Model Checking: From Tools to Theory⋆

Rajeev Alur

University of Pennsylvania

Abstract. Model checking is often cited as a success story for transitioning and engineering ideas rooted in logics and automata to practice. In this paper, we discuss how the efforts aimed at improving the scope and effectiveness of model checking tools have revived the study of logics and automata leading to unexpected theoretical advances whose impact is not limited to model checking. In particular, we describe how our efforts to add context-free specifications to software model checking led us to the model of *nested words* as a representation of data with both a linear ordering and a hierarchically nested matching of items. Such dual structure occurs in diverse corners of computer science ranging from executions of structured programs where there is a well-nested matching of entries to and exits from functions and procedures, to XML documents with the hierarchical structure specified by start-tags matched with end-tags. Finite-state acceptors of nested words define the class of regular languages of nested words that has all the appealing theoretical properties that the class of regular word languages enjoys. We review the emerging theory of nested words, its extension to nested trees, and its potential applications.

1   Introduction

The abstract for the talk titled "The Birth of Model Checking" by Ed Clarke at the *25 Years of Model Checking* symposium begins as follows

> The most important problem in model checking is the *State Explosion Problem*. In particular, it is far more important than the logic or specification formalism that is used – CTL, LTL, CTL*, Büchi automata, or the μ-calculus.

Indeed, without the spectacular progress on combating the state explosion problem, it is not clear if model checking would have had any impact on industrial practice at all. However, we would like to argue that theory, in particular, specification languages based on temporal logics, automata, and fixpoint logics, have contributed significantly to the success of model checking. First, theory of regular languages of finite and infinite words and trees, gives a clear understanding of which properties are algorithmically checkable. Second, modern

industrial-strength specification languages such as PSL are rooted in the theory of temporal logics [PSL05]. Such standardized specification languages have an important role beyond model checking, namely, in testing as well as simulation. Third, since fixpoint logic has a strong computational flavor, logics have suggested ways of implementing symbolic model checkers. Finally, the vigorous debate on relative merits of different specification languages has contributed to the intellectual health and growth of the field. It is also worth noting that one key manner in which model checking differs from program analysis is the use of specification languages: model checking typically has focussed on efficiently checking generic classes of properties such as safety and liveness, while program analysis has emphasized specific analysis questions such as pointer analysis and buffer overflows.

The foundational work on monadic second order logics and ω-automata over words and trees dates back to research in 1960s. Particularly noteworthy results include

1. Büchi's Theorem: A language of infinite words is definable using monadic second order logic of linear order (S1S) iff it is accepted by a (finite) Büchi automaton [Büc62].
2. Kamp's Theorem: A property of infinite words is expressible in first-order theory of linear order iff it is expressible in linear temporal logic LTL [Kam68].
3. Rabin's Theorem: The monadic second order theory of binary trees (S2S) is decidable [Rab69].

The automata-theoretic approach to verification, advocated by Vardi and others, connects model checking tools to the above results and their subsequent refinements, and has been celebrated with numerous awards including the 2006 ACM Kannellakis Theory in Practice Award [WVS83, VW94, KVW00, Tho90, Hol97, Kur94]. We wish to argue that, as the success of model checking tools brought intense focus on expressiveness and decidability boundary, and this led to fundamental advances in theory. Since automata and logics have applications to other areas of computing, such as databases, document processing, and planning, model checking continues to contribute to these areas. We list two such developments for illustrative purposes.

Tree automata, μ-calculus, and parity games: The use of branching-time logics such as CTL [CE81] and μ-calculus [Koz83] in symbolic model checking tools such as SMV [McM93] led researchers revisit the theory of infinite trees. While classical theory of trees considers binary trees and their regular properties, programs are best modeled by trees that are unordered and unranked, and we want to focus on properties that do not distinguish among bisimilar systems (the notion of bisimilarity was introduced in theory of concurrency [Mil89]). The resulting body of research led to new notions of automata such as *alternating tree automata* [EJ91, JW96, MS85, CDG$^+$02]. We now know that, for a set L of infinite, unordered, unranked trees, the following are equivalent: (1) L is bisimulation-closed and definable using monadic second order logic, (2) L is definable in μ-calculus, and (3) L is accepted by an alternating parity

tree automaton. This work also connects to deciding two-player games with parity winning condition, and provides the basis for *synthesis* of correct controllers with respect to LTL specifications and modular verification of open systems [Tho02, KVW01, AHK02].

Timed automata: Traditional automata do not admit an explicit modeling of time, and consequently, in order to extend model checking techniques to timed circuits, *timed automata* [AD94] were introduced as a formal notation to model the behavior of real-time systems. Timed automata accept *timed languages* consisting of sequences of events tagged with their occurrence times. Many analysis problems for timed automata are solvable, and this has led to tools such as Uppaal for verifying finite-state real-time systems [LPY97, DOTY96]. Theory of regular timed languages has also been developed with an accompanying study of real-time temporal logics [ACD93, AH94, AH93, HRS98]. Timed automata are now used as a formal model of real-time computation in contexts beyond model checking (see, for instance, textbooks on Signals and systems [LV02] and control theory [CL99]). The main technique for analysis of timed automata relies on constructing a finite quotient of the infinite space of real-valued state vectors [AD94], and this has led to many abstraction techniques for dynamical and hybrid systems [AHLP00, PS02].

In the rest of this paper, we focus in detail on our current line of research. We describe how our efforts to understand limits of algorithmically checkable properties of pushdown models led us to the model of nested words as a representation of data with both a linear ordering and a hierarchically nested matching of items. Such dual structure occurs in diverse corners of computer science ranging from executions of structured programs where there is a well-nested matching of entries to and exits from functions and procedures, to XML documents with the well-nested structure given by start-tags matched with end-tags. We review the emerging theory of nested words and its potential applications [AM04, AEM04, AKMV05, ACM06a, ACM06b, AM06, KMV06a, Alu07, AAB⁺07].

2   History of Verification of Pushdown Systems

Pushdown automata naturally model the control flow of sequential computation in typical programming languages with nested, and potentially recursive, invocations of program modules such as procedures and method calls. Consequently, a variety of program analysis, compiler optimization, and model checking questions can be formulated as decision problems for pushdown automata.

When viewed as a generator of words, a pushdown model specifies a context-free language of words. Decidability of regular requirements of pushdown models, then, follows from classical results on pushdown automata: the product of a pushdown automaton and a finite-state automaton gives a pushdown automaton, and the emptiness of the language of a pushdown automaton can be checked in polynomial-time (see any standard textbook on automata theory, such as, [HU79]). The decision procedure for emptiness of pushdown automata, in fact,

forms the basis for many inter-procedural dataflow analysis problems [SP81, RHS95] (see [Rep98] for a survey).

In the context of model checking, a pushdown automaton can be interpreted as a generator of a context-free language of infinite words. Model checking of LTL requirements against pushdown models is known to be decidable [BS92, BEM97] (see, also, [ABE+05] for refined complexity bounds). Checking μ-calculus requirements of pushdown models, and similarly, solving games over pushdown graphs with winning condition specified in LTL, are also known to be decidable [Wal01]. The emergence of software model checking, as implemented in tools such as SLAM and BLAST, brought pushdown verification to forefront [BR01, HJM+02]. In these tools, a C program is mapped to a pushdown model (more specifically, to Boolean programs that allow stack-based control flow, but with only Boolean data variables) using predicate abstraction, and then symbolic model checking is used to analyze the resulting model.

The typical program analysis tools over control-flow graphs and BDD-based model checking tools such as Bebop, are based on the so-called summary computation for pushdown models [BR00, Rep98]. Intuitively, the analysis algorithm computes, for each procedure or a component, summaries of the form (x, y), meaning that if the component is invoked with input x, it may return with output y. The number of such summaries is finite, and can be computed by an inductive fixpoint computation. An alternative view is based on the so-called *regular model checking* [BEM97]. In a pushdown model, the state is completely described by the control state and a finite word over the alphabet of stack symbols describing the contents of the stack. It turns out that the set of reachable states of a model is regular and can be represented by a finite-state automaton. Model checking can be viewed as computation of the edges of this automaton, and the model checker Moped is based on this approach [EHRS00]. Finally, there exist interesting decidability results for logics interpreted over pushdown graphs, typically using interpretation over trees [Cau03, KPV02].

While many analysis problems can be captured as regular requirements, and hence, specifiable in LTL or μ-calculus, many others require inspection of the stack or matching of calls and returns, and are context-free. Even though the general problem of checking context-free properties of pushdown automata is undecidable, algorithmic solutions have been proposed for checking many different kinds of non-regular properties. For example, access control requirements such as "a module A should be invoked only if the module B belongs to the call-stack," and bounds on stack size such as "the number of interrupt-handlers in the call-stack should never exceed 5," require inspection of the stack, and decision procedures for certain classes of stack properties already exist [JMT99, CW02, EKS03, CMM+04]. Our own efforts to add expressiveness to LTL, while maintaining decidability of model checking with respect to pushdown models, led to the definition of temporal logic CARET that allows matching of calls and returns. CARET can express the classical correctness requirements of program modules with pre and post conditions, such as "if p holds when a module is invoked, the module must return, and q holds upon return" [AEM04].

This suggests that the answer to the question "which class of properties are algorithmically checkable against pushdown models?" should be more general than "regular." The key feature of checkable requirements, such as stack inspection and matching calls and returns, is that the stacks in the model and the property are correlated: while the stacks are not identical, the two synchronize on when to push and when to pop, and are always of the same depth. We first formalized this intuition by defining *visibly pushdown automata* (VPA). Such an automaton operates over words over an alphabet that is partitioned into three disjoint sets of calls, returns, and internal symbols. While reading a *call* symbol, the automaton must push, while reading a *return* symbol, it must pop (if the stack is non-empty), and while reading an *internal* symbol, it can only update its control state. A language over a partitioned alphabet is a *visibly pushdown language* if there is such an automaton that accepts it. This class has desirable closure properties, tractable decision problems, multiple equivalent characterizations, and adequate for formulating program analysis questions.

We now believe that a better way of exposing the matching call-return structure of the input word is by explicitly adding nesting edges [AM06]. Nested words integrate trees and words as the underlying signature has both a linear order and a hierarchical nesting relation. Finite-state acceptors of nested words define the class of regular languages of nested words that has all the appealing theoretical properties that the class of classical regular word languages enjoys. As we will describe, this allows us to view programs as finite-state generators of regular languages of nested words, as opposed to (infinite-state) pushdown generators of (restricted classes of) context-free languages of words, thereby allowing model checking of stronger requirements.

3   Nested Words

A nested word consists of a sequence of linearly ordered positions, augmented with nesting edges connecting calls to returns (or open-tags to close-tags). The edges create a properly nested hierarchical structure, while allowing some of the edges to be pending. We will present definitions for finite nested words, but the theory extends to infinite words.

We use edges starting at $-\infty$ and edges ending at $+\infty$ to model "pending" edges. A *nesting relation* $\rightsquigarrow$ of length ℓ is a subset of $\{-\infty, 1, 2, \ldots \ell\} \times \{1, 2, \ldots \ell, +\infty\}$ such that if $i \rightsquigarrow j$ then $i < j$; if $i \rightsquigarrow j$ and $i \rightsquigarrow j'$ and $i \neq -\infty$ then $j = j'$, if $i \rightsquigarrow j$ and $i' \rightsquigarrow j$ and $j \neq +\infty$ then $i = i'$, and if $i \rightsquigarrow j$ and $i' \rightsquigarrow j'$ then it is not the case that $i < i' \leq j < j'$. The definition ensures that nesting edges go only forward, do not cross, and every position is involved in at most one nesting edge. Source positions for nesting edges are *call* positions, target positions for nesting edges are *return* positions, and a position that is neither a call or a return is called *internal*. A *nested word* n over an alphabet Σ is a pair $(a_1 \ldots a_\ell, \rightsquigarrow)$, such that a_i, for each $1 \leq i \leq \ell$, is a symbol in Σ, and $\rightsquigarrow$ is a nesting relation of length ℓ.
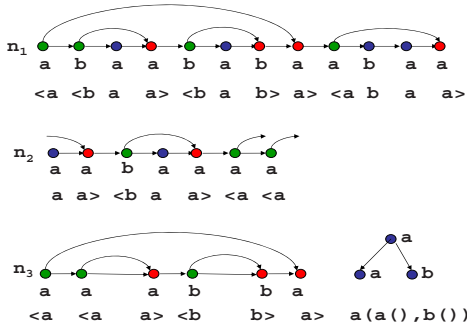
Fig. 1. Sample nested words

This nesting structure can be uniquely represented by a sequence specifying the types of positions (calls, returns, and internals). In particular, we assume that $\langle$ and $\rangle$ are special symbols that do not appear in the alphabet Σ. Then, define the *tagged alphabet* $\hat{\Sigma}$ to be the set that contains the symbols $\langle a$, a, and $a\rangle$ for each $a \in \Sigma$. Given a nested word over Σ, we can map it to a word over $\hat{\Sigma}$: at every call position labeled a, output $\langle a$; at every return position labeled a, output $a\rangle$; and at every internal position labeled a, output a. This correspondence between nested words and words over tagged symbols is a bijection. Figure 1 shows some nested words over the alphabet $\{a, b\}$ along with their linear encodings.

Finite-state acceptors over nested words can process both linear and hierarchical structure. A *nested word automaton* (NWA) A over an alphabet Σ consists of a finite set Q of states, an initial state $q_0 \in Q$, a set of final states $F \subseteq Q$, a call-transition function $\delta_c : Q \times \Sigma \mapsto Q \times Q$, an internal-transition function $\delta_i : Q \times \Sigma \mapsto Q$, and a return-transition function $\delta_r : Q \times Q \times \Sigma \mapsto Q$. The automaton A starts in the initial state, and reads the nested word from left to right. The state is propagated along the linear edges as in case of a standard word automaton. However, at a call, the nested word automaton can propagate a state along the outgoing nesting edge also. At a return, the new state is determined based on the states propagated along the linear as well as the nesting incoming edges. Formally, a *run* r of the automaton A over a nested word $n = (a_1 \ldots a_\ell, \leadsto)$ is a linear sequence $q_0, \ldots, q_\ell$ of states and a nesting sequence q_{ij}, for $i \leadsto j$, of states such that for each position $1 \leq i \leq \ell$, if i is a call with $i \leadsto j$, then $\delta_c(q_{i-1}, a_i) = (q_i, q_{ij})$; if i is an internal, then $\delta_i(q_{i-1}, a_i) = q_i$; and if i is a return such that $j \leadsto i$, then $\delta_r(q_{i-1}, q_{ji}, a_i) = q_i$, where if $j = -\infty$ then $q_{ji} = q_0$. For a given nested word n, the automaton has precisely one run over n. The automaton A accepts the nested word n if in this run, $q_\ell \in F$. The language $L(A)$ of a nested-word automaton A is the set of nested words it accepts. The resulting class of *regular* languages of nested words seems to have all the appealing theoretical properties that the classes of classical regular word and tree languages enjoy.

It is easy to see that if L is a regular language of nested words, then the corresponding language of words over tagged symbols is a context-free language.

This is because a nested word automaton can be interpreted as a pushdown automaton over words: call transitions can be simulated by pushing the state along nesting edge, and return transitions can access this state by popping the stack. Languages of words with well-bracketed structure have been studied as Dyck languages and *parenthesis* languages, and shown to have some special properties compared to context-free languages (for example, decidable equivalence problem) [McN67, Knu67]. The new insight is that the matching among left and right parenthesis can be considered to be an explicit component of the input structure, and this leads to a robust notion of regular languages using finite-state acceptors.

There is an emerging and growing body of literature studying nested word automata, and we review some of the results below.

Closure: The class of regular languages of nested words is (effectively) closed under union, intersection, complementation, concatenation, and Kleene-$*$. If L is a regular language of nested words then all the following languages are regular: the set of all prefixes of all the words in L; the set of all suffixes of all the words in L; the set of reversals of all the words in L. Regular languages are closed under tree-like operations that use hierarchical structure.

Determinization: A *nondeterministic* NWA A has finite set Q of states, a set of initial states $Q_0 \subseteq Q$, a set $F \subseteq Q$ of final states, a call-transition relation $\delta_c \subseteq Q \times \Sigma \times Q \times Q$, an internal-transition relation $\delta_i \subseteq Q \times \Sigma \times Q$, and a return-transition relation $\delta_r \subseteq Q \times Q \times \Sigma \times Q$. The automaton now has a choice at every step, and accepts a word if one of the possible runs accepts. Nondeterministic nested word automata are no more expressive than the deterministic ones: given a nondeterministic automaton A with s states, one can effectively construct a deterministic NWA B with 2^{s^2} states such that $L(B) = L(A)$. The construction is a generalization of the classical subset construction for determinizing word automata, and a state of B is set of pairs of states of A.

Logic based characterization: The classical correspondence between monadic second order logic and finite recognizability for words and trees continues to hold for nested words. The *monadic second-order logic of nested words* (MSO) is given by the syntax:

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

where $a \in \Sigma$, x, y are first-order variables, and X is a second order variable. The semantics is defined over nested words in a natural way. A language L of nested words over Σ is regular iff there is an MSO sentence ϕ over Σ such that L is the set of all nested words that satisfy ϕ.

The correspondence between linear temporal logic and first-order logic continues to hold too. The logic *Nested Word Temporal Logic* (NWTL) has atomic propositions, logical connectives, the linear *next* and *previous* operators, the hierarchical next and previous operators (e.g., "hierarchical-next φ" holds at a call position iff φ holds at the matching return), and *until* and *since* operators that are interpreted over the "summary" paths. The summary path between two

positions i and j is the shortest path in the graph of the nested word: if the summary path from i to j reaches a call position k such that $i \leq k \rightsquigarrow k' \leq j$, then it will follow the nesting edge from k to k'. A language L of nested words is definable in first-order logic of nested words (that is, the logic above without the second-order variables X) iff it is expressible in the temporal logic NWTL [AAB+07].

Decision problems: Given a nested word automaton A and a nested word n, the membership problem (is n in $L(A)$?) can be solved in linear time. The space required is proportional to the depth of n since one needs to remember the labeling of pending nesting edges at every position. If A is nondeterministic, membership problem can be solved in time $O(|A|^3 \ell)$ using dynamic programming similar to the one used for membership for pushdown word automata.

The emptiness problem for nested word automata(is $L(A)$ empty?) can be solved in cubic time using techniques similar to the ones used for pushdown word automata or tree automata.

Problems such as language inclusion and language equivalence are decidable. These problems can be solved using constructions for complementation and language intersection, and emptiness test. If one of the automata is nondeterministic, then this would require determinization, and both language inclusion and equivalence are EXPTIME-complete for nondeterministic NWAs.

4   Revised Formulation of Software Model Checking

Traditionally, execution of a program is modeled as a word over an alphabet Σ, where 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. 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), and ex to denote the ending of the current scope, and sk to denote all other actions of the program. A program P generates, then, a set $L(P)$ of words over this alphabet. The specification S is given as a set of "desirable" words, and verification corresponds to checking whether the inclusion $L(P) \subseteq S$ holds. Since typical programming languages are Turing complete, the verification problem is undecidable. The first step in algorithmic program verification is to approximate a program using *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 [SH97,BMMR01,HJM+02]. The resulting model P' hence has finite data and stack-based control flow (see Boolean programs [BR00] and recursive state machines [ABE+05] as concrete instances of pushdown models of programs). The language $L(P')$ is a context-free language of words. If the specification S is a regular language, then the verification question $L(P') \subseteq S$ can be solved. Consider the requirement that every write access is followed by a read access. This can be expressed by the LTL formula $\square(wr \rightarrow \lozenge rd)$, and is indeed a regular property. However, if we want to express the requirement that "if a procedure writes to x, it must read x," we must capture the scope of each procedure by matching of en and ex symbols, and the requirement is not a regular

language, and thus, not expressible in the specification languages supported by existing software model checkers such as SLAM [BR00] and BLAST [HJM$^+$02]. The specification is a context-free language, but this is not useful for algorithmic verification since context-free languages are not closed under intersection, and decision problems such as language inclusion and emptiness of intersection of two languages are undecidable for context-free languages.

In the revised formulation, an execution is modeled as a nested word. In addition to the linear sequence of symbols given by the program execution, from each entry symbol *en*, there is a nesting edge to the matching exit symbol *ex*. Following the nesting edge corresponds to skipping the called procedure, and a path that uses only nesting and internal edges gives the part of the execution that is local to a procedure. We can interpret the abstracted program P' as a nested word automaton, and associate with it a regular language $L'(P')$ of nested words. It is worth noting that, in general, pushdown models can be interpreted as nested word automata as syntactically the two definitions are same (in NWAs, stack alphabet coincides with the set of states, acceptance is by final state, call transitions are same as push transitions, and return transitions are same as pop transitions). The difference is only in the semantics: pushdown automata define word languages while NWAs define nested word languages.

The specification, now, is given as a language S' over nested words, and verification reduces to the language-inclusion problem for nested words: $L'(P') \subseteq S'$. The question is solvable as long as S' is a regular language of nested words. Clearly, every regular language of words is also a regular language of nested words. The requirement that "if a procedure writes to x, it must read x" also becomes regular now, and there is a natural two-state deterministic nested-word automaton that specifies it. 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 $j = 0, 1$, $\delta_i(q_j, rd) = q_0$; $\delta_i(q_j, wr) = q_1$; $\delta_i(q_j, sk) = q_j$; $\delta_c(q_j, en) = (q_0, q_j)$; 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.

Further, we can design temporal logics for programs that exploit the nested structure. An example of such a temporal logic is CARET [AEM04], which extends linear temporal logic by *local* modalities such as $\bigcirc^a \phi$, which holds at a call if the return-successor of the call satisfies ϕ. The formula $\square(wr \rightarrow \lozenge^a rd)$ captures the specification "if a procedure writes to x, it must read x." 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 [VW94], 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 [AM04, AEM04].

Software model checking tools such as SLAM and BLAST support an assertion language for writing monitors checking for violations of safety properties. The monitor M is observing the executions of P, and reaches an error state if an undesirable execution is detected. The verification question is to check if the monitor can reach an error state. Given a C program P and a monitor M written in the query language, the model checker first constructs an annotated C program P' such that the verification problem reduces to analysis of P'. While current assertion languages for monitors support automata over words, now we can strengthen them to allow automata over nested words. The transformation of P to the annotated program P', to account for M, can be done with equal ease even for this more expressive language. The resulting program P' can be subjected to different analysis techniques such as testing, runtime monitoring, static analysis, and model checking. Thus, the nested-word formulation can be useful for any analysis technique. Even though we have emphasized pushdown models in the theory of nested words, the proposed reformulation is useful even if programs are not recursive as long as they are structured with stack-based control flow.

5   Fixpoints for Local and Global Program Flows

In the branching-time approach to program verification, a program P is modeled by an unranked unordered infinite tree T_P such that nodes in T_P are labeled with program states, and paths in T_P correspond to executions of P. The branching-time specification specifies the set S of desirable trees, and model checking corresponds to the membership test $T_P \in S$. The μ-calculus [Koz83] is a modal logic with fixpoints, and is an extensively studied branching-time specification formalism with applications to program analysis, computer-aided verification, and database query languages [Eme90, Sti91]. From a theoretical perspective, its status as the *canonical* temporal logic for regular requirements is due to the fact that its expressiveness exceeds all commonly used temporal logics such as LTL, CTL, and CTL*, and equals *alternating parity tree automata* or the bisimulation-closed fragment of monadic second-order theory over trees [EJ91, JW96]. From a practical standpoint, iterative computation of fixpoints naturally suggests symbolic evaluation, and symbolic model checkers such as SMV check CTL properties of finite-state models by compiling them into μ-calculus formulas [BCD+92, McM93].

There are at least three reasons that motivated us to extend the theory of nested words to the branching-time case. First, while algorithmic verification of μ-calculus properties of pushdown models is possible [Wal01, BS99], classical μ-calculus cannot express pushdown specifications that require inspection of the stack or matching of calls and returns. This raises the question about the right theoretical extension of μ-calculus that can capture CaRet and nested word automata. Second, in the program analysis literature, it has been argued that

data flow analysis, such as the computation of live variables and very busy expressions, can be viewed as evaluating μ-calculus formulas over abstractions of programs [Ste91, Sch98]. This correspondence does not hold when we need to account for *local* data flow paths. For instance, for an expression e that involves a variable local to a procedure P, the set of control points within P at which e is very busy (that is, e is guaranteed to be used before any of its variables get modified), cannot be specified using a μ-calculus formula even though interprocedural dataflow analysis can compute this information. Can we extend μ-calculus so that it can capture interprocedural dataflow analysis? Finally, the standard reachability property "some p-state is reachable" is expressed by the μ-calculus formula $\varphi = \mu X.(p \vee \bigcirc X)$. The meaning of φ is the smallest set X such that if a state satisfies p or has a successor in X then it is in X. While this formula captures reachability over all models, over finite-state models, the specification also encodes the symbolic algorithm for computing the set of states satisfying φ by successive approximations of the fixpoint: let X_0 to be the set of states satisfying p, and at each step i, compute X_{i+1} from X_i by adding states that can reach X_i in one step (termination is obtained when $X_i = X_{i+1}$). Over pushdown models, such a computation may not terminate. The correct way to compute reachability, as implemented in dataflow analysis or tools such as SLAM, is based on "summarization" of paths. The summarization algorithm can be viewed as a fixpoint computation over pairs of states of the form (x, y) meaning that state y is reachable if the current procedure is called with input state x. This raises the question if there is a different way of expressing reachability over pushdown models.

A *nested tree* is a labeled tree T augmented with a nesting relation $\rightsquigarrow$ over the vertices of T such that every path through the tree is a nested word (see [ACM06a] for precise definition). In context of program verification, the tree T_P corresponding to a program P, will be unranked, unordered, and infinite, and the nesting relation is obtained by adding edges from call nodes to matching returns. Note that a call node can have multiple matching returns (and, no matching returns along some paths corresponding to executions in which the called procedure does not return). It turns out there is an appealing fixpoint calculus NT_μ over nested trees that has the following properties:

1. The model-checking problem for NT_μ is effectively solvable against pushdown models with no more effort than that required for weaker logics such as CTL (EXPTIME-complete).

2. Evaluating NT_μ formulas over pushdown models captures the standard summary-based analysis algorithms, and thus, expressing a property in NT_μ amounts to describing symbolic computation for evaluation.

3. The logic NT_μ encompasses all properties expressed by nested word automata as well as by the classical μ-calculus. This makes NT_μ the most expressive known program logic for which algorithmic software model checking is feasible. In fact, the decidability of most known program logics (μ-calculus, temporal logics LTL and CTL, CARET, etc.) can be understood by their

interpretation in the monadic second-order logic over trees. This is not true for the logic NT_μ, making it a new powerful tractable program logic.

4. The logic NT_μ can capture local as well as global program flows, and thus, interprocedural dataflow analysis problems can be stated in NT_μ.

5. Expressiveness of NT_μ coincides with *alternating parity automata* over nested trees (APNTA). An APNTA is a finite-state tree automaton such that (a) its transition relation is alternating, so along an edge it can send multiple copies, (b) its acceptance condition is defined using parity condition over the infinite run, and (c) like a nested word automaton, at a call node, the automaton sends states to the immediate tree successor as well as to the return successors along nesting edges, and at a return node, the state can depend on the state at the immediate tree parent as well as the state along the nesting edge from the matching call parent.

6. While the correspondence between alternating tree automata and fixpoint calculus holds as in the classical tree case, the correspondence between monadic second order logic and fixpoint calculus fails: the monadic second order logic over nested trees and NT_μ seem to have incomparable expressiveness (though this is not proved formally yet). Both logics have undecidable satisfiability problem [ACM06b].

We intuitively describe the logic NT_μ below. The variables of the calculus evaluate not over sets of vertices, but rather over sets of subtrees that capture *summaries* of computations in the "current" program block. The fixpoint operators in the logic then compute fixpoints of summaries. For a given vertex s of a nested tree, consider the subtree rooted at s such that the leaves correspond to the matching returns as specified by the nesting relation (while modeling program, such a subtree captures all the computations till the procedure that s belongs to returns). 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 vertex s has a call-edge to a vertex t, and there exists a summary $\langle t, V \rangle$ satisfying φ and for each leaf v that belongs to V, the subtree $\langle v, U \rangle$ satisfies ψ.

This logic is best explained using the specification of local reachability: let us identify the set of all summaries $\langle s, U \rangle$ such that there is a *local* path from s to some node in U (i.e. all calls from the initial procedure must have returned before reaching U). In our logic, this is written as the formula $\varphi = \mu X. \langle ret \rangle R_1 \vee \langle loc \rangle X \vee \langle call \rangle X\{X\}$. The above means that X is the smallest set of summaries of the form $\langle s, U \rangle$ such that (1) there is a return-edge from s to some node in U, (2) there is an internal edge from s to t and there is a summary $\langle t, U \rangle$ in X, or (3) there is a call-edge from s to t and a summary $\langle t, V \rangle$ in X such that from each $v \in V$, $\langle v, U \rangle$ is a summary in X. Notice that the above formula identifies the summaries in the natural way it will be *computed* on a pushdown system:

compute the local summaries of each procedure, and update the reachability relation using the call-to-return summaries found in the procedures called.

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

6   Modeling and Processing Linear-Hierarchical Data

While nested words were motivated by program verification, they can potentially be used to model data with the dual–linear and hierarchical, structure. Such dual structure exists naturally in many contexts including XML documents, annotated linguistic data, and primary/secondary bonds in genomic sequences. Also, in some applications, even though the only logical structure on data is hierarchical, linear sequencing is added either for storage or for stream processing. Data with linear-hierarchical structure is traditionally modeled using binary (or more generally, ordered) trees and queried using tree automata (see [Nev02, Lib05, Sch04] for recent surveys on applications of tree automata and tree logics to document processing).

Even though tree models and tree automata are extensively studied with a well-developed theory with appealing properties (see [CDG$^+$02]), they seem ill suited to capture and query the linear structure. First, tree-based approach implicitly assumes that the input linear document can be parsed into a tree, and thus, one cannot represent and process data that may not parse correctly. Word operations such as prefixes, suffixes, and concatenation, while natural for document processing, do not have analogous tree operations. Second, tree automata do not generalize word automata. Finite-state word automata can be exponentially more succinct than tree automata. For example, the query that patterns $p_1, \ldots p_n$ appear in the document in that order (that is, the regular expression $\Sigma^* p_1 \Sigma^* \ldots p_n \Sigma^*$) compiles into a deterministic word automaton with $n + 1$ states, but standard deterministic bottom-up tree automaton for this query must be of size exponential in n. This deficiency shows up more dramatically if we consider pushdown acceptors: a query such as "the document contains an equal number of occurrences of patterns p and q" is a context-free word language but is not a context-free tree language.

In a nutshell, binary/ordered trees encode both linear and hierarchical structure, but not on an equal footing. Recently we have argued that the model of nested words is a better integration of the two orderings, and can either simplify or improve existing ways of document processing [KMV06b, Alu07]. We have already seen that words are nested words where all positions are internals. Ordered trees can be interpreted as nested words using the following traversal:

to process an *a*-labeled node, first print an *a*-labeled call, process all children in order, and print an *a*-labeled return. Binary trees, ranked trees, unranked trees, forests, and documents that do not parse correctly, all can be represented with equal ease. Figure 1 shows the ordered tree corresponding to the third nested word, the first two do not correspond to trees.

Since XML documents already contain tags that specify the position type, they can be interpreted as tagged encoding of nested words without any pre-processing. As we have seen already, the class of regular languages of nested words seems to have all the appealing theoretical properties that the classes of classical regular word and tree languages enjoy, and decision problems such as membership, emptiness, language inclusion, and language equivalence are all decidable, typically with the same complexity as the corresponding problem for tree automata.

In order to study the relationship of nested word automata to various kinds of word and tree automata, let us consider restricted classes of nested word automata and the impact of these restrictions on expressiveness and succinct-ness [Alu07]. *Flat* automata do not propagate information along the nesting edges at calls, and correspond exactly to classical word automata accepting the weaker class of regular word languages. *Bottom-up* automata, on the other hand, do not propagate information along the linear edges at calls. Over the sub-class of nested words corresponding to ordered trees, these automata correspond exactly to bottom-up tree automata for binary trees and stepwise bottom-up tree automata [BKMW01] for unranked trees. However, there is an exponential price in terms of succinctness due to this restriction. The class of *joinless* automata avoids a nontrivial join of information along the linear and nesting edges at returns, and this concept is a generalization of the classical top-down tree automata. While deterministic joinless automata are strictly less expressive, nondeterministic ones can accept all regular languages of nested words. The suc-cinctness gap between nested word automata and traditional tree automata holds even if we restrict attention to paths (that is, unary trees): nested word automata are exponentially more succinct than both bottom-up and top-down automata. We have also studied *pushdown nested word automata* by adding a stack to the finite-state control of nondeterministic joinless automata. Both pushdown word automata and pushdown tree automata are special cases, but pushdown nested word automata are strictly more expressive than both. In terms of complexity of analysis problems, they are similar to pushdown tree automata: membership is Np-complete and emptiness is Exptime-complete.

These results suggest that nested words and nested word automata may be a more suitable way to model and process linear-hierarchical data. We need to explore if compiling existing XML query languages into nested word automata reduces query processing time in practice.

References

[AAB+07]   Alur, R., Arenas, M., Barcelo, P., Etessami, K., Immerman, N., Libkin, L.:
           First-order and temporal logics for nested words (unpublished manuscript,
           2007)

[ABE+05]   Alur, R., Benedikt, M., Etessami, K., Godefroid, P., Reps, T., Yannakakis,
           M.: Analysis of recursive state machines. ACM Transactions on Program-
           ming Languages and Systems 27(4), 786–818 (2005)

[ACD93]   Alur, R., Courcoubetis, C., Dill, D.L.: Model-checking in dense real-time.
           Information and Computation 104(1), 2–34 (1993)

[ACM06a]   Alur, R., Chaudhuri, S., Madhusudan, P.: A fixpoint calculus for local and
           global program flows. In: Proceedings of the 33rd Annual ACM Symposium
           on Principles of Programming Languages, pp. 153–165 (2006)

[ACM06b]   Alur, R., Madhusudan, P., Chaudhuri, S.: Languages of Nested Trees. In:
           Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 329–342.
           Springer, Heidelberg (2006)

[AD94]   Alur, R., Dill, D.L.: A theory of timed automata. Theoretical Computer
           Science 126, 183–235 (1994)

[AEM04]   Alur, R., Etessami, K., Madhusudan, P.: A Temporal Logic of Nested
           Calls and Returns. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS,
           vol. 2988, pp. 467–481. Springer, Heidelberg (2004)

[AH93]   Alur, R., Henzinger, T.A.: Real-time logics: complexity and expressiveness.
           Information and Computation 104(1), 35–77 (1993)

[AH94]   Alur, R., Henzinger, T.A.: A really temporal logic. Journal of the
           ACM 41(1), 181–204 (1994)

[AHK02]   Alur, R., Henzinger, T.A., Kupferman, O.: Alternating-time temporal
           logic. Journal of the ACM 49(5), 1–42 (2002)

[AHLP00]   Alur, R., Henzinger, T.A., Lafferriere, G., Pappas, G.: Discrete abstrac-
           tions of hybrid systems. Proceedings of the IEEE 88(7), 971–984 (2000)

[AKMV05]   Alur, R., Madhusudan, P., Viswanathan, M., Kumar, V.: Congruences for
           Visibly Pushdown Languages. In: Caires, L., Italiano, G.F., Monteiro, L.,
           Palamidessi, C., Yung, M. (eds.) ICALP 2005. LNCS, vol. 3580, pp. 1102–
           1114. Springer, Heidelberg (2005)

[Alu07]   Alur, R.: Marrying words and trees (unpublished manuscript, 2007)

[AM04]   Alur, R., Madhusudan, P.: Visibly pushdown languages. In: Proceedings of
           the 36th ACM Symposium on Theory of Computing, pp. 202–211 (2004)

[AM06]   Alur, R., Madhusudan, P.: Adding Nesting Structure to Words. In: H.
           Ibarra, O., Dang, Z. (eds.) DLT 2006. LNCS, vol. 4036, pp. 1–13. Springer,
           Heidelberg (2006)

[BCD+92]   Burch, J.R., Clarke, E.M., Dill, D.L., Hwang, L.J., McMillan, K.L.: Sym-
           bolic model checking: 10^{20} states and beyond. Information and Computa-
           tion 98(2), 142–170 (1992)

[BEM97]   Boujjani, A., Esparza, J., Maler, O.: Reachability analysis of pushdown au-
           tomata: Applications to model checking. In: Mazurkiewicz, A., Winkowski,
           J. (eds.) CONCUR 1997. LNCS, vol. 1243, pp. 135–150. Springer, Heidel-
           berg (1997)

[BKMW01]   Brüggemann-Klein, A., Murata, M., Wood, D.: Regular tree and regu-
           lar hedge languages over unranked alphabets: Version 1. Technical Report
           HKUST-TCSC-2001-0, The Hongkong University of Science and Technol-
           ogy (2001)

[BMMR01] Ball, T., Majumdar, R., Millstein, T.D., Rajamani, S.K.: Automatic predicate abstraction of C programs. In: SIGPLAN Conference on Programming Language Design and Implementation, pp. 203–213 (2001)

[BR00] Ball, T., Rajamani, S.: Bebop: A symbolic model checker for boolean programs. In: Havelund, K., Penix, J., Visser, W. (eds.) SPIN 2000. LNCS, vol. 1885, pp. 113–130. Springer, Heidelberg (2000)

[BR01] Ball, T., Rajamani, S.: The SLAM toolkit. In: Computer Aided Verification, 13th International Conference (2001)

[BS92] Burkart, O., Steffen, B.: Model checking for context-free processes. In: Cleaveland, W.R. (ed.) CONCUR 1992. LNCS, vol. 630, pp. 123–137. Springer, Heidelberg (1992)

[BS99] Burkart, O., Steffen, B.: Model checking the full modal mu-calculus for infinite sequential processes. Theoretical Computer Science 221, 251–270 (1999)

[Büc62] Büchi, J.R.: On a decision method in restricted second-order arithmetic. In: Proceedings of the International Congress on Logic, Methodology, and Philosophy of Science 1960, pp. 1–12. Stanford University Press (1962)

[Cau03] Caucal, D.: On infinite transition graphs having a decidable monadic theory. Theoretical Computer Science 290(1), 79–115 (2003)

[CDG+02] Comon, H., Dauchet, M., Gilleron, R., Lugiez, D., Tison, S., Tommasi, M.: Tree automata techniques and applications. Draft (2002), http://www.grappa.univ-lille3.fr/tata/

[CE81] Clarke, E.M., Emerson, E.A.: Design and synthesis of synchronization skeletons using branching time temporal logic. In: Kozen, D. (ed.) Logic of Programs 1981. LNCS, vol. 131, pp. 52–71. Springer, Heidelberg (1982)

[CL99] Cassandras, C.G., Lafortune, S.: Introduction to discrete event systems. Kluwer Academic Publishers, Dordrecht (1999)

[CMM+04] Chatterjee, K., Ma, D., Majumdar, R., Zhao, T., Henzinger, T.A., Palsberg, J.: Stack size analysis for interrupt driven programs. Information and Computation 194(2), 144–174 (2004)

[CW02] Chen, H., Wagner, D.: Mops: an infrastructure for examining security properties of software. In: Proceedings of ACM Conference on Computer and Communications Security, pp. 235–244 (2002)

[DOTY96] Daws, C., Olivero, A., Tripakis, S., Yovine, S.: The tool KRONOS. In: Alur, R., Sontag, E.D., Henzinger, T.A. (eds.) HS 1995. LNCS, vol. 1066, pp. 208–219. Springer, Heidelberg (1996)

[EHRS00] Esparza, J., Hansel, D., Rossmanith, P., Schwoon, S.: Efficient algorithms for model checking pushdown systems. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 232–247. Springer, Heidelberg (2000)

[EJ91] Emerson, E.A., Jutla, C.S.: Tree automata, mu-calculus, and determinacy. In: Proceedings of the 32nd IEEE Symposium on Foundations of Computer Science, pp. 368–377 (1991)

[EKS03] Esparza, J., Kucera, A., Schwoon, S.S.: Model-checking LTL with regular valuations for pushdown systems. Information and Computation 186(2), 355–376 (2003)

[Eme90] Emerson, E.A.: Temporal and modal logic. In: van Leeuwen, J. (ed.) Handbook of Theoretical Computer Science, vol. B, pp. 995–1072. Elsevier Science Publishers, Amsterdam (1990)

[HJM+02] Henzinger, T.A., Jhala, R., Majumdar, R., Necula, G.C., Sutre, G., Weimer, W.: Temporal-Safety Proofs for Systems Code. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 526–538. Springer, Heidelberg (2002)

[Hol97] Holzmann, G.J.: The model checker SPIN. IEEE Transactions on Software Engineering 23(5), 279–295 (1997)

[HRS98] Henzinger, T.A., Raskin, J.-F., Schobbens, P.: The Regular Real-Time Languages. In: Larsen, K.G., Skyum, S., Winskel, G. (eds.) ICALP 1998. LNCS, vol. 1443, pp. 580–593. Springer, Heidelberg (1998)

[HU79] Hopcroft, J.E., Ullman, J.D.: Introduction to Automata Theory, Languages, and Computation. Addison-Wesley (1979)

[JMT99] Jensen, T., Le Metayer, D., Thorn, T.: Verification of control flow based security properties. In: Proceedings of the IEEE Symposium on Security and Privacy, pp. 89–103 (1999)

[JW96] Janin, D., Walukiewicz, I.: On the expressive completeness of the propositional mu- calculus with respect to monadic second order logic. In: Sassone, V., Montanari, U. (eds.) CONCUR 1996. LNCS, vol. 1119, pp. 263–277. Springer, Heidelberg (1996)

[Kam68] Kamp, J.: Tense Logic and the Theory of Linear Order. PhD thesis, University of California, Los Angeles (1968)

[KMV06a] Kumar, V., Madhusudan, P., Viswanathan, M.: Minimization, Learning, and Conformance Testing of Boolean Programs. In: Baier, C., Hermanns, H. (eds.) CONCUR 2006. LNCS, vol. 4137, pp. 203–217. Springer, Heidelberg (2006)

[KMV06b] Kumar, V., Madhusudan, P., Viswanathan, M.: Visibly pushdown languages for XML. Technical Report UIUCDCS-R-2006-2704, UIUC (2006)

[Knu67] Knuth, D.E.: A characterization of parenthesis languages. Information and Control 11(3), 269–289 (1967)

[Koz83] Kozen, D.: Results on the propositional mu-calculus. Theoretical Computer Science 27, 333–354 (1983)

[KPV02] Kupferman, O., Piterman, N., Vardi, M.Y.: Model Checking Linear Properties of Prefix-Recognizable Systems. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 371–385. Springer, Heidelberg (2002)

[Kur94] Kurshan, R.P.: Computer-aided Verification of Coordinating Processes: the automata-theoretic approach. Princeton University Press (1994)

[KVW00] Kupferman, O., Vardi, M.Y., Wolper, P.: An automata-theoretic approach to branching-time model checking. Journal of the ACM 47(2), 312–360 (2000)

[KVW01] Kupferman, O., Vardi, M.Y., Wolper, P.: Module checking. Information and Computation 164(2), 322–344 (2001)

[Lib05] Libkin, L.: Logics for Unranked Trees: An Overview. In: Caires, L., Italiano, G.F., Monteiro, L., Palamidessi, C., Yung, M. (eds.) ICALP 2005. LNCS, vol. 3580, pp. 35–50. Springer, Heidelberg (2005)

[LPY97] Larsen, K., Pettersson, P., Yi, W.: UPPAAL in a nutshell. Springer International Journal of Software Tools for Technology Transfer 1 (1997)

[LV02] Lee, E.A., Varaiya, P.: Structure and interpretation of signals and systems. Addison-Wesley (2002)

[McM93] McMillan, K.L.: Symbolic model checking: an approach to the state explosion problem. Kluwer Academic Publishers (1993)

[McN67] McNaughton, R.: Parenthesis grammars. Journal of the ACM 14(3), 490–500 (1967)

[Mil89]   Milner, R.: Communication and Concurrency. Prentice-Hall (1989)

[MS85]   Muller, D.E., Schupp, P.E.: The theory of ends, pushdown automata, and second-order logic. Theoretical Computer Science 37, 51–75 (1985)

[Nev02]   Neven, F.: Automata, Logic, and XML. In: Bradfield, J.C. (ed.) CSL 2002 and EACSL 2002. LNCS, vol. 2471, pp. 2–26. Springer, Heidelberg (2002)

[PS02]   Pappas, G.J., Simic, S.: Consistent abstractions of affine control systems. IEEE Transactions on Automatic Control 47(5), 745–756 (2002)

[PSL05]   IEEE 1850 standard for property specification language (PSL) (2005)

[Rab69]   Rabin, M.O.: Decidability of second order theories and automata on infinite trees. Transactions of the AMS 141, 1–35 (1969)

[Rep98]   Reps, T.: Program analysis via graph reachability. Information and Software Technology 40(11-12), 701–726 (1998)

[RHS95]   Reps, T., Horwitz, S., Sagiv, S.: Precise interprocedural dataflow analysis via graph reachability. In: Proceedings of the ACM Symposium on Principles of Programming Languages, pp. 49–61 (1995)

[Sch98]   Schmidt, D.A.: Data flow analysis is model checking of abstract interpretations. In: Proceedings of the 25th Annual ACM Symposium on Principles of Programming Languages, pp. 68–78 (1998)

[Sch04]   Schwentick, T.: Automata for XML – a survey. Technical report, University of Dortmund (2004)

[SH97]   Graf, S., Saidi, H.: Construction of abstract state graphs with PVS. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 72–83. Springer, Heidelberg (1997)

[SP81]   Sharir, M., Pnueli, A.: Two approaches to inter-procedural data-flow analysis. In: Program flow analysis: Theory and applications. Prentice-Hall, Englewood Cliffs (1981)

[Ste91]   Steffen, B.: Data flow analysis as model checking. In: Ito, T., Meyer, A.R. (eds.) TACS 1991. LNCS, vol. 526, pp. 346–365. Springer, Heidelberg (1991)

[Sti91]   Stirling, C.S.: Modal and temporal logic. In: Handbook of Logic in Computer Science, pp. 477–563. Oxford University Press (1991)

[Tho90]   Thomas, W.: Automata on infinite objects. In: van Leeuwen, J. (ed.) Handbook of Theoretical Computer Science, vol. B, pp. 133–191. Elsevier Science Publishers (1990)

[Tho02]   Thomas, W.: Infinite Games and Verification. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 58–64. Springer, Heidelberg (2002)

[VW94]   Vardi, M.Y., Wolper, P.: Reasoning about infinite computations. Information and Computation 115(1), 1–37 (1994)

[Wal01]   Walukiewicz, I.: Pushdown processes: Games and model-checking. Information and Computation 164(2), 234–263 (2001)

[WVS83]   Wolper, P., Vardi, M.Y., Sistla, A.P.: Reasoning about infinite computation paths. In: Proceedings of the 24th IEEE Symposium on Foundations of Computer Science, pp. 185–194 (1983)