Nontraditional Applications of Automata Theory

Moshe Y. Vardi

Dept. of Computer Science
Rice University
P.O.Box 1892
Houston, TX 77251-1892, USA
vardi@cs.rice.edu

Abstract

Finite automata have two traditional applications in computer science: modeling of finite-state systems and description of regular set of finite words. In the last few years, several new applications for finite-state automata have emerged, e.g., optimization of logic programs and specification and verification of protocols. These applications use finite-state automata to describe regular sets of infinite words and trees. I will describe such applications and will argue that we need change the way we teach automata theory.

1 Introduction

The theory of finite automata is one of the fundamental building blocks of theoretical computer science. It is covered in numerous textbooks and in any basic undergraduate curriculum in computer science. Since its introduction in the 1950's, the theory had numerous applications in practically all branches of computer science. In these applications, finite automata are typically used in one of two roles: as models or as descriptors; finite automata are very often the appropriate abstraction to model finite-state systems, and finite automata can be used as descriptors of regular languages.

Over the last decade, I have been involved in two areas of research in which automata theory is an essential source of tools: optimization of logic programs and specification and verification of protocols. In these applications, however, automata are not used in the same roles mentioned above. They are used as descriptors, but *not* of regular languages, i.e., regular sets of finite words. Instead, they are used to describe sets of more complex objects, namely, infinite words and trees. The theory of such automata had been a subject of research since the 1960s [Bu62, TW68, Ra69] and is also covered in some books [TB73, Ei74, GS84]; see [Ha85, Tho90] for recent surveys. Unlike, however, the theory of automata on finite words, the theory of automata on infinite words or trees is not widely known. Furthermore, my experience has been that many researchers are not very comfortable working with this theory.

In this paper I will describe two applications of the theory of automata on infinite words and trees: the first application, taken from [CV92] (see also [Va92]), is to optimization of logic database programs and the second application, taken from [VW86b], is to verification of finite-state programs. The description here is sketchy on some of the details; for more details the reader is referred to the original papers. My goal in this paper is twofold. First, I'd like to promote the theory of automata on infinite words and trees by demonstrating that it provides a powerful set of abstractions and tools to computer scientists. My hope is that this theory will become more widely known and applicable to a variety of problems in computer science. Second, and perhaps more important, I'd like to argue that we ought to change the way we teach automata theory. I will come back to this point at the concluding section of the paper.

2 Optimization of Datalog Programs

It has been recognized for some time that first-order database query languages are lacking in expressive power [AU79, GM78, Zl76]. Since then, many higher-order query languages have been investigated [AV89, Ch81, CH80, CH82, Im86, Va82]. A query language that has received considerable attention recently is *Datalog*, the language of logic programs (known also as Horn-clause programs) without function symbols [K90, Ul88, Ul89], which is essentially a fragment of fixpoint logic [CH85, Mo74].

The gain in expressive power does not, however, come for free; evaluating Datalog programs is harder than evaluating first-order queries [Va82]. Recent works have addressed the problems of finding efficient evaluation methods for Datalog programs (see survey in [BR86]) and developing optimization techniques for Datalog (see [MP91, Na89b, NRSU89]). The techniques to optimize evaluation of queries are often based on the ability to transform a query into an equivalent one that can be evaluated more efficiently [RSUV93]. Therefore, determining equivalence of queries is one of the most fundamental optimization problems. Naturally, the problem of determining equivalence of Datalog programs has received attention. Unfortunately, Datalog program equivalence is undecidable [Shm87].

Since the source of the difficulty in evaluating Datalog programs is their recursive nature, the first line of attack in trying to optimize such programs is to eliminate the recursion. The following example is from [Na89a].

Example 1. Consider the following Datalog program Π_1:

$$buys(X, Y) :- likes(X, Y).$$
$$buys(X, Y) :- trendy(X), buys(Z, Y).$$

It can be shown that Π_1 is equivalent to the following nonrecursive program.

$$buys(X, Y) :- likes(X, Y).$$
$$buys(X, Y) :- trendy(X), likes(Z, Y).$$

Consider, on the other hand, the following Datalog program Π_2:

$$buys(X, Y) : -likes(X, Y).$$
$$buys(X, Y) : -knows(X, Z), buys(Z, Y).$$

It can be shown that Π_2 is not equivalent to the following nonrecursive program:

$$buys(X, Y) : -likes(X, Y).$$
$$buys(X, Y) : -knows(X, Z), likes(Z, Y).$$

In fact, Π_2 is inherently recursive, i.e., it is not equivalent to any nonrecursive program. ∎

Thus, a problem of special interest is that of determining the equivalence of a given recursive Datalog program to a given nonrecursive[1] program. This problem is the main focus of this section.

A nonrecursive program can be rewritten as a union of conjunctive queries. Thus, containment of a nonrecursive program in a recursive program can be reduced to the containment of a conjunctive query in a recursive program. The latter problem was shown to be decidable; in fact it is EXPTIME-complete [CK86, CLM81, Sa88b]. Thus, what was left open is the other direction, i.e., the problem of determining whether a recursive program is contained in a nonrecursive program. Chaudhuri and Vardi [CV92] attacked this problem by investigating the containment of recursive programs in unions of conjunctive queries. They showed that containment of recursive programs in unions of conjunctive queries is decidable. It follows that equivalence to nonrecursive programs is decidable (see also [Mey93]).

The key idea underlying the result is the observation that a recursive program can be viewed as an infinite union of conjunctive queries. These conjunctive queries can be represented by proof trees, and the set of proof trees corresponding to a given recursive program can be represented by a tree automaton. This representation enables us to reduce containment of recursive programs in unions of conjunctive queries to containment of tree automata, which is known to be decidable in exponential time [Se90]. The size of the tree automata obtained in the reduction is exponential in the size of the input; as a result, we obtain a doubly-exponential time upper bound for containment in unions of conjunctive queries.

2.1 Preliminaries

Automata on Trees: Let N denote the set of positive integers. The variables x and y denote elements of N^*. A *tree* τ is a subset of N^*, such that if $xi \in \tau$, where $x \in N^*$ and $i \in N$, then also $x \in \tau$ and and if $i > 1$ then also $x(i-1) \in \tau$. The elements of τ are called *nodes*. If x and xi are nodes of τ, then x is the *parent* of xi and xi is the *child* of x. The node x is a *leaf* if it has no children. By

[1] A Datalog program is nonrecursive if the dependency graph among the predicates is acyclic.

definition, the empty sequence ϵ is a member of every tree; it is called the *root*. A *branch* of τ is a subset $\Xi \subseteq \tau$ such that $\epsilon \in \Xi$ and for each $x \in \Xi$ either x is a leaf or there is a unique i such that $xi \in \Xi$.

A *Σ-labeled tree*, for a finite alphabet Σ, is a pair (τ, π), where τ is a tree and $\pi : \tau \to \Sigma$ assigns to every node a label. *Labeled trees* are often referred to as *trees*; the intention will be clear from the context. The set of Σ-labeled trees is denoted $trees(\Sigma)$.

Proviso: In this section we consider only finite trees.

A *tree automaton* A is a tuple $(\Sigma, S, S_0, \delta, F)$, where Σ is a finite alphabet, S is a finite set of states, $S_0 \subseteq S$ is a set of initial states, $F \subseteq S$ is a set of accepting states, and $\delta : S \times \Sigma \to 2^{S^*}$ is a transition function such that $\delta(s, a)$ is finite for all $s \in S$ and $a \in \Sigma$. A *run* $r : \tau \to S$ of A on a Σ-labeled tree (τ, π) is a labeling of τ by states of A, such that the root is labeled by a initial state and the transitions obey the transition function δ; that is, $r(\epsilon) \in S_0$, and if x is not a leaf and x has k children, then $\langle r(x1), \ldots, r(xk) \rangle \in \delta(r(x), \pi(x))$. If for every leaf x of τ there is a tuple $\langle s_1, \ldots, s_l \rangle \in \delta(r(x), \pi(x))$ such that $\{s_1, \ldots, s_l\} \subseteq F$, then r is *accepting*. A *accepts* (τ, π) if it has an accepting run on (τ, π). The *tree language* of A, denoted $T(A)$, is the set of trees accepted by A.

An important property of tree automata is their closure under Boolean operations.

Proposition 1. [Cos72] *Let A_1, A_2 be automata over an alphabet Σ. Then there are automata A_3, A_4, and A_5 such that $L(A_3) = \Sigma^* - L(A_1)$, $L(A_4) = L(A_1) \cap L(A_2)$, and $L(A_5) = L(A_1) \cup L(A_2)$.*

The constructions for union and intersection involve only a polynomial blowup in the size of the automata, while complementation may involve an exponential blow-up in the size of the automaton.

The *emptiness problem* for tree automata is to decide, given a tree automaton A, whether $T(A)$ is empty.

Proposition 2. [Do70, TW68] *The emptiness problem for tree automata is decidable in polynomial time.*

Proof: Let $A = (\Sigma, S, S_0, \delta, F)$ be the given tree automaton. Let $accept(A)$ be the minimal set of states in S such that

- $F \subseteq accept(A)$, and
- if s is a state such that there are a letter $a \in \Sigma$ and a transition $\langle s_1, \ldots, s_k \rangle \in \delta(s, a) \cap accept(A)^*$, then $s \in accept(A)$.

It is easy to see that $T(A)$ is nonempty iff $S_0 \cap accept(A) \neq \emptyset$. Intuitively, $accept(A)$ is the set of all states that label the roots of accepting runs. Thus, $T(A)$ is empty precisely when no initial state is in $accept(A)$. The claim follows, since $accept(A)$ can be computed bottom-up in polynomial time. ∎

A problem related to emptiness is the *containment problem*, which is to decide, given tree automata A_1 and A_2, whether $T(A_1) \subseteq T(A_2)$. Note that

$TL(A_1) \subseteq T(A_2)$ iff $T(A_1) \cap \overline{T(A_2)} = \emptyset$. Thus, by Proposition 1, the containment problem is reducible to the emptiness problem, though the reduction may be computationally expensive.

Proposition 3. [Se90] *The containment problem for tree automata is EXPTIME-complete.*

Conjunctive Queries and Datalog: A *conjunctive query* is a positive existential conjunctive first-order formula, i.e., the only propositional connective allowed is $\wedge$ and the only quantifier allowed is $\exists$. Without loss of generality, we can assume that conjunctive queries are given as formulas $\theta(x_1, \ldots, x_k)$ of the form $(\exists y_1, \ldots, y_m)(a_1 \wedge \ldots a_n)$ with free variables among $x_1, \ldots, x_k$, where the a_i's are atomic formulas of the form $p(z_1, \ldots, z_l)$ over the variables $x_1, \ldots, x_k, y_1, \ldots, y_m$. For example, the conjunctive query $(\exists y)(E(x, y) \wedge E(y, z))$ is satisfied by all pairs $\langle x, z \rangle$ such that there is a path of length 2 between x and z. The free variables are also called *distinguished variables*. We distinguish between variables and *occurrences* of variables in a conjunctive query, but we only consider occurrences of variables in the atomic formulas of the query. For example, the variables x and y have each two occurrences in $(\exists y)(E(x, y) \wedge E(y, z))$. An occurrence of a distinguished variable in a conjunctive query is called a *distinguished occurrence*. A union of conjunctive queries is a disjunction

$$\bigvee_{i=1}^{s} \theta_i(x_1, \ldots, x_k)$$

of conjunctive queries.

A union of conjunctive queries $\Theta(x_1, \ldots, x_k)$ can be applied to a database D. The result

$$\Theta(D) = \{(a_1, \ldots, a_k) | D \models \Theta(a_1, \ldots, a_k)\}$$

is the set of k-ary tuples that satisfy Θ in D. If Θ has no distinguished variables, then it is viewed as a Boolean query; the result is either the empty relation or the relation containing the 0-ary tuple.

A (Datalog) program consists of a set of Horn clauses. A predicate that occurs in head of a rule is called an *intensional* (IDB) predicate. The rest of the predicates are called *extensional* (EDB) predicates. Let Π be a Datalog program. Let $Q^i_\Pi(D)$ be the collection of facts about an IDB predicate Q that can be deduced from a database D by at most i applications of the rules in Π and let $Q^\infty_\Pi(D)$ be the collection of facts about Q that can be deduced from D by any number of applications of the rules in Π, that is,

$$Q^\infty_\Pi(D) = \bigcup_{i \geq 0} Q^i_\Pi(D).$$

If Q is 0-ary, then Q^∞_Π is viewed as a Boolean query.

We say that the program Π with goal predicate Q is *contained* in a union of conjunctive queries Θ if $Q^\infty_\Pi(D) \subseteq \Theta(D)$ for each database D. It is known (cf.

[MUV84, Na89a]) that the relation defined by an IDB predicate in a Datalog program Π, i.e., $Q_\Pi^\infty(D)$, can be defined by an *infinite* union of conjunctive queries. That is, for each IDB predicate Q there is an infinite sequence $\varphi_0, \varphi_1, \ldots$ of conjunctive queries such that for every database D, we have $Q_\Pi^\infty(D) = \bigcup_{i=0}^\infty \varphi_i(D)$. The φ_i's are called the *expansions* of Q.

A predicate P *depends* on a predicate Q in a program Π, if Q occurs in the body of a rule r of Π and P is the predicate at the head of r. The *dependence graph* of Π is a directed graph whose nodes are the predicates of Π, and whose edges captures the dependence relation, i.e., there is an edge from Q to P if P depends on P. A program Π is *nonrecursive* if its dependence graph is acyclic, i.e., no predicates depends recursively on itself. It is well-known that a nonrecursive program has only finitely many expansions (up to renaming of variables). Thus, a nonrecursive program is equivalent to a union of conjunctive queries.

Containment of Conjunctive Queries: Let $\theta(x_1, \ldots, x_k)$ and $\psi(x_1, \ldots, x_k)$ are two conjunctive queries with the same vector of distinguished variables. We say that θ is *contained* in ψ if $\theta(D) \subseteq \psi(D)$ for each database D, i.e., if the following implication is valid

$$\forall x_1 \ldots \forall x_k (\theta(x_1, \ldots, x_k) \rightarrow \psi(x_1, \ldots, x_k))$$

A *containment mapping* from a conjunctive query ψ to a conjunctive query θ is a renaming of variables subject to the following constraints: (a) every distinguished variable must map to itself, (b) a constant must map to itself, and (c) after renaming, every literal in ψ must be among the literals of θ. Conjunctive-query containment can be characterized in terms of containment mappings (cf. [Ul89]).

Theorem 4. *A conjunctive query $\theta(x_1, \ldots, x_k)$ is contained in a conjunctive query $\psi(x_1, \ldots, x_k)$ iff there is a containment mapping from ψ to θ.*

It will be convenient to view a containment mapping h from ψ to θ as a mapping from occurrences of variables in ψ to occurrences to variables in θ. Such a mapping has the property that v_1 and v_2 are occurrences of the same variable in ψ, then $h(\mathsf{v}_1)$ and $h(\mathsf{v}_2)$ are occurrences of the same variable in θ.

Expansion Trees: Expansions can be described in terms of *expansion trees*. The nodes of an expansion tree for a Datalog program Π are labeled by pairs of the form (α, ρ), where α is an IDB atom and ρ is an instance of a rule r of Π such that the head of ρ is α. The atom labeling a node x is denoted α_x and the rule labeling a node x is denoted ρ_x. In an expansion tree for an IDB predicate Q, the root is labeled by a Q-atom. Consider a node x, where α_x is the atom $R(\mathbf{t})$, ρ_x is the rule

$$R(\mathbf{t}) :- R_1(\mathbf{t}^1), \ldots, R_m(\mathbf{t}^m),$$

and the IDB atoms in the body of the rule are $R_{i_1}(\mathbf{t}^{i_1}), \ldots, R_{i_l}(\mathbf{t}^{i_l})$. Then x has children $x1, \ldots, xl$ labeled with the atoms $R_{i_1}(\mathbf{t}^{i_1}), \ldots, R_{i_l}(\mathbf{t}^{i_l})$. In particular, if

all atoms in ρ_x are EDB atoms, then x must be a leaf. The query corresponding to an expansion tree is the conjunction of all EDB atoms in ρ_x for all nodes x in the tree, with the variables in the root atom as the free variables. Thus, we can view an expansion tree τ as a conjunctive query. Let $trees(Q, \Pi)$ denote the set of expansion trees for an IDB predicate Q in Π. (Note that $trees(Q, \Pi)$ is an infinite set.) Then for every database D, we have

$$Q_{\Pi}^{\infty}(D) = \bigcup_{\tau \in trees(Q,\Pi)} \tau(D).$$

It follows that Π is contained in a conjunctive query θ of there is a containment mapping from θ to each expansion tree τ in $trees(Q, \Pi)$, i.e., a mapping, which maps distinguished variables to distinguished variables and maps the atoms of θ to atoms in the bodies of rules labeling nodes of τ.

Of particular interest are expansion trees that are obtained by "unfolding" the program Π. An expansion tree τ of a Datalog program Π is an *unfolding expansion tree* if it satisfies the following conditions: (a) the atom labeling the root is the head of a rule in Π, and (b) if a node x is labeled by (α_x, ρ_x), then the variables in the body of ρ_x either occur in α_x or they do not occur in the label of any node above x. Intuitively, an unfolding expansion tree is obtained by starting with a head of a rule in Π as the atom labeling the root, and then creating children by unifying an atom labeling a node with a "fresh" copy of a rule in Π. Note that if a variable v occur in the atom labeling a node x but not in the atoms labeling the children of x, then v will not occur in the label of any descendant of x.

We denote the collection of unfolding expansion trees for an IDB predicate Q in a program Π by $u\_trees(Q, \Pi)$. It is easy to see that every expansion tree can be obtained by renaming variables in an unfolding expansion tree. Thus, every expansion tree, viewed as a conjunctive query, is contained in an unfolding expansion tree. The following proposition follows immediately.

Proposition 5. *Let Π be a program with a goal predicate Q. For every database D, we have*

$$Q_{\Pi}^{\infty}(D) = \bigcup_{\tau \in u\_trees(Q,\Pi)} \tau(D).$$

Example 2. Figure 1 shows expansion trees for the IDB predicate p in the following transitive closure program.

$$r1: \ p(X, Y) :- e(X, Z), p(Z, Y)$$
$$r0: \ p(X, Y) :- e'(X, Y)$$

Note that the variable X is re-used in the child of the root of the expansion tree, while a new variable W is used instead of X in the child of the root of the unfolding expansion tree. ∎

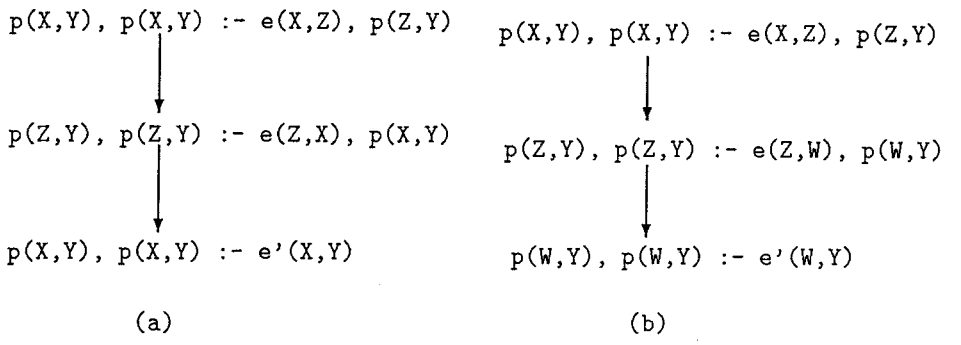

Fig. 1. (a) Expansion Tree (b) Unfolding Expansion Tree

2.2 Proof Trees

The basic idea behind *proof trees* is to describe expansion trees using a finite number of labels. We bound the number of labels by bounding the set of variables that can occur in labels of nodes in the tree. If r is a rule of a Datalog program Π, then let $var\_num(r)$ be the number of variables occurring in IDB atoms in r (head or body). Let $var\_num(\Pi)$ be twice the maximum of $var\_num(r)$ for all rules r in Π. Let $var(\Pi)$ be the set $\{x_1, \ldots, x_{var\_num(\Pi)}\}$. A proof tree for Π is simply an expansion tree for Π all of whose variables are from $var(\Pi)$. We denote the set of proof trees for a predicate Q of a program Π by $p\_trees(Q, \Pi)$.

The intuition behind proof tree is that variables are re-used. In an unfolding expansion tree, when we "unfold" a node x we take a "fresh" copy of a rule r in Π. In a proof tree, we take instead an instance of r over $var(\Pi)$. Since the number of variables in $var(\Pi)$ is twice the number of variables in any rule of Π, we can instantiate the variables in the body of r by variables different from those in the goal α_x.

Example 3. Figure 2 describes an unfolding expansion tree and a proof tree for the IDB predicate p in the transitive-closure program of Example 2. In the proof tree, instead of using a new variable W, we re-use the variable X. ∎

A proof tree represents an expansion tree where variables are re-used. In other words, the same variable is used to represent a set of distinct variables in the expansion tree. Intuitively, to reconstruct an expansion tree for a given proof tree, we need to distinguish among occurrences of variables.

Let x_1 and x_2 be nodes in a proof tree τ, with a lowest common ancestor x, a and let v_1 and v_2 be occurrences, in x_1 and x_2, respectively, of a variable v. We

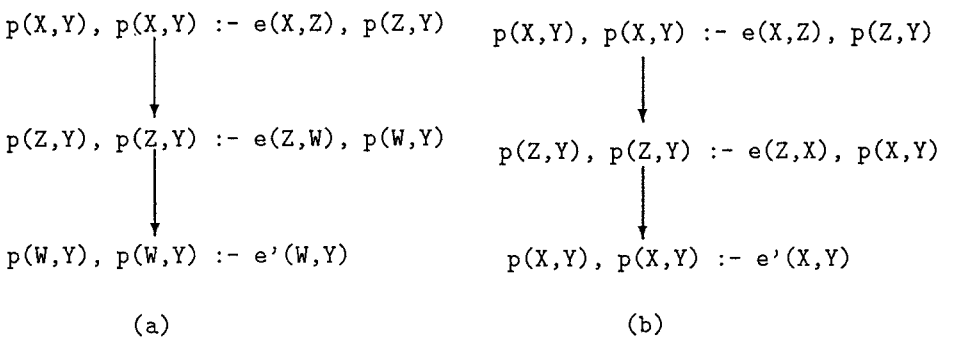

Fig. 2. (a) Unfolding Expansion Tree (b) Proof Tree

say that v_1 and v_2 are *connected* in τ if the goal of every node, except perhaps for x, on the simple path connecting x_1 and x_2 has an occurrence of v. We say that an occurrence v of a variable v in τ is a *distinguished occurrence* if it is connected to an occurrence of v in the atom labeling the root of τ. From this definition, it follows that connectedness is an equivalence relation and it partitions the occurrences of variables in the proof tree. We denote the equivalence class of an occurrence v of a variable v in a proof tree τ by $[v]_\tau$. We will omit τ when it is clear from the context.

Example 4. Consider the proof tree in Figure 2. The occurrences of the variable Y in the root and in the interior node are connected. Both occurrences of Y are distinguished. The occurrences of the variable X in the root and in the leaf are not connected. The occurrence of X in the root is distinguished, but the occurrence of X in the leaf is not distinguished. ∎

Every proof tree corresponds to an expansion tree and hence to an expansion. We want to define containment mappings from conjunctive queries to proof trees such that there is a containment mapping from a conjunctive query to a proof tree iff there is a containment mapping from the conjunctive query to the expansion corresponding to the proof tree. The definition should force a variable in the conjunctive query to map to a unique variable in the expansion corresponding to the proof tree.

A *strong* containment mapping from a conjunctive query θ to a proof tree τ is a containment mapping h from θ to τ with the following properties:

- h maps distinguished occurrences in θ to distinguished occurrences in τ, and
- if v_1 and v_2 are two occurrences of a variable v in θ, then the occurrences $h(v_1)$ and $h(v_2)$ in τ are connected.

We now relate containment of programs and strong containment mappings.

Theorem 6. *Let Π be a program with goal predicate Q, and let $\Theta = \cup_i \theta_i$ be a union of conjunctive queries. Then Π is contained in Θ if and only if for every proof tree $\tau \in p\_trees(Q, \Pi)$ there is a strong containment mappings from some θ_i to τ.*

We will use the characterization above to obtain upper bound for containment of programs in conjunctive queries.

2.3 Upper Bounds

The main feature of proof trees, as opposed to expansion trees, is the fact that the numbers of possible labels is finite; it is actually exponential in the size of Π. Because the set of labels is finite, the set of proof trees $p\_trees(Q, \Pi)$, for an IDB predicate Q in a program Π, can be described by a tree automaton.

Proposition 7. *Let Π be a Datalog program with a goal predicate Q. Then there is an automaton $A^{p\_trees}_{Q,\Pi}$, whose size is exponential in the size of Π, such that $T(A^{p\_trees}_{Q,\Pi}) = p\_trees(Q, \Pi)$.*

Proof: We sketch the construction of the automaton

$$A^{p\_trees}_{Q,\Pi} = (\Sigma, \mathcal{I} \cup \{accept\}, \mathcal{I}_Q, \delta, \{accept\})$$

The state set $\mathcal{I}$ is the set of all IDB atoms with variables among $var(\Pi)$. The initial-state set is the set of all atoms $Q(\mathbf{s})$, where the variables of $\mathbf{s}$ are in $var(\Pi)$. The alphabet $\Sigma = \mathcal{I} \times \mathcal{R}$ where $\mathcal{R}$ is the set of instances of rules of Π over $var(\Pi)$. The transition function δ is constructed as follows:

- Let ρ be a rule instance

$$R(\mathbf{t}) :- R_1(\mathbf{t}^1), \ldots, R_m(\mathbf{t}^m),$$

  in $\mathcal{R}$, where the IDB atoms in the body of the rule are $R_{i_1}(\mathbf{t}^{i_1}), \ldots, R_{i_l}(\mathbf{t}^{i_l})$. Then $\langle R_{i_1}(\mathbf{t}^{i_1}), \ldots, R_{i_l}(\mathbf{t}^{i_l}) \rangle \in \delta(R(\mathbf{t}), (R(\mathbf{t}), \rho))$.
- Let ρ be a rule instance

$$R(\mathbf{t}) :- R_1(\mathbf{t}^1), \ldots, R_m(\mathbf{t}^m),$$

  in $\mathcal{R}$, where all atoms in the body of the rule are EDB atoms. Then $\langle accept \rangle \in \delta(R(\mathbf{t}), (R(\mathbf{t}), \rho))$.

It is easy to see that the number of states and transitions in the automaton is exponential in the size of Π.

∎

We now show that strong containment of proof trees in a conjunctive query can be checked by tree automata as well.

Proposition 8. *Let Π be a Datalog program Π with goal predicate Q, and let θ be a conjunctive query. Then there is an automaton $A^{\theta}_{Q,\Pi}$, whose size is exponential in the size of Π and θ, such that $T(A^{\theta}_{Q,\Pi})$ is the set of proof trees τ in $p\_trees(Q,\Pi)$ where there is a strong containment mapping from θ to τ.*

Proof: We sketch the construction of $A^{\theta}_{Q,\Pi}$.

Every state of the automaton includes a subset of atoms of θ that has not yet been strongly mapped to τ. Such unmapped atoms may share variables with atoms that have already been mapped. Therefore, also included in the state description is a partial mapping that indicates the images of the mapped variables. A transition on input symbol (α, ρ) results in mapping of zero or more unmapped atoms to the body of ρ. The remainder of the unmapped atoms are partitioned among the sequence of states prescribed by the transition.

The automaton $A^{\theta}_{Q,\Pi}$ is $(\Sigma, S \cup \{accept\}, S_Q, \delta, \{accept\})$. The sets $\mathcal{I}$ and $\Sigma = \mathcal{I} \times \mathcal{R}$ are as in the proof of Proposition 7. We assume that the conjunctive query θ has a set of variables V_θ. The state set S is the set $\mathcal{I} \times 2^{\Theta} \times 2^{V_\Theta \times var(\Pi)}$. The second component in S represents the collection of subsets (of atoms) of θ and the final component contains the set of partial mappings from V_θ to $var(\Pi)$. The start-state set S_Q Consists of all triples $(Q(\mathbf{s}), \theta, M_{\theta,\mathbf{s}})$, where the variable of s are in $var(\Pi)$ and $M_{\theta,\mathbf{s}}$ is a mapping of the distinguished variables of θ into the variables of s. The transition function is constructed as follows:

- Let ρ be a rule instance

$$R(\mathbf{t}) :- R_1(\mathbf{t}^1), \ldots, R_m(\mathbf{t}^m),$$

in $\mathcal{R}$, where the IDB atoms in the body of the rule are $R_{i_1}(\mathbf{t}^{i_1}), \ldots, R_{i_l}(\mathbf{t}^{i_l})$. Then

$$\langle (R_{i_1}(\mathbf{t}^{i_1}), \beta_1, M') \ldots, (R_{i_l}(\mathbf{t}^{i_l}), \beta_l, M') \rangle \in \delta((R(\mathbf{t}), \beta, M), (R(\mathbf{t}), \rho))$$

if the following hold:

1. β can be partitioned into $\beta', \beta_1, \ldots, \beta_l$, where β' is mapped to atoms in the body of ρ by a mapping $M_{\beta'}$ that is consistent with M,
2. M' is a partial mapping that extends M and is consistent with $M_{\beta'}$.
3. β_j and β_k can share a variable only if this variable is in the domain of M' and its image is in both $\mathbf{t}^{i_j}$ and $\mathbf{t}^{i_k}$.
4. If a variable occurs in β_j and it is in the domain of M', then its image is in $\mathbf{t}^{i_j}$.

- Let ρ be a rule instance

$$R(\mathbf{t}) :- R_1(\mathbf{t}^1), \ldots, R_m(\mathbf{t}^m),$$

in $\mathcal{R}$, where all atoms in the body of the rule are EDB atoms. Then $\langle accept \rangle \in \delta((R(\mathbf{t}), \beta, M), (R(\mathbf{t}), \rho))$ if there is a mapping that extends M and maps all literals in β to atoms in the body of ρ.

It is easy to see that the number of states and transition in the automaton is exponential in the size of Π and θ. ∎

We can now reduce the containment problem for Datalog programs in unions of conjunctive queries to an automata-theoretic problem.

Theorem 9. *Let Π be a program with goal predicate Q, and let $\Theta = \cup_i \theta_i$ be a union of conjunctive queries. Then Π is contained in Θ if and only if*

$$T(A^{p\_trees}_{Q,\Pi}) \subseteq \bigcup_i T(A^{\theta_i}_{Q,\Pi}).$$

Proof: By Theorem 6, Π is contained in Θ if and only if for every proof tree $\tau \in p\_trees(Q, \Pi)$ there is a strong containment mappings from some θ_i to τ. By Propositions 7 and 8, the latter condition is equivalent to

$$T(A^{p\_trees}_{Q,\Pi}) \subseteq \bigcup_i T(A^{\theta_i}_{Q,\Pi}).$$

∎

Theorem 10. *Containment of a recursive Datalog program in a union of conjunctive queries is in 2EXPTIME.*

Proof: By Proposition 1, we can obtain an automaton $A^{\Theta}_{Q,\Pi}$, whose size is exponential in the size of Π and Θ, such that

$$T(A^{\Theta}_{Q,\Pi}) = \bigcup_i T(A^{\theta_i}_{Q,\Pi}).$$

Thus, by Theorem 9, containment in a union of conjunctive queries can be reduced to containment of tree (resp. word) automata of exponential size. Since containment of tree automata can be decided in exponential time (Proposition 3), the result follows. ∎

One may wonder whether the algorithm given by Theorem 10 can be improved. It turns out that the algorithm is essentially optimal, since it is shown in [CV92] that the containment of recursive programs in union of conjunctive queries is complete for 2EXPTIME.

3 Verification of Finite-State Programs

While *program verification* was always a desirable, but never an easy task, the advent of *concurrent programming* has made it significantly more necessary and difficult. Indeed, the conceptual complexity of concurrency increases the likelihood of the program containing errors. To quote from [OL82]: "There is rather large body of sad experience to indicate that a concurrent program can withstand very careful scrutiny without revealing its errors."

The first step in program verification is to come up with a *formal specification* of the program. One of the more widely used specification languages for concurrent programs is *temporal logic*, which was introduced by Pnueli [Pn77] (c.f. [MP92]). Temporal logic comes in two varieties: linear time and branching time ([EH86, La80]); we concentrate here on linear time. A linear temporal specification describes the computations of the program, so a program *meets* the specification (is *correct*) if all its computations satisfy the specification.

In the traditional approach to concurrent program verification (cf. [HO83, MP81, OL82]) the correctness of the program is expressed as a formula in first-order temporal logic. To prove that the program is correct, one has to prove that the correctness formula is a theorem of a certain deductive system. Constructing this proof is done manually and is usually quite difficult. It often requires an intimate understanding of the program. Furthermore, there is no hope of constructing the proof completely algorithmically. The only extent of automation that one can hope for, is to have the proof *checked* by a machine and possibly to have some limited heuristic help in finding the proof.

A different approach was introduced in [CES86, QS82] for *finite-state* programs, i.e., programs in which the variables range over finite domains. The significance of this class follows from the fact that a significant number of the communication and synchronization protocols studied in the literature are in essence finite-state programs. Since each state is characterized by a finite amount of information, this information can be described by certain *atomic propositions*. This means that a finite-state program can be viewed as a finite *propositional Kripke structure* and that it can be specified using *propositional* temporal logic. Thus, to verify the correctness of the program, one has only to check that the program, viewed as a finite Kripke structure, satisfies (is a model of) the propositional temporal logic specification. This approach is called *verification by model checking*. The advantage of the model-checking approach, described in [AK86] as "one of the most exciting developments in the theory of program correctness", is that it can be done algorithmically. Model checking was originally developed for branching time [CES86, QS82], but was later extended also to linear time [LP85]. See [CG87] for a more recent survey.

In view of the attractiveness of the model-checking approach, one would like to extend its applicability as much as possible. Unfortunately, the *tableau-based* model-checking algorithms in the literature (cf. [LP85]) involve the intricacies of the logic at hand and do not make intuitively clear what extensions are possible. On the other hand, an approach based on the connection between propositional temporal logic and automata theory seems to be more extensible.

The connection between propositional temporal logic and automata theory has been quite extensively studied [GPSS80, Ka68, LPZ85, Pe85, Si83, SVW87, VW94]. This connection is based on the fact that a computation is essentially an infinite sequence of states. Since every state is completely described by a finite set of atomic propositions, a computation can be viewed as an infinite word over the alphabet of truth assignments to the atomic propositions. One of the most enlightening results in this area is the fact that temporal logic formulas

can be viewed as *finite-state acceptors*. More precisely, given any propositional temporal formula, one can construct a finite automaton on infinite words that accepts precisely the sequences satisfied by the formula [VW94].

To use the above connection, we view a finite-state program as a *finite-state generator* of infinite words. Thus, if P is the program and φ is the specification, then P meets φ if every infinite word generated by P, viewed as a finite-state generator, is accepted by φ, viewed as a finite-state acceptor. This reduces the verification problem to a purely automata-theoretic problem: the problem of determining whether the language $L(P) - L(\varphi)$ is empty, where $L(P)$ is the language generated by P and $L(\varphi)$ is the language accepted by φ.

There are a number of benefits from this approach. First, we obtain a very simple and clean algorithm for model checking for linear time temporal logic (compared to the algorithm in [LP85]). This algorithm makes the complexity bounds of [LP85] obvious and even lets us extend them. We can easily show that the space complexity of model checking is polynomial in the size of the specifications and polylogarithmic (in fact $O(log^2 n)$) in the size of the model. Note that this is quite significant as the programs to which model checking is applied can be very large and using even linear space could make implementation difficult. Another aspect of model checking that is made much more straightforward is the introduction of a fairness assumption on the execution of the program as is done in [CES86, EL85a, EL85b, LP85].

A second benefit of our approach is that it makes extending model checking to more expressive temporal logics easy. The standard temporal logic, which consists of the connectives X ("next"), G ("always"), and U ("until"), either cannot express certain properties [Wo83] or cannot express them conveniently [BK84, KVR83, LPZ85]. For that reason, *extended temporal logics* was introduced in [Wo83, VW94], and *past* temporal connectives were introduced in [BK84, KVR83, LPZ85]. (Tableau-based model checking was extended to temporal logic with past connectives in [LP85]). To extend our model-checking algorithm to these logics, one only needs to show that given a formula in these logics, one can build a finite automaton on infinite words that accepts the models of the formula. The rest of the algorithm goes unchanged.

3.1 Preliminaries

Automata on Infinite Words The type of finite automata on infinite words we consider is the one defined by Büchi [Bu62]. A *Büchi automaton* is a tuple $A = (\Sigma, S, \rho, s_0, F)$, where

- Σ is an alphabet,
- S is a set of states,
- $s_0 \in S$ is the initial state, and
- $\rho : S \times \Sigma \rightarrow 2^S$ is a nondeterministic transition function,
- $F \subseteq S$ is a set of accepting states.

A *run* of A over a infinite word $w = a_0 a_1 \ldots$, is a sequence $s_0, s_1, \ldots$, where $s_i \in \rho(s_{i-1}, a_{i-1})$, for all $i \geq 1$. A run $s_0, s_1, \ldots$ is *accepting* if there is some ac-

cepting state that repeats infinitely often, i.e., for some $s \in F$ there are infinitely many i's such that $s_i = s$. The infinite word w is *accepted* by A if there is an accepting run of A over w. The set of infinite words accepted by A is denoted $L(A)$.

An important property of tree automata is their closure under Boolean operations.

Proposition 11. [Ch74, Sa88, SVW87] *Let A_1, A_2 be Büchi automata over an alphabet Σ. Then there are Büchi automata A_3, A_4, and A_5 such that $L(A_3) = \Sigma^\omega - L(A_1)$, $L(A_4) = L(A_1) \cap L(A_2)$, and $L(A_5) = L(A_1) \cup L(A_2)$.*

The constructions for union and intersection involve only a polynomial blowup in the size of the automata, while complementation may involve an exponential blow-up in the size of the automaton.

The following theorem states some important results about the *emptiness problem* for Büchi automata, i.e., the problem of determining for a given Büchi automaton A whether A accepts *no* word.

Theorem 12.

1. [EL85a, EL85b] *The emptiness problem for Büchi automata is solvable in linear time.*
2. [VW94] *The emptiness problem for Büchi automata is complete for NLOGSPACE.*

(The second clause in the Theorem uses the equality NLOGSPACE=co-NLOGSPACE; cf. [Im88].)

We also consider *alternating* Büchi automata. An alternating Büchi automaton is a tuple $A = (\Sigma, S, \rho, S_0, F)$, where

- Σ is an alphabet,
- S is a set of states,
- $s_0 \in S$ is the initial state, and
- $\rho : S \times \Sigma \to \mathcal{B}^+(S)$ is a transition function,
- $F \subseteq S$ is a set of designated states.

Here $\mathcal{B}^+(S)$ is the set of positive Boolean formulas over S (i.e., Boolean formulas built from elements in S using $\wedge$ and $\vee$), where we also allow the formulas **true** and **false**.

A *run tree* of A over a infinite word $w = a_0 a_1 \ldots$, is an infinite S-labeled tree r such that $r(\epsilon) = s_0$ and the following holds:

> if $|x| = i$, $r(x) = s$, and $\rho(s, a_i) = \theta$, then x has as children nodes $x1, \ldots, xk$ for some $k \leq |S|$, and the truth assignment that assigns *true* to the states in $r(x1), \ldots, r(xk)$ and assigns *false* to the other states satisfy θ.

For example, if $\rho(s_0, a_0)$ is $(s_1 \vee s_2) \wedge (s_3 \vee s_4)$, then the nodes of the run tree at level 1 includes the label s_1 or the label s_2 and also include the label s_3 or the label s_4. Note that if $\rho(s, a_i) = \mathbf{true}$, then x need not have any children, and we cannot have $\rho(s, a_i) = \mathbf{false}$, since **false** is not satisfiable. Note also that a Büchi automaton is an alternating Büchi automaton where all transitions are simple disjunctions. (The formalization of alternation in terms of Boolean transitions is in the spirit of [BL80, Le81], as opposed to the distinction between existential and universal states in [CKS81].)

A run tree r is *accepting* if every infinite branch in r includes infinitely many labels in F. The infinite word w is *accepted* by A if there is an accepting run of A over w. The set of infinite words accepted by A is denoted $L(A)$.

What is the relationship between Büchi automata and alternating Büchi automata? This is answered by the following theorem:

Theorem 13. [MH84] *Every alternating Büchi automaton A is equivalent (i.e., accepts the same language) as some Büchi automaton A'. Furthermore, the number of states of A' is at most exponential in the number of states of A.*

Temporal Logics; Linear time propositional temporal logic (PTL) has been defined in a number of publications [GPSS80, Pn77]. Formulas of PTL are built from a set *Prop* of atomic propositions and are closed under the application of Boolean connectives, the unary temporal connective X (next), and the binary temporal connective U (until). PTL is interpreted over *computations*. A computation is a function $\pi : \omega \rightarrow 2^{Prop}$, which assigns truth values to the elements of *Prop* at each time instant (natural number). For a computation π and a point $i \in \omega$, we have that:

- $\pi, i \models p$ for $p \in Prop$ iff $p \in \pi(i)$.
- $\pi, i \models \xi \wedge \psi$ iff $\pi, i \models \xi$ and $\pi, i \models \psi$.
- $\pi, i \models \neg\varphi$ iff not $\pi, i \models \varphi$
- $\pi, i \models X\varphi$ iff $\pi, i+1 \models \varphi$.
- $\pi, i \models \xi U \psi$ iff for some $j \geq i$, $\pi, j \models \psi$ and for all k, $i \leq k < j$ $\pi, k \models \xi$.

We will say that π *satisfies* a formula φ, denoted $\pi \models \varphi$, iff $\pi, 0 \models \varphi$.

Computations can also be viewed as infinite words over the alphabet 2^{Prop}. We shall see that the set of computations satisfying a given formula are exactly those accepted by some finite automaton on infinite words.

3.2 Model Checking

Propositional Temporal Logic and Büchi Automata: The following theorem establishes the correspondence between PTL and alternating Büchi automata.

Theorem 14. [MSS88] *Given a PTL formula φ, one can build an alternating Büchi automaton $A_\varphi = (\Sigma, S, \rho, S_0, F)$, where $\Sigma = 2^{Prop}$ and $|S|$ is in $O(|\varphi|)$, such that $L(A_\varphi)$ is exactly the set of computations satisfying the formula φ.*

Proof: The set S of state consists of all subformulas of φ and their negation, with the addition of two new states $\mathbf{T}$ and $\mathbf{F}$. The start state s_0 is φ itself. The set F of accepting states consists of all formulas in S of the form $\neg \xi U \psi$. It remains to define the transition function ρ. In the following definition, the dual $\overline{\theta}$ of a formula obtained from θ by switching $\vee$ and $\wedge$, by switching $\mathbf{T}$ and $\mathbf{F}$, and by negating subformulas of φ; e.g., $\overline{\mathbf{F} \vee (\mathbf{p} \wedge \mathbf{q})}$ is $\mathbf{T} \wedge (\neg \mathbf{p} \vee \neg \mathbf{q})$.

- $\rho(\mathbf{T}, \mathbf{a}) = \mathbf{true}$ for each $a \in \Sigma$,
- $\rho(\mathbf{F}, \mathbf{a}) = \mathbf{false}$ for each $a \in \Sigma$,
- $\rho(p, a) = \mathbf{T}$ if $p \in a$,
- $\rho(p, a) = \mathbf{F}$ if $p \notin a$,
- $\rho(\xi \wedge \psi, a) = \rho(\xi, a) \wedge \rho(\psi, a)$,
- $\rho(\neg \psi, a) = \overline{\rho(\psi, a)}$,
- $\rho(X\psi, a) = \psi$,
- $\rho(\xi U \psi, a) = \rho(\psi, a) \vee (\rho(\xi, a) \wedge \xi U \psi)$.

Note that $\rho(\psi, a)$ is defined by induction on the structure of ψ. ∎

By applying Theorem 13, we get:

Corollary 15. [VW94] *Given a PTL formula φ, one can build a Büchi automaton $A_\varphi = (\Sigma, S, \rho, S_0, F)$, where $\Sigma = 2^{Prop}$ and $|S|$ is in $2^{O(|\varphi|)}$, such that $L(A_\varphi)$ is exactly the set of computations satisfying the formula φ.*

We remark that the proof of Corollary 15 in [VW94] is different than the proof here; rather than go via alternating automata, the proof there goes through "subword automata" and "set-subword automata". See the discussion in [VW94] for a comparison between the two approaches.

Model Checking We are given a finite-state program and a PTL formula that specifies the legal computations of the program. The problem is to check whether all computations of the program are legal. Before going further, let us define these notions more precisely.

A *finite-state program* is a structure of the form $P = (W, s_0, R, V)$, where W is a finite set of states, $s_0 \in W$ is the initial state, $R \subseteq W^2$ is a total accessibility relation, and $V : W \to 2^{Prop}$ assigns truth values to propositions in $Prop$ for each state in W. Let $\mathbf{u}$ be an infinite sequence $u_0, u_1 \ldots$ of states in W such that $u_0 = s_0$, and $u_i R u_{i+1}$ for all $i \geq 0$. Then the sequence $V(u_0), V(u_1) \ldots$ is a *computation* of P. We will say that P *satisfies* an PTL formula φ if all computations of P satisfy φ. The *verification problem* is to check whether P satisfies φ.

The complexity of the verification problem can be measured in three different ways. First, one can fix the specification φ and measure the complexity with respect to the size of the program. We call this measure the *program-complexity* measure. More precisely, the program complexity of the verification problem is the complexity of the sets $\{P \mid P \textit{ satisfies } \varphi\}$ for a fixed φ. Secondly, one can fix the program P and measure the complexity with respect to the size of the

specification. We call this measure the *specification-complexity* measure. More precisely, the specification complexity of the verification problem is the complexity of the sets $\{\varphi \mid P \; satisfies \; \varphi\}$ for a fixed P. Finally, the complexity in the combined size of the program and the specification is the *combined complexity.* (These notions, implicitly suggested in [LP85], are the analogues of the notions of *data complexity*, *expression complexity*, and *combined complexity* defined in [Va82].)

Let C be a complexity class. We say that the program complexity of the verification problem is in C if $\{P \mid P \; satisfies \; \varphi\} \in C$ for any formula φ. We say that the program complexity of the verification problem is hard for C if $\{P \mid P \; satisfies \; \varphi\}$ is hard for C for some formula φ. We say that the program complexity of the verification problem is logspace complete for C if it is in C and is logspace hard for C. Similarly, we say that the specification complexity of the verification problem is in C if $\{\varphi \mid P \; satisfies \; \varphi\} \in C$ for any program P, we say that the specification complexity of the verification problem is hard for C if $\{\varphi \mid P \; satisfies \; \varphi\}$ is hard for C for some program P, and we say that the specification complexity of the verification problem is logspace complete for C if it is in C and is logspace hard for C.

We now describe our automata-theoretic approach to the verification problem. A finite-state program $P = (W, s_0, R, V)$ can be viewed as a Büchi automaton $A_P = (\Sigma, W, s_0, \rho, W)$, where $\Sigma = 2^{Prop}$ and $s' \in \rho(s, a)$ iff $(s, s') \in R$ and $a = V(s)$. As this automaton has a set of accepting states equal to the whole set of states, any infinite run of the automaton is accepting. Thus, $L(A_P)$ is the set of computations of P.

Hence, for a finite-state program P and a PTL formula φ, the verification problem is to verify that all sequences accepted by the automaton A_P satisfy the formula φ. By Corollary 15, we know that we can build a Büchi automaton A_φ that accepts exactly the sequences satisfying the formula φ. The verification problem thus reduces to the automata-theoretic problem of checking that all sequences accepted by the automaton A_P are also accepted by the automaton A_φ. Equivalently, we need to check that the automaton that accepts $L(A_P) \cap \overline{L(A_\varphi)}$ is empty, where $\overline{L(A_\varphi)} = \Sigma^\omega - L(A_\varphi)$.

First, note that one can build an automaton that accepts the language $\overline{L(A_\varphi)}$ by building the automaton $A_{\neg\varphi}$. By Corollary 15, the number of states in this automaton is in $2^{O(|\varphi|)}$. (A straightforward approach, starting with the automaton A_φ and then using Proposition 11 to complement it, would result in a doubly exponential blow-up.) To take the intersection of the two automata, we use Theorem 11. Consequently, we can build an automaton for $L(A_P) \cap \overline{L(A_\varphi)}$ having $|W| \cdot 2^{O(|\varphi|)}$ states. We need to check this automaton for emptiness. Using Theorem 12, we get the following results.

Theorem 16.

1. *The program complexity of the verification problem is logspace complete for NLOGSPACE.*

2. *The specification complexity of the verification problem is logspace complete for PSPACE.*
3. *Checking whether a formula φ is satisfied by a finite-state program P can be done in time $O(||P|| \cdot 2^{O(|\varphi|)})$ or in space $O((log||P|| + |\varphi|)^2)$.*

We note that a time upper bound that is polynomial in the size of the program and exponential in the size of the specification is considered here to be reasonable, since the specification is usually rather short [LP85].

Theorems 16 refines the results in [LP85, SC85]. We believe, however, that the automata-theoretic approach yields an algorithm that is much simpler and clearer (we urge the reader to compare). We also believe that our space bound for the program complexity is quite significant as the programs to which model checking is applied are often very large. For instance, if the finite-state program is given as a product of small components $(P_1, \ldots, P_k)$, then model checking can be done without building the product program, using space $O((log||P_1|| + \ldots + log||P_k||)^2)$, which is usually much less than the space needed to store the product program. For a practical algorithm that is based on these ideas, see [CVWY92].

4 Concluding Remarks

The examples above demonstrate the power and versatility of the theory of automata on infinite words and trees. Nevertheless, as I mentioned in the introduction, my experience has been that many researchers are not very comfortable working with this theory. I believe that this is caused by two reasons. First, the theory is not well-known and it does lack the physical intuition that one can bring to the theory of automata on finite words (it is hard to conceive of a physical device reading an input of infinite length). Second, I believe that this is a result of the way that the theory of finite automata is typically taught.

Finite-automata theory is typically taught as a mathematical theory of computation with some applications to compiler construction. That is, finite automata are thought as a very simple and robust model of computation with applications such as lexical analysis. One rarely, however, encounter applications in a finite-automata course; concrete applications are usually left to compiler-construction courses. Thus, most students are left with the impression of finite-automata theory as a fairly abstract mathematical theory. I believe that the theory ought be to be taught as a useful set of abstractions and tools for the working computer scientist. The educational goal should be more than just to train the students in rigorous and formal thinking, it should also be to provide the students with the knowledge of basic tools. To that end, much more emphasis should be given to applications of automata theory. It will be useful to identify applications that can be incorporated in undergraduate and graduate automata-theory courses.

References

[AV89] Abiteboul, S., Vianu, V.: Fixpoint extensions of first-order logic and Datalog-like languages, *Proc. 4th IEEE Symp. on Logic in Computer Science*, 1989, pp. 71–79.

[AU79] Aho, A.V., Ullman, J.D.: Universality of data retrieval languages, *Proc. 6th ACM Symp. on Principles of Programming Languages*, 1979, pp. 110–117.

[AK86] Apt, K.R., Kozen, D.C.: Limits for automatic verification of finite-state concurrent systems, *Information Processing Letters* 22(1986), pp. 307–309.

[BR86] Bancilhon, F., Ramakrishnan, R.: An amateur's introduction to recursive query processing strategies, *Proc. ACM Conf. on Management of Data*, Washington, 1986, pp. 16–52.

[BK84] Barringer, H., Kuiper, R.: A temporal logic specification method supporting hierarchical development, *Proc. NSF/SERC Seminar on Concurrency*, Pittsburgh, 1984.

[BL80] Brzozowski, J.A., Leiss, E.: On equations for regular languages, finite automata, and sequential networks, *Theoretical Computer Science* 10(1980), pp. 19–35.

[Bu62] Büchi, J.R.: On a decision method in restricted second order arithmetic, *Proc. Internat. Congr. Logic, Method. and Philos. Sci. 1960*, Stanford University Press, 1962, pp. 1–12.

[Ch81] Chandra, A.K.,: Programming primitives for database languages, *Proc. 8th ACM Symp. on Principles of Programming Languages*, Williamsburg, 1981, pp. 50–62.

[CH80] Chandra, A.K., Harel, D.: Computable queries for relational databases, *J. Computer and Systems Sciences* 21(1980), pp. 156–178.

[CH82] Chandra, A.K., Harel, D.: Structure and complexity of relational queries, *J. Computer and Systems Sciences* 25(1982), pp. 99–128.

[CH85] Chandra, A.K., Harel, D.: Horn-clause queries and generalizations, *J. Logic Programming*, 2(1985), pp. 1–15.

[CKS81] Chandra, A., Kozen, A., Stockmeyer, L.: Alternation, *J. ACM* 28(1981), pp. 114–133.

[CLM81] Chandra, A. K., Lewis, H.R., Makowsky, J.A.: Embedded implicational dependencies and their inference problem, *Proc. 13th ACM Symp. on Theory of Computing*, 1981, pp. 342–354.

[CV92] Chaudhuri, S., Vardi, M.Y.: On the equivalence of Datalog programs, *Proc. 11th ACM Symp. on Principles of Database Systems*, 1992, pp. 55–66 (see also IBM Research Report RJ9596, Nov. 1993).

[Ch74] Choueka, Y.: Theories of Automata on ω-tapes: a simplified approach, *J. Computer and System Sciences* 8(1974), pp. 117–141.

[CES86] Clarke, E.M., Emerson, E.A., Sistla, A.P.: Automatic verification of finite-state concurrent systems using temporal logics specifications: a practical approach, *ACM Trans. on Programming Languages and Systems*, 8(1986), pp. 244–263.

[CG87] Clarke, E.M., Grumberg, E.M.: Research on automatic verification of finite-state concurrent systems, in *Annual Review of Computer Science* 2(1987), pp. 269–290.

[CK86] Cosmadakis, S.S., Kanellakis, P.: Parallel evaluation of recursive rule queries, *Proc. 5th ACM Symp. on Principles of Database Systems*, Cambridge, 1986, pp. 280–293.

[Cos72] Costich, O.L.: A Medvedev characterization of sets recognized by generalized finite automata, *Math. System Theory* 6(1972), pp. 263–267.

[CVWY92] Courcoubetis, C., Vardi, M.Y., Wolper, P., Yannakakis, M.: Memory Efficient Algorithms for the Verification of Temporal Properties, *Formal Methods in System Design* 1(1992), pp. 275–288

[Do70] Doner, J.E.: Tree acceptors and some of their applications, *J. Computer and System Sciences* 4(1971), pp. 406–451.

[Ei74] Eilenberg, S.: *Automata,languages, and machines*, Academic Press, 1974.

[EH86] Emerson, E.A., Halpern, J.Y.: "Sometimes" and "Not Never" revisited: on branching vs. linear time temporal logic, *J. ACM* 33(1986), pp. 166-178.

[EL85a] Emerson, E.A., Lei, C.L.:, Temporal model checking under generalized fairness constraints, *Proc. 18th Hawaii Int'l Conference on System Sciences,* 1985.

[EL85b] Emerson, E.A., Lei, C.L.:, Modalities for model checking: branching time strikes back", *Proc. 12th ACM Symp. on Principles of Programming Languages,* New Orleans, 1985, pp. 84–96.

[GPSS80] Gabbay, D., Pnueli, A., Shelah, S., Stavi, J.: The temporal analysis of fairness, *Proc. 7th ACM Symp. on Principles of Programming Languages,* Las Vegas, 1980, pp. 163–173.

[GM78] Gallaire, H., Minker, J.: *Logic and databases*, Plenum Press, 1978.

[GS84] Gecseg, F., Steinby, M.: *Tree automata*, Akademiai Kiado, Budapest, 1984.

[HO83] Hailpern, B.T., Owicki, S.S.: Modular verification of computer communication protocols, *IEEE Trans. on Comm.* COM-31(1983), 1983, pp. 56–68.

[Ha85] Hayashi, T.: Finite automata on infinite objects, *Math. Res. Kyushu University* 15(1985), pp. 13–66.

[Im86] Immerman, N.: Relational queries computable in polynomial time, *Information and Control* 68(1986), pp. 86–104.

[Im88] Immerman, N.: Nondeterministic space is closed under complementation, *SIAM J. on Computing* 17(1988), pp.935–938.

[Ka68] Kamp, J.A.W.: *Tense logic and the theory of linear order*, Ph.D. Thesis, University of California, Los Angeles, 1968.

[K90] Kanellakis P.C.: Elements of relational database theory, in *Handbook of Theoretical Computer Science*, Vol. B. (J. v. Leeuwen et al., eds.), Elsevier, 1990.

[KVR83] Koymans, R., Vytopil, J., de Roever, W.P.: Real-time programming and asynchronous message passing, *Proc. 2rd ACM Symp. on Principles of Distributed Computing,* Montreal, 1983, pp. 187–197.

[La80] Lamport, L.: "Sometimes" is sometimes "Not Never", *Proc. 7th ACM Symp. on Principles of Programming Languages*, 1980, pp. 174–185.

[Le81] Leiss, E.: Succinct representation of regular languages by boolean automata, *Theoretical Computer Science* 13(1981), pp. 323–330.

[LP85] Lichtenstein, O., Pnueli, A.: Checking that finite-state concurrent programs satisfy their linear specifications, *Proc. 12th ACM Symp. on Principles of Programming Languages,* New Orleans, 1985, pp. 97–107.

[LPZ85] Lichtenstein, O., Pnueli, A., Zuck, L.: The glory of the past, *Proc. Workshop on Logics of Programs*, Brooklyn, 1985, Springer-Verlag, Lecture Notes in Computer Science 193, pp. 196–218.

[MP81] Manna, Z., Pnueli, A.: Verification of concurrent programs: the temporal framework, in *The Correctness Problem in Computer Science* (R.S. Boyer and J.S. Moore, eds.), Academic Press, 1981, pp. 215–273.

[MP92] Manna, Z., Pnueli, A.: *The temporal logic of reactive and concurrent systems*, Springer-Verlag, 1992.

[MUV84] Maier, D., Ullman, J.D., Vardi, M.Y.: On the foundations of the universal relation model, *ACM Trans. on Database Systems* 9(1984), pp. 283–308.

[Mey93] van der Meyden, R.: Recursively definite databases, *Theoretical Computer Science*, 116(1993), pp. 151–194.

[MH84] Miyano, S., Hayashi, T.: Alternating automata on ω-words, *Theoretical Computer Science* 32(1984), pp. 321–330.

[Mo74] Moschovakis, Y.N.: *Elementary induction on abstract structures*, North Holland, 1974.

[MSS88] Muller, D.E., Saoudi, A., Schupp, P.E.: Weak alternating automata give a simple explanation of why most temporal and dynamic logic are decidable in exponential time, *Proc. 3rd IEEE Symp. on Logic in Computer Science*, 1988, pp. 422–427.

[MP91] Mumick, I.S.,Pirahesh, H.: Overbound and right-linear queries, *Proc. 10th ACM Symp. on Principles of Database Systems*, 1991, pp. 127–141.

[Na89a] Naughton, J.F.: Data independent recursion in deductive databases, *J. Computer and System Sciences*, 38(1989), pp. 259–289.

[Na89b] Naughton, J.F.: Minimizing function-free recursive definitions, *J. ACM* 36(1989), pp. 69–91.

[NRSU89] Naughton, J.F., Ramakrishnan, R., Sagiv, Y., Ullman, J.D.: Efficient evaluation of right , left , and multilinear rules, *Proc. ACM-SIGMOD Int'l Conf. on Management of Data*, 1989, pp. 235–242.

[OL82] Owicki, S., Lamport, L.: Proving liveness properties of concurrent programs, *ACM Trans. on Programming Languages and Systems,* 4(1982), pp. 455–495.

[Pe85] Peikert, R.: *ω-regular languages and propositional temporal logic*, Report No. 85-01, ETH, Zurich, 1985.

[Pn77] Pnueli, A.: The temporal logic of programs, *Proc. 8th IEEE Symp. on Foundations of Computer Science,* Providence, 1977, pp. 46–57.

[QS82] Queille, J.P., Sifakis, J.: *Fairness and related properties in transition systems,* Research Report no. 292, IMAG, Grenoble, 1982.

[Ra69] Rabin, M.O.: Decidability of second-order theories and automata on infinite trees, *Trans. AMS* 141(1969), pp. 1–35.

[RSUV93] Ramakrishnan, R., Sagiv, Y., Ullman, J.D., Vardi, M.Y.: Logical query optimization by proof-tree transformation, *J. Computer and System Sciences* 47(1993), pp. 222–248

[Sa88] Safra, S.: On the complexity of ω-automata, *Proc. 29th IEEE Symp. on Foundation of Computer Science*, 1988, pp. 319–327.

[Sa88b] Sagiv, Y.: Optimizing Datalog programs, In *Foundations of Deductive Databases and Logic Programming*, J. Minker (ed.), Morgan Kaufmann Publishers, 1988, pp. 659–698.

[Se90] Seidl, H.: Deciding equivalence of finite tree automata, *SIAM J. Computing* 19(1990), pp. 424–437.

[Shm87] Shmueli, O.: Decidability and expressiveness aspects of logic queries, *Proc. 6th ACM Symp. on Principles of Database Systems*, 1987, pp. 237–249.

[SC85] Sistla, A.P., Clarke E.M.: The complexity of propositional linear temporal logic, *J. ACM* 32(1985), pp. 733–749.

[Si83] Sistla, A.P.: *Theoretical issues in the design and analysis of distributed systems*, Ph.D. Thesis, Harvard University, 1983.

[SVW87] Sistla, A.P., Vardi, M.Y., Wolper, P.: The complementation problem for Büchi automata with applications to temporal logic, *Theoretical Computer Science* 49(1987), pp. 217–237.

[TW68] Thatcher, J.W., Wright, J.B.: Generalized finite automata theory with an application to a decision problem of second-order logic, *Mathematical System Theory* 2(1968), pp. 57–81.

[Tho90] Thomas, W.: Automata on infinite objects, in *Handbook of Theoretical Computer Science*, Vol. B. (J. v. Leeuwen et al., eds.), Elsevier, 1990, pp. 135–191.

[TB73] Trakhtenbrot, B.A., Barzdin, Y.M.: *Finite automata: behavior and synthesis*, North-Holland, 1973.

[Ul88] Ullman, J.D.: *Principles of database and knowledge base systems*, Vol. 1, Computer Science Press, 1988.

[Ul89] Ullman, J.D.: *Principles of database and knowledge base systems*, Vol. 2, Computer Science Press, 1989.

[Va82] Vardi, M.Y.: The complexity of relational query languages, *Proc. 14th ACM Symp. on Theory of Computing*, San Francisco, 1982, pp. 137–146.

[Va92] Vardi, M.Y.: Automata theory for database theoreticians, in *Theoretical Studies in Computer Science* (J.D. Ullman, ed.), Academic Press, 1992, pp. 153–180.

[VW86b] Vardi, M.Y., Wolper, P.: An automata-theoretic approach to automatic program verification, *Proc. 1st IEEE Symp. on Logic in Computer Science*, Boston, 1986, pp. 332–334.

[VW94] Vardi, M.Y., Wolper, P.: Reasoning about infinite computation, to appear in *Information and Computation*, 1994.

[Wo83] Wolper, P.: Temporal logic can be more expressive, *Information and Control* 56(1983), pp. 72–99.

[Zl76] Zloof, M.; *Query-by-Example: operations on the transitive closure*, IBM Research Report RC5526, 1976.