Adding Nesting Structure to Words

Rajeev Alur[1] and P. Madhusudan[2]

[1] University of Pennsylvania, USA
[2] University of Illinois at Urbana-Champaign, USA

1 Introduction

We propose *nested words* to capture models where there is *both* a natural linear sequencing of positions and a hierarchically nested matching of positions. Such dual structure exists for executions of structured programs where there is a natural well-nested correspondence among entries to and exits from program components such as functions and procedures, and for XML documents where each open-tag is matched with a closing tag in a well-nested manner.

We define and study finite-state automata as acceptors of nested words. A nested-word automaton is similar to a classical finite-state word automaton, and reads the input from left to right according to the linear sequence. However, at a position with two predecessors, one due to linear sequencing and one due to a hierarchical nesting edge, the next state depends on states of the run at both these predecessors. The resulting class of *regular* languages of nested words has all the appealing theoretical properties that the class of classical regular word languages enjoys: deterministic nested word automata are as expressive as their nondeterministic counterparts; the class is closed under operations such as union, intersection, complementation, concatenation, and Kleene-$*$; decision problems such as membership, emptiness, language inclusion, and language equivalence are all decidable; definability in monadic second order logic of nested words corresponds exactly to finite-state recognizability; and finiteness of the congruence induced by a language of nested words is a necessary and sufficient condition for regularity.

The motivating application area for our results has been software verification. Given a sequential program P with stack-based control flow, the execution of P is modeled as a nested word with nesting edges from calls to returns. Specification of the program is given as a nested word automaton A, and verification corresponds to checking whether every nested word generated by P is accepted by A. Nested-word automata can express a variety of requirements such as stack-inspection properties, pre-post conditions, and interprocedural data-flow properties. If we were to model program executions as words, all of these properties are non-regular, and hence inexpressible in classical specification languages based on temporal logics, automata, and fixpoint calculi (recall that context-free languages cannot be used as specification languages due to nonclosure under intersection and undecidability of key decision problems such as language inclusion). In finite-state software model checking, the data variables in the program are abstracted into a set of boolean variables, and in that case, the set of nested

words generated by the abstracted program is regular. This implies that algorithmic software verification is possible for all regular specifications of nested words. We believe that the nested-word view will provide a unifying basis for the next generation of specification logics for program analysis, software model checking, and runtime monitoring. As explained in Section 3, another potential area of application is XML document processing.

Related Work

The finite automata on nested words that we study here have been motivated by our recent work on *visibly pushdown automata* [6]. A visibly pushdown automaton is one in which the input alphabet Σ is partitioned into three parts, $\langle \Sigma_c, \Sigma_i, \Sigma_r \rangle$ such that the automaton pushes exactly one symbol when reading symbols from Σ_c, pops one symbol from the stack when reading a symbol in Σ_r, and does not touch the stack when reading letters of Σ_i. The input word hence has an implicit nesting structure defined by matching occurrences of symbols in Σ_c with symbols in Σ_r. In nested words, this nesting is given explicitly, and this lets us define an automaton without a stack [1]. We believe that nested words is a more appealing and simpler formulation of the insights in the theory of visibly pushdown languages. However, in terms of technical results, this paper only reformulates the corresponding results for visibly pushdown languages in [6].

Visibly pushdown languages are obviously related to Dyck languages, which is the class of languages with well-bracketed structure. The class of *parenthesis* languages studied by McNaughton comes closest to our notion of visibly pushdown languages [16]. A parenthesis language is one generated by a context free grammar where every production introduces a pair of parentheses that delimit the scope of the production. Viewing the nesting relation as that defined by the parentheses, parenthesis languages are a subclass of visibly pushdown languages. In [16, 11], it was shown that parenthesis languages are closed under union, intersection and difference, and that the equivalence problem for them is decidable. However, parenthesis languages are a strict subclass of visibly pushdown languages, and are not closed under Kleene-$*$.

The class of visibly pushdown languages, was considered in papers related to parsing *input-driven languages* [22, 9]. Input-driven languages are precisely visibly pushdown languages (the stack operations are *driven* by the input). However, the papers considered only the membership problem for these languages (namely showing that membership is easier for these languages than for general context-free languages) and did not systematically study the *class* of languages defined by such automata.

[1] It is worth noting that most of the algorithms for inter-procedural program analysis and context-free reachability compute summary edges between control locations to capture the computation of the called procedure (see, for example [18]).

2 Nested Words

Definition

A *nested relation* ν of width k, for $k \geq 0$, is a binary relation over $\{1, 2 \dots k\}$ such that (1) if $\nu(i, j)$ then $i < j$; (2) if $\nu(i, j)$ and $\nu(i, j')$ then $j = j'$, and if $\nu(i, j)$ and $\nu(i', j)$ then $i = i'$; (3) if $\nu(i, j)$ and $\nu(i', j')$ and $i < i'$ then either $j < i'$ or $j' < j$.

Let ν be a nested relation. When $\nu(i, j)$ holds, the position j is called a *return-successor* of the position i, and the position i is called a *call-predecessor* of the position j. Our definition requires that a position has at most one return-successor and at most one call-predecessor, and a position cannot have both a return-successor and a call-predecessor. A position is called a *call* position if it has a return successor, a *return* position if it has a call-predecessor, and an *internal* position otherwise.

A *nested word* nw over an alphabet Σ is a pair $(a_1 \dots a_k, \nu)$, for $k \geq 0$, such that a_i, for each $1 \leq i \leq k$, is a symbol in Σ, and ν is a nested relation of width k. Let us denote the set of nested words over Σ as $NW(\Sigma)$. A language of nested words over Σ is a subset of $NW(\Sigma)$.

Example: Program Executions as Nested Words

Execution of a program is typically modeled as a word over an alphabet Σ. The choice of Σ depends on the desired level of detail. As an example, suppose we are interested in tracking read/write accesses to a program variable x. The variable x may get redefined, for example, due to a declaration of a local variable within a called procedure, and we need to track the scope of these definitions. For simplicity, let's assume every change in context redefines the variable. Then, we can choose the following set of symbols: *rd* to denote a read access to x, *wr* to denote a write access to x, *en* to denote beginning of a new scope (such as a call to a function or a procedure), *ex* to denote the ending of the current scope, and *sk* to denote all other actions of the program. Note that in any structured programming language, in a given execution, there is a natural nested matching of the symbols *en* and *ex*. Figure 1 shows a possible execution as a word as well as a nested word. The nesting edges are shown as dotted edges. A vertical path can be interpreted as a local path through a procedure. There is a natural connection between nested words and binary trees, and is also depicted in Figure 1. In this view, at a call node, the left subtree encodes the computation within the called procedure, while a path along the right children gives the local computation within a procedure.

In modeling the execution as a word, the matching between calls and returns is only implicit, and a pushdown automaton is needed to reconstruct the matching. The tree view makes the hierarchical structure explicit: every matching exit is a right-child of the corresponding entry node. However, this view loses linearity: the left and right subtrees of an entry node are disconnected, and (top-down)

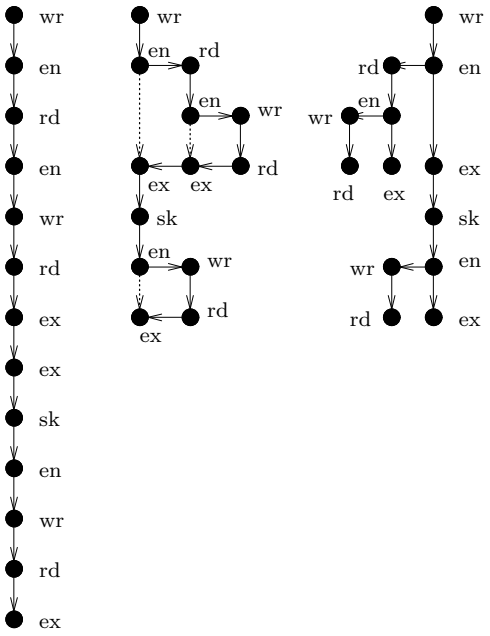

Fig. 1. Execution as a word, as a nested word, and as a tree

tree automata need nondeterminism to relate the properties of the subtrees [2]. Our hypothesis is that the nested-word view is the most suitable view for program verification. In this view, a program will be a *generator* of nested words, and will be modeled as a language of nested words. For acceptors, linearity is used to obtain a left-to-right deterministic acceptor, while nesting is exploited to keep the acceptor finite state.

Operations on Nested Words

Analogs of a variety of operations on words and word languages can be defined for nested words and corresponding languages. We describe a few of the interesting ones here.

Given two nested words $nw_1 = (w_1, \nu_1)$ and $nw_2 = (w_2, \nu_2)$, of lengths k_1 and k_2, respectively, the *concatenation* of nw_1 and nw_2 is the nested word $nw_1.nw_2 = (w_1.w_2, \nu)$ of length $k_1 + k_2$, where ν is the nested relation $\nu_1 \cup \{(k_1 + i, k_1 + j) \mid (i, j) \in \nu_2\}$. The concatenation extends to languages of nested words.

[2] It is worth mentioning that in program verification, trees are used for a different purpose: an execution tree encodes all possible executions of a program, and branching corresponds to the choice within the program. It is possible to define *nested trees* in which each path encodes a structured execution as a nested word [3].

The Kleene-$*$ operation is defined as usual: if L is a language of nested words over Σ, then L^* is the set of nested words $nw_1.nw_2 \dots nw_i$, where $i \in \mathbb{N}$, and each $w_j \in L$.

Given a nested word $nw = (a_1 \dots a_k, \nu)$ of length k, its *reverse* is $nw^r = (a_k \dots a_1, \nu^r)$ where $\nu^r = \{(i,j) \mid (k+1-j, k+1-i) \in \nu\}$.

Finally, we define a notion of *insertion* for nested words. A *context* is a pair (nw, i) where nw is a nested word of length k, and $0 \leq i \leq k$. Given a context (nw, i), for $nw = (a_1 \dots a_k, \nu)$, and a nested word nw', with $nw' = (w', \nu')$, $(nw, i) \oplus nw'$ is the nested word obtained by inserting the nested word nw' at position i in nw. More precisely, $(nw, i) \oplus nw'$ is the nested word $(a_1 \dots a_i.w'.a_{i+1} \dots a_k, \nu'')$, where $\nu'' = \{(\pi_1(j), \pi_1(j')) \mid (j,j') \in \nu\} \cup \{(\pi_2(j), \pi_2(j')) \mid (j,j') \in \nu'\}$ where $\pi_1(j)$ is j, if $j \leq i$, and $|w'|+j$ otherwise, and $\pi_2(j) = i + j$.

Note that our definition of nested word requires one-to-one matching between call and return positions. It is possible to generalize this definition and allow a nested relation to contain pairs of the form $(i, \perp)$ and $(\perp, j)$ corresponding to unmatched call and return positions, respectively. Concatenation of two nested words would match the last unmatched call in the first word with the first unmatched return in the second one. Natural notions of prefix and suffix exist in this generalized definition. The results of this paper can be adapted to this general definition also.

3 Regular Languages of Nested Words

Automata over Nested Words

A *nested word automaton* (NWA) A over an alphabet Σ is a structure (Q, Q_{in}, δ, Q_f) consisting of

- a finite set Q of states,
- a set of initial states $Q_{in} \subseteq Q$,
- a set of final states $Q_f \subseteq Q$,
- a set of transitions $\langle \delta_c, \delta_r, \delta_i \rangle$ where
  - $\delta_c \subseteq Q \times \Sigma \times Q$ is a transition relation for call positions, and
  - $\delta_i \subseteq Q \times \Sigma \times Q$ is a transition relation for internal positions, and
  - $\delta_r \subseteq Q \times Q \times \Sigma \times Q$ is a transition relation for return positions.

The automaton A starts in an initial state, and reads the word from left to right. At a call or an internal position, the next state is determined by the current state and the input symbol at the current position, while at a return position, the next state can additionally depend on the state of the run just before the matching call-predecessor. Formally, a *run* ρ of the automaton A over a nested word $nw = (a_1 \dots a_k, \nu)$ is a sequence $q_0, \dots, q_k$ over Q such that $q_0 \in Q_{in}$, and for each $1 \leq i \leq k$,

- if i is a call position of ν, then $(q_{i-1}, a_i, q_i) \in \delta_c$;

– if i is a internal position, then $(q_{i-1}, a_i, q_i) \in \delta_i$;
– if i is a return position with call-predecessor is j, then $(q_{i-1}, q_{j-1}, a_i, q_i) \in \delta_r$.

The automaton A accepts the nested word nw if it has a run $q_0, \ldots, q_k$ over nw such that $q_k \in Q_f$. The language $L(A)$ of a nested-word automaton A is the set of nested words it accepts.

A language L of nested words over Σ is *regular* if there exists a nested-word automaton A over Σ such that $L = L(A)$.

Observe that if L is a regular language of words over Σ, then $\{(w, \nu) \mid w \in L\}$ is a regular language of nested words. Conversely, if L is a regular language of nested words, then $\{w \mid (w, \nu) \in L \text{ for some } \nu\}$ is a context-free language of words, but need not be regular.

The fact that a nested automaton at a return position can look at the state at the corresponding call position is, of course, crucial to expressiveness as it allows a run to implicitly encode a stack. In automata theory, such a definition that allows combining of states is quite common. For example, bottom-up tree automata allow such a join. Various notions of automata on partial-orders and graphs are also defined this way [20]. In fact, one can define a more general notion of automata on nested words by giving *tiling systems* that tile the positions using a finite number of tiles with local constraints that restrict the tiles that can occur at a position, given the tiles in its neighborhood. The notion of neighborhood for a node in a nested word would be its linear successor and predecessor, and its return-predecessor or call-successor. It turns out that automata defined in this fashion also define regular nested languages.

Determinization

A nested-word automaton $A = (Q, Q_{in}, (\delta_c, \delta_i, \delta_r), Q_f)$ is said to be *deterministic* if $|Q_{in}| = 1$, and for every $a \in \Sigma$ and $q, q' \in Q$, $|\{q'' \mid (q, a, q'') \in \delta_c\}| = 1$ and $|\{q'' \mid (q, a, q'') \in \delta_i\}| = 1$ and $|\{q'' \mid (q, q', a, q'') \in \delta_r\}| = 1$. Thus, a deterministic nested-word automaton has a single initial state, and the transition relations δ_c and δ_i are functions from $Q \times \Sigma$ to Q, and the transition relation δ_r is a function from $Q \times Q \times \Sigma$ to Q. Given a nested word nw, a deterministic nested-word automaton has exactly one run over nw.

Adapting the classical subset construction for determinizing finite automata over words turns out to be slightly tricky, but possible:

Theorem 1. *Given a nested-word automaton A over Σ, there exists a deterministic nested-word automaton A' over Σ such that $L(A) = L(A')$. Furthermore, if A has n states, then A' has at most 2^{n^2} states.*

Proof. The deterministic automaton will keep track of *summaries* of state-transitions, rather than just the states reached. More precisely, after reading the first i positions of a nested word $nw = (w, \nu)$, if j is the last call position at or before i (if there is none, choose $j = 1$), then the automaton will be in a state $S \subseteq Q \times Q$ where S is the set of pairs of states (q, q') such that A has a

run from q to q' on reading the nested word starting at position j to i. It hence starts in the initial state $\{(q,q) \mid q \in Q\}$. At an internal position labeled a, the automaton replaces each pair (q,q') in the current state by pairs of the form (q,q'') such that $(q',a,q'') \in \delta_i$. At a call position labeled a, the summary gets reinitialized: the new state contains pairs of the form (q,q'), where $(q,a,q') \in \delta_c$. Consider a return position labeled a, and suppose S denotes the current state and S' denotes the state just before the call-predecessor. Then (q,q') belongs to the new state, provided there exist states q_1, q_2 such that $(q,q_1) \in S'$ and $(q_1,q_2) \in S$ and $(q_2,q_1,a,q') \in \delta_r$. A state is final if it contains a pair of the form (q,q') with $q \in Q_{in}$ and $q' \in Q_f$. □

Since the call and internal transition relations are separate, our definition allows the automaton to check whether the current position is a call or an internal position. It is easy to verify that this distinction is not necessary for nondeterministic automata. However, for deterministic automata, removing this distinction will reduce expressiveness. On the other hand, as the above proof shows, a deterministic NWA can accept all regular languages of nested words, even if we restrict the call transition function to depend only on the current symbol.

Closure Properties

The class of regular nested languages enjoy many closure properties, similar to the class of regular languages over words.

Theorem 2. *The class of regular languages of nested words is (effectively) closed under union, intersection, complementation, concatenation, Kleene-$*$, and reverse.*

Application: Software Model Checking and Program Analysis

In the context of software verification, a popular paradigm to verification is through *data abstraction*, where the data in a program is abstracted using a finite set of boolean variables that stand for predicates on the data-space [7, 10]. The resulting models hence have finite-state but stack-based control flow (see Boolean programs [8] and recursive state machines [1] as concrete instances of pushdown models of programs). Given a program P modeled as a pushdown automaton, we can view P as a generator of nested words in the following manner. We choose an observation alphabet Σ, and associate an element of Σ with every transition of P. At every step of the execution of P, if the transition of P is a *push* transition, then the corresponding position is a call position; if the transition of P does not update the stack, then the corresponding position is an internal position; and if P executes a *pop* transition, then the corresponding position is a return, with a nesting edge from the position where the corresponding element was pushed. We assume that P pushes or pops at most one element, and halts when the stack is empty. Then, the nesting edges satisfy the desired constraints. Let $L(P)$ be the

set of nested words generated by a pushdown model P. Then, $L(P)$ is a regular language of nested words.

The requirements of a program can also be described as regular languages of nested words. For instance, consider the example of Section 2. Suppose we want to specify that within each scope (that is, between every pair of matching entry and exit), along the local path (that is, after deleting every enclosed matching subword from an entry to an exit), every write access is followed by a read access. Viewed as a property of words, this is not a regular language, and thus, not expressible in the specification languages supported by existing software model checkers such as SLAM [8] and BLAST [10]. However, over nested words, there is a natural two-state deterministic nested-word automaton. The initial state is q_0, and has no pending obligations, and is the only final state. The state q_1 denotes that along the local path of the current scope, a write-access has been encountered, with no following read access. The transitions are: for $j = 0, 1$, $\delta_i(q_j, rd) = q_0$; $\delta_i(q_j, wr) = q_1$; $\delta_i(q_j, sk) = q_j$; $\delta_c(q_j, en) = q_0$; and $\delta_r(q_0, q_j, ex) = q_j$. The automaton reinitializes the state to q_0 upon entry, while processing internal read/write symbols, it updates the state as in a finite-state word automaton, and at a return, if the current state is q_0 (meaning the called context satisfies the desired requirement), it restores the state of the calling context. (Formally, we need one more state q_3 in order to make the automaton complete; when in state q_1 and reading a return, the automaton will go to state q_3, and all transitions from q_3 will go to q_3.)

Further, we can build *specification* logics for programs that exploit the nested structure. An example of such a temporal logic is CARET [4], which extends linear temporal logic by *local* modalities such as $\langle a \rangle \varphi$, which holds at a call if the return-successor of the call satisfies φ. CARET can state many interesting properties of programs, including stack-inspection properties, pre-post conditions of programs, local flows in programs, etc. Analogous to the theorem that a linear temporal formula can be compiled into an automaton that accepts its models [21], any CARET formula can be compiled into a nested word automaton that accepts its models. Decidability of inclusion then yields a decidable model-checking problem for program models against CARET [6, 4].

Application: XML Document Processing

Turning to XML, XML documents (which resemble HTML documents in structure) are hierarchically structured data with open- and close-tag constructs used to define the hierarchy. An XML document is naturally a nested word, where each open-tag is matched with its corresponding closing tag. Document type definitions (DTDs) and their specialized counterparts (SDTDs) are used to define classes of documents, using a grammar. The grammar however is special in that the non-terminals always stand for tags. Consequently, type definitions can be encoded using nested word automata. Though trees and automata on unranked trees are traditionally used in the study of XML (see [17, 14] for recent surveys), nested word automata lend more naturally to describing the document especially when the document needs to be processed as a word being read from left

to right (as in the case of processing streaming XML documents). The closure and determinization theorems for nested word automata have immediate consequences in checking type-inclusion and in checking streaming XML documents against SDTDs. Further, minimization theorems for nested word automata can be exploited to construct minimal machines to process XML documents [13].

4 Alternative Characterizations

We now show alternate characterizations of the class of regular nested word languages.

Monadic Second Order Logic of Nested Words

Let us fix a countable set of first-order variables FV and a countable set of monadic second-order (set) variables SV. We denote by x, y, x', etc., elements in FV and by X, Y, X', etc., elements of SV.

The *monadic second-order logic of nested words* is given by the syntax:

$$\varphi := Q_a(x) \mid x = y \mid x \leq y \mid \nu(x, y) \mid \varphi \vee \varphi \mid \neg\varphi \mid \exists x.\varphi \mid \exists X.\varphi,$$

where $a \in \Sigma$, $x, y \in FV$, and $X \in SV$.

The semantics is defined over nested words in a natural way. The first-order variables are interpreted over positions of the nested word, while set variables are interpreted over sets of positions. $Q_a(x)$ holds if the letter at the position interpreted for x is a, $x \leq y$ holds if the position interpreted for x is before the position interpreted for y, and $\nu(x, y)$ holds if the positions x and y are ν-related. For example,

$$\forall x.\forall y.\ (Q_a(x) \wedge \nu(x, y)) \Rightarrow Q_b(y)$$

holds in a nested word iff for every call labeled a, the corresponding return-successor is labeled b.

For a sentence φ (a formula with no free variables), the language it defines is the set of all nested words that satisfy φ. The corresponding result for visibly pushdown languages [6] can be used to show that:

Theorem 3. *A language L of nested words over Σ is regular iff there is an MSO sentence φ over Σ that defines L.*

Finite Congruence

Let L be a language of nested words. Define the following relation on nested words. For two nested words nw_1 and nw_2, $nw_1 \sim_L nw_2$ if for every context (nw, i), $(nw, i) \oplus nw_1 \in L$ iff $(nw, i) \oplus nw_2 \in L$. Note that $\sim_L$ is an equivalence relation and a congruence (i.e. if $nw_1 \sim_L nw_2$ and $nw_1' \sim_L nw_2'$, then $nw_1.nw_1' \sim_L nw_2.nw_2'$). We can now show that the finiteness of this congruence characterizes regularity for nested-word languages using the corresponding result for visibly pushdown languages [5].

Theorem 4. *For a set L of nested words, L is regular iff $\sim_L$ has finitely many congruence classes.*

Proof. Let L be a regular language of nested words. Let A be a NWA that accepts L, and let its set of states be Q. Now, define the following relation on nested words: $nw \approx_A nw'$ if for every $q, q' \in Q$, A has a run from q to q' on nw if and only if A has a run from q to q' on nw'. It is easy to verify that $\approx_A$ is an equivalence relation and, in fact, a congruence. Clearly there are no more than $|Q|^2$ congruence classes defined by it. Also, it is easy to see that whenever $nw \approx_A nw'$, it is the case that $nw \sim_L nw'$. It follows that $\sim_L$ is of finite index.

For the converse, assume L is such that $\sim_L$ is of finite index, and let us denote by $[nw]$ the equivalence class of $\sim_L$ that a nested word nw belongs to. Then let $A = (Q, Q_{in}, \delta, Q_f)$, where:

- $Q = \{[nw] \mid nw \textit{ is a nested word}\}$,
- $Q_{in} = \{[nw_0]\}$, where nw_0 is the empty nested word,
- $Q_f = \{[nw] \mid nw \in L\}$, and
- $\delta = \langle \delta_c, \delta_i, \delta_r \rangle$ where
  - $\delta_c = \{([nw], a, [(a, \emptyset)]) \mid nw \in NW(\Sigma), a \in \Sigma\}$
  - $\delta_i = \{([nw], a, [nw.a]) \mid nw \in NW(\Sigma), a \in \Sigma\}$
  - $\delta_r([(w_1, \nu_1)], [(w_2, \nu_2)], a) = [w_2.w_1.a, \nu]$, where, if $(w_2, \nu_2).(w_1, \nu_1).a = (w_1.w_2, \nu'$, then $\nu = \nu' \cup \{(|w_2| + 1, |w_2| + |w_1| + 1)\}$.

It can then be proved that A is well-defined and accepts L. □

Visibly Pushdown Word Languages

A nested word over Σ can be encoded as a word over a finite *structured* alphabet in the following manner. Let $\Sigma' = \{c, int, r\} \times \Sigma$. Let the set of *well-matched* words over Σ' (denoted $WM(\Sigma)$) be the words generated by the grammar

$$W := \epsilon \mid (int, a)W \mid (c, a)W(r, a') \mid W.W,$$

for $a, a' \in \Sigma$. Given a nested word $(a_1 \dots a_k, \nu)$ over Σ, we will encode it as the well-matched word u over Σ' by setting $u = (x_1, a_1) \dots (x_k, a_k)$ where $x_i = c$ if i is a call, $x_i = r$ if i is a return, and $x_i = int$ otherwise. Let us call this mapping $nw2w : NW(\Sigma) \to WM(\Sigma)$. It is also clear that for every well-matched word over Σ', there is a unique nested word over Σ that corresponds to it. Consequently, we can treat languages of nested words over Σ as languages of words over the structured alphabet Σ'.

A finite automaton on nested words over Σ can be simulated by a *pushdown* automaton on the corresponding word over Σ'. The pushdown automaton would simply push the current state at each call position, and at return positions it would pop the state to retrieve the state at the corresponding call. Note that this pushdown automaton is restricted in that it pushes exactly one symbol when reading symbols of the form (c, a), pops the stack when reading symbols of the form (r, a), and does not touch the stack when reading symbols (int, a). This kind of pushdown automaton is called a *visibly pushdown automaton* [6]. The automaton accepts if it reaches a final state and the stack is empty.

Proposition 1. *A language L of nested words over Σ is regular iff $nw2w(L)$ is accepted by a visibly pushdown automaton over the structured alphabet Σ'.*

Regular Tree Languages

Given a nested word over Σ, we can associate it with a Σ-labeled (ranked) binary tree that represents the nested word, where each position in the word corresponds to a node of the tree. Further, the tree will encode the return position corresponding to a call right next to the call. See Figure 1 for an example of a tree-encoding of a nested word. Formally, we define the map from nested words to trees using the function $nw2t$ that maps nested words to sets of trees (we allow more than one tree to correspond to a nested word since we do not differentiate the left-child from a right-child when a node has only one child):

- For the empty nested word $nw = (\epsilon, \nu)$, $nw2t(nw)$ is the empty tree.
- For a nested word $nw = (a_1.w_1, \nu)$ such that the first position is internal, let nw_1 be the nested word corresponding to w_1. Then $nw2t(nw)$ is any tree whose root is labeled a_1, the root has one child, and the subtree at this child is in $nw2t(nw_1)$.
- For a nested word $nw = (a_1.w_1.a_2.w_2, \nu)$ such that $(1, |w_1| + 2) \in \nu$, let $nw_1 = (w_1, \nu_1)$ be the nested word corresponding to the w_1 portion, and $nw_2 = (w_2, \nu_2)$ be the nested word corresponding to the w_2 portion. Then $nw2t(nw)$ is any tree whose root is labeled a_1, the subtree rooted at its left-child is in $nw2t(nw_1)$, its right-child u is labeled a_2, and u has one child and the the subtree rooted at this child is in $nw2t(nw_2)$.

We can now show that the class of regular nested word languages precisely corresponds to regular languages of trees:

Theorem 5. *A language T of trees is a regular tree language iff the set of nested words $\{nw2t^{-1}(t) \mid t \in T\}$ is a regular nested word language.*

Note that the closure of nested languages under various operations as stated in Theorem 2 can be proved using this connection to regular tree languages. However, the determinization result (Theorem 1) does not follow from the theory of tree automata.

5 Decision Problems

The emptiness problem (given A, is $L(A) = \emptyset$?) and the membership problem (given A and nw, is $nw \in L(A)$?) for nested word automata are solvable since we can reduce it to the emptiness and membership problems for pushdown automata (using Proposition 1).

If the automaton A is *fixed*, then we can solve the membership problem in simultaneously linear time and linear space, as we can determinize A and simply simulate the word on A. In fact, this would be a *streaming* algorithm that uses at

most space $O(d)$ where d is the *depth* of nesting of the input word. A streaming algorithm is one where the input must be read left-to-right, and can be read only once. Note that this result comes useful in type-checking streaming XML documents, as the depth of documents is often not large [19, 13]. When A is fixed, the result in [22] exploits the visibly pushdown structure to solve the membership problem in logarithmic space, and [9] shows that membership can be checked using boolean circuits of logarithmic depth. These results lead to:

Theorem 6. *The emptiness problem for nested word automata is decidable in time $O(|A|^3)$.*

The membership problem for nested word automata, given A and w, can be solved in time $O(|A|^3.|w|)$. When A is fixed, it is solvable (1) in time $O(|w|)$ and space $O(d)$ (where d is the depth of the nesting in w) in a streaming setting; (2) in space $O(log\ |w|)$ and time $O(|w|^2.log\ |w|)$; and (3) by (uniform) Boolean circuits of depth $O(log\ |w|)$.

The inclusion problem (and hence the equivalence problem) for nested word automata is decidable. Given A_1 and A_2, we can check $L(A_1) \subseteq A_2$ by checking if $L(A_1) \cap \overline{L(A_2)}$ is empty, since regular nested languages are effectively closed under complement and intersection. It follows from the results in [6] that:

Theorem 7. *The inclusion and equivalence problems for nested word automata are* EXPTIME*-complete.*

6 Conclusions

Nested words allow capturing linear and hierarchical structure simultaneously, and automata over nested words lead to a robust class of languages with appealing theoretical properties. This theory offers a way of extending the expressiveness of specification languages supported in model checking and program analysis tools: instead of modeling a boolean program as a context-free language of words and checking regular properties, one can model both the program and the specification as regular languages of nested words.

The theory of regular languages of nested words is a reformulation of the theory of visibly pushdown languages by moving the nesting structure from labeling to the underlying shape. Besides the results reported here, many results already exist for visibly pushdown automata: visibly pushdown languages over infinite words have been studied in [6]; games on pushdown graphs against visibly pushdown winning conditions are decidable [15]; congruence based characterizations and minimization theorems for visibly pushdown automata exist [5]; and active learning, conformance testing, and black-box checking for visibly pushdown automata are studied in [12]. The nested structure on words can be extended to trees, and automata on nested trees are studied in [3, 2]. Finally, a version of the μ-calculus on nested structures has been defined in [3], and is shown to be more powerful than the standard μ-calculus, while at the same time remaining robust and tractable [3, 2].

Acknowledgments We thank Swarat Chaudhuri, Kousha Etessami, Viraj Kumar, Leonid Libkin, Christof Löding, Mahesh Viswanathan, and Mihalis Yannakakis for fruitful discussions related to this paper. This research was partially supported by ARO URI award DAAD19-01-1-0473 and NSF award CCR-0306382.

References

1. R. Alur, M. Benedikt, K. Etessami, P. Godefroid, T. Reps, and M. Yannakakis. Analysis of recursive state machines. *ACM Transactions on Programming Languages and Systems*, 27(4):786–818, 2005.
2. R. Alur, S. Chaudhuri, and P. Madhusudan. Automata on nested trees. Under submission, 2006.
3. R. Alur, S. Chaudhuri, and P. Madhusudan. A fixpoint calculus for local and global program flows. In *ACM POPL*, pages 153–165, 2006.
4. R. Alur, K. Etessami, and P. Madhusudan. A temporal logic of nested calls and returns. In *TACAS*, LNCS 2988, pages 467–481, 2004.
5. R. Alur, V. Kumar, P. Madhusudan, and M. Viswanathan. Congruences for visibly pushdown languages. In *ICALP*, LNCS 3580, pages 1102–1114, 2005.
6. R. Alur and P. Madhusudan. Visibly pushdown languages. In *ACM STOC*, pages 202–211, 2004.
7. T. Ball, R. Majumdar, T.D. Millstein, and S.K. Rajamani. Automatic predicate abstraction of C programs. In *AACM PLDI*, pages 203–213, 2001.
8. T. Ball and S. Rajamani. Bebop: A symbolic model checker for boolean programs. In *SPIN Workshop*, LNCS 1885, pages 113–130, 2000.
9. P. Dymond. Input-driven languages are in *log n* depth. *Inf. Process. Lett.*, 26(5):247–250, 1988.
10. T.A. Henzinger, R. Jhala, R. Majumdar, G.C. Necula, G. Sutre, and W. Weimer. Temporal-safety proofs for systems code. In *CAV*, LNCS 2404, 526–538, 2002.
11. D.E. Knuth. A characterization of parenthesis languages. *Information and Control*, 11(3):269–289, 1967.
12. V. Kumar, P. Madhusudan, and M. Viswanathan. Minimization, learning, and conformance testing of boolean programs. Under submission, 2006.
13. V. Kumar, P. Madhusudan, and M. Viswanathan. Visibly pushdown languages for XML. Technical Report UIUCDCS-R-2006-2704, UIUC, 2006.
14. L. Libkin. Logics for unranked trees: An overview. In *ICALP*, LNCS 3580, pages 35–50, 2005.
15. C. Löding, P. Madhusudan, and O. Serre. Visibly pushdown games. In *FSTTCS*, LNCS 3328, pages 408–420, 2004.
16. R. McNaughton. Parenthesis grammars. *Journal of the ACM*, 14(3):490–500, 1967.
17. F. Neven. Automata, logic, and XML. In *CSL*, pages 2–26, 2002.
18. T. Reps, S. Horwitz, and S. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *ACM POPL*, pages 49–61, 1995.
19. L. Segoufin and V. Vianu. Validating streaming XML documents. In *ACM PODS*, pages 53–64, 2002.
20. W. Thomas. On logics, tilings, and automata. In *ICALP*, LNCS 510, pages 441–454, 1991.
21. M.Y. Vardi and P. Wolper. Reasoning about infinite computations. *Information and Computation*, 115(1):1–37, 1994.
22. B. von Braunmühl and R. Verbeek. Input-driven languages are recognized in *log n* space. In *FCT*, LNCS 158, pages 40–51, 1983.