Software Model Checking Using Languages of Nested Trees

RAJEEV ALUR, University of Pennsylvania
SWARAT CHAUDHURI, Rice University
P. MADHUSUDAN, University of Illinois

While model checking of pushdown systems 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 analysis of software. Second, unlike in the finite-state world, where the μ-calculus has a symbolic model-checking algorithm and serves as an "assembly language" to which temporal logics can be compiled, there is no common formalism—either fixpoint-based or automata-theoretic—to model-check requirements on pushdown models. In this article, we introduce a new theory of temporal logics and automata that addresses the above issues, and provides a unified foundation for the verification of pushdown systems.

The key idea here is to view a program as a generator of structures known as *nested trees* as opposed to trees. A fixpoint logic (called NT-μ) and a class of automata (called *nested tree automata*) interpreted on languages of these structures are now defined, and branching-time model-checking is phrased as language inclusion and membership problems for these languages. We show that NT-μ and nested tree automata allow the specification of a new frontier of requirements usable in software verification. At the same time, 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. We also show that our logics and automata define a robust class of languages—in particular, just as the μ-calculus is equivalent to alternating parity automata on trees, NT-μ is equivalent to alternating parity automata on nested trees.

Categories and Subject Descriptors: D.2.4 [**Software Engineering**]: Software/Program Verification—*Model checking*; F.1.1 [**Computation by abstract devices**]: Models of computation—*Automata*; 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

Additional Key Words and Phrases: Logic, specification, verification, μ-calculus, infinite-state, model-checking, games, pushdown systems, interprocedural analysis

ACM Reference Format:
Alur, R., Chaudhuri, S., and Madhusudan, P. 2011. Software model checking using languages of nested trees. ACM Trans. Program. Lang. Syst. 33, 5, Article 15 (November 2011), 45 pages.
DOI = 10.1145/2039346.2039347 http://doi.acm.org/10.1145/2039346.2039347

This work was partially supported by the National Science Foundation under CAREER award #0953507 and CAREER award #0747041.
Authors' addresses: R. Alur, Computer and Information Science Department, University of Pennsylvania, 3451 Walnut Street, Philadelphia, PA 19104; email: alur@cis.upoenn.edu; S. Chaudhuri, Computer Science Department, Rice University, 6100 Main, Houston, TX 77005; email: swarat@rice.edu; P. Madhusudan, University of Illinois at Urbana-Champaign, 601 East John Street, Champaign, IL 61802; email: madhu@illinois.edu.

DOI 10.1145/2039346.2039347 http://doi.acm.org/10.1145/2039346.2039347

1. INTRODUCTION

Because of concerted research over the last three decades, model checking of finite-state reactive systems is now well understood theoretically as well as applied in practice [Clarke et al. 1999]. The theories of temporal logics and automata have played a foundational role in this area. For example, in branching-time model checking [Emerson and Clarke 1982; Emerson and Lei 1985], the problem is to verify requirements such as: "Every execution of the system must eventually reach a state s such that every state reachable from s satisfies a certain property." These requirements are written formally using finite-state tree automata or temporal logics like the μ-calculus. Due to deep relationships between these formalisms and the theory of tree languages, the verification problem becomes equivalent to the following language-theoretic question: does the tree unfolding of the system belong to an ω-regular language of trees satisfying this requirement? Decision procedures based on fixpoints [Emerson and Lei 1985] or tree automata [Kupferman et al. 2000] are now used to answer this question.

Verification of software, however, is a different story. Unlike in the verification of finite-state systems, now we must handle unbounded data structures, allocated on the heap as well as the stack. The software model checking approach [Ball and Rajamani 2001] to this problem is based on *data abstraction*. Here, the data in the program is abstracted by a finite number of bits, but the semantics of control—loops, branches and procedure calls—are modeled precisely. As most realistic programs have procedural control flow, it is common in this setting to abstract programs using *pushdown models*, or finite-state machines equipped with a pushdown stack (variants such as *recursive state machines* [Alur et al. 2005] and boolean programs [Ball and Rajamani 2000] have also been considered). Such a machine is viewed as a generator of a tree unfolding capturing all executions of the program; the program is correct if this tree belongs to the language of "correct" trees. Model checking is performed as before using language-theoretic decision procedures.

There are, of course, deviations from the classical setting: since pushdown models have 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 [Walukiewicz 2001; Burkart and Steffen 1999]. However, many attractive computational properties that hold in the finite-state world are lost. For instance, consider the property: "A state satisfying a proposition p is reachable from the current state." This property is 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 the structure of the formula. In particular, they cannot capture the natural, "summarization"-based fixpoint computations for interprocedural software analysis that have been known for years [Reps et al. 1995; Sharir and Pnueli 1981].

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 unbounded nesting of calls and returns found in recursive programs. Consider the property of *local 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

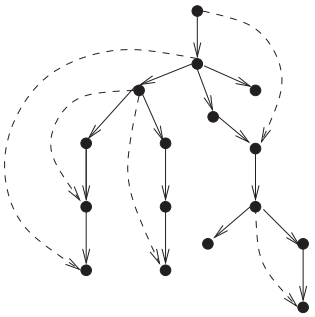

Fig. 1. A nested tree.

by regular languages of words or trees. Other requirements include Floyd-Hoare-style preconditions and postconditions [Hoare 1969] ("if p holds at a procedure call, then q holds on return"), interface contracts used in real-life specification languages such as JML [Burdy et al. 2003], stack-sensitive access control requirements arising in software security [Wallach and Felten 1998], and interprocedural dataflow analysis [Reps 1998].

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 [Reps et al. 1995] can compute dataflow information involving local variables. Less understood is the class of languages to which these properties correspond and the way they relate to each other. Is there a unified logical formalism that can connect all these seemingly disparate dots, extending the model-checking paradigm to properties such as above? Can we offer the programmer a flexible, decidable temporal logic or automaton model 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.

1.1. Our Contributions

In this article, we offer a new theory of logics and automata that forms a unified formal basis for branching-time model checking of procedural programs. (The article consolidates results that we have previously published as conference articles [Alur et al. 2006a, 2006b], and generalizes similar efforts for the simpler linear-time setting [Alur and Madhusudan 2009; 2004; 2006].) Unlike in prior approaches, we do not view the program as the generator of a tree unfolding. Instead, a program is modeled by a pushdown model called a *nested state machine*, whose unfolding is given by a graph called a *nested tree* (Figure 1). This graph is obtained by augmenting the infinite tree unfolding of the program with a set of extra edges, known as *jump-edges*, that connect a node in the tree representing a procedure call to the tree nodes representing the matching returns of the call. As a call may have a number of matching returns along the different paths from it, a node may have multiple outgoing jump-edges. As calls

and returns in executions of a structured program are properly nested, jump-edges never cross.

We develop a theory of regular languages of nested trees through a fixpoint logic and a class of ω-automata interpreted on nested trees. The former is analogous to, and a generalization of, the μ-calculus [Kozen 1983; Grädel et al. 2002] for trees; the latter are generalizations of tree automata. The branching-time model-checking question now becomes: Does the *nested tree* generated by a program belong to the language of nested trees defined by the requirement?

Our fixpoint calculus over nested trees is known as NT-μ. The variables of this 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.

We show that NT-μ can express requirements like local reachability, Hoare-style pre- and postconditions, and stack-sensitive access control properties, which refer to the nested structure of procedure calls and returns and are not expressible in traditional temporal logics. We also show that model checking NT-μ on pushdown models is EXPTIME-complete, and therefore equal in complexity to the problem of model checking the far weaker logic CTL [Walukiewicz 2001] on these abstractions. Like the classical symbolic algorithm for model checking the μ-calculus on finite-state systems, but unlike the far more complex algorithm for μ-calculus model checking on pushdown systems, our algorithm computes symbolic fixpoints in a syntax-directed way (except in this case, the sets used in the fixpoint computation are sets of summaries rather than states). The kind of summary computation traditionally known in interprocedural program analysis is a special case of this algorithm. Thus, just like the μ-calculus in case of finite-state programs, NT-μ can be used as a language into which interprocedural program analyses can be compiled.

As for automata on nested trees, they are a natural generalization of automata on trees. While reading a node in a tree, a tree automaton can nondeterministically pick different combinations of states to be passed along tree edges. In contrast, an automaton on nested trees can send states along tree edges *and jump edges*, so that its state while reading a node depends on the states at its parent and the jump-predecessor (if one exists). As jump-edges connect calls to matching returns, these automata naturally capture the nesting of procedural contexts.

Like tree automata, automata on nested trees come in nondeterministic and alternating flavors, and can accept nested trees by various acceptance conditions. As parity is the most powerful of the acceptance conditions common in ω-automata theory, we mainly focus on two classes of such automata: *nondeterministic parity automata* on nested trees (NPNTAS) and *alternating parity automata* on nested trees (APNTAS). These automata can nondeterministically label a nested tree with states while maintaining

constraints like "If a node is labeled q, then all its tree-children are labeled with the states q_1 and q_2, and all its jump-children are labeled q_2 and q_3" (this is an example of an alternating constraint). We find that, unlike in the setting of tree automata, nondeterministic and alternating automata have different expressive power here, and ApNTAs enjoy more robust mathematical properties. For example, these automata are closed under all Boolean operations. Also, automata-theoretic model checking using ApNTAs is EXPTIME-complete, matching that for alternating tree automata on pushdown models.

In a result analogous to the equivalence between the μ-calculus and alternating parity tree automata, we find that NT-μ has the same expressive power as ApNTAs. This strengthens our belief that NT-μ is not just another fixpoint logic, but captures the essence of regularity in nested trees. Our proof offers polynomial translations from ApNTAs to NT-μ and vice-versa, as well as insights about the connection between runs of ApNTAs and the notion of summaries in NT-μ. This result is especially intriguing as the model checking algorithms for NT-μ and ApNTAs are very different in flavor—while the latter reduces to pushdown games, the former seems to have no connection to the various previously known results about trees, context-free languages, and pushdown graphs. It also helps us compare the expressiveness of NT-μ with that of classical temporal logics and the temporal logic CARET [Alur et al. 2004], which is a linear-time temporal logic for context-sensitive specification. Finally, we show that the satisfiability problem for NT-μ and the emptiness problem of ApNTAs are undecidable—another intriguing difference between languages of nested trees and languages of trees.

1.2. Organization

The structure of this article is as follows. In Section 2, we define nested trees, and introduce nested state machines as abstractions of structured programs. In Section 3, we present the logic NT-μ and show that it is closed under bisimulation; in Section 4, we demonstrate its use in specifying program properties. In Section 5, we discuss in detail our symbolic model-checking algorithm for NT-μ. In Section 6, we introduce automata on nested trees. In Section 7, we study expressiveness results concerning NT-μ and automata on nested trees—in particular, the equivalence of NT-μ and ApNTAs. We conclude with some discussion in Section 8.

2. NESTED TREES

In the formal methods literature, the branching behavior of a nondeterministic program is commonly modeled using infinite trees [Clarke et al. 1999]. The nondeterminism in the program is modeled via tree branching, so that each possible program execution is a path in the tree. Nested trees are obtained by augmenting this tree with an extra edge relation, known as the *jump-edge relation*. A jump-edge connects a tree node representing a procedure call to the node representing the matching return. Thus, a nested tree model of program behavior carries more information about the structure of the program than a tree model.

As calls and returns in structured programs are nested, 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 ∞.

Definition 2.1 (*Nested Tree*). 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 $\stackrel{+}{\longrightarrow}$ 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 $\pi = s_1 s_2 \cdots s_n \cdots$ over S, where $n \geq 2$ and $s_i \rightarrow s_{i+1}$ for all $1 \leq i$.

```
input x;

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

Fig. 2. A sample program.

A *nested tree* is a directed acyclic graph $(T, \hookrightarrow)$, where $\hookrightarrow \subseteq S \times (S \cup \{\infty\})$ 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* (similarly, *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. The jump-edges must satisfy the following properties.

(1) If $s \hookrightarrow t$ or $s \hookrightarrow \infty$, then there is no t such that $t \hookrightarrow s$. In other words, the sets of call and return nodes are disjoint (also, by definition, the set of local nodes is disjoint from both of these sets).
(2) If $s \hookrightarrow t$, then $s \xrightarrow{+} t$, and we do not have $s \rightarrow t$. In other words, jump-edges represent nontrivial forward jumps.
(3) If $s \hookrightarrow t$ and $s \hookrightarrow t'$, then neither $t \xrightarrow{+} t'$ nor $t' \xrightarrow{+} t$. In other words, a call-node has at most one matching return along every path from it.
(4) if $s \hookrightarrow t$ and $s' \hookrightarrow t$, then $s = s'$. In other words, every return node has a unique matching call.
(5) For every call node s, we have either (a) on every path from s, there is a node t such that $s \hookrightarrow t$, or (b) $s \hookrightarrow \infty$. In other words, a call node has a jump-edge to ∞ if there is a path along which the call does not return.
(6) 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 $t' \xrightarrow{+} t$. Intuitively, jump-edges along a path do not cross.
(7) 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 $t' \xrightarrow{+} s'$. Intuitively, if a call does not return, neither do the calls that were pending when it was invoked.

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

We refer to $\rightarrow$ as the *tree-edge* relation. 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 $\lambda : S \rightarrow \Sigma$ is a node-labeling function. All nested trees in this article are Σ-labeled.

Consider the recursive procedure `foo` in Figure 2. The procedure may read or write an expression e or perform an action `think`, has branching dependent on an input variable x, and can call itself recursively. Actions of the program are marked by labels L1–L5

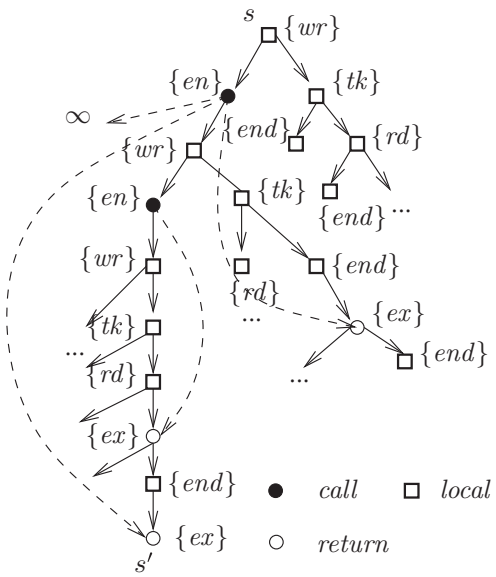

Fig. 3. A nested tree.

for easy reference. We will abstract this program and its behaviors, and subsequently specify it using temporal logics and automata.

Figure 3 shows a part of a nested tree modeling the branching behavior of this program. As the loop and the branch in the procedure depend on an environment-dependent variable, we model them by a nondeterministic loop and a nondeterministic branch. The choice of the alphabet Σ labeling this tree depends on the desired level of detail. We choose it to consist of subsets of a set of atomic propositions AP, comprising the propositions *wr*, *rd*, *en*, *ex*, *tk*, and *end*, respectively encoding a `write` statement, a `read` statement, 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`. A node is labeled by the proposition for a statement if it is the control point from which the statement is executed— for example, the control point immediately preceding a read statement is labeled *rd*. Each path in the underlying tree captures a sequence of program statements—for example, the path fragment starting at the node s and ending at s' captures a (partial) execution that first executes `write`, then calls `foo` recursively, then writes again, then makes another recursive call, ending once it has exited both calls. Note that some of the maximal paths are finite—these capture terminating executions of the program—and some are not. Also, a call may return along some paths from it, and yet not on some others. A path consisting of tree- and jump-edges that takes a jump-edge whenever possible is interpreted as a *local path* through the top-level context.

If $s \hookrightarrow t$, then s is the *jump-predecessor* of t and t the *jump-successor* of s. Let us now consider the set of tree edges. If s is a call node (i.e., if $s \hookrightarrow t$ for some t, or $s \hookrightarrow \infty$), then each tree-edge out of s is called a *call edge*. If s is a return node, then every tree-edge with destination s is a *return edge*. The remaining tree-edges are said to be *local*.

The fact that a tree-edge (s, t) exists and is a call, return or local edge is respectively denoted by $s \xrightarrow{call} t$, $s \xrightarrow{ret} t$, or $s \xrightarrow{loc} t$. Note that given the restrictions we have imposed on jump-edges, the sets of call, return and local edges define a partition of the set of tree-edges. Also, if a node has an outgoing tree-edge labeled *call*, then all its outgoing

tree-edges are labeled *call*. Finally, 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.

The labeling of tree-edges as call, return, or local edges will prove extremely useful to us; in particular, our fixpoint calculus will use the labels *call*, *ret*, and *loc* as modalities. Interestingly, the jump-edges in a nested tree are completely captured by the classification of the tree edges into call, return, local edges. To see why, let us define the *tagged tree* of a nested tree as follows.

Definition 2.2 (*Tagged Tree*). 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 $Struct(\mathcal{T}) = (T, \lambda, \eta : E \rightarrow \{call, ret, loc\})$, where $\eta(s,t) = a$ iff $s \xrightarrow{a} t$.

Now consider *any* (nonnested) tree $T = (S, r, \longrightarrow)$ whose edges are labeled by tags *call*, *ret* and *loc* and that satisfies the constraint: if a node has an outgoing edge labeled *call*, then all its outgoing edges are labeled *call*. Let us call a word $\beta \in I^*$ *balanced* if it is of the form

$$\beta \ := \ call\ \beta'\ ret \qquad\qquad \beta' \ := \ \beta'\beta' \mid call\ \beta'\ ret \mid loc.$$

We define a relation $\hookrightarrow' \subseteq S \times S$ as: for all s, s', we have $s \hookrightarrow' t$ iff

(1) There is a path $s_0 s_1 s_2 \cdots s_n$ such that $s_0 = s$ and $s_n = s'$ in $Struct(\mathcal{T})$.
(2) The word $\eta(s_0, s_1).\eta(s_1, s_2) \cdots \eta(s_{n-1}, s_n)$ is balanced.

Consider the set S_{uc} of nodes s_{uc} such that: (1) outgoing edges from s_{uc} are labeled *call*, and (2) there is at least one path $s_{uc} s_1 s_2 \cdots$ in T such that for no $i \geq 1$ do we have $s_{uc} \hookrightarrow' s_i$. Intuitively, S_{uc} consists of calls that do not return along at least one path. Let us now construct the relation $\hookrightarrow'' = \hookrightarrow' \cup \{(s_{uc}, \infty) : s_{uc} \in S_{uc}\}$. It is easily verified that $(T, \hookrightarrow'')$ is a nested tree, and that if $T = Struct(\mathcal{T})$ for some nested tree $\mathcal{T}$, then $\mathcal{T} = (T, \hookrightarrow'')$. In other words, T is the tagged tree of a unique nested tree, and the latter can be inferred given T.

Ordered, Binary Nested Trees. Note that in the definition of nested trees we have given, the tree structure underlying a nested tree is unordered. While this is the definition we will use as the default definition in this thesis, we will find use for *ordered, binary nested trees* in a few occasions.

Definition 2.3 (*Ordered, Binary Nested Tree*). Let $T = (S, r, \rightarrow_1, \rightarrow_2)$ be an ordered binary tree, where S is a set of nodes, r is the root, and $\rightarrow_1, \rightarrow_2 \subseteq S \times S$ are the left-and right-edge relations. Then $(T, \hookrightarrow)$ is an ordered, binary nested tree if $((S, r, \rightarrow_1 \cup \rightarrow_2), \hookrightarrow)$ is a nested tree by Definition 2.1.

Labeled, ordered nested trees are analogous to labeled, unordered nested trees: for an alphabet Σ, a Σ-labeled ordered nested tree is a structure $\mathcal{T} = (T, \hookrightarrow, \lambda)$, where $(T, \hookrightarrow)$ is a nested tree with node set S, and $\lambda : S \rightarrow \Sigma$ is a node-labeling map.

2.1. Nested State Machines

Now we define a class of abstractions for recursive programs—called *nested state machines*—whose branching-time semantics is defined by nested trees. Like pushdown automata and recursive state machines [Alur et al. 2005], nested state machines (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 a nested tree rather than using a stack.

Syntax. Let AP be a fixed set of atomic propositions; let us fix $\Sigma = 2^{AP}$ as an alphabet of *observables*. We give the following definition.

Definition 2.4 (*Nested State Machine*). A *nested state machine* (NSM) is a structure of the form $\mathcal{M} = \langle V_{loc}, V_{call}, V_{ret}, v_{in}, \kappa, \Delta_{loc}, \Delta_{call}, \Delta_{ret} \rangle$. Here, V_{loc} is a finite set of *local states*, V_{call} a finite set of *call states*, and V_{ret} a finite set of *return states*. We write $V = V_{loc} \cup V_{call} \cup V_{ret}$. The state $v_{in} \in V$ is the *initial state*, and the map $\kappa : V \to \Sigma$ labels each state with an observable. There are three transition relations: a *local transition relation* $\Delta_{loc} \subseteq (V_{loc} \cup V_{ret}) \times (V_{loc} \cup V_{call})$, a *call transition relation* $\Delta_{call} \subseteq V_{call} \times (V_{loc} \cup V_{call})$, and a *return transition relation* $\Delta_{ret} \subseteq (V_{loc} \cup V_{ret}) \times V_{call} \times V_{ret}$.

A transition is said to be *from the 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 call state models a program state from which a procedure call is performed; the call itself is modeled by a call transition in Δ_{call}. A return state of an NSM models a state to which the control returns once a called procedure terminates. The shift of control to a return state is modeled by a return transition (v, v', v'') in Δ_{ret}. Here, 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 when the NSM made a call transition from v', it pushed the state v' on an implicit stack. On return, v' is on top of the stack right before the return-move, which can depend on this state and, on completion, pops it off the stack. This captures the ability of a structured program to use its procedural stack, which is the essence of context-sensitivity. A state that is neither a call nor a return is a local state, and a transition that does not modify the program stack is a local transition.

Let us now abstract our example program (Figure 2) into a nested state machine $\mathcal{M}_{\texttt{foo}}$. 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. The set V_{loc} of local states is $\{v_1, v_3, v_4, v_5\}$, the single call state is v_2, and the single return state is v_2'. The initial state is v_1. Now, let us have propositions *rd*, *wr*, *tk*, *en*, *ex*, and *end* that hold respectively iff the current state represents the control point immediately before a read, a write, a think-statement, a procedure call, a return point after a call, and a 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
—$\Delta_{ret} = \{(v_5, v_2, v_2')\}$.

Branching-Time Semantics. The branching-time semantics of $\mathcal{M}$ is defined via a 2^{AP}-labeled unordered nested tree $\mathcal{T}(\mathcal{M})$, known as the *unfolding* of $\mathcal{M}$. Consider the V-labeled (unordered) 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) for every node s and every distinct call, return or local transition in $\mathcal{M}$ from $\lambda(s)$, s has precisely one outgoing call, return or local tree edge;
(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 $\mathcal{M}$.

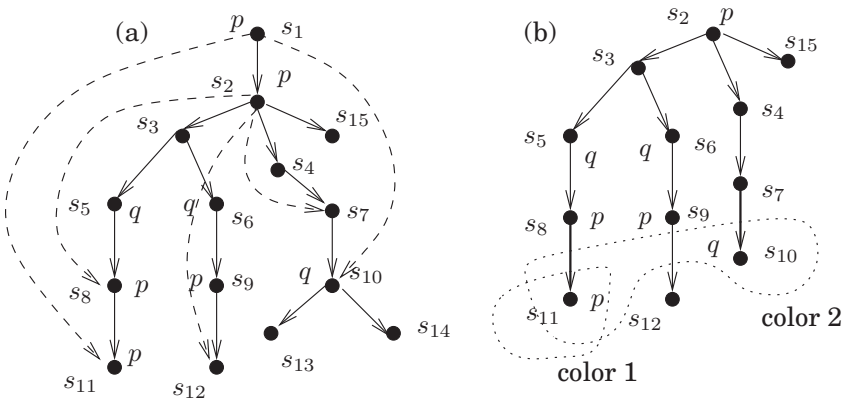

Fig. 4. (a) A nested tree (b) A 2-colored summary.

Note that a node s is a call or return node in this nested tree respectively iff $\lambda(s)$ is a call and return state of $\mathcal{M}$. 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 Figure 3 is the unfolding of $\mathcal{M}_{foo}$.

While unfoldings of nested state machines are most naturally viewed as unordered nested trees, we can also define an NSM's unfolding as an ordered, binary nested tree. In this case, we fix an order on the transitions out of a state and allow at most two outgoing transitions from every state (we can expand the state set to make this possible). The left and right edge relations in the unfolding $\mathcal{T}_{ord}(\mathcal{M})$ respectively correspond to the 1st and 2nd transitions out of a state. We leave out the detailed definition.

3. NT-μ: A FIXPOINT CALCULUS FOR NESTED TREES

In this section, we develop NT-μ, our modal fixpoint calculus interpreted on nested trees. The variables of this logic are evaluated not over sets of states, but over sets of subtrees that capture *summaries* of computations capturing procedural context. The fixpoint operators in the logic then compute fixpoints of summaries. The main technical result is that the logic NT-μ can be model-checked effectively on nested state machine abstractions of software.

3.1. Summaries

Now we define *summaries*, the objects on which our logic is interpreted. These may be viewed as substructures of nested trees capturing procedural contexts; a summary models the branching behavior of a program from a state s to each return point of its context. Also, to capture different temporal obligations to be met on exiting via different exits, we introduce a *coloring* of these exits—intuitively, an exit gets color i if it is to satisfy the ith requirement.

Formally, let a node t of $\mathcal{T}$ be called a *matching exit* of a node s if $s \xrightarrow{+} t$, and 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 $s'' \hookrightarrow t''$. Note that matching exits are defined for all nodes, not just calls. Intuitively, a matching exit of s is the first "unmatched" return along some path from s, for instance, in Figure 4(a), the nodes s_8 and s_{12} are the matching exits of the node s_3, and s_{11} and s_{10} are the matching exits of s_2. Let the set of matching exits of s be denoted by $ME(s)$. Now we define as follows.

Definition 3.1 (*Summary*). For a nonnegative integer k, a k-colored *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).

For example, in the nested tree in Figure 4(a), $\langle s_1 \rangle$ is a 0-colored summary, and $\langle s_2, \{s_{11}\}, \{s_{10}, s_{11}\} \rangle$ and $\langle s_3, \{s_8\}, \emptyset \rangle$ are 2-colored summaries. The set of summaries in a nested tree $\mathcal{T}$, each k-colored for some k, is denoted by $\mathbb{S}$. Note that such colored summaries are defined for all s, not just "entry" nodes of procedures.

Observe how each summary describes a subtree along with a coloring of some of its leaves. For instance, the summary $\mathbf{s} = \langle s_2, \{s_{11}\}, \{s_{10}, s_{11}\} \rangle$ marks the subtree in Figure 4(b). Such a tree may be constructed by taking the subtree of $\mathcal{T}$ rooted at node s_2, and *chopping off* the subtrees rooted at $ME(s_2)$. Note that because of unmatched infinite paths from the root, such a tree may in general be infinite. Now, the node s_{11} is assigned the color 1, and nodes s_{10} and s_{11} are colored 2. Note that the same matching exit might get multiple colors.

It is useful to contrast our definition of summaries with the corresponding definition for the linear-time setting. In this case, a pair (s, s'), where $s' \in ME(s)$, would suffice as a summary— in fact, this is the way in which traditional summarization-based decision procedures have defined summaries. For branching-time reasoning, however, such a simple definition is not enough.

3.2. Syntax

In addition to being interpreted over summaries, the logic NT-μ differs from classical calculi like the modal μ-calculus [Kozen 1983] in a crucial way: its syntax and semantics explicitly recognize the procedural structure of programs. This is done using modalities such as $\langle call \rangle$, $\langle ret \rangle$ and $\langle loc \rangle$ that can distinguish between call, return, and local edges in a nested tree. Also, an NT-μ formula can enforce different "return conditions" at differently colored returns in a summary by passing formulas as "parameters" to *call* modalities. We give the following definition.

Definition 3.2 (*Syntax of* NT-μ). 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 $k \geq 0$, formulas φ of NT-μ are defined by:

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

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 NT-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:

$$\begin{aligned} ind(\varphi_1 \vee \varphi_2) &= ind(\varphi_1 \wedge \varphi_2) = \max\{ind(\varphi_1), ind(\varphi_2)\} \\ ind(\langle loc \rangle \varphi) &= ind([loc]\varphi) = ind(\mu X.\varphi) = ind(\nu X.\varphi) \\ &= ind(\varphi) \\ ind(\langle ret \rangle R_i) &= ind([ret]R_i) = i \\ ind(p) &= ind(X) = 0 \text{ for } p \in AP, X \in Var \end{aligned}$$

$$\begin{aligned} ind(\langle call\rangle\varphi\{\psi_1,\ldots,\psi_k\}) &= ind([call]\varphi\{\psi_1,\ldots,\psi_k\}) \\ &= \max\{ind(\psi_1),\ldots,ind(\psi_k)\}. \end{aligned}$$

We are only interested in formulas where for every subformula $\langle call\rangle\chi'\{\psi_1,\ldots,\psi_k\}$ 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 NT-μ formula φ is defined as:

$$\begin{aligned} Free(\varphi_1 \vee \varphi_2) &= Free(\varphi_1 \wedge \varphi_2) = Free(\varphi_1) \cup Free(\varphi_2) \\ Free(\langle loc\rangle\varphi) &= Free([loc]\varphi) = Free(\varphi) \\ Free(\langle ret\rangle R_i) &= Free([ret]R_i) = \emptyset \\ Free(\langle call\rangle\varphi\{\psi_1,\ldots,\psi_k\}) &= Free([call]\varphi\{\psi_1,\ldots,\psi_k\}) = Free(\varphi) \cup \left(\bigcup_i^k Free(\psi_i)\right) \\ Free(p) &= Free(\neg p) = \emptyset \text{ for } p \in AP \\ Free(X) &= \{X\} \text{ for } X \in Var \\ Free(\mu X.\varphi) &= Free(\nu X.\varphi) = Free(\varphi) \setminus \{X\}. \end{aligned}$$

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.

3.3. Semantics

Like in the modal 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 ($\langle loc\rangle$ and $[loc]$) modalities in NT-μ, consider the 1-colored summary $\mathbf{s} = \langle s_3, \{s_8\}\rangle$ in the tree $\mathcal{T}$ in Figure 4(a). We observe that when control moves from node s_3 to s_5 along a local edge, the current context stays the same, though the set of returns that can end it and are reachable from the current control point can get restricted — that is, $ME(s_5) \subseteq ME(s_3)$. Consequently, the 1-colored summary $\mathbf{s}' = \langle s_5, \{s_8\}\rangle$ describes program flow from s_5 to the end of the current context, and is the *local successor* of the summary $\mathbf{s}$. NT-μ allows us to use modalities $\langle loc\rangle$ and $[loc]$ to assert requirements on such local successors. For instance, in this case, the summary $\mathbf{s}$ will be said to satisfy the formula $\langle loc\rangle q$, as $\mathbf{s}'$ satisfies q.

An interesting visual insight about the structure of the tree $\mathcal{T}_s$ for $\mathbf{s}$ comes from Figure 5(a). Note that the tree $\mathcal{T}_{s'}$ for $\mathbf{s}'$ "hangs"' from the former 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 $\mathbf{s}'$.

Succession along call edges is more complex, because along such an edge, a frame is pushed on a program's stack and a new procedural context gets defined. In Figure 4(a), take the summary $\mathbf{s} = \langle s_1\rangle$, and demand that it satisfies the two-parameter call formula $\langle call\rangle\varphi'\{q, p\}$. This formula asserts a condition on a subtree that: (1) is rooted at a child of s_1, and (2) has colors 1 and 2 assigned respectively to the leaves satisfying p and q. Clearly, a possible such summary is $\mathbf{s}' = \langle s_2, \{s_{10}\}, \{s_{11}\}\rangle$. 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 NT-μ formulas.

Visually, succession along call edges requires a split of the nested tree $\mathcal{T}_s$ for summary $\mathbf{s}$ in the way shown in Figure 5(b). The root of this structure 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 $\mathcal{T}_r$ satisfying ψ_i such that (1) every leaf in $\mathcal{T}_r$ is a leaf of $\mathcal{T}_s$, and (2) each such leaf gets the same set of colors in $\mathcal{T}_s$ and $\mathcal{T}_r$.

As for the return modalities, we use them to assert that we return at a node colored i. Because the binding of these colors to temporal requirements was fixed at a context

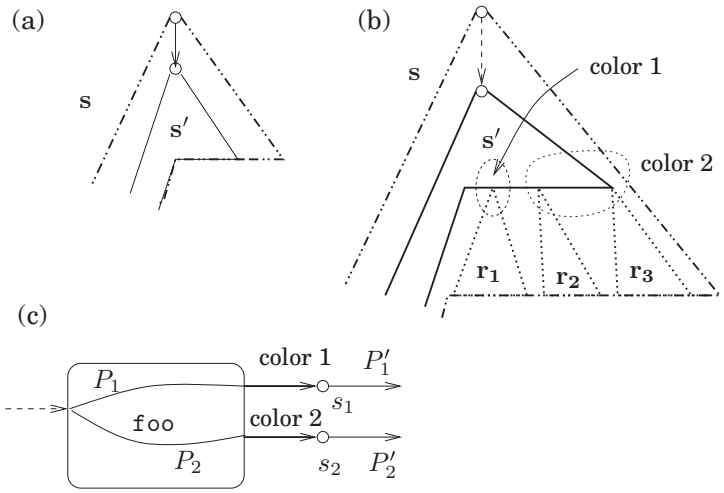

Fig. 5. (a) Local modalities; (b) Call modalities; (c) Matching contexts.

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 Figure 5(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.

Now we define the semantics of NT-μ formally. A NT-μ formula φ is interpreted in an *environment* that interprets variables in $\mathit{Free}(\varphi)$ as sets of summaries in a nested tree $\mathcal{T}$ with node set S. Formally, an *environment* is a map $\mathcal{E} : \mathit{Free}(\varphi) \to 2^{\mathbb{S}}$. Let us write $[\![\varphi]\!]_{\mathcal{E}}^{\mathcal{T}}$ 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 $[\![\varphi]\!]_{\mathcal{E}}$). We give Definition 3.3.

Definition 3.3 (*Semantics of* NT-μ). 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, that is, $\mathbf{s} \in [\![\varphi]\!]_{\mathcal{E}}$, 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 $\mathbf{s} \in [\![\varphi_2]\!]_{\mathcal{E}}$
—$\varphi = \langle call\rangle \varphi'\{\psi_1, \psi_2, \ldots, \psi_m\}$, 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 $1 \le i \le m$, $V_i = ME(t) \cap \{s' : \langle s', U_1 \cap ME(s'), \ldots, U_k \cap ME(s')\rangle \in [\![\psi_i]\!]_{\mathcal{E}}\}$, is such that $\mathbf{t} \in [\![\varphi']\!]_{\mathcal{E}}$
—$\varphi = [call]\ \varphi'\{\psi_1, \psi_2, \ldots, \psi_m\}$, 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 $1 \le i \le m$, $V_i = ME(t) \cap \{s' : \langle s', U_1 \cap ME(s'), \ldots, U_k \cap ME(s')\rangle \in [\![\psi_i]\!]_{\mathcal{E}}\}$, is such that $\mathbf{t} \in [\![\varphi']\!]_{\mathcal{E}}$
—$\varphi = \langle loc\rangle\ \varphi'$, 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 $\mathbf{t} \in [\![\varphi']\!]_{\mathcal{E}}$
—$\varphi = [loc]\ \varphi'$, 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 $\mathbf{t} \in [\![\varphi']\!]_{\mathcal{E}}$

—$\varphi = \langle ret \rangle\ R_i$, 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 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.

A few observations are in order. First, while NT-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 NT-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 $\varphi = \langle call \rangle\ \varphi'\{\psi_1, \psi_2, \ldots, \psi_k\}$, then
$Neg(\varphi) = [call]\ Neg(\varphi')\{Neg(\psi_1), Neg(\psi_2), \ldots, Neg(\psi_k)\}$
—If $\varphi = [call]\ \varphi'\{\psi_1, \psi_2, \ldots, \psi_k\}$, 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 $Neg(\nu X.\varphi) = \mu X.Neg(\varphi)$.

Define the unique *empty environment* as $\perp$: $\emptyset \to \mathbb{S}$. Now we have the following theorem.

THEOREM 3.4. *For all closed* NT-μ *formulas* φ, $[\![\varphi]\!]_\perp = \mathbb{S} \setminus [\![Neg(\varphi)]\!]_\perp$.

PROOF. For an environment $\mathcal{E}$, let $Neg(\mathcal{E})$ be the environment such that for all variables X, $Neg(\mathcal{E})(X) = \mathbb{S} \setminus \mathcal{E}(X)$. Also, for a summary $\mathbf{s} = \langle s, U_1, \ldots, U_k \rangle$, define $\mathit{Flip}(\mathbf{s})$ to be the summary $\langle s, ME(s) \setminus U_1, \ldots, ME(s) \setminus U_k \rangle$. Thus, a leaf is colored i in $\mathit{Flip}(\mathbf{s})$ iff it is not colored i in $\mathbf{s}$. We lift the map Flip to sets of summaries in the natural way.

Now, by induction on the structure of φ, we prove a stronger assertion: for an NT-μ formula φ and an environment $\mathcal{E}$, we have $[\![\varphi]\!]_\mathcal{E} = \mathbb{S} \setminus \mathit{Flip}([\![Neg(\varphi)]\!]_{Neg(\mathcal{E})})$. Note that the theorem follows when we restrict ourselves to variable and marker-closed formulas.

Cases $\varphi = X$, $\varphi = p$ and $\varphi = \neg p$ are trivial; the cases $\varphi = \mu X.\varphi'$ and $\varphi = \nu X.\varphi'$ are easily shown as well. We handle a few other interesting cases.

Suppose $\varphi = \langle ret \rangle R_i$. In this case, $\mathit{Flip}([\![Neg(\varphi)]\!]_{Neg(\mathcal{E})})$ contains the set of summaries $\mathbf{t} = \langle t, U_1, \ldots, U_k \rangle$ such that for all t' satisfying $t \xrightarrow{ret} t'$, we have $t \notin U_i$. It is easy to see that the claim holds.

If $\varphi = \langle call \rangle \varphi'\{\psi_1, \ldots, \psi_k\}$, then $\mathit{Flip}([\![Neg(\varphi)]\!]_{Neg(\mathcal{E})})$ equals the set of summaries $\mathbf{t} = \langle t, U_1, \ldots, U_k \rangle$ such that the following holds: for all t' satisfying $t \xrightarrow{call} t'$, the summary $\mathbf{t}' = \langle t', V_1, V_2, \ldots, V_m \rangle$, where for all $1 \leq i \leq m$, $V_i = ME(t') \cap \{s' : \mathit{Flip}(\langle s', ME(s') \setminus U_1, \ldots, ME(s') \setminus U_k \rangle) \in [\![Neg(\psi_i)]\!]_{Neg(\mathcal{E})}\}$, satisfies $\mathbf{t}' \in [\![Neg(\varphi')]\!]_{Neg(\mathcal{E})}$. Using the induction hypothesis first for the ψ_i-s and then for φ', we can now obtain our claim. □

Note that the semantics of closed NT-μ formulas is *independent* of the environment. Customarily, we will evaluate such formulas in the empty environment $\perp$. 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."

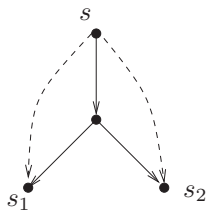

Fig. 6. Negated return conditions.

Third, every NT-μ formula $\varphi(X)$ with a free variable X can be viewed as a map $\varphi(X) : 2^{\mathbb{S}} \to 2^{\mathbb{S}}$ defined as follows: for all environments $\mathcal{E}$ and all summary sets $\mathbf{S} \subseteq \mathbb{S}$, $\varphi(X)(\mathbf{S}) = [\![\varphi(X)]\!]_{\mathcal{E}[X:=\mathbf{S}]}$. Then, we have the following proposition.

PROPOSITION 3.5. *The map $\varphi : 2^{\mathbb{S}} \to 2^{\mathbb{S}}$ is monotonic— that is, if $\mathbf{S} \subseteq \mathbf{S}' \subseteq \mathbb{S}$, then we have $\varphi(\mathbf{S}) \subseteq \varphi(\mathbf{S}')$.*

It is not hard to verify that the map $\varphi(X)$ 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 NT-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 $\mathbb{S}, \varphi(\mathbb{S}), \varphi(\varphi(\mathbb{S})), \ldots$.

Alternately, a NT-μ formula φ may 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 $U_1, U_2, \ldots, U_k \subseteq ME(s)$ 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 NT-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, we have Proposition 3.6.

PROPOSITION 3.6. *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* NT-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 $s' \in V_i$, $\langle s', U_1 \cap ME(s'), U_2 \cap ME(s'), \ldots, U_k \cap ME(s')\rangle \in [\![\psi_i]\!]_\mathcal{E}$. 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, NT-μ and this modified logic have equivalent semantics.

Finally, let us see what would happen if we did allow formulas of form $\langle ret\rangle\neg R_i$, which holds at a summary $\langle s, U_1, \ldots, U_k\rangle$ if and only if there is an edge $s \xrightarrow{ret} t$ such that $t \notin U_i$. In other words, such a formula permits us to state what *must not* hold at a colored matching exit in addition to what must. 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 the nested tree in Figure 6. Let $\mathbf{S_1} = \{\langle s_1\rangle\}$, and $\mathbf{S_2} = \{\langle s_1\rangle, \langle s_2\rangle\}$. Viewing φ as a map $\varphi : 2^{\mathbb{S}} \to 2^{\mathbb{S}}$, we see that: (1) $\varphi(\mathbf{S_2}) = \emptyset$, and (2) $\varphi(\mathbf{S_1}) = \langle s\rangle$.

Thus, even though $\mathbf{S_1} \subseteq \mathbf{S_2}$, we have $\varphi(\mathbf{S_1}) \not\subseteq \varphi(\mathbf{S_2})$. In other words, the monotonicity property breaks down.

3.4. 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 NT-μ, that is, we want to understand when two nodes satisfy the same set of NT-μ formulas.

Consider two nested trees $\mathcal{T}_1$ and $\mathcal{T}_2$ with node sets S_1 and S_2 (we can assume that the sets S_1 and S_2 are disjoint) and node labeling maps λ_1 and λ_2. Let $S = S_1 \cup S_2$ (we can assume that the sets S_1 and S_2 are disjoint), and let λ denote the labeling of S as given by λ_1 and λ_2. Also, we denote by $\mathbb{S}$ the set of all summaries in $\mathcal{T}_1$ and $\mathcal{T}_2$.

Definition 3.7 (*Bisimulation*). The *bisimulation relation* $\sim \subseteq S \times S$ is the greatest relation such that whenever $s \sim t$ holds, we have:

(1) $\lambda(s) = \lambda(t)$,
(2) for $a \in \{call, ret, loc\}$ and for every edge $s \xrightarrow{a} s'$, there is an edge $t \xrightarrow{a} t'$ such that $s' \sim t'$, and
(3) for $a \in \{call, ret, loc\}$ and for every edge $t \xrightarrow{a} t'$, there is an edge $s \xrightarrow{a} s'$ such that $s' \sim t'$.

Let r_1 and r_2 be the roots of $\mathcal{T}_1$ and $\mathcal{T}_2$ respectively. We write $\mathcal{T}_1 \sim \mathcal{T}_2$ if $r_1 \sim r_2$.

NT-μ is interpreted over summaries, so we need to lift the bisimulation relation to summaries. We define this as follows.

Definition 3.8 (*Bisimulation-closed summaries*). A summary $\langle s, U_1, \ldots, U_k\rangle \in \mathbb{S}$ is said to be *bisimulation-closed* if for every pair $u, v \in ME(s)$ of matching exits 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 distinguish 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 $u \in ME(s)$ and $v \in ME(t)$, 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 exits 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 the nested trees $\mathcal{S}$ and $\mathcal{T}$ in Figure 7. 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.

Our goal now is to prove that bisimilar summaries satisfy the same NT-μ formulas. For an inductive proof, we need to consider the environment also. We assume that the environment $\mathcal{E}$ maps NT-μ 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 3.9. *If $\mathcal{E}$ is a bisimulation-closed environment and φ is a* NT-μ *formula, then $[\![\varphi]\!]_{\mathcal{E}}$ is bisimulation-closed.*

PROOF. The proof is by induction on the structure of the formula φ. Consider two bisimulation-closed bisimilar summaries $\mathbf{s} = \langle s, U_1, \ldots, U_k\rangle$ and $\mathbf{t} = \langle t, V_1, \ldots, V_k\rangle$, and

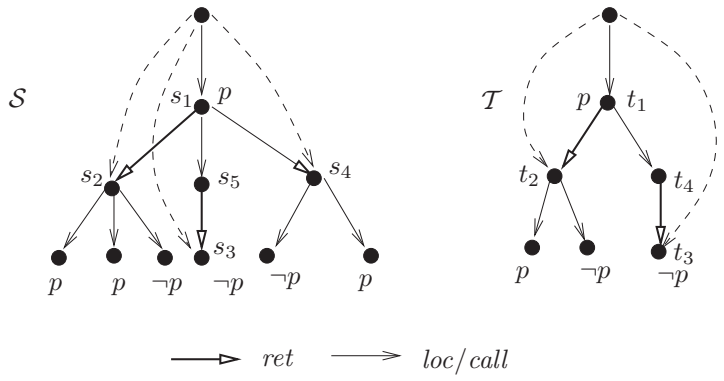

Fig. 7. Bisimilarity.

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

To handle the 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 $[\![\varphi]\!]_{\mathcal{E}}$. □

As a corollary, we get the following.

COROLLARY 3.10. *If $\mathcal{T}_1 \sim \mathcal{T}_2$, then for every closed* NT-μ *formula φ, $\mathcal{T}_1 \models \varphi$ precisely when $\mathcal{T}_2 \models \varphi$.*

The proof also shows that to decide whether a nested tree satisfies a closed NT-μ formula, during the fixpoint evaluation, one can restrict attention only to bisimulation-closed summaries. In other words, we can redefine the semantics of NT-μ so that the set $\mathbb{S}$ of summaries contains only bisimulation-closed summaries. It also suggests that to evaluate a closed NT-μ formula over a nested tree, one can reduce the nested tree by collapsing bisimilar nodes as in the case of classical model checking.

If the two nested trees $\mathcal{T}_1$ and $\mathcal{T}_2$ 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 NT-μ formula that distinguishes the two trees because NT-μ formulas cannot assert requirements across return-edges in a direct way. However,

as we show in Section 7 via an automata-theoretic proof, every closed formula of the μ-calculus may be converted into an equivalent formula in NT-μ. Thus, two nested trees satisfy the same set of closed NT-μ formulas precisely when they are bisimilar.

Let us now consider two arbitrary nodes s and t (in the same nested tree, or in two different nested trees). When do these two nodes satisfy the same set of closed NT-μ formulas? From the arguments so far, bisimilarity is sufficient. However, the satisfaction of a closed NT-μ formula at a node s in a nested tree $\mathcal{T}$ depends solely on the subtree rooted at s that is truncated at the matching exits of s. In fact, the full subtree rooted at s may not be fully contained in a nested tree, as it can contain excess returns. As a result, we define the notion of a *nested subtree* rooted at s as the subgraph obtained by taking the tree rooted at s and deleting the nodes in $ME(s)$ along with the subtrees rooted at them and the return-edges leading to them (the jump-edge relation is restricted in the natural way).

For instance, in Figure 7, $\mathcal{S}_{s_1}$ comprises nodes s_1 and s_5 and the *loc*-edge connecting them. It is easy to check that for a node s in a nested tree $\mathcal{T}$ and a closed NT-μ formula φ, the summary $\langle s \rangle$ satisfies φ in the original nested tree precisely when $\mathcal{T}_s$ satisfies φ. If s and t are not bisimilar, and the non bisimilarity can be established within the nested subtrees $\mathcal{T}_s$ and $\mathcal{T}_t$ rooted at these nodes, then some closed NT-μ formula can distinguish them.

THEOREM 3.11. *Two nodes s and t satisfy the same set of closed* NT-μ *formulas precisely when $\mathcal{T}_s \sim \mathcal{T}_t$.*

4. REQUIREMENT SPECIFICATION USING NT-μ

In this section, we explore how to use NT-μ as a specification language. On one hand, we show how NT-μ and classical temporal logics differ fundamentally in their styles of expression; on the other, we express properties not expressible in logics like the μ-calculus. The example program in Figure 2 (reproduced, along with the corresponding nested tree, in Figure 8) is used to illustrate some of our specifications. As fixpoint formulas are typically hard to read, we define some syntactic sugar for NT-μ using CTL-like temporal operators.

Reachability. Let us express in NT-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 exit s'' of s' is reached, and at s'', *Reach* is once again satisfied.

To deal with case (2), we mark a matching exit 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 $\langle s, ME(s) \rangle$ to satisfy $\langle call \rangle \varphi'\{X\}$, where φ' states that s' can reach one of its matching exits 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 (we write $EF_c\ p$ to denote "p is true before the end of the current context ends"), the formula we want is

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

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

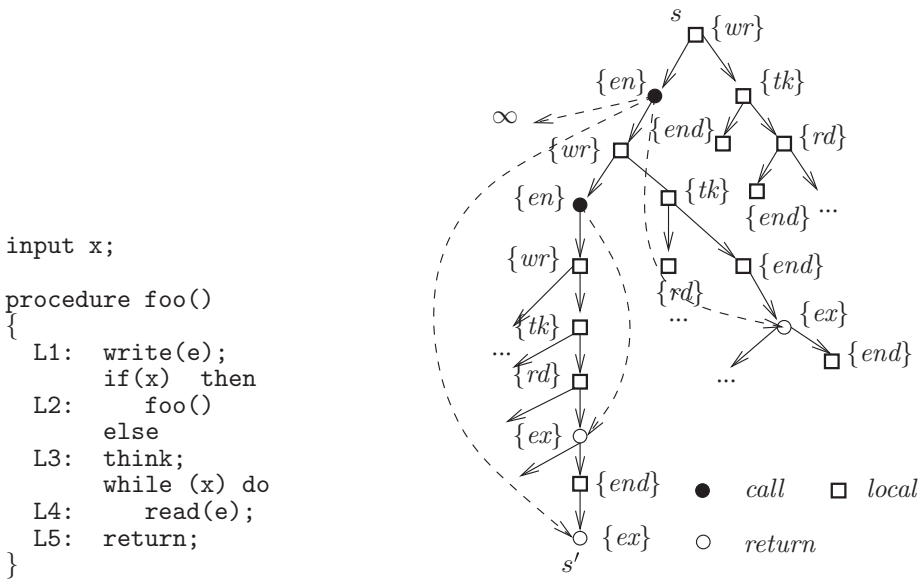

Fig. 8. A program and its nested tree.

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 $\boldsymbol{EF}_c^l\ p$ for this property is easily obtained by restricting the formula $\boldsymbol{EF}_c\ p$:

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

Generalizing, we can allow p to be any NT-μ formula that keeps $\boldsymbol{EF}_c\ p$ and $\boldsymbol{EF}_c^l\ p$ closed.

For example, consider the nested tree in Figure 8 that models the unfolding of the program in the same figure. In that case, $\boldsymbol{EF}_c^l\ rd$ and $\boldsymbol{EF}_c\ wr$ are true at the control point right before the recursive call in L2 in the top-level invocation of *foo* (node s in the figure); however, $\boldsymbol{EF}_c^l wr$ is not.

It is now easy to verify that the formula $AF_c\ p$, which states that "along *all* paths from the current node, a node satisfying p is reached before the current context terminates," is given by

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

where φ'' demands that a matching exit 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_c^l that asserts local reachability along all paths. For instance, in Figure 8, $AF_c^l\ rd$ does *not* hold at node s (as the program can skip its while-loop altogether).

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 $\mu X.(p \vee \langle\rangle X)$ 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 NT-μ formula $EF_c\ p$. 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 $EF_c\ p$. At the end of the ith 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 NT-μ formula $EG_c\ p$ for this is obtained from the identity $EG_c\ p = Neg(AF_c\ Neg(p))$. The formula $AG_c\ p$, 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 $E(p_1\ W_c\ p_2)$) and the *existential until* ($E(p_1\ U_c\ p_2)$) 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 ($A(p_1\ U_c\ p_2)$ and $A(p_1\ W_c\ p_2)$) 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 NT-μ formula $EG^l p$ for *existential local invariance* demands that p holds on some such π', while $AG_c^l p$ asserts the same for all π'. Similarly, we can define existential and universal *local until* properties, and corresponding NT-μ formulas $E(p_1\ U_c^l\ p_2)$ and $A(p_1\ U_c^l\ p_2)$. For instance, in Figure 8, $E(\neg wr\ U_c^l\ rd)$ holds at node s (whereas $E(\neg wr\ U_c\ rd)$ 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

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

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

Interprocedural Dataflow Analysis It is well known that many classic dataflow analysis problems can be reduced to temporal logic model-checking over program abstractions [Steffen 1991; Schmidt 1998]. 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 immediately preceding a statement evaluating e by a proposition $use(e)$, and every node representing a program state about to redefine a variable in e by $mod(e)$. For example, if e is as in the program in Figure 8, every node labeled *wr* in the corresponding nested tree is also labeled $mod(e)$, and every node labeled *rd* is also labeled $use(e)$.

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 NT-μ formula that demands that e is very busy at s is

$$A((EF_c\ use(e) \wedge \neg mod(e))\ W_c\ use(e)).$$

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

However, complications arise if we are considering interprocedural paths and e has local as well as global variables. Note that if e in Figure 8 contains global variables, then it is not very busy at the point right before the recursive call to *foo*. This is because e may be written in the new context. However, if e only contains local variables, then this modification, which happens in an invoked procedural context, does not affect the value of e in the original context. While facts involving global variables and expressions flow through program paths across contexts, data flow involving local variables follow program paths within the same context.

Local temporal properties are useful in capturing these two different types of data flow. Let us handle the general case, where the expression e may have global as well as local variables. 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 NT-μ property we assert at s is

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

where the formula ψ tracks global variables in new contexts:

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

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 NT-μ stands out most clearly from previously studied fixpoint calculi is that of *pushdown specifications*, that is, specifications involving the program stack. We have already introduced a class of such specifications expressible in NT-μ: that of local temporal properties. For instance, the formula $EF_c^l p$ 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 [Alur et al. 2004]. 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 NT-μ formulas for (nonlocal) reachability and invariance. To understand why, recall that NT-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 [Jensen et al. 1999; Esparza et al. 2003], and are enforced at runtime in the Java or .NET stack inspection framework. However, because a program's stack can be unbounded, they 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 NT-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

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

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 NT-μ include "when procedure `foo` is called, all procedures on the stack must have the necessary privilege." Like the previous requirement, this requirement protects a a privileged callee from a malicious caller. However, NT-μ also comes in handy to express properties that protect the caller from the callee. For one such scenario [Abadi and Fournet 2003], consider a malicious method A which, via side-effects or exceptional return conditions, may compromise the security of methods that the caller method B calls subsequently. To prevent such a scenario, we may assert requirements such as "If A has ever been on the stack, do not execute the sensitive operation X." Note that stack inspection cannot handle this specification. This is an example of a *dynamic security constraint* (mentioned but not formalized in Jensen et al. [1999]), which combines reasoning about the program stack with reasoning about the global evolution of the program, allowing privileges of procedures to change dynamically depending on the privileges used so far.

Stack Overflow. Stack overflow, caused by unbounded recursion, is a serious security vulnerability in programs written in C-like languages. NT-μ can specify requirements that safeguard against such errors. Once again, nested modalities come handy. Suppose we assert $AG_c(\langle call\rangle ff\{\})$ 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 Figure 8 will fail this property because of a real recursive cycle. We capture it by asserting AG_c *Termin* at the initial program point, where

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

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 NT-μ formula $\langle jump\rangle p = \langle call\rangle(EF_c^l\ \langle ret\rangle R_1)\{p\}$. 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 [Burdy et al. 2003]. 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 NT-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

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

Total correctness is expressed as

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

Access Control. The ability of NT-μ to handle local and global variables simultaneously is useful in other domains, for example, 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 $\varphi = \neg priv \Rightarrow AG_c\ (\neg access)$ 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 $\varphi' = \neg priv \Rightarrow AG_c^l\ (\neg access)$, asserted at every call site for A.

Multiple Return Conditions. As we shall see in Section 7.2, the theoretical expressiveness of NT-μ depends on the fact that we can pass multiple return conditions as "parameters" to NT-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 8, and observe that in any particular invocation of *foo*, it is possible to exit the routine (1) having read the value of e that was written in this invocation, and (2) not having read this value. Suppose that we demand that in case (2), the expression e must be read at least once before the end of the current context— that is, the value written in the last write must be read by that point. We do not require this in case (1)— in this case control may skip the loop. In addition, let us require that every path in an invocation of *foo* returns and that e is written at least once in this path.

We express these requirements by asserting the NT-μ formula φ at the program point right before the recursive call to *foo*:

$$\varphi = \langle call \rangle \psi'\{tt, \boldsymbol{EF}_c\ rd\},$$

where ψ' is a fixed-point property that states that: each path in the new context must either:

—see a node labeled *wr* followed, not necessarily immediately, by a node labeled *rd*, and then read $\langle ret \rangle R_1$ without seeing *wr* again,
—see a node labeled *wr* and then not see a node labeled *rd* till a node satisfying $\langle ret \rangle R_2$ is reached.

5. MODEL CHECKING

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

For a specification given by a (closed) NT-μ formula φ and an NSM $\mathcal{M}$ abstracting a program (recall Section 2.1), the *model checking problem* is to determine if $\mathcal{T}(\mathcal{M})$ satisfies φ. We will now offer an algorithm for this problem.

Let V be the set of vertices of $\mathcal{M}$, and consider a node s in the execution tree $\mathcal{T}^V(\mathcal{M})$ of $\mathcal{M}$ (defined in Section 2.1). 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 Section 2). 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 π from t' to s such that the sequence of edge labels along π forms 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 let us consider two k-colored summaries $\mathbf{s} = \langle s, U_1, U_2, \ldots, U_k\rangle$ and $\mathbf{s}' = \langle s', U'_1, U'_2, \ldots, U'_k\rangle$ in the execution tree $\mathcal{T}^V(\mathcal{M}) = (T, \hookrightarrow, \lambda)$ 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 *$\mathcal{M}$-equivalent* (written as $\mathbf{s} \equiv_{\mathcal{M}} \mathbf{s}'$ or simply $\mathbf{s} \equiv \mathbf{s}'$ when $\mathcal{M}$ is clear from the context) 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. Let us call a set $\mathbf{S}$ of summaries in $\mathcal{T}^V(\mathcal{M})$ *$\mathcal{M}$-equivalence-closed* (written $\equiv$-closed) if for any two summaries $\mathbf{s}$ and $\mathbf{s}'$ such that $\mathbf{s} \equiv \mathbf{s}'$, we have $\mathbf{s} \in \mathbf{S}$ iff $\mathbf{s}' \in \mathbf{S}$. In other words, such a set is the union of a certain number of equivalence classes induced by $\equiv$. Let us call an environment $\mathcal{E}$ *$\equiv$-closed* if for every variable X, the set $\mathcal{E}(X)$ is $\equiv$-closed.

Note that these definitions involve summaries in the *execution tree* $\mathcal{T}^V(\mathcal{M})$. However, each summary in the execution tree may also be viewed as a summary in the unfolding $\mathcal{T}(\mathcal{M})$ of $\mathcal{M}$, in which case we can interpret NT-μ formulas involving atomic propositions on them.

Now we prove the following.

LEMMA 5.1. *For any $\equiv$-closed environment $\mathcal{E}$,* NT-μ *formula φ, and $\equiv$-equivalent summaries $\mathbf{s}$ and $\mathbf{s}'$, we have $\mathbf{s} \in [\![\varphi]\!]_{\mathcal{E}}^{\mathcal{T}(\mathcal{M})}$ iff $\mathbf{s}' \in [\![\varphi]\!]_{\mathcal{E}}^{\mathcal{T}(\mathcal{M})}$.*

PROOF. The proof is by structural induction on the formula φ. Let $\mathbf{s}$ and $\mathbf{s}'$ be $\mathcal{M}$-equivalent summaries as in the definition of $\mathcal{M}$-equivalence above, and let us continue to denote the bijection between the i-colored exits of $\mathbf{s}$ and $\mathbf{s}'$ by $\Omega_i : U_i \to U'_i$. Let us denote by λ the labeling of nodes in the execution tree $\mathcal{T}^V(\mathcal{M})$. We denote $\lambda(s) = \lambda(s')$ by v_0 and $\lambda(t) = \lambda(t')$ by v_1. It suffices to show that assuming $\mathcal{E}$ is $\equiv$-closed, $\mathbf{s} \in [\![\varphi]\!]_{\mathcal{E}} \Rightarrow \mathbf{s}' \in [\![\varphi]\!]_{\mathcal{E}}$.

We handle some interesting cases.

—If $\varphi = X$, then the claim holds as $\mathcal{E}$ is $\mathcal{M}$-equivalence-closed.
—Suppose $\varphi = \mu X.\varphi'$. Let $X_0 = \emptyset$, and 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 X_0 is $\mathcal{M}$-equivalence-closed, and $\mathcal{E}$ is $\mathcal{M}$-equivalence-closed, and $[\![\varphi']\!]_{\mathcal{E}'}$ is $\mathcal{M}$-equivalence-closed for all $\mathcal{M}$-equivalence-closed environments $\mathcal{E}'$, by induction on i, $[\![\varphi]\!]_{\mathcal{E}}$ is $\mathcal{M}$-equivalence-closed.
—Suppose $\varphi = \langle ret\rangle R_i$. That means that there is some $s \xrightarrow{ret} u$ such that $u \in U_i$. Let $\lambda(u) = v_2$; then $\lambda(\Omega(u)) = v_2$. However, in that case the same transition $(v_0, v_1) \xrightarrow{ret} v_2$ can be "fired" from s' to $\Omega(u)$— that is, $s' \xrightarrow{ret} \Omega(u)$. In other words $\mathbf{s}' \in [\![\varphi]\!]_{\mathcal{E}}$.
—Suppose $\varphi = \langle call\rangle\varphi'\{\psi_1, \ldots, \psi_k\}$. This means that there is some call-edge $s \xrightarrow{call} u$ and a summary $\mathbf{u} = \langle u, V_1, \ldots, V_k\rangle$ satisfying φ' such that for all i and all $w \in V_i$, we have $\mathbf{w} = \langle w, U_1 \cap ME(w), \ldots, U_k \cap ME(w)\rangle$ satisfying ψ_i.

  Now note that, because $\lambda(s) = \lambda(s')$, there is a call-edge $s' \xrightarrow{call} u'$. We can show inductively that there is a bijection $\Omega' : ME(u) \to ME(u')$ such that for any node $w \in ME(u)$, we have $\lambda(w) = \lambda(\Omega'(w))$. The reason is that any path in the tree from s

to $ME(s)$ is independent of the labeling $\lambda(w)$ of any node w such that $w \xrightarrow{+} Anc(s)$ (we can prove this using induction). As $\lambda(s) = \lambda(s')$ and $\lambda(Anc(s)) = \lambda(Anc(s'))$, every path in from s to a node in $ME(s)$ is simulated by a unique path from s' to a node in $ME(s')$.
By restricting Ω' to the sets $V_1, V_2, \ldots \subseteq ME(u)$, we get the bijections $\Omega'_i : V_i \to V'_i$, for $1 \leq i \leq k$. Let us now construct $\mathbf{u}' = \langle u', V'_1, \ldots, V'_k \rangle$. We observe that $\mathbf{u}$ and $\mathbf{u}'$ are $\mathcal{M}$-equivalent. In that case $\mathbf{u}'$ satisfies φ' in environment $\mathcal{E}$. Likewise, for $w \in V_i$ as before, let $w' = \Omega_i(w)$, and construct the summary $\mathbf{w}' = \langle w', V'_1 \cap ME(w'), \ldots, V'_k \cap ME(w') \rangle$. Then, $\mathbf{w}$ and $\mathbf{w}'$ are $\mathcal{M}$-equivalent, and both satisfy ψ_i. In that case, $\mathbf{s}'$ satisfies φ.

The remaining cases are handled similarly, establishing the claim. □

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 $\langle v, v', V_1, \ldots, V_k \rangle$, where $v \in V$, $v' \in V \cup \{\perp\}$, and $V_i \subseteq V$ for all i—for this class of the summary $\mathbf{s}$, 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 $\langle v, v', V_1, \ldots, V_k \rangle$ 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'', $\langle v'', v', V''_1, \ldots, V''_k \rangle$ 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 [Alur et al. 2005], 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 we give definition 5.2.

Definition 5.2 (*Bounded summaries*). Let n be the arity of the formula φ that we seek to model-check. A *bounded summary* is a tuple $\langle u, u', V_1, \ldots, V_k \rangle$, 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 $\mathcal{E}_{BS} : \mathit{Free}(\varphi) \to 2^{BS}$ be a *bounded environment* mapping free variables in φ to sets of bounded summaries, and let $\perp_B$ denote the empty environment. We define a map $Eval(\varphi, \mathcal{E}_{BS})$ assigning a set of bounded summaries to a NT-μ formula φ:

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

FIXPOINT $(X, \varphi, \mathcal{E}_{BS})$
1 $X' = Eval(\varphi, \mathcal{E}_{BS})$
2 **if** $X' = \mathcal{E}_{BS}(X)$
3 **then return** X'
4 **else return** FIXPOINT $(X, \varphi', \mathcal{E}_{BS}[X := X'])$

Fig. 9. Fixpoint computation for NT-μ.

—If $\varphi = \neg p$, for $p \in AP$, then $Eval(\varphi, \mathcal{E}_{BS})$ consists of all bounded summaries $\langle u, u', V_1, V_2, \ldots, V_k\rangle$ such that $p \notin \kappa(u)$ and $k \leq n$.
—If $\varphi = X$, for $X \in Var$, then $Eval(\varphi, \mathcal{E}_{BS}) = \mathcal{E}_{BS}(X)$.
—If $\varphi = \varphi_1 \vee \varphi_2$ then $Eval(\varphi, \mathcal{E}_{BS}) = Eval(\varphi_1, \mathcal{E}_{BS}) \cup Eval(\varphi_2, \mathcal{E}_{BS})$.
—If $\varphi = \varphi_1 \wedge \varphi_2$ then $Eval(\varphi, \mathcal{E}_{BS}) = Eval(\varphi_1, \mathcal{E}_{BS}) \cap Eval(\varphi_2, \mathcal{E}_{BS})$.
—If $\varphi = \langle call\rangle\ \varphi'\{\psi_1, \ldots, \psi_m\}$, then $Eval(\varphi, \mathcal{E}_{BS})$ 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 $\langle u'', u'', V'_1, V'_2, \ldots, V'_m\rangle \in Eval(\varphi', \mathcal{E}_{BS})$, and for all $v \in V'_i$, where $i = 1, \ldots, m$, we have $\langle v, u', V''_1, \ldots, V''_k\rangle \in Eval(\psi_i, \mathcal{E}_{BS})$, where $V''_j = V_j \cap MES(v, u')$ for all $j \leq k$.
—If $\varphi = [call]\ \varphi'\{\psi_1, \ldots, \psi_m\}$, then $Eval(\varphi, \mathcal{E}_{BS})$ 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 $\langle u'', u'', V'_1, V'_2, \ldots, V'_m\rangle \in Eval(\varphi', \mathcal{E}_{BS})$, and for all $v \in V'_i$, where $i = 1, \ldots, m$, we have $\langle v, u', V''_1, \ldots, V''_k\rangle \in Eval(\psi_i, \mathcal{E}_{BS})$, where $V''_j = V_j \cap MES(v, u')$ for all $j \leq k$.
—If $\varphi = \langle loc\rangle\ \varphi'$, then $Eval(\varphi, \mathcal{E}_{BS})$ consists of all bounded summaries of the form $\langle u, u', V_1 \ldots, V_k\rangle$ such that for some v such that there is a transition $u \xrightarrow{loc} v$, we have $\langle v, u', V_1 \cap MES(v, u'), \ldots, V_k \cap MES(v, u')\rangle \in Eval(\varphi', \mathcal{E}_{BS})$.
—If $\varphi = [loc]\ \varphi'$, then $Eval(\varphi, \mathcal{E}_{BS})$ consists of all bounded summaries of the form $\langle u, u', V_1 \ldots, V_k\rangle$ such that for all v such that there is a transition $u \xrightarrow{loc} v$, we have $\langle v, u', V_1 \cap MES(v, u'), \ldots, V_k \cap MES(v, u')\rangle \in Eval(\varphi', \mathcal{E}_{BS})$.
—If $\varphi = \langle ret\rangle\ R_i$, then $Eval(\varphi, \mathcal{E}_{BS})$ consists of all bounded summaries of the form $\langle u, u', V_1, \ldots, V_k\rangle$ such that for some $u'' \in V_i$, $\mathcal{M}$ has a transition $(u, u') \xrightarrow{ret} u''$.
—If $\varphi = [ret]\ R_i$, then $Eval(\varphi, \mathcal{E}_{BS})$ consists of all bounded summaries of the form $\langle u, u', V_1, \ldots, V_k\rangle$ such that for all transitions of the form $(u, u') \xrightarrow{ret} u''$, we have $u'' \in V_i$.
—If $\varphi = \mu X.\varphi'$, then $Eval(\varphi, \mathcal{E}_{BS}) = FixPoint\ (X, \varphi', \mathcal{E}_{BS}[X := \emptyset])$.
—If $\varphi = \nu X.\varphi'$, then $Eval(\varphi, \mathcal{E}_{BS}) = FixPoint\ (X, \varphi', \mathcal{E}_{BS}\ [X := BS])$.

Here $FixPoint\ (X, \varphi, \mathcal{E}_{BS})$ 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 Figure 9.

Now we show that to model check a formula φ on an NSM $\mathcal{M}$, we only have to compute $Eval(\varphi, \perp_B)$. Let us first define, for any bounded summary $\mathbf{b} = \langle v, v', V_1, \ldots, V_k\rangle$, the set $Unbound(\mathbf{b})$ of summaries in $\mathcal{T}^V(\mathcal{M})$ that it captures. Let λ be denote the node-labeling map in $\mathcal{T}^V(\mathcal{M})$. A summary $\mathbf{s} = \langle s, U_1, \ldots, U_k\rangle$ is in $Unbound(\mathbf{b})$ iff $\lambda(s) = v$, $\lambda(Anc(s)) = v'$, and for all $1 \leq i \leq k$, we have $\lambda(U_i) = V_i$.

The map is lifted to sets of bounded summaries in the natural way. We also lift the map $Unbound$ to bounded environments. For a bounded environment $\mathcal{E}_{BS}$, the set $Unbound(\mathcal{E}_{BS})$ is the environment $\mathcal{E}$ in $\mathcal{T}^V(\mathcal{M})$ such that for every variable X, we have $\mathcal{E}(X) = Unbound(\mathcal{E}_{BS}(X))$.

Now observe that:

(1) for any bounded summary $\mathbf{b}$, the set $Unbound(\mathbf{b})$ is $\equiv_{\mathcal{M}}$-closed.
(2) for any bounded environment $\mathcal{E}_{BS}$, the environment $Unbound(\mathcal{E}_{BS})$ is $\equiv$-closed.

Next we show inductively that:

LEMMA 5.3. *For any* NT-μ *formula* φ*, bounded environment* $\mathcal{E}_{BS}$*, and bounded summary* $\mathbf{b}$*, and for all* $\mathbf{s} \in Unbound(\mathbf{b})$*, we have*

$$\mathbf{b} \in Eval(\varphi, \mathcal{E}_{BS}) \text{ iff } \mathbf{s} \in [\![\varphi]\!]_{Unbound(\mathcal{E}_{BS})}.$$

Now note that if initial state of $\mathcal{M}$ is v_{in} and the root of $\mathcal{T}(\mathcal{M})$ is s_{in}, then $\langle s_{in}\rangle \in Unbound(\langle v_{in}\rangle)$. Also note that $Unbound(\bot_B) = \bot$, where $\bot$ is the empty environment for $\mathcal{T}(\mathcal{M})$. From Lemma 5.3, Theorem 5.4 follows.

THEOREM 5.4. *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* $\langle v_{in}\rangle \in Eval(\varphi, \bot_B)$*. Further,* $Eval(\varphi, \bot_B)$ *is inductively computable.*

To understand this more concretely, let us see how this model-checking algorithm runs on a simple NSM. Consider the NSM abstraction $\mathcal{M}_{foo}$ in Section 2.1. The states of this NSM are v_1, v_2, v_3, v_4, v_5, and v_2'; the initial state is v_1. The states are labeled by the atomic propositions *rd*, *wr*, *tk*, *en*, and *ex* using a map defined as: $\kappa(v_0) = \emptyset$, $\kappa(v_1) = \{wr\}$, $\kappa(v_2) = \{en\}$, $\kappa(v_2') = \{ex\}$, $\kappa(v_3) = \{tk\}$, $\kappa(v_4) = \{rd\}$, and $\kappa(v_5) = \{end\}$. The transition relations of $\mathcal{M}_{foo}$ are given by:

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

Now 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 the set of summaries $\mathbf{S_2} = \mathbf{S_1} \cup \{\langle v_2', v_2, \{v_2'\}\rangle, \langle v_3, v_2, \{v_2'\}\rangle, \langle v_4, v_2, \{v_2'\}\rangle\}$, 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 $\langle call\rangle\varphi'\{\psi_1, \ldots, \psi_n\}$, 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 $Eval(\varphi, \bot_B)$ is $O(N^3|\varphi|N_V)$. For a formula φ of alternation depth d, this evaluation takes time $O(N^{3d}N_V^d\ |\varphi|)$, that is, 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 [Walukiewicz 2001]. Following constructions similar to those in Section 4, we can generate a NT-μ formula φ from a μ-calculus formula f expressing an alternating reachability 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 the following.

THEOREM 5.5. *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 [Walukiewicz 2001; Burkart and Steffen 1999]. These methods are complex, very different from what branching-time model checking looks like in the finite-state setting, and seem difficult to implement. In contrast, our algorithm is a simple fixpoint computation that computes, in a syntax-driven way, the 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.

6. AUTOMATA ON NESTED TREES

In this section, we study finite-state automata operating on nested trees, our generalization of automata on trees. Recall that for tree automata, the state while reading a (nonroot) tree node depends on its state at the node's parent. The state of a *nested tree automaton* at a node in a nested tree depends on its states at the node's parent and the node's jump-predecessor (if it exists). We define these automata in nondeterministic and alternating flavors; the natural semantics of these are respectively over ordered and unordered nested trees. Regarding acceptance conditions, we focus on the parity condition for the most part as it is the most powerful among the popular ω-acceptance conditions; however, we also discuss acceptance by final state.

We start with nondeterministic automata.

Definition 6.1 (NPNTA). A (top-down) *nondeterministic parity nested tree automaton* (NPNTA) over Σ is a structure $\mathcal{A} = (Q, q_0, \Delta, \Omega)$ where Q is a finite set of *states*, $q_0 \in Q$ is the initial state, $\Delta \subseteq Q \times \Sigma \times (TT \times TT)$, where $TT = Q \cup (Q \times Q) \cup \{\bot\}$, is a *transition relation*, and $\Omega : Q \to \{0, 1, \ldots, n\}$, for some $n \in \mathbb{N}$, is the *parity accepting condition* that assigns a *priority* to each automaton state.

NPNTAs accept languages of ordered, binary nested trees. A run of $\mathcal{A}$ on an ordered, binary nested tree $\mathcal{T} = ((S, r, \to_1, \to_2), \hookrightarrow, \lambda)$ is a labeling $\rho : S \to Q$ of nodes of $\mathcal{T}$ by automaton states such that:

(1) $\rho(r) = q_0$;
(2) for all s, if $\rho(s) = q$ and $\lambda(s) = \sigma$, then
   —if s_1 and s_2 are the left and right children of s, then for some $(q, \sigma, (\tau_1, \tau_2)) \in \Delta$, we have: (a) if s_i, for $i \in \{1, 2\}$, is a call or local node, then $\tau_i = \rho(s_i)$, and (b) if s_i is a return node, then $\tau_i = (\rho(t), \rho(s_i))$, where $t \hookrightarrow s_i$

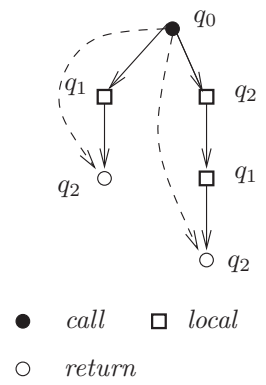

Fig. 10. A run of an NPNTA.

—if s' is the left (similarly, right) child of s and s has no right (similarly, left) child, then for some $(q, \sigma, (\tau', \bot)) \in \Delta$ (similarly, $(q, \sigma, (\bot, \tau')) \in \Delta$), we have: (a) if s' is a call or local node, then $\tau' = \rho(s')$, and (b) if s' is a return node, then $\tau' = (\rho(t), \rho(s'))$, where $t \hookrightarrow s'$

Let π_i denote the ith vertex in a path π in $\mathcal{T}$. A run ρ of $\mathcal{A}$ on $\mathcal{T}$ is *accepting* if for all infinite paths π in $\mathcal{T}$, $\theta' = \max\{\theta : \Omega(\rho(\pi_i)) = \theta \text{ for infinitely many } i\}$ is even. An ordered, binary nested tree $\mathcal{T}$ is *accepted* if $\mathcal{A}$ has an accepting run on it. The *language* $\mathcal{L}(\mathcal{A})$ of $\mathcal{A}$ is the set of nested trees it accepts.

Figure 10 illustrates part of a run of an NPNTA on a nested tree (let us assume that the label of every node is σ and that every node has a left child). Transitions include $(q_1, \sigma, ((q_0, q_2), \bot))$ and $(q_0, \sigma, (q_1, q_2))$.

We can define an equivalent semantics of NPNTAs by letting the automaton manipulate a stack rather than consult a node's jump-predecessor. In this case, $\mathcal{A}$ pushes the current state while taking a call edge, pops on a return edge, and leaves the stack unchanged on a local edge. As jump-edges are nested, the top of the automaton stack at a return node stores the state at the node's matching call.

As we shall see, unlike their analogs on trees, nondeterministic automata on nested trees do not have robust closure properties. However, this problem goes away for *alternating* nested tree automata, which we now consider.

We interpret our alternating automata on *unordered* nested trees.

Definition 6.2 (APNTA). For a finite set Q, define the set $TT(Q)$ of *transition terms* whose members f are of the form $f := tt \mid ff \mid f \vee f \mid f \wedge f \mid \langle loc \rangle q \mid [loc]q \mid \langle call \rangle q \mid [call]q \mid \langle ret, q' \rangle q \mid [ret, q']q$, where $q, q' \in Q$. An *alternating parity nested tree automaton* (APNTA) over Σ is a structure $\mathcal{A} = (Q, q_0, \Delta, \Omega)$, where Q is a finite set of states, $q_0 \in Q$ is the initial state, $\Delta : Q \times \Sigma \to TT(Q)$ is a transition function, and $\Omega : Q \to \{0, 1, \ldots, n\}$ is the parity accepting condition.

We define the semantics of an APNTA $\mathcal{A} = (Q, q_0, \Delta, \Omega)$ via a parity game. The acceptance game $\mathcal{G}(\mathcal{A}, \mathcal{T})$ of a Σ-labeled nested tree $\mathcal{T} = (T, \hookrightarrow, \lambda)$ by $\mathcal{A}$ is played by two players A and E. The vertex set of the game graph is $\mathcal{V} = T \times Q \times Q^* \times TT$, and the set of moves $\Rightarrow \subseteq \mathcal{V} \times \mathcal{V}$ is the least set such that:

—for all $v \in \mathcal{V}$ of the form $(s, q, \alpha, f_1 \vee f_2)$ or $(s, q, \alpha, f_1 \wedge f_2)$ for some $v' \in \mathcal{V} \cup \{\epsilon\}$, we have $v \Rightarrow (s, q, \alpha, f_1)$ and $v \Rightarrow (s, q, \alpha, f_2)$;

—for all $v \in \mathcal{V}$ of the form $(s, q, \alpha, \langle loc \rangle q'))$ or $(s, q, \alpha, [loc]q'))$, and for all s' such that $s \xrightarrow{loc} s'$, we have $v \Rightarrow (s', q', \alpha, f)$, where $f = \Delta(q', \lambda(s'))$;
—for all $v \in \mathcal{V}$ of the form $(s, q, \alpha, \langle call \rangle q')$ or $(s, q, \alpha, [call]q')$, and for all s' such that $s \xrightarrow{call} s'$, we have $v \Rightarrow (s', q', q.\alpha, f)$, where $f = \Delta(q', \lambda(s'))$;
—for all $v \in \mathcal{V}$ of the form $(s, q, q''.\alpha, \langle ret, q'' \rangle q')$ or $(s, q, q''.\alpha, [ret, q'']q')$, and for all s' such that $s \xrightarrow{ret} s'$, we have $v \Rightarrow (s', q', \alpha, f)$, where $f = \Delta(q', \lambda(s'))$;

The vertex set $\mathcal{V}$ is partitioned into two sets $\mathcal{V}_{\mathsf{E}}$ and $\mathcal{V}_{\mathsf{A}}$ corresponding to the two players. The set $\mathcal{V}_{\mathsf{A}}$ comprises vertices of the form (s, q, α, f), where s, q and α are arbitrary and f has the form tt, $[call]q$, $[loc]q$, $[ret, q']q$, or $(f_1 \wedge f_2)$. The remaining vertices constitute $\mathcal{V}_{\mathsf{E}}$. We also lift the priority map Ω to $\Omega_{\mathcal{V}} : \mathcal{V} \to \{0, 1, \ldots, n\}$ by defining $\Omega_{\mathcal{V}}(s, q, \alpha, f) = \Omega(q)$ for all s, q, α, and f.

The two players A and E play on the graph starting from the initial position $v_{in} = (s_0, q_0, \epsilon, \Delta(q_0, \lambda(s_0)))$ by moving a token along edges of the game graph. Whenever the token is in a position v, the player who owns the vertex must move the token. Formally, a *play* of $\mathcal{G}$ is a nonempty, finite or infinite sequence $\alpha = v_1 v_2 \cdots$ that is a path in the game graph, where $v_1 = v_{in}$. A finite play is winning for player A if the last position is a player E vertex from which there is no move; analogously, we define winning finite plays for player E. An infinite play α is winning for player E if $\theta' = \max\{\theta : \Omega_{\mathcal{V}}(v_i) = \theta \text{ for infinitely many } i\}$ is even; otherwise, A wins the play (this is the standard *max-parity* acceptance condition for parity games). A *strategy* for player E (or A) is a subset of edges $Str \subseteq \Rightarrow$ such that all these edges originate in a vertex in $\mathcal{V}_{\mathsf{E}}$ (or $\mathcal{V}_{\mathsf{A}}$).[1] A play is in accordance with a strategy Str if all edges in the play are in Str. A strategy is winning if all maximal plays in accordance with the strategy are winning.

An ApNta $\mathcal{A}$ *accepts* a nested tree $\mathcal{T}$ if E has a winning strategy in $G(\mathcal{A}, \mathcal{T})$. The *language* $\mathcal{L}(\mathcal{A})$ of $\mathcal{A}$ is the set of nested trees accepted by $\mathcal{A}$.

We also consider automata that accept by the weaker *final-state condition*. For nondeterministic versions of such automata, a nested tree is accepted if a special *final state* q_f is seen along every path in some run on it. In alternating versions, all infinite plays are won by A, and if a play reaches a game vertex (s, q_f, α, f) for some s, α, and f, then the game terminates and E is the winner.

6.1. Decision Problems

The *model checking* problem for ApNtas on nested state machines is the problem of deciding, given an ApNta $\mathcal{A}$ and a nested state machine $\mathcal{M}$, whether $\mathcal{T}(\mathcal{M}) \in \mathcal{L}(\mathcal{A})$.

Theorem 6.3. *The model-checking problem for* ApNtas *on nested state machines is* Exptime*-complete.*

Proof. We obtain an Exptime procedure for this problem via a reduction to a *pushdown parity game*. A two-player pushdown parity game is a parity game played on the configuration graph of a pushdown system. It is known that pushdown parity games are solvable in Exptime [Walukiewicz 2001]. Now, given an ApNta $\mathcal{A}$ and a nested state machine $\mathcal{M}$, $\mathcal{T}(\mathcal{M}) \in \mathcal{L}(\mathcal{A})$ iff player E wins the acceptance game of $\mathcal{A}$. Now recall that call-edges (similarly, return-edges) in $\mathcal{T}(\mathcal{M})$ encode call transitions (similarly, return transitions) of $\mathcal{M}$— that is, these capture pushes (similarly pops) to the stack implicit in an NSM abstraction. However, these edges are also where the stack of states in the semantics of $\mathcal{A}$ is pushed (popped). Thus, the implicit stack of $\mathcal{M}$ is "synchronized" with

[1]Strategies are often defined in a more general way that refer to the history of the play. This definition suffices as parity games always admit zero-memory strategies [Grädel et al. 2002].

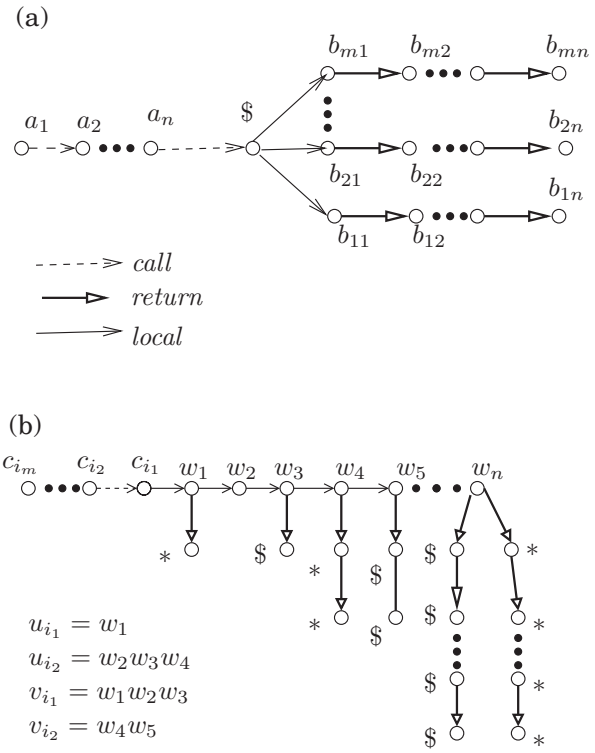

Fig. 11. (a) Closure properties of ApNTas and NpNTas; (b) Gadget for undecidability.

the implicit stack of $\mathcal{A}$, so that the graph of the acceptance game of $\mathcal{T}(\mathcal{M})$ by $\mathcal{A}$ happens to be the configuration graph of a pushdown system that is roughly the "synchronized product" of $\mathcal{M}$ and $\mathcal{A}$.

An EXPTIME-hardness result for this problem follows from the known hardness of the model-checking problem for alternating tree automata on pushdown systems [Walukiewicz 2001]. □

While model checking for ApNTas is decidable, emptiness is not.[2]

THEOREM 6.4. *Universality for nondeterministic nested tree automata and emptiness for alternating nested tree automata are undecidable problems, even for acceptance by final state.*

PROOF. The proof employs a reduction from the Post's Correspondence Problem (PCP) [Hopcroft and Ullman 1979]. Consider a tuple $((u_1, \ldots, u_k), (v_1, \ldots, v_k))$, where the u_i's and v_i's are finite words over an alphabet A; the PCP is to determine if there is a sequence $i_1, \ldots, i_m$, where $i_j \leq k$, such that $u_{i_1} u_{i_2} \cdots u_{i_m} = v_{i_1} v_{i_2} \cdots v_{i_m} = w$. Now consider nested trees of the form in Figure 11(b) (the jump-edges are omitted to keep the figure simple) such that the initial call-chain is of length m and is labeled by symbols from the alphabet $\{1, \ldots, k\}$, and the symbols w_i on the "stem" of local nodes succeeding this chain form the string w. Now suppose the sequence of input symbols on the call chain is $c_{i_m} \cdots c_{i_1}$. There are two kinds of return chains hanging from the stem—the

[2]This result was obtained independently by C. Löding (2005, Private communication).

ones marked with the symbol $*$ (similarly \$) are exactly at the points where w may be possibly factored into $u_{i_1}, u_{i_2}, \ldots, u_{i_m}$ (similarly $v_{i_1}, \ldots, v_{i_m}$). Also, the ith return chain (counting from left) of either type is of length i. Then such a nested tree is a witness for an instance of PCP being positive. We can, however, show that there is an alternating NTA accepting by final state that accepts the set of nested trees bisimilar to such witnesses. In fact, we can show that there is a nondeterministic final-state NTA that accepts any nested tree *not* of the previous form (under some ordering of edges). □

However, we can prove the emptiness problem of NPNTAs to be solvable in EXPTIME by reducing it to that for pushdown tree automata [Kupferman et al. 2002].

6.2. Closure Properties

Now we investigate the closure properties of APNTAs and NPNTAs.

THEOREM 6.5. APNTA*s are closed under union, intersection, and complement.* NPNTA*s are closed under union and intersection.*

PROOF. Proofs that APNTAs are closed under union and intersection are easy, and mirror corresponding results for alternating tree automata. We give a proof sketch of the closure of APNTAs under complement.

Consider an APNTA $\mathcal{A} = (Q, q_0, \Delta, \Omega)$. Let us define, for every transition term f, the transition term $Neg(f)$ as follows:

$$\begin{aligned}
Neg(f_1 \vee f_2) &= Neg(f_1) \wedge Neg(f_2)\\
Neg(f_1 \wedge f_2) &= Neg(f_1) \vee Neg(f_2)\\
Neg(tt) &= ff\\
Neg(ff) &= tt\\
Neg(\langle loc\rangle q) &= [loc]q\\
Neg([loc]q) &= \langle loc\rangle q\\
Neg(\langle call\rangle q) &= [call]q\\
Neg(()[call]q) &= \langle call\rangle q\\
Neg(\langle ret, q'\rangle q) &= [ret, q']q\\
Neg([ret, q']q) &= \langle ret, q'\rangle q.
\end{aligned}$$

Let $\Delta_\neg$ be the transition function defined as: if $\Delta(q, \sigma) = f$ for some q, σ, f, then $\Delta_\neg(q, \sigma) = Neg(f)$. Also, we define a new parity acceptance condition $\Omega_\neg$ as: for all $q \in Q$, $\Omega_\neg(q) = 1 + \Omega(q)$. Now we construct an APNTA $\mathcal{A}_\neg$ which is the same as $\mathcal{A}$, except its transition function is $\Delta_\neg$ and acceptance condition $\Omega_\neg$.

Now suppose that in the acceptance game $\mathcal{G}$ for $\mathcal{A}$ and a nested tree $\mathcal{T}$, Player E has a winning strategy (i.e., $\mathcal{T}$ is not accepted by $\mathcal{A}$). It is known, from research on parity games, that this strategy is *memoryless* [Grädel et al. 2002]. It can now be shown that the same strategy, applied to the acceptance game for $\mathcal{T}$ and $\mathcal{A}_neg$, is winning for Player A. As for the other direction, suppose Player E has no winning strategy in $\mathcal{G}$. As parity games are *determined*, this means A has a memoryless winning strategy in $\mathcal{G}$. We can now show that the same strategy is winning for E in the acceptance game for $\mathcal{A}_\neg$ (the arguments are the same as in the proof of closure of alternating tree automata under complement [Grädel et al. 2002]— as a result, we leave the details out). It follows that $\mathcal{A}_\neg$ accepts the complement of $\mathcal{L}(\mathcal{A})$.

Regarding NPNTAs, their closure under union is trivial. As for closure under intersection, consider NPNTAs $\mathcal{A}_1 = (Q^1, q_0^1, \Delta^1, \Omega^1)$ and $\mathcal{A}_2 = (Q^2, q_0^2, \Delta^2, \Omega^2)$, and construct an NPNTA $\mathcal{A}_\cap = (Q, q_0, \Delta, \Omega)$. Here, $Q = Q^1 \times Q^2$ and $q_0 = (q_0^1, q_0^2)$. Let us now define the "product" $Prod(\tau_1, \tau_2)$ of two transition terms τ_1 and τ_2 as follows:

—if $\tau_1 = \perp$ and $\tau_2 = \perp$, then $Prod(\tau_1, \tau_2) = \perp$;
—if $\tau_1 = q_1$ and $\tau_2 = q_2$, then $Prod(\tau_1, \tau_2) = (q_1, q_2)$;
—if $\tau_1 = (q_1', q_1)$ and $\tau_2 = (q_2', q_2)$, then $Prod(\tau_1, \tau_2) = ((q_1', q_2'), (q_1, q_2))$.

Then the transition relation Δ is the least relation such that for each transition $(q^1, a, (\tau_1^1, \tau_2^1))$ in Δ^1 and $(q^2, a, (\tau_1^2, \tau_2^2))$ in Δ_2 such that $Prod(\tau_1^1, \tau_1^2)$ and $Prod(\tau_2^1, \tau_2^2)$ are defined, we have a transition $((q^1, q^2), a, (Prod(\tau_1^1, \tau_1^2), Prod(\tau_2^1, \tau_2^2)))$ in Δ. Finally, we define Ω such that for all $(q^1, q^2) \in Q^1 \times Q^2$, we have $\Omega((q^1, q^2)) = \max\{\Omega^1(q^1), \Omega^2(q^2)\}$.
It can now be verified that $\mathcal{L}(\mathcal{A}_\cap) = \mathcal{L}(\mathcal{A}_1) \cap \mathcal{L}(\mathcal{A}_2)$. □

Observe that by our definition, languages accepted by ApNTAs are closed under bisimulation, while those accepted by NpNTAs are not in general. To compare the expressiveness of an ApNTA and an NpNTA meaningfully, we need to consider the language obtained by starting with the language L of the NpNTA, stripping the order between tree edges off nested trees in L, and closing it under bisimulation.[3] Formally, for a language L of ordered nested trees, we define $Unord(L)$ as the bisimulation closure of the set of nested trees $((S, r, \rightarrow), \hookrightarrow, \lambda)$ such that $\rightarrow = \rightarrow_1 \cup \rightarrow_2$ for some $((S, r, \rightarrow_1, \rightarrow_2), \hookrightarrow, \lambda) \in L$. Now we show the following.

THEOREM 6.6. *There is a language L of ordered, binary nested trees such that:* (1) *there is no* NpNTA *accepting L, and* (2) *there is an* ApNTA *accepting* $Unord(L)$.

PROOF. Consider ordered nested trees of the form in Figure 11(a), where $\Sigma = \{0, 1, \$\}$, and $a_i, b_{ij} \in \Sigma$ for all i, j (while the structure in the figure is not binary, it can be encoded as such; also, the jump-edges, omitted to keep the figure clean, can be reconstructed). Let L_{gap} be the language of such structures where for all $i \leq n$, there is some $k \leq m$ such that $a_{n-i+1} = b_{ki}$. First, we note that L_{gap} cannot be recognized by an NpNTA $\mathcal{A}_N$ with N states. To see why, take a structure as above where $n = m > N$, and for each $1 \leq i \leq n$, there is a *distinct* branch k such that $a_{n-i+1} = b_{ki}$. In any run, $\mathcal{A}_N$ must enter two branches in the same state; also, the sequence of states at calls unmatched till these points are the same. We can replace one of these branches with the other to get an accepting run on a structure not in L_{gap}.
On the other hand, it is easy to build an ApNTA $\mathcal{A}$ that recognizes $Unord(L_{gap})$. The automaton has a state q, q_0^* and q_1^*. While reading the sequence of symbols a_i, the automaton uses alternation to assert a condition at each i (we can view this as the automaton "forking" a copy). This is done such that the copy of $\mathcal{A}$ forked at the node labeled a_{n-i+1} has state $q^*_{a_{n-i+1}}$ at that node. On reading the next symbol, this copy changes its state to q; it continues reading the structure in this state till it reaches the node marked \$, at which point it nondeterministically chooses to "check" branch k. When it reaches the node labeled b_{ki} (i.e., the node s such that at the node t satisfying $t \hookrightarrow s$, the automaton had state q_j^* for some j), it can check if $b_{ki} = a_{n-i+1}$ and accept the nested tree. $\mathcal{A}$ accepts the structure iff every "copy" forked this way accepts it. □

This is an example of how automata theory for nested trees differs from the theory of tree automata. In the latter setting, allowing alternation does not affect an automaton's expressiveness. In fact, this is also a point of difference between nested tree automata and *automata on nested words* [Alur and Madhusudan 2004; 2006], as alternating and nondeterministic automata on nested words are expressively equivalent.
Further, Theorem 6.6 implies the following.

THEOREM 6.7. NpNTA*s are not closed under complementation.*

[3]Alternatively, we could define ApNTAs on ordered nested trees. Under this definition as well, ApNTAs are strictly more powerful than NpNTAs.

PROOF. We give an NPNTA $\mathcal{A}_N$ that accepts the complement of the language L_{gap} in the proof for Theorem 6.6. Consider a nested tree $\mathcal{T}$ in this language. We can assume that $\mathcal{T}$ has the same "shape" as the nested tree in Figure 11(a)— that is, there is a chain of nodes from the root that ends with a node with multiple children, each of which leads to a chain of nodes. This is because an NPNTA can determine if a nested tree violates this property, and accept it if it does.

While running on nested trees of this form, $\mathcal{A}_N$ nondeterministically guesses the i such that a_i cannot be "matched" along any of the branches, and lets its state at the node labeled a_i be $q^*_{a_i}$ (at other nodes along the stem of a-s, its state is q). Now it sends a state to each branch to determine a node s such that the automaton state at the node t, where $t \hookrightarrow s$, is of the form q^*_j. It compares the label b_{jk} of this node with a_i, and rejects the nested tree if $b_{jk} = a_i$. The accepts if all the copies sent along the different branches accept. It can now be easily verified that $\mathcal{A}_N$ accepts the complement of L_{gap}. □

The projection over Σ_1 of a language L of (ordered, unordered) nested trees over $\Sigma_1 \times \Sigma_2$ is the language $Proj_{\Sigma_1}(L)$ obtained by replacing every label (a, b) in every nested tree $\mathcal{T} \in L$ by a. We can show the following.

THEOREM 6.8. *NPNTAs are closed under projection, but* APNTAs *are not.*

PROOF. First we show that NPNTAs are closed under projection. Consider nested trees over $\Sigma_1 \times \Sigma_2$, and let there be an NPNTA $\mathcal{A}$ accepting a language of such trees. An NPNTA $\mathcal{A}'$ accepting the language $Proj_{\Sigma_1}(L)$ is constructed as follows. While reading a node labeled $a \in \Sigma_1$, $\mathcal{A}'$ guesses a second component $b \in \Sigma_2$ using nondeterminism, and mimics a move of $\mathcal{A}$ on the label (a, b). It is easy to see that $\mathcal{A}'$ accepts $Proj_{\Sigma_1}(L)$.

As for the second part of the theorem, consider a PCP instance as in Theorem 6.4. Now consider a simple NSM $\mathcal{M}$ such that every word over $\Sigma_1 \times \Sigma_2$ is the label of a path in the nested tree unfolding of $\mathcal{M}$. Now suppose a device existed to select a substructure in Figure 11(b) from the unfolding $\mathcal{T}(\mathcal{M})$. As in Theorem 6.4, an APNTA can check if such a substructure is a witness for the given PCP instance. Therefore, if an APNTA $\mathcal{A}$ could select an arbitrary substructure of a nested tree before operating on it, then we would be able to decide the PCP problem by model checking $\mathcal{A}$ on $\mathcal{M}$. This is, however, impossible.

Now note that if APNTAs were closed under projection, then the preceding selection operation could be performed by an APNTA. By contradiction, APNTAs are not closed under projection. □

7. EXPRESSIVENESS

In this section, we explore the expressiveness of NT-μ and nested tree automata. Our main result is a theorem that APNTAs are exactly as expressive as NT-μ. This result is the analog of the equivalence between the modal μ-calculus and alternating parity automata on trees [Emerson and Jutla 1991]. We also establish a *hierarchy theorem* for NT-μ, by which the expressiveness of NT-μ formulas is connected to their arity.

7.1. Relationship between NT-μ and APNTAs

We begin by proving our main expressiveness result.

THEOREM 7.1. *Given any closed* NT-μ *formula* φ*, we can construct an* APNTA $\mathcal{A}_\varphi$ *such that for any nested tree* $\mathcal{T}$*,* $\mathcal{T} \in \mathcal{L}(\varphi)$ *iff* $\mathcal{T} \in \mathcal{L}(\mathcal{A}_\varphi)$*. The size of* $\mathcal{A}_\varphi$ *is polynomial in the size of* φ*.*

PROOF. The proof is similar in spirit to the known translation from the μ-calculus to alternating tree automata [Emerson and Jutla 1991]. The APNTA $\mathcal{A}_\varphi$ is over an input alphabet 2^{AP}. For every subformula ψ of φ, $\mathcal{A}_\varphi$ has a state q_ψ. The initial state is q_φ.

For any variable X in φ, let $\Psi(X)$ be the subformula of the form $\mu X.\varphi'$ or $\nu X.\varphi'$ that binds X (we assume that each variable in φ is bound at most once). For instance, if $\varphi = \langle call\rangle(\mu X.(p \vee X))\{q\}$, then $\Psi(X) = \mu X.(p \vee X)$. For each bound variable X in φ, the state q_X is *identified* with the state $q_{\Psi(X)}$.

Let $p \in AP$, and $\sigma \in 2^{AP}$. The transition relation Δ of $\mathcal{A}_\varphi$ is defined inductively over the structure of φ:

$$
\begin{aligned}
\Delta(q_p, \sigma) &= tt \text{ if } p \in \sigma, \text{ else } ff \\
\Delta(q_{\varphi_1 \wedge \varphi_2}, \sigma) &= \Delta(q_{\varphi_1}, \sigma) \wedge \Delta(q_{\varphi_2}, \sigma) \\
\Delta(q_{\varphi_1 \vee \varphi_2}, \sigma) &= \Delta(q_{\varphi_1}, \sigma) \vee \Delta(q_{\varphi_2}, \sigma) \\
\Delta(q_{\mu X.\varphi'}, \sigma) &= \Delta(q_{\varphi'}, \sigma) \\
\Delta(q_{\nu X.\varphi'}, \sigma) &= \Delta(q_{\varphi'}, \sigma) \\
\Delta(q_{\langle call\rangle(\varphi')\{\psi_1,\ldots,\psi_k\}}, \sigma) &= \langle call\rangle q_{\varphi'} \\
\Delta(q_{[call](\varphi')\{\psi_1,\ldots,\psi_k\}}, \sigma) &= [call] q_{\varphi'} \\
\Delta(q_{\langle loc\rangle\varphi'}, \sigma) &= \langle loc\rangle q_{\varphi'} \\
\Delta(q_{[loc]\varphi'}, \sigma) &= [loc] q_{\varphi'} \\
\Delta(q_{\langle ret\rangle R_i}, \sigma) &= \bigvee_{\phi', \psi_{1\le j\le k}} (\langle ret, q_{\langle call\rangle(\phi')\{\psi_1,\ldots,\psi_k\}}\rangle q_{\psi_i} \vee \\
&\qquad \langle ret, q_{[call](\phi')\{\psi_1,\ldots,\psi_k\}}\rangle q_{\psi_i}) \\
\Delta(q_{[ret]R_i}, \sigma) &= \bigvee_{\phi', \psi_{1\le j\le k}} ([ret, q_{\langle call\rangle\phi'\{\psi_1,\ldots,\psi_k\}}] q_{\psi_i} \\
&\qquad \vee [ret, q_{[call]\phi'\{\psi_1,\ldots,\psi_k\}}] q_{\psi_i}).
\end{aligned}
$$

The priority of states of the form $q_{\mu X.\varphi}$ and $q_{\nu X.\varphi}$ are respectively odd and even, and roughly equal to the alternation depth of φ. The priority for all other states is 0. We do not define the acceptance condition in detail as we do not prove its correctness in this setting—this is because this part of the proof is exactly the same as in the translation from the μ-calculus to alternating parity tree automata.

We will, however, discuss in some more detail the transition relation of $\mathcal{A}_\varphi$. The automaton implements the operational semantics of the formula φ. If $\varphi = p$ or $\varphi = \neg p$, $\mathcal{A}_\varphi$ checks if the atomic proposition p holds at the node of the nested tree currently being read. Conjunction and disjunction in φ is captured respectively by conjunction and disjunction in the transition relation of $\mathcal{A}_\varphi$. If $\varphi = X$, then in the operational semantics of φ, the fixpoint formula binding X is executed recursively. Accordingly, $\mathcal{A}_\varphi$ loops to the state $\Psi(X)$. At a fixpoint formula of the form $\mu X.\varphi'$ or $\mu X.\varphi'$, the automaton starts the corresponding fixpoint computations by moving to the state for φ'. All this is exactly the same as for the μ-calculus and alternating tree automata. The difference from the tree setting is in the treatment of modal operators.

If $\varphi = \langle loc\rangle\varphi'$, the corresponding automaton makes a transition to the state for φ' along some *loc*-edge from the current node. The [*loc*] modality is similar. Now suppose $\varphi = \langle call\rangle\varphi'\{\psi_1, \ldots, \psi_k\}$. In this case $\mathcal{A}_\varphi$ transitions to the state for the formula φ' along some *call*-edge from the current node (the [*call*] modality is similar, except the automaton sends copies along all *call*-edges). The constraint is that the automaton must be at the state q_{ψ_i} at the jump-successors of the current node marked by color i. The automaton checks this constraint using its $\langle ret\rangle$ and [*ret*] modalities. Consider a formula $\langle ret\rangle R_i$ that asserts that some *ret*-successor of the current node has color i. Consider such a successor t and its jump-predecessor s. The automaton checks, using a disjunction, if the automaton state at node s (in the current copy of the automaton)

corresponds to *any* subformula of φ starting with a $\langle call\rangle$ or $[call]$ modality. If so—that is, if a formula of the form $\langle call\rangle\varphi'\{\psi_1,\ldots,\psi_k\}$ or $[call]\varphi'\{\psi_1,\ldots,\psi_k\}$ was asserted at the jump-predecessor s of this *ret*-successor— the automaton has to check that the nested tree from node t on satisfies the ith return obligation—that is, ψ_i—asserted by that formula. Accordingly, it changes state to q_{ψ_i}. □

THEOREM 7.2. *Given any* APNTA $\mathcal{A}$*, one can construct an* NT-μ *formula* $\varphi_{\mathcal{A}}$ *such that for any nested tree* $\mathcal{T}$*,* $\mathcal{T}\in\mathcal{L}(\varphi_{\mathcal{A}})$ *iff* $\mathcal{T}\in\mathcal{L}(\mathcal{A})$*. The size of* $\varphi_{\mathcal{A}}$ *is polynomial in the size of* φ*.*

We will first establish the above for alternating nested tree automata $\mathcal{A}$ accepting by a *final state* q_f. The reason is that the proof is simpler and more intuitive in this case. After this, we will present the full proof.

PROOF. It will be simpler for us to write the formula $\varphi_{\mathcal{A}}$ using a set of equations rather than in the standard form. Translation from this equational form to the standard form is as for the modal μ-calculus [Grädel et al. 2002].

Let $Q=\{q_1,\ldots,q_n\}$ and TT, respectively, be the sets of states and transition conditions of $\mathcal{A}$. For each $q\in Q$, we have a marker R_q; for each pair of states $q,q'\in Q$, we have a variable $X_{q,q'}$. Intuitively, a summary $\langle s,U_{q_1},\ldots,U_{q_n}\rangle$ is collected in $X_{q,q'}$ iff $\mathcal{A}$ has a way to start at node s at state q, and end up at a return $s'\in U_{q_j}$ in state q_j, having checked that q' was the state of the automaton in the current play at the jump-predecessor of s'. Now for each pair of states $q,q'\in Q$, we define a map $\mathcal{F}_{q,q'}:TT\to\Phi$, where Φ is the set of NT-μ formulas:

$$\begin{aligned}
\mathcal{F}_{q,q'}(tt) &= tt\\
\mathcal{F}_{q,q'}(ff) &= ff\\
\mathcal{F}_{q,q'}(f_1\wedge f_2) &= \mathcal{F}_{q,q'}(f_1)\wedge\mathcal{F}_{q,q'}(f_2)\\
\mathcal{F}_{q,q'}(f_1\vee f_2) &= \mathcal{F}_{q,q'}(f_1)\vee\mathcal{F}_{q,q'}(f_2)\\
\mathcal{F}_{q,q'}(\langle call\rangle q'') &= \langle call\rangle(X_{q'',q})\{X_{q_1,q'},\ldots,X_{q_n,q'}\}\\
\mathcal{F}_{q,q'}([call]q'') &= [call](X_{q'',q})\{X_{q_1,q'},\ldots,X_{q_n,q'}\}\\
\mathcal{F}_{q,q'}(\langle loc\rangle q'') &= \langle loc\rangle X_{q'',q'}\\
\mathcal{F}_{q,q'}([loc]q'') &= [loc]X_{q'',q'}\\
\mathcal{F}_{q,q'}(\langle ret,q\rangle q'') &= \langle ret\rangle(R_{q''})\\
\mathcal{F}_{q,q'}([ret,q]q'') &= [ret]R_{q''}.
\end{aligned}$$

Then the formula $\varphi_{\mathcal{A}}$ is the formula corresponding to X_{q_0,γ_0} when taking the least fixpoint of the following equations:

$$X_{q,q'}=\begin{cases} tt & \text{if } q=q_f\\ \bigvee_{\sigma\subseteq AP}((\bigwedge_{p\in\sigma}p)\wedge(\bigwedge_{p\notin\sigma}\neg p)\wedge\mathcal{F}_{q',q}(\Delta(q,\sigma))) & \text{otherwise.}\end{cases}$$ □

Now we give the general proof. Our translation from APNTAS to NT-μ uses finite-state alternating parity tree automata, which we will define now. The automata we use are bisimulation-closed and run on unranked trees—similar definitions may be found in Kirsten's survey chapter [Grädel et al. 2002] on alternating tree automata. For a set I of tags, a set Q of states, and a set AP of atomic propositions, let a *(finite-state) transition term* be of the form $g=p\mid\neg p\mid q\mid\langle a\rangle g\mid[a]g\mid g\wedge g\mid g\vee g$, where $a\in I$, $p\in AP$, and $q\in Q$. Fix a set TT^{at} of such terms. A finite-state alternating parity tree automaton (TA) over TT^{at} is a structure $\mathcal{M}=(Q,q_0,\gamma_0,\Delta,\Omega)$, where Q is a finite set of *automaton states*, $q_0\in Q$ is the initial state, $\Delta:Q\to TT^{at}$ is a *transition function*, and $\Omega:Q\to\{1,2,\ldots,n\}$ is the *parity accepting condition* that assigns a *priority* to each automaton state.

TAs run on unordered infinite trees whose nodes are labeled by 2^{AP} and edges by I. We skip a formal definition of the semantics. Intuitively, the term p means that the proposition p holds at the current node, $\langle a\rangle q$ means that the automaton propagates the state q along some a-labeled edge, and the term $[a]q$ means that the state q is passed along all a-edges. Note that we allow complex terms like $[a_1]\langle a_2\rangle q$, which means that the automaton first takes all edges labeled a_1, and then, from each child of the current node so reached, picks an edge labeled a_2 and passes the state q to it. Terms can be combined conjunctively and disjunctively, as is par course for alternating automata. Our acceptance condition is max-parity, meaning a run is accepting if along all its infinite paths, the maximum priority seen infinitely often is even. We write $\mathcal{T} \models \mathcal{M}$ if a labeled tree $\mathcal{T}$ is accepted by $\mathcal{M}$.

Our proof also depends on a translation from TAs to the modal μ-calculus (actually, we will be interested in a syntactic fragment of the μ-calculus). Recall that formulas of the μ-calculus over a set of variables *Var* have syntax

$$\varphi = p \mid \neg p \mid \langle a\rangle\varphi \mid [a]\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid X \mid \mu X.\varphi \mid \nu X.\varphi,$$

where $a \in I$, $X \in \mathit{Var}$ and $p \in AP$.

Such formulas are interpreted over $(2^{AP}, I)$-labeled trees under environments $\mathcal{E} : X \mapsto S'$, where S' is a set of nodes in the tree in question. Again, we skip a formal definition of the semantics. We write $T, \mathcal{E} \models \varphi$ if a tree $\mathcal{T}$ satisfies the formula φ under an environment $\mathcal{E}$ (sometimes we write $t, \mathcal{E} \models \varphi$ if $\mathcal{T}$ in the above is rooted at the node t). If φ is closed, we can omit $\mathcal{E}$.

Now we define a special class of labeled trees obtained by applying a "summarization" transformation to nested trees. Speaking roughly, these trees have summaries as their nodes (in addition to some intermediate nodes), and trees rooted at returns are "plucked" and "lifted" to the matching call-sites. Formally, let NT be the set of all nested trees and $\mathbb{T}$ the set of all node and edge-labeled trees. We define a map $\mathit{Summarize} : NT \times \mathbb{N} \to \mathbb{T}$.

Let $k \geq 0$ and $I = \{\mathit{call}, \mathit{ret}, \mathit{loc}\}$. We consider the augmented set of atomic propositions $\widehat{AP} = AP \cup \{\mathit{leaf}_i : 1 \leq i \leq k\}$, and the set of edge labels $\widehat{I} = I \cup \{\mathit{choose}, \mathit{in}\} \cup \{i : 1 \leq i \leq k\}$. Then for a nested tree $\mathcal{S} = (((S, s_0, \to), \hookrightarrow), \lambda)$ whose nodes are labeled by the alphabet 2^{AP}, we define a node and edge-labeled tree $\mathcal{T} = \mathit{Summarize}(\mathcal{S}, k)$. Let $\mathit{Summ}_k^{\mathcal{S}}$ be the set of k-colored summaries in $\mathcal{S}$. Then, the set of nodes of $\mathcal{T}$ is $T \subseteq (\mathit{Summ}_k^{\mathcal{S}} \cup T')^+$, where $T' = \{(U_1, U_2, \ldots, U_k) :$ for all $1 \leq j \leq k$, $U_j \subseteq \mathit{ME}(u)$ for some $u \in S\}$ (the node set will be defined more precisely soon). The root of $\mathcal{T}$ is $t_0 = \langle s_0, \emptyset, \ldots, \emptyset\rangle$. The edges of the tree are labeled by the alphabet $\widehat{I}$ using a map η_T; the nodes are labeled by the alphabet $\widehat{\Sigma} = 2^{\widehat{AP}}$ using a map λ_T.

The set T of nodes and the set E_T of edges of $\mathcal{T}$ are those obtained when the computation described below reached a fixpoint. The set *Leaves* of leaf-nodes where we chop $\mathcal{S}$ is also obtained as part of the same least fixpoint. Initially, $T = \{t_0\}$ and $\mathit{Leaves} = E_T = \emptyset$. Now, let us write $t \overset{a}{\rightarrowtail} t'$ if there is an a-labeled edge from node t to node t' in $\mathcal{T}$. Then we have the following.

(1) For each s' such that $s \xrightarrow{\mathit{loc}} s'$ in $\mathcal{S}$ and each node $t.\mathbf{s} \notin \mathit{Leaves}$ in $\mathcal{T}$, where $\mathbf{s} = \langle s, U_1, U_2, \ldots, U_k\rangle$ is a summary in $\mathcal{S}$, we add a node $t.\mathbf{s}.\mathbf{s}'$ to T, where $\mathbf{s}' = \langle s', U_1', U_2', \ldots, U_k'\rangle$ such that $U_j' = U_j \cap \mathit{MR}(s')$ for all j. Also, we add an edge $t.\mathbf{s} \overset{\mathit{loc}}{\rightarrowtail} t.\mathbf{s}.\mathbf{s}'$ to $\mathcal{T}$.

(2) For each s' such that $s \xrightarrow{\mathit{call}} s'$ in $\mathcal{S}$ and each node $t.\mathbf{s} \notin \mathit{Leaves}$, where $\mathbf{s} = \langle s, U_1, U_2, \ldots, U_k\rangle \notin \mathit{Leaves}$ is a summary of $\mathcal{S}$:

(a) we add to T the node $t_1 = t.\mathbf{s}.s'$. We also add the edge $t.\mathbf{s} \overset{choose}{\rightarrowtail} t_1$.
(b) For every t_1, we add to T every node that is of the form $t_2 = t_1.(V_1, V_2, \ldots, V_k)$, where $V_1, \ldots, V_k \subseteq ME(s')$. We also add edges $t_1 \overset{choose}{\rightarrowtail} t_2$ for each such t_2.
(c) For every t_2, we add to T the node $t_3 = t_2.\mathbf{s}'$, where $\mathbf{s}' = \langle s', V_1, V_2, \ldots, V_k\rangle$, and for every $s'' \in V_j$ for some j, we add the node $t_4 = t_2.\mathbf{s}''$, where $\mathbf{s}'' = \langle s'', U_1', U_2', \ldots, U_k'\rangle$ is a summary such that $U_i' = U_i \cap ME(s'')$. We also add the edge $t_2 \overset{in}{\rightarrowtail} t_3$, and, for each t_4 as above such that $s'' \in V_j$, the edge $t_2 \overset{j}{\rightarrowtail} t_4$.

Nodes of the form $t.s$ and $t.(V_1, \ldots, V_k)$, where $s \in S$, $t \in \{\epsilon\} \cup T$ and $V_1, \ldots, V_k \subseteq T$, will be sometimes referred to as *intermediate nodes*.

(3) For each s' such that $s \overset{ret}{\longrightarrow} s'$ in $\mathcal{S}$ and each node $t.\mathbf{s} \notin \mathit{Leaves}$ of $\mathcal{T}$ such that $\mathbf{s} = \langle s, U_1, U_2, \ldots, U_k\rangle$ is a summary, we add the node $t.\mathbf{s}.\widetilde{s'}$ to T, where $\widetilde{s'} = (s', U_1, U_2, \ldots, U_k)$. Note that $\widetilde{s'}$ is not a summary. We also add the edge $t.\mathbf{s} \overset{ret}{\rightarrowtail} t.\mathbf{s}.\widetilde{s'}$ to E_T, and $t.\mathbf{s}.\widetilde{s'}$ to Leaves.

Note that this also defines the edge-labeling function η_T in $\mathcal{T}$. The node-labeling map $\lambda_T : T \to \widehat{\Sigma}$ is defined as: for $t \in T \setminus \mathit{Leaves}$, $\lambda_T(t.\mathbf{s}) = \lambda(s)$ if $t \in \{\epsilon\} \cup T$ and $\mathbf{s} = \langle s, U_1, \ldots, U_k\rangle \in \mathit{Summ}_k^S$, and $\lambda_T(t.(V_1, \ldots, V_k)) = \lambda_T(t.s) = \emptyset$ for all $t, s, V_1, \ldots, V_k$. For $t' = t.\widetilde{s} \in \mathit{Leaves}$, where $\widetilde{s} = (s, U_1, \ldots, U_k)$, we set $\lambda_T(t') = \{\mathit{leaf}_j : s \in U_j\}$.

Now we proceed to the lemmas to be used in this proof. First, let AP be defined as before; now for a set Q of TA states, fix the set of atomic terms $TT^{at}(Q, AP)$, comprising the terms

p	$\neg p$
$\langle loc\rangle q$	$[loc]q$
$\langle ret\rangle \mathit{leaf}_i$	$[ret]\mathit{leaf}_i$
$\langle call\rangle\langle choose\rangle([in]q \wedge [1]\, q_1 \wedge \cdots \wedge [k]\, q_k)$	
$[call]\langle choose\rangle([in]\, q \wedge [1]\, q_1 \wedge \cdots \wedge [k]\, q_k)$	

for all $q, q_1, \ldots, q_k \in Q$, $1 \le i \le k$, and $p \in AP$. Then we have the following Lemma.

LEMMA 7.3. *Given an* ApNTa $\mathcal{A}$*, we can effectively construct a TA* $\mathcal{F}(\mathcal{A})$ *over* $TT^{at}(Q, AP)$ *such that for any nested tree* $\mathcal{S}$*, we have* $\mathcal{S} \in \mathcal{L}(\mathcal{A})$ *iff* $\mathit{Summarize}(\mathcal{S}, k) \models \mathcal{F}(\mathcal{A})$*. The size of* $\mathcal{F}(\mathcal{A})$ *is polynomial in the size of* $\mathcal{A}$*.*

The second lemma is an augmentation of the translation from TAs to the modal μ-calculus. Consider TAs over the set of atomic terms $TT^{at}(Q, AP)$ defined previously. Now consider the syntactic fragment of the μ-calculus, parameterized by TT^{at}, whose formulas φ are given by:

$$\begin{aligned}\varphi \;=\; & p \mid \neg p \mid X \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \langle loc\rangle\, \varphi \mid [loc]\, \varphi \\ & \mid \langle call\rangle\langle choose\rangle([in]\varphi \wedge [1]\varphi \wedge \cdots \wedge [k]\varphi) \\ & \mid [call]\langle choose\rangle([in]\varphi \wedge [1]\varphi \wedge \cdots \wedge [k]\varphi) \\ & \mid \langle ret\rangle \mathit{leaf}_i \mid [ret]\mathit{leaf}_i \mid \mu X.\varphi \mid \nu X.\varphi,\end{aligned}$$

where $p \in AP$ and $X \in \mathit{Var}$. Let $\Phi(TT^{at})$ be the set of formulas in this form. Then we have Lemma 7.4.

LEMMA 7.4. *For any TA* $\mathcal{M}$ *with state set* Q *and over the set of atomic terms* $TT^{at}(Q, AP)$*, we can construct a closed formula* $\Psi(\mathcal{M}) \in \Phi(TT^{at})$ *such that for any* $(\widehat{\Sigma}, \widehat{I})$*-labeled tree* $\mathcal{T}$*, we have* $\mathcal{T} \models \Psi(\mathcal{M})$ *iff* $\mathcal{T} \models \mathcal{M}$*. The size of* $\Psi(\mathcal{M})$ *is polynomial in the size of* $\mathcal{M}$*.*

The third lemma connects μ-calculus formulas of the above form to NT-μ formulas.

LEMMA 7.5. *Let φ be a closed μ-calculus formula in $\Phi(TT^{at})$. Then, from φ we can construct a* NT-μ *formula $Nest(\varphi)$ such that for any $(\widehat{\Sigma}, \widehat{\Gamma})$-labeled tree $\mathcal{T}$ satisfying $\mathcal{T} = Summarize(\mathcal{S}, k)$ for some nested tree $\mathcal{S}$ over 2^{AP}, we have $\mathcal{T} \models \varphi$ iff $\mathcal{S} \in \mathcal{L}(Nest(\varphi))$. The size of $Nest(\varphi)$ is polynomial in the size of φ.*

Let us first establish that if these three lemmas hold, then Theorem 7.2 holds. This is proved by the following construction. Given $\mathcal{A}$, we construct the NT-μ formula $\varphi_{\mathcal{A}} = Nest(\Psi(\mathcal{F}(\mathcal{A})))$. By these lemmas, this construction is possible. Now fix any structured tree $\mathcal{S}$. If $\mathcal{S} \models \mathcal{A}$, then by Lemma 7.3, $Summarize(\mathcal{S}, k) \models \mathcal{F}(\mathcal{A})$, and by Lemma 7.4, $Summarize(\mathcal{S}, k) \models \Psi(\mathcal{F}(\mathcal{A}))$. But then, by Lemma 7.5, $\mathcal{S} \models \varphi_{\mathcal{A}}$. Similarly, if $\mathcal{S} \models \varphi_{\mathcal{A}}$, it is easily established that $\mathcal{S} \models \mathcal{A}$. □

Let us now prove Lemmas 7.3-7.5.

PROOF OF LEMMA 7.3. Let $\mathcal{A} = (Q, q_0, \Delta, \Omega : Q \to \{1, \ldots, n\})$ be an APNTA. Let $\perp$ be a special state not in Q. Then states of $\mathcal{F}(\mathcal{A})$ are of the form $r = (q, \gamma, \theta, m)$, where $q \in Q$, $\gamma \in Q \cup \{\perp\}$, and $\theta, m \in \{1, \ldots, n\}$ are priorities of $\mathcal{A}$. Let us now consider a map *indexmap* : $Q \times \{1, \ldots, n\} \to \{1, \ldots, |Q|n\}$ that assigns a unique index to every tuple (q, θ). Intuitively, *indexmap* assigns an index to the fact that "$\mathcal{A}$ can end up at a matching exit of the current node in state q, with θ as the minimum priority seen in the current context."

We will present the semantics of some of the interesting transitions of $\mathcal{F}(\mathcal{A})$ in English. Since treatment of alternation is similar in APNTAs and TAs, we will focus on "atomic" transition terms. Suppose $\mathcal{F}(\mathcal{A})$ is in state $r = (q, \gamma, \theta, p)$ while reading a node t of $\mathcal{T}$. Let us now have, say, $\Delta(q, \lambda(t)) = \langle call \rangle q'$ or $\Delta(q, \lambda(t)) = [call]q'$, In these two cases, $\mathcal{F}(\mathcal{A})$ forks copies to check that the $\lambda(t)$ is precisely the set of propositions satisfied at t; it also forks a copy that respectively reads one or all of the *call*-children of t. Each of these children are intermediate nodes, each with (uncountably) many $choose$-children. $\mathcal{F}(\mathcal{A})$ now reads *one* of these $choose$-children; once this is done, it is at an intermediate node with outgoing transitions labeled $in, 1, \ldots, k$. At this point $\mathcal{F}(\mathcal{A})$ passes:

(1) the state $(q', q, 0, \Omega(q'))$ to the unique child along the edge labeled in;
(2) the state $(q'', q, \max(\theta, m), m)$ to *every* child along a transition labeled j, for every j, iff $j = indexmap(q'', m)$. Now the move is over.

If $\Delta(q, \lambda(t)) = \langle loc \rangle q'$ or $[loc]q'$, then from state r, respectively along some or all *loc*-edges, the TA $\mathcal{F}(\mathcal{A})$ passes the state $(q', z, \max(\theta, \Omega(q)), \Omega(q))$ (also, copies need to be forked to ensure that $\lambda(t)$ is satisfied currently).

Let us now assume that $\Delta(q, \lambda(t)) = \langle ret, q'' \rangle q'$ or $[ret, q'']q'$. Again, it is made sure that $\lambda(t)$ holds currently. Now, if r is the current state of $\mathcal{F}(\mathcal{A})$, the state $(q', q'', \max(\theta, \Omega(q)), \Omega(q))$ is passed respectively along some or all *ret*-edges out of t. By definition of $\mathcal{T}$, the child $t' = t.\widetilde{s'}$ is a leaf. At this point $\mathcal{F}(\mathcal{A})$ accepts if $leaf_j \in \lambda_{\mathcal{T}}(t')$, where $j = indexmap(q', \max(\theta, \Omega(q)))$, and rejects otherwise.

The priority function $\widehat{\Omega}$ of a state $r = (q, \gamma, \theta, m)$ is defined as: $\widehat{\Omega}(r) = m$.

The TA $\mathcal{F}(\mathcal{A})$ is clearly over the restricted set of transition terms $TT^{at}(Q, AP)$ that we fixed. Lemma 7.3 may now be established using arguments used by Walukiewicz in the context of pushdown parity games [Walukiewicz 2001]. Finally, note that $\mathcal{F}(\mathcal{A})$ is polynomial in the size of $\mathcal{A}$. □

PROOF OF LEMMA 7.4. We follow a translation from TAs to the μ-calculus discussed in a survey chapter by Alberucci [Grädel et al. 2002]. The key observation that we exploit is that this translation can be carried out even when we restrict TAs to the special form that is under consideration here.

The proof proceeds by induction on the *index n* of the TA $\mathcal{M}$, defined as the maximum number of distinct priorities in a strongly connected component in the state transition graph of $\mathcal{M}$. It also uses a lemma about the existence of simultaneous fixpoints. We will not reproduce the arguments here in detail, but will present a skeleton.

First, we note that the simultaneous fixpoint theorem holds even for the subset of the μ-calculus we consider here.

LEMMA 7.6. *Let $T_1, \ldots, T_k$ be sets of nodes of $\mathcal{T}$, and let $\delta_1, \ldots, \delta_k \in \Phi(TT^{at})$ be μ-calculus formulas in $\Phi(TT^{at})$ with k free variables each. Now consider the monotone map $F : (T_1, \ldots, T_k) \mapsto (\llbracket\delta_1(T_1, \ldots, T_k)\rrbracket, \ldots, \llbracket\delta_k(T_1, \ldots, T_k)\rrbracket)$. There are μ-formulas $\tau_1, \ldots, \tau_k \in \Phi(TT^{at})$ and ν-formulas $\rho_1, \ldots, \rho_k \in \Phi(TT^{at})$ such that $(\tau_1, \ldots, \tau_k)$ and $(\rho_1, \ldots, \rho_k)$ are, respectively, the least and greatest fixpoints of F.*

The proof of this follows from the fact that simultaneous fixpoints are obtained by substitution of variables by formulas, so that the basic structure of the modalities remains unaltered.

Let us now go through the induction. The idea is to inductively replace states appearing in infinite cycles by variables. *Base case*: $n = 0$, so that the automaton does not have any cycles in its state transition graph. In this case, the formula $\Psi(\mathcal{M})$ is obtained by "expanding" the transitions; given the structure of TT^{at}, we conclude that the $\Psi(\mathcal{M})$ is in $\Phi(TT^{at})$.

For the induction step, consider the TA $\mathcal{M}(Q', q')$ obtained by duplicating the state $q' \in Q' \subseteq Q$ and declaring q' as the new initial state, and then replacing all states $q \in Q'$ by propositions. Intuitively, this operation identifies the runs of $\mathcal{M}$ starting from state q', while "chopping off" said runs at states in Q'. Clearly, $\mathcal{M}(Q', q')$ is also a TA over TT^{at}; it turns out that $\mathcal{M}(Q', q')$ has index less than n and hence can be translated to an equivalent formula in $\Phi(TT^{at})$. Now we let Q' be the set of states of maximum priority in $\mathcal{M}$, set q_i to be the i-th state in Q', and consider the formula $\delta_i \in \Phi(TT^{at})$ equivalent to $\mathcal{M}(Q', q_i)$. We do this for all i, then plug these formulas δ_i into Lemma 7.6 and obtain the formulas $\tau_1, \ldots, \tau_k$ corresponding to their simultaneous fixpoints (depending on whether the maximum priority is odd or even, we will need least or greatest fixpoints). We also look at the formula τ_0 equivalent to $\mathcal{M}(Q')$, which is simply $\mathcal{M}$ with states in Q' moved to propositions.

Then, $\Psi(\mathcal{M}) = \tau_0(\tau_1, \ldots, \tau_k)$, which, by our arguments, belongs to $\Phi(TT^{at})$. Further, as the proof in Alberucci's chapter shows, $\Psi(\mathcal{M})$ is polynomial in the size of $\mathcal{M}$. □

PROOF OF LEMMA 7.5. For subformulas of a μ-calculus formula φ in $\Phi(TT^{at})$, let us define $Nest(\varphi)$ inductively:

$$Nest(p) = p \tag{1}$$
$$Nest(\neg p) = \neg p \tag{2}$$
$$Nest(X) = X \tag{3}$$
$$Nest(\varphi_1 \wedge \varphi_2) = Nest(\varphi_1) \wedge Nest(\varphi_2) \tag{4}$$
$$Nest(\varphi_1 \vee \varphi_2) = Nest(\varphi_1) \vee Nest(\varphi_2) \tag{5}$$
$$Nest(\langle loc\rangle\varphi) = \langle loc\rangle Nest(\varphi) \tag{6}$$
$$Nest([loc]\ \varphi) = [loc]\ Nest(\varphi) \tag{7}$$
$$Nest(\langle call\rangle\langle choose\rangle ([in]\varphi \wedge [1]\psi_1 \wedge \cdots \wedge [k]\psi_k)) = \langle call\rangle(Nest(\varphi))\ \{Nest(\psi_1), \ldots, Nest(\psi_k)\} \tag{8}$$
$$Nest([call]\langle choose\rangle ([in]\varphi \wedge [1]\varphi_1 \wedge \cdots \wedge [k]\varphi_k)) = [call](Nest(\varphi))\ \{Nest(\psi_1), \ldots, Nest(\psi_k)\} \tag{9}$$
$$Nest(\langle ret\rangle leaf_i) = \langle ret\rangle R_i \tag{10}$$

$$Nest([ret]\ leaf_i) = [ret]R_i \quad (11)$$
$$Nest(\mu X.\varphi) = \mu X.Nest(\varphi) \quad (12)$$
$$Nest(\nu X.\varphi) = \nu X.Nest(\varphi) \quad (13)$$

Now, let $\mathcal{S}$ be any nested tree and $\mathcal{T} = Summarize(\mathcal{S}, k)$ as before; recall that $Summ_k^{\mathcal{S}}$ is the set of summaries in $\mathcal{S}$ and T the set of nodes of $\mathcal{T}$. Consider a μ-calculus environment $\widehat{\mathcal{E}} : Var \to T$ for $\mathcal{T}$; for any two bisimilar nodes t_1, t_2 in $\mathcal{T}$, we can assume that $t_1 \in \widehat{\mathcal{E}}$ iff $t_2 \in \widehat{\mathcal{E}}$. Now note that even if we prevent our environments from containing nodes not of the form $t.\mathbf{s}$ (i.e., if we remove every "nonsummary" node from each of the fixpoint sets as they are computed), the semantics of φ at "summary" nodes of form $t.\mathbf{s}$ (and hence at the root of $\mathcal{T}$) is not affected. This can be verified by a simple induction. We will assume environments for $\mathcal{T}$ to have this property from now on. Now consider the NT-μ environment $\mathcal{E} : Var \to Summ_k^{\mathcal{S}}$ defined as: for all $X \in Var$, $\mathcal{E}(X) = \{\mathbf{s} : \mathbf{s}$ is a summary, and $t.\mathbf{s} \in \widehat{\mathcal{E}}(X)$ for some $t\}$.

Our inductive hypothesis is: for any environment $\widehat{\mathcal{E}}$ in $\mathcal{T}$, any node $t.\mathbf{s}$ in $\mathcal{T}$ where $\mathbf{s}$ is a summary, and any $\varphi \in \Phi(TT^{at})$, $\mathbf{s} \in [\![Nest(\varphi)]\!]_{\mathcal{E}}$ iff $t.\mathbf{s}, \widehat{\mathcal{E}} \models \varphi$.

Cases (1), (2), (4), (5), (6), (7), (10), and (11) are easy. For the case $\varphi = X$ (case 3), recall that $\widehat{\mathcal{E}}$ is bisimulation-closed. For cases (8) and (9) (the call-clauses), note that the branching of $\mathcal{T}$ exactly captures the semantics of the call clauses of NT-μ. For the case $\varphi = \mu X.\varphi'$ (the case $\nu X.\varphi'$ is similar), assume that T_i and BS_i are the i-th sets in the fixpoint computation in φ and $Nest(\varphi)$ (we can assume that T_i is bisimulation-closed). We show that $BS_i = \{\mathbf{s} : t.\mathbf{s} \in T_i\}$ by induction. Initially, $T_0 = \emptyset$ and $BS_0 = \emptyset$. Now, $T_{i+1} = \{t : t, \widehat{\mathcal{E}}[X := T_i] \models \varphi'\}$, and $BS_{i+1} = [\![Nest(\varphi')]\!]_{\mathcal{E}[X:=BS_i}$. It is easy to see that $BS_{i+1} = \{\mathbf{s} : t.\mathbf{s} \in T_{i+1}\}$. It is also easily seen that $Nest(\varphi)$ is linear in the size of φ. This establishes the theorem. □

Corollaries. Theorems 7.1 and 7.2 imply a few results for NT-μ that we have not already derived. Let us define the *satisfiability problem* for NT-μ as the problem of determining, given an NT-μ formula φ, if $\mathcal{L}(\varphi) = \emptyset$. Then, we have the following theorem.

THEOREM 7.7. *Given a* NT-μ *formula, the problem of checking whether there is some nested tree that satisfies it is undecidable.*

PROOF. By Theorem 6.4, determining if $\mathcal{L}(\mathcal{A}) = \emptyset$ is undecidable. However, by Theorem 7.2, for every APNTA $\mathcal{A}$ there is an NT-μ formula $\varphi_{\mathcal{A}}$ such that $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\varphi_{\mathcal{A}})$. Thus, if we could determine, given an NT-μ formula φ, if $\mathcal{L}(\varphi) = \emptyset$, then we could solve the emptiness problem for APNTAs. This means the satisfiability problem for NT-μ is undecidable. □

The automata-theoretic characterization also makes it easy for us to compare the expressiveness of NT-μ with that of other temporal logics. As APNTAs are more expressive than alternating parity tree automata, which are polynomially interconvertible with closed μ-calculus formulas, we have the following.

THEOREM 7.8. *Any closed μ-calculus formula φ may be converted into an equivalent* NT-μ *formula φ'. The size of φ' is polynomial in the size of φ.*

We may now also relate NT-μ to the temporal logic CARET [Alur et al. 2004], which can express many linear-time context-sensitive properties of programs. This is because any CARET formula may be translated to an exponentially larger nondeterministic finite automaton on nested words. As such automata form a subclass of APNTAs, we have the following.

THEOREM 7.9. *Any* CARET *formula φ may be translated to an equivalent* NT-μ *formula φ'. The size of φ' is at worst exponential in the size of φ.*

7.2. A Hierarchy Theorem for N_{T}-μ

Now we show that the expressiveness of NT-μ formulas increases with their arity. For two nested trees $\mathcal{T}_1$ and $\mathcal{T}_2$, respectively, rooted at nodes s_1 and s_2, we say $\mathcal{T}_1$ and $\mathcal{T}_2$ are *distinguished* by a closed, k-ary NT-μ formula φ if and only if s_1 satisfies φ and s_2 does not. Then we have the following theorem.

THEOREM 7.10. *For every $k \geq 1$, there is a a closed $(k+1)$-ary formula φ_{k+1}, and two nested trees $\mathcal{T}_1$ and $\mathcal{T}_2$, such that φ_{k+1} can distinguish between $\mathcal{T}_1$ and $\mathcal{T}_2$, but no closed k-ary* NT-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 NT-μ respects this preorder.

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 NT-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 7.10. Consider the two nonbisimilar nested trees $\mathcal{S}$ and $\mathcal{T}$ in Figure 12 with initial nodes s_0 and t_0 (the jump-edges are not shown, but can be reconstructed from the edge labeling). It is easy to see that the 2-ary NT-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

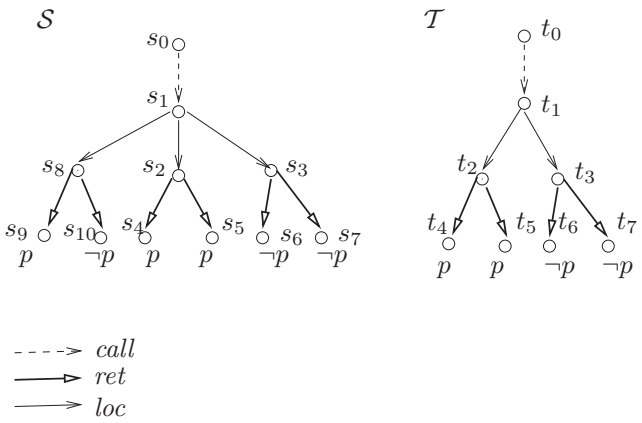

Fig. 12. An arity hierarchy.

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.

8. CONCLUSION

In this article, we have offered a new framework for software model checking of procedural programs based on *regular languages of nested trees*. Temporal specification logics like the μ-calculus, while mainstays of traditional model checking, cannot specify program properties that refer to the nesting of procedural contexts. Our fixpoint logic NT-μ based on nested trees, on the other hand, can express context-sensitive program requirements such as pre/post-conditions, combinations of local and global temporal properties, and stack-sensitive security requirements, while admitting tractable, symbolic model checking. The logic unifies and generalizes many existing logics and fixpoint computations, identifying a new class of decidable properties of programs. We have further explored the theory of regular languages of nested trees through automata on nested trees. We have shown that these languages have many of the attractive properties of tree automata— for example, they are closed under logical operations and allow model checking— while being more expressive. In a result that "lifts" the equivalence between the μ-calculus and alternating parity tree automata and suggests that NT-μ is a canonical calculus for these structures, we have shown NT-μ and alternating parity automata on nested trees are interconvertible.

Regarding future directions and open questions, we note that our decision procedure for model checking NT-μ is very different from known methods for branching-time model-checking of pushdown models [Walukiewicz 2001; Burkart and Steffen 1999]. The latter seem too complex to work in practice; our algorithm, being symbolic in nature, appears more implementable. Also, note that our algorithm directly implements the operational semantics of NT-μ formulas over bounded summaries. In fact, in this regard NT-μ resembles the modal μ-calculus in the setting of finite-state systems 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. There is one key

difference, however: in the latter setting, efficient data structures such as binary decision diagrams (BDDs) are used to represent sets of system states. On the other hand, we do not yet know of an efficient data structure to store sets of bounded summaries. Note that the problem is more complex than that of representing sets of states; in our case, we need to store sets of tuples of the form $\langle v, U_1, \ldots, U_k \rangle$, where v is a state and the V_i-s are sets of states. We leave this as an open question.

Also of interest will be an axiomatization of NT-μ and its applications to the verification of general programs (where data is not abstracted out). Also, note that NT-μ expresses properties using *forward* modalities. As argued in Schmidt [1998], several program analysis problems also require backward modalities; extending NT-μ to backward modalities will result in expressing several other dataflow problems. How will this affect expressiveness and decidability? In general, we believe that nested trees are conceptually fundamental and merit further study. It would be interesting to examine languages of nested trees in other ways: do they, for example, have algebraic characterizations? Also, can it have applications of nested trees beyond program verification? Nested word structures are already known to have connections with XML query languages, since XML documents have a natural matching tag structure that can be modeled by jump-edges. Do nested trees have similar applications?

REFERENCES

ABADI, M. AND FOURNET, C. 2003. Access control based on execution history. In *Proceedings of the Network and IT Security Conference (NDSS)*.

ALUR, R., BENEDIKT, M., ETESSAMI, K., GODEFROID, P., REPS, T., AND YANNAKAKIS, M. 2005. Analysis of recursive state machines. *ACM Trans. Prog. Lang. Syst. 27,* 4, 786–818.

ALUR, R., CHAUDHURI, S., AND MADHUSUDAN, P. 2006a. A fixpoint calculus for local and global program flows. In *Proceedings of the 33rd Annual ACM Symposium on Principles of Programming Languages*.

ALUR, R., CHAUDHURI, S., AND MADHUSUDAN, P. 2006b. Languages of nested trees. In *Proceedings of the Symposium on Computer-Aided Verification (CAV'06)*.

ALUR, R., ETESSAMI, K., AND MADHUSUDAN, P. 2004. A temporal logic of nested calls and returns. In *Proceedings of the 10th International Conference on Tools and Algorithms for the Construction and Analysis of Software*. Lecture Notes in Computer Science, vol. 2988. Springer, 467–481.

ALUR, R. AND MADHUSUDAN, P. 2004. Visibly pushdown languages. In *Proceedings of the 36th ACM Symposium on Theory of Computing*. 202–211.

ALUR, R. AND MADHUSUDAN, P. 2006. Adding nesting structure to words. In *Proceedings of the Symposium on Developments in Language Theory*.

ALUR, R. AND MADHUSUDAN, P. 2009. Adding nesting structure to words. *J. ACM 56,* 3.

BALL, T. AND RAJAMANI, S. 2000. Bebop: A symbolic model checker for boolean programs. In *Proceedings of the Workshop on Model Checking of Software*. Lecture Notes in Computer Science, vol. 1885. Springer, 113–130.

BALL, T. AND RAJAMANI, S. 2001. The SLAM toolkit. In *Proceedings of the 13th International Conference on Computer Aided Verification*.

BURDY, L., CHEON, Y., COK, D., ERNST, M., KINIRY, J., LEAVENS, G., LEINO, R., AND POLL, E. 2003. An overview of JML tools and applications. In *Proceedings of the 8th International Workshop on Formal Methods for Industrial Critical Systems*. 75–89.

BURKART, O. AND STEFFEN, B. 1999. Model checking the full modal mu-calculus for infinite sequential processes. *Theoret. Comput. Sci. 221*, 251–270.

CLARKE, E., GRUMBERG, O., AND PELED, D. 1999. *Model Checking*. MIT Press. CLA e 99:1 1.Ex.

EMERSON, E. AND CLARKE, E. 1982. Using branching-time temporal logic to synthesize synchronization skeletons. *Sci. Comput. Prog. 2*, 241–266.

EMERSON, E. AND JUTLA, C. 1991. Tree automata, mu-calculus, and determinacy. In *Proceedings of the 32nd IEEE Symposium on Foundations of Computer Science*. 368–377.

EMERSON, E. AND LEI, C. 1985. Modalities for model-checking: Branching time logic strikes back. In *Proceedings of the 12th ACM Symposium on Principles of Programming Languages*. 84–96.

ESPARZA, J., KUCERA, A., AND SCHWOON, S. S. 2003. Model-checking LTL with regular valuations for pushdown systems. *Inf. Computation 186,* 2, 355–376.

GRÄDEL, E., THOMAS, W., AND WILKE, T., Eds. 2002. *Automata, Logics, and Infinite Games: A Guide to Current Research*. Lecture Notes in Computer Science, vol. 2500. Springer.

HOARE, C. 1969. An axiomatic basis for computer programming. *Comm. ACM 12,* 10, 576–580.

HOPCROFT, J. AND ULLMAN, J. 1979. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley.

JENSEN, T., METAYER, D. L., AND THORN, T. 1999. Verification of control flow based security properties. In *Proceedings of the IEEE Symposium on Security and Privacy*. 89–103.

KOZEN, D. 1983. Results on the propositional mu-calculus. *Theoret. Comput. Sci. 27*, 333–354.

KUPFERMAN, O., PITERMAN, N., AND VARDI, M. 2002. Pushdown specifications. In *Proceedings of the 9th International Conference on Logics for Programming, Artifical Intelligence, and Reasoning*. Lecture Notes in Computer Science, vol. 2514. Springer, 262–277.

KUPFERMAN, O., VARDI, M., AND WOLPER, P. 2000. An automata-theoretic approach to branching-time model checking. *J. ACM 47,* 2, 312–360.

REPS, T. 1998. Program analysis via graph reachability. *Inf. Softw. Tech. 40,* 11-12, 701–726.

REPS, T., HORWITZ, S., AND SAGIV, S. 1995. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the ACM Symposium on Principles of Programming Languages*. 49–61.

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

SHARIR, M. AND PNUELI, A. 1981. Two approaches to interprocedural dataflow analysis. In *Program Flow Analysis: Theory and Applications*, 189–234.

STEFFEN, B. 1991. Data flow analysis as model checking. In *Proceedings of the Symposium on Theoretical Aspects of Computer Software (TACS'91)*. Lecture Notes in Computer Science, vol. 526. 346–365.

WALLACH, D. S. AND FELTEN, E. W. 1998. Understanding Java stack inspection. In *Proceedings of the IEEE Symposium on Security and Privacy*. 52–63.

WALUKIEWICZ, I. 2001. Pushdown processes: Games and model-checking. *Inf. Comput. 164,* 2, 234–263.

Received November 2010; accepted April 2011