Synthesizing Reactive Programs

P. Madhusudan

University of Illinois at Urbana-Champaign, USA
madhu@cs.illinois.edu

Abstract

Current theoretical solutions to the classical Church's synthesis problem are focussed on synthesizing *transition systems* and not *programs*. Programs are compact and often the true aim in many synthesis problems, while the transition systems that correspond to them are often large and not very useful as synthesized artefacts. Consequently, current practical techniques first synthesize a transition system, and then extract a more compact representation from it. We reformulate the synthesis of reactive systems directly in terms of program synthesis, and develop a theory to show that the problem of synthesizing programs over a fixed set of Boolean variables in a simple imperative programming language is decidable for regular ω-specifications. We also present results for synthesizing programs with recursion against both regular specifications as well as visibly-pushdown language specifications. Finally, we show applications to program repair, and conclude with open problems in synthesizing distributed programs.

1998 ACM Subject Classification I.2.2 Automatic Programming; F.3.1 Specifying and Verifying and Reasoning about Programs; F.4.3 Formal Languages

Keywords and phrases Program synthesis, Boolean programs, Automata theory, Temporal logics

Digital Object Identifier 10.4230/LIPIcs.xxx.yyy.p

1 Introduction

The synthesis problem for reactive systems is a classical problem in computer science, and stems from a problem posed by Church in 1957 on synthesizing digital circuits from specifications written in a restricted logic of arithmetic [6]. This problem was solved first by Büchi and Landweber in 1969 [5] (see [25] for an account of the history of this problem). The 70s saw the emergence of the elegant theory of *automata on infinite trees* by Rabin [19], which has now been well-studied and honed into a beautiful theory that underlies many of the decidability results in logic and automata theory [24, 8]. Coupled with the temporal-logic to automata connection on words [16, 27, 20], tree-automata theory gives the most elegant solution to Church's problem: compile the specification into a *deterministic* parity automaton on infinite words [20], using this build a parity tree-automaton that accepts the trees that correspond to *strategies* for the system to generate outputs for inputs so that all paths in the tree are accepted by the specification automaton, and, finally, check the emptiness of the tree automaton and build a finite-state transition system from a regular tree that's accepted by this automaton [17].

The synthesis problem has received a lot of attention in recent years, both in theory as well as in practice. Theoretical approaches include extensions to branching time specifications [11, 13], the very non-trivial problem of synthesizing *distributed systems* [18, 14, 13], and synthesis with incomplete information where the environment and system may not have perfect information about the state of each other [10].

Most of the current theoretical techniques in synthesis are geared towards designing *transition systems*. In other words, the algorithms for synthesis in the end output transition systems that meet the specification. While transition systems are appropriate for defining semantics of systems, systems are seldom designed by explicitly describing their transition

Conference title on which this volume is based on.
Editors: Billy Editor, Bill Editors; pp. 1–15

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

systems. Systems are instead designed using high-level and succinct representations such as *programs*.

The extensive literature on synthesizing transition systems does not help in building directly the compact programs we seek. Current practical techniques build a transition system (or an automaton that depicts several different strategies for winning), and implement it in a symbolic manner such as a program. Several such approaches currently exist; for instance, the approach in [15] builds automata from which a symbolic algorithm using memory variables is extracted and the work by Bloem et al [4] reports on practical ways of synthesizing PSL circuit code from symbolic BDD representations of the solution.

Consider, for instance, the problem of synthesizing a data-structure to maintain subsets of a fixed set of n elements, where the environment is allowed to maintain the set by adding or removing elements from it, and querying the set for membership. It is easy to see that a program for implementing it can be written that uses n Boolean variables, one for each element, that tracks whether that element belongs to the set or not. Furthermore, a short while-program (of length about $O(n\ log\ n)$) can be written. On the other hand, a transition system realizing the specification will necessarily be exponential in n. For $n = 50$, a transition system with 2^{50} states is not very useful as a synthesized artefact, while a program that's about 150 lines of code is. One can, of course, construct a particular transition system and then try to synthesize a program from it (as in the works cited above), but these are not guaranteed to yield small programs.

Another drawback (and a subtle one) in current synthesis techniques is that the systems that are synthesized depend on *how* specifications are written, rather than only its semantics. For instance, assume φ_1 and φ_2 are two specifications that are syntactically different but semantically equivalent. Then the current synthesis algorithms based on tree automata will produce *different* sets of (finite) transition systems for these specifications. The reason is that for a given specification, current synthesis algorithms build an automaton that accepts unfoldings of transitions systems that satisfy the specification. Though these automata for φ_1 and φ_2 accept precisely the same set of trees, the automata themselves are different, as they depend on the *syntax* of the specifications (i.e., on φ_1 and φ_2). Consequently, the non-emptiness algorithm for these automata can synthesize different transition systems for the two semantically-equivalent specifications. Intuitively, the synthesized finite-state transition system not only encodes a correct algorithm, but also an elaborate *proof* as to why it meets the specification, and this proof grows with the way the specification is formulated. In particular, the more complex the same specification is written, the more complex will be the synthesized transition system!

There is some work in the literature (see [21]) that addresses the above problem and can synthesize transition systems (but not programs) that are within a bound, independent of the specification. However, we do not know of any work on specification-syntax agnostic synthesis that works in the unbounded setting.

The main contribution of this paper is a theory of synthesis where imperative programs (written in a particular syntax, and over a fixed set of Boolean variables) are first-class objects and where the synthesis algorithms directly synthesize programs that meet reactive regular ω-specifications. We lay out this theory of synthesizing imperative programs, showing two main results: (a) that imperative program synthesis is decidable against regular specifications, and (b) that synthesis of imperative programs with recursive functions is decidable against both regular specifications as well as visibly-pushdown specifications.

The technical results are automata-theoretic, are geared towards synthesizing programs directly, and use two-way alternating ω-automata working on *finite* trees that represent

programs. In particular, the synthesis algorithms in this paper build tree automata on finite trees that accept *all* programs satisfying a specification, and hence do not depend on the way the specification is written. We can therefore pull out a program that accords to our needs (for example, we can pull out the smallest program satisfying the specification, for some notion of length of a program).

There is a flurry of research in the programming languages community in the last few years on *program synthesis* [9, 23, 22]. In general, these algorithms are aimed at practical synthesis approaches towards solving standard algorithmic problems (such as sorting, Strassen's multiplication algorithm, Excel scripts, etc.). In these papers, the prevailing theme is to *fix* a template for the program, and use SAT and SMT solvers to find a program matching the template and simultaneously a proof (which also comes with a template) that proves the program correct. In these algorithms, the search space of programs for synthesis is *finite*, and the focus is on efficiency, programmability, and usability. Hence a theory of synthesizing programs, albeit finite-state programs, seems worthy of study. In this context, this paper provides a sound and complete procedure for synthesizing Boolean programs of arbitrary length that satisfy a specification.

The paper is structured as follows: Section 2 defines imperative programs without recursion and regular specifications, while Section 3 introduces the automata theory on trees that we will use. Section 4 lays down the results for synthesizing non-recursive programs, and Section 5 shows how to extend this to synthesize recursive programs against regular as well as visibly pushdown specifications. Section 6 concludes with a discussion of applications of our results to program repair, and open problems in distributed program synthesis.

2 Programs and regular specifications

We define in this section the class of imperative programs that we work with, and also define the class of regular specifications against which we synthesize programs.

Let us fix two numbers $N_I, N_O \in \mathbb{N}$. We will design programs that in every round take N_I bits as input and output N_O bits.

Programs are parameterized over a finite set of Boolean variables B that it uses (naturally, we assume $|B| \geq max\{N_I, N_O\}$). The class of programs over B is given as follows (where b, b_i range over variables in B, and $\vec{b}$ stands for a vector of variables in B):

$$
\begin{array}{lcl}
\langle stmt \rangle & ::= & \langle stmt \rangle; \langle stmt \rangle \mid \textbf{skip} \mid b := \langle expr \rangle \mid \textbf{input } \vec{b} \mid \textbf{output } \vec{b} \\
 & & \textbf{if } (\langle expr \rangle) \textbf{ then } \langle stmt \rangle \textbf{ else } \langle stmt \rangle \mid \textbf{while } (\langle expr \rangle) \{\langle stmt \rangle\} \\
\langle expr \rangle & ::= & b \mid \textbf{tt} \mid \textbf{ff} \mid \langle expr \rangle \vee \langle expr \rangle \mid \neg \langle expr \rangle
\end{array}
$$

We assume that all input and output statements have tuples of width N_I and N_O, respectively.

The semantics of a reactive program is the natural one. The "**input** $\vec{b}$" statement takes (reactively) an input in $\{0,1\}^{N_I}$ from the environment and stores it in the variables $\vec{b}$, while "**output** $\vec{b}$" outputs the values of the variables in $\vec{b}$. The program can execute any number of internal steps between an input and an output statement (though synthesized programs will have the property that the program eventually does produce an output). During this internal computation, the program can manipulate its variables using assignments, conditionals, and iteration. Note that programs are, of course, finite in length, though they can (and typically will) interact with their environment reactively and infinitely often.

Representing programs using finite trees: We will represent programs as finite trees. Intuitively, the *bracketing* of blocks of code (as defined by the statements under conditionals

and while-loops) can be nested arbitrarily, and hence we need trees to capture roughly the parse-trees of programs according to the grammar given above.

For brevity, we will use binary trees (with every node having zero, one, or two children) and represent them using terms. A term $f(t_1, t_2)$ corresponds to a tree with f as the label of the root, and with the trees corresponding to t_1 and t_2 as the left and right subtrees of the root; a unary term $g(t)$ corresponds to a tree with g as the label of the root, and where the root has only one child (say the left child), and the subtree at this child is isomorphic to the tree associated with t.

The tree associated with a program P is $(root, tree(p))$, where *tree* is inductively defined as follows:

$$
\begin{aligned}
tree(b) &= b \\
tree(\mathbf{tt}) &= \mathbf{tt} \\
tree(\mathbf{ff}) &= \mathbf{ff} \\
tree(\varphi_1 \vee \varphi_2) &= \vee\ (tree(\varphi_1),\ tree(\varphi_2)) \\
tree(\neg\varphi) &= \neg\ (tree(\varphi_1))
\end{aligned}
\qquad
\begin{aligned}
tree(s; s') &= ;\ (tree(s),\ tree(s')) \\
tree(\mathbf{skip}) &= \mathbf{skip} \\
tree(\mathbf{input}\ \vec{b}) &= \mathbf{input}\ \vec{b} \\
tree(\mathbf{output}\ \vec{b}) &= \mathbf{output}\ \vec{b} \\
tree(b := e) &= \textit{assign-b}\ (tree(e)) \\
tree(\mathbf{if}\ (e)\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2) &= \mathbf{if}\ (tree(e), \\
&\quad\ \mathbf{then}(tree(s_1),\ tree(s_2))) \\
tree(\mathbf{while}\ (e)\{s\}) &= \mathbf{while}\ (tree(e),\ tree(s))
\end{aligned}
$$

A program over a set of Boolean variables B is hence encoded as a binary tree over the alphabet:

$$
\begin{aligned}
\Sigma = \{root, \neg, \vee, ;, \mathbf{if}, \mathbf{then}, \mathbf{while}\} \cup B \cup \{\textit{assign-b} \mid b \in B\} \\
\cup \{\mathbf{input}\ \vec{b} \mid \vec{b} \in B^{N_I}\} \cup \{\mathbf{output}\ \vec{b} \mid \vec{b} \in B^{N_O}\}
\end{aligned}
$$

The tree automata we build will accept such trees that represent programs that satisfy the given specification. Of course, the set of trees corresponding to *all* programs over a particular *finite* set of Boolean variables B is regular; we skip this proof, as it follows pretty much from the fact that the set of parse trees of any context-free grammar is regular.

▶ **Lemma 1.** *Let B be a finite set of variables. Then the set of trees corresponding to programs over variables B is regular.* ◀

Regular specifications:

A (linear-time) specification over (N_I, N_O) is a subset of ω-sequences $L \subseteq (\{0,1\}^{N_I+N_O})^\omega$ that depicts the correct infinite sequences of input-output behavior allowed by the specification. A specification L is said to be *regular* if there is a *non-deterministic Büchi automaton* over the alphabet $\{0,1\}^{N_I+N_O}$ that precisely accepts L (we don't define word automata in this paper; we refer the reader to [24]).

We will assume that regular specifications L are given by a non-deterministic automaton that accepts the set of sequences *not* in L, i.e., by a non-deterministic Büchi automaton accepting $\overline{L} = (\{0,1\}^{N_I+N_O})^\omega \setminus L$. This is without loss of generality as regular languages are effectively closed under complement.

For any linear-time temporal logic (LTL) specification φ over a set of $N_I + N_O$ propositions can be seen as defining a regular specification $L_\varphi = \{\alpha \in (\{0,1\}^{N_I+N_O})^\omega \mid \alpha \models \varphi\}$. Furthermore, given an LTL specification φ, we can construct a Büchi automaton A accepting $\overline{L}$ in time exponential in $|\varphi|$ and whose size is exponential in $|\varphi|$, by building the automaton accepting the models of $\neg\varphi$, using the now-classic temporal-logic–automata connection [27].

We will work with regular ω-specifications given by automata accepting $\overline{L}$ in the sequel, derive the complexity results for synthesis in terms of the size of this automaton, and, as a corollary, derive the complexity of synthesis for LTL specifications.

3 Trees and Alternating automata

We use labeled binary *finite* trees throughout this paper. Given a finite set of labels Σ, an Σ-labeled tree is a pair $T = (V, \lambda)$, where $V \subseteq \{L, R\}^*$ that is finite and prefix closed, and $\lambda : V \to \Sigma$. The edges of the tree are implicit: for every $v \in V$, if $v.L \in V$, then $v.L$ is the left-child of v, and if $v.R \in V$, then $v.R$ is the right-child of v; the node ϵ is the root of the tree. The function λ assigns a label in Σ to each node of the tree.

For convenience, let us overload the concatenation operator so that for any $v \in \{L, R\}^*$, $v.U = v'$ if $v'.L = v$ or $v'.R = v$; i.e., $v.U$ refers to the parent of v in the tree, obtained by going *up* from v.

Non-deterministic finite automata on trees are the classic top-down tree-automata, with different transition functions defined for nodes that have a left-child only, a right-child only, or has both children. We refer the reader to a textbook on tree automata for details [7]; we fix here only a brief definition and notation.

A *non-deterministic finite tree automaton* on Σ-labeled trees is a structure $\mathcal{A} = (Q, q_0, \delta_L, \delta_R, \delta_{LR}, F)$, where Q is a finite set of states, $q_0 \in Q$, $\delta_L, \delta_R : Q \times \Sigma \to 2^Q$, $\delta_{LR} : Q \times \Sigma \to 2^{Q \times Q}$, and $F \subseteq Q$.

A run of such a tree automaton on a finite tree (V, λ) is a Q-labeled tree (V, ρ) where: $\rho(\epsilon) = q_0$, and for every $v \in V$, (i) if v has a left-child but no right-child, then $\rho(v.L) \in \delta_L(q, \lambda(v))$; (ii) if v has a right-child but no left-child, then $\rho(v.R) \in \delta_R(q, \lambda(v))$; and (iii) if v has both children, then $(\rho(v.L), \rho(v.R)) \in \delta_{LR}(q, \lambda(v))$

A run (V, ρ) is accepting if for every leaf v of V, $\rho(v) \in F$. A tree is accepted by the automaton if there is an accepting run on it. The language of the tree automaton is the set of all Σ-labeled trees accepted by it.

Two-way alternating ω-automata on finite trees:
We now define two-way alternating ω-automata on finite trees. This is a bit unusual; ω-automata are usually defined on *infinite trees*, not on finite ones. However, we will deal with only accepting finite-trees in this paper, which will be used to encode programs (which have a finite description, of course). However, in order to simulate these programs on ω-length sequences of inputs, we would need tree automata working on finite trees for infinitely many steps, going up and down the tree.

For any set S, let $\mathcal{B}^+(S)$ denote the set of all positive Boolean formulas over S; i.e., the set defined by the grammar:

$$\varphi ::= \mathit{true} \mid \mathit{false} \mid s \mid \varphi \vee \varphi \mid \varphi \wedge \varphi$$

where $s \in S$.

A *two-way alternating Büchi tree automaton* over Σ-labeled trees, is a tuple $A = (Q, q_0, \delta_L, \delta_R, \delta_{LR}, \delta_\emptyset, F)$, where Q is a finite set of states, $q_0 \in Q$, $F \subseteq Q$, and where:

- $\delta_L : Q \times \Sigma \times \{L, D\} \to \mathcal{B}^+(Q \times \{L, U\})$
- $\delta_R : Q \times \Sigma \times \{R, D\} \to \mathcal{B}^+(Q \times \{R, U\})$
- $\delta_{LR} : Q \times \Sigma \times \{L, R, D\} \to \mathcal{B}^+(Q \times \{L, R, U\})$
- $\delta_\emptyset : Q \times \Sigma \times \{D\} \to \mathcal{B}^+(Q \times \{U\})$

Intuitively, $\delta_{LR}(q, a, dir) = \varphi$ denotes the actions the automaton can take when in state q, reading a node n whose label is a, and when the *last* move it did is given by dir, where dir=L (dir=R) means that in the last move the tree automaton came *up* from the left child (respectively, right child) of n, and dir=D means that in the last move the tree automaton came *down* from the parent of n. The tree automaton, at such a point, is allowed to choose any Boolean valuation of the set $(Q \times \{L, R, U\})$ that satisfies the formula φ, and for every (q', g) that it sets to true, it must pass a copy of itself in state q' along the direction g, where $g = L, R, U$ is interpreted as left-child, right-child, and up to the parent, respectively. The transitions δ_L and δ_R (for nodes with only a left-child or only a right-child) and $\delta_\emptyset$ (for leaves of the tree) are similarly interpreted. By convention, we assume that at the beginning, the tree automaton starts at the root with the last move set to dir=D. The tree automaton accepts the tree if all its branches formed by the implicit infinite tree it defines by propagating states meet the set F infinitely often.

Two-way alternating co-Büchi automata are similarly defined; here the tree automaton accepts if all its branches meet F only *finitely* often.

Semantics of two-way alternating automata:
Formally, let us define the acceptance of a tree by an automaton $A = (Q, q_0, \delta_L, \delta_R, \delta_{LR}, \delta_\emptyset, F)$ using a game, played between two players, the automaton-player (player 0) and the path-finder player (player 1) (we can also equivalently define acceptance using infinite trees).

A *finite-state two-player Büchi arena* is a tuple $G = (P_0, P_1, E, p_0, F)$ where P_0 and P_1 are two finite disjoint sets representing the positions from where players 0 and 1 play, respectively, $E \subseteq (P_0 \times P_1) \cup (P_1 \times P_0)$ is a set of edges that defines a bipartite graph over P_0 and P_1, $p_0 \in P_0$ is the initial position, and $F \subseteq P_0 \cup P_1$ is a set of Büchi positions.

A *strategy for player* 0 is a function $f_0 : (P_0P_1)^* \to P_1$, such that for any $\sigma \in (P_0P_1)^*$ and $p \in P_1$, $(p, f_0(\sigma.p)) \in E$. In other words, f_0 encodes a strategy for player 0 to choose a successor vertex after any finite sequence of moves that is a partial play in the game.

A *play* is a finite or infinite path in the graph defined by the arena, and denoted by a sequence in $\{p_0\}.(P_1P_0)^*(\epsilon + P_1) \cup \{p_0\}.(P_1P_0)^\omega$. A maximal play is a play that cannot be extended (an ω-length play is maximal; a finite-length play is maximal only if the final vertex has no out-going edges). A play σ *conforms* to a strategy f_0 for player 0 if for every proper prefix $\sigma' \in (P_0P_1)^*P_0$ of σ, $\sigma' f(\sigma')$ is also a prefix of σ. A strategy for player 0, f_0, is said to be winning if for all maximal plays σ that conform to f_0, σ is not finite and some position in F occurs infinitely often in σ. We say that player 0 wins the game on the arena if it has a winning strategy.

Intuitively, a strategy for player 0 is winning if along any play conforming to the strategy player 1 gets "stuck" (cannot make a move) or the play is infinite and meets the Büchi final state set infinitely often.

We can now define when a two-way alternating automaton $A = (Q, q_0, \delta_L, \delta_R, \delta_{LR}, \delta_\emptyset, F)$ accepts a tree (V, λ). Let us define the arena corresponding to A and the tree as (P_0, P_1, E, p_0, F') where:

- $P_0 = (V \times Q \times \{L, R, D\})$
- $P_1 = (V \times 2^{Q \times \{L,R,U\}})$
- E contains the following edges:
  - An edge from (v, q, dir) to (v, S) iff setting S to true and the other elements in $Q \times \{L, R, U\}$ to false satisfies $\delta_h(q, \lambda(v), dir)$, where h=L if v has only a left-child, h=R

if v has only a right-child, and h=LR if v has two children, and $h = \emptyset$ if v is a leaf.

- An edge from (v, S) to (v', q', dir) iff $(q', g) \in S$ and $v' = v.g$ and either $g \in \{L, R\}$ and $dir = D$, or $g = U$ and $v'.dir = v$

The initial position is $p_0 = \{(\epsilon, q_0, D)\}$. The set of Büchi states is $F' = V \times F \times \{L, R, D\}$.

Then the automaton $\mathcal{A}$ *accepts the tree* (V, λ) if player 0 has a winning strategy on the corresponding arena. The language of the automaton $\mathcal{A}$ is the set of trees accepted by it.

We can similarly define *co-Büchi* automata; here player 0 wins iff if the set F is met only *finitely* often.

The size of a two-way alternating automaton is the length of its description encoded as a string.

Two-way alternating automata to one-way non-deterministic automata: It is well-known that two-way alternating tree automata can be converted to non-deterministic tree automata with an exponential blow-up in the state-space (see [26] for example, where such a construction is shown for automata on infinite trees). We can do a similar construction to convert alternating Büchi or co-Büchi tree automata over finite trees to an equivalent non-deterministic automaton over finite trees, with an exponential blow-up. We omit the proof here; a gist of the proof can be found in [12]. Consequently two-way Büchi and co-Büchi alternating tree automata on finite trees capture only regular tree languages, and their emptiness problem can be decided in exponential time.

As an auxiliary notation, in the sequel, we sometimes write transitions of the form $\delta(q, a, dir) = (q', LR)$, where we mean by (q', LR) that the automaton passes the state q' to the right-child of the left-child of the current node. We do this for brevity; such transitions can easily be converted to standard transitions using intermediate states.

4 Synthesizing Reactive Programs

In this section, we prove our first main result: for any regular specification and a set of Boolean variables B, we can build a tree automaton that precisely accepts the class of all tree-encodings of programs over B that satisfy the specification. By checking emptiness of this tree automaton, we can synthesize programs that satisfy the specification, and in particular, synthesize the smallest programs satisfying the specification.

Let us fix input and output arities $N_I, N_O \in \mathbb{N}$ for the rest of this section. Let A_{spec} be a non-deterministic Büchi automaton that accepts the set of sequences in $(\{0,1\}^{N_I+N_O})^\omega$ that *do not* satisfy the specification. Let us also fix a set of Boolean variables B.

Consider the set of all trees corresponding to programs over the variables B with input and output arities N_I and N_O. By Lemma 1, this is a regular set of trees, and let's assume that A_{pgm} is a tree automaton that accepts precisely these trees.

We now build a two-way alternating Büchi tree automaton that accepts a tree encoding a program iff the program does *not* respect the specification. Intuitively, this automaton $\mathcal{A}$ will guess a particular run of the program by non-deterministically choosing a sequence of inputs, simulating the program on these inputs, and checking whether there is a run of the specification automaton $\mathcal{A}_{spec}$ that accepts this execution; if so, it will accept the tree.

The two-way alternating tree automaton will have as states two kinds of tuples. The first kind are tuples of the form $\langle s, q, i, m, t\rangle$ where s is the current state of the program's simulation (i.e., the valuation of the variables B), q is the current state of the specification automaton A_{spec} that is simultaneously simulated on the input-output sequence observed, i is the *last* input received by the program, and m is a mode $m \in \{inp, out\}$ that remembers whether the next I/O operation the program must do is an input or an output. The final

bit $t \in \{0,1\}$ is a toggle that switches to 1 each time the specification state is updated, and then gets set back to 0.

The second kind of state is of the form $\langle s, v\rangle$ where s is the current state of the program's simulation and v is a Boolean value; these states are used on subtrees that encode Boolean expressions (right-hand sides of assignments and conditionals in if- and while-statements), and are meant to check whether the expression evaluates to the value v in the current state of the program s.

Intuitively, the automaton walks over the program tree, interpreting every statement, and computing the current state of the variables in s. In this process, it may have to move up and down the tree as `while`-loops require traversing the same blocks of statements multiple times. When it meets an output-statement, it updates the specification automaton's state on the last input i and the output valuation. When it meets an input statement, it stores it in the appropriate variables in s, and updates the component i.

We now define the automaton formally. Let $Bool = \{0,1\}$. For such a valuation s of B, we denote by $s[b/v]$, where $b \in B$ and $v \in Bool$, the valuation that s' such that $s'(b) = v$ and for every $b' \in B, b' \neq b$, $s'(b') = s(b)$. We extend this to tuples of replacements: $s[\vec{b}/val]$ where val is a valuation of $\vec{b}$ is defined as the valuation s modified so that $\vec{b}$ evaluates to val.

Let S denote the set of all valuations of the variables B. Let $I = \{0,1\}^{N_I}$ denote the set of all inputs. Let $A_{spec} = (Q, q_0, \delta_{spec}, F_{spec})$.

The two-way alternating automaton is $\mathcal{A} = (P, p_0, \delta_L, \delta_R, \delta_{LR}, \delta_\emptyset, F)$ is defined as follows. The set of states is:

$$P = (S \times Bool) \cup (S \times Q \times I \times \{inp, out\} \times Bool)$$

The transitions are defined as follows, where $s \in S$, $v \in Bool$, $q \in Q$, $i \in I$, $m \in \{inp, out\}$, and $t \in \{0,1\}$.

- **Transitions from root:**
  $\delta_L(p_0, root, D) = (p_0, L)$; $\quad \delta_L((s,q,i,m,t), root, U) = true$;
- **Transitions to evaluate Boolean expressions:**
  - $\delta_\emptyset((s,1), \mathbf{tt}, D) = true$; $\quad \delta_\emptyset((s,0), \mathbf{tt}, D) = false$
  - $\delta_\emptyset((s,1), \mathbf{ff}, D) = false$; $\quad \delta_\emptyset((s,0), \mathbf{ff}, D) = true$
  - $\delta_\emptyset((s,v), b, D) = true \quad if\ s[b] = 1$
    $= false \quad otherwise.$
  - $\delta_{LR}((s,1), \vee, D) = ((s,1), L) \vee ((s,1), R)$
    $\delta_{LR}((s,0), \vee, D) = ((s,0), L) \wedge ((s,0), R)$
  - $\delta_L((s,v), \neg, D) = ((s, 1{-}v), L)$
- **Transitions to evaluate non I/O statements:**
  - $\delta_\emptyset((s,q,i,m,t), \mathbf{skip}, D) = ((s,q,i,m,0), U)$
  - $\delta_L((s,q,i,m,t), assign_b, D) =$
    $(\ (s[b/0],q,i,m,0), U) \wedge ((s,0), L)\ )\ \vee\ (\ (s[b/1],q,i,m,0), U) \wedge ((s,1), L)\ )$
  - $\delta_{LR}((s,q,i,m,t), \mathbf{if}, D) =$
    $(\ ((s,1), L) \wedge ((s,q,i,m,0), RL)\ )\ \vee\ (\ ((s,0), L) \wedge ((s,q,i,m,0), RR)\ )$
  - $\delta_{LR}((s,q,i,m,t), \mathbf{while}, D)$
    $= \delta_{LR}((s,q,i,m,t), \mathbf{while}, R)$
    $= (\ ((s,1), L) \wedge ((s,q,i,m,0), R)\ )\ \vee\ (\ ((s,0), L) \wedge ((s,q,i,m,0), U)\ )$
- **Transitions to evaluate input and output:**
  - $\delta_\emptyset((s,q,i,inp,t), \mathbf{input}\ \vec{b}) = \quad \bigvee_{valuations\ val\ over\ \vec{b}} ((s[\vec{b}/val], q, val, out, 0), U)$
  - $\delta_\emptyset((s,q,i,out,t), \mathbf{output}\ \vec{b}) = \quad \bigvee_{q' \in \delta_{A_{spec}}(q, \vec{i}, s[\vec{b}])} ((s, q', i, inp, 1), U)$
- **Transitions to move to next statement in program:**
  - $\delta_{LR}((s,q,i,m,t), \mathbf{;}, D) = ((s,q,i,m,t), L)$

- $\delta_{LR}((s,q,i,m,t),\mathbf{;},L) = ((s,q,i,m,t),R)$
- $\delta_{LR}((s,q,i,m,t),\mathbf{;},R) = ((s,q,i,m,t),U)$
- $\delta_{LR}((s,q,i,m,t),\mathbf{then},L)$
  $= \delta_{LR}((s,q,i,m,t),\mathbf{then},R)$
  $= ((s,q,i,m,t),U)$
- $\delta_{LR}((s,q,i,m,t),\mathbf{if},R) = ((s,q,i,m,t),U)$

All other transitions evaluate to *false*.

The initial state is $p_0 = (s_0, q_0, i_0, inp, 0)$ where s_0 is the function that maps every variable in b to F (to reflect the initial state of the variables), q_0 is the initial state of the specification automaton A_{spec}, $i_0 = 0^{N_I}$ and where the mode is set to receive an input (which will overwrite i_0). This state is passed on from the root to the first statement of the program.

The set of Büchi final states is $F = \{(s,q,i,m,1) \mid s \in S, q \in F_{spec}, i \in I, m \in M\}$. Since the toggle t is 1 in all these states, the automaton is forced to truly hit a Büchi final state of A_{spec} infinitely often (as long as it runs forever), which in turn forces it to infinitely-often react with its environment.

A state of the form (s,v) is meant to check whether the state s satisfies the expression encoded at the subtree of the node the tree is reading. The transitions reflect this check: the expression b checks if it matches the value of v, and disjunctive and negated expressions are checked by sending appropriate copies to check sub-expressions. Note that these copies always go *down* the tree, and terminate at the leaves.

The processing of non-I/O statements requires walking up and down the tree. A skip-statement at a leaf is handled simply by going up. An assignment to a variable b is handled by guessing a Boolean value v, sending a copy down the tree to check that the expression evaluates to v, and sending another copy up with the value of b updated to v in the state. Conditionals are handled by again guessing whether the value of the expression is true or not, sending a copy to the left-child to check if this is correct, and sending a copy to the appropriate child of the right-tree to execute the if-branch or the else-branch. Recursive while-loops are handled similarly; the value of the condition is guessed, a copy is sent down the left-branch to check it, and another copy is either sent down to the right-branch to execute the body of the loop, or sent up to exit the loop.

The input-statement is handled only when in input mode (*inp*), and the automaton evaluates the input variables to an arbitrary valuation, stores this both in the state s and the input component i, and switches the mode to output (*out*). An output-statement does not change the state of the system, but changes the state of the specification, which is updated by simulating the specification automaton A_{spec} non-deterministically on the last input and the current output.

Notice that the last toggle bit t always gets set to 0, except when processing an output statement, where it gets set to 1. This ensures that specification states seen at intermediate points while simulating the program do not count towards meeting the Büchi specification in A_{spec}. Also note that if the program *halts* after a finite input sequence, the automaton will reach the root, and accept the tree; hence the automaton accepts also all programs that have a terminating computation.

There are also several other transitions that help move between statements. When a ";" is met going down a tree, the control goes to the left-child to process the first statement, when it comes up from left-child, it goes to the right-child to process the second statement, and when that comes up, the control passes to the parent. When control comes up to an if-

or then- node, the control moves up the tree.

We build a second two-way Büchi alternating automaton $A_{non\text{-}reactive}$ that accepts all programs that do *not* infinitely interact with the environment (i.e., those programs that never produce an output after a finite sequence of interactions with the environment). We skip this construction, as it is very similar and simpler than the construction of A above.

Finally, we take the union of the two two-way Büchi alternating tree-automata A and $A_{non\text{-}reactive}$, *complement* it (by dualizing the transition relation and making the acceptance condition co-Büchi), and intersect it with the automaton A_{pgm}. This gives a two-way alternating co-Büchi automaton that accepts precisely those programs that continually interact with the environment on all possible input sequences and satisfy the specification. This is then transformed to an equivalent non-deterministic tree-automaton, incurring an exponential blow-up.

The following theorem captures the correctness and complexity of the construction (note that $|B|$ dominates N_I and N_O); a gist of its proof can be found in [12].

▶ **Theorem 2.** *Let B be a finite set of Boolean variables, and let $N_I, N_O \in \mathbb{N}$. Let A_{spec} be a non-deterministic Büchi automaton over the alphabet $\{0,1\}^{N_I+N_O}$. Then we can construct a non-deterministic tree automaton $\mathcal{B}$ that precisely accepts the trees corresponding to reactive programs over B and with input/output type (N_I, N_O) that on all executions generate I/O sequences that are not in $L(A)$. Furthermore, this tree-automaton can be constructed to be of size $O(exp\ (|A_{spec}|,\ exp(B)))$.* ◀

Note that the final automaton is doubly-exponential in B and singly exponential in the size of the specification automaton. For a fixed B, N_I, N_O, this gives an ExpTime decision procedure. As a corollary, it follows that for specifications given in Ltl, the synthesis procedure is in 2ExpTime. Recall that the *transition-system* synthesis problem for LTL (with no parametrization like the variables B in program synthesis) is 2ExpTime-complete [17].

5 Synthesizing Recursive Reactive Programs

We extend the results of the previous section to synthesizing Boolean reactive programs *with function-calls and recursion*, provided the *number* of functions and their signatures are fixed. The proof strategy is similar but more complex: we encode programs with recursion also as trees, where functions are encoded as subtrees of the program tree, and show that the class of recursive programs that meet a regular specification is regular.

The idea of synthesizing recursive programs for regular specifications is motivated by three reasons. First, we are interested in synthesizing the smallest programs that satisfy a specification, and the smallest programs may involve recursion. Second, there may be *no programs* that satisfy a regular specification R over a set of Boolean variables B, while there *exist* recursive programs over B that satisfy R. This issue does not apply in the classical case of synthesizing transition systems as if there is a transition-system at all that satisfies a regular specification, there is always a finite-state transition system that satisfies it. In the program synthesis setting, the additional parametrization of B causes a smaller search space of programs. Finally, in applications to *program repair* (see Section 7 for a discussion), where we want to repair an existing recursive program against a regular specification, the natural problem that is required is one that synthesizes all recursive programs that satisfy a specification.

Our proof procedure also extends smoothly to synthesizing recursive programs against *visibly pushdown automata* [3] specifications, which are a class of specifications larger than

that of regular specifications. This includes the class of temporal logics for recursive programs that can be compiled into visibly pushdown Büchi automata, such as CaRet [2] and and NWTL [1]. We note that we are unaware of any natural analog in the transition-system world corresponding to the results on synthesizing recursive programs against non-regular specifications in this section.

Recursive programs: Boolean programs with recursion are defined as sequential programs, except that we have functions that can call each other (with call-by-value semantics), functions have local variables that they can manipulate and can return a tuple of Boolean values, and can have side-effects by changing the valuation of globally declared variables.

Let us fix $B = (B_G, B_l)$, where B_G is a finite set of *global* Boolean variables and B_L and a finite set of local Boolean variables. Let us also fix a finite number of function-names F with a special function *main* in F. Let $Inp : F \to \mathbb{N}$ and $Ret : F \to \mathbb{N}$ be two functions that give the number of input parameters and the number of return values for each function, and let us fix $N_I, N_O \in \mathbb{N}$.

Then the set of recursive Boolean programs over $\langle B, F, In, Out, N_I, N_O\rangle$ is given by:

$$
\begin{array}{lcl}
\langle pgm\rangle & ::= & f(\vec{b})\ \{\ \langle stmt\rangle\ \} \ \mid\ \langle pgm\rangle\ \ \langle pgm\rangle \\
\langle stmt\rangle & ::= & \langle stmt\rangle; \langle stmt\rangle \ \mid\ \textbf{skip} \ \mid b := \langle expr\rangle \ \mid\ \textbf{input}\ \vec{b} \ \mid\ \textbf{output}\ \vec{b} \ \mid \\
 & & \textbf{if}\ (\langle expr\rangle)\ \textbf{then}\ \{\langle stmt\rangle\}\ \textbf{else}\ \{\langle stmt\rangle\} \ \mid\ \textbf{while}\ (\langle expr\rangle)\ \{\langle stmt\rangle\} \ \mid \\
 & & \langle b_1, \ldots, b_k\rangle = f(b'_1, \ldots, b'_i) \ \mid\ \textbf{return}\ (b_1, \ldots, b_r) \\
\langle expr\rangle & ::= & b \mid \textbf{tt} \mid \textbf{ff} \mid (\langle expr\rangle \vee \langle expr\rangle) \mid (\neg\langle expr\rangle)
\end{array}
$$

Programs consist of a series of definitions of the functions, where each function, in addition to the usual statements, is allowed to call other functions and assign the returned values to variables, as well as return a tuple of Boolean values. We assume natural restrictions on programs: each function is defined precisely once, and the arities of its input parameters and returns match the *In* and *Ret* functions.

The semantics is once again the natural one. When a function f_1 calls f_2, its local variables are pushed onto an (unbounded) stack along with the *program counter* at which the call occurred, and the control switches to the beginning of f_2, with its local variables reset to default initial values. When the function f_2 executes a **return**-statement, the stack is popped to retrieve the state of the caller function, the control switches to the caller at the appropriate program counter, and the returned values get stored in the appropriate variables mentioned in the statement that called f_2. The valuations of global variables can change across a call.

Representing recursive programs as finite trees: We can represent recursive programs using finite trees by essentially encoding each function as a sub-tree. Intuitively, we build a tree whose right-most path is labeled with a special symbol \$, and the sub-trees that hang from these nodes as left-children encode the various functions in the program.

Formally, we extend the function *tree* defined in the previous section with the following definitions, where $p, p' \in \langle pgm\rangle$.

$$
\begin{array}{rcl|rcl}
tree(p\ p') & = & \$(tree(p), tree(p')) & tree(\vec{b} := f(\vec{b'})) & = & call\text{-}f\text{-}\vec{b}\text{-}\vec{b'} \\
tree(f(\vec{b})\{\langle stmt\rangle\}) & = & f\text{–}\vec{b}(tree(\langle stmt\rangle)) & tree(\textbf{return}\ \vec{b}) & = & ret\text{-}\vec{b}
\end{array}
$$

Synthesizing recursive programs: The synthesis procedure is similar to that of non-recursive programs. We show that the class of *all* recursive programs over $\langle\langle B_G, B_L\rangle, F, In, Out, N_I, N_O\rangle$ that satisfy a specification (given by an automaton $\mathcal{A}_{spec}$ that

accepts the sequences that do *not* confirm to the specification) is regular. The construction of the tree-automaton accepting the set of correct programs is considerably more complex. We give a gist of it below, highlighting the main parts of the construction.

Intuitively, when simulating the program, when we are at a state s, and the specification automaton state is q, and we process a *call* to a function f (i.e., a statement of the form $\vec{b} = f(\vec{b'})$), the tree automaton *guesses* the precise state s' and precise state q' the program and specification automaton would be in *after* returning from f, and sends two copies, one to continue computation in the current function from s' and q', and another to check whether the call to f does indeed transform the state from (s, q) to (s', q'). Note that the tree automaton is also guessing the *input* to the program on-the-fly as it simulates it; since the tree automaton is guessing only *one* input sequence on which the program responds, the methodology above to handle function calls works.

Recall the construction of the automaton in Section 4. The states there are of the form (s, q, i, m, t). In the new construction, the states will be of the form $(s, q, i, m, t, s', q', i', m', h)$ where $s, s' \in S$, $q, q' \in Q$, $m, m' \in \{inp, out\}$, and $t, h \in \{0, 1\}$. Intuitively, this state means that the program is in state (s, q, i, m, t) (as before) and is supposed to return from the current function at the state (s', q', m', t') (for some $t' \in \{0, 1\}$), and, if $h = 1$, it must see in the interim a Büchi final state.

The details of the construction are much more tedious, and we skip the construction here; a detailed construction can be found in [12]. The intent is however fairly straightforward: in a state $(s, q, i, m, t, \widehat{s}, \widehat{q}, \widehat{i}, \widehat{m}, h)$, reading a call to a function f (i.e., a statement of the form $\vec{b} := f(\vec{b'})$, the automaton will non-deterministically pick a quadruple (s', q', i', m') and $t', h', h'' \in \{0, 1\}$, and will (a) send a copy $(s'', q, i, m, s', q', i', m')$ up the tree to the first statement in the definition of the function f, where s'' is the appropriate state with formal parameters updated according to $\vec{b'}$, and will send another copy to the next statement in the current function in the state $(s''', q', i', m', 0, \widehat{s}, \widehat{q}, \widehat{i}, \widehat{m}, h'')$, where $h'' = h + h'$, and s''' is the state obtained from s by replacing $\vec{b}$ with the values returned at state s'. The latter copy will also have to pass through a transient intermediate Büchi final state if $h'' = 1$ (i.e., if f promises to meet a Büchi final state). Furthermore, we also need to provide a possibility for the function call f to *never return*; this will involve sending the current state to f with the proviso that if it meets a **return**-statement, then the tree would be rejected. We skip further details, and conclude with the main theorem for this section:

▶ **Theorem 3.** *Let $B = \langle B_G, B_L \rangle$ be a finite set of global and local Boolean variables, let F be a finite set of function-names, with arity functions In and Ret as above, and let $N_I, N_O \in \mathbb{N}$. Let A_{spec} be a non-deterministic Büchi automaton over the alphabet $\{0, 1\}^{N_I + N_O}$. Then we can construct a non-deterministic automaton that accepts precisely the trees corresponding to recursive reactive programs over $\langle B, F, In, Ret, N_I, N_O \rangle$ that on all executions generate I/O sequences that are not in $L(A)$. Furthermore, this tree-automaton can be constructed to be of size $O(exp\ (|A_{spec}|,\ |F|,\ exp(B)))$.* ◀

Handling visibly pushdown automata specifications:
The results in the above section smoothly extend to specification given as *non-deterministic visibly pushdown Büchi automata on ω-words.* [3].

Given a set of function-names F, a visibly pushdown automaton is over a triple alphabet $\langle \Sigma_c, \Sigma_r, \Sigma_i \rangle$ of calls, returns, and internal actions, respectively. A program's behavior is redefined to be such that its function-calls and returns are made *visible*. More precisely, let us assume that $\Sigma_c = F$, $\Sigma_r = \{\overline{f} \mid f \in F\}$, and let $\Sigma_i = \{0, 1\}^{N_I} \cup \{0, 1\}^{N_O}$. Given a run of a program, we note down not just the input-output sequences it does (which have now

been split), but also record the calls and returns the program makes. A visibly pushdown automaton is a pushdown automaton (a finite automaton with a stack) that is restricted so that it can *only* push on calls, only pop on returns, and cannot touch the stack on internal actions. Visibly pushdown automata can specify properties of the runs of a program: for example, given a pre- and post-condition for a function f, the visibly pushdown automaton can specify that f computes a function that conforms to it; such a specification is *not* regular but is a visibly pushdown language. Again, we assume that specifications are given by a visibly pushdown automaton that accepts the set of sequences that *do not* conform to the specification (visibly pushdown languages are closed under complement [3]).

Since the visibly pushdown automaton's stack is *synchronous* with the program's stack, the above synthesis procedure for recursive programs can be extended; at a call, the tree automaton will send an *updated* state of the specification (on the corresponding call) and update the automaton state on the corresponding return. We skip the construction, as it is almost identical to the the previous one save for the update of the specification automaton's state. The complexity of the construction also remains the same.

6 Discussion and Future Directions

While we have focussed on imperative programs in this paper, the synthesized artefacts can be other compact representations in other domains as well: for example, synthesis problems can be targeted towards functional programs, towards non-reactive programs that compute one output from one input, or even hardware designs in a high-level hardware description language, like Verilog, RTL, or SystemC. We hope that the work presented here will inspire extending existing transition-system synthesis algorithms to artefact-oriented synthesis.

Program Repair: The results we have presented in this paper are extremely well-suited for *repairing programs*. Given a program P (with or without recursion) that does not satisfy a specification φ, the program-repair problem is to change P (in certain minimally allowed ways) so that it satisfies the specification φ.

Given a program P, let us assume that the set of *repaired versions* of the program that we want to search over is S. Repaired versions of programs may be defined as versions of the program obtained in certain ways, for example, changing only the conditionals in particular parts of the program, redefining only a particular function f, or having a hole in a program that needs to be filled by arbitrary code. Let us further assume that the trees corresponding to the programs in S defines a *regular* class of trees (this is a reasonable assumption; many repair conditions for programs are regular).

We can now synthesize a repair by constructing the class of all programs over the appropriate set of variables and functions (as defined by P and S), and construct a tree automaton $\mathcal{T}$ that accepts the trees of all these programs. The emptiness of the intersection of the languages of S and $\mathcal{T}$ gives the *repaired* versions of P that meet the specification.

A future direction we see is to *apply* Boolean program repair (facilitated by this paper) along with *abstraction* to repair programs over *unbounded* data domains.

Designing programs from automata accepting transition systems: One point worth making is that we can build programs from automata that accept transition systems. More precisely, assume that a transition-system synthesis procedure builds a tree-automaton $\mathcal{A}$ that accepts precisely the set of all trees that satisfy a particular specification. Then, we can build a tree automaton $\mathcal{B}$ that accepts precisely the set of all trees that correspond to (non-recursive or recursive) programs whose transition-system unfolding is accepted by $\mathcal{A}$ (we skip the details here). Hence *any* synthesis procedure that builds a tree-automaton

accepting unfoldings of transition systems can be turned into a synthesis procedure that constructs programs. However, the procedures laid out in the previous sections directly synthesize programs, and avoid the extra exponential blow-up that would be incurred by first building a tree automaton for transition systems followed by one for synthesizing programs.

Distributed synthesis: The above remark on synthesizing programs using a synthesizer of transition systems tempts us to think that any tree-automata based decision procedure for the synthesis problem for transition systems can be transformed to a synthesizer for programs. However, this is not clear for *distributed synthesis*, where we are required to synthesize programs at different sites of a distributed architecture with synchronous communication between sites [18].

First, the distributed transition-system synthesis problem is *undecidable* even in the simple architecture where there are two disconnected sites P_1 and P_2, each receiving inputs from the environment. It is not hard to adapt the undecidability proofs given in [18] to show that *program synthesis* for this architecture (as well as all other undecidable architectures for transition-system synthesis [18]) remain undecidable for program synthesis.

The classic *decidable* architecture for transition-system synthesis is that of a *pipeline* architecture, where the architecture consists of n processes, $P_1, \ldots, P_n$, where only P_1 receives input from its environment, where all processes have outputs, and where there are channels from P_i to P_{i+1}, for every $1 \leq i < n$. Pnueli and Rosner showed that this architecture has a decidable transition-system synthesis problem [18]. Their procedure (slightly modified) works by first taking the process P_1 and generating a tree-automaton accepting the set of all *communication trees over the first channel from P_1 to P_2* such that there is some strategy for P_1 to generate this tree and there is a strategy for the rest of the system (i.e., $P_2, \ldots P_n$) to generate outputs by reading this tree so as to satisfy the specification. The procedure then *walks* down the pipeline, producing at each point an automaton that accepts communication trees for the channels that admit a feasible synthesis. When we reach the last process, the procedure creates a transition system for P_n, and then *walks back* creating transition systems for the processes P_{n-1} all the way up to P_1.

The above decision procedure, however, does not seem adaptable for *program synthesis*. We can, of course, synthesize the program for P_n. But fixing a *particular program for P_n restricts the choices we have for other sites*. Consequently, when walking back, we may find that there is no program that satisfies the communication tree that we need for synthesis.

The distributed *program*-synthesis problem for pipelines is hence an open problem. Other decidable distributed synthesis problems, such as transition system synthesis for doubly-flanked pipelines against local specifications [14, 13]), also do not readily adapt to program synthesis, and remain open questions.

Acknowledgements: This research was partially supported by NSF CAREER award #0747041.

References

1 Rajeev Alur, Marcelo Arenas, Pablo Barceló, Kousha Etessami, Neil Immerman, and Leonid Libkin. First-order and temporal logics for nested words. *Logical Methods in Computer Science*, 4(4), 2008.

2 Rajeev Alur, Kousha Etessami, and P. Madhusudan. A temporal logic of nested calls and returns. In *TACAS*, volume 2988 of LNCS, pages 467–481. Springer, 2004.

3 Rajeev Alur and P. Madhusudan. Adding nesting structure to words. *J. ACM*, 56(3), 2009.

4 Roderick Bloem, Stefan Galler, Barbara Jobstmann, Nir Piterman, Amir Pnueli, and Martin Weiglhofer. Specify, compile, run: Hardware from PSL. *ENTCS*, 190(4):3–16, 2007.

5 J.R. Büchi and L.H. Landweber. Solving sequential conditions by finite-state strategies. *Trans. AMS*, 138:295–311, 1969.

6 Alonzo Church. Application of recursive arithmetic to the problem of circuit synthesis,. In *Summaries of talks presented at the Summer Institute for Symbolic Logic, Cornell University 1957*, pages 3–50, Princeton, 1957. Institute for Defense Analyses.

7 H. Comon, M. Dauchet, R. Gilleron, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. Draft, Available at `http://www.grappa.univ-lille3.fr/tata/`, 2002.

8 Erich Grädel, Wolfgang Thomas, and Thomas Wilke, editors. *Automata, Logics, and Infinite Games: A Guide to Current Research*, volume 2500 of LNCS. Springer, 2002.

9 Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. In *POPL*, pages 317–330. ACM, 2011.

10 O. Kupferman and M.Y. Vardi. Church's problem revisited. *The Bulletin of Symbolic Logic*, 5(2):245 – 263, 1999.

11 Orna Kupferman, P. Madhusudan, P. S. Thiagarajan, and Moshe Y. Vardi. Open systems in reactive environments: Control and synthesis. In *CONCUR*, volume 1877 of LNCS, pages 92–107. Springer, 2000.

12 P. Madhusudan. *Synthesizing Reactive Programs.* Full version available at `http://www.cs.uiuc.edu/~madhu/csl11.html`.

13 P. Madhusudan. *Control and Synthesis of Open Reactive Systems.* PhD thesis, University of Madras, India, 2001.

14 P. Madhusudan and P. S. Thiagarajan. Distributed control and synthesis for local specificati ons. In *Proc., ICALP*, volume 2076 of *LNCS*, Greece, July 2001.

15 Nir Piterman, Amir Pnueli, and Yaniv Sa'ar. Synthesis of reactive(1) designs. In *VMCAI*, volume 3855 of LNCS, pages 364–380. Springer, 2006.

16 A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th IEEE Symposium on Foundations of Computer Science*, pages 46–77, 1977.

17 A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Proceedings of the 16th ACM Symposium on Principles of Programming Languages*, 1989.

18 Amir Pnueli and Roni Rosner. Distributed reactive systems are hard to synthesize. In *FOCS*, volume II, pages 746–757. IEEE, 1990.

19 M.O. Rabin. Automata on infinite objects and Church's problem. *Amer. Mathematical Society*, 1972.

20 S. Safra. On the complexity of ω-automata. In *Proceedings of the 29th IEEE Symposium on Foundations of Computer Science*, pages 319–327, 1988.

21 Sven Schewe and Bernd Finkbeiner. Bounded synthesis. In *ATVA*, volume 4762 of LNCS, pages 474–488. Springer, 2007.

22 Armando Solar-Lezama. The sketching approach to program synthesis. In *APLAS*, volume 5904 of LNCS, pages 4–13. Springer, 2009.

23 Saurabh Srivastava, Sumit Gulwani, and Jeffrey S. Foster. From program verification to program synthesis. In *POPL*, pages 313–326. ACM, 2010.

24 W. Thomas. Languages, automata, and logic. *Handbook of Formal Language Theory*, III:389–455, 1997.

25 Wolfgang Thomas. Facets of synthesis: Revisiting church's problem. In *FOSSACS*, volume 5504 of LNCS, pages 1–14. Springer, 2009.

26 Moshe Y. Vardi. Reasoning about the past with two-way automata. In *ICALP*, volume 1443 of LNCS, pages 628–641. Springer, 1998.

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