_q,q',b \) if \( s \) is not at the top-level, there is a path that ends with an unmatched return edge, and the automaton has a run from \( q \) to \( q' \) along this path such that at the unmatched return \( b \) is popped from the stack to reach \( q' \).

We will write the VP-µ formula \( \varphi_A \) using a set of equations rather than in the standard form. Translation from this equational form to the standard form proceeds as in the modal µ-calculus [16]; we leave out the details. Let us denote internal transitions as \( (q,v,q') \), push transitions as \( (q,v,q',b) \) (if in state \( q \) and reading \( \langle \text{call}, v \rangle \), push \( b \) onto stack and go to \( q' \) ), and pop transitions as \( (q,v,b,q') \) (if in state \( q \) with \( b \) on top-of-stack and reading \( \langle \text{ret}, v \rangle \), pop \( b \) and go to \( q' \) ). Let the set of transitions of \( A \) be \( \Delta \). Also, for each valuation \( v \in 2^P \), let \( \psi_v = \lambda_p \in p \wedge \lambda_\neg p \neg p \).

Then, the formula \( \varphi_A \) will be the formula corresponding to \( X_{q_0} \) when taking the least fixpoint of the following equations:

\[
X_q = \bigvee_{(q,v,q') \in \Delta} (\psi_v \wedge \langle \text{loc} \rangle X_{q'}) \\
= \bigvee_{(q,v,q',b) \in \Delta} (\psi_v \wedge \langle \text{call} \rangle X_{q'}(ff)) \\
= \bigvee_{(q,v,q',b) \in \Delta} (\psi_v \wedge \langle \text{call} \rangle S_{q_1,q',b}(X_{q'})) \\
S_{q,q',b} = \bigvee_{(q,v,b,q') \in \Delta} (\psi_v \wedge \langle \text{ret} \rangle R_1) \\
= \bigvee_{(q,v,q') \in \Delta} (\psi_v \wedge \langle \text{loc} \rangle S_{q',q,b}) \\
= \bigvee_{(q,v,q',b) \in \Delta} (\psi_v \wedge \langle \text{call} \rangle S_{q_1,q_2,b'}(S_{q_2,q,b})).
\]
Figure 7. (a) An arity hierarchy (b) Tree encoding for undecidability.

Consider a pair of $k$-colored summaries $s = (s, U_1, \ldots, U_k)$ and $t = (t, V_1, \ldots, V_k)$ such that each path in the trees rooted at $s$ and $t$ comprises a chain of $\text{loc}$-edges followed by one $\text{ret}$-edge leading to a leaf. Let $S$ and $T$ respectively be the sets of non-leaf nodes in these trees. We say that $s$ and $t$ are quasi-bisimilar if there is a relation $\preceq \subseteq S \times T$ such that $s \preceq t$ and

1. For all $s' \preceq t'$, we have $\lambda(s') = \lambda(t')$
2. If $s' \preceq t'$, then for every $s''$ such that $s' \xrightarrow{\text{loc}} s''$, there is a $t''$ such that $t' \xrightarrow{\text{loc}} t''$ and $s'' \preceq t''$. Also, for every $t''$ such that $t' \xrightarrow{\text{loc}} t''$, there is an $s''$ such that $s' \xrightarrow{\text{loc}} s''$ and $s'' \preceq t''$.
3. If $s' \preceq t'$, then for every $s''$ such that $s' \xrightarrow{\text{ret}} s''$, there is a $t''$ such that $t' \xrightarrow{\text{ret}} t''$, and for every $t''$ such that $t' \xrightarrow{\text{ret}} t''$, there is an $s''$ such that $s' \xrightarrow{\text{ret}} s''$. Further, if $s'' \in U$, then $t'' \in V$, for all $i$ (note that this is not an “iff” condition).

Now, we can show inductively that if $s$ and $t$ are quasi-bisimilar, then for every variable-free $\text{VP-\mu}$ formula $\phi$, if $s$ satisfies $\phi$, then $t$ satisfies $\phi$ as well (note that the converse is not true; for instance, $t$ may satisfy $\text{ret}_1R_i$ even when $s$ does not). We skip the proof.

Let us now come back to Theorem 6. Consider the two non-bisimilar structured trees $S$ and $T$ in Fig. 7a with initial nodes $s_0$ and $t_0$. It is easy to see that the 2-ary $\text{VP-\mu}$ formula $\varphi = (\text{call})((\text{loc})(\text{ret})R_1 \land (\text{ret})R_2)(p, \neg p)$ distinguishes $s_0$ and $t_0$. Let us now see if there is a closed, 1-ary formula $\phi$ that can distinguish between $S$ and $T$. First, if $\phi$ is a disjunction or conjunction, we can get a smaller witness for this distinction. Further, because trees $S$ and $T$ are of fixed depth, we need only consider fixpoint-free formulas. The interesting case is that of formulas of the form $\phi = (\text{call})\varphi'([\psi])$. Assume this formula is satisfied by $\langle s_0 \rangle$; then there is a bisimulation-closed summary of the form $s = (s_1, U)$ that satisfies $\varphi'$. For each such $s$, we find a $t = (t_1, V)$. Note that $s$ can assume only four values; these are $\langle s_1, s_0, s_3, s_5 \rangle$, $\langle s_1, s_6, s_8, s_7 \rangle$, $\langle s_1, s_9, s_4, s_3, s_8, s_7 \rangle$, and $\langle s_1, \emptyset \rangle$. The corresponding values of $t$ are $\langle t_1, t_2, t_3 \rangle$, $\langle t_1, t_4, t_5 \rangle$, $\langle t_1, t_4, t_5, t_6, t_7 \rangle$, and $\langle t_1, \emptyset \rangle$ respectively. Note that for any value $s$ takes, the corresponding $t$ is quasi-bisimilar to it, which means that $t$ satisfies $\varphi'$.

Further, for each $v \in V$ there is a bisimilar node $u \in U$. It follows that if all $u \in U$ satisfy $\psi$, then so do all $v \in V$. Then $(t_0)$ satisfies $\varphi'$. Similarly one can show that $(t_0)$ satisfies $\varphi$ only if $(s_0)$ satisfies $\varphi$.

To extend the proof to arbitrary $k$, we consider a structure $S'$ where, like in $S$, the root has one $\text{call}$-child $s_1$—except $s_1$ now has a large number $N$ of $\text{loc}$-children $s'$. From each $s'$, we have $(k + 1)$ $\text{ret}$-edges leading to “leaves” $s''$, each of which is labeled with exactly one proposition from the set $AP = \{p_1, p_2, \ldots, p_{k+1}\}$. For $(N - 1)$ values of $s'$, the leaves of the trees rooted at $s'$ are labeled such that only $k$ of them have distinct labels. But there is a particular $s'$ (call it $s'_d$) for which these leaves get distinct labels $p_1, \ldots, p_{k+1}$.

Now take a structure $T'$ that is obtained by removing the subtree rooted at node $s'_d$ from $S'$. Following the methods for the case $k = 1$, we can show that $S'$ and $T'$ may be distinguished by a $(k + 1)$-ary formula, but by no $k$-ary formula. We skip the details.

6.3 Satisfiability

The logic $\text{VP-\mu}$ can express several surprising complex properties by exploiting the branching nature of the models. Consider the tree fragment depicted in Figure 7b.

We have a set of propositions $P$ from which the $a_1$'s and $b_i$'s are drawn; we also have extra propositions $\$ and *. It turns out that we can write a $\text{VP-\mu}$ formula $\phi$ whose models consist only of trees that are bisimilar to the tree given in the figure, with the restriction that the word $a_1 \ldots a_m = b_m \ldots b_1$. This is surprising as even a visibly pushdown word automaton cannot accept precisely the words encoded along the branch $a_1 \ldots a_m b_1 \ldots b_m$ (it could if the second portion encoding the $b$'s are returns, but not when they are calls). The $\text{VP-\mu}$ formula can accept this by expressing mainly the following requirements: (a) there are only two $\text{ret}$-edges after $b_1$, (b) consider any position $a_i$ and let $b_j$ be a position such that the last return in the string of returns after $b_j$ matches the call at $a_i$; then we require that the string of returns following $b_{j+1}$ ends up matching the call at $a_i$ followed by exactly two more returns.

These requirements ensure that the string of returns below the $b_j$ symbols grow according to the sequence $2, 4, 6, \ldots$. Now, the $\text{VP-\mu}$ formula also demands that for any position $a_i$, if the returns below $b_j$ end by matching the call at $a_i$, then $b_j = a_i$. This ensures that $m = n$ and $a_1 \ldots a_n = b_m \ldots b_1$. The formula has to be written carefully and is complex, and we omit its exact description.

The above structure turns out to be very powerful. Notice that no finite recursive state machine will have its unfolding as the above tree (which is why model-checking is decidable!). However, we can show, using the above construction, that trees of the kind above concatenated to each other can encode the computations of a Turing machine, and hence the satisfiability problem for $\text{VP-\mu}$ is undecidable (we omit details of the proof).

Theorem 7. Given a $\text{VP-\mu}$ formula, the problem of checking whether there is some structured tree that satisfies it is undecidable.

7. Conclusions

We have defined a powerful fixpoint logic over execution trees of structured programs that captures pushdown specifications taking into account both local and global program flows. It can express several useful and interesting properties, both in program verification as well as dataflow analysis, and yet admits tractable model-checking. The logic unifies and generalizes many existing logics, leading to a new class of decidable properties of programs that, we believe, will be a basis for future software model checking tools. In fact, the decidability of well known program logics ($\mu$-calculus, temporal logics LTL and CTL, Cares, etc.) can be understood...
by interpreting them in the monadic second-order logic over trees, which is decidable (this can also be used to show that the satisfiability problem for these logics is decidable). However, there is no such embedding of the logic VP-µ into the MSO theory of trees (the fact that its satisfiability problem is undecidable argues that there cannot be such an effective embedding).

This paper can lead to work in several interesting directions. One natural question that arises is how robust the logic VP-µ is. The modal µ-calculus has been shown to be the canonical bisimulation closed modal logic, as it captures all bisimulation-closed properties definable using monadic second order logic [18], and is exactly equivalent to trees accepted by a restriction of alternating parity tree automata that by design can accept only bisimulation-closed sets of trees [14]. While we have not discussed this aspect in this paper, the logic VP-µ was carefully designed to have such a canonical expressive power. We have established in a subsequent paper [1] that VP-µ is expressively equivalent to alternating visibly pushdown parity tree automata, a natural variant of tree automata on structured trees. As a corollary, it follows that for any closed modal µ-calculus formula \( f \), we can construct a closed VP-µ formula \( \varphi_f \) such that a node in a structured tree satisfies \( f \) if and only if it satisfies \( \varphi_f \).

Finally, the logic VP-µ expresses properties using forward modalities. As argued in [23], several dataflow analysis problems also require backward modalities; extending VP-µ to backward modalities will result in expressing several other dataflow problems.

References
