Safraless Compositional Synthesis[⋆]

Orna Kupferman[1⋆⋆], Nir Piterman[2], and Moshe Y. Vardi[3⋆⋆⋆]

[1] Hebrew University
[2] Ecole Polytechnique Fédéral de Lausanne (EPFL)
[3] Rice University and Microsoft Research

Abstract. In automated synthesis, we transform a specification into a system that is guaranteed to satisfy the specification. In spite of the rich theory developed for system synthesis, little of this theory has been reduced to practice. This is in contrast with model-checking theory, which has led to industrial development and use of formal verification tools. We see two main reasons for the lack of practical impact of synthesis. The first is algorithmic: synthesis involves determinization of automata on infinite words, and a solution of parity games with highly complex state spaces; both problems have been notoriously resistant to efficient implementation. The second is methodological: current theory of synthesis assumes a single comprehensive specification. In practice, however, the specification is composed of a set of properties, which is typically evolving – properties may be added, deleted, or modified.

In this work we address both issues. We extend the Safraless synthesis algorithm of Kupferman and Vardi so that it handles LTL formulas by translating them to nondeterministic generalized Büchi automata. This leads to an exponential improvement in the complexity of the algorithm. Technically, our algorithm reduces the synthesis problem to the emptiness problem of a nondeterministic Büchi tree automaton $\mathcal{A}$. The generation of $\mathcal{A}$ avoids determinization, avoids the parity acceptance condition, and is based on an analysis of runs of universal generalized co-Büchi tree automata. The clean and simple structure of $\mathcal{A}$ enables optimizations and a symbolic implementation. In addition, it makes it possible to use information gathered during the synthesis process of properties in the process of synthesizing their conjunction.

1 Introduction

One of the most significant developments in the area of program verification over the last two decades has been the development of algorithmic methods for verifying temporal specifications of *finite-state* programs; see [5]. A frequent criticism against this approach, however, is that verification is done *after* significant resources have already

[⋆] A full version with full proofs can be downloaded from www.cs.huji.ac.il/∼ornak/cav06.pdf.

[⋆⋆] Supported in part by BSF grant 9800096, and by a grant from Minerva.

[⋆⋆⋆] Supported in part by NSF grants CCR-9988322, CCR-0124077, CCR-0311326, and ANI-0216467, by BSF grant 9800096, and by Texas ATP grant 003604-0058-2003. Part of this work was done while the author was visiting the Isaac Newton Institute for Mathematical Science, as part of a Special Programme on Logic and Algorithm.

been invested in the development of the program. Since programs invariably contain errors, verification simply becomes part of the debugging process. The critics argue that the desired goal is to use the specification in the program development process in order to guarantee the design of correct programs. This is called *program synthesis*.

In the late 1980s, several researchers realized that the classical approach to program synthesis, where a program is extracted from a proof that the specification is satisfiable, is well suited to *closed* systems, but not to *open* (also called *reactive*) systems [1, 6, 23]. In reactive systems, the program interacts with the environment, and a correct program should then satisfy the specification with respect to all environments. These researchers argued that the right way to approach synthesis of reactive systems is to consider the situation as a (possibly infinite) game between the environment and the program. A correct program can be then viewed as a winning strategy in this game. It turns out that satisfiability of the specification is not sufficient to guarantee the existence of such a strategy. Abadi et al. called specifications for which a winning strategy exists *realizable*. Thus, a strategy for a program with inputs in I and outputs in O maps finite sequences of inputs (words in $(2^I)^*$ – the actions of the environment so far) to an output in 2^O – a suggested action for the program. A strategy can then be viewed as a labeling of a tree with directions in 2^I by labels in 2^O. The traditional algorithm for finding a winning strategy transforms the specification into a parity automaton over such trees such that a program is realizable precisely when this tree automaton is nonempty, i.e., it accepts some infinite tree [23]. A finite generator of an infinite tree accepted by this automaton can be viewed as a finite-state program realizing the specification. This is closely related to the approach taken, e.g., in [25], to solve Church's *solvability problem* [4]. Several works during the 1990s showed how this approach to program synthesis can be carried out in a variety of settings.

In spite of the rich theory developed for program synthesis, little of this theory has been reduced to practice. In fact, the main approaches to tackle synthesis are either to use heuristic approaches (e.g., [12]) or to restrict the kind of allowed specification (e.g., [22]). Some people argue that this is because the realizability problem for linear-temporal logic (LTL) specifications is 2EXPTIME-complete [23, 26], but this argument is not compelling. First, experience with verification shows that even nonelementary algorithms can be practical, since the worst-case complexity does not arise often (cf., the model-checking tool MONA [7]). Furthermore, in some sense, synthesis is not harder than verification. This may seem to contradict the known fact that while verification is "easy" (linear in the size of the model and at most exponential in the size of the specification [16]), synthesis is hard (2EXPTIME-complete). There is, however, something misleading in this fact: while the complexity of synthesis is given with respect to the specification only, the complexity of verification is given with respect to the specification and the program, which can be much larger than the specification. In particular, it is shown in [26] that there are temporal specifications for which every realizing program must be at least doubly exponentially larger than the specifications. Clearly, the verification of such programs is doubly exponential in the specification, just as the cost of synthesis.

We believe that there are two reasons for the lack of practical impact of synthesis theory. The first is algorithmic and the second is methodological. Consider first the algorithmic problem. First, constructing tree automata for realizing strategies uses determinization of Büchi automata. Safra's determinization construction has been notoriously resistant to efficient implementations [2, 29] (An alternative construction is equally hard [2]. Piterman's improvement of Safra includes the tree structures that proved hard to implement [21].) Second, determinization results in automata with a very complicated state space. The best-known algorithms for parity-tree-automata emptiness [13] are nontrivial already when applied to simple state spaces. Implementing them on top of the messy state space that results from determinization is awfully complex, and is not amenable to optimizations and a symbolic implementation.

Another major issue is methodological. The current theory of program synthesis assumes that one gets a comprehensive set of temporal assertions as a starting point. This cannot be realistic in practice. A more realistic approach would be to assume an *evolving* formal specification: temporal assertions can be added, deleted, or modified. Since it is rare to have a complete set of assertions at the very start of the design process, there is a need to develop *compositional* synthesis algorithms. Such algorithms can, for example, refine designs when provided with additional temporal properties.

In this paper we address both issues. We focus on the case where forbidden behaviors are described by nondeterministic generalized Büchi automata on infinite words, which are Büchi automata with multiple acceptance sets (corresponding to the *impartiality* fairness condition of [17]). Our interest in specifying forbidden behaviors and in using the generalized Büchi condition is motivated by the fact that LTL formulas (and their negation) can be conveniently translated to nondeterministic generalized Büchi automata [9]. Equivalently, one can specify allowed behavior by universal generalized co-Büchi automata. Following [15], we offer an alternative to the standard automata-theoretic approach. The crux of our approach is avoiding the use of determinization constructions and of nondeterministic parity tree automata. In the approach described here, one checks whether the specification ψ is realizable using the following steps: (1) construct a universal generalized co-Büchi tree automaton $\mathcal{A}_\psi$ that accepts all realizing strategies for ψ, (2) reduce[4] $\mathcal{A}_\psi$ to an alternating weak tree automaton $\mathcal{A}^w_\psi$, (3) translate $\mathcal{A}^w_\psi$ to a nondeterministic Büchi tree automaton $\mathcal{A}^n_\psi$, and (4) check that the language of $\mathcal{A}^n_\psi$ is nonempty. The key is avoiding determinization, by using universal generalized co-Büchi automata instead of deterministic parity automata.[5]

The difference between our approach here and the approach in [15] is that here we use *generalized* co-Büchi automata, unlike the co-Büchi automata used there. This leads to an exponential improvement in the complexity of our algorithm, as we describe below. Extending the framework of [15] to generalized co-Büchi automata requires two key technical steps. First, as our Safraless approach used a "Safraful" bound on the size

[4] We use "reduce A_1 to A_2", rather than "translate A_1 to A_2" to indicate that A_2 accepts a subset of the language of A_1, yet the language of $\mathcal{A}_1$ is empty iff the language of $\mathcal{A}_2$ is empty.

[5] A note to readers who are discouraged by the fact our method goes via several intermediate automata: it is possible to combine the reductions into one construction, and in fact we describe here also a direct translation of universal generalized co-Büchi automata into nondeterministic Büchi automata. In practice, however, it is beneficial to have many intermediate automata, as each intermediate automaton undergoes optimization constructions that are suitable for its particular type, cf. [11].

of the realizing strategies, we need to extend Safra's construction to nondeterministic generalized Büchi automata, obtaining an exponential improvement (with respect to an approach that first translates the generalized Büchi automaton to a Büchi automaton) in that construction. Second, we need to show how the co-Büchi ranks devised in [14] for the analysis of runs of universal automata on words can be applied to the analysis of runs of universal automata on finitely generated trees.

Beyond the improvement in complexity, the advantage of the Safraless approach is that we get tree automata with cleanly described state spaces, which enables the application of symbolic algorithms for Büchi tree automata emptiness. Further, we can now obtain a *compositional* algorithm. Given a specification ψ, we first check its realizability. Suppose now that we get an additional specification ψ'. We can, of course, simply check the realizability of $\psi \wedge \psi'$ from scratch. Instead, we suggest to first check also the realizability of ψ'. We then show how, thanks to the simple structure of the tree automata, much of the work used in checking the realizability of ψ and ψ' in isolation can be reused in checking the realizability of $\psi \wedge \psi'$. The compositional algorithm we suggest can be combined with an *incremental* algorithm, in which we iteratively increase the bound on the size of the realizing strategy. As demonstrated in [11] for the linear setting, the bound that is needed in practice is usually much smaller than the worst-case bound. In addition, we explain how the incremental and compositional algorithm can be implemented symbolically.

2 Preliminaries

We assume familiarity with the basic notions of alternating automata on infinite trees, cf. [10].

Given an alphabet Σ and a set D of directions, a *Σ-labeled D-tree* is a pair $\langle T, \tau \rangle$, where $T \subseteq D^*$ is a tree over D and $\tau : T \to \Sigma$ maps each node of T to a letter in Σ. A *transducer* is a labeled finite graph with a designated start node, where the edges are labeled by D and the nodes are labeled by Σ. A Σ-labeled D-tree is *regular* if it is the unwinding of some transducer. More formally, a transducer is a tuple $\mathcal{T} = \langle D, \Sigma, S, s_{in}, \eta, L \rangle$, where D is a finite set of directions, Σ is a finite alphabet, S is a finite set of states, $s_{in} \in S$ is an initial state, $\eta : S \times D \to S$ is a deterministic transition function, and $L : S \to \Sigma$ is a labeling function. We define $\eta : D^* \to S$ in the standard way: $\eta(\varepsilon) = s_{in}$, and for $x \in D^*$ and $d \in D$, we have $\eta(x \cdot d) = \eta(\eta(x), d)$. Intuitively, A Σ-labeled D-tree $\langle D^*, \tau \rangle$ is regular if there exists a transducer $\mathcal{T} = \langle D, \Sigma, S, s_{in}, \eta, L \rangle$ such that for every $x \in D^*$, we have $\tau(x) = L(\eta(x))$. We then say that the size of the regular tree $\langle D^*, \tau \rangle$, denoted $\|\tau\|$, is $|S|$, the number of states of $\mathcal{T}$.

We denote an alternating tree automaton by a tuple $\mathcal{A} = \langle \Sigma, D, Q, q_{in}, \delta, \alpha \rangle$, where Σ is the input alphabet, D is a set of directions, Q is a finite set of states, $\delta : Q \times \Sigma \to \mathcal{B}^+(D \times Q)$ is a transition function, $q_{in} \in Q$ is an initial state, and α specifies the acceptance condition A run of $\mathcal{A}$ is accepting if all its infinite paths satisfy the acceptance condition. For a path π, we denote the set of automaton states visited infinitely often along this path by $inf(\pi)$. We consider here four acceptance conditions defined as follows

- A path π satisfies a *generalized Büchi* condition $\alpha = \{F_1, F_2, \dots, F_k\} \subseteq 2^Q$ iff for all $1 \leq i \leq k$ we have $inf(\pi) \cap F_i \neq \emptyset$. The number k of sets in α is called the *index* of the automaton. If $|\alpha| = 1$ we call α a *Büchi* condition.
- A path π satisfies a *generalized co-Büchi* condition $\alpha = \{F_1, F_2, \dots, F_k\} \subseteq 2^Q$ iff for some $1 \leq i \leq k$ we have $inf(\pi) \cap F_i = \emptyset$. The number k of sets in α is called the *index* of the automaton. If $|\alpha| = 1$ we call α a *co-Büchi* condition.
- A path π satisfies a *parity* condition $\alpha = \langle F_0, \dots, F_k \rangle$ where $F_0, \dots F_k$ form a partition of Q iff for some even i we have $inf(\pi) \cap F_i \neq \emptyset$ and forall $i' < i$ we have $inf(\pi) \cap F_{i'} = \emptyset$. We call k the number of *priorities* of α.

For the three conditions, an automaton accepts a tree iff there exists a run that accepts it. We denote by $\mathcal{L}(\mathcal{A})$ the set of all Σ-labeled trees that $\mathcal{A}$ accepts. We also refer to a fourth condition, which is a special case of the Büchi condition, and is referred to as the *weak* condition [20].

Below we discuss some special cases of alternating automata. The alternating automaton $\mathcal{A}$ is *nondeterministic* if for all the formulas that appear in δ, if (d_1, q_1) and (d_2, q_2) are conjunctively related, then $d_1 \neq d_2$. (i.e., if the transition is rewritten in disjunctive normal form, there is at most one element of $\{d\} \times Q$, for each $d \in D$, in each disjunct). The automaton $\mathcal{A}$ is *universal* if all the formulas that appear in δ are conjunctions of atoms in $D \times Q$, and $\mathcal{A}$ is *deterministic* if it is both nondeterministic and universal. The automaton $\mathcal{A}$ is a *word* automaton if $|D| = 1$. Then, we can omit D from the specification of the automaton and denote the transition function of $\mathcal{A}$ as $\delta : Q \times \Sigma \rightarrow \mathcal{B}^+(Q)$. If the word automaton is nondeterministic or universal, then $\delta : Q \times \Sigma \rightarrow 2^Q$.

We denote each of the different types of automata by an acronym in $\{D, N, U, A\} \times \{B, GB, C, GC, P\} \times \{W, T\}$, where the first letter describes the branching mode of the automaton (deterministic, nondeterministic, universal, or alternating), the second letter describes the acceptance condition (Büchi, generalized Büchi, co-Büchi, generalized co-Büchi, or parity), and the third letter describes the object over which the automaton runs (words or trees). For example, APT are alternating parity tree automata and UGCT are universal generalized co-Büchi tree automata.

3 Synthesis

Consider an UGCW $\mathcal{S}$ over the alphabet $2^{I \cup O}$, for sets I and O of input and output signals. The *realizability problem* for $\mathcal{S}$ [23] is to decide whether there is a *strategy* $f : (2^I)^* \rightarrow 2^O$, generated by a transducer[6] such that all the computations of the system generated by f are in $L(\mathcal{S})$. We call such a strategy, a *good* strategy. A computation $\rho \in (2^{I \cup O})^\omega$ is *generated by* f if $\rho = (i_0 \cup o_0), (i_1 \cup o_1), (i_2 \cup o_2), \dots$ and for all $j \geq 1$, we have $o_j = f(i_0 \cdot i_1 \cdots i_{j-1})$.

In practice, the UGCW $\mathcal{S}$ originates from an LTL formula ψ that specifies the desired properties of the program we synthesize. In order to get $\mathcal{S}$, we first translate $\neg\psi$ to an NGBW $\mathcal{A}_{\neg\psi}$, and then dualize $\mathcal{A}_{\neg\psi}$ by viewing it as a UGCW. By [31, 9], $\mathcal{A}_{\neg\psi}$, and

[6] As $\mathcal{S}$ recognizes an ω-regular language, if some transducer that generates f exists, then there is also a finite-state transducer.

thus also $\mathcal{S}$, have $2^{O(|\psi|)}$ states and index $O(|\psi|)$. Alternatively, one can define properties directly using UGCW, as done, for example, in the framework of Generalized Symbolic Trajectory Evaluation [32], by means of *fair assertion graphs*.

Theorem 1. *The realizability problem for a UGCW can be reduced to the nonemptiness problem of a UGCT with the same state space and index.*

Proof: A strategy $f : (2^I)^* \to 2^O$ can be viewed as a 2^O-labeled 2^I-tree. Given a UGCW S, we define a UGCT $\mathcal{S}'$ such that $\mathcal{S}'$ accepts a 2^O-labeled 2^I-tree $\langle T, \tau\rangle$ iff τ is a good strategy for $\mathcal{S}$.

Let $\mathcal{S} = \langle 2^{I\cup O}, Q, q_{in}, \delta, \alpha\rangle$. Then, $\mathcal{S}' = \langle 2^O, 2^I, Q, q_{in}, \delta', \alpha\rangle$, where for every $q \in Q$ and $o \in 2^O$, we have $\delta'(q, o) = \bigwedge_{i\in 2^I} \bigwedge_{q'\in\delta(q,i\cup o)}(i, q')$. Thus, from state q, reading the output assignment $o \in 2^O$, the automaton $\mathcal{S}'$ branches to each direction $i \in 2^I$, with all the states q' to which δ branches when it reads $i \cup o$ in state q. It is not hard to see that $\mathcal{S}'$ accepts a 2^O-labeled 2^I-tree $\langle T, \tau\rangle$ iff for all the paths $\{\varepsilon, i_0, i_0 \cdot i_1, i_0 \cdot i_1 \cdot i_2, \ldots\}$ of T, the infinite word $(i_0 \cup \tau(\varepsilon)), (i_1 \cup \tau(i_0)), (i_2 \cup \tau(i_0 \cdot i_1)), \ldots$ is accepted by the UGCW $\mathcal{S}$ as required. □

We now describe an emptiness preserving translation of UGCT to NBT. The correctness proof of the construction is given in Sections 4.1 and 4.2. There, we also suggest to use ABT as an intermediate step in the construction. While this adds a step to our chain of reductions, it enables further optimizations of the result.

For an integer c, let $[c]$ denote the set $\{0, 1, \ldots, c\}$, and let $[c]^{odd}$ and $[c]^{even}$ denote the set of odd and even members of $[c]$, respectively. Also, let $R_k(c) = [2c]^{even} \cup ([2c]^{odd} \times \{1, \ldots, k\})$, and $\leq$ be the lexicographical order on the elements of $R_k(c)$. We refer to the members of $R_k(c)$ in $[2c]^{even}$ as *even ranks* and refer to the members of $R_k(c)$ in $[2c]^{odd} \times \{j\}$ as *odd ranks with index* j. Note that the size of $R_k(c)$ is $c(k + 1) + 1$. Our construction refers to a function $Det(n, k)$, which, as we show later, is bounded from above by $n^{2n+2}k^n$.

Theorem 2. *Let $\mathcal{A}$ be a UGCT with n states and index k. There is an NBT $\mathcal{A}'$ over the same alphabet such that all the following hold.*

- *$\mathcal{L}(\mathcal{A}') \subseteq \mathcal{L}(\mathcal{A})$,*
- *$\mathcal{L}(\mathcal{A}) \neq \emptyset$ implies $\mathcal{L}(\mathcal{A}') \neq \emptyset$, and*
- *the number of states in $\mathcal{A}'$ is $2^{O(n^2(\log n+\log k))}$.*

Proof: Let $\mathcal{A} = \langle \Sigma, D, Q, q_{in}, \delta, \{F_1, \ldots, F_k\}\rangle$, and let $c = Det(n, k)$. Note that c is $2^{O(n(\log n+\log k))}$. Let $\mathcal{R}_k(c)$ be the set of functions $f : Q \to R_k(c)$ in which $f(q)$, for all $q \in F_j$, is not odd with index j. For $g \in \mathcal{R}_k(c)$, let $odd(g) = \{q : g(q) \text{ is odd}\}$. We define $\mathcal{A}' = \langle \Sigma, D, Q', q'_{in}, \delta', \alpha'\rangle$, where

- $Q' = 3^Q \times \mathcal{R}_k(c)$. For technical convenience, we refer to the states of Q' as triples $\langle S, O, f\rangle$ with $O \subseteq S \subseteq Q$ and $f \in \mathcal{R}_k(c)$.
- $q'_{in} = \langle\{q_{in}\}, \emptyset, g_0\rangle$, where g_0 maps all states to $2c$.
- For $q \in Q$, $\sigma \in \Sigma$, and $d \in D$, let $\delta(q, \sigma, d) = \{q' \mid (d, q') \in \delta(q, \sigma)\}$. For $S \subseteq Q$, $\sigma \in \Sigma$, and $d \in D$ we define $\delta(S, \sigma, d)$ in the natural way. For two functions g and g' in $\mathcal{R}_k(c)$, a letter σ, and direction $d \in D$, we say that g' *covers* $\langle g, \sigma, d\rangle$ if for all q and q' in Q, if $q' \in \delta(q, \sigma, d)$, then $g'(q') \leq g(q)$. Let $g' \preceq \langle g, \sigma, d\rangle$ denote that g' covers $\langle g, \sigma, d\rangle$. Then, for all $\langle S, O, g\rangle \in Q'$ and $\sigma \in \Sigma$, we define δ as follows.

- If $O \neq \emptyset$, then

$$\delta'(\langle S, O, g\rangle, \sigma) = \bigwedge_{d \in D} \bigvee_{g_d \preceq \langle g, \sigma, d\rangle} (d, \langle \delta(S, \sigma, d), \delta(O, \sigma, d) \setminus odd(g_d), g_d\rangle)$$

- If $O = \emptyset$, then

$$\delta'(\langle S, O, g\rangle, \sigma) = \bigwedge_{d \in D} \bigvee_{g_d \preceq \langle g, \sigma, d\rangle} (d, \langle \delta(S, \sigma, d), \delta(S, \sigma, d) \setminus odd(g_d), g_d\rangle)$$

– $\alpha' = 2^Q \times \{\emptyset\} \times \mathcal{R}_k(c)$.

In Section 4 we sketch the proof that this automaton indeed satisfies the conditions of the theorem. □

In fact, $\mathcal{A}'$ accepts every regular tree in the language of $\mathcal{A}$ that is produced by a "small" transducer. We show that whenever $\mathcal{A}$ accepts some regular tree, there exists some "small" regular tree that is accepted by $\mathcal{A}'$. Thus, if $\mathcal{A}$ accepts some regular tree, it accepts a regular tree produced by a small transducer, and this regular tree is also accepted by $\mathcal{A}'$.

Corollary 1. *The realizability problem for an NGBW with n states and index k can be reduced to the nonemptiness problem of an NBT with $2^{O(n^2(\log n + \log k))}$ states.*

These bounds are exponentially better than those established in [15]. There, the NGBW is converted to an NBW with nk states and the overall resulting complexity is $2^{O((nk)^2(\log k + \log n))}$.[7]

The *synthesis problem* for $\mathcal{S}$ is to find a transducer that generates a strategy realizing $\mathcal{S}$. Known algorithms for the nonemptiness problem can be easily extended to return a transducer [24]. The algorithm we present here also enjoys this property, thus it can be used to solve not only the realizability problem but also the synthesis problem. (For a comparison of the Safraless and the Safraful approaches to synthesis from the perspective of program size, see [15].)

4 From UGCT to NBT

Recall that runs of alternating tree automata are labeled trees. By merging nodes that are roots of identical subtrees, it is possible to maintain runs in graphs. In Section 4.2, we prove a bounded-size run graph property for UGCT. In Section 4.2, we show how the bounded-size property enables a simple translation of UGCT to ABT, which we then translate to an NBT. Combining the translations results in the UGCT to NBT construction described in Theorem 2. While our construction avoids using the determinization construction, the proof of the bounded-size run-graph property makes use of the bound the construction provides to the blow-up involved in determinization. Since we handle the generalized co-Büchi construction, we need a bound on the blow-up involved in the determinization of NGBW. We provide such a bound in Section 4.1.

[7] We can use the improved bound on determinization established in [21] to improve the bounds in [15]. This, however, reduces only the constants in the exponent.

4.1 NGBW to DPW

There are two known approaches to determinization of NGBW. The first is to convert the NGBW to an NBW [3] and then use determinization [27, 21]. The second is to view the NGBW as a Streett automaton and apply determinization of Streett automata [28, 21]. Both approaches produce automata with $(nk)^{O(nk)}$ states. In this section we show how to extend the determinization construction for the case of generalized Büchi automata. Our construction below produces a DPW with $(nk)^{O(n)}$ states, exponentially fewer states than the approaches described.

We offer here a succinct description of the improvement. The basis of our construction is Safra's determinization [27], as improved by Piterman [21]. The key is to augment compact Safra trees with an indexing function. In Piterman's construction, the DPW refers to a visit in the set of accepting states as a good event. In our extension, a good event occurs only after visits to all the sets in the generalized Büchi condition. Thus, the idea is similar to the indexing used in the translation of NGBW to NBW [9], but the challenge is to combine this indexing in the state space of the DPW in a way that minimizes the blow-up in terms of k. the improved construction is used only to generate the improved bound. The synthesis algorithm uses this bound but it does *not* use the determinization construction.

Theorem 3. *Given an NGBW with n states and index k, we can construct an equivalent DPW with at most $n^{2n+2}k^n$ states and $2n$ priorities.*

Proof: Let $\mathcal{N} = \langle \Sigma, S, \delta, s_0, \alpha \rangle$ be an NGBW with $|S| = n$ and $\alpha = \{F_1, \ldots, F_k\}$. Let $V = [n]$. We construct the DPW $\mathcal{D}$ equivalent to $\mathcal{N}$. Let $\mathcal{D} = \langle \Sigma, D, \rho, d_0, \alpha' \rangle$, where the components of $\mathcal{D}$ are as follows.

- A *generalized compact Safra tree* t is $\langle N, 1, p, l, h, r, g \rangle$ where $N \subseteq V$ is a set of nodes, $1 \in N$ is the root node, $p : N \to N$ is the parenthood function, $l : N \to 2^S$ is a labeling of the nodes with subsets of S, $h : N \to [k]$ is an indexing function associating with every node an index in $[k]$, and $r, g \in [n+1]$ are used to define the parity condition. In addition, the label of every node is a proper superset of the union of the labels of its children. The labels of two siblings are disjoint. The set of nodes is always consecutive and includes the first $|N|$ elements in V (i.e., $1, \ldots, |N|$). The set D of states is the set of *generalized compact Safra trees* over S and k.
- $d_0 \in D$ has a unique node 1 where $l(1){=}\{s_0\}$, $h(1){=}1$, $r{=}2$, and $g{=}1$.
- The parity acceptance condition is $\alpha'{=}\{F'_0, \ldots, F'_{2n-1}\}$ where
  - $F'_0 = \{d \in D \mid g = 1\}$
  - $F'_{2i+1} = \{d \in D \mid r = i + 2 \text{ and } g \geq r\}$
  - $F'_{2i+2} = \{d \in D \mid g = i + 2 \text{ and } r > g\}$
- For every tree $d \in D$ and letter $\sigma \in \Sigma$ the transition $d' = \rho(d, \sigma)$ is the result of the following transformations on d. (1) For every node v with label S' replace S' by $\delta(S', \sigma)$. (2) For every node v with label S' such that $h(v) = i$ and $S' \cap F_i \neq \emptyset$, create a son v' such that v' is the minimal value in V that is greater than all other nodes. Set its label to $S' \cap F_i$ and its index to 1. We may use temporarily nodes in the range $[(n{+}1)..(2n)]$. (3) For every node v with label S' and state $s \in S'$ such that s belongs also to some sibling v' of v such that $v' < v$, remove s from the label of

v and all its descendants. (4) For every node v whose label is equal to the union of the labels of its children, remove all descendants of v. If $h(v) = k$, change $h(v)$ to 1 and call v *green*. If $h(v) < k$, increase $h(v)$ by one. Set g to the minimum of $n+1$ and the green nodes. (5) Remove all nodes with empty labels. Set r the minimum of $n+1$ and all the nodes removed during all stages of the transformation. (6) Let Z denote the set of nodes removed during all previous stages of the transformation. For every node v let $rem(v)$ be $|\{v' \in Z \mid v' < v\}|$. For every node v such that $l(v) \neq \emptyset$ we replace v by $v - rem(v)$.

□

Let $Det(n, k)$ be the number of generalized compact Safra trees for NGBW with n states and index k. By Theorem 3, $Det(n, k)$ is bounded from above by $n^{2n+2}k^n$.

4.2 From UGCT to NBT

A bounded-size run graph property for UGCT Let $\mathcal{A} = \langle \Sigma, D, Q, q_{in}, \delta, \alpha \rangle$ be a UGCT with $\alpha = \{F_1, \ldots, F_k\}$. Recall that a run $\langle T_r, r \rangle$ of $\mathcal{A}$ on a Σ-labeled D-tree $\langle T, \tau \rangle$ is a $(T \times Q)$-labeled tree in which a node y with $r(y) = \langle x, q \rangle$ stands for a copy of $\mathcal{A}$ that visits the state q when it reads the node x. Assume that $\langle T, \tau \rangle$ is regular, and is generated by a transducer $\mathcal{T} = \langle D, \Sigma, S, s_{in}, \eta, L \rangle$. For two nodes y_1 and y_2 in T_r, with $r(y_1) = \langle x_1, q_1 \rangle$ and $r(y_2) = \langle x_2, q_2 \rangle$, we say that y_1 and y_2 are *similar* iff $q_1 = q_2$ and $\eta(x_1) = \eta(x_2)$. By merging similar nodes into a single vertex, we can represent the run $\langle T_r, r \rangle$ by a finite graph $G = \langle V, E \rangle$, where $V = S \times Q$ and $E(\langle s, q \rangle, \langle s', q' \rangle)$ iff there is $c \in D$ such that $(c, q') \in \delta(q, L(s))$ and $\eta(s, c) = s'$. We restrict G to vertices reachable from the vertex $\langle s_{in}, q_{in} \rangle$. We refer to G as the *run graph of $\mathcal{A}$ on $\mathcal{T}$*. A run graph of $\mathcal{A}$ is then a run graph of $\mathcal{A}$ on some transducer $\mathcal{T}$. We say that G is accepting iff every infinite path of G has only finitely many F_j-vertices (vertices in $S \times F_j$), for some $1 \leq j \leq k$. Since $\mathcal{A}$ is universal and $\mathcal{T}$ is deterministic, the run $\langle T_r, r \rangle$ is *memoryless* in the sense that the merging does not introduce to G paths that do not exist in $\langle T_r, r \rangle$, and thus, it preserves acceptance. Formally, we have the following:

Lemma 1. *Consider a UGCT $\mathcal{A}$. Let $\langle T, \tau \rangle$ be a tree generated by a transducer $\mathcal{T}$. The run tree $\langle T_r, r \rangle$ of $\mathcal{A}$ on $\langle T, \tau \rangle$ is accepting iff the run graph G of $\mathcal{A}$ on $\mathcal{T}$ is accepting.*

Note that G is finite, and its size is bounded by $S \times Q$. We now bound S and get a bounded-size run-graph property for UGCT. The bound on S depends on the blow-up involved in NGBW determinization, which we studied in Section 4.1. Essentially, the bound depends on the size of an NPT equivalent to the UGCT, and in order to get such an NPT we have to determinize an NGBW that accepts bad paths in runs of the UGCT.

Theorem 4. *A UGCT $\mathcal{A}$ with n states and index k is not empty iff $\mathcal{A}$ has an accepting run graph with at most $Det(n, k) \cdot n$ vertices.*

From UGCT to NBT via ABT Consider a graph $G' \subseteq G$. We say that a vertex $\langle s, q \rangle$ is *finite* in G' iff all the paths that start at $\langle s, q \rangle$ are finite. For $1 \leq j \leq k$, we say that a vertex $\langle s, q\rangle$ is F_j*-free* in G' iff all the vertices in G' that are reachable from $\langle s, q\rangle$ are not F_j-vertices. Note that, in particular, an F_j-free vertex is not an F_j-vertex.

Given a run $\langle T_r, r\rangle$, we define an infinite sequence of graphs $G_0 \supseteq G_1^1 \supseteq G_1^2 \supseteq \ldots G_1^k \supseteq G_1^{k+1} \supseteq G_3^1 \supseteq \ldots G_3^{k+1} \supseteq G_5^1 \ldots$ as follows. To simplify notations, we sometimes refer to G_{2i+1}^1 as G_{2i+1} and to G_{2i+1}^{k+1} as G_{2i+2}. Thus, $G_1 = G_1^1$, $G_2 = G_1^{k+1}$, $G_3 = G_3^1$, $G_4 = G_3^{k+1}$, and so on.

- $G_0 = G$.
- $G_{2i+1}^1 = G_{2i} \setminus \{\langle s, q\rangle \mid \langle s, q\rangle \text{ is finite in } G_{2i}\}$.
- $G_{2i+1}^{j+1} = G_{2i+1}^j \setminus \{\langle s, q\rangle \mid \langle s, q\rangle \text{ is } F_j\text{-free in } G_{2i+1}^j\}$, for $1 \leq j \leq k$.

Lemma 2. *A run graph $G = \langle V, E\rangle$ is accepting iff there is $i \leq |V|$ for which G_{2i} is empty.*

Let G be an accepting run graph. Given a vertex $\langle s, q\rangle$ in G, the *rank* of $\langle s, q\rangle$, denoted $rank(s, q)$, is defined as follows:

$$rank(s,q) = \begin{bmatrix} 2i & \text{If } \langle s,q\rangle \text{ is finite in } G_{2i}. \\ \langle 2i+1, j\rangle & \text{If } \langle s,q\rangle \text{ is } F_j\text{-free in } G_{2i+1}^j. \end{bmatrix}$$

Recall that, for an integer c, we have defined $R_k(c) = [2c]^{even} \cup ([2c]^{odd} \times \{1, \ldots, k\})$, as a set of $c(k + 1)$ ranks, and defined $\leq$ as the lexicographical order on the elements of $R_k(c)$. For an odd rank $\rho = \langle 2i+1, j\rangle$, we refer to G_{2i+1}^j as G_ρ. Let $c = |V|$. By Lemma 2, there is $i \leq c$ for which G_{2i} is empty. Therefore, every vertex gets a well-defined rank in $R_k(c)$.

Lemma 3. *In every infinite path in an accepting run graph G, there exists a vertex $\langle s, q\rangle$ with an odd rank such that all the vertices $\langle s', q'\rangle$ on the path that are reachable from $\langle s, q\rangle$ have $rank(s', q') \leq rank(s, q)$.*

We can now use the analysis of ranks in order to translate UGCT to NBT. In order to enable further optimizations, we use ABT as an intermediate step in the construction.

Theorem 5. *Let $\mathcal{A}$ be a UGCT with n states and index k. There is an ABT $\mathcal{A}'$ over the same alphabet such that all the following hold.*

- *$\mathcal{L}(\mathcal{A}') \subseteq \mathcal{L}(\mathcal{A})$,*
- *$\mathcal{L}(\mathcal{A}) \neq \emptyset$ implies $\mathcal{L}(\mathcal{A}') \neq \emptyset$, and*
- *the number of states in $\mathcal{A}'$ is $2^{O(n(\log n + \log k))}$.*

As detailed in the proof of the Theorem, the ABT $\mathcal{A}'$ accepts all the regular trees $\langle T, \tau\rangle \in \mathcal{L}(\mathcal{A})$ that are generated by a transducer $\mathcal{T} = \langle D, \Sigma, S, s_{in}, \eta, L\rangle$ with at most $Det(n, k)$ states. Note that the run graph of $\mathcal{A}$ on such $\langle T, \tau\rangle$ is accepting and is of size most $Det(n, k) \cdot n$. By Theorem 4, we have that $\mathcal{L}(\mathcal{A}') \neq \emptyset$ iff $\mathcal{L}(\mathcal{A}) \neq \emptyset$.

The state space of $\mathcal{A}'$ is $Q' = Q \times R_k(c)$. Intuitively, when $\mathcal{A}'$ is in state $\langle q, \rho\rangle$ as it reads the node $x \in T$, it guesses that the rank of the vertex $\langle \eta(x), q\rangle$ of G is ρ. The transitions of $\mathcal{A}'$ allows the guessed ranks to decrease, but makes sure that if a state is in F_j, the guessed rank for it cannot be odd with index j. By Lemma 3, the guessed ranks should eventually converge to some odd rank, which is checked by the acceptance condition of $\mathcal{A}'$.[8]

In [18], Miyano and Hayashi describe a translation of ABW to NBW. In Theorem 6 below (see also [19]), we present (a technical variant of) their translation, adapted to tree automata,

Theorem 6. *Let $\mathcal{A}$ be an ABT with n states. There is an NBT $\mathcal{A}'$ with $2^{O(n)}$ states, such that $\mathcal{L}(\mathcal{A}') = \mathcal{L}(\mathcal{A})$.*

Combining Theorems 5 and 6, one can reduce the nonemptiness problem for UGCT to the nonemptiness problem for NBT. Consider a UGCT $\mathcal{A}$ with n states and index k. If we translate $\mathcal{A}$ to an NBT by going through the ABT we have obtained in Theorem 5, we end up with an NBT with $2^{2^{O(n(\log n+\log k))}}$ states, as the ABT has $2^{O(n(\log n+\log k))}$ states. In order to complete the construction, and get the NBT described in the proof of Theorem 2, we exploit the special structure of the ABT and show that only $2^{O(n^2(\log n+\log k))}$ states of the NBT constructed in Theorem 6 may participate in an accepting run.

5 Compositional Synthesis

A serious drawback of current synthesis algorithms is that they assume a comprehensive set of temporal assertions as a starting point. In practice, however, specifications are evolving: temporal assertions are added, deleted, or modified during the design process. In this section we describe how our synthesis algorithm can support *compositional* synthesis, where the temporal assertions are given one by one. We show how the Safraless approach enables us, when we check the realizability of $\psi \wedge \psi'$, to use much of the work done in checking the realizability of ψ and ψ' in isolation. Devising compositional synthesis algorithms to other forms of composition, e.g., $\psi' \rightarrow \psi$, is an interesting research problem.

Our compositional algorithm extends the *incremental-synthesis* algorithm described in [15]. Essentially, we show that when we construct and check the emptiness of the NBT to which realizability of $\psi \wedge \psi'$ is reduced, we can use much of the work done in the process of checking the emptiness of the two (much smaller) NBTs to which realizability of ψ and ψ' is reduced (in isolation).

We first review the incremental-synthesis idea from [15]. Recall that our construction is based on the fact we can bound the maximal rank that a vertex in an accepting run graph G gets. Often, the sequence $G_0, G_1, G_2, \ldots$ of graphs described in Section 4.2 converges to the empty graph very quickly, making the bound on the maximal rank much smaller (see [11] for an analysis and experimental results for the case of UCW). Accordingly, one can regard the bound c as a parameter in the construction: start with a small parameter, and increase it if necessary.

[8] Readers familiar with weak automata [20], would note that our automaton is in fact an alternating weak tree automaton. It is the special structure of weak automata that enables some of the optimizations we describe below.

To see how this is done, consider the combined construction described in Theorem 2. Starting with a UGCT $\mathcal{A}$ with state space Q of size n, we took $c = Det(n,k) \cdot n$ (an upper bound on the size of the minimal accepting run graph of $\mathcal{A}$), and constructed an NBT $\mathcal{A}'$ with state space $3^Q \times \mathcal{R}_k(c)$, where $\mathcal{R}_k(c)$ is the set of functions $f : Q \rightarrow R_k(c)$ in which $f(q)$ is not odd with index j for all $q \in F_j$. For $l \leq c$, let $\mathcal{R}_k[l]$ be the restriction of $\mathcal{R}_k$ to functions with range $R_k(l)$, and let $\mathcal{A}'[l]$ be the NBT $\mathcal{A}'$ resulting from replacing the functions $\mathcal{R}_k[c]$ by $\mathcal{R}_k[c]$. Recall that the NBT $\mathcal{A}'[l]$ is empty iff all the run graphs of $\mathcal{A}$ of size at most l are not accepting. Thus, coming to check the emptiness of $\mathcal{A}$, the incremental approach proceeds as follows: start with a small l and check the nonemptiness of $\mathcal{A}'[l]$. If $\mathcal{A}'[l]$ is not empty, then $\mathcal{A}$ is not empty, and we can terminate with a "nonempty" output. Otherwise, increase l, and repeat the procedure. When $l = c$ and $\mathcal{A}'[l]$ is still empty, we can terminate with an "empty" output.

As argued for UCTs in [15], it is possible to take advantage of the work done during the emptiness test of $\mathcal{A}'[l_1]$, when testing emptiness of $\mathcal{A}'[l_2]$, for $l_2 > l_1$. To see this, note that the state space of $\mathcal{A}'[l_2]$ consists of the union of $3^Q \times \mathcal{R}_k[l_1]$ (the state space of $\mathcal{A}'[l_1]$) with $3^Q \times (\mathcal{R}_k[l_2] \setminus \mathcal{R}_k[l_1])$ (states whose $f \in \mathcal{R}_k[l_2]$ has a state that is mapped to a rank greater than l_1). Also, since ranks can only decrease, once the NBT $\mathcal{A}'[l_2]$ reaches a state of $\mathcal{A}'[l_1]$, it stays in such states forever. So, if we have already checked the nonemptiness of $\mathcal{A}'[l_1]$ and have recorded the classification of its states to empty and nonempty, the additional work needed in the nonemptiness test of $\mathcal{A}'[l_2]$ concerns only states in $3^Q \times (\mathcal{R}[l_2] \setminus \mathcal{R}_k[l_1])$.

We now describe how the incremental approach can be extended to a compositional one. Let $\mathcal{S} = \langle \Sigma, Q, \delta, q_{in}, \{F_1, \ldots, F_k\} \rangle$ and $\mathcal{S}' = \langle \Sigma, Q', \delta', q'_{in}, \{F'_1, \ldots, F'_{k'}\} \rangle$ be UGCWs specifying required behaviors. Let $n = |Q|$ and $n' = |Q'|$. Without loss of generality, assume that the state spaces Q and Q' are disjoint. We can define the intersection of $\mathcal{S}$ and $\mathcal{S}'$ as the UGCW P obtained by putting $\mathcal{S}$ and $\mathcal{S}'$ "side by side"; thus[9] $P = \langle \Sigma, Q \cup Q', \delta \cup \delta', \{q_{in}, q'_{in}\}, \{F_1 \cup Q', \ldots, F_k \cup Q', F'_1 \cup Q, \ldots, F'_{k'} \cup Q\} \rangle$. Note that it is indeed the case that P has an accepting run on a word w iff both $\mathcal{S}$ and $\mathcal{S}'$ has an accepting run on w.

Let $\mathcal{A}$ and $\mathcal{A}'$ be the NBTs to which realizability of $\mathcal{S}$ and $\mathcal{S}'$ is reduced, respectively. A non-compositional approach generates the NBT that corresponds to P. By Theorem 2, this results in an NBT $\mathcal{U}$ with state space $3^{Q \cup Q'} \times R_{k+k'}(p)^{Q \cup Q'}$, for $p = Det(n+n', k+k') \cdot (n+n')$. On the other hand, the state spaces of $\mathcal{A}$ and $\mathcal{A}'$ are much smaller, and are $3^Q \times R_k(c)^Q$ and $3^{Q'} \times R_{k'}(c')^{Q'}$, for $c = Det(n,k) \cdot n$ and $c' = Det(n', k') \cdot n'$. respectively.

Let us examine the structure of the state space of $\mathcal{U}$ more carefully. Each of its states can be viewed as a triplet $\langle S \cup S', O \cup O', f \rangle$, for $O \subseteq S \subseteq Q$, $O' \subseteq S' \subseteq Q'$, and $f : Q \cup Q' \rightarrow R_{k+k'}(p)$. For f as above, let $f_{|Q}$ and $f_{|Q'}$ denote the restrictions of f to Q and Q', respectively. Note that if f maps the states in S to ranks in $R_k(c)$ and maps states in S' to ranks in $R_{k'}(c')$, then the state $\langle S \cup S', O \cup O', f \rangle$ corresponds to the states $\langle S, O, f_{|Q} \rangle$ of $\mathcal{A}$ and $\langle S', O', f_{|Q'} \rangle$ of $\mathcal{A}'$. Moreover, if one of these states is empty, so is $\langle S \cup S', O \cup O', f \rangle$. This observation is the key to our compositional algorithm.

[9] For technical simplicity, we allow P to have two initial states. This can be easily avoided by adding a new initial state whose transitions are the union of the transitions from q_{in} and q'_{in}.

For $l \leq c$ and $l' \leq c'$, let $\mathcal{U}[l, l']$ denote the NBT $\mathcal{U}$ restricted to states $\langle S \cup S', O \cup O', f\rangle$ in which $f(q)$, for $q \in S$, is in $R_k(l)$ and $f(q')$, for $q' \in S'$, is in $R_{k'}(l')$. We check the emptiness of $\mathcal{U}$ incrementally and compositionally as follows. We start with small l_1 and l'_1 and check the emptiness of $\mathcal{U}[l_1, l'_1]$. Doing so, we first mark as empty all states $\langle S \cup S', O \cup O', f\rangle$ for which either $\langle S, O, f_{|Q}\rangle$ is empty in $\mathcal{A}$ or $\langle S', O', f_{|Q'}\rangle$ is empty in $\mathcal{A}'$, and continue the emptiness check only in the (expectedly much smaller) state space. If $\mathcal{U}[l_1, l'_1]$ is not empty, we are done. Otherwise, we increase our parameters to l_2 and l'_2, with $l_2 \geq l_1$ and $l'_2 \geq l'_1$. Note that we need not increase both parameters. Checking the emptiness of $\mathcal{U}[l_2, l'_2]$, we make use of the information gathered in the emptiness checks of $\mathcal{A}[l_2]$, $\mathcal{A}'[l'_2]$, as well as $\mathcal{U}[l_1, l'_1]$. The procedure continues until we either reach l_j and l'_j for which $\mathcal{U}[l_j, l'_j]$ is not empty, in which case the specification is realizable, or we find that $\mathcal{U}[p, p]$ is empty, in which case the specification is not realizable.

We note that, as with the incremental approach, the significant advantage of the compositional approach is when the specification is realizable, and especially when $\mathcal{U}[l, l']$ is not empty for l and l' smaller than c and c' – thus we can use information about $\mathcal{A}$ and $\mathcal{A}'$ all the way to the positive response. We also note that the incremental approach is possible due to the simple structure of the state spaces of the NBTs to which we have reduced the realizability problem. This simple structure also makes it easy to implement our approach symbolically: the state space of the NBT consists of sets of states and a ranking function, it can be encoded by Boolean variables, and the NBT's transitions can be encoded by relations on these variables and a primed version of them. The fixpoint solution for the nonemptiness problem of NBT (c.f., [30]) then yields a symbolic solution to the original UGCT nonemptiness problem. Moreover, checking the emptiness of $\mathcal{U}[l_j, l'_j]$, we can use BDDs for the empty states in $\mathcal{A}[l_j]$, $\mathcal{A}[l'_j]$, and $\mathcal{U}[l_{j-1}, l'_{j-1}]$. Finally, as discussed in [15], the BDDs that are generated by the symbolic nonemptiness procedure can be used to generate a symbolic witness strategy, from which we can synthesize a sequential circuit implementing the strategy.

References

1. M. Abadi, L. Lamport, and P. Wolper. Realizable and unrealizable concurrent program specifications. In *16th ICALP*, LNCS 372, pp 1–17. Springer-Verlag, 1989.
2. C. S. Althoff, W. Thomas, and N. Wallmeier. Observations on determinization of büchi automata. In *10th CIAA*, LNCS. Springer-Verlag, 2005.
3. Y. Choueka. Theories of automata on ω-tapes: A simplified approach. *JCSS*, 8:117–141, 1974.
4. A. Church. Logic, arithmetics, and automata. In *ICM, 1962*, pp 23–35, 1963.
5. E.M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.
6. D.L. Dill. *Trace theory for automatic hierarchical verification of speed independent circuits*. MIT Press, 1989.
7. J. Elgaard, N. Klarlund, and A. Möller. Mona 1.x: new techniques for WS1S and WS2S. In *10th CAV*, LNCS 1427, pp 516–520. Springer-Verlag, 1998.
8. E.A. Emerson. Automata, tableaux, and temporal logics. In *WLP*, LNCS 193, pp 79–87. Springer-Verlag, 1985.
9. R. Gerth, D. Peled, M.Y. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Protocol Specification, Testing, and Verification*, pp 3–18. 1995.

10. E. Grädel, W. Thomas, and T. Wilke. *Automata, Logics, and Infinite Games: A Guide to Current Research*. LNCS 2500. Springer-Verlag, 2002.
11. S. Gurumurthy, O. Kupferman, F. Somenzi, and M.Y. Vardi. On complementing nondeterministic Büchi automata. In *12th CHARME*, LNCS 2860, pp 96–110. Springer-Verlag, 2003.
12. A. Harding, M. Ryan, and P.Y. Schobbens. A new algorithm for strategy synthesis in ltl games. In *11th TACAS*, LNCS 3440, pp 477–492. Springer-Verlag, 2005.
13. M. JurzińSki. Small progress measures for solving parity games. In *17th STACS*, LNCS 1770, pp 290–301. Springer-Verlag, 2000.
14. O. Kupferman and M.Y. Vardi. From complementation to certification. In *10th TACAS*, LNCS 2988, pp 591–606. Springer-Verlag, 2004.
15. O. Kupferman and M.Y. Vardi. Safraless decision procedures. In *46th FOCS*, 2005.
16. O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *12th POPL*, pp 97–107, 1985.
17. Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, 1992.
18. S. Miyano and T. Hayashi. Alternating finite automata on ω-words. *TCS*, 32:321–330, 1984.
19. A.W. Mostowski. Regular expressions for infinite trees and a standard form of automata. In *CT*, LNCS 208, pp 157–168. Springer-Verlag, 1984.
20. D.E. Muller, A. Saoudi, and P.E. Schupp. Alternating automata, the weak monadic theory of the tree and its complexity. In *13th ICALP*, LNCS 226. Springer-Verlag, 1986.
21. N. Piterman. From nondeterministic Büchi and Streett automata to deterministic parity automata. In *25th LICS*, 2006. to appear.
22. N. Piterman, A. Pnueli, and Y. Saar. Design synthesis in action: Solving a 2exptime-complete problem in n^3. In *7th VMCAI*, LNCS 3855, pp 364–380. Springer-Verlag, 2006.
23. A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *16th POPL*, pp 179–190, 1989.
24. M.O. Rabin. Weakly definable relations and special automata. In *Symp. Math. Logic and Foundations of Set Theory*, pp 1–23. 1970.
25. M.O. Rabin. Automata on infinite objects and Church's problem. *AMS*, 1972.
26. R. Rosner. *Modular Synthesis of Reactive Systems*. PhD thesis, Weizmann Institute of Science, 1992.
27. S. Safra. On the complexity of ω-automata. In *29th FOCS*, pp 319–327, 1988.
28. S. Safra. Exponential determinization for ω-automata with strong-fairness acceptance condition. In *24th STOC*, 1992.
29. S. Tasiran, R. Hojati, and R.K. Brayton. Language containment using non-deterministic omega-automata. In *8th CHARME*, LNCS 987, pp 261–277, 1995. Springer-Verlag.
30. M.Y. Vardi and P. Wolper. Automata-theoretic techniques for modal logics of programs. *JCSS*, 32(2):182–221, 1986.
31. M.Y. Vardi and P. Wolper. Reasoning about infinite computations. *IC*, 115(1):1–37, 1994.
32. J. Yang and C.J.H. Seger. Introduction to generalized symbolic trajectory evaluation. In *19th DAC*, pp 360–367. IEEE, 2001.