Logics and Automata for Software Model-Checking [1]

Rajeev ALUR [a] and Swarat CHAUDHURI [a]
[a] *University of Pennsylvania*

Abstract. While model-checking of pushdown models is by now an established technique in software verification, temporal logics and automata traditionally used in this area are unattractive on two counts. First, logics and automata traditionally used in model-checking cannot express requirements such as pre/post-conditions that are basic to software analysis. Second, unlike in the finite-state world, where the μ-calculus has a symbolic model-checking algorithm and serves as an "assembly language" of temporal logics, there is no unified formalism to model-check linear and branching requirements on pushdown models. In this survey, we discuss a recently-proposed re-phrasing of the model-checking problem for pushdown models that addresses these issues. The key idea is to view a program as a generator of structures known as *nested words* and *nested trees* (respectively in the linear and branching-time cases) as opposed to words and trees. Automata and temporal logics accepting languages of these structures are now defined, and linear and branching time model-checking phrased as language inclusion and membership problems for these languages. We discuss two of these formalisms—automata on nested words and a fixpoint calculus on nested trees—in detail. While these formalisms allow a new frontier of program specifications, their model-checking problem has the same worst-case complexity as their traditional analogs, and can be solved symbolically using a fixpoint computation that generalizes, and includes as a special case, "summary"-based computations traditionally used in interprocedural program analysis.

Keywords. Temporal and fixpoint logics, Automata, Software model-checking, Verification, Program analysis

1. Introduction

Because of concerted research over the last twenty-five years, model-checking of reactive systems is now well-understood theoretically as well as applied in practice. The theories of temporal logics and automata have played a foundational role in this area. For example, in linear-time model-checking, we are interested in questions such as: "do all traces of a protocol satisfy a certain safety property?" This question is phrased language-theoretically as: is the set of all possible system

[1] This research was partially supported by ARO URI award DAAD19-01-1-0473 and NSF award CPA 0541149.

traces included in the language of safe behaviors? In the world of finite-state reactive programs, both these languages are ω-regular [22]. On the other hand, in branching-time model-checking, the specification defines an ω-regular language of trees, and the model-checking problem is to determine if the tree unfolding of the system belongs to this language [17].

Verification of software is a different ball-game. Software written in a modern programming language has many features such as the stack, the heap, and concurrent execution. Reasoning about these features in any automated manner is a challenge—finding ways to model-check them is far harder. The approach that software model-checking takes [10] is that of *data abstraction*: finitely approximate the data in the program, but model the semantics of procedure calls and returns precisely. The chosen abstractions are, thus, *pushdown models* or finite-state machines equipped with a pushdown stack (variants such as *recursive state machines* [1] and boolean programs [9] have also been considered). Such a machine is now viewed as a generator of traces or trees modeling program executions or the program unfolding.

There are, of course, deviations from the classical setting: since pushdown models have unbounded stacks and therefore infinitely many configurations, answering these queries requires *infinite-state model-checking*. Many positive results are known in this area—for instance, model-checking the μ-calculus, often called the "assembly language for temporal logics," is decidable on sequential pushdown models [24,12]. However, many attractive computational properties that hold in the finite-state world are lost. For instance, consider the reachability property: "a state satisfying a proposition p is reachable from the current state," expressible in the μ-calculus by a formula $\varphi = \mu X.(p \vee \langle\rangle X)$. In finite-state model checking, φ not only states a property, but syntactically encodes a symbolic fixpoint computation: start with the states satisfying p, add states that can reach the previous states in one step, then two steps, and so on. This is the reason why hardware model-checkers like SMV translate a specification given in a simpler logic into the μ-calculus, which is now used as a directive for fixpoint computation. Known model-checking algorithms for the μ-calculus on pushdown models, however, are complex and do not follow from the semantics of the formula. In particular, they cannot capture the natural, "summarization"-based fixpoint computations for interprocedural software analysis that have been known for years [19,21].

Another issue with directly applying classical temporal specifications in this context is expressiveness. Traditional logics and automata used in model-checking define *regular* languages of words and trees, and cannot argue about the balanced-parenthesis structure of calls and returns. Suppose we are now interested in *local reachability* rather than reachability: "a state satisfying p is reachable *in the same procedural context* (i.e., before control returns from the current context, and not within the scope of new contexts transitively spawned from this context via calls)." This property cannot be captured by regular languages of words or trees. Other requirements include Hoare-Floyd-style preconditions and postconditions [16] ("if p holds at a procedure call, then q holds on return"), interface contracts used in real-life specification languages such as JML [11] and SAL [14], stack-sensitive access control requirements arising in software security [23], and interprocedural dataflow analysis [18].

While checking pushdown requirements on pushdown models is undecidable in general, individual static analysis techniques are available for all the above applications. There are practical static checkers for interface specification languages and stack inspection-type properties, and interprocedural dataflow analysis [19] can compute dataflow information involving local variables. Their foundations, unfortunately, are not properly understood. What class of languages do these properties correspond to? Can we offer the programmer flexible, decidable temporal logics or automata to write these requirements? These are not merely academic questions. A key practical attraction of model-checking is that a programmer, once offered a temporal specification language, can tailor a program's requirements without getting lost in implementation details. A logic as above would extend this paradigm to interprocedural reasoning. Adding syntactic sugar to it, one could obtain domain-specific applications—for example, one can conceive of a language for module contracts or security policies built on top of such a formalism.

In this paper, we summarize some recent work on software model-checking [4, 6,7,2,3] that offers more insights into these issues by re-phrasing the model-checking problem for sequential pushdown models. In classical linear-time model-checking, the problem is to determine whether the set of linear behaviors of a program abstraction is included in the set of behaviors satisfying the specification. In branching-time model-checking, the question is whether the tree unfolding of the program belongs to the language of trees satisfying the requirement. In other words, a program model is viewed as a generator of a word or tree structure. In the new approach, programs are modeled by pushdown models called *nested state machines*, whose executions and unfoldings are given by graphs called *nested words* and *nested trees*. More precisely, a nested word is obtained by augmenting a word with a set of extra edges, known as *jump-edges*, that connect a position where a call happens to its matching return. As calls and returns in program executions are properly nested, jump-edges never cross. To get a nested tree, we add a set of jump-edges to a tree. As a call may have a number of matching returns along the different paths from it, a node may now have multiple outgoing jump-edges. Temporal logics and finite-state automata accepting languages of nested words and trees are now defined. The linear-time model-checking question then becomes: is the set of *nested words* modeling executions of a program included in the set of nested words accepted by the specification formula/automaton? For branching-time model-checking, we ask: does the *nested tree* generated by a program belong to a specified language of nested trees? It turns out that this tweak makes a major difference computationally as well as in expressiveness.

Let us first see what an automaton on nested words (NWA) [6,7] would look like. Recall that in a finite automaton, the state of the automaton at a position depends on the state and the input symbol at the preceding position. In a nested word, a "return" position has two incoming edges—one from the preceding time point, and one from the "matching call." Accordingly, the state of an NWA may depend on the states of the automaton at both these points. To see how this would help, consider the local reachability property. Consider an automaton following a program execution, and suppose it is in a state q that states that a target state has not yet been seen. Now suppose it encounters a procedure call. A finite automaton on words would follow the execution into the new procedural context and eventually "forget" the current state. However, an NWA can retrieve the state q once the execution returns from this call, and continue to search only the current context. Properties such as this can also be expressed using temporal logics on nested words [4,13], though we will not discuss the latter in detail in this survey.

For branching-time model-checking, we use a fixpoint calculus called NT-μ [2]. The variables of the calculus evaluate not over sets of states, but rather over sets of substructures that capture *summaries* of computations in the "current" program block. The fixpoint operators in the logic then compute fixpoints of summaries. For a node s of a nested tree representing a call, consider the tree rooted at s such that the leaves correspond to exits from the current context. 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 node s represents a "call" to a new context starting with node t, 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. Intuitively, a formula $\langle call \rangle \varphi\{\psi\}$ asserts a constraint φ on the new context, and requires ψ to hold at a designated set of return points of this context. To state local reachability, we would ask, using the formula φ, that control returns to the current context, and, using ψ, that the local reachability property holds at some return point. While this requirement seems self-referential, it may be captured using a fixpoint formula.

It turns out that NWAs and NT-μ have numerous attractive properties. For instance, NWAs have similar closure properties such as regular languages, easily solvable decision problems, Myhill-Nerode-style characterizations, etc. NT-μ can express all properties expressible in the μ-calculus or using NWAs, and has a characterization in terms of *automata on nested trees* [3]. In this survey, we focus more on the computational aspects as applicable to program verification, in particular the model-checking problem for NT-μ. The reason is that NT-μ can capture both linear and branching requirements and, in spite of its expressiveness, can be model-checked efficiently. In fact, the complexity of its model-checking problem on pushdown models (EXPTIME-complete) is the same as that of far weaker logics such as CTL or the alternation-free μ-calculus. Moreover, just as formulas of the μ-calculus syntactically encode a terminating, symbolic fixpoint computation on finite-state systems, formulas of NT-μ describe a directly implementable symbolic model-checking algorithm. In fact, this fixpoint computation generalizes the kind of summary computation traditionally known in interprocedural program analysis, so that, just like the μ-calculus in case of finite-state programs, NT-μ can arguably be used as an "assembly language" for interprocedural computations.

The structure of this paper is as follows. In Sec. 2, we define nested words and trees, and introduce nested state machines as abstractions of structured programs. In Sec. 3, we define specifications on nested structures, studying NWAs and NT-

μ in some detail. In Sec. 4, we discuss in detail the symbolic model-checking algorithm for NT-μ.

2. Models

```
procedure foo()
{
L1:  write(x);
     if(*)
L2:    foo();
     else
L3:    think;
     while (*)
L4:    read(x);
L5:  return;
}
```

A typical approach to software model-checking uses *data abstraction*, where the data in a structured program is abstracted using a finite set of boolean variables that stand for predicates on the data-space [8,15]. The resulting models have finite-state but stack-based control flow. In this section, we define nested state machines, one such model. The behaviors of these machines are modeled by nested words and trees, the structures on which our specifications are interpreted.

As a running example, in the rest of this paper, we use the recursive procedure *foo*. The procedure may read or write a global variable x or perform an action *think*, has nondeterministic choice, and can call itself recursively. Actions of the program are marked by labels L1–L5 for easy reference. We will abstract this program and its behaviors, and subsequently specify it using temporal logics and automata.

2.1. Nested words

Nested words form a class of directed acyclic graphs suitable for abstracting executions of structured programs. In this application, a nested word carries information about a sequence of program states as well as the nesting of procedure calls and returns during the execution. This is done by adding to a word a set of extra edges, known as *jump-edges*, connecting positions where calls happen to their matching returns. Because of the balanced-parentheses semantics of calls and returns, jump-edges are properly nested.

Formally, let Σ be a finite alphabet. Let a finite word w of length n over Σ be a map $w : \{0, 1, \dots, n-1\} \to \Sigma$, and an infinite word be a map $w : \mathbb{N} \to \Sigma$. We sometimes view a word as a graph with positions as nodes, and edges (i, j) connecting successive positions i, j. A *nested word* over Σ is a pair $\mathcal{W} = (w, \hookrightarrow)$, where w is a finite or infinite word over Σ, and $\hookrightarrow \subseteq \mathbb{N} \times (\mathbb{N} \cup \infty)$ is a set of *jump-edges.* A position i in a nested word such that $i \hookrightarrow \infty$ or $i \hookrightarrow j$ for some j is called a *call position*, and a position j such that $i \hookrightarrow j$ for some i is called *return position*). The remaining positions are said to be *local.* The idea is that if a jump-edge (i, j) exists, then position j is the matching return of a call at position i; a jump-edge (i, ∞) implies that there is a call at position i that never returns. The jump-edge relation must satisfy the following conditions:

1. if $i \hookrightarrow j$, then $i < j - 1$ (in other words, a jump-edge is a non-trivial forward jump in the word);
2. for each i, there is at most one $x \in \mathbb{N} \cup \{\infty\}$ such that $i \hookrightarrow x$ or $x \hookrightarrow i$ (a call either never returns or has a unique matching return, and a return has a unique matching call);

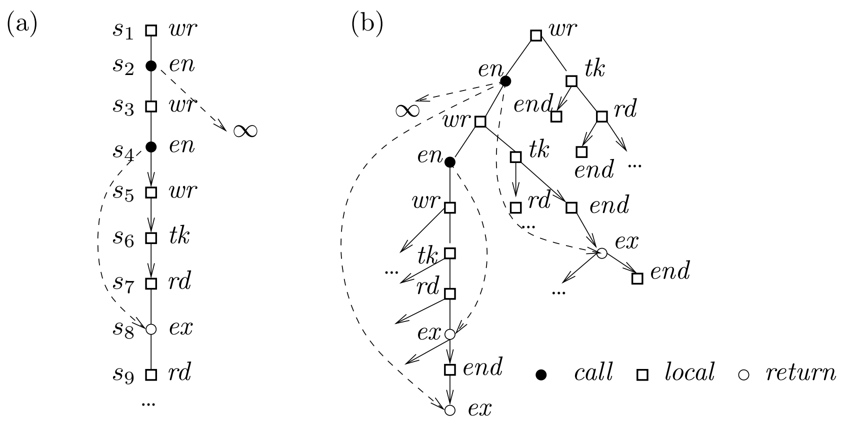

Figure 1. (a) A nested word (b) A nested tree

3. if $i \hookrightarrow j$ and $i' \hookrightarrow j'$ and $i < i'$, then either $j < i'$ or $j' < j$ (jump-edges are properly nested);
4. if $i \hookrightarrow \infty$, then for all calls $i' < i$, either $i' \hookrightarrow \infty$ or $i' \hookrightarrow j$ for some $j < i$ (if a call never returns, neither do the calls that are on the stack when it is invoked).

If $i \hookrightarrow j$, then we call i the *jump-predecessor* of j and j the *jump-successor* of i. Let an edge $(i, i+1)$ in w be called a call and return edge respectively if i is a call and $(i+1)$ is a return, and let all other edges be called *local*.

Let us now turn to our running example. We will model an execution by a nested word over an alphabet Σ. The choice of Σ depends on the desired level of detail—we pick the symbols *wr*, *rd*, *en*, *ex*, *tk*, and *end*, respectively encoding *write(x)*, *read(x)*, a procedure call leading to a beginning of a new context, the return point once a context ends, the statement *think*, and the statement *return*. Now consider the execution where *foo* calls itself twice recursively, then executes *think*, then returns once, then loops infinitely. The word encoding this execution is $w = wr.en.wr.en.wr.tk.rd.ex.(rd)^\omega$. A prefix of the nested word is shown in Fig. 1-(a). The jump-edges are dashed, and call, return and local positions are drawn in different styles. Note the jump-edge capturing the call that never returns.

Now we show a way to encode a nested word using a word. Let us fix a set of *tags* $I = \{call, ret, loc\}$. The *tagged word* $Tag(\mathcal{W})$ of a nested word $\mathcal{W} = (w, \hookrightarrow)$ over Σ is obtained by labeling the edges in w with tags indicating their types. Formally, $Tag(\mathcal{W})$ is a pair (w, η), where η is a map labeling each edge $(i, i+1)$ such that $\eta(i, i+1)$ equals *call* if i is a call, *ret* if $(i+1)$ is a return, and *loc* otherwise. Note that this word is well-defined because jump-edges represent non-trivial forward leaps.

While modeling a program execution, a tagged word defines the sequence of types (call, return or local) of actions in this execution. We note that the construction of $Tag(\mathcal{W})$ requires us to know the jump-edge relation $\hookrightarrow$. More interestingly, the jump-edges in $\mathcal{W}$ are completely captured by the tagged word (w, η) of $\mathcal{W}$, so that we can *reconstruct* a nested word from its tagged word. To see why, call a word $\beta \in I^*$ *balanced* if it is of the form $\beta := \beta\beta \mid call.\beta.ret \mid loc$, and define a relation $\hookrightarrow' \subseteq \mathbb{N} \times \mathbb{N}$ as: for all $i < j-1$, $i \hookrightarrow' j$ iff i is the greatest integer such that the word $\eta(i,i+1).\eta(i+1,i+2)\dots\eta(j-1,j)$ is balanced. It is easily verified that $\hookrightarrow' = \hookrightarrow$.

Let us denote the set of (finite and infinite) nested words over Σ as $NW(\Sigma)$. A language of nested words over Σ is a subset of $NW(\Sigma)$.

2.2. Nested trees

While nested words are suitable for linear-time reasoning, *nested trees* are necessary to specify branching requirements. Such a structure is obtained by adding jump-edges to an infinite tree whose paths encode all possible executions of the program. As for nested words, jump-edges in nested trees do not cross, and calls and returns are defined respectively as sources and targets of jump-edges. In addition, since a procedure call may not return along all possible program paths, a call-node s may have jump-successors along some, but not all, paths from it. If this is the case, we add a jump-edge from s to a special node ∞.

Formally, let $T = (S, r, \rightarrow)$ be an unordered infinite tree with node set S, root r and edge relation $\rightarrow \subseteq S \times S$. Let $\xrightarrow{+}$ denote the transitive (but not reflexive) closure of the edge relation, and let a (finite or infinite) *path* in T from node s_1 be a (finite or infinite) sequence $\pi = s_1 s_2 \dots s_n \dots$ over S, where $n \geq 2$ and $s_i \rightarrow s_{i+1}$ for all $1 \leq i$.

A *nested tree* is a directed acyclic graph $(T, \hookrightarrow)$, where $\hookrightarrow \subseteq T \times (T \cup \infty)$ is a set of jump-edges. A node s such that $s \hookrightarrow t$ or $s \hookrightarrow \infty$ (similarly $t \hookrightarrow s$) for some t is a *call* (*return*) node; the remaining nodes are said to be *local*. The intuition is that if $s \hookrightarrow t$, then a call at s returns at t; if $s \hookrightarrow \infty$, then there exists a path from s along which the call at s never returns. We note that the sets of call, return and local nodes are disjoint. The jump-edges must satisfy:

1. if $s \hookrightarrow t$, then $s \xrightarrow{+} t$, and we do not have $s \rightarrow t$ (in other words, jump-edges represent non-trivial forward jumps);
2. if $s \hookrightarrow t$ and $s \hookrightarrow t'$, then neither $t \xrightarrow{+} t'$ nor $t' \xrightarrow{+} t$ (this captures the intuition that a call-node has at most one matching return along every path from it);
3. if $s \hookrightarrow t$ and $s' \hookrightarrow t$, then $s = s'$ (every return node has a unique matching call);
4. for every call node s, one of the following holds: (a) on every path from s, there is a node t such that $s \hookrightarrow t$, and (b) $s \hookrightarrow \infty$ (a call either returns along all paths, or does not);
5. if there is a path π such that for nodes s, t, s', t' lying on π we have $s \xrightarrow{+} s'$, $s \hookrightarrow t$, and $s' \hookrightarrow t'$, then either $t \xrightarrow{+} s'$ or $t' \xrightarrow{+} t$ (jump-edges along a path do not cross);
6. for every pair of call-nodes s, s' on a path π such that $s \xrightarrow{+} s'$, if there is no node t on π such that $s' \hookrightarrow t$, then a node t' on π can satisfy $s \hookrightarrow t'$ only if $t' \xrightarrow{+} s'$ (if a call does not return, neither do the calls pending when it was invoked).

For an alphabet Σ, a Σ-labeled nested tree is a structure $\mathcal{T} = (T, \hookrightarrow, \lambda)$, where $(T, \hookrightarrow)$ is a nested tree with node set S, and $\lambda : S \to \Sigma$ is a node-labeling function. All nested trees in this paper are Σ-labeled.

Fig. 1-(b) shows a part of the tree unfolding of our example. Note that some of the maximal paths are finite—these capture terminating executions of the program—and some are not. Note in particular how a call may return along some paths from it, and yet not on some others. A path in the nested tree that takes a jump-edge whenever possible is interpreted as a local path through a procedure.

If $s \hookrightarrow t$, then s is the jump-predecessor of t and t the jump-successor of s. Edges from a call node and to a return node are known as *call* and *return* edges; the remaining edges are *local*. The fact that an edge (s, t) exists and is a call, return or local edge is denoted by $s \xrightarrow{call} t$, $s \xrightarrow{ret} t$, or $s \xrightarrow{loc} t$. For a nested tree $\mathcal{T} = (T, \hookrightarrow, \lambda)$ with edge set E, the *tagged tree* of $\mathcal{T}$ is the node and edge-labeled tree $Tag(\mathcal{T}) = (T, \lambda, \eta : E \to \{call, ret, loc\})$, where $\eta(s, t) = a$ iff $s \xrightarrow{a} t$.

A few observations: first, the sets of call, return and local edges define a partition of the set of tree edges. Second, if $s \xrightarrow{ret} s_1$ and $s \xrightarrow{ret} s_2$ for distinct s_1 and s_2, then s_1 and s_2 have the same jump-predecessor. Third, the jump-edges in a nested tree are completely captured by the edge labeling in the corresponding structured tree, so that we can reconstruct a nested tree $\mathcal{T}$ from $Tag(\mathcal{T})$.

Let $NT(\Sigma)$ be the set of Σ-labeled nested trees. A language of nested trees is a subset of $NT(\Sigma)$.

2.3. Nested state machines

Now we define our program abstractions: *nested state machines* (NSMs). Like pushdown system and recursive state machines [1], NSMs are suitable for precisely modeling changes to the program stack due to procedure calls and returns. The main difference is that the semantics of an NSM is defined using nested structures rather than a stack and a configuration graph.

Let AP be a fixed set of atomic propositions, and let us set $\Sigma = 2^{AP}$ as an alphabet of *observables*. A *nested state machine* (NSM) is a tuple $\mathcal{M} = \langle V, v_{in}, \kappa, \Delta_{loc}, \Delta_{call}, \Delta_{ret} \rangle$, where V is a finite set of states, $v_{in} \in V$ is the *initial state*, the map $\kappa : V \to \Sigma$ labels each state with what is observable at it, and $\Delta_{loc} \subseteq V \times V$, $\Delta_{call} \subseteq V \times V$, and $\Delta_{ret} \subseteq V \times V \times V$ are respectively the *local*, *call*, and *return* transition relations.

A transition is said to be *from state* v if it is of the form (v, v') or (v, v', v''), for some $v', v'' \in V$. If $(v, v') \in \Delta_{loc}$ for some $v, v' \in V$, then we write $v \xrightarrow{loc} v'$; if $(v, v') \in \Delta_{call}$, we write $v \xrightarrow{call} v'$; if $(v, v', v'') \in \Delta_{ret}$, we write $(v, v') \xrightarrow{ret} v''$. Intuitively, while modeling a program by an NSM, a transition (v, v') in Δ_{call} models a procedure call that pushes the current state on the stack, and a transition (v, v') in Δ_{loc} models a local action (a move that does not modify the stack). In a return transition (v, v', v''), the states v and v'' are respectively the current and target states, and v' is the state from which the last "unmatched" call-move was made. The intuition is that v' is on top of the stack right before the return-move, which pops it off the stack.

Let us now abstract our example program into a nested state machine $\mathcal{M}_{foo}$. The abstraction simply captures control flow in the program, and consequently, has states v_1, v_2, v_3, v_4, and v_5 corresponding to lines L1, L2, L3, L4, and L5. We also have a state v_2' to which control returns after the call at L2 is completed. Now, let us have propositions *rd*, *wr*, *tk*, *en*, *ex*, and *end* that hold respectively iff the current state represents a read, write, think statement, procedure call, return point after a call, and return instruction. More precisely, $\kappa(v_1) = \{wr\}$, $\kappa(v_2) = \{en\}$, $\kappa(v_2') = \{ex\}$, $\kappa(v_3) = \{tk\}$, $\kappa(v_4) = \{rd\}$, and $\kappa(v_5) = \{end\}$ (for easier reading, we will, from now on, abbreviate singletons such as $\{rd\}$ just as *rd*).

The transition relations of $\mathcal{M}_{foo}$ are given by:

- $\Delta_{call} = \{(v_2, v_1)\}$
- $\Delta_{loc} = \{(v_1, v_2), (v_1, v_3), (v_2', v_4), (v_2', v_5), (v_3, v_4), (v_3, v_5), (v_4, v_4), (v_4, v_5)\}$, and
- $\Delta_{ret} = \{(v_5, v_2, v_2')\}$.

Linear-time semantics The linear-time semantics of a nested state machine $\mathcal{M} = \langle V, v_{in}, \kappa, \Delta_{loc}, \Delta_{call}, \Delta_{ret} \rangle$ is given by a language $\mathcal{L}(\mathcal{M})$ of *traces*; this is a language of nested words over the alphabet 2^{AP}. First consider the language $\mathcal{L}^V(\mathcal{M})$ of *nested executions* of $\mathcal{M}$, comprising nested words over the alphabet V of states. A nested word $\mathcal{W} = (w, \hookrightarrow)$ is in $\mathcal{L}^V(\mathcal{M})$ iff the tagged word (w, η) of $\mathcal{W}$ is such that $w(0) = v_{in}$, and for all $i \geq 0$, (1) if $\eta(i, i+1) \in \{call, loc\}$, then $w(i) \xrightarrow{\eta(i,i+1)} w(i+1)$; and (2) if $\eta(i, i+1) = ret$, then there is a j such that $j \hookrightarrow (i+1)$ and we have $(w(i), w(j)) \xrightarrow{ret} w(i+1)$. Now, a trace produced by an execution is the sequence of observables it corresponds to. Accordingly, the trace language $\mathcal{L}(\mathcal{M})$ of $\mathcal{M}$ is defined as $\{(w', \hookrightarrow) :$ for some $(w, \hookrightarrow) \in \mathcal{L}^V(\mathcal{M})$ and all $i \geq 0$, $w'(i) = \kappa(w_i)\}$. For example, the nested word in Fig. 1-(a) belongs to the trace language of $\mathcal{M}_{foo}$.

Branching-time semantics The branching-time semantics of $\mathcal{M}$ is defined via a 2^{AP}-labeled tree $\mathcal{T}(\mathcal{M})$, known as the *unfolding* of $\mathcal{M}$. For branching-time semantics to be well-defined, an NSM must satisfy an additional condition: every transition from a state v is of the same type (call, return, or local). The idea is to not allow the same node to be a call along one path and, say, a return along another. Note that this is the case in NSMs whose states model statements in programs.

Now consider the V-labeled nested tree $\mathcal{T}^V(\mathcal{M}) = (T, \hookrightarrow, \lambda)$, known as the *execution tree*, that is the unique nested tree satisfying the following conditions:

1. if r is the root of T, then $\lambda(r) = v_{in}$;
2. every node s has precisely one child t for every distinct transition in $\mathcal{M}$ from $\lambda(s)$;
3. for every pair of nodes s and t, if $s \xrightarrow{a} t$, for $a \in \{call, loc\}$, in the tagged tree of this nested tree, then we have $\lambda(s) \xrightarrow{a} \lambda(t)$ in $\mathcal{M}$;
4. for every s, t, if $s \xrightarrow{ret} t$ in the tagged tree, then there is a node t' such that $t' \hookrightarrow t$ and $(\lambda(s), \lambda(t')) \xrightarrow{ret} \lambda(t)$ in $\mathcal{M}$.

Note that this definition is possible as we assume transitions from the same state of $\mathcal{M}$ to be of the same type. Now we have $\mathcal{T}(\mathcal{M}) = (T, \hookrightarrow, \lambda')$, where $\lambda'(s) =$

$\kappa(\lambda(s))$ for all nodes s. For example, the nested tree in Fig. 1-(b) is the unfolding of $\mathcal{M}_{foo}$.

3. Specifications

In this section, we define automata and a fixpoint logic on nested words and trees, and explore their applications to program specification. Automata on nested words are useful for linear-time model-checking, where the question is: "is the language of nested traces of the abstraction (an NSM) included in the language of nested words allowed by the specification?" In the branching-time case, model checking question is: "is the unfolding of the NSM a member of the set of nested trees allowed by the specification?" Our hypothesis is these altered views of the model-checking problem are better suited to software verification.

3.1. Automata on nested words

We start with finite automata on nested words [7,6]. A *nested Büchi word automaton* (NWA) over an alphabet Σ is a tuple $\mathcal{A} = \langle Q, \Sigma, q_{in}, \delta_{loc}, \delta_{call}, \delta_{ret}, G \rangle$, where Q is a set of states Q, q_{in} is the initial state, and $\delta_{loc} \subseteq Q \times \Sigma \times Q$, $\delta_{call} \subseteq Q \times \Sigma \times Q$, and $\delta_{ret} \subseteq Q \times Q \times \Sigma \times Q$ are the local, call and return transition relations. The *Büchi acceptance condition* $G \subseteq Q$ is a set of *accepting states*. If $(q, \sigma, q') \in \delta_{loc}$ for some $q, q' \in Q$ and $\sigma \in \Sigma$, then we write $q \overset{loc,\sigma}{\longrightarrow} q'$; if $(q, \sigma, q') \in \delta_{call}$, we write $q \overset{call,\sigma}{\longrightarrow} q'$; if $(q, q', \sigma, q'') \in \delta_{ret}$, we write $(q, q') \overset{ret,\sigma}{\longrightarrow} q''$.

The automaton $\mathcal{A}$ starts in the initial state, and reads a nested word from left to right. At a call or local position, the current state is determined by the state and the input symbol (in case of traces of NSMs, the observable) at the previous position, while at a return position, the current state can additionally depend on the state of the run just before processing the symbol at the jump-predecessor. Formally, a *run* ρ of the automaton $\mathcal{A}$ over a nested word $\mathcal{W} = (\sigma_1\sigma_2 \ldots, \hookrightarrow)$ is an infinite sequence $q_0, q_1, q_2, \ldots$ over Q such that $q_0 = q_{in}$, and:

- for all $i \geq 0$, if i is a call position of $\mathcal{W}$, then $(q_i, \sigma_i, q_{i+1}) \in \delta_{call}$;
- for all $i \geq 0$, if i is a local position, then $(q_i, \sigma_i, q_{i+1}) \in \delta_{loc}$;
- for $i \geq 2$, if i is a return position with jump-predecessor j, then $(q_{i-1}, q_{j-1}, \sigma_i, q_i) \in \delta_{ret}$.

The automaton $\mathcal{A}$ accepts a finite nested word $\mathcal{W}$ if it has a run $q_0, q_1, q_2, \ldots q_n$ over $\mathcal{W}$ such that $q_n \in G$. An infinite nested word is accepted if there is a run $q_0, q_1, q_2, \ldots$ where a state $q \in G$ is visited infinitely often. The language $\mathcal{L}(\mathcal{A})$ of a nested-word automaton $\mathcal{A}$ is the set of nested words it accepts.

A language L of nested words over Σ is *regular* if there exists a nested-word automaton $\mathcal{A}$ over Σ such that $L = \mathcal{L}(\mathcal{A})$. Observe that if L is a regular language of words over Σ, then $\{(w, \hookrightarrow) \mid w \in L\}$ is a regular language of nested words. Conversely, if L is a regular language of nested words, then $\{w \mid (w, \hookrightarrow) \in L \text{ for some } \hookrightarrow\}$ is a context-free language of words, but need not be regular.

Let us now see an example of how NWAs may be used for specification. Consider the following property to be tested on our running example: "in every execution of the program, every occurrence of *write(x)* is followed (not necessarily immediately) by an occurrence of *read(x)*." This property can be expressed by a finite-state, Büchi *word automaton*. As before, we have $\Sigma = \{wr, rd, en, ex, tk, end\}$. The automaton $\mathcal{S}$ has states q_1 and q_2; the initial state is q_1. The automaton has transitions $q_1 \xrightarrow{wr} q_2$, $q_2 \xrightarrow{wr} q_2$, $q_2 \xrightarrow{rd} q_1$, and $q_1 \xrightarrow{rd} q_1$ (on all input symbols other that wr and rd, $\mathcal{S}$ stays at the state from which the transition fires). The idea is that at the state q_2, $\mathcal{S}$ expects to see a read some time in the future. Now, we have a single Büchi accepting state q_1, which means the automaton cannot get stuck in state q_2, thus accepting precisely the set of traces satisfying our requirement.

However, consider the property: "in every execution of the program, every occurrence of *write(x)* is followed (not necessarily immediately) by an occurrence of *read(x) in the same procedural context* (i.e., before control returns from the current context, and not within the scope of new contexts transitively spawned from this context via calls)." A finite-state word automaton cannot state this requirement, not being able to reason about the balanced-parentheses structure of calls and returns. On the other hand, this property can be expressed simply by a NWA $\mathcal{A}$ with states q_1, q_2 and q_e—here, q_2 is the state where $\mathcal{A}$ expects to see a read action in the same context at some point in the future, q_e is an error state, and q_1 is the state where there is no requirement for the current context. The initial state is q_1. As for transitions:

- we have $q_1 \xrightarrow{loc,wr} q_2$, $q_1 \xrightarrow{loc,rd} q_1$, $q_2 \xrightarrow{loc,rd} q_1$, and $q_2 \xrightarrow{loc,wr} q_2$ (these transitions are for the same reason as in $\mathcal{S}$);
- we have $q_1 \xrightarrow{call,en} q_1$ and $q_2 \xrightarrow{call,en} q_1$ (as the requirement only relates reads and writes in the same context, we need to "reset" the state when a new context starts due to a call);
- for $q' \in \{q_1, q_2\}$, we have $(q_1, q') \xrightarrow{ret,end} q'$ (suppose we have reached the end of a context. So long as there is no requirement pending within this context, we must, on return, restore the state to where it was before the call. Of course, this transition is only fired in contexts that are not at the top-level.) We also have, for $q' \in \{q_1, q_2\}$, $(q_2, q') \xrightarrow{ret,end} q_e$ (in other words, it is an error to end a context before fulfilling a pending requirement).
- Also, for $q' \in \{q_1, q_2\}$, we have $q' \xrightarrow{loc,tk} q'$ and $q' \xrightarrow{loc,ex} q'$.

The single Büchi accepting state, as before, is q_1.

More "realistic" requirements that may be stated using automata on nested words include:

- *Pre/post-conditions:* Consider partial and total correctness requirements based on pre/post-conditions, which show up in Hoare-Floyd-style program verification as well as in modern interface specification languages such JML [11] and SAL [14]. Partial correctness for a procedure `A` asserts that if precondition *Pre* is satisfied when `A` is called, then if `A` terminates, post-condition *Post* holds upon return. Total correctness, additionally, requires `A` to terminate. If program executions are modeled using nested words, these properties are just assertions involving the current state and jump-successors, and can be easily stated using automata on nested words.

- *Access control:* Specifications such as "in all executions of a proggram, a procedure `A` can access a database only if all the frames on the stack have high privilege" are useful in software security and are partially enforced at runtime in programming languages such as Java. Such "stack inspection" properties cannot be stated using traditional temporal logics and automata on words. It can be shown, however, that they are easily stated using nested word languages.
- *Boundedness:* Using nested word languages, we can state requirements such as "the height of the stack is bounded by k along all executions," useful to ensure that there is no stack overflow. Another requirement of this type: "every call in every program execution eventually returns."

We will now list a few properties of regular languages of nested words. The details may be found in the original papers [7,6,5].

- The class of regular languages of nested words is (effectively) closed under union, intersection, complementation, and projection.
- Language membership, inclusion, and emptiness are decidable.
- Automata on finite nested words can be determinized.
- Automata on finite nested words can be characterized using Myhill-Nerode-style congruences, and a subclass of these may be reduced to a unique minimum form.
- Automata on finite or infinite nested words are expressively equivalent to monadic second order logic (MSO) augmented with a binary "jump" predicate capturing the jump-edge relation in nested words. This generalizes the equivalence of regular word languages and the logic S1S.

An alternative way to specify linear-time behaviors of nested executions of programs is to use *temporal logics on nested words.* First consider the logic Caret [4], which may be viewed as an extension of LTL on nested words. Like LTL, this logic has formulas such as $\bigcirc\varphi$ (the property φ holds at the next time point), $\Box\varphi$ (φ holds at every point in the present and future), and $\Diamond\varphi$ (φ holds eventually). The formulas are evaluated as LTL formulas on the word w in a nested word $\mathcal{W} = (w, \hookrightarrow)$. In addition, Caret defines the notion of an "abstract successor" in a nested word—the abstract successor of a call position is its jump-successor, and that of a return or local position is its successor—and has formulas such as $\bigcirc^a\varphi$ (the property φ holds at the abstract successor) and $\Diamond^a\varphi$ (φ holds at some future point in the current context). The full syntax and semantics may be found in the original reference. For a concrete example, consider the property we specified earlier using nested word automata. In Caret, this specification is given by a formula $\varphi = \Box(wr \Rightarrow \Diamond^a rd)$, which is interpreted as "every write is followed (not necessarily immediately) by a read in the same context," and asserted at the initial program state. So far as model-checking goes, every Caret specification may be compiled into an equivalent (and, at worst, exponentially larger) Büchi NWA, so that the model-checking problem for Caret reduces to that for NWAs.

More recently, the linear-time μ-calculus has been extended to nested word models [13]. This logic has modalities $\bigcirc$ and $\bigcirc^a$, which assert requirements respectively at the successor and abstract successor of a position, and, in addition, has set-valued variables x and fixpoint formulas such as $\mu X.\varphi(X)$. We will not go into the details in this paper, but recall that a property "a position satisfying rd is reached eventually" can be stated in the linear-time μ-calculus as $\varphi = \mu X.(rd \vee \bigcirc X)$ (the notation is standard and hence not defined in detail). A property "rd is reached eventually in the current context" is expressed in the linear-time μ-calculus on nested words by the formula $\varphi = \mu X.(rd \vee \bigcirc^a X)$. It turns out that this logic has a number of attractive properties—for example, it is expressively equivalent to MSO-logic interpreted on nested words, and collapses to its alternation-free fragment on finite nested words. Like Caret, formulas in this logic can also be compiled into equivalent NWAs.

3.2. A fixpoint calculus on nested trees

Now we introduce a fixpoint calculus, known as NT-μ, for nested trees [2]. This logic may be viewed as an analog of the modal μ-calculus for nested trees. Recall that a μ-calculus formula is interpreted at a state s of a program, or, equivalently, on the full subtree rooted at a node corresponding to s in the program's tree unfolding. NT-μ is interpreted on substructures of nested trees wholly contained within "procedural" contexts; such a structure models the branching behavior of a program from a state s to each exit point of its context. Also, to demand different temporal requirements at different exits, we introduce a *coloring* of these exits—intuitively, an exit gets color i if it is to satisfy the i-th requirement.

Formally, let a node t of $\mathcal{T}$ be called a *matching exit* of a node s if there is an s' such that $s' \xrightarrow{+} s$ and $s' \hookrightarrow t$, and there are no s'', t'' such that $s' \xrightarrow{+} s'' \xrightarrow{+} s \xrightarrow{+} t''$, and $s'' \hookrightarrow t''$. Intuitively, a matching exit of s is the first "unmatched" return along some path from s—for instance, in Fig. 1-(a), the node s_8 is the single matching exit of the nodes s_5, s_6, and s_7. Let the set of matching exits of s be denoted by $ME(s)$. For a non-negative integer k, a *summary* $\mathbf{s}$ in $\mathcal{T}$ is a tuple $\langle s, U_1, U_2, \ldots, U_k\rangle$, where s is a node, $k \geq 0$, and $U_1, U_2, \ldots, U_k \subseteq ME(s)$ (such a summary is said to be *rooted* at s). The set of summaries in a nested tree $\mathcal{T}$ is denoted by $Summ^{\mathcal{T}}$. Note that such colored summaries are defined for all s, not just "entry" nodes of procedures.

In addition to being interpreted over summaries, the logic NT-μ, can distinguish between call, return and local edges in a nested tree via modalities such as $\langle call\rangle$, $\langle ret\rangle$, and $\langle loc\rangle$. Also, an NT-μ formula can enforce different "return conditions" at differently colored returns by passing subformulas as "parameters" to *call* modalities. Let AP be a finite set of atomic propositions, Var be a finite set of *variables*, and $R_1, R_2, \ldots$ be a countable, ordered set of *markers*. For $p \in AP$, $X \in Var$, and $m \geq 0$, formulas φ of NT-μ are defined by:

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

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. The set of free variables is defined in the usual way. Also, we require our *call*-formulas to bind all the markers in their scope—for example, formulas such as $\varphi = \langle call\rangle(p \wedge \langle ret\rangle R_1)\{q\} \wedge \langle ret\rangle R_1$ are not permitted. A formula that satisfies this criterion is called *closed* if it has no free variables. The *arity* of a for-

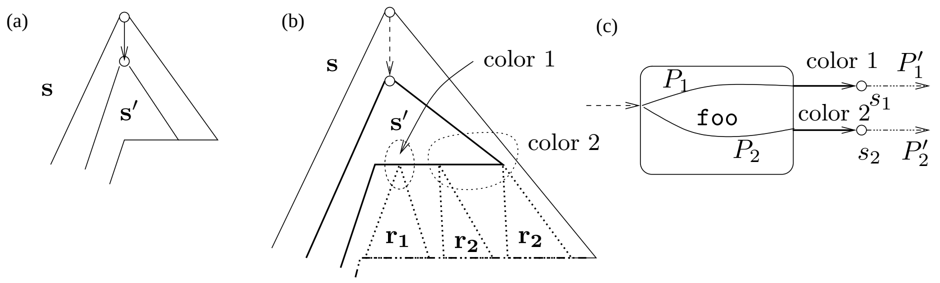

Figure 2. (a) Local modalities (b) Call modalities (c) Matching contexts.

mula φ is the maximum m such that φ has a subformula $\langle call\rangle\varphi'\{\psi_1,\ldots,\psi_m\}$ or $[call]\varphi'\{\psi_1,\ldots,\psi_m\}$. Also, we define the constants *tt* and *ff* in the standard way.

Like in the μ-calculus, formulas in NT-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 (e.g. $\langle loc\rangle$) modalities in NT-μ, consider a node s in a nested tree with a local edge to a node s'. Note that $ME(s') \subseteq ME(s)$, and consider two summaries $\mathbf{s}$ and $\mathbf{s}'$ rooted respectively at s and s'. Now look at Fig. 2-a. Note that the substructure $\mathcal{T}_{s'}$ captured by the summary $\mathbf{s}'$ "hangs"' from the substructure for $\mathbf{s}$ by a local edge; additionally, (1) every leaf of $\mathcal{T}_{s'}$ is a leaf of $\mathcal{T}_s$, and (2) such a leaf gets the same color in $\mathbf{s}$ and $\mathbf{s}'$. A formula $\langle loc\rangle\varphi$ asserted at $\mathbf{s}$ requires some $\mathbf{s}'$ as above to satisfy φ.

Succession along call edges is more complex, because along such an edge, a new context gets defined. Suppose we have $s \xrightarrow{call} s'$, and let there be no other edges from s. Consider the summary $\mathbf{s} = \langle s, \{s_1\}, \{s_2, s_3\}\rangle$, and suppose we want to assert a 2-parameter call formula $\langle call\rangle\varphi'\{p_1,p_2\}$ at s. This requires us to consider a 2-colored summary of the context starting at s', where matching returns of s' satisfying p_1 and p_2 are respectively marked by colors 1 and 2. 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 NT-μ formulas. We find that the above requires a split of the nested tree $\mathcal{T}_s$ 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 $\mathcal{T}_{s'}$ colored i, we must be able to *concatenate* a summary tree $\mathcal{T}_{s''}$ satisfying ψ_i such that (1) every leaf in $\mathcal{T}_{s''}$ is a leaf of $\mathcal{T}_s$, and (2) each such leaf gets the same set of colors in $\mathcal{T}_s$ and $\mathcal{T}_{s''}$.

The return modalities are used to assert that we return at a point colored i. As the binding of these colors to requirements gets fixed at a context calling 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 NT-μ capture the pushdown structure of branching-time runs of a procedural program.

Let us now describe the semantics of NT-μ formally. An NT-μ formula φ is interpreted in an *environment* that interprets variables in $\mathit{Free}(\varphi)$ as sets of summaries in a nested tree $\mathcal{T}$. Formally, an *environment* is a map $\mathcal{E} : \mathit{Free}(\varphi) \rightarrow 2^{\mathit{Summ}^{\mathcal{T}}}$. Let us write $[\![\varphi]\!]^{\mathcal{T}}_{\mathcal{E}}$ to denote the set of summaries in $\mathcal{T}$ satisfying φ in environment $\mathcal{E}$ (usually $\mathcal{T}$ 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 $U_i \subseteq ME(s)$ for all i, $\mathbf{s}$ satisfies φ, i.e., $\mathbf{s} \in [\![\varphi]\!]_{\mathcal{E}}$, iff 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 $\mathbf{s} \in [\![\varphi_2]\!]_{\mathcal{E}}$
- $\varphi = \langle \mathit{call} \rangle \varphi' \{\psi_1, \psi_2, ..., \psi_m\}$, 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 $1 \leq i \leq m$, $V_i = ME(t) \cap \{s' : \langle s', U_1 \cap ME(s'), \ldots, U_k \cap ME(s') \rangle \in [\![\psi_i]\!]_{\mathcal{E}}\}$, is such that $\mathbf{t} \in [\![\varphi']\!]_{\mathcal{E}}$
- $\varphi = [\mathit{call}]\ \varphi' \{\psi_1, \psi_2, ..., \psi_m\}$, 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 $1 \leq i \leq m$, $V_i = ME(t) \cap \{s' : \langle s', U_1 \cap ME(s'), \ldots, U_k \cap ME(s') \rangle \in [\![\psi_i]\!]_{\mathcal{E}}\}$, is such that $\mathbf{t} \in [\![\varphi']\!]_{\mathcal{E}}$
- $\varphi = \langle \mathit{loc} \rangle\ \varphi'$, 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 $V_i = ME(t) \cap U_i$, is such that $\mathbf{t} \in [\![\varphi']\!]_{\mathcal{E}}$
- $\varphi = [\mathit{loc}]\ \varphi'$, 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 $V_i = ME(t) \cap U_i$, is such that $\mathbf{t} \in [\![\varphi']\!]_{\mathcal{E}}$
- $\varphi = \langle \mathit{ret} \rangle\ R_i$, and there is a $t \in S$ such that $s \xrightarrow{ret} t$ and $t \in U_i$
- $\varphi = [\mathit{ret}]\ R_i$, 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 $\mathbf{S} \subseteq \mathit{Summ}^{\mathcal{T}}$ satisfying $[\![\varphi']\!]_{\mathcal{E}[X:=\mathbf{S}]} \subseteq \mathbf{S}$
- $\varphi = \nu X.\varphi'$, and there is some $\mathbf{S} \subseteq \mathit{Summ}^{\mathcal{T}}$ 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 nested tree $\mathcal{T}$ rooted at s_0 is said satisfy φ if s_0 satisfies φ (we denote this by $\mathcal{T} \models \varphi$). The language of φ, denoted by $\mathcal{L}(\varphi)$, is the set of nested trees satisfying φ.

While formulas such as $\neg\varphi$ (negation of φ) are not directly given by the syntax of NT-μ, we can show that *closed* formulas of NT-μ are closed under negation. Also, note that the semantics of closed NT-μ formulas is independent of the environment. Also, the semantics of such a formula φ does not depend on current color assignments; in other words, a summary $\mathbf{s} = \langle s, U_1, \ldots, U_k \rangle$ satisfies a closed formula iff $\langle s \rangle$ satisfies φ. Consequently, when φ is closed, we can infer that "node s satisfies φ" from "summary $\mathbf{s}$ satisfies φ." Finally, every NT-μ formula $\varphi(X)$ with a free variable X can be viewed as a map $\varphi(X) : 2^{\mathit{Summ}^{\mathcal{T}}} \rightarrow 2^{\mathit{Summ}^{\mathcal{T}}}$ defined as follows: for all environments $\mathcal{E}$ and all summary sets $\mathbf{S} \subseteq \mathit{Summ}^{\mathcal{T}}$, $\varphi(X)(\mathbf{S}) = [\![\varphi(X)]\!]_{\mathcal{E}[X:=\mathbf{S}]}$. 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. This means the set of summaries satisfying $\mu X.\varphi(X)$, for instance, lies in the sequence of summary sets $\emptyset, \varphi(\emptyset), \varphi(\varphi(\emptyset)), \ldots$.

Just as the μ-calculus can encode linear-time logics such as LTL as well as branching-time logics such as CTL, NT-μ can capture linear and branching properties on nested trees. Let us now specify our example program using a couple of requirements. Consider the simple property *Reach* asserted at the initial state of the program: "the instruction *read(x)* is reachable from the current node." Let us continue to use the atomic propositions *rd*, *wr*, etc. that we have been using through the paper. This property may be stated in the μ-calculus as $\varphi_{Reach} = (\mu X.rd \vee \langle\rangle X)$ (the notation is standard—for instance, $\langle\rangle\varphi$ holds at a node iff φ holds at a node reached by some edge). However, let us try to define it using NT-μ.

First consider a nontrivial witness π for *Reach* that starts with an edge $s \xrightarrow{call} s'$. There are two possibilities: (1) a node satisfying *rd* 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 *rd* by color 1. Let X store the set of summaries of form $\langle s''\rangle$, where s'' satisfies *Reach*. Then we want the summary $\langle s, ME(s)\rangle$ 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, the formula we want is $\varphi_{Reach} = \mu X.(rd \vee \langle loc\rangle X \vee \langle call\rangle X\{\} \vee \langle call\rangle\varphi'\{X\})$.

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'. It is easy to verify that: $\varphi' = \mu Y.(\langle ret\rangle R_1 \vee \langle loc\rangle Y \vee \langle call\rangle Y\{Y\})$.

Let us now suppose we are interested in *local reachability*: "a node satisfying *rd* is reached in the current context." This property cannot be expressed by finite-state automata on words or trees, and hence cannot be captured by the μ-calculus. However, we note that the property φ' is very similar in spirit to this property. While we cannot merely substitute *rd* for $\langle ret\rangle R_1$ in φ' to express local reachability of *rd*, a formula for this property is easily obtained by restricting the formula for reachability: $\varphi_{LocalReach} = \mu X.(rd \vee \langle loc\rangle X \vee \langle call\rangle\varphi'\{X\})$.

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 NT-μ and the μ-calculus. Consider the fixpoint computation for the μ-calculus formula $\mu X.(rd \vee \langle\rangle X)$ that expresses reachability of a node satisfying *rd*. 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 *rd*. Contrast this with the evaluation of the outer fixpoint in the NT-μ formula φ_{Reach}. 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 φ_{Reach}. 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 rd. However, because of the subformula $\langle call \rangle \varphi' \{X\}$, it also includes all $\mathbf{s}$ where s reaches rd 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.

More details about specification using NT-μ may be found in the original reference [2]. Here we list some other requirements expressible in NT-μ:

- Any closed μ-calculus formula, as well as any property expressible in Caret or automata on nested words, may be expressed in NT-μ. Consequently, NT-μ can express pre/post-conditions on procedures, access control requirements involving the stack, and requirements on the height of the stack, as well as traditional linear and branching-time requirements.
- *Interprocedural dataflow requirements:* It is well-known that many classic dataflow analysis problems, such as determining whether an expression is very busy, can be reduced to the problem of finding the set of program points where a certain μ-calculus property holds [20]. However, the μ-calculus is unable to state that an expression is very busy at a program point if it has local as well as global variables and we are interested in interprocedural paths—the reason is that dataflow involving global variables follows a program execution through procedure calls, while dataflow for local variables "jumps" across procedure calls, and the μ-calculus cannot track them both at the same time. On the other hand, the ability of NT-μ to assert requirements along jump-edges as well as tree edges lets it express such requirements.

We end this discussion by listing some known mathematical properties of NT-μ.

- Generalizing the notion of bisimulation on trees, we may define bisimulation relations on nested trees [2]. Then two nested trees satisfy the same set of closed NT-μ formulas iff they are bisimilar.
- The satisfiability problem for NT-μ is undecidable [2].
- Just as the modal μ-calculus is expressively equivalent to alternating parity tree automata, NT-μ has an automata-theoretic characterization. Generalizing automata on nested words, we can define automata on nested trees; generalizing further, we can define *alternating parity automata on nested trees*. It turns out that every closed formula of NT-μ has a polynomial translation to such an automaton accepting the same set of nested trees, and vice versa [3].

4. Model-checking

In this section, we show how to model-check specifications on nested structures generated by NSMs. Our chosen specification language in this section is the logic NT-μ—the reason is that it can express linear as well as branching-time temporal specifications, and lends itself to iterative, symbolic model-checking. Appealingly, this algorithm follows directly from the operational semantics of the logic and has the same complexity (EXPTIME) as the best algorithms for model-checking CTL or the alternation-free μ-calculus over similar abstractions.

For a specification given by a (closed) NT-μ formula φ and an NSM $\mathcal{M}$ abstracting a program, the *model-checking problem* is to determine if $\mathcal{T}(\mathcal{M})$ satisfies φ. It is also useful to define the model-checking problem for NWAs: here, a problem instance comprises an NSM $\mathcal{M}$ abstracting a program, and an NWA $\mathcal{A}_\neg$ accepting the nested words that model program executions that are *not acceptable*. The model-checking problem in this case is whether any of the possible program traces are "bad", i.e., if $\mathcal{L}(\mathcal{M}) \cap \mathcal{L}(\mathcal{A}_\neg)$ is non-empty. Of course, instead of phrasing the problem this way, we could have also let the instance consist of an NSM and a specification automaton $\mathcal{A}'$, in which case we would have to check if $\mathcal{L}(\mathcal{M}) \cap \overline{\mathcal{L}(\mathcal{A}')}$ is non-empty. However, complementation of $\mathcal{A}'$, while possible, is costly, and this approach would not be practical.

Now, intersection of the languages of two NWAs is done by a product construction [6]. The model-checking problem thus boils down to checking the emptiness of a Büchi NWA $\mathcal{A}$. Let us now view $\mathcal{A}$ as an NSM where a state is marked by a proposition g iff it is a Büchi accepting state. An NWA on infinite nested words is then non-empty iff there are infinitely many occurrences of g along some path in the unfolding of $\mathcal{A}$, a requirement can be expressed as a fixpoint formula in the μ-calculus, and hence NT-μ. To determine that an NWA on finite nested words is non-empty, we merely need to ensure that a node satisfying g is reachable in this unfolding—an NT-μ formula for this property is as in the example in Sec. 3.2.

We will now show how to do NT-μ model-checking for an NSM $\mathcal{M}$ with vertex set V and an NT-μ formula φ. Consider a node s in the nested tree $\mathcal{T}^V(\mathcal{M})$. The set $ME(s)$, as well as the return-formulas that hold at a summary $\mathbf{s}$ rooted at s, depend on states at call nodes on the path from the root to s. However, we observe that the history of call-nodes up to s is relevant to a formula only because they may be consulted by return-nodes in the future, and no formula interpreted at $\mathbf{s}$ can probe "beyond" the nodes in $ME(s)$. Thus, so far as satisfaction of a formula goes, we are only interested in the *last* "pending" call-node; in fact, the state of the automaton at this node is all that we need to record about the past.

Let us now try to formalize this intuition. First we define the *unmatched call-ancestor* $Anc(s)$ of a node s in a nested tree $\mathcal{T}$. Consider the tagged tree of $\mathcal{T}$, and recall the definition of a balanced word over tags (given in Sec. 2.1). If $t = Anc(s)$, then we require that $t \xrightarrow{call} t'$ for some node t' such that in the tagged tree of $\mathcal{T}$, there is a path from t' to s the edge labels along which concatenate to form a balanced word. Note that every node in a nested tree has at most one unmatched call-ancestor. If a node s does not have such an ancestor, we set $Anc(s) = \perp$.

Now consider two k-colored summaries $\mathbf{s} = \langle s, U_1, U_2, \dots, U_k \rangle$ and $\mathbf{s}' = \langle s', U'_1, U'_2, \dots, U'_k \rangle$ in the unfolding $\mathcal{T}^V(\mathcal{M}) = (T, \hookrightarrow, \lambda)$ of the NSM $\mathcal{M}$, and let $Anc(s) = t$ and $Anc(s') = t'$, where t, t' can be nodes or the symbol $\perp$ (note that if we have $Anc(s) = \perp$, then $ME(s) = \emptyset$, so that $U_i = \emptyset$ for all i).

Now we say $\mathbf{s}$ and $\mathbf{s}'$ are *NSM-equivalent* (written as $\mathbf{s} \equiv \mathbf{s}'$) if:

- $\lambda(s) = \lambda(s')$;
- either $t = t' = \perp$, or $\lambda(t) = \lambda(t')$;

- for each $1 \leq i \leq k$, there is a bijection $\Omega_i : U_i \to U'_i$ such that for all $u \in U_i$, we have $\lambda(u) = \lambda(\Omega_i(u))$.

It is easily seen that the relation $\equiv$ is an equivalence. We can also prove that any two NSM-equivalent summaries $\mathbf{s}$ and $\mathbf{s}'$ satisfy the same set of closed NT-μ formulas.

Now note that the number of equivalence classes that $\equiv$ induces on the set of summaries is bounded! Each such equivalence class may be represented by a tuple $\langle v, v', V_1, \dots, V_k \rangle$, where $v \in V$, $v' \in V \cup \{\perp\}$, and $V_i \subseteq V$ for all i—for the class of the summary $\mathbf{s}$ above, for instance, we have $\lambda(s) = v$ and $\lambda(U_i) = V_i$; we also have $\lambda(t) = v'$ in case $t \neq \perp$, and $v' = \perp$ otherwise. Let us call such a tuple a *bounded summary.* The idea behind the model-checking algorithm of NT-μ is that for any formula φ, we can maintain, symbolically, the set of bounded summaries that satisfy it. Once this set is computed, we can compute the set of bounded summaries for formulas defined inductively in terms of φ. This computation follows directly from the semantics of the formula; for instance, the set for the formula $\langle loc \rangle \varphi$ contains all bounded summaries $\langle v, v', V_1, \dots, V_k \rangle$ such that for some $v'' \in V$, we have $v \xrightarrow{loc} v''$, and, letting V''_i comprise the elements of V_i that are reachable from v'', $\langle v'', v', V''_1, \dots, V''_k \rangle$ satisfies φ.

Let us now define bounded summaries formally. Consider any state u in an NSM $\mathcal{M}$ with state set V. A state u' is said to be the *unmatched call-ancestor state* of state u if there is a node s labeled u in $\mathcal{T}^V(\mathcal{M})$ such that u' is the label of the unmatched call-ancestor of s (we have a predicate $Anc_V(u', u)$ that holds iff this is true). Note that a state may have multiple unmatched call-ancestor states. If there is a node s labeled u in $\mathcal{T}^V(\mathcal{M})$ such that $Anc(s) = \perp$, we set $Anc_V(\perp, u)$.

A state v is a *matching exit state* for a pair (u, u'), where $Anc_V(u', u)$, if there are nodes s, s', t in $\mathcal{T}^V(\mathcal{M})$ such that $t \in ME(s)$, s' is the unmatched call-ancestor of s, and labels of s, s', and t are u, u', and v respectively (a pair $(u, \perp)$ has no matching exit state).

The modeling intuition is that from a program state modeled by NSM state u and a stack with a single frame modeled by the state u', control may reach a u'' in the same context, and then return at the state v via a transition $(u'', u') \xrightarrow{ret} v$. Using well-known techniques for pushdown models [1], we can compute, given a state u, the set of u' such that $Anc_V(u', u)$, and for every member u' of the latter, the set $MES(u, u')$ of matching exit states for (u, u'), in time polynomial in the size of $\mathcal{M}$.

Now, let n be the arity of the formula φ in whose model-checking problem we are interested. A *bounded summary* is a tuple $\langle u, u', V_1, \dots, V_k \rangle$, where $0 \leq k \leq n$, $Anc_V(u', u)$ and for all i, we have $V_i \subseteq MES(u, u')$. The set of all bounded summaries in $\mathcal{M}$ is denoted by BS.

Let $\mathcal{E}_{SL} : Free(\varphi) \to 2^{BS}$ be an environment mapping free variables in φ to sets of bounded summaries, and let $\mathcal{E}_\emptyset$ denote the empty environment. We define a map $Eval(\varphi, \mathcal{E}_{SL})$ assigning a set of bounded summaries to a NT-μ formula φ:

- If $\varphi = p$, for $p \in AP$, then $Eval(\varphi, \mathcal{E}_{SL})$ consists of all bounded summaries $\langle u, u', V_1, \dots, V_k \rangle$ such that $p \in \kappa(u)$ and $k \leq n$.
- If $\varphi = \neg p$, for $p \in AP$, then $Eval(\varphi, \mathcal{E}_{SL})$ consists of all bounded summaries $\langle u, u', V_1, V_2, \dots, V_k \rangle$ such that $p \notin \kappa(u)$ and $k \leq n$.

- If $\varphi = X$, for $X \in \mathit{Var}$, then $\mathit{Eval}(\varphi, \mathcal{E}_{SL}) = \mathcal{E}_{SL}(X)$.
- If $\varphi = \varphi_1 \vee \varphi_2$ then $\mathit{Eval}(\varphi, \mathcal{E}_{SL}) = \mathit{Eval}(\varphi_1, \mathcal{E}_{SL}) \cup \mathit{Eval}(\varphi_2, \mathcal{E}_{SL})$.
- If $\varphi = \varphi_1 \wedge \varphi_2$ then $\mathit{Eval}(\varphi, \mathcal{E}_{SL}) = \mathit{Eval}(\varphi_1, \mathcal{E}_{SL}) \cap \mathit{Eval}(\varphi_2, \mathcal{E}_{SL})$.
- If $\varphi = \langle \mathit{call} \rangle\ \varphi'\{\psi_1, ..., \psi_m\}$, then $\mathit{Eval}(\varphi, \mathcal{E}_{SL})$ consists of all bounded summaries $\langle u, u', V_1, \ldots, V_k \rangle$ such that for some transition $u \xrightarrow{call} u''$ of $\mathcal{M}$, we have a bounded summary $\langle u'', u'', V_1', V_2', ..., V_m' \rangle \in \mathit{Eval}(\varphi', \mathcal{E}_{SL})$, and for all $v \in V_i'$, where $i = 1, \ldots, m$, we have $\langle v, u', V_1'', \ldots, V_k'' \rangle \in \mathit{Eval}(\psi_i, \mathcal{E}_{SL})$, where $V_j'' = V_j \cap \mathit{MES}(v, u')$ for all $j \leq k$.
- If $\varphi = [\mathit{call}]\ \varphi'\{\psi_1, ..., \psi_m\}$, then $\mathit{Eval}(\varphi, \mathcal{E}_{SL})$ consists of all bounded summaries $\langle u, u', V_1, \ldots, V_k \rangle$ such that for all u'' such that there is a transition $u \xrightarrow{call} u''$ in $\mathcal{M}$, we have a bounded summary $\langle u'', u'', V_1', V_2', ..., V_m' \rangle \in \mathit{Eval}(\varphi', \mathcal{E}_{SL})$, and for all $v \in V_i'$, where $i = 1, \ldots, m$, we have $\langle v, u', V_1'', \ldots, V_k'' \rangle \in \mathit{Eval}(\psi_i, \mathcal{E}_{SL})$, where $V_j'' = V_j \cap \mathit{MES}(v, u')$ for all $j \leq k$.
- If $\varphi = \langle \mathit{loc} \rangle\ \varphi'$, then $\mathit{Eval}(\varphi, \mathcal{E}_{SL})$ consists of all bounded summaries $\langle u, u', V_1 \ldots, V_k \rangle$ such that for some v such that there is a transition $u \xrightarrow{loc} v$, we have $\langle v, u', V_1 \cap \mathit{MES}(v, u'), \ldots, V_k \cap \mathit{MES}(v, u') \rangle \in \mathit{Eval}(\varphi', \mathcal{E}_{SL})$.
- If $\varphi = \langle \mathit{loc} \rangle\ \varphi'$, then $\mathit{Eval}(\varphi, \mathcal{E}_{SL})$ consists of all bounded summaries $\langle u, u', V_1 \ldots, V_k \rangle$ such that for some v such that there is a transition $u \xrightarrow{loc} v$, we have $\langle v, u', V_1 \cap \mathit{MES}(v, u'), \ldots, V_k \cap \mathit{MES}(v, u') \rangle \in \mathit{Eval}(\varphi', \mathcal{E}_{SL})$.
- If $\varphi = \langle \mathit{ret} \rangle\ R_i$, then $\mathit{Eval}(\varphi, \mathcal{E}_{SL})$ consists of all bounded summaries $\langle u, u', V_1, \ldots, V_k \rangle$ such that (1) $V_i = \{u''\}$, (2) $\mathcal{M}$ has a transition $(u, u') \xrightarrow{ret} u''$, and (3) for all $j \neq i$, $V_j = \emptyset$.
- If $\varphi = \langle \mathit{ret} \rangle\ R_i$, then $\mathit{Eval}(\varphi, \mathcal{E}_{SL})$ consists of all bounded summaries $\langle u, u', V_1, \ldots, V_k \rangle$ such that for all transitions of the form $(u, u') \xrightarrow{ret} u''$, we have (1) $V_i = \{u''\}$, and (2) for all $j \neq i$, $V_j = \emptyset$.
- If $\varphi = \mu X.\varphi'$, then $\mathit{Eval}(\varphi, \mathcal{E}_{SL}) = \mathit{FixPoint}\ (X, \varphi', \mathcal{E}_{SL}[X := \emptyset])$.
- If $\varphi = \nu X.\varphi'$, then $\mathit{Eval}(\varphi, \mathcal{E}_{SL}) = \mathit{FixPoint}\ (X, \varphi', \mathcal{E}_{SL}\ [X := \mathit{BS}])$.

Here $\mathit{FixPoint}\ (X, \varphi, \mathcal{E}_{SL})$ is a fixpoint computation function that uses the formula φ as a monotone map between subsets of BS, and iterates over variable X. This computation is as in Algorithm 1:

Algorithm 1 Calculate $FixPoint\ (X, \varphi, \mathcal{E}_{SL})$

```
$X' \leftarrow \mathit{Eval}(\varphi, \mathcal{E}_{SL})$
if $X' = \mathcal{E}_{SL}(X)$ then
  return $X'$
else
  return $FixPoint\ (X, \varphi', \mathcal{E}_{SL}[X := X'])$
end if
```

Now we can easily show that for an NSM $\mathcal{M}$ with initial state v_{in} and a closed NT-μ formula φ, $\mathcal{T}(\mathcal{M})$ satisfies φ if and only if $\langle v_{in} \rangle \in \mathit{Eval}(\varphi, \mathcal{E}_\emptyset)$, and that $\mathit{Eval}(\varphi, \mathcal{E}_\emptyset)$ is inductively computable. To understand this more concretely, let us see how this model-checking algorithm runs on our running example. Consider the NSM abstraction $\mathcal{M}_{foo}$ in Sec. 2.3, and suppose we want to check if a write action is locally reachable from the initial state. The NT-μ property specifying this requirement is $\varphi = \mu X.(wr \vee \langle loc \rangle X \vee \langle call \rangle \varphi' \{X\})$, where $\varphi' = \mu Y.(\langle ret \rangle R_1 \vee \langle loc \rangle Y \vee \langle call \rangle Y\{Y\})$.

We show how to compute the set of bounded summaries satisfying φ'—the computation for φ is very similar. After the first iteration of the fixpoint computation that builds this set, we obtain the set $\mathbf{S_1} = \{\{\langle v_5, v_2, \{v_2'\}\rangle\}$ (the set of summaries satisfying $\langle ret \rangle R_1$). After the second step, we obtain $\mathbf{S_2} = \mathbf{S_1} \cup \{\langle v_2', v_2, \{v_2'\}\rangle, \langle v_3, v_2, \{v_2'\}\rangle, \langle v_4, v_2, \{v_2'\}\rangle\}$, and the next set computed is $\mathbf{S_3} = \mathbf{S_2} \cup \{\langle v_1, v_2, \{v_2'\}\rangle\}$. Note that in these two steps, we only use local edges in the NSM. Now, however, we have found a bounded summary starting at the "entry state" of the procedure *foo*, which may be plugged into the recursive call to *foo*. More precisely, we have $(v_2, v_1) \in \Delta_{call}$, $\langle v_1, v_2, \{v_2'\}\rangle \in \mathbf{S_3}$, and $\langle v_2', v_2, \{v_2'\}\rangle \in \mathbf{S_3}$, so that we may now construct $\mathbf{S_4} = \mathbf{S_3} \cup \langle v_2, v_2, \{v_2'\}\rangle$. This ends the fixpoint computation, so that $\mathbf{S_4}$ is the set of summaries satisfying φ'.

Let us now analyze the complexity of this algorithm. Let N_V be the number of states in $\mathcal{M}$, and let n be the arity of the formula in question. Then the total number of bounded summaries in $\mathcal{M}$ that we need to consider is bounded by $N = N_V^2 2^{N_V n}$. 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 $n \leq |\varphi|$ is bounded by $O(N^2 |\varphi| N_v)$ (the most expensive modality is $\langle call \rangle \varphi' \{\psi_1, \dots, \psi_n\}$, 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 $Eval(\varphi, \mathcal{E}_\emptyset)$ is $O(N^3 |\varphi| N_V)$. For a formula φ of alternation depth d, this evaluation takes time $O(N^{3d} N_V^d \; |\varphi|)$, i.e., exponential in the sizes of $\mathcal{M}$ as well as φ.

It is known that model-checking alternating reachability specifications on a pushdown model is EXPTIME-hard [24]. It is not hard to generate a NT-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) $\mathcal{M}$ satisfies φ if and only if $\mathcal{M}$ satisfies f. It follows that model-checking a closed NT-μ formula φ on an NSM $\mathcal{M}$ is EXPTIME-hard. Combining, we conclude that model-checking a NT-μ formula φ on an NSM $\mathcal{M}$ is EXPTIME-complete. Better bounds may be obtained if the formula has a certain restricted form. For instance, it can be shown that for linear time (Büchi or reachability) requirements, model-checking takes time polynomial in the number of states of $\mathcal{M}$. The reason is that in this case, it suffices to only consider bounded summaries of the form $\langle v, v', \{v''\}\rangle$, which are polynomial in number. The fixpoint computation stays the same.

Note that our decision procedure is very different from known methods for branching-time model-checking of pushdown models [24,12]. The latter are not really implementable; our algorithm, being symbolic in nature, seems to be a step in the direction of practicality. An open question here is how to represent sets of bounded summaries symbolically. Also, note that our algorithm directly implements the operational semantics of NT-μ formulas over bounded summaries. In this regard NT-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.

References

[1] R. Alur, M. Benedikt, K. Etessami, P. Godefroid, T. Reps, and M. Yannakakis. Analysis of recursive state machines. *ACM Transactions on Programming Languages and Systems*, 27(4):786–818, 2005.

[2] R. Alur, S. Chaudhuri, and P. Madhusudan. A fixpoint calculus for local and global program flows. In *Proceedings of the 33rd Annual ACM Symposium on Principles of Programming Languages*, 2006.

[3] R. Alur, S. Chaudhuri, and P. Madhusudan. Languages of nested trees. In *Computer-Aided Verification, CAV'06*, 2006.

[4] R. Alur, K. Etessami, and P. Madhusudan. A temporal logic of nested calls and returns. In *TACAS'04: Tenth International Conference on Tools and Algorithms for the Construction and Analysis of Software*, LNCS 2988, pages 467–481. Springer, 2004.

[5] R. Alur, V. Kumar, P. Madhusudan, and M. Viswanathan. Congruences for visibly pushdown languages. In *Automata, Languages and Programming: Proceedings of the 32nd ICALP*, LNCS 3580, pages 1102–1114. Springer, 2005.

[6] R. Alur and P. Madhusudan. Visibly pushdown languages. In *Proceedings of the 36th ACM Symposium on Theory of Computing*, pages 202–211, 2004.

[7] R. Alur and P. Madhusudan. Adding nesting structure to words. In *Developments in Language Theory*, 2006.

[8] T. Ball, R. Majumdar, T.D. Millstein, and S.K. Rajamani. Automatic predicate abstraction of C programs. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 203–213, 2001.

[9] 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.

[10] T. Ball and S. Rajamani. The SLAM toolkit. In *Computer Aided Verification, 13th International Conference*, 2001.

[11] 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.

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

[13] H. Comon, M. Dauchet, R. Gilleron, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. Draft, Available at `http://www.grappa.univ-lille3.fr/tata/`, 2002.

[14] B. Hackett, M. Das, D. Wang, and Z. Yang. Modular checking for buffer overflows in the large. In *ICSE*, pages 232–241, 2006.

[15] T.A. Henzinger, R. Jhala, R. Majumdar, G.C. Necula, G. Sutre, and W. Weimer. Temporal-safety proofs for systems code. In *CAV 02: Proc. of 14th Conf. on Computer Aided Verification*, LNCS 2404, pages 526–538. Springer, 2002.

[16] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[17] O. Kupferman, M.Y. Vardi, and P. Wolper. An automata-theoretic approach to branching-time model checking. *Journal of the ACM*, 47(2):312–360, 2000.

[18] T. Reps. Program analysis via graph reachability. *Information and Software Technology*, 40(11-12):701–726, 1998.

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

[20] 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.

[21] M. Sharir and A. Pnueli. *Two approaches to interprocedural data flow analysis*, chapter 7, pages 189–234. Prentice-Hall, Englewood Cliffs, NJ, 1981.

[22] M.Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proceedings of the First IEEE Symposium on Logic in Computer Science*, pages 332–344, 1986.

[23] D. S. Wallach and E. W. Felten. Understanding Java stack inspection. In *IEEE Symp. on Security and Privacy*, pages 52–63, 1998.

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