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 μ-calculus* (VP-μ), as an extension of the modal μ-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 μ-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-μ 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-μ: 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 μ-calculus. This makes VP-μ the most expressive known program logic for which algorithmic software model checking is feasible. In fact, the decidability of most known program logics (μ-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-μ, 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, μ-calculus, infinite-state, model-checking, games, pushdown systems

1. Introduction

The μ-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 μ-calculus formulas [8, 21].

In this paper, we focus on the role of μ-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 μ-calculus properties of such models is possible [26, 10], classical μ-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 5" 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 μ-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

* This research was partially supported by ARO URI award DAAD19-01-1-0473 and NSF award CCR-0306382.

POPL'06 January 11–13, 2006, Charleston, South Carolina, USA.

busy (that is, e is guaranteed to be used before any of its variables get modified), cannot be specified using a μ-calculus formula even though interprocedural dataflow analysis can compute this information. The goal of this paper is to identify a fixpoint calculus that can express such pushdown requirements and yet has a decidable model checking problem with respect to pushdown models.

Our search for such a calculus was guided by the recently proposed framework of *visibly pushdown languages* for linear-time properties [4]. In this variation of pushdown automata over words, the input symbol determines when the pushdown automaton can push or pop, and thus the stack depth at every position. The resulting class of languages is closed under union, intersection, and complementation, and problems such as inclusion that are undecidable for context-free languages are decidable for visibly pushdown automata. This implies that checking pushdown properties of pushdown models is feasible as long as the calls and returns are made *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 interprocedural 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 temporal 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-regular properties, without causing undecidability.

In order to develop a *visibly pushdown branching-time logic*, we consider *structured trees* as models. In a structured tree, nodes are labeled with atomic propositions as in Kripke models, and edges are tagged as *call*, *return*, or *local*. To associate a structured tree with a program (or its abstraction), we must choose the set of observable 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 abstractions of programs.

We define the *visibly pushdown μ-calculus* (VP-μ) over structured 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 fixpoint operators 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: different 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 corresponds to concatenation of call trees: the formula $\langle call\rangle\varphi\{\psi\}$ holds at a summary $\langle s, U\rangle$ if the state s has a call-edge to a state t, and there exists a summary $\langle t, V\rangle$ satisfying φ and for each leaf v that belongs to V, the subtree $\langle v, U\rangle$ satisfies ψ.

Our logic is best explained using the specification of local reachability: let us identify the set of all summaries $\langle s, U\rangle$ such that there is a *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 $\varphi = \mu X.\langle ret\rangle R_1 \vee \langle loc\rangle X \vee \langle call\rangle X\{X\}$. The above means that X is the smallest set of summaries of the form $\langle s, U\rangle$ such that (1) there is a ret-labeled edge from s to some node in U, (2) there is a loc-labeled edge from s to t and there is a summary $\langle t, U\rangle$ in X, or (3) there is a $call$-labeled edge from s to t and a summary $\langle t, V\rangle$ in X such that from each $v \in V$, $\langle v, U\rangle$ is a summary in X. Notice that the above formula identifies the summaries in the natural way it will be *computed* on a pushdown system: compute the local summaries of each procedure, 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: $\mu Y.(p \vee \langle loc\rangle Y \vee \langle call\rangle\varphi\{Y\})$ 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 μ-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 $\langle call\rangle\varphi\{\psi_1, \ldots \psi_k\}$ says that the called procedure should satisfy φ, and the subtrees at the return nodes labeled with color i should satisfy the requirement ψ_i. While the concatenation operation 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-μ can be model-checked effectively. Given a model of a program as a recursive state machine [3, 7], or equivalently a pushdown system, and a VP-μ formula φ, we show that we can model-check whether the tree unfolding of the model satisfies φ in exponential time (the procedure is exponential in both the formula and the model). For a fixed formula φ, 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 algorithm. The complexity of model-checking VP-μ is EXPTIME-complete, which matches the complexity of model-checking the standard μ-calculus on pushdown systems (in fact, model-checking alternating reachability properties is already EXPTIME hard [26]).

Finally, we study some expressiveness issues for the logic VP-μ. We first show that VP-μ captures the temporal logic CARET, which is a linear-time temporal logic over visibly pushdown *words* that can capture several interesting pushdown specification properties. This shows that our branching-time logic captures the relevant counterpart logic over linear models, much the same way as the standard μ-calculus captures the temporal logic LTL. This makes VP-μ 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 *hierarchy theorem*: formulas of VP-μ that use k colors are strictly weaker than formulas that use $(k + 1)$ colors. Finally, we show that the satisfiability problem for VP-μ 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-μ is.

The paper is organized as follows. Section 2 introduces structured trees and summaries and Section 3 defines the logic VP-μ. In Section 4 we present various properties that can be expressed using VP-μ, including reachability, local reachability, expressions for various temporal modalities like *eventually* and *until*, security properties that involve inspection of stack, stack overflow properties, 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-μ 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 $I = \{call, ret, loc\}$ 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 $\mathcal{S} = (S, s_0, E, \lambda, \eta)$, 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 *call*), procedure returns (*ret*), or local statements within procedures (*loc*). For $a \in I$, we write $s \xrightarrow{a} s'$ as shorthand for "$(s, s') \in E$ 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 $1 \leq i < n$. We will extend η to paths in $\mathcal{S}$ as follows. Let e_i represent the transition (s_i, s_{i+1}) in the above path π. 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 $\mathcal{S}$ as *matched.* A path π in $\mathcal{S}$ is called matched if and only if $w = \eta(\pi)$ is of the form

$$w := loc \mid call\ w\ ret \mid ww.$$

Given nodes s and s' in $\mathcal{S}$, we call s' a *matching return* of s if and only if there is a matched path $\pi = s s_1 s_2 \ldots s_n$ such that $s_n \xrightarrow{ret} s'$. 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 that satisfies $MR(s_0) = \emptyset$.*

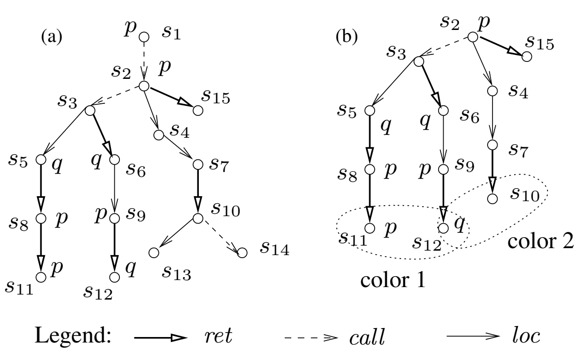

Figure 1. (a) A structured tree (b) A 2-colored summary

Intuitively, paths from the root in structured trees do not have "excess" 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 *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_{15} 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 *summary* 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-μ.

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

Observe how each summary describes a subtree along with a coloring of some of its leaves. For instance, the summary $\mathbf{s} = \langle s_2, \{s_{11}, s_{12}\}, \{s_{10}, s_{12}\} \rangle$ marks the subtree in Fig. 1-b. Such a tree may be constructed by taking the subtree of $\mathcal{S}$ 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_{15} 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-μ differs from classical calculi like the modal μ-calculus [20] in a crucial way: its syntax and semantics explicitly recognize the procedural structure of programs via modalities *call*, *ret* and *loc*. A distinction is made between *call*-edges, along which a program pushes frames on its stack, *ret*-edges, which require a pop from the stack, and *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 *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 φ of VP-μ are defined by:

$$\begin{aligned}\varphi \;:=\; & p \mid \neg p \mid X \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \mid \mu X.\phi \mid \nu X.\phi \\ & \langle call \rangle\ \varphi\{\psi_1, \psi_2, ..., \psi_k\} \mid [call]\ \varphi\{\psi_1, \psi_2, ..., \psi_k\} \mid \\ & \langle loc \rangle\ \varphi \mid [loc]\ \varphi \mid \langle ret \rangle\ R_i \mid [ret]\ R_i,\end{aligned}$$

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

Intuitively, the markers R_i in a formula are bound by $\langle call \rangle$ and $[call]$ modalities, and variables X are bound by fixpoint quantifiers μX and νX. We require our *call*-formulas to bind all the markers in their scope. Formally, let the *maximum marker index* $ind(\varphi)$ of a formula φ be defined inductively as: $ind(\varphi_1 \vee \varphi_2) = ind(\varphi_1 \wedge \varphi_2) = \max\{ind(\varphi_1), ind(\varphi_2)\}$; $ind(\langle loc \rangle \varphi) = ind([loc]\varphi) = ind(\mu X.\varphi) = ind(\nu X.\varphi) = ind(\varphi)$; and

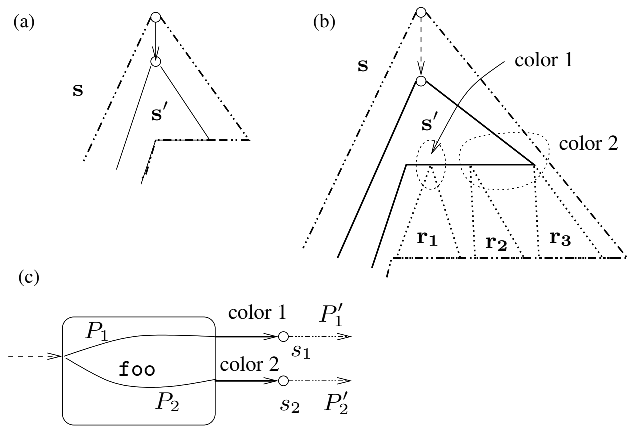

Figure 2. (a) Local modalities (b) Call modalities (c) Matching contexts.

$ind(\langle ret\rangle R_i) = ind([ret]R_i) = i$. For each $p \in AP$ and $X \in Var$, let us define $ind(p) = ind(X) = 0$. Finally, let us have $ind(\langle call\rangle\varphi\{\psi_1,\ldots,\psi_k\}) = ind([call]\varphi\{\psi_1,\ldots,\psi_k\}) = \max\{ind(\psi_1),\ldots,ind(\psi_k)\}$. We will only be interested in formulas where for every subformula χ of the form $\langle call\rangle\chi'\{\psi_1,\ldots,\psi_k\}$ or $[call]\chi'\{\psi_1,\ldots,\psi_k\}$, we have $ind(\chi') \leq k$. Such a formula φ is said to be *marker-closed* if $ind(\varphi) = 0$.

The set $Free(\varphi)$ of free variables in a VP-μ formula φ is defined as: $Free(\varphi_1 \vee \varphi_2) = Free(\varphi_1 \wedge \varphi_2) = Free(\varphi_1) \cup Free(\varphi_2)$; $Free(\langle loc\rangle\varphi) = Free([loc]\varphi) = Free(\varphi)$; and $Free(\langle ret\rangle R_i) = Free([ret]R_i) = \emptyset$. We have $Free(\langle call\rangle\varphi\{\psi_1,\ldots,\psi_k\}) = Free([call]\varphi\{\psi_1,\ldots,\psi_k\}) = Free(\varphi) \cup Free(\psi_1) \cup \ldots \cup Free(\psi_k)$; for each $p \in AP$ and $X \in Var$, $Free(p) = \emptyset$ and $Free(X) = \{X\}$. Finally, we have $Free(\mu X.\varphi) = Free(\nu X.\varphi) = Free(\varphi) \setminus \{X\}$. A formula φ is said to be *variable-closed* if it has $Free(\varphi) = \emptyset$. We call φ *closed* if it is marker-closed and variable-closed.

3.2 Semantics

Like in the modal μ-calculus, formulas in VP-μ encode sets, in this case sets of summaries. Also like in the μ-calculus, modalities and boolean and fixed-point operators allow us to encode computations on these sets.

To understand the semantics of local ($\langle loc\rangle$ and $[loc]$) modalities in VP-μ, consider the 2-colored summary $\mathbf{s} = \langle s_3, \{s_6\}, \{s_8\}\rangle$ in the tree $\mathcal{S}$ in Fig. 1-a. We observe that when control moves from node s_3 to s_5 along a local edge, the current context stays the same, but the set of returns that can end it and are reachable from the current control point gets restricted ($MR(s_5) \subseteq MR(s_3)$). The temporal requirements that we demand on return from the current context stay the same modulo this restriction. Consequently, the 2-colored summary $\mathbf{s}' = \langle s_5, \emptyset, \{s_8\}\rangle$ describes program flow from this point to the end of the current context and the requirements to be satisfied at the latter. We use modalities $\langle loc\rangle$ and $[loc]$ to reason about such *local succession*. For instance, in this case, summary $\mathbf{s}$ will be said to satisfy the formula $\langle loc\rangle q$.

An interesting visual insight about the structure of the tree $\mathcal{S}_\mathbf{s}$ for $\mathbf{s}$ comes from Fig. 2-a. Note that the tree $\mathcal{S}_{\mathbf{s}'}$ for $\mathbf{s}'$ "hangs"' from the former by a local edge; additionally, (1) every leaf of $\mathcal{S}_{\mathbf{s}'}$ is a leaf of $\mathcal{S}_\mathbf{s}$, and (2) such a leaf gets the same color in $\mathbf{s}$ and $\mathbf{s}'$.

Succession along call edges is more complex, because along such an edge, a frame is pushed on a program's stack and a new calling context gets defined. In Fig. 1-a, take the summary $\mathbf{s} = \langle s_2, \{s_{11}\}, \{s_{12}\}\rangle$, and suppose we want to assert a 3-parameter call formula $\langle call\rangle\varphi'\{q, p, tt\}$ at s_2. This requires us to consider a 3-colored summary of the context starting at s_3, where matching returns of s_3 satisfying q, p and tt are respectively marked by colors 1, 2 and 3. Clearly, this summary is $\mathbf{s}' = \langle s_3, \{s_6\}, \{s_8\}, \{s_6, s_8\}\rangle$. Our formula requires that $\mathbf{s}'$ satisfies φ'. In general, we could have formulas of the form $\varphi = \langle call\rangle\varphi'\{\psi_1, \psi_2, \ldots, \psi_k\}$, where ψ_i are arbitrary VP-μ formulas. To see what this means, look at the summaries $\mathbf{r_1} = \langle s_6, \emptyset, \{s_{12}\}\rangle$ and $\mathbf{r_2} = \langle s_8, \{s_{11}\}, \emptyset\rangle$, 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 φ is satisfied, we will need to consider a summary $\mathbf{s}''$ rooted at s_3 where the color i is assigned to nodes s_6 and s_8 precisely when $\mathbf{r_1}$ and $\mathbf{r_2}$ respectively satisfy ψ_i. Now, we require $\mathbf{s}''$ to satisfy φ'.

So far as the structures of these trees go, we find that the above requires a split of the tree $\mathcal{S}_\mathbf{s}$ for summary $\mathbf{s}$ 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 $\mathbf{s}'$, which must satisfy φ. At each leaf of $\mathcal{S}_{\mathbf{s}'}$ colored i, we must be able to *concatenate* a summary tree $\mathcal{S}_{\mathbf{s}''}$ satisfying ψ_i such that (1) every leaf in $\mathcal{S}_{\mathbf{s}''}$ is a leaf of $\mathcal{S}_\mathbf{s}$, and (2) each such leaf gets the same set of colors in $\mathcal{S}_\mathbf{s}$ and $\mathcal{S}_{\mathbf{s}''}$.

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-μ capture the pushdown structure of branching-time runs of a procedural program.

Let us now describe the semantics of VP-μ formally. A VP-μ formula φ is interpreted in an *environment* that interprets variables in $Free(\varphi)$ as sets of summaries in a structured tree $\mathcal{S}$. Formally, an *environment* is a map $\mathcal{E} : Free(\varphi) \rightarrow 2^{\mathbb{S}}$. Let us write $[\![\varphi]\!]^{\mathcal{S}}_{\mathcal{E}}$ to denote the set of summaries in $\mathcal{S}$ satisfying φ in environment $\mathcal{E}$ (usually $\mathcal{S}$ will be understood from the context, and we will simply write $[\![\varphi]\!]_{\mathcal{E}}$). For a summary $\mathbf{s} = \langle s, U_1, U_2, \ldots, U_k\rangle$, where $s \in S$ and $U_i \subseteq MR(s)$ for all i, $\mathbf{s}$ satisfies φ, i.e., $\mathbf{s} \in [\![\varphi]\!]_{\mathcal{E}}$, 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 $\mathbf{s} \in \mathcal{E}(X)$
- $\varphi = \varphi_1 \vee \varphi_2$ such that $\mathbf{s} \in [\![\varphi_1]\!]_{\mathcal{E}}$ or $\mathbf{s} \in [\![\varphi_2]\!]_{\mathcal{E}}$
- $\varphi = \varphi_1 \wedge \varphi_2$ such that $\mathbf{s} \in [\![\varphi_1]\!]_{\mathcal{E}}$ and $\mathbf{s} \in [\![\varphi_2]\!]_{\mathcal{E}}$
- $\varphi = \langle call\rangle\varphi'\{\psi_1, \psi_2, ..., \psi_m\}$, and there is a $t \in S$ such that (1) $s \xrightarrow{call} t$, and (2) the summary $\mathbf{t} = \langle t, V_1, V_2, \ldots, V_m\rangle$, where for all $1 \leq i \leq m$, $V_i = MR(t) \cap \{s' : \langle s', U_1 \cap MR(s'), \ldots, U_k \cap MR(s')\rangle \in [\![\psi_i]\!]_{\mathcal{E}}\}$, is such that $\mathbf{t} \in [\![\varphi']\!]_{\mathcal{E}}$
- $\varphi = [call]\ \varphi'\{\psi_1, \psi_2, ..., \psi_m\}$, and for all $t \in S$ such that $s \xrightarrow{call} t$, the summary $\mathbf{t} = \langle t, V_1, V_2, \ldots, V_m\rangle$, where for all $1 \leq i \leq m$, $V_i = MR(t) \cap \{s' : \langle s', U_1 \cap MR(s'), \ldots, U_k \cap MR(s')\rangle \in [\![\psi_i]\!]_{\mathcal{E}}\}$, is such that $\mathbf{t} \in [\![\varphi']\!]_{\mathcal{E}}$
- $\varphi = \langle loc\rangle\ \varphi'$, and there is a $t \in S$ such that $s \xrightarrow{loc} t$ and the summary $\mathbf{t} = \langle t, V_1, V_2, \ldots, V_k\rangle$, where $V_i = MR(t) \cap U_i$, is such that $\mathbf{t} \in [\![\varphi']\!]_{\mathcal{E}}$
- $\varphi = [loc]\ \varphi'$, and for all $t \in S$ such that $s \xrightarrow{loc} t$, the summary $\mathbf{t} = \langle t, V_1, V_2, \ldots, V_k\rangle$, where $V_i = MR(t) \cap U_i$, is such that $\mathbf{t} \in [\![\varphi']\!]_{\mathcal{E}}$

- $\varphi = \langle ret \rangle\, R_i$, and there is a $t \in S$ such that $s \xrightarrow{ret} t$ and $t \in U_i$
- $\varphi = [ret]\, R_i$, and for all $t \in S$ such that $s \xrightarrow{ret} t$, we have $t \in U_i$
- $\varphi = \mu X.\varphi'$, and $\mathbf{s} \in \mathbf{S}$ for all $\mathbf{S} \subseteq \mathbb{S}$ satisfying $[\![\varphi']\!]_{\mathcal{E}[X:=\mathbf{S}]} \subseteq \mathbf{S}$
- $\varphi = \nu X.\varphi'$, and there is some $\mathbf{S} \subseteq \mathbb{S}$ such that (1) $\mathbf{S} \subseteq [\![\varphi']\!]_{\mathcal{E}[X:=\mathbf{S}]}$ and (2) $\mathbf{s} \in \mathbf{S}$.

Here $\mathcal{E}[X := \mathbf{S}]$ is the environment $\mathcal{E}'$ such that (1) $\mathcal{E}'(X) = \mathbf{S}$, and (2) $\mathcal{E}'(Y) = \mathcal{E}(Y)$ for all variables $Y \neq X$. We say a node s *satisfies* a formula φ if the 0-colored summary $\langle s \rangle$ satisfies φ. A structured tree $\mathcal{S}$ rooted at s_0 is said satisfy φ if s_0 satisfies φ (we denote this by $\mathcal{S} \models \varphi$).

A few observations are in order. First, while VP-μ does not allow formulas of form $\neg\varphi$, it is closed under negation so long as we stick to closed formulas. Given a closed VP-μ formula φ, consider the formula $Neg(\varphi)$, defined inductively in the following way:

- $Neg(p) = \neg p$, $Neg(\neg p) = p$, $Neg(X) = X$
- $Neg(\varphi_1 \vee \varphi_2) = Neg(\varphi_1) \wedge Neg(\varphi_2)$, and $Neg(\varphi_1 \wedge \varphi_2) = Neg(\varphi_1) \vee Neg(\varphi_2)$
- If $\varphi = \langle call \rangle\; \varphi'\{\psi_1, \psi_2, ..., \psi_k\}$, then $Neg(\varphi) = [call]\; Neg(\varphi')\{Neg(\psi_1), Neg(\psi_2), \ldots, Neg(\psi_k)\}$
- If $\varphi = [call]\; \varphi'\{\psi_1, \psi_2, ..., \psi_k\}$, then $Neg(\varphi) = \langle call \rangle\; Neg(\varphi')\{Neg(\psi_1), Neg(\psi_2), \ldots, Neg(\psi_k)\}$
- $Neg(\langle loc \rangle \varphi') = [loc] Neg(\varphi')$, and $Neg([loc]\varphi') = \langle loc \rangle Neg(\varphi')$
- $Neg(\langle ret \rangle R_i) = [ret] R_i$, and $Neg([ret] R_i) = \langle ret \rangle R_i$
- $Neg(\mu X.\varphi) = \nu X.Neg(\varphi)$, and $Neg(\nu X.\varphi) = \mu X.Neg(\varphi)$

Performing induction on the structure of φ, we obtain:

THEOREM 1. *For all closed* VP-μ *formulas* φ, $[\![\varphi]\!]_{\perp} = \mathbb{S} \setminus [\![Neg(\varphi)]\!]_{\perp}$.

Second, note that the semantics of closed VP-μ formulas is independent of the environment; customarily, we will evaluate such formulas in the unique empty environment $\perp$: $\emptyset \rightarrow \mathbb{S}$. More importantly, the semantics of such a formula φ does not depend on current color assignments; in other words, for all $\mathbf{s} = \langle s, U_1, U_2, \ldots, U_k \rangle$, $\mathbf{s} \in [\![\varphi]\!]_{\perp}$ iff $\langle s \rangle \in [\![\varphi]\!]_{\perp}$. Consequently, when φ is closed, we can infer that "node s satisfies φ" from "summary $\mathbf{s}$ satisfies φ."

Third, every VP-μ formula $\varphi(X)$ with a free variable X can be viewed as a map $\varphi(X) : 2^{\mathbb{S}} \rightarrow 2^{\mathbb{S}}$ defined as follows: for all environments $\mathcal{E}$ and all summary sets $\mathbf{S} \subseteq \mathbb{S}$, $\varphi(X)(\mathbf{S}) = [\![\varphi(X)]\!]_{\mathcal{E}[X:=\mathbf{S}]}$. 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 $\mu X.\varphi(X)$ and $\nu X.\varphi(X)$ respectively evaluate to these two sets. From Tarski-Knaster, we also know that for a VP-μ formula φ with one free variable X, the set $[\![\mu X.\varphi]\!]_{\perp}$ lies in the sequence of summary sets $\emptyset, \varphi(\emptyset), \varphi(\varphi(\emptyset)), \ldots$, and that $[\![\nu X.\varphi]\!]_{\perp}$ is a member of the sequence $\mathbb{S}, \varphi(\mathbb{S}), \varphi(\varphi(\mathbb{S})), \ldots$.

Fourth, a VP-μ formula φ may also be viewed as a map $\varphi : (U_1, U_2, \ldots, U_k) \mapsto S'$, where S' is the set of all nodes s such that $U_1, U_2, \ldots, U_k \subseteq MR(s)$ and the summary $\langle s, U_1, U_2, \ldots, U_k \rangle$ satisfies φ. Naturally, $S' = \emptyset$ if no such s exists. Now, while a VP-μ formula can demand that the color of a return from the current context is i, it cannot assert that the color of a return *must not be* i (i.e., there is no formula of the form, say, $\langle ret \rangle \neg R_i$). It follows that the output of the above map will stay the same if we grow any of the sets U_i of matching returns provided as input. Formally, let $\mathbf{s} = \langle s, U_1, \ldots, U_k \rangle$ and $\mathbf{s}' = \langle s, U_1', \ldots U_k' \rangle$ be two summaries such that $U_i \subseteq U_i'$ for all i. Then for every environment $\mathcal{E}$ and every VP-μ formula φ, $\mathbf{s}' \in [\![\varphi]\!]_{\mathcal{E}}$ if $\mathbf{s} \in [\![\varphi]\!]_{\mathcal{E}}$.

Such monotonicity over markings has an interesting ramification. Let us suppose that in the semantics clauses for formulas of the form $\langle call \rangle \varphi'\{\psi_1, \psi_2, \ldots, \psi_k\}$ and $[call]\varphi'\{\psi_1, \psi_2, \ldots, \psi_k\}$, we allow $\mathbf{t} = \langle t, V_1, \ldots, V_k \rangle$ to be *any k-colored summary* such that (1) $\mathbf{t} \in [\![\varphi']\!]_{\mathcal{E}}$, and (2) for all i and all $s' \in V_i$, $\langle s', U_1 \cap MR(s'), U_2 \cap MR(s'), \ldots, U_k \cap MR(s') \rangle \in [\![\psi_i]\!]_{\mathcal{E}}$. Intuitively, from such a summary, one can grow the sets U_i to get the "maximal" $\mathbf{t}$ that we used in these two clauses. From the above discussion, VP-μ and this modified logic have equivalent semantics.

Finally, let us see what would happen if we did allow formulas of form $\langle ret \rangle \neg R_i$ (at a summary $\langle s, U_1, \ldots, U_k \rangle$, the above holds iff there is an edge $s \xrightarrow{ret} t$ such that $t \notin U_i$). It turns out that formulas involving the above need not be monotonic, and hence their fixpoints may not exist. To see why, consider the formula $\varphi = \langle call \rangle (\langle ret \rangle R_1 \wedge \langle ret \rangle (\neg R_1))\{X\})$ and a structured tree where the root s leads to two *ret*-children s_1 and s_2, both of which are leaves. Let $\mathbf{S_1} = \{\langle s_1, \emptyset \rangle\}$, and $\mathbf{S_2} = \{\langle s_1, \emptyset \rangle, \langle s_2, \emptyset \rangle\}$. Viewing φ as a map $\varphi : 2^{\mathbb{S}} \rightarrow 2^{\mathbb{S}}$, we see that $\varphi(\mathbf{S_1})$ is not a subset of $\varphi(\mathbf{S_2})$.

3.3 Bisimulation closure

Bisimulation is a fundamental relation in the analysis of labeled transition systems. The equivalence induced by a variety of branching-time logics, including the μ-calculus, coincides with bisimulation. In this section, we study the equivalence induced by VP-μ, that is, we want to understand when two nodes satisfy the same set of VP-μ formulas.

Consider two structured trees $\mathcal{S}_1 = (S_1, in_1, E_1, \lambda_1, \eta_1)$ and $\mathcal{S}_2 = (S_2, in_2, E_2, \lambda_2, \eta_2)$. Let S be $S_1 \cup S_2$ (we can assume that the sets S_1 and S_2 are disjoint), $\mathbb{S}$ be the set of all summaries in $\mathcal{S}_1$ and $\mathcal{S}_2$, and η denote the labeling of S as given by η_1 and η_2.

The *bisimulation relation* $\sim \;\subseteq S \times S$ is the greatest relation such that whenever $s \sim t$ holds, (1) $\eta(s) = \eta(t)$, (2) for every edge $s \xrightarrow{a} s'$, there is an edge $t \xrightarrow{a} t'$ such that $s' \sim t'$, and (3) for every edge $t \xrightarrow{a} t'$, there is an edge $s \xrightarrow{a} s'$ such that $s' \sim t'$. We write $\mathcal{S}_1 \sim \mathcal{S}_2$ if $in_1 \sim in_2$.

VP-μ is interpreted over summaries, so we need to lift the bisimulation relation to summaries. A summary $\langle s, U_1, \ldots U_k \rangle \in \mathbb{S}$ is said to be *bisimulation-closed* if for every pair $u, v \in MR(s)$ of matching returns of s, if $u \sim v$, then for each $1 \leq i \leq k$, $u \in U_i$ precisely when $v \in U_i$. Thus, in a bisimulation-closed summary, the marking does not distinguish among bisimilar nodes, and thus, return formulas (formulas of the form $\langle ret \rangle R_i$ and $[ret]R_i$) do not disntinguish among bisimilar nodes. Two bisimulation-closed summaries $\mathbf{s} = \langle s, U_1, \ldots, U_k \rangle$ and $\mathbf{t} = \langle t, V_1, \ldots, V_k \rangle$ in $\mathbb{S}$ and having the same number of colors are said to be *bisimilar*, written $\mathbf{s} \sim \mathbf{t}$, iff $s \sim t$, and for each $1 \leq i \leq k$, for all $u \in MR(s)$ and $v \in MR(t)$, if $u \sim v$, then $u \in U_i$ precisely when $v \in V_i$. Thus, roots of bisimilar summaries are bisimilar and the corresponding markings are unions of the same equivalence classes of the partitioning of the matching returns induced by bisimilarity. Note that every 0-ary summary is bisimulation-closed, and bisimilarity of 0-ary summaries coincides with bisimilarity of their roots.

Consider trees $\mathcal{S}$ and $\mathcal{T}$ in Fig. 3. We have named the nodes s_1, s_2, t_1, t_2 etc. and labeled some of them with proposition p. Note that $s_2 \sim s_4$, hence the summary $\langle s_1, \{s_2\}, \{s_4\} \rangle$ in $\mathcal{S}$ is not bisimulation-closed. Now consider the bisimulation-closed summaries $\langle s_1, \{s_2, s_4\}, \{s_3\} \rangle$ and $\langle t_1, \{t_2\}, \{t_3\} \rangle$. By our definition they are bisimilar. However, the (bisimulation-closed) summaries $\langle s_1, \{s_2, s_4\}, \{s_3\} \rangle$ and $\langle t_1, \{t_3\}, \{t_2\} \rangle$ are not.

We now want to prove that bisimilar summaries satisfy the same VP-μ formulas. For an inductive proof, we need to consider the environment also. We assume that the environment $\mathcal{E}$ maps VP-μ

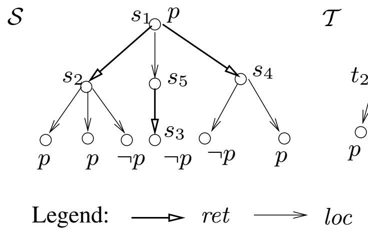

Figure 3. Bisimilarity.

variables to subsets of $\mathbb{S}$ (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 $\mathbf{s} \sim \mathbf{t}$, $\mathbf{s} \in \mathcal{E}(X)$ precisely when $\mathbf{t} \in \mathcal{E}(X)$.

LEMMA 1. *If $\mathcal{E}$ is a bisimulation-closed environment and φ is a* VP-μ *formula, $[\![\varphi]\!]_{\mathcal{E}}$ is bisimulation-closed.*

Proof: The proof is by induction on the structure of the formula φ. Consider two bisimulation-closed bisimilar summaries $\mathbf{s} = \langle s, U_1, \dots U_k \rangle$ and $\mathbf{t} = \langle t, V_1, \dots V_k \rangle$, and a bisimulation-closed environment $\mathcal{E}$. We want to show that $\mathbf{s} \in [\![\varphi]\!]_{\mathcal{E}}$ precisely when $\mathbf{t} \in [\![\varphi]\!]_{\mathcal{E}}$.

If φ is a proposition or negated proposition, the claim follows from bisimilarity of nodes s and t. When φ is a variable, the claim follows from bisimulation closure of $\mathcal{E}$. We consider a few interesting cases.

Suppose $\varphi = \langle ret \rangle R_i$. $\mathbf{s}$ satisfies φ precisely when s has a return-edge to some node s' in U_i. 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_i, and thus $\mathbf{t}$ must satisfy φ.

Suppose $\varphi = \langle call \rangle \varphi' \{\psi_1, \dots \psi_m\}$. Suppose $\mathbf{s}$ satisfies φ. Then there is a call-successor s' of s such that $\langle s', U'_1, \dots U'_m \rangle$ satisfies φ', where $U'_i = \{u \in MR(s') \mid \langle u, U_1 \cap MR(u), \dots U_k \cap MR(u) \rangle \in [\![\psi_i]\!]_{\mathcal{E}}\}$. Since s and t are bisimilar, there exists a call-successor t' of t such that $s' \sim t'$. For each $1 \leq i \leq m$, let $V'_i = \{v \in MR(t') \mid \exists u \in U'_i.\, u \sim v\}$. Verify that the summaries $\langle s', U'_1, \dots U'_m \rangle$ and $\langle t', V'_1, \dots V'_m \rangle$ are bisimilar. By induction hypothesis, $\langle t', V'_1, \dots V'_m \rangle$ satisfies φ'. Also, for each $v \in V'_i$, for $1 \leq i \leq m$, the summary $\langle v, V_1 \cap MR(v), \dots V_k \cap MR(v) \rangle$ is bisimilar to $\langle u, U_1 \cap MR(u), \dots U_k \cap MR(u) \rangle$, for some $u \in U_i$, and hence, by induction hypothesis, satisfies ψ_i. This establishes that $\mathbf{t}$ satisfies φ.

Case $\varphi = \mu X.\varphi'$. Let $X_0 = \emptyset$. For $i \geq 0$, let $X_{i+1} = [\![\varphi']\!]_{\mathcal{E}[X:=X_i]}$. Then $[\![\varphi]\!]_{\mathcal{E}} = \cup_{i \geq 0} X_i$. Since $\mathcal{E}$ is bisimulation closed, and X_0 is bisimulation-closed, by induction, for $i \geq 0$, each X_i is bisimulation-closed, and so is $[\![\varphi]\!]_{\mathcal{E}}$. $\square$

As a corollary, we get that if $\mathcal{S}_1 \sim \mathcal{S}_2$, then for every closed VP-μ formula φ, $\mathcal{S}_1 \models \varphi$ precisely when $\mathcal{S}_2 \models \varphi$. The proof also shows that to decide whether a structured tree satisfies a closed VP-μ formula, during the fixpoint evaluation, one can restrict attention only to bisimulation-closed summaries. In other words, we can redefine the semantics of VP-μ so that the set $\mathbb{S}$ of summaries contains only bisimulation-closed summaries. It also suggests that to evaluate a closed VP-μ formula over a structured tree, one can reduce the structured tree by collapsing bisimilar nodes as in the case of classical model checking.

If the two structured trees $\mathcal{S}_1$ and $\mathcal{S}_2$ are not bisimilar, then there exists a μ-calculus formula (in fact, of the much simpler Hennessy-Milner modal logic, which does not involve any fixpoints) that is satisfied at the roots of only one of the two trees. This does not immediately yield a VP-μ formula that distinguishes the two trees because VP-μ formulas cannot assert requirements across return-edges in a direct way. However, a more complex encoding is possible. We defer the details to the full paper. Thus, two structured trees satisfy the same set of closed VP-μ formulas precisely when they are bisimilar.

Let us consider two arbitrary nodes s and t (in the same structured tree, or in two different structured trees). When do these two nodes satisfy the same set of closed VP-μ formulas? From the arguments so far, bisimilarity is sufficient. However, the satisfaction of a closed VP-μ formula at a node s depends solely on the subtree rooted at s and truncated at the matching returns of s. In fact, the full subtree rooted at s may not be *structured* as it can contain excess returns. For a structured tree $\mathcal{S}$, and a node s, let $\mathcal{S}_s$ denote the structured tree rooted at s obtained by deleting all the return-edges leading to the nodes in $MR(s)$. For instance, in Fig. 3, $\mathcal{S}_{s_1}$ comprises nodes s_1 and s_5 and the *loc*-edge connecting them. It is easy to check that if φ is a closed VP-μ formula then $\langle s \rangle$ satisfies φ in the original structured tree precisely when $\mathcal{S}_s$ satsifies φ. If s and t are not bismilar, and the non-bisimilarity can be established within the structured subtrees $\mathcal{S}_s$ and $\mathcal{S}_t$ rooted at these nodes, then some closed VP-μ formula can distinguish them.

THEOREM 2. *Two nodes s and t satisfy the same set of closed* VP-*μ formulas precisely when $\mathcal{S}_s \sim \mathcal{S}_t$.*

4. Specifying requirements

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

Reachability Let us express in VP-μ the reachability property $Reach$ 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 π for $Reach$ that starts with an edge $s \xrightarrow{call} s'$. There are two possibilities: (1) a node satisfying p 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'', $Reach$ 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 $Reach$. Then we want the summary $\langle s, MR(s) \rangle$ to satisfy $\langle call \rangle \varphi' \{X\}$, where φ' 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 $\langle call \rangle X\{\}$.

Before we get to φ', note that the formula $\langle loc \rangle X$ captures the case when π starts with a local transition. Combining the two cases and using CTL-style notation, the formula we want is

$$EF\ p = \mu X.(p \vee \langle loc \rangle X \vee \langle call \rangle X\{\} \vee \langle call \rangle \varphi'\{X\}).$$

Now observe that φ' also expresses reachability, except (1) its target needs to satisfy $\langle ret \rangle R_1$, 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 $\langle ret \rangle R_1$. It is easy to verify that

$$\varphi' = \mu Y.(\langle ret \rangle R_1 \vee \langle loc \rangle Y \vee \langle call \rangle Y\{Y\}).$$

We cannot merely substitute p for $\langle ret \rangle R_1$ in φ' to express local reachability of p. However, a formula $EF^l\ p$ for this property is easily obtained by restricting the formula $EF\ p$:

$$EF^l\ p = \mu X.(p \vee \langle loc\rangle X \vee \langle call\rangle \varphi'\{X\}).$$

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 $free(x)$ that is true immediately after a line where x is freed, $EF^l\ free(x)$ holds at the entry point of procedure `foo` (node s_1).

Generalizing, we will allow p to be any VP-μ formula that keeps $EF\ p$ and $EF^l\ p$ 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

$$AF\ p = \mu X.(p \vee ([loc]X \wedge [call]\varphi''\{X\})),$$

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

$$\varphi'' = \mu Y.(p \vee ([ret]R_1 \wedge [loc]Y \wedge [call]Y\{Y\})).$$

As in the previous case, we can define a corresponding operator AF^l that asserts local reachability along all paths. For instance, in Fig. 4, $AF^l\ free(x)$ 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 so obtained into their call sites. This "interprocedural" reasoning distinguishes it from logics such as the μ-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-μ and the μ-calculus. Consider the fixpoint computation for the μ-calculus formula $\mu X.(p \vee \langle\rangle X)$ 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-μ formula $EF\ p$. Assume that φ' (intuitively, the set of "jumps" from calls to returns") has already been evaluated, and consider the set $\mathbf{S_X}$ of summaries for $EF\ p$. At the end of the i-th phase, this set contains all $\mathbf{s} = \langle s\rangle$ such that s has a path consisting of $(i-1)$ *call* and *loc*-transitions to a node satisfying p. However, because of the subformula $\langle call\rangle\varphi'\{X\}$, it also includes all $\mathbf{s}$ 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-μ formula $EG\ p$ for this is obtained from the identity $EG\ p = Neg(AF\ Neg(p))$. The formula $AG\ p$, which asserts that p holds on each point 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 π from the current node such that p_1 holds at every point on π till it reaches the end of the current context or a node satisfying p_2 (if π 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 π. The for-all-paths analogs of these properties ($A(p_1\ U\ p_2)$ and $A(p_1\ W\ p_2)$) aren't hard to write either.

Neither is it difficult to express local or same-context versions of these properties. Consider the maximal subsequence π' of a program path π from s such that each node of π' belongs to the same procedural context as s. A VP-μ formula $EG^l p$ for *existential local invariance* demands that p holds on some such π', while $AG^l p$ asserts the same for all π'. Similarly, we can define existential and universal *local until* properties, and corresponding VP-μ formulas $E(p_1\ U^l\ p_2)$ and $A(p_1\ U^l\ p_2)$. For instance, in Fig. 4, $E(\neg free(g)\ U^l\ free(x))$ holds at node s_1 (whereas $E(\neg free(g)\ U\ free(x))$ 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

$$E(p_1\ W^l\ p_2) = \nu X.((p_1 \vee p_2) \wedge (p_2 \vee \langle loc\rangle X \vee \langle call\rangle\varphi'\{X\})),$$

where φ' asserts local reachability of $\langle ret\rangle R_1$ as before.

```
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 }
```

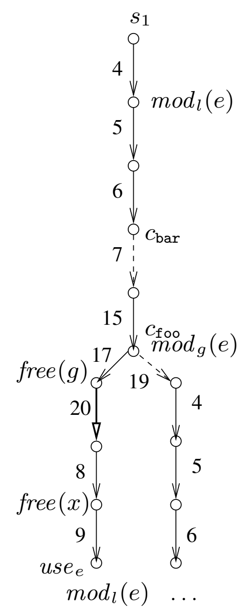

Figure 4. A C example

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 $mod(e)$ (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-μ formula that demands that e is very busy at s is

$$A((EF\ use(e) \wedge \neg mod(e))\ W\ use(e)).$$

Note that this property uses the power of VP-μ to reason about branching time.

However, complications arise if we are considering interprocedural paths 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 $e = (\mathtt{a}^2 + \mathtt{b}^2)$, used in line 9, is very busy at line 6. We would, as before, track changes to a and b between lines 6 and 9. But we must note that as soon as an interprocedural path π between these two points leaves the current context, the observable b falls out of scope. This path may subsequently come back to procedure foo because of recursion, and a new instance of b may be created. However, modification of this new instance of b should not cause e not to be very busy in the current context. In other words, we should only be concerned with the *local* uses of b. For the same reason, use of e in a different context should not be of interest of us. On the other hand, the global variable a needs to be tracked through every context along a path before a local use of e on it.

Local temporal properties come of use in covering such cases. Let us define two propositions $mod_g(e)$ and $mod_l(e)$ that are true at points where, respectively, a global or a local variable in e is modified. The VP-μ property we assert at s is

$$\nu X.(((EF^l\ use(e)) \wedge \neg mod_g(e) \wedge \neg mod_l(e)) \vee use(e)) \\ \wedge (use(e) \vee ([loc]X \wedge [call]\psi\{X, tt\})),$$

where the formula ψ tracks global variables like a in new contexts:

$$\psi \quad = \quad \mu Y.(\neg mod_g(e) \wedge (([ret]R_1 \wedge \langle ret\rangle R_2) \\ \vee ([call]Y\{Y, tt\} \wedge [loc]Y))).$$

Note the use of the formula $\langle ret\rangle R_2$ to ensure that $[ret]R_1$ is not vacuously true.

Pushdown specifications The domain where VP-μ 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-μ: that of local temporal properties. For instance, the formula $EF^l p$ 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 μ-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-μ formulas for (non-local) reachability and invariance. To understand why, recall that VP-μ 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 foo is called, procedure 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 μ-calculus. While the temporal logic CARET can express such properties, it requires a past-time operator called *caller* to do so. To express this property in VP-μ, we define propositions $c_{\texttt{foo}}$ and $c_{\texttt{bar}}$ that respectively hold at every call site for foo and bar. Now, assuming control starts in foo, consider the formula

$$\varphi = EF(c_{\texttt{bar}} \wedge \langle call\rangle(EF\ c_{\texttt{foo}})\{\}).$$

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

Other stack inspection properties expressible in VP-μ include "when procedure 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-μ 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-μ can specify requirements that safeguard against such errors. Once again, nested modalities come handy. Suppose we assert $AG(\langle call\rangle ff\{\})$ throughout 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 $AG\ Termin$ at the initial program point, where

$$Termin = [call](AF^l(\langle ret\rangle R_1))\{tt\}.$$

Preconditions and postconditions For a program state s, let us consider the set $Jmp(s)$ of nodes to which a call from s may return. Then the requirement: "property p holds at some node in $Jmp(s)$" is captured by the VP-μ formula $\langle jump\rangle p = \langle call\rangle(EF^l\ \langle ret\rangle R_1)\{p\}$. The dual formula $[jump]p$, 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 Pre is satisfied when A is called, then if A terminates, postcondition $Post$ holds upon return. Total correctness, additionally, requires A to terminate. These requirements cannot be expressed using regular specifications. In VP-μ, let us say that at every call site to procedure A, proposition c_A holds. Then a formula for partial correctness, asserted at the initial program state, is

$$AG((Pre \wedge c_A) \Rightarrow [jump]Post).$$

Total correctness is expressed as

$$AG((Pre \wedge c_A) \Rightarrow (Termin \wedge [jump]Post)).$$

Access control The ability of VP-μ 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 $access$ is true when it does) only if it is called with a high privilege ($priv$ holds when it is). It is tempting to write a property $\varphi = \neg priv \Rightarrow AG\ (\neg access)$ 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 φ will not let A access the database even in this new context. The formula we are looking for is really $\varphi' = \neg priv \Rightarrow AG^l\ (\neg access)$, asserted at every call site for A.

Multiple return conditions As we shall see in Section 6.2, the theoretical expressiveness of VP-μ depends on the fact that we can pass multiple return conditions as "parameters" to VP-μ 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 φ at the program point calling `bar`:

$$\varphi = \langle call\rangle\psi'\{[loc]\neg free(x), [loc]free(x)\},$$

where ψ' 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 $\langle ret\rangle R_1$, or (2) satisfy $\neg free(g)$ until $\langle ret\rangle R_2$ 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 $(\langle M_1, M_2, \ldots, M_m\rangle, start)$, where each M_i is a *procedure* of the form $(L_i, B_i, Y_i, En_i, Ex_i, \delta_i, \kappa_i)$. 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*.
- $Y_i : B_i \to \{1, 2, \ldots, m\}$ is a map that assigns a procedure to every box.
- $En_i \subseteq L_i$ is a non-empty set of *entry locations*, and $Ex_i \subseteq L_i$ is a non-empty set of *exit locations*.
- Let $Calls_i = \{(b, en) : b \in B_i, en \in En_{Y_i(b)}\}$ denote the set of *calls* in M_i, and let $Retns_i = \{(b, ex) : b \in B_i, ex \in Ex_{Y_i(b)}\}$ denote the set of *returns* in M_i. Then $\delta_i \subseteq (L_i \cup Retns_i) \times (L_i \cup Calls_i)$ defines the set of RSM edges.
- κ_i is a labeling function $\kappa_i : (L_i \cup Calls_i \cup Retns_i) \to 2^{AP}$ that associates a set of propositions to each control location, call and return.

A control location $start \in \bigcup_i L_i$ in one of the components is chosen as the *initial location*. We assume that for every distinct i and j, L_i, $Calls_i$, $Retns_i$, N_j, $Calls_j$, and $Retns_j$ are pairwise disjoint. We refer to arbitrary calls, returns and control locations in M as *vertices*. The set all vertices is given by $V = \bigcup_i (L_i \cup Calls_i \cup Retns_i)$, 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 δ_i, κ_i and Y_i are denoted respectively by $\delta : V \to V$, $\kappa : V \to 2^{AP}$, and $Y : B \to \{1, 2, \ldots, m\}$.

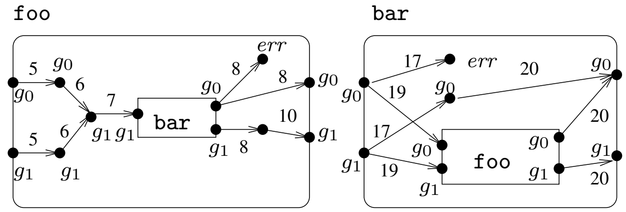

Figure 5. A recursive state machine.

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 `ALLOC` 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 $\mathcal{C}(M) = (C, c_0, E_C, \lambda_C, \eta_C)$, 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 $\lambda_C : C \to 2^{AP}$ and $\eta_C : E_C \to \{call, ret, loc\}$ respectively label configurations and transitions. Stealing notation for structured trees, we write $c \xrightarrow{s} c'$ if $(c, c') \in E_C$ and $\eta_C((c, c')) = a$.

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

- $\gamma = \epsilon$ 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 $i \in \{1, \ldots, n-1\}$, $b_{i+1} \in B_{Y(b_i)}$.

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

- **Local move:** $u \in (L_i \cup Retns_i) \setminus Ex_i$, $(u, u') \in \delta_i$, $\gamma' = \gamma$, and $a = loc$;
- **Procedure call:** $u = (b, en) \in Calls_i$, $u' = en$, $\gamma' = \gamma.b$, and $a = call$;
- **Return from a call:** $u \in Ex_i$, $\gamma = \gamma'.b$, $u' = (b, u)$, and $a = ret$.

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 $\mathcal{C}(M) = (C, c_0, E_C, \lambda_C, \eta_C)$. The configuration tree of M is a structured tree $Conf(M) = (S, s_0, E, \lambda, \eta)$, 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. $c_0 \in S$.

2. Let $s.c \in S$ for some $s \in C^*$ and some configuration $c \in C$, and suppose $c \xrightarrow{a} c'$ for some $a \in \{call, loc, ret\}$ and some $c' \in C$. Then $s.c.c' \in S$ and $s.c \xrightarrow{a} s.c.c'$.

The above also defines the transition-labeling map η in $Conf(M)$. The node-labeling function λ is given by: for each node $s = s'.c$, $\lambda(s) = \lambda_C(c)$. The initial node is $s_0 = c_0$. Finally, we define a map $Curr : S \to C$ that gives us the *current configuration* for any node in $Conf(M)$ as follows: for all $s \in C^*$ and all $c \in C$, if $s.c \in S$ then $Curr(s.c) = c$.

Summaries in $Conf(M)$ are now defined as in Section 2.1. We identify the summary $\mathbf{s_0} = \langle s_0 \rangle$ as the *initial summary* in $Conf(M)$. We say that the RSM M *satisfies* a closed VP-μ formula φ if and only if $\mathbf{s_0} \in [\![\varphi]\!]_\perp$.

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 $Curr(s) = Curr(s')$ 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-μ formulas are concerned.

5.2 Model-checking VP-μ over RSMs

For a closed VP-μ formula φ and an RSM M, the *model-checking problem* of φ over M is to determine if M satisfies φ.

Recall that configurations of M are of the form (γ, u), where γ is a stack of boxes and u is a vertex in M. Clearly, the set $MR(s)$ for the current node s and the set φ_{ret} of ret-subformulas that hold at the current summary depend on the current stack γ. 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-μ formulas go, we are only interested in the *top* of γ.

To formalize this intuition, let us define a map $Erase : (\gamma, u) \mapsto u$ 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 $\mathbf{s} = \langle s, U_1, U_2, \ldots, U_k \rangle$ and $\mathbf{s}' = \langle s', U_1', U_2', \ldots, U_k' \rangle$, where $Curr(s) = (\gamma b, u)$ and $Curr(s') = (\gamma' b, u)$. We call $\mathbf{s}$ and $\mathbf{s}'$ *top-equivalent* if and only if for all i, $Erase(U_i) = Erase(U_i')$. Then we have:

LEMMA 2. *Let $\mathbf{s_1}$ and $\mathbf{s_2}$ be two top-equivalent k-colored summaries in the configuration tree of an RSM M. Then for any closed* VP-μ *formula φ, $\mathbf{s_1}$ satisfies φ iff $\mathbf{s_2}$ satisfies φ.*

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-μ 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-stack reachable* from u if there is a path in $Conf(M)$ from (ϵ, u) to (ϵ, v). It is well-known that for any u, the set $Reach_\epsilon(u)$ of vertices empty-stack 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 $u \in V_l$ is in procedure M_l. The set $Ret_b(u)$ of possible returns from u to a box b with $Y(b) = l$ consists of all (b, ex) such that $ex \in Reach_\epsilon(u) \cap Ex_l$. Clearly, for any u and b, the set $Ret_b(u)$ can be computed in time polynomial in M. Also, we will use the notation $Ret(u) = \cup_b Ret_b(u)$.

Now, let n be the arity of the formula φ. A *stackless summary* is a tuple $\langle u, Ret_1, Ret_2, \ldots, Ret_k \rangle$, where $0 \leq k \leq n$, and for some b, $Ret_j \subseteq Ret_b(u)$ for all j. The set of all stackless summaries in M is denoted by SLS.

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

- If $\varphi = p$, for $p \in AP$, then $Eval(\varphi, \mathcal{E}_{SL})$ consists of all $\langle v, Ret_1, Ret_2, \ldots, Ret_k \rangle$ such that $p \in \kappa(v)$ and $k \leq n$.
- If $\varphi = \neg p$, for $p \in AP$, then $Eval(\varphi, \mathcal{E}_{SL})$ consists of all $\langle v, Ret_1, Ret_2, \ldots, Ret_k \rangle$ such that $p \notin \kappa(v)$ and $k \leq n$.
- If $\varphi = X$, for $X \in Var$, then $Eval(\varphi, \mathcal{E}_{SL}) = \mathcal{E}_{SL}(X)$.
- If $\varphi = \varphi_1 \vee \varphi_2$ then $Eval(\varphi, \mathcal{E}_{SL}) = Eval(\varphi_1, \mathcal{E}_{SL}) \cup Eval(\varphi_2, \mathcal{E}_{SL})$.
- If $\varphi = \varphi_1 \wedge \varphi_2$ then $Eval(\varphi, \mathcal{E}_{SL}) = Eval(\varphi_1, \mathcal{E}_{SL}) \cap Eval(\varphi_2, \mathcal{E}_{SL})$.
- If $\varphi = \langle call \rangle\ \varphi'\{\psi_1, \psi_2, ..., \psi_m\}$, then $Eval(\varphi, \mathcal{E}_{SL})$ consists of all $\langle (b, en), Ret_1, Ret_2, \ldots, Ret_k \rangle$ such that for some $\langle en, Ret_1', Ret_2', ..., Ret_m' \rangle \in Eval(\varphi', \mathcal{E}_{SL})$, and for all $(b', ex') \in Ret_i'$, where $i = 1, \ldots, m$, we have:
  1. $b' = b$
  2. $\langle (b', ex'), Ret_1'', Ret_2'', \ldots, Ret_k'' \rangle \in Eval(\psi_i, \mathcal{E}_{SL})$, where $Ret_j'' = Ret_j \cap Ret((b', ex'))$ for all $j \leq k$.
- If $\varphi = [call]\ \varphi'\{\psi_1, \ldots, \psi_m\}$, then we have $Eval(\varphi, \mathcal{E}_{SL}) = Eval(\rho, \mathcal{E}_{SL}) \cup Noncall$, where $\rho = \langle call \rangle\ \varphi'\{\psi_1, \psi_2, ..., \psi_m\}$, and $Noncall$ comprises all summaries $\langle u, Ret_1, \ldots, Ret_k \rangle$, for $k \leq n$, such that u is not a call. This works because the $[call]$ modality may hold vacuously, and because a node in an RSM can have at most one outgoing call-transition.
- If $\varphi = \langle loc \rangle\ \varphi'$, then $Eval(\varphi, \mathcal{E}_{SL})$ consists of all stackless summaries $\langle u, Ret_1, \ldots, Ret_k \rangle$ such that for some v satisfying $(u, v) \in \delta$, we have $\langle v, Ret_1 \cap Ret(v), \ldots, Ret_k \cap Ret(v) \rangle \in Eval(\varphi', \mathcal{E}_{SL})$.
- If $\varphi = [loc]\ \varphi'$, then $Eval(\varphi, \mathcal{E}_{SL})$ consists of all stackless summaries $\langle u, Ret_1, \ldots, Ret_k \rangle$ such that for all v satisfying $(u, v) \in \delta$, we have $\langle v, Ret_1 \cap Ret(v), \ldots, Ret_k \cap Ret(v) \rangle \in Eval(\varphi', \mathcal{E}_{SL})$.
- If $\varphi = \langle ret \rangle\ R_i$, then $Eval(\varphi, \mathcal{E}_{SL})$ consists of all summaries $\langle ex, Ret_1, \ldots, Ret_k \rangle$ such that (1) $Ret_i = \{(b, ex)\}$, where $(b, ex) \in Ret(ex)$, and (2) for all $j \neq i$, $Ret_j = \emptyset$.
- If $\varphi = [ret]\ R_i$, then $Eval(\varphi, \mathcal{E}_{SL}) = Eval(\langle ret \rangle R_i, \mathcal{E}_{SL}) \cup Nonret$, where $Nonret$ has all summaries $\langle u, Ret_1, \ldots, Ret_k \rangle$ such that u is not an exit.
- If $\varphi = \mu X.\varphi'$, then $Eval(\varphi, \mathcal{E}_{SL}) = FixPoint\ (X, \varphi', \mathcal{E}_{SL}[X := \emptyset])$.
- If $\varphi = \nu X.\varphi'$, then $Eval(\varphi, \mathcal{E}_{SL}) = FixPoint\ (X, \varphi', \mathcal{E}_{SL}\ [X := SLS])$.

Here $FixPoint\ (X, \varphi, \mathcal{E}_{SL})$ is a fixpoint computation function that uses the formula φ as a monotone map between subsets of SLS, and iterates over variable X. This computation in described in Fig. 6.

```
FIXPOINT (X, φ, E_SL)
  X' = Eval(φ, E_SL)
  if X' = E_SL(X)
    then return X'
    else return FIXPOINT (X, φ', E_SL[X := X'])
```

Figure 6. Fixpoint computation for VP-μ.

The following theorem is easily proved:

THEOREM 3. *For an RSM M and a closed* VP-μ *formula φ, $Conf(M)$ satisfies φ if and only if $\langle s_0 \rangle \in Eval(\varphi, \perp)$. Furthermore, $Eval(\varphi, \perp)$ 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 $N = nN_V 2^{\theta n}$. 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 φ of arity k is bounded by $O(N^2\theta|\varphi|)$ (the most expensive modality is $\langle call\rangle\varphi'\{\psi_1, \ldots, \psi_n\}$, where we have to match an "inner" summary satisfying φ' as well as n "outer" summaries satisfying the ψ_i-s). For a fixpoint formula φ with one fixpoint variable, we may need N such evaluations, so that the total time required to evaluate $Eval(\varphi, \bot)$ is $O(N^3\theta|\varphi|)$. For a formula φ of alternation depth d, this evaluation takes time $O(N^{3d}\theta^d\ |\varphi|)$.

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 φ from a μ-calculus formula f expressing such a property such that (1) the size of φ is linear in the size of f, and (2) M satisfies φ if and only if M satisfies f. It follows that model-checking a closed VP-μ formula φ on an RSM M is EXPTIME-hard.

Combining all of the above, we have:

THEOREM 4. *Model-checking a* VP-μ *formula* φ *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* $\mathcal{A}$ is a pushdown automaton with an added restriction: its input alphabet Σ is partitioned into a *call alphabet* Σ_c, a *local alphabet* Σ_l, and a *return alphabet* Σ_r. On reading a call or return symbol, $\mathcal{A}$ must respectively push and pop precisely one symbol on/from the stack; on a local input symbol, $\mathcal{A}$ 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 φ over the set of propositions P, there exists $\mathcal{A}_\varphi$ over infinite words with a *Büchi* acceptance condition, over the alphabet $\Sigma_i = \{(loc, v) \mid v \in 2^P\}$, $\Sigma_c = \{(call, v) \mid v \in 2^P\}$ and $\Sigma_r = \{(ret, v) \mid v \in 2^P\}$ that accepts precisely the models that satisfy φ. We can show that:

THEOREM 5. *For any nondeterministic Büchi visibly pushdown automaton* $\mathcal{A}$ *over the alphabets described above, there is a* VP-μ *formula* $\varphi_\mathcal{A}$ *that holds in a structured tree if and only if there is some path in the tree that is accepted by* $\mathcal{A}$.

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 φ, 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 not 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, $\varphi_\mathcal{A}$ will be a 1-ary formula, and will have variables X_q and $S_{q,q',b}$, where q, q' in $\mathcal{A}$ 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 $\varphi_\mathcal{A}$ 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 $(call, v)$, 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 (ret, v), pop b and go to q'). Let the set of transitions of $\mathcal{A}$ be Δ. Also, for each valuation $v \in 2^P$, let $\psi_v = \bigwedge_{p\in v} p\ \wedge\ \bigwedge_{p\notin v} \neg p$.

Then, the formula $\varphi_\mathcal{A}$ will be the formula corresponding to $X_{q_{in}}$ when taking the least fixpoint of the following equations:

$$\begin{aligned}
X_q \quad &= \quad \bigvee_{(q,v,q')\in\Delta} (\psi_v \wedge \langle loc\rangle X_{q'}) \\
&\quad \vee \bigvee_{(q,v,q',b)\in\Delta} (\psi_v \wedge \langle call\rangle X_{q'}\{\mathit{ff}\}) \\
&\quad \vee \bigvee_{(q,v,q_1,b)\in\Delta} \bigvee_{q'} (\psi_v \wedge \langle call\rangle S_{q_1,q',b}\{X_{q'}\}) \\
S_{q,q',b} \quad &= \quad \bigvee_{(q,v,b,q')\in\Delta} (\psi_v \wedge \langle ret\rangle R_1) \\
&\quad \vee \bigvee_{(q,v,q'')\in\Delta} (\psi_v \wedge \langle loc\rangle S_{q'',q',b}) \\
&\quad \vee \bigvee_{(q,v,q_1,b')\in\Delta} \bigvee_{q_2} (\psi_v \wedge \langle call\rangle S_{q_1,q_2,b'}\{S_{q_2,q',b}\}).
\end{aligned}$$

6.2 An arity hierarchy

Now we show that the expressiveness of VP-μ formulas increases with their arity. For two structured trees $\mathcal{S}_1$ and $\mathcal{S}_2$ with initial nodes s_1 and s_2, we say $\mathcal{S}_1$ and $\mathcal{S}_2$ are *distinguished* by a closed, k-ary VP-μ formula φ if and only if s_1 satisfies φ and s_2 does not. Then we have:

THEOREM 6. *For every* $k \geq 1$*, there is a a closed* $(k+1)$*-ary formula* φ_{k+1}*, and two structured trees* $\mathcal{S}_1$ *and* $\mathcal{S}_2$*, such that* φ_{k+1} *can distinguish between* $\mathcal{S}_1$ *and* $\mathcal{S}_2$*, but no closed* k*-ary* VP-μ *formula can.*

We will sketch the proof for the case $k = 1$. Before we do so, we need some extra machinery. More precisely, we will define a preorder called *quasi-bisimilarity* over summaries that takes into account their coloring. It turns out that VP-μ respects this preorder.

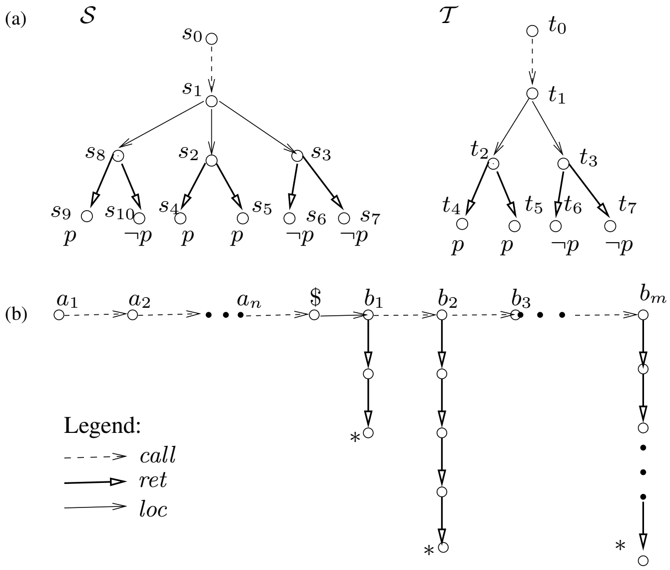

Figure 7. (a) An arity hierarchy (b) Tree encoding for undecidability.

Consider a pair of k-colored summaries $\mathbf{s} = \langle s, U_1, \ldots, U_k \rangle$ and $\mathbf{t} = \langle t, V_1, \ldots, V_k \rangle$ such that each path in the trees rooted at s and t comprises a chain of *loc*-edges followed by one *ret*-edge leading to a leaf. Let S and T respectively be the sets of non-leaf nodes in these trees. We say that $\mathbf{s}$ and $\mathbf{t}$ 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 $s' \xrightarrow{loc} s''$, there is a t'' such that $t' \xrightarrow{loc} t''$ and $s'' \preceq t''$. Also, for every t'' such that $t' \xrightarrow{loc} t''$, there is an s'' such that $s' \xrightarrow{loc} s''$ and $s'' \preceq t''$.
3. If $s' \preceq t'$, then for every s'' such that $s' \xrightarrow{ret} s''$, there is a t'' such that $t' \xrightarrow{ret} t''$, and for every t'' such that $t' \xrightarrow{ret} t''$, there is an s'' such that $s' \xrightarrow{ret} s''$. Further, if $s'' \in U_i$ then $t'' \in V_i$, for all i (note that this is not an "iff" condition).

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

Let us now come back to Theorem 6. Consider the two non-bisimilar structured trees $\mathcal{S}$ and $\mathcal{T}$ in Fig. 7-a with initial nodes s_0 and t_0. It is easy to see that the 2-ary VP-μ formula $\varphi = \langle call \rangle (\langle loc \rangle (\langle ret \rangle R_1 \wedge \langle ret \rangle R_2)) \{p, \neg p\}$ distinguishes s_0 and t_0. Let us now see if there is a closed, 1-ary formula φ that can distinguish between $\mathcal{S}$ and $\mathcal{T}$. First, if φ is a disjunction or conjunction, we can get a smaller witness for this distinction. Further, because trees $\mathcal{S}$ and $\mathcal{T}$ are of fixed depth, we need only consider fixpoint-free formulas. The interesting case is that of formulas of the form $\varphi = \langle call \rangle \varphi' \{\psi\}$.

Assume this formula is satisfied by $\langle s_0 \rangle$; then there is a bisimulation-closed summary of the form $\mathbf{s} = \langle s_1, U \rangle$ that satisfies φ'. For each such $\mathbf{s}$, we find a $\mathbf{t} = \langle t_1, V \rangle$. Note that $\mathbf{s}$ can assume only four values; these are $\langle s_1, \{s_9, s_4, s_5\} \rangle$, $\langle s_1, \{s_{10}, s_6, s_7\} \rangle$, $\langle s_1, \{s_9, s_4, s_5, s_{10}, s_6, s_7\} \rangle$, and $\langle s_1, \emptyset \rangle$. The corresponding values of $\mathbf{t}$ are $\langle t_1, \{t_6, t_7\} \rangle$, $\langle t_1, \{t_4, t_5\} \rangle$, $\langle t_1, \{t_4, t_5, t_6, t_7\} \rangle$, and $\langle t_1, \emptyset \rangle$ respectively. Note that for any value $\mathbf{s}$ takes, the corresponding $\mathbf{t}$ is quasi-bisimilar to it, which means that $\mathbf{t}$ satisfies φ'. Further, for each $v \in V$ there is a bisimilar node $u \in U$. It follows that if all $u \in U$ satisfy ψ, then so do all $v \in V$. Then $\langle t_0 \rangle$ satisfies φ.

Similarly one can show that $\langle t_0 \rangle$ satisfies φ only if $\langle s_0 \rangle$ satisfies φ.

To extend the proof to arbitrary k, we consider a structure $\mathcal{S}'$ where, like in $\mathcal{S}$, the root has one *call*-child s_1—except s_1 now has a large number N of *loc*-children s'. From each s', we have $(k+1)$ *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 $\mathcal{T}'$ that is obtained by removing the subtree rooted at node s'_d from $\mathcal{S}'$. Following the methods for the case $k = 1$, we can show that $\mathcal{S}'$ and $\mathcal{T}'$ may be distinguished by a $(k+1)$-ary formula, but by no k-ary formula. We skip the details.

6.3 Satisfiability

The logic VP-μ can express several surprisingly complex properties by exploiting the branching nature of the models. Consider the tree fragment depicted in Figure 7-b.

We have a set of propositions P from which the a_i's and b_i's are drawn from; we also have extra propositions \$ and $*$. It turns out that we can write a VP-μ formula φ whose models consist only of trees that are bisimilar to the tree given in the figure, with the restriction that the word $a_1 \ldots a_n = b_m \ldots b_1$. This is surprising as even a *visibly pushdown word* automaton cannot accept precisely the words encoded along the branch $a_1 \ldots a_n \$ b_1 \ldots b_m$ (it could if the second portion encoding the b's are returns, but not when they are calls). The VP-μ formula can accept this by expressing mainly the following requirements: (a) there are only two *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 VP-μ 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 VP-μ is undecidable (we omit details of the proof):

THEOREM 7. *Given a* VP-μ *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 most known program logics (μ-calculus, temporal logics LTL and CTL, CARET, 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 φ_f such that a node in a structured tree satisfies f if and only if it satisfies φ_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

[1] R. Alur, S. Chaudhuri, and P. Madhusudan. Visibly pushdown tree languages. `http://www.cis.upenn.edu/~swarat/pubs/vptl.ps`.

[2] R. Alur, K. Etessami, and P. Madhusudan. A temporal logic of nested calls and returns. In *10th Int. Conf. on Tools and Algorithms for the Const. and Analysis of Software*, LNCS 2988, pages 467–481, 2004.

[3] R. Alur, K. Etessami, and M. Yannakakis. Analysis of recursive state machines. In *Proc. of the 13th International Conference on Computer Aided Verification*, LNCS 2102, pages 207–220. Springer, 2001.

[4] R. Alur and P. Madhusudan. Visibly pushdown languages. In *Proc. of the 36th STOC*, pages 202–211, 2004.

[5] T. Ball and S. Rajamani. Bebop: A symbolic model checker for boolean programs. In *SPIN 2000 Workshop on Model Checking of Software*, LNCS 1885, pages 113–130. Springer, 2000.

[6] T. Ball and S. Rajamani. The SLAM project: debugging system software via static analysis. In *Proc. of the 29th ACM Symposium on Principles of Programming Languages*, pages 1–3, 2002.

[7] M. Benedikt, P. Godefroid, and T. Reps. Model checking of unrestricted hierarchical state machines. In *28th ICALP*, volume LNCS 2076, pages 652–666. Springer, 2001.

[8] J.R. Burch, E.M. Clarke, D.L. Dill, L.J. Hwang, and K.L. McMillan. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, 1992.

[9] L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G.T. Leavens, R. Leino, and E. Poll. An overview of JML tools and applications. In *Proceedings of the 8th International Workshop on Formal Methods for Industrial Critical Systems*, pages 75–89, 2003.

[10] O. Burkart and B. Steffen. Model checking the full modal mu-calculus for infinite sequential processes. *Theoretical Computer Science*, 221:251–270, 1999.

[11] K. Chatterjee, D. Ma, R. Majumdar, T. Zhao, T.A. Henzinger, and J. Palsberg. Stack size analysis for interrupt driven programs. In *Proceedings of the 10th International Symposium on Static Analysis*, volume LNCS 2694, pages 109–126, 2003.

[12] H. Chen and D. Wagner. Mops: an infrastructure for examining security properties of software. In *Proceedings of ACM Conference on Computer and Communications Security*, pages 235–244, 2002.

[13] E.A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 995–1072. Elsevier Science Publishers, 1990.

[14] E.A. Emerson and C.S. Jutla. Tree automata, mu-calculus, and determinacy. In *Proceedings of the 32nd IEEE Symposium on Foundations of Computer Science*, pages 368–377, 1991.

[15] J. Esparza, A. Kucera, and S. Schwoon. Model-checking LTL with regular valuations for pushdown systems. *Information and Computation*, 186(2):355–376, 2003.

[16] E. Grädel, W. Thomas, and T. Wilke, editors. *Automata, Logics, and Infinite Games: A Guide to Current Research [outcome of a Dagstuhl seminar, February 2001]*, volume 2500 of *Lecture Notes in Computer Science*. Springer, 2002.

[17] T.A. Henzinger, R. Jhala, R. Majumdar, G.C. Necula, G. Sutre, and W. Weimer. Temporal-safety proofs for systems code. In *Proc. of 14th CAV Conference*, LNCS 2404, pp. 526–538, 2002.

[18] D. Janin and I. Walukiewicz. On the expressive completeness of the propositional mu-calculus with respect to monadic second order logic. In *CONCUR'96: Seventh International Conference on Concurrency Theory*, LNCS 1119, pages 263–277. Springer-Verlag, 1996.

[19] T. Jensen, D. Le Metayer, and T. Thorn. Verification of control flow based security properties. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 89–103, 1999.

[20] D. Kozen. Results on the propositional mu-calculus. *Theoretical Computer Science*, 27:333–354, 1983.

[21] K.L. McMillan. *Symbolic model checking: an approach to the state explosion problem*. Kluwer Academic Publishers, 1993.

[22] T. Reps, S. Horwitz, and S. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proc. of the ACM Symposium on Principles of Programming Languages*, pages 49–61, 1995.

[23] D.A. Schmidt. Data flow analysis is model checking of abstract interpretations. In *Proceedings of the 25th Annual ACM Symposium on Principles of Programming Languages*, pages 68–78, 1998.

[24] B. Steffen. Data flow analysis as model checking. In *Theoretical Aspects of Computer Software*, LNCS 526, pages 346–365, 1991.

[25] C.S. Stirling. Modal and temporal logic. In *Handbook of Logic in Computer Science*, pages 477–563. Oxford University Press, 1991.

[26] I. Walukiewicz. Pushdown processes: Games and model-checking. *Information and Computation*, 164(2):234–263, 2001.