

Synthesis of Trigger Properties*

Orna Kupferman[†]

Moshe Vardi[‡]

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 temporal 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 address this problem here by considering a certain class of PSL properties; this class covers most of the properties used in practice by system designers. We refer to this class as the class of trigger properties.

We show that the synthesis problem for trigger properties is more amenable to implementation than that of general PSL properties. While the problem is still 2EXPTIME-complete, it can be solved using techniques that are significantly simpler than the techniques used in general temporal synthesis. Not only can we avoid the use of Safra's determinization, but we can also avoid the use of progress ranks. Rather, the techniques used are based on classical subset constructions. This makes our approach amenable also to symbolic implementation, as well as an incremental implementation, in which the specification evolves over time.

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 [CGP99]. A frequent criticism against this approach, however, is that verification is done *after* significant resources have already 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*.

The classical approach to program synthesis is to extract a program from a proof that the specification is satisfiable [?, EC82, MW80, MW84]. In the late 1980s, several researchers realized that the classical approach to program synthesis is well suited to *closed* systems, but not to *open* (also called *reactive*) systems [ALW89, Dil89, PR89a]. In reactive systems, the program interacts with the environment, and a correct program should satisfy the specification with respect to all environments. Accordingly, 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. Thus, a strategy can 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 [PR89a]. A finite generator of an infinite tree accepted by

*Work supported in part by NSF grant CCF-0728882, by BSF grant 9800096, and by gift from Intel. Part of this work was done while the second author was on sabbatical at the Hebrew University of Jerusalem.

[†]School of Engineering and Computer Science, Hebrew University, Jerusalem, Israel. E-mail: orna@cs.huji.ac.il.

[‡]Department of Computer Science, Rice University, Houston, TX 77251-1892, U.S.A. E-mail: vardi@cs.rice.edu.

this automaton can be viewed as a finite-state program realizing the specification. This is closely related to the approach taken in [BL69, Rab72] in order to solve Church’s *solvability problem* [Chu63]. In [KV00, PR89b, WTD91, Var95] it was shown 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, and recent demonstrations of its applicability [BGJ⁺07], little of this theory has been reduced to practice. Some people argue that this is because the realizability problem for linear-temporal logic (LTL) specifications is 2EXPTIME-complete [PR89a, Ros92], 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. For example, while the model-checking problem for specifications in second-order logic has nonelementary complexity, the model-checking tool MONA [EKM98, Kl98] successfully verifies many specifications given in second-order logic. 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 [LP85]), 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 [Ros92] 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.

As argued in [KPV06], 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. The traditional approach for constructing tree automata for realizing strategies uses determinization of Büchi automata. Safra’s determinization construction has been notoriously resistant to efficient implementations [ATW05, THB95]¹, results in automata with a very complicated state space, and involves the parity acceptance condition. The best-known algorithms for parity-tree-automata emptiness [Jur00] are nontrivial already when applied to simple state spaces. Implementing them on top of the messy state space that results from determinization is highly complex, and is not amenable to optimizations and a symbolic implementation. In [KV05c, KPV06], we suggested an alternative approach, which avoids determinization and circumvents the parity condition. While the Safraless approach is much simpler and can be implemented symbolically, it is based on progress ranks. The need to manipulate ranks requires multi-valued data structures, making the symbolic implementation difficult [TV07, DR09]. This is in contrast with symbolic implementations of algorithms based on the subset construction without ranks, which perform well in practice [MS08a, MS08b].

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 *incremental* synthesis algorithms. Such algorithms can, for example, refine designs when provided with additional temporal properties.

One approach to tackle the algorithmic problems has been to restrict the class of allowed specification. In [AMPS98], the authors studied the case where the LTL formulas are of the form $\Box p$, $\Diamond p$, $\Box \Diamond p$, or $\Diamond \Box p$.² In [AT04], the authors considered the fragment of LTL consisting of boolean combinations of formulas of the form $\Box p$, as well as a richer fragment in which the \bigcirc operator is allowed. Since the

¹An alternative construction is equally hard [ATW05]. Piterman’s improvement of Safra includes the tree structures that proved hard to implement [Pit07].

²The setting in [AMPS98] is of real-time games, which generalizes synthesis.

games corresponding to formulas of these restricted fragments are simpler, the synthesis problem is much simpler; it can be solved in PSPACE or EXPSPACE, depending on the specific fragment. Another fragment of LTL, termed $GR(1)$, was studied in [PPS06]. In the $GR(1)$ fragment (generalized reactivity(1)) the formulas are of the form $(\Box \Diamond p_1 \wedge \Box \Diamond p_2 \wedge \dots \Box \Diamond p_m) \rightarrow (\Box \Diamond q_1 \wedge \Box \Diamond q_2 \wedge \dots \Box \Diamond q_n)$, where each p_i and q_i is a Boolean combination of atomic propositions. It is shown in [PPS06] that for this fragment, the synthesis problem can be solved in EXPTIME, and with only $O((mn \cdot 2^{|AP|})^3)$ symbolic operations, where AP is the set of atomic propositions.

We continue the approach on special classes of temporal properties, with the aim of focusing on properties that are used in practice. We study here the synthesis problem for TRIGGER LOGIC. Modern industrial-strength property-specification languages such as Sugar [BBE⁺01], ForSpec [AFF⁺02], and the recent standards PSL [EF06], and SVA [VR05] include regular expressions. TRIGGER LOGIC is a fragment of these logics that covers most of the properties used in practice by system designers. Technically, TRIGGER LOGIC consists of positive Boolean combinations of assertions about regular events, connected by the usual regular operators as well as temporal implication, \mapsto (“triggers”). For example, the TRIGGER LOGIC formula $(\mathbf{true}[*]; req; ack) \mapsto (\mathbf{true}[*]; grant)$ holds in an infinite computation if every request that is immediately followed by an acknowledge is eventually followed by a grant. Also, the TRIGGER LOGIC formula $(\mathbf{true}[*]; err) \mapsto !(\mathbf{true}[*]; ack)$ holds in a computation if once an error is detected, no acks can be sent.

We show that TRIGGER LOGIC formulas can be translated to deterministic Büchi automata using the two classical subset constructions: the determinization construction of [RS59] and the break-point construction of [MH84]. Accordingly, while the synthesis problem for TRIGGER LOGIC is still 2EXPTIME-complete, our synthesis algorithm is significantly simpler than the one used in general temporal synthesis. We show that this also yields several practical consequences: our approach is quite amenable to symbolic implementation, it can be applied to evolving specifications in an incremental fashion, and it can also be applied in an assume-guarantee setting. We believe that the simplicity of the algorithm and its practical advantages, coupled with the practical expressiveness of TRIGGER LOGIC, make an important step in bridging the gap between temporal-synthesis theory and practice.

2 Trigger Logic

The introduction of temporal logic to computer science, in [Pnu77], was a watershed point in the specification of reactive systems, which led to the development of model checking [CGP99]. The success of model checking in industrial applications led to efforts to develop “industrial” temporal logics such as Sugar [BBE⁺01] and ForSpec [AFF⁺02], as well as two industry-standard languages, PSL [EF06] and SVA [VR05].

A common feature of these languages is the use of regular expressions to describe temporal patterns. For example, the regular expression $request; \mathbf{true}^+; grant; \mathbf{true}^+; ack$ describes an occurrence of *request*, followed by *grant*, followed by *ack*, where these events are separated by nonempty intervals of arbitrary length. The advantage of using regular expressions over the classical temporal operators of LTL is that it avoids the need for deep nesting of untils. For that reason, regular expressions have proved to be quite popular with verification engineers,³ to the point that the regular layer is that main layer of SVA [VR05]. The key observation is that a very large fraction of temporal properties that arise in practice can be expressed in the form of $e_1 \mapsto e_2$ or $e_1 \mapsto !e_2$ (we generally use PSL syntax in this paper), which means that an e_1 pattern should, or should not, be followed by an e_2 pattern; see, for example, [SDC01]. As an example, consider the property: “If a snoop hits a modified line in the L1 cache, then the next transaction

³See <http://www.cs.rice.edu/~vardi/acclera-properties.pdf>.

must be a snoop writeback.” It can be expressed using the PSL formula

$$(\mathbf{true}[*]; \text{snoop}\&\&\text{modified}) \mapsto (!\text{trans_start}[*]; \text{trans_start}\&\&\text{writeback}).$$

The extension of LTL with regular expressions is called RELTL [BFG⁺05]. Here we study TRIGGER LOGIC– the fragment of RELTL consisting of positive Boolean combinations of formulas of the form $e_1 \mapsto e_2$ or $e_1 \mapsto !e_2$. We now describe this logic formally.

Let Σ be a finite *alphabet*. A *finite word* over Σ is a (possibly empty) finite sequence $w = \sigma_0 \cdot \sigma_1 \cdots \sigma_n$ of concatenated letters in Σ . The length of a word w is denoted by $|w|$. The symbol ε denotes the empty word. We use $w[i, j]$ to denote the subword $\sigma_i \cdots \sigma_j$ of w . If $i > j$, then $w[i, j] = \varepsilon$. *Regular Expressions* (REs) define languages by inductively applying union, concatenation, and repetition operators. Formally, an RE over an alphabet Σ is one of the following.

- \emptyset , ε , or σ , for $\sigma \in \Sigma$.
- $r_1|r_2$, $r_1;r_2$, $r[*]$, or $r[+]$, for REs r , r_1 , and r_2 .

We use $L(r)$ to denote the language that r defines. For the base cases, we have $L(\emptyset) = \emptyset$, $L(\varepsilon) = \{\varepsilon\}$, and $L(\sigma) = \{\sigma\}$. The operators $|$, $;$, $[*]$, and $[+]$ stand for union, concatenation, possibly empty repetition, and strict repetition, respectively. Formally,

- $L(r_1|r_2) = L(r_1) \cup L(r_2)$.
- $L(r_1;r_2) = \{w_1;w_2 : w_1 \in L(r_1) \text{ and } w_2 \in L(r_2)\}$.
- Let $r^0 = \{\varepsilon\}$ and let $r^i = r^{i-1};r_1$, for $i \geq 1$. Thus, $L(r^i)$ contains words that are the concatenation of i words in $L(r_1)$. Then, $L(r[*]) = \bigcup_{i \geq 0} r^i$ and $L(r[+]) = \bigcup_{i \geq 1} r^i$.

For a set X of elements, let $\mathcal{B}(X)$ denote the set of all Boolean functions $b : 2^X \rightarrow \{\mathbf{true}, \mathbf{false}\}$. In practice, members of $\mathcal{B}(X)$ are expressed by Boolean expressions over X , using with disjunction ($|$), conjunction ($\&\&$), and negation ($!$). Let $\mathcal{B}^+(X)$ be the restriction of $\mathcal{B}(X)$ to positive Boolean functions. That is, functions induced by formulas constructed from atoms in X with disjunction and conjunction, and we also allow the constants **true** and **false**. For a function $b \in \mathcal{B}(X)$ and a set $Y \subseteq X$, we say that Y satisfies b if assigning **true** to the elements in Y and **false** to the elements in $X \setminus Y$ satisfies b .

For a set AP of atomic propositions, let $\Sigma = \mathcal{B}(AP)$, and let \mathcal{R} be a set of atoms of the form r or $!r$, for a regular expression r over Σ . For example, for $AP = \{p, q\}$, the set \mathcal{R} contains the regular expression $(p|!q)[*]|(p;p)$ and also contains $!((p|!q)[*]|(p;p))$.

The linear temporal logic TRIGGER LOGIC is a formalism to express temporal implication between regular events. We consider TRIGGER LOGIC in a positive normal form, where formulas are constructed from atoms in \mathcal{R} by means of Boolean operators, regular expressions, and temporal implication (\mapsto). The syntax of TRIGGER LOGIC is defined as follows (we assume a fixed set AP of atomic propositions, which induces the fixed sets Σ and \mathcal{R}).

1. A *regular assertion* is a positive Boolean formula over \mathcal{R} .
2. A *trigger formula* is of the form $r \mapsto \theta$, for a regular expression r over Σ and a regular assertion θ .
3. A TRIGGER LOGIC formula is a positive Boolean formula over trigger formulas.

Intuitively, $r \mapsto \theta$ asserts that all prefixes satisfying r are followed by a suffix satisfying θ . The linear temporal logic TRIGGER LOGIC is a formalism to express temporal implication between regular events. For example, $(\mathbf{true}[*]; p) \mapsto (\mathbf{true}[*]; q)$ is regular formula, equivalent to the LTL formula $G(p \rightarrow Fq)$. We use $\theta(e_1, \dots, e_k, !e'_1, \dots, !e'_k)$ to indicate that the regular assertion θ is over the regular expressions e_1, \dots, e_k and the negations of the regular expressions e'_1, \dots, e'_k . Note that we do not allow nesting of \mapsto in our formulas.

The semantics of TRIGGER LOGIC formulas is defined with respect to infinite words over the alphabet 2^{AP} . Consider an infinite word $\pi = \pi_0, \pi_1, \dots \in (2^{AP})^\omega$. For indices i and j with $0 \leq i \leq j$, and a language $L \subseteq \Sigma^*$, we say that π_i, \dots, π_{j-1} *tightly satisfies* L , denoted $\pi, i, j, \models L$, if there is a word $b_0 \cdot b_1 \cdots b_{j-1-i} \in L$ such that for all $0 \leq k \leq j-1-i$, we have that $b_i(\pi_{i+k}) = \mathbf{true}$. Note that when $i = j$, the interval

π_i, \dots, π_{j-1} is empty, in which case $\pi, i, j \models L$ iff $\varepsilon \in L$. For an index $i \geq 0$ and a language $L \subseteq \Sigma^*$, we say that π_i, π_{i+1}, \dots satisfies L , denoted $\pi, i \models L$, if $\pi, i, j \models L$ for some $j \geq i$. Dually, π_i, π_{i+1}, \dots satisfies $!L$, denoted $\pi, i \models !L$, if there is no $j \geq i$ such that $\pi, i, j \models L$. Note that $\pi, i \models !L$ iff $\pi, i \not\models L$; note that both are different from $\pi, i \models \Sigma^* \setminus L$. For a regular assertion θ , we say that π_i, π_{i+1}, \dots satisfies θ , denoted $\pi, i \models \theta$ if there is a set $Y \subseteq \mathcal{R}$ such that Y satisfies θ , $\pi, i \models L(r)$ for all $r \in Y$, and $\pi, i \models !L(r)$ for all $r \in Y$.

We can now define the semantics of the \mapsto operator.

- $\pi, i \models (r \mapsto \theta)$ if for all $j \geq i$ such that $\pi, i, j \models L(r)$, we have $\pi, j \models \theta$.

For a TRIGGER LOGIC formula ψ , a path π satisfies ψ in index i , denoted $\pi, i \models \psi$, if π, i satisfies a set X of regular formulas such that X satisfies ψ . Finally, π satisfies ψ if π satisfies ψ in index 0.

Thus, the formula $(\mathbf{true}[*]; p) \mapsto (\mathbf{true}[*]; q)$ holds in an infinite word $\pi \in 2^{\{p,q\}}$ if every p is eventually followed by q . Indeed, for all $j \geq 0$, if $\pi, 0, j \models L(\mathbf{true}[*]; p)$, which holds iff $\pi_j \models p$, then $\pi, j \models \mathbf{true}[*]; q$. The latter holds iff there is $k \geq j$ such that $\pi, j, k \models \mathbf{true}[*]; q$, which holds iff $\pi_k \models q$.

3 Automata on Words and Trees

An *automaton on infinite words* is a tuple $\mathcal{A} = \langle \Sigma, Q, q_0, \rho, \alpha \rangle$, where Σ is the input alphabet, Q is a finite set of states, $\rho : Q \times \Sigma \rightarrow 2^Q$ is a transition function, $q_0 \in Q$ is an initial state, and α is an acceptance condition (a condition that defines a subset of Q^ω). Intuitively, $\rho(q, \sigma)$ is the set of states that \mathcal{A} can move into when it is in state q and it reads the letter σ . Since the transition function of \mathcal{A} may specify many possible transitions for each state and letter, \mathcal{A} is not *deterministic*. If ρ is such that for every $q \in Q$ and $\sigma \in \Sigma$, we have that $|\rho(q, \sigma)| = 1$, then \mathcal{A} is a deterministic automaton. We extend ρ to sets of states in the expected way, thus, for $S \subseteq Q$, we have that $\rho(S, \sigma) = \bigcup_{s \in S} \rho(s, \sigma)$.

A *run* of \mathcal{A} on w is a function $r : \mathbb{N} \rightarrow Q$ where $r(0) = q_0$ (i.e., the run starts in the initial state) and for every $l \geq 0$, we have $r(l+1) \in \rho(r(l), \sigma_l)$ (i.e., the run obeys the transition function). In automata over finite words, acceptance is defined according to the last state visited by the run. When the words are infinite, there is no such thing “last state”, and acceptance is defined according to the set $\text{Inf}(r)$ of states that r visits *infinitely often*, i.e., $\text{Inf}(r) = \{q \in Q : \text{for infinitely many } l \in \mathbb{N}, \text{ we have } r(l) = q\}$. As Q is finite, it is guaranteed that $\text{Inf}(r) \neq \emptyset$. The way we refer to $\text{Inf}(r)$ depends on the acceptance condition of \mathcal{A} . In *Büchi automata*, $\alpha \subseteq Q$, and r is accepting iff $\text{Inf}(r) \cap \alpha \neq \emptyset$. Dually, in *co-Büchi automata*, $\alpha \subseteq Q$, and r is accepting iff $\text{Inf}(r) \cap \alpha = \emptyset$.

Since \mathcal{A} is not deterministic, it may have many runs on w . In contrast, a deterministic automaton has a single run on w . There are two dual ways in which we can refer to the many runs. When \mathcal{A} is an *existential* automaton (or simply a *nondeterministic* automaton, as we shall call it in the sequel), it accepts an input word w iff there exists an accepting run of \mathcal{A} on w .

Automata can also run on trees. For our application, we only need deterministic Büchi tree automata. Given a set D of directions, a *D-tree* is a set $T \subseteq D^*$ such that if $x \cdot c \in T$, where $x \in D^*$ and $c \in D$, then also $x \in T$. If $T = D^*$, we say that T is a full *D-tree*. The elements of T are called *nodes*, and the empty word ε is the *root* of T . For every $x \in T$, the nodes $x \cdot c$, for $c \in D$, are the *successors* of x . Each node D^* has a *direction* in D . The direction of the root is d_0 , for some designated $d_0 \in D$, called the *root direction*. The direction of a node $x \cdot d$ is d . We denote by $\text{dir}(x)$ the direction of node x . A *path* π of a tree T is a set $\pi \subseteq T$ such that $\varepsilon \in \pi$ and for every $x \in \pi$, either x is a leaf or there exists a unique $c \in D$ such that $x \cdot c \in \pi$. Given an alphabet Σ , a Σ -labeled *D-tree* is a pair $\langle T, \tau \rangle$ where T is a tree and $\tau : T \rightarrow \Sigma$ maps each node of T to a letter in Σ .

A *deterministic Büchi tree automaton* is $\mathcal{A} = \langle \Sigma, D, Q, \delta, q_0, \alpha \rangle$, where Σ , Q , q_0 , and α , are as in Büchi word automata, and $\delta : Q \times \Sigma \rightarrow Q^{|D|}$ is a (deterministic) transition function. Intuitively, in each of its transitions, \mathcal{A} splits into $|D|$ copies, each proceeding to a subtree whose root is the successor of

the current node. For a direction $d \in D$, having $\delta(q, \sigma)(d) = q'$ means that if \mathcal{A} is now in state q and it reads the letter σ , then the copy that proceeds to direction d moves to state q' .

Formally, a *run* of \mathcal{A} on an input Σ -labeled D -tree $\langle D^*, \tau \rangle$, is a Q -labeled tree $\langle D^*, r \rangle$ such that $r(\varepsilon) = q_0$ and for every $x \in D^*$, and direction $d \in D$, we have that $r(x \cdot d) = \delta(r(x), \tau(x))(d)$. If, for instance, $D = \{0, 1\}$, $r(0) = q_2$, $\tau(0) = a$, and $\delta(q_2, a)(0) = q_1$ and $\delta(q_2, a)(1) = q_2$, then $r(0 \cdot 0) = q_1$ and $r(0 \cdot 1) = q_2$. Given a run $\langle D^*, r \rangle$ and a path $\pi \subset D^*$, we define $\text{inf}(r|\pi) = \{q \in Q : \text{for infinitely many } x \in \pi, \text{ we have } r(x) = q\}$. A run r is accepting iff for all paths $\pi \subset D^*$, we have $\text{inf}(r|\pi) \cap \alpha \neq \emptyset$. That is, iff for each path $\pi \subset D^*$ there exists a state in α that r visits infinitely often along π . An automaton \mathcal{A} accepts $\langle D^*, \tau \rangle$ its run on it is accepting.

We use three-letter acronyms in $\{D, N\} \times \{B, C\} \times \{W, T\}$ to describe types of automata. The first letter describes the transition structure (deterministic or nondeterministic), the second letter describes the acceptance condition (Büchi or co-Büchi), and the third letter designates the objects recognized by the automata (words or trees). Thus, for example, NCW stands for nondeterministic Büchi word automata and NBT stands for nondeterministic Büchi tree automata.

4 Expressiveness

In this section we characterize the expressive power of TRIGGER LOGIC and show that is equivalent to that of DBW.

Proposition 4.1. *Given a regular formula ψ of the form $r \mapsto \theta(e_1, \dots, e_k, !e'_1, \dots, !e'_k)$, we can construct an NCW with $|r| + (2^{|e_1| + \dots + |e_k|} |e'_1| \dots |e'_k|)$ states that accepts exactly all computations that violate ψ .*

Proof. We start with the special case where $k = k' = 1$ and θ is a disjunction, thus the formula we consider is $\psi = r \mapsto (e \vee !e')$. A path $\pi = \pi_0, \pi_1, \dots$ violates the formula $r \mapsto (e \vee !e')$ iff there is $i \geq 0$ such that $\pi, 0, i \models L(r)$, $\pi, i \models L(e')$, and $\pi, i \not\models !L(e)$.

We describe an NCW \mathcal{U} that accepts paths that violate ψ . Let $\mathcal{A}_1, \mathcal{A}_2$, and \mathcal{A}_3 be NFWs for r, e , and e' , respectively. Let \mathcal{A}'_2 be the DCW obtained by determinizing \mathcal{A}_2 , replacing its accepting states by rejecting sinks, and making all other states accepting. Also, let \mathcal{A}'_3 be the NCW obtained by replacing the accepting states of \mathcal{A}_3 by accepting sinks. Finally, Let \mathcal{A} be the product of \mathcal{A}'_2 and \mathcal{A}'_3 . The NCW \mathcal{U} starts with \mathcal{A}_1 . From every accepting state of \mathcal{A}_1 , it can start executing \mathcal{A} . The acceptance condition requires a run to eventually get stuck in an accepting sink of \mathcal{A}'_3 that is not a rejecting sink of \mathcal{A}'_2 . Formally, for $i \in \{1, 2, 3\}$, let $\mathcal{A}_i = \langle \Sigma, Q_i, \delta_i, Q_i^0, \alpha_i \rangle$. Then, $\mathcal{A}'_2 = \langle \Sigma, 2^{Q_2}, \delta'_2, \{Q_2^0\}, \alpha'_2 \rangle$, where $\alpha'_2 = \{S : S \cap \alpha_2 = \emptyset\}$, and for all $S \in 2^{Q_2}$ and $\sigma \in \Sigma$, we have

$$\delta'_2(S, \sigma) = \begin{cases} \delta_2(S, \sigma) & \text{if } S \cap \alpha_2 = \emptyset \\ S & \text{otherwise} \end{cases}$$

Note that \mathcal{A}'_2 accepts exactly all infinite words none of whose prefixes are accepted by \mathcal{A}_2 . Also, $\mathcal{A}'_3 = \langle \Sigma, Q_3, \delta'_3, Q_3^0, \alpha_3 \rangle$, where for all $q \in Q_3$ and $\sigma \in \Sigma$, we have

$$\delta'_3(q, \sigma) = \begin{cases} \delta_3(q, \sigma) & \text{if } q \notin \alpha_3 \\ \{q\} & \text{otherwise} \end{cases}$$

Note that \mathcal{A}'_3 accepts exactly all infinite words that have a prefix accepted by \mathcal{A}_3 .

Now, $\mathcal{U} = \langle \Sigma, Q_1 \cup (2^{Q_2} \times Q_3), \delta, Q_1^0, \alpha \rangle$, where for all $q \in Q_1$ and $\sigma \in \Sigma$, we have

$$\delta(q, \sigma) = \begin{cases} \delta_1(q, \sigma) & \text{if } q \notin \alpha_1 \\ \delta_1(q, \sigma) \cup (\{\delta'_2(Q_2^0, \sigma)\} \times \delta'_3(Q_3^0, \sigma)) & \text{otherwise} \end{cases}$$

Also, for all $\langle S, q \rangle \in 2^{Q_2} \times Q_3$ and $\sigma \in \Sigma$, we have

$$\delta(\langle S, q \rangle, \sigma) = \{\delta'_2(S, \sigma)\} \times \delta'_3(q, \sigma).$$

We can partition the state space of \mathcal{U} to three sets:

- $P_1 = Q_1 \cup \{\langle S, q \rangle : S \cap \alpha_2 = \emptyset \text{ and } q \notin \alpha_3\}$,
- $P_2 = \{\langle S, q \rangle : S \cap \alpha_2 = \emptyset \text{ and } q \in \alpha_3\}$, and
- $P_3 = \{\langle S, q \rangle : S \cap \alpha_2 \neq \emptyset \text{ and } q \in Q_3\}$.

The acceptance condition requires an accepting run to eventually leave the automaton \mathcal{A}_1 and then, in the product of \mathcal{A}'_2 with \mathcal{A}'_3 , avoid the rejecting sinks of \mathcal{A}'_2 and get stuck in the accepting sinks of \mathcal{A}'_3 . By the definition of δ , each run of \mathcal{U} eventually gets trapped in a set P_i . Hence, the above goal can be achieved by defining the co-Büchi condition $\alpha = P_1 \cup P_3$.

In the general case, where the formula is of the form $r \mapsto \theta(e_1, \dots, e_k, !e'_1, \dots, !e'_{k'})$, we define, in a similar way, an NCW \mathcal{U} for paths that violate the formula. As in the special case detailed above, we determinize the NFWs for e_1, \dots, e_k and make their accepting states rejecting sinks. Let $\mathcal{A}_2^1, \dots, \mathcal{A}_2^k$ be the automata obtained as described above. Then, we take the NFWs for $e'_1, \dots, e'_{k'}$ and make their accepting states accepting sinks. Let $\mathcal{A}_3^1, \dots, \mathcal{A}_3^{k'}$ be the automata obtained as described above. The NCW \mathcal{U} starts with the NFW \mathcal{A}_1 for r . From every accepting state of \mathcal{A}_1 it can take the transitions from the initial states of the product \mathcal{A} of $\mathcal{A}_2^1, \dots, \mathcal{A}_2^k, \mathcal{A}_3^1, \dots, \mathcal{A}_3^{k'}$. In the product \mathcal{A} , each state is of the form $\langle S_1, \dots, S_k, q_1, \dots, q_{k'} \rangle$ and we partition the states to sets according to the membership of the underlying states in the sinks. Thus, \mathcal{U} is partitioned to $1 + 2^{k+k'}$ sets: one for the states of \mathcal{A}_1 , and then a set P_v , for each $v \in 2^{k+k'}$. For $v \in 2^{k+k'}$, the set P_v contains exactly all states $\langle S_1, \dots, S_k, q_1, \dots, q_{k'} \rangle$ such that for all $1 \leq i \leq k$, we have $S_i \cap \alpha_2^i \neq \emptyset$ iff $v[i] = 0$ and for all $1 \leq j \leq k'$, we have $q_j \in \alpha_3^j$ iff $v[k+j] = 0$.

It is not hard to see that the sets P_v are ordered: $P_v \geq P_{v'}$ (that is, a transition from P_v to $P_{v'}$ is possible) iff for each index $1 \leq i \leq k+k'$, we have $v[i] \geq v'[i]$. It is left to define the acceptance condition. Recall that θ is a positive Boolean formula over $e_1, \dots, e_k, !e'_1, \dots, !e'_{k'}$. In order to violate a requirement associated with e_i , the projection of a run of \mathcal{U} on the component of \mathcal{A}_2^i has to avoid its rejecting sinks. In order to violate a requirement associated with $!e'_j$, the projection of a run of \mathcal{U} on the component of \mathcal{A}_3^j has to reach an accepting sink. Accordingly, given θ , we say that $v \in 2^{k+k'}$ satisfies θ if assigning **true** to e_i , for $1 \leq i \leq k$, such that $v[i] = 0$ and to $!e'_j$, for $1 \leq j \leq k'$, such that $v[k+j] = 1$, and assigning **false** to all other atoms, satisfies θ . Now, we define the acceptance condition of \mathcal{U} to that a run is accepting if it gets stuck in a set P_v for which v satisfies θ . Thus, α is the union of the sets P_v for which v does not satisfy θ . As required, \mathcal{U} has $|r| + (2^{|e_1| + \dots + |e_k|} |e'_1| \dots |e'_{k'}|)$ states.

Note that we determinize only NFWs associated with regular expressions that are not in the scope of $!$. Also, a tighter construction can take the structure of θ into an account and handle conjunctions in θ by nondeterminism rather than by taking the product. \square

Theorem 4.2. *A TRIGGER LOGIC formula can be translated to equivalent DBW. The blow-up in the translation is doubly exponential.*

Proof. Consider an NCW \mathcal{A} with n states. By applying to \mathcal{A} a construction dual to the break-point construction of [MH84] we get a DCW \mathcal{A}' equivalent to \mathcal{A} with $3^{O(n)}$ states. For completeness, we describe the construction below. Intuitively, \mathcal{A}' follows the standard subset construction applied to \mathcal{A} . In order to make sure that every infinite path visits states in α only finitely often, \mathcal{A}' maintains in addition to the set S that follows the subset construction, a subset O of S consisting of states along runs that have not visited α since the last time the O component was empty. The acceptance condition of \mathcal{A}' then requires O to become empty only finitely often. Indeed, this captures the fact that there is a run of \mathcal{A} that

eventually prevents the set O from becoming empty, and thus it is a run along which α is visited only finitely often.

Formally, let $\mathcal{A} = \langle \Sigma, Q, q_0, \rho, \alpha \rangle$. Then, $\mathcal{A}' = \langle \Sigma, Q', q'_0, \rho', \alpha' \rangle$, where

- $Q' \subseteq 2^Q \times 2^Q$ is such that $\langle S, O \rangle \in Q'$ if $O \subseteq S \subseteq Q$.
- $q'_0 = \langle \{q_{in}\}, \emptyset \rangle$,
- $\rho' : Q' \times \Sigma \rightarrow Q'$ is defined, for all $\langle S, O \rangle \in Q'$ and $\sigma \in \Sigma$, as follows.

$$\rho'(\langle S, O \rangle, \sigma) = \begin{cases} \langle \rho(S, \sigma), \rho(O, \sigma) \setminus \alpha \rangle & \text{if } O \neq \emptyset \\ \langle \rho(S, \sigma), \rho(S, \sigma) \setminus \alpha \rangle & \text{if } O = \emptyset. \end{cases}$$

- $\alpha' = 2^Q \times \{\emptyset\}$.

Given a TRIGGER LOGIC formula Ψ , let ψ_1, \dots, ψ_n be its underlying regular formulas. We saw in Proposition 4.1 that given a regular formula ψ of the form $r \mapsto \theta(e_1, \dots, e_i, !e'_k, \dots, !e'_{k'})$, we can construct an NCW with $|r| + (2^{|e_1| + \dots + |e_k|} |e'_1| \dots |e'_{k'}|)$ states that accepts exactly all computations that violate ψ . Let $\mathcal{A}_1, \dots, \mathcal{A}_n$ be the NCWs corresponding to the negations of ψ_1, \dots, ψ_n . For every $1 \leq i \leq n$, we can construct, as described above, a DCW \mathcal{A}'_i equivalent to \mathcal{A}_i . By dualizing \mathcal{A}'_i , we get a DBW for ψ_i . Now, since DBWs are closed under union and intersection (cf. [Cho74]), we can construct a DBW \mathcal{A} for Ψ . Note that \mathcal{A} is doubly exponential in the size of Ψ . \square

It remains to show that we can translate from DBW to TRIGGER LOGIC.

Theorem 4.3. *Given a DBW \mathcal{A} , we can construct a TRIGGER LOGIC formulas of size exponential in $|\mathcal{A}|$ that is satisfied precisely by the computations that are accepted by \mathcal{A} .*

Proof. Let $\mathcal{A} = \langle \Sigma, Q, q_0, \rho, \alpha \rangle$. For $q \in \alpha$, let \mathcal{A}_q be the DFW $\mathcal{A} = \langle \Sigma, Q, q_0, \rho, \{q\} \rangle$, and let \mathcal{A}_q^q be the DFW $\mathcal{A} = \langle \Sigma, Q, q, \rho, \{q\} \rangle$. We do not want \mathcal{A}_q and \mathcal{A}_q^q to accept the empty word ε , so the initial state can be renamed if needed. Let e_q and e_q^q be regular expressions equivalent to \mathcal{A}_q and \mathcal{A}_q^q . By [HU79], the lengths of e_q and e_q^q are exponential in \mathcal{A} .

A word $w \in \Sigma^\omega$ is accepted by \mathcal{A} iff there is $q \in \alpha$ such that the run of \mathcal{A} on q visits q infinitely often. Thus, the run visits q eventually, and all visits to q are followed by another visits in the (strict) future. We can therefore specifies the set of words that are accepted by \mathcal{A} using the TRIGGER LOGIC formula

$$\bigvee_{q \in \alpha} ((\mathbf{true} \mapsto e_q) \wedge (e_q \mapsto e_q^q)).$$

\square

The class of linear temporal properties that can be expressed by DBW was studied in [KV05b], where it is shown to be precisely the class of linear temporal properties that can be expressed in the alternation-free μ -calculus (AFMC). The translation is with respect to Kripke structures. A given DBW \mathcal{A} can be translated to an AFMC formula φ_A such that for every Kripke structure K we have that $K \models \mathcal{A}$ iff $K \models \varphi_A$, where $K \models \mathcal{A}$ if all computations of K are accepted by \mathcal{A} . Generally, the translation to AFMC requires going through DBW, which may involve a doubly exponentially blow-up, as in our translation from TRIGGER LOGIC to DBW. For TRIGGER LOGIC, we can go to AFMC via the NCW constructed Proposition 4.1, with an exponential, rather than a doubly exponential blow-up.

5 Synthesis

A *transducer* is a labeled finite graph with a designated start node, where the edges are labeled by D (“input alphabet”) and the nodes are labeled by Σ (“output alphabet”). 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, $\eta : S \times D \rightarrow S$ is a deterministic transition function, and $L : S \rightarrow \Sigma$ is a labeling function. We define $\eta : D^* \rightarrow S$ in the standard way: $\eta(\varepsilon) = s_{in}$, and for $x \in D^*$ and $d \in D$, we have $\eta(x \cdot d) = \eta(\eta(x), d)$. Now, 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} .

Given a TRIGGER LOGIC formula ψ over sets I and O of input and output signals (that is, $AP = I \cup O$), the *realizability problem* for ψ is to decide whether there is a *strategy* $f : (2^I)^* \rightarrow 2^O$, generated by a transducer⁴ such that all the computations of the system generated by f satisfy ψ [PR89a]. Formally, 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 $o_j = f(i_0 \cdot i_1 \cdots i_{j-1})$.

5.1 Upper bound

In this section we show that the translation of TRIGGER LOGIC formulas to automata, described earlier, yields a 2EXPTIME synthesis algorithm for TRIGGER LOGIC.

Theorem 5.1. *The synthesis problem for TRIGGER LOGIC is in 2EXPTIME.*

Proof. Consider a TRIGGER LOGIC formula Ψ over $I \cup O$. By Theorem 4.2, the formula Ψ can be translated to a DBW \mathcal{A} . The size of \mathcal{A} is doubly exponential in the length of Ψ , and its alphabet is $\Sigma = 2^{I \cup O}$. Let $\mathcal{A} = \langle 2^{I \cup O}, Q, q_0, \delta, \alpha \rangle$, and let $\mathcal{A}_I = \langle 2^O, 2^I, Q, q_0, \delta_I, \alpha \rangle$ be the DBT obtained by expanding \mathcal{A} to 2^O -labeled 2^I -trees. Thus, for every $q \in Q$ and $o \in 2^O$, we have⁵

$$\delta_I(q, o) = \bigwedge_{i \in 2^I} (i, \delta(q, i \cup o)).$$

We now have that \mathcal{A} is realizable iff \mathcal{A}_I is not empty. Indeed, \mathcal{A}_I accepts exactly all 2^O -labeled 2^I -trees all of whose computations are in $L(\mathcal{A})$. Furthermore, by the nonemptiness-test algorithm of [VW86], the DBT \mathcal{A}_I is not empty iff there is a finite state transducer that realizes $L(\mathcal{A})$. \square

We discuss the practical advantages of our synthesis algorithm for TRIGGER LOGIC in Section 6.

5.2 Lower Bound

The doubly-exponential lower bound for LTL synthesis is tightly related to the fact a translation of an LTL formula to a deterministic automaton may involve a doubly-exponential blow-up [KV98a]. For TRIGGER LOGIC formulas, such a blow-up seems less likely, as the translation of regular expressions to nondeterministic automata is linear, while the translation of LTL to automata is exponential [VW94]. As we show below, the translation does involve a doubly exponential blow-up, even for formulas of the form $r \mapsto e$, that is, when the underlying regular expressions appear positively. Intuitively, it follows from the need to monitor all the possible runs of an NFW for e on different suffixes (these whose corresponding prefix satisfies r) of the input word.

Theorem 5.2. *There is a regular expression r and a family of regular expression e_1, e_2, \dots such that for all $n \geq 1$, the length of e_n is polynomial in n and the smallest DBW for the TRIGGER LOGIC formula $r \mapsto e_n$ is doubly-exponential in n .*

⁴It is known that if some transducer that generates f exists, then there is also a finite-state transducer [PR89a].

⁵Note that the fact \mathcal{A} is deterministic is crucial. A similar construction for a nondeterministic \mathcal{A} results in \mathcal{A}_I whose language may be strictly contained in the required language.

Proof. Let $\psi_n = r \mapsto e_n$. We define r and e_n over $\Sigma = \{0, 1, \#, \$\}$ so that the language of $!\psi_n$ contains exactly all words w such that there is a position j with $w[j] = \#, w[j+1, j+n+1] \in (0|1)^n$, and either there is no position $k > j$ with $w[k] = \$$, or $w[j+1, j+n+1] = w[k+1, k+n+1]$ for the minimal position $k > j$ with $w[k] = \$$.

By [CKS81], the smallest deterministic automaton that recognizes $!\psi_n$ has at least 2^{2^n} states. The proof in [CKS81] considers a language of the finite words. The key idea, however, is valid also for our setting, and implies that the smallest DBW for ψ has at least 2^{2^n} states: whenever the automaton reads $\$,$ it should remember the set of words in $\#; (0|1)^n$ that have appeared since the last $\$$ (or the beginning of the word, if we are in the first $\$$).

We define $r = (0|1|\#)[*]; \#$, and e_n is the union of the following REs:

- $\mathbf{true}^i; (\#|\$)$, for $1 \leq i \leq n$: the suffix does not begin with a word in $(0|1)^n$.
- $(\mathbf{true}^i; 0; (!\$)[*]; \$; \mathbf{true}^i; !0)$, for $1 \leq i \leq n$: there is $1 \leq i \leq n$ such that the letter in the i -th position is 0 and is different from the letter in the i -th position after the first $\$$ in the suffix.
- $(\mathbf{true}^i; 1; (!\$)[*]; \$; \mathbf{true}^i; !1)$, for $1 \leq i \leq n$: there is $1 \leq i \leq n$ such that the letter in the i -th position is 1 and is different from the letter in the i -th position after the first $\$$ in the suffix.

It is not hard to see that a word w satisfies ψ_n if for every position j , if $w[j] = \#$, then either $w[j+1, j+n+1] \notin (0|1)^n$ or there is $k > j$ such that $w[k] = \$$ and $w[j+1, j+n+1] \neq w[k+1, k+n+1]$, for the minimal $k > j$ with $w[k] = \$$. Thus, as required, a word w satisfies $!\psi_n$ if there is a position j with $w[j] = \#, w[j+1, j+n+1] \in (0|1)^n$, and either there is no position $k > j$ with $w[k] = \$$, or $w[j+1, j+n+1] = w[k+1, k+n+1]$ for the minimal position $k > j$ with $w[k] = \$$. \square

Theorem 5.2 implies that our algorithm, which involves a translation of TRIGGER LOGIC formulas to DBWs, may indeed require a doubly-exponential time. In Theorem 5.3 below we show that one can not do better, as the synthesis problem is 2EXPTIME-hard. Thus, our algorithm is optimal and the synthesis problem for TRIGGER LOGIC is 2EXPTIME-complete.

Theorem 5.3. *The synthesis problem for TRIGGER LOGIC formulas is 2EXPTIME-hard.*

Proof. As in the 2EXPTIME-hardness for CLT* satisfiability [VS85], we do a reduction from the problem whether an alternating exponential-space Turing machine T accepts the empty tape. That is, given such a machine T and a number n in unary, we construct a trigger formula ψ such that T accepts the empty tape using space 2^n iff ψ is realizable. Let $T = \langle \Gamma, Q_u, Q_e, \rightarrow, q_0, q_{acc} \rangle$, where Γ is the tape alphabet, the sets Q_u and Q_e of states are disjoint, and contain the universal and the existential states, respectively, q_0 is the initial state, and q_{acc} is the accepting state. We denote the union $Q_u \cup Q_e$ by Q . Our model of alternation prescribes that the transition relation $\rightarrow \subseteq Q \times \Gamma \times Q \times \Gamma \times \{L, R\}$ has branching degree two, $q_0 \in Q_e$, and the machine T alternates between existential and universal set. When a universal or an existential state of T branches into two states, we distinguish between the left and the right branches. Accordingly, we use $(q, \sigma) \rightarrow \langle (q_l, b_l, \Delta_l), (q_r, b_r, \Delta_r) \rangle$ to indicate that when T is in state $q \in Q_u \cup Q_e$ reading a symbol σ , it branches to the left with (q_l, b_l, Δ_l) and to the right with (q_r, b_r, Δ_r) . (Note that the directions left and right here have nothing to do with the movement direction of the head; these are determined by Δ_l and Δ_r .)

For a configuration c of T , let $succ_l(c)$ and $succ_r(c)$ be the successors of c when applying to it the left and right choices in \rightarrow , respectively. Given an input w , a computation tree of T on w is a tree in which each node corresponds to a configuration of T . The root of the tree corresponds to the initial configuration. A node that corresponds to a universal configuration c has two successors, corresponding to $succ_l(c)$ and $succ_r(c)$. A node that corresponds to an existential configuration c has a single successor, corresponding to either $succ_l(c)$ or $succ_r(c)$. The tree is an accepting computation tree if all its branches eventually reach an accepting configuration – one in which the state is q_{acc} . We assume that once a computation reaches an accepting configuration it stays in q_{acc} forever.

We encode a configuration of T by a word $\gamma_1 \gamma_2 \dots (q, \gamma_i) \dots \gamma_{2^n}$. That is, all the letters in the configuration are in Γ , except for one letter in $Q \times \Gamma$. The meaning of such a configuration is that the j 's cell of T , for $1 \leq j \leq 2^n$, is labeled γ_j , the reading head points on cell i , and T is in state q . For example, the initial configuration of T on the empty tape is $@_1, (q_0, \#), \# \dots \#, @_2$, where $\#$ stands for the empty cell, and $@_1$ and $@_2$ are special tape-end symbols. We can now encode a computation of T by a sequence of configurations.

Let $\Sigma = \Gamma \cup (Q \times \Gamma)$. We can encode letters in Σ by a set $AP(T) = \{p_1, \dots, p_m, p'_1, \dots, p'_m\}$ (with $m = \lceil \log |\Sigma| \rceil$) of atomic propositions. The propositions p'_1, \dots, p'_m are auxiliary; their roles is made clear shortly. We define our formulas over the set $AP = AP(T) \cup \{v_1, \dots, v_n, v'_1, \dots, v'_n\} \cup \{real, left_{in}, left_{out}, e\}$ of atomic propositions. The propositions v_1, \dots, v_n encode the locations of the cells in a configuration. The propositions v'_1, \dots, v'_n help in increasing the value encoded by v_1, \dots, v_n properly. The task of the last four atoms is explained shortly.

The set AP of propositions is divided into input and output propositions. The input propositions are *real* and *left_{in}*. All other propositions are output propositions. With two input propositions, a strategy can be viewed as a 4-ary tree. Recall that the branching degree of T is 2. Why then do we need a 4-ary tree? Intuitively, the strategy should describe a legal and accepting computation tree of T in a “real” 2-ary tree embodied in the strategy tree. This real 2-ary tree is the one in which the input proposition *real* always holds. Branches in which *real* is eventually false do not correspond to computations of T and have a different role. Within the real tree, the input proposition *left_{in}* is used in order to distinguish between the left and right successors of a configurations.

The propositions v_1, \dots, v_n encode the location of a cell in a configuration of T , with v_1 being the most significant bit. Since T is an exponential-space Turing machine, this location is a number between 0 and $2^n - 1$. To ensure that v_1, \dots, v_n act as an n -bit counters we need the following formulas:

1. The counter starts at 0.
 - $\mathbf{true} \mapsto \&\&_{i=1}^n !v_i$
2. The counter is increased properly. For this we use v'_1, \dots, v'_n as carry bits.
 - $\mathbf{true}[+] \mapsto v'_n$
 - $(\mathbf{true}[*]; (v_i \&\& v'_i)) \mapsto \mathbf{true}; (!v_i \&\& v'_{i-1})$, for $i = 2, \dots, n$
 - $(\mathbf{true}[*]; ((v_i \&\& (!v'_i)) \mid ((!v_i) \&\& v'_i))) \mapsto \mathbf{true}; (v_i \&\& (!v'_{i-1}))$, for $i = 2, \dots, n$
 - $(\mathbf{true}[*]; ((!v_i) \&\& (!v'_i))) \mapsto \mathbf{true}; ((!v_i) \&\& (!v'_{i-1}))$, for $i = 2, \dots, n$
 - $(\mathbf{true}[*]; ((v_1 \&\& v'_1) \mid ((!v_1) \&\& (!v'_1)))) \mapsto \mathbf{true}; !v_1$
 - $(\mathbf{true}[*]; ((v_1 \&\& (!v'_1)) \mid ((!v_1) \&\& v'_1))) \mapsto \mathbf{true}; v_1$

Let $\sigma_1 \dots \sigma_{2^n}, \sigma'_1 \dots \sigma'_{2^n}$ be two successive configurations of T . For each triple $\langle \sigma_{i-1}, \sigma_i, \sigma_{i+1} \rangle$ with $1 < i < 2^n$, we know for each transition relation of T , what σ'_i should be. Let $next(\langle \sigma_{i-1}, \sigma_i, \sigma_{i+1} \rangle)$ denote our expectation for σ'_i . I.e. ⁶

- $next(\langle \gamma_{i-1}, \gamma_i, \gamma_{i+1} \rangle) = \gamma_i$.
- $next(\langle (q, \gamma_{i-1}), \gamma_i, \gamma_{i+1} \rangle) = \begin{cases} \gamma_i & \text{If } (q, \gamma_{i-1}) \rightarrow (q', \gamma'_{i-1}, L). \\ (q', \gamma_i) & \text{If } (q, \gamma_{i-1}) \rightarrow (q', \gamma'_{i-1}, R). \end{cases}$
- $next(\langle \gamma_{i-1}, (q, \gamma_i), \gamma_{i+1} \rangle) = \gamma'_i$ where $(q, \gamma_i) \rightarrow (q', \gamma'_i, \Delta)$.
- $next(\langle \gamma_{i-1}, \gamma_i, (q, \gamma_{i+1}) \rangle) = \begin{cases} \gamma_i & \text{If } (q, \gamma_{i+1}) \rightarrow (q', \gamma'_{i+1}, R). \\ (q', \gamma_i) & \text{If } (q, \gamma_{i+1}) \rightarrow (q', \gamma'_{i+1}, L). \end{cases}$

Since we have two transitions relations, we actually obtain two functions, $next_l$ and $next_r$.

Consistency with *next* gives us a necessary condition for a path in the computation tree to encode a legal computation. In addition, the computation should start with the initial configuration and reach an accepting state. It is easy to specify the requirements about the initial and accepting configurations. For

⁶Special handling of end cases is needed, when the head of T read the left or right end markers. For simplicity, we ignore this technicality here.

a letter $\sigma \in \Sigma$, let $\eta(\sigma)$ be the propositional formula over AP in which p_1, \dots, p_n encode σ . That is, $\eta(\sigma)$ holds in a node iff the truth value of the propositions $p_1 \dots, p_m$ in that node encodes the symbol σ . Similarly, $\eta'(\sigma)$ is the propositional formula over AP in which p'_1, \dots, p'_n encode σ . Thus, to say that the first configuration correspond to the empty word we use the following formulas, where *ones* abbreviates $\bigwedge_{i=1}^n v_i$, and # denotes the empty symbol:

- $\mathbf{true} \mapsto \eta(@_1); \eta(\langle q_0, \# \rangle)$
- $(\mathbf{true}; \mathbf{true}; (!ones)[+]) \mapsto \eta(\#)$
- $((!ones)[+]; ones) \mapsto \mathbf{true}; \eta(@_2)$

We come back to the acceptance condition shortly.

The propositions p'_1, \dots, p'_m capture the symbol encoded in the previous cell, and special symbols at initial cells. We use the following formula, where *zeros* abbreviates $\bigwedge_{i=1}^n (!v_i)$.

- $(\mathbf{true}[*]; zero) \mapsto \eta'(@_2)$
- $(\mathbf{true}[*]; ((!ones) \& \& p_j)) \mapsto (\mathbf{true}; p'_j)$
- $(\mathbf{true}[*]; ((!ones) \& \& (!p_j))) \mapsto (\mathbf{true}; (!p'_j))$

The output proposition e marks existential configurations. Recall that computations of T start in existential configurations and alternate between universal and existential configurations. The value of e is maintained throughout the configuration. This is expressed using the following formulas:

- $\mathbf{true} \mapsto e$
- $(\mathbf{true}[*]; ((!ones) \& \& e)) \mapsto (\mathbf{true}; e)$
- $(\mathbf{true}[*]; ((!ones) \& \& (!e))) \mapsto (\mathbf{true}; (!e))$
- $(\mathbf{true}[*]; (ones \& \& e)) \mapsto (\mathbf{true}; (!e))$
- $(\mathbf{true}[*]; (ones \& \& (!e))) \mapsto (\mathbf{true}; e)$

The output proposition $left_{out}$ marks configurations that are left successors. The value of $left_{out}$ is determined according to the value of $left_{in}$ at the end of the previous configuration, and is maintained throughout the configuration, where it is used in order to decide whether the configuration should be consistent with $next_l$ or with $next_r$. The following formulas ensure that the value is indeed maintained and that universal configurations are followed by both left and right configurations. On the other hand, for the successors of existential configurations, the strategy has no restrictions on the value of $left_{out}$, and can choose the same value for the two successors.

- $(\mathbf{true}[*]; ((!ones) \& \& left_{out})) \mapsto (\mathbf{true}; left_{out})$
- $(\mathbf{true}[*]; ((!ones) \& \& (!left_{out}))) \mapsto (\mathbf{true}; (!left_{out}))$
- $(\mathbf{true}[*]; (ones \& \& (!e) \& \& (!left_{in}))) \mapsto (\mathbf{true}; left_{out})$
- $(\mathbf{true}[*]; (ones \& \& (!e) \& \& left_{in})) \mapsto (\mathbf{true}; (!left_{out}))$

The difficult part in the reduction is in guaranteeing that the sequence of configurations is indeed consistent with $next_l$ and $next_r$. To enforce this, we have to relate σ_{i-1}, σ_i , and σ_{i+1} with σ'_i for each i in every two successive configurations $\sigma_1 \dots \sigma_{2^n}, \sigma'_1 \dots \sigma'_{2^n}$. One natural way to do so is by a conjunction of formulas like “whenever we meet a cell at location $i - 1$ and the labeling of the next three cells forms the triple $\langle \sigma_{i-1}, \sigma_i, \sigma_{i+1} \rangle$, then the next time we meet a cell at location i , this cell is labeled $next(\langle \sigma_{i-1}, \sigma_i, \sigma_{i+1} \rangle)$ ”. The problem is that, as i can take a value from 0 to $2^n - 1$, there are exponentially many such conjuncts. This is where the non-real part of the tree is helpful [VS85].

Recall that the input proposition *real* is used to labeled the “real” part of the strategy tree – the one that corresponds to the computation tree of T . Once we branch according to $!real$, we move to the auxiliary part of the tree. Consider now an arbitrary trace, either it is a real trace, on which *real* is always true, or it reaches the auxiliary part of the tree, where *real* is false. We refer to the latter trace as an *auxiliary trace*. The point at which *real* is true for the last time is the *end* of this auxiliary trace.

Consider a point x on an auxiliary trace that is followed by the end point y . There are the following possibilities:

1. *ones* holds less than or more than once between x and y , which means that x and y do not belong to successive configurations.
2. *ones* holds once between x and y , which means that they belong to successive configurations, but the assignment to p_1, \dots, p_n at x and y disagree, which means that they are not corresponding cells.
3. *ones* holds once between x and y , which means that they belong to successive configurations, and the assignments to p_1, \dots, p_n at x and y agree, which means that they are corresponding cells.

Accordingly, in order to ensure correct succession of configurations of T , we use the formula

- $real[+] \mapsto \psi$,

where ψ is a union of the following regular expressions:

- $(\mathbf{true}[+]; \bigvee_{\gamma \in \Gamma} \eta(\langle s_a, \gamma \rangle))$: the trace reaches an accepting configuration;
- $(\mathbf{!ones} \&\& \mathbf{real}[+]; \mathbf{ones})$: the points x and y belong to same configuration;
- $real[*]; \mathbf{ones} \&\& \mathbf{real}; real[+]; \mathbf{ones} \&\& \mathbf{real}$: the points belong to non-successive configurations;
- $v_i \&\& \mathbf{real}; real[+]; (\mathbf{!}v_i) \&\& \mathbf{real}; \mathbf{!real}$, for $i = 1, \dots, n$: the points do not agree on the value of the i -th bit in the encoding of their address and therefore they have different cell locations;
- $\eta'(\sigma_1) \&\& \eta(\sigma_2) \&\& \mathbf{real}; \eta(\sigma_3) \&\& \mathbf{real}; real[+]; \mathbf{left}_{out} \&\& \mathbf{next}_l(\sigma_1, \sigma_2, \sigma_3) \&\& \mathbf{real}; \mathbf{!real}$, for $\sigma_1, \sigma_2, \sigma_3 \in \Sigma$: the points x and y are in the same cell of a configuration and its left successor, and \mathbf{next}_l is respected. Note that the propositions p'_i are used in order to refer to the cell before x .
- $(\eta'(\sigma_1) \&\& \eta(\sigma_2)) \&\& \mathbf{real}; \eta(\sigma_3) \&\& \mathbf{real}; real[+]; (\mathbf{!} \mathbf{left}_{out}) \&\& \mathbf{next}_r(\sigma_1, \sigma_2, \sigma_3) \&\& \mathbf{real}; \mathbf{!real}$, for $\sigma_1, \sigma_2, \sigma_3 \in \Sigma$: the points x and y are in the same cell of a configuration and its right successor, and \mathbf{next}_r is respected.

□

Note that the TRIGGER LOGIC formula constructed in the reduction is a conjunction of formulas of the form $r \mapsto e$. Thus, the problem is 2EXPTIME-hard already for this fragment of TRIGGER LOGIC.

6 Practice Issues

In Section 5.1, we proved that the synthesis problem for TRIGGER LOGIC can be solved in doubly-exponential time. This bound is no better on its face than the doubly-exponential time upper bound proved in [PR89a, KV05c] for LTL synthesis. A closer examination reveals, however, that the algorithms in [PR89a, KV05c] have time complexity of the form 4^{4^n} , while the algorithm described here has time complexity of the form 4^{2^n} . This, however, is not what we view as the main advantage of this algorithm. Rather, its main advantage is that it is significantly simpler. Unlike the algorithm in [PR89a], we need not apply Safra's determinization construction nor solving complex games. Unlike [KV05b], we need not use progress measures. Our algorithm is based solely on using the subset construction and solving the emptiness problem for Büchi tree automata.

In this section we show that our algorithm for TRIGGER LOGIC has additional appealing properties in practice.

A symbolic implementation Theorem 5.1 reduces the TRIGGER LOGIC synthesis problem to the nonemptiness problem of a DBT obtained by dualizing a DCW that is the result of applying the break-point construction of [MH84] to the NCW that corresponds to the negation of the TRIGGER LOGIC formula. In [MS08a, MS08b], the authors described a symbolic implementation of the break-point construction for word automata. For tree automata, the symbolic algorithm for the nonemptiness construction is not more difficult, as both word emptiness and tree emptiness for Büchi automata are based on nested-fixpoint algorithms [EL86, VW86], using a quadratic number of symbolic operations.

In more details, the state space of the DBT consists of sets of states, it can be encoded by Boolean variables, and the DBT's transitions can be encoded by relations on these variables and a primed version of them. The fixpoint solution for the nonemptiness problem of DBT (c.f., [VW86]) then yields a symbolic solution to the synthesis problem. Moreover, the BDDs that are generated by the symbolic decision procedure can be used to generate a symbolic witness strategy. The Boolean nature of BDDs then makes it very easy to go from this BDD to a sequential circuit for the strategy. It is known that a BDD can be viewed as an expression (in DAG form) that uses the “if then else” as a single ternary operator. Thus, a BDD can be viewed as a circuit built from if-then-else gates. More advantages of the symbolic approach are described in [HRS05]. As mentioned above, [HRS05] also suggests a symbolic solution for the LTL synthesis problem. However, the need to circumvent Safra's determinization causes the algorithm in [HRS05] to be complete only for a subset of LTL. Likewise, the need to implement the progress ranks of [KV05a] using a binary encoding challenges BDD-based implementations [TV07]. Our approach here circumvents both Safra's determinization and ranks, facilitating a symbolic implementation.

Incremental 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. Here, we describe how our synthesis algorithm can support *incremental* synthesis, where the temporal assertions are given one by one. We show how working with DBWs enables us, when we check the realizability of $\psi \& \& \psi'$, to use much of the work done in checking the realizability of ψ and ψ' in isolation.

Essentially, we show that when we construct and check the emptiness of the DBT to which realizability of $\psi \& \& \psi'$ is reduced, we can use much of the work done in the process of checking the emptiness of the two (much smaller) DBTs to which realizability of ψ and ψ' is reduced (in isolation). Let \mathcal{A} and \mathcal{A}' be the DBTs to which realizability of ψ and ψ' is reduced, respectively. Recall that \mathcal{A} and \mathcal{A}' are obtained from NCWs with state spaces Q and Q' . A non-incremental approach generates the DBT that corresponds to $\psi \& \& \psi'$. By Theorem 5.1, this results in a DBT \mathcal{U} with state space $3^{Q \cup Q'}$. On the other hand, the state spaces of \mathcal{A} and \mathcal{A}' are much smaller, and are 3^Q and $3^{Q'}$, respectively.

Let us examine the structure of the state space of \mathcal{U} more carefully. Each of its states can be viewed as a pair $\langle S \cup S', O \cup O' \rangle$, for $O \subseteq S \subseteq Q$ and $O' \subseteq S' \subseteq Q'$. The state corresponds to the states $\langle S, O \rangle$ of \mathcal{A} and $\langle S', O' \rangle$ of \mathcal{A}' . Clearly, if one of these states is empty (that is, if the automaton accept no tree starting from these states), then so is $\langle S \cup S', O \cup O' \rangle$. Thus, an incremental algorithm can start by marking all such states as empty and continue the emptiness check only with the (hopefully much smaller) state space.

(Note that this idea does not apply to disjunctions. Suppose that neither ψ nor ψ' is realizable, and we want to check if $\psi \parallel \psi'$ is realizable. It is not clear how to leverage realizability checking of ψ and ψ' , when we check realizability of $\psi \parallel \psi'$.)

Adding assumptions The method described above cannot be applied for formulas of the form $\psi' \rightarrow \psi$, with ψ' and ψ formulas in TRIGGER LOGIC. Note that since TRIGGER LOGIC is not closed under negation, the specification $\psi' \rightarrow \psi$ is not a TRIGGER LOGIC formula. Still, such an implication arises naturally when we want to synthesize ψ with respect to environments satisfying ψ' . To handle such specifications, we apply the automata-theoretic constructions of Section 5.1 to both ψ' and ψ obtaining DBT $\mathcal{A}_i^{\psi'}$ and \mathcal{A}_i^{ψ} , with acceptance conditions α' and α . We now take the product of $\mathcal{A}_i^{\psi'}$ and \mathcal{A}_i^{ψ} , and use as acceptance condition the Streett pair $\langle \alpha', \alpha \rangle$. A symbolic algorithm for Streett tree automata is described in [KV98b]. For Streett(1) condition, that is, a single pair Streett condition, the algorithm requires a cubic number of symbolic operations.

References

- [AFF⁺02] R. Armoni, L. Fix, A. Flaisher, R. Gerth, B. Ginsburg, T. Kanza, A. Landver, S. Mador-Haim, E. Singerman, A. Tiemeyer, M.Y. Vardi, and Y. Zbar. The ForSpec temporal logic: A new temporal property-specification logic. In *Proc. 8th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, volume 2280 of *Lecture Notes in Computer Science*, pages 196–211. Springer, 2002.
- [ALW89] M. Abadi, L. Lamport, and P. Wolper. Realizable and unrealizable concurrent program specifications. In *Proc. 25th Int. Colloq. on Automata, Languages, and Programming*, volume 372 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 1989.
- [AMPS98] E. Asarin, O. Maler, A. Pnueli, and J. Sifakis. Controller synthesis for timed automata. In *IFAC Symposium on System Structure and Control*, pages 469–474. Elsevier, 1998.
- [AT04] R. Alur and S. La Torre. Deterministic generators and games for ltl fragments. *ACM Transactions on Computational Logic*, 5(1):1–25, 2004.
- [ATW05] C.S. Althoff, W. Thomas, and N. Wallmeier. Observations on determinization of Büchi automata. In *Proc. 10th Int. Conf. on the Implementation and Application of Automata*, volume 3845 of *Lecture Notes in Computer Science*, pages 262–272. Springer, 2005.
- [BBE⁺01] I. Beer, S. Ben-David, C. Eisner, D. Fisman, A. Gringauze, and Y. Rodeh. The temporal logic Sugar. In *Proc 13th Int. Conf. on Computer Aided Verification*, volume 2102 of *Lecture Notes in Computer Science*, pages 363–367. Springer, 2001.
- [BFG⁺05] D. Bustan, A. Flaisher, O. Grumberg, O. Kupferman, and M.Y. Vardi. Regular vacuity. In *Proc. 13th Conf. on Correct Hardware Design and Verification Methods*, volume 3725 of *Lecture Notes in Computer Science*, pages 191–206. Springer, 2005.
- [BGJ⁺07] R. Bloem, S. Galler, B. Jobstmann, N. Piterman, A. Pnueli, and M. Weiglhofer. Automatic hardware synthesis from specifications: a case study. In *Proc. Conference on Design, Automation and Test in Europe*, pages 1188–1193. ACM, 2007.
- [BL69] J.R. Büchi and L.H. Landweber. Solving sequential conditions by finite-state strategies. *Trans. AMS*, 138:295–311, 1969.
- [CGP99] E.M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.
- [Cho74] Y. Choueka. Theories of automata on ω -tapes: A simplified approach. *Journal of Computer and Systems Science*, 8:117–141, 1974.
- [Chu63] A. Church. Logic, arithmetics, and automata. In *Proc. Int. Congress of Mathematicians, 1962*, pages 23–35. Institut Mittag-Leffler, 1963.
- [CKS81] A.K. Chandra, D.C. Kozen, and L.J. Stockmeyer. Alternation. *Journal of the Association for Computing Machinery*, 28(1):114–133, 1981.
- [Dil89] D.L. Dill. *Trace theory for automatic hierarchical verification of speed independent circuits*. MIT Press, 1989.
- [DR09] L. Doyen and J.-F. Raskin. Antichains for the automata-based approach to model-checking. *Logical Methods in Computer Science*, 5(1), 2009.
- [EC82] E.A. Emerson and E.M. Clarke. Using branching time logic to synthesize synchronization skeletons. *Science of Computer Programming*, 2:241–266, 1982.
- [EF06] C. Eisner and D. Fisman. *A Practical Introduction to PSL*. Springer, 2006.
- [EKM98] J. Elgaard, N. Klarlund, and A. Möller. Mona 1.x: new techniques for WS1S and WS2S. In *Proc 10th Int. Conf. on Computer Aided Verification*, volume 1427 of *Lecture Notes in Computer Science*, pages 516–520. Springer, 1998.
- [EL86] E.A. Emerson and C.-L. Lei. Efficient model checking in fragments of the propositional μ -calculus. In *Proc. 1st IEEE Symp. on Logic in Computer Science*, pages 267–278, 1986.
- [HRS05] A. Harding, M. Ryan, and P. Schobbens. A new algorithm for strategy synthesis in LTL games. In *Proc. 11th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, volume 3440 of *Lecture Notes in Computer Science*, pages 477–492. Springer, 2005.
- [HU79] J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation*.

- Addison-Wesley, 1979.
- [Jur00] M. Jurdzinski. Small progress measures for solving parity games. In *Proc. 17th Symp. on Theoretical Aspects of Computer Science*, volume 1770 of *Lecture Notes in Computer Science*, pages 290–301. Springer, 2000.
- [Kla98] N. Klarlund. Mona & Fido: The logic-automaton connection in practice. In *Proc. 6th Annual Conf. of the European Association for Computer Science Logic*, *Lecture Notes in Computer Science*, 1998.
- [KPV06] O. Kupferman, N. Piterman, and M.Y. Vardi. Safrless compositional synthesis. In *Proc 18th Int. Conf. on Computer Aided Verification*, volume 4144 of *Lecture Notes in Computer Science*, pages 31–44. Springer, 2006.
- [KV98a] O. Kupferman and M.Y. Vardi. Freedom, weakness, and determinism: from linear-time to branching-time. In *Proc. 13th IEEE Symp. on Logic in Computer Science*, pages 81–92, 1998.
- [KV98b] O. Kupferman and M.Y. Vardi. Weak alternating automata and tree automata emptiness. In *Proc. 30th ACM Symp. on Theory of Computing*, pages 224–233, 1998.
- [KV00] O. Kupferman and M.Y. Vardi. Synthesis with incomplete information. In *Advances in Temporal Logic*, pages 109–127. Kluwer Academic Publishers, 2000.
- [KV05a] O. Kupferman and M.Y. Vardi. Complementation constructions for nondeterministic automata on infinite words. In *Proc. 11th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, volume 3440 of *Lecture Notes in Computer Science*, pages 206–221. Springer, 2005.
- [KV05b] O. Kupferman and M.Y. Vardi. From linear time to branching time. *ACM Transactions on Computational Logic*, 6(2):273–294, 2005.
- [KV05c] O. Kupferman and M.Y. Vardi. Safrless decision procedures. In *Proc. 46th IEEE Symp. on Foundations of Computer Science*, pages 531–540, 2005.
- [LP85] O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *Proc. 12th ACM Symp. on Principles of Programming Languages*, pages 97–107, 1985.
- [MH84] S. Miyano and T. Hayashi. Alternating finite automata on ω -words. *Theoretical Computer Science*, 32:321–330, 1984.
- [MS08a] A. Morgenstern and K. Schneider. From ltl to symbolically represented deterministic automata. In *Proc. 9th Int. Conf. on Verification, Model Checking, and Abstract Interpretation*, volume 4905 of *Lecture Notes in Computer Science*, pages 279–293, 2008.
- [MS08b] A. Morgenstern and K. Schneider. Generating deterministic ω -automata for most LTL formulas by the breakpoint construction. In *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen*, Freiburg, Germany, 2008.
- [MW80] Z. Manna and R. Waldinger. A deductive approach to program synthesis. *ACM Transactions on Programming Languages and Systems*, 2(1):90–121, 1980.
- [MW84] Z. Manna and P. Wolper. Synthesis of communicating processes from temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 6(1):68–93, 1984.
- [Pit07] N. Piterman. From nondeterministic Büchi and Streett automata to deterministic parity automata. *Logical Methods in Computer Science*, 3(3):5, 2007.
- [Pnu77] A. Pnueli. The temporal logic of programs. In *Proc. 18th IEEE Symp. on Foundations of Computer Science*, pages 46–57, 1977.
- [PPS06] N. Piterman, A. Pnueli, and Y. Saar. Synthesis of reactive(1) designs. In *Proc. 7th Int. Conf. on Verification, Model Checking, and Abstract Interpretation*, volume 3855 of *Lecture Notes in Computer Science*, pages 364–380. Springer, 2006.
- [PR89a] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Proc. 16th ACM Symp. on Principles of Programming Languages*, pages 179–190, 1989.
- [PR89b] A. Pnueli and R. Rosner. On the synthesis of an asynchronous reactive module. In *Proc. 16th Int. Colloq. on Automata, Languages, and Programming*, volume 372 of *Lecture Notes in Computer Science*, pages 652–671. Springer, 1989.
- [Rab72] M.O. Rabin. Automata on infinite objects and Church’s problem. *Amer. Mathematical Society*, 1972.

- [Ros92] R. Rosner. *Modular Synthesis of Reactive Systems*. PhD thesis, Weizmann Institute of Science, 1992.
- [RS59] M.O. Rabin and D. Scott. Finite automata and their decision problems. *IBM Journal of Research and Development*, 3:115–125, 1959.
- [SDC01] K. Shimizu, D.L. Dill, and C.-T. Chou. A specification methodology by a collection of compact properties as applied to the intel itanium processor bus protocol. In *Proc. 11th Conf. on Correct Hardware Design and Verification Methods*, volume 2144 of *Lecture Notes in Computer Science*, pages 340–354. Springer, 2001.
- [THB95] S. Tasiran, R. Hojati, and R.K. Brayton. Language containment using non-deterministic omega-automata. In *Proc. 8th Conf. on Correct Hardware Design and Verification Methods*, volume 987 of *Lecture Notes in Computer Science*, pages 261–277. Springer, 1995.
- [TV07] D. Tabakov and M.Y. Vardi. Model checking Büchi specifications. In *1st International Conference on Language and Automata Theory and Application Principles of Programming Languages*, 2007.
- [Var95] M.Y. Vardi. An automata-theoretic approach to fair realizability and synthesis. In *Proc 7th Int. Conf. on Computer Aided Verification*, volume 939 of *Lecture Notes in Computer Science*, pages 267–292. Springer, 1995.
- [VR05] S. Vijayaraghavan and M. Ramanathan. *A Practical Guide for SystemVerilog Assertions*. springer, 2005.
- [VS85] M.Y. Vardi and L. Stockmeyer. Improved upper and lower bounds for modal logics of programs. In *Proc. 17th ACM Symp. on Theory of Computing*, pages 240–251, 1985.
- [VW86] M.Y. Vardi and P. Wolper. Automata-theoretic techniques for modal logics of programs. *Journal of Computer and Systems Science*, 32(2):182–221, 1986.
- [VW94] M.Y. Vardi and P. Wolper. Reasoning about infinite computations. *Information and Computation*, 115(1):1–37, 1994.
- [WTD91] H. Wong-Toi and D.L. Dill. Synthesizing processes and schedulers from temporal specifications. In E.M. Clarke and R.P. Kurshan, editors, *Proc 2nd Int. Conf. on Computer Aided Verification*, volume 3 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 177–186. AMS, 1991.