Synthesizing Structured Reactive Programs via Deterministic Tree Automata

Benedikt Brütsch
RWTH Aachen University, Lehrstuhl für Informatik 7, Germany
bruetsch@automata.rwth-aachen.de

Existing approaches to the synthesis of reactive systems typically involve the construction of transition systems such as Mealy automata. However, in order to obtain a succinct representation of the desired system, structured programs can be a more suitable model. In 2011, Madhusudan proposed an algorithm to construct a structured reactive program for a given ω-regular specification without synthesizing a transition system first. His procedure is based on two-way alternating ω-automata on finite trees that recognize the set of "correct" programs.

We present a more elementary and direct approach using only deterministic bottom-up tree automata that compute so-called *signatures* for a given program. In doing so, we extend Madhusudan's results to the wider class of programs with bounded delay, which may read several input symbols before producing an output symbol (or vice versa). As a formal foundation, we inductively define a semantics for such programs.

1 Introduction

Algorithmic synthesis is a rapidly developing field with many application areas such as reactive sytems, planning and economics. Most approaches to the synthesis of reactive systems, for instance [2, 12, 11, 8], revolve around synthesizing transition systems such as Mealy or Moore automata. Unfortunately, the resulting transition systems can be very large. This has motivated the development of techniques for the reduction of their state space (for example, [6]). Furthermore, the method of bounded synthesis [14, 4] can be used to synthesize minimal transition systems by iteratively increasing the bound on the size of the resulting system until a solution is found. However, it is not always possible to obtain small transition systems. For example, for certain specifications in linear temporal logic (LTL), the size of the smallest transition systems satisfying these specifications is doubly exponential in the length of the formula [13].

Aminof, Mogavero and Murano [1] provide a round-based algorithm to synthesize hierarchical transition systems, which can be exponentially more succinct than corresponding "flat" transition systems. The desired system is constructed in a bottom-up manner: In each round, a specification is provided and the algorithm constructs a corresponding hierarchical transition system from a given library of available components and the hierarchical transition systems created in previous rounds. Thus, in order to obtain a small system in the last round, the specifications in the previous rounds have to be chosen in an appropriate way.

Current techniques for the synthesis of (potentially) succinct implementations in the form of circuits or programs typically proceed in an indirect way, by converting a transition system into such an implementation. For example, Bloem et al. [3] first construct a symbolic representation (a binary decision diagram) of an appropriate transition system and then extract a corresponding circuit. However, this indirect approach does not necessarily yield a succinct result.

F. Mogavero, A. Murano, and M.Y. Vardi (Eds.):
1st Workshop on Strategic Reasoning 2013 (SR'13)
EPTCS 112, 2013, pp. 107–113, doi:10.4204/EPTCS.112.16

Madhusudan addresses this issue in [10], where he proposes a procedure to synthesize programs without computing a transition system first. He considers *structured reactive programs* over a given set of Boolean variables, which can be significantly smaller (regarding the length of the program code) than equivalent transition systems. To some degree, these programs separate control flow from memory. Such a separation can also be found in a related approach that has recently been introduced by Gelderie [5], where strategies for infinite games are represented by *strategy machines*, which are equipped with control states and a memory tape.

Given a finite set of Boolean variables and a nondeterministic Büchi automaton recognizing the complement of the specification, Madhusudan constructs a two-way alternating ω-automaton on finite trees that recognizes the set of *all* programs over these variables that satisfy the specification. This automaton can be transformed into a nondeterministic tree automaton (NTA) to check for emptiness and extract a minimal program (regarding the height of the corresponding tree) from that set. In contrast to the transition systems constructed by classical synthesis algorithms, the synthesized program does not depend on the specific syntactic formulation of the specification, but only on its meaning.

In this paper, we present a direct construction of a deterministic bottom-up tree automaton (DTA) recognizing the set of correct programs, without a detour via more intricate types of automata. The DTA inductively computes a representation of the behavior of a given program in the form of so-called *signatures*. A similar representation is used by Lustig and Vardi in their work on the synthesis of reactive systems from component libraries [9] to characterize the behavior of the components.

Our approach is not limited to programs that read input and write output in strict alternation, but extends Madhusudan's results to the more general class of programs with *bounded delay*: In general, a program may read multiple input symbols before writing the next output symbol, or vice versa, causing a delay between the input sequence and the output sequence. In a game-theoretic context, such a program corresponds to a strategy for a controller in a game against the environment where in each move the controller is allowed to either choose at least one output symbol or skip and wait for the next input (see [7]). We consider programs that never cause a delay greater than a given bound $k \in \mathbb{N}$.

For a fixed k, the complexity of our construction matches that of Madhusudan's algorithm. In particular, the size of the resulting DTA is exponential in the size of the given nondeterministic Büchi automaton recognizing the complement of the specification, and doubly exponential in the number of program variables. In fact, we establish a lower bound, showing that the set of all programs over n Boolean variables that satisfy a given specification cannot even be recognized by an NTA with less than $2^{2^{n-1}}$ states, if any such programs exist. However, note that a DTA (or NTA) accepting precisely these programs enables us to extract a minimal program for the given specification and the given set of program variables. Hence, the synthesized program itself might be rather small.

To lay a foundation for our study of the synthesis of structured reactive programs, we define a formal semantics for such programs, which is only informally indicated by Madhusudan. To that end, we introduce the concept of *Input/Output/Internal machines (IOI machines)*, which are composable in the same way as structured programs. This allows for an inductive definition of the semantics.

2 Syntax and Semantics of Structured Programs

We consider a slight modification of the structured programming language defined in [10], using only single Boolean values as input and output symbols to simplify notation. *Expressions* and *programs* over a finite set B of Boolean variables are defined by the following grammar, where $b \in B$:

$$\langle expr \rangle \ ::= \ \texttt{true} \mid \texttt{false} \mid b \mid \langle expr \rangle \wedge \langle expr \rangle \mid \langle expr \rangle \vee \langle expr \rangle \mid \neg \langle expr \rangle$$

$$\langle prog \rangle \ ::= \ b := \langle expr \rangle \mid \texttt{input } b \mid \texttt{output } b \mid \langle prog \rangle ; \langle prog \rangle$$
$$\texttt{if } \langle expr \rangle \texttt{ then } \langle prog \rangle \texttt{ else } \langle prog \rangle \mid \texttt{while } \langle expr \rangle \texttt{ do } \langle prog \rangle$$

Intuitively, "`input` b" reads a Boolean value and stores it in the variable b. Conversely, "`output` b" writes the current value of b. To define a formal semantics we associate with each program a so-called *IOI machine*. An IOI machine is a transition system with designated entry and exit states. It can have input, output and internal transitions, with labels of the form $(a_{\text{in}}, \varepsilon)$, $(\varepsilon, a_{\text{out}})$ or $(\varepsilon, \varepsilon)$, respectively, where $a_{\text{in}}, a_{\text{out}} \in \mathbb{B} = \{0,1\}$. An IOI machine is equipped with a finite set B of Boolean variables, whose valuation is uniquely determined at each state. A *valuation* is a function $\sigma \colon B \to \mathbb{B}$ that assigns a Boolean value to each variable.

The associated IOI machine of an atomic program (i.e., an input statement, output statement or assignment) has one entry state and exit state for each possible variable valuation, and its transitions lead from entry states to exit states. For example, at each entry state of the associated IOI machine of an atomic program of the form "`input` b", there are two outgoing input transitions – one for each possible input symbol. The target of such an input transition is the exit state whose variable valuation is obtained by replacing the value of b with the respective input symbol. The IOI machine of a composite program can be constructed inductively from the IOI machines of its subprograms, leveraging their entry and exit states and the variable valuations of these states.

A *computation* ϱ of a program is a finite or infinite sequence of subsequent transitions of the corresponding IOI machine:

$$\varrho = q_1 \xrightarrow{(a_1,b_1)} q_2 \xrightarrow{(a_2,b_2)} q_3 \xrightarrow{(a_3,b_3)} \cdots$$

The *label* of ϱ is the pair of finite or infinite words $(a_1a_2a_3\ldots,\ b_1b_2b_3\ldots) \in (\mathbb{B}^* \cup \mathbb{B}^\omega) \times (\mathbb{B}^* \cup \mathbb{B}^\omega)$. An *initial computation* starts at the unique entry state where all variables have the value 0. The *infinite behavior* $\langle\langle p \rangle\rangle$ of a program p is the set of infinite input/output sequences $(\alpha, \beta) \in \mathbb{B}^\omega \times \mathbb{B}^\omega$ that can be produced by an initial computation of p. Furthermore, we call a program *reactive* if all its initial computations can be extended to infinite computations that yield an infinite input and output sequence.

At any given time during a computation ϱ as above, the length of the input sequence $a_1a_2\ldots a_i$ and the output sequence $b_1b_2\ldots b_i$ might differ. The supremum of these length differences along a computation is called the *delay* of the computation. If the delay of a computation does not exceed a given bound $k \in \mathbb{N}$ then we call this computation *k-bounded*. A program is said to be k-bounded if all its computations are k-bounded. By restricting the infinite behavior of a program p to labels of k-bounded initial computations, we obtain the *k-bounded infinite behavior* $\langle\langle p \rangle\rangle_k$ of p.

3 Solving the Synthesis Problem Using Deterministic Tree Automata

The synthesis problem for structured reactive programs with bounded delay can be formulated as follows: Given an ω-regular specification $R \subseteq (\mathbb{B} \times \mathbb{B})^\omega$ representing the permissible input/output sequences, a finite set of Boolean variables B and a delay bound $k \in \mathbb{N}$, the task is to construct a structured reactive program p over B with k-bounded delay such that $\langle\langle p \rangle\rangle \subseteq R$ – or detect that no such program exists. (However, our results can easily be generalized to finite input and output alphabets other than $\mathbb{B}$ by allowing input and output statements that process multiple Boolean values as in [10].) In the following we assume that the specification R is provided in the form of a *nondeterministic Büchi automaton (NBA)* $\mathscr{A}_{\bar{R}}$ over the alphabet $\mathbb{B} \times \mathbb{B}$ that recognizes the complement of the specification, i.e., $\mathscr{L}(\mathscr{A}_{\bar{R}}) = (\mathbb{B} \times \mathbb{B})^\omega \setminus R$, which is always possible for ω-regular specifications.

Our synthesis procedure is based on the fact that programs can be viewed as trees. Figure 1 shows an example for a tree representation of a program. We use *deterministic bottom-up tree automata* (*DTAs*, see, for example, [15]) to recognize sets of programs. More specifically, we show the following theorem:

Theorem 1. *Let B be a finite set of Boolean variables, let $k \in \mathbb{N}$ and let $\mathcal{A}_{\bar{R}}$ be a nondeterministic Büchi automaton recognizing the complement of a specification $R \subseteq (\mathbb{B} \times \mathbb{B})^\omega$. We can construct a DTA that accepts a tree p iff p is a reactive program over B with k-bounded delay and $\langle\langle p \rangle\rangle \subseteq R$, such that the size of this DTA is doubly exponential in $|B|$ and k and exponential in the size of $\mathcal{A}_{\bar{R}}$.*

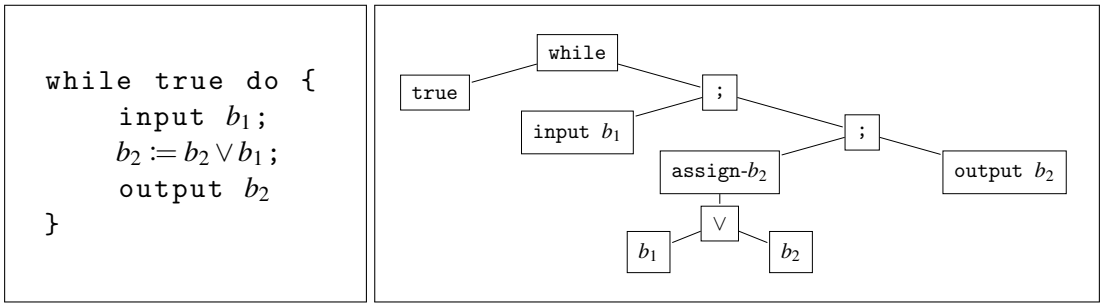

Figure 1: Example: A program and its tree representation.

An emptiness test on this DTA yields a solution to the synthesis problem. We obtain the desired tree automaton by intersecting three DTAs: The first DTA $\mathcal{B}_{\text{sat}}(B,k,\mathcal{A}_{\bar{R}})$ recognizes the set of programs over B whose k-bounded computations satisfy the specification R. That means, a program p is accepted iff $\langle\langle p \rangle\rangle_k \subseteq R$. The second DTA $\mathcal{B}_{\text{reactive}}(B)$ recognizes the reactive programs over B. Finally, we use a third DTA $\mathcal{B}_{\text{delay}}(B,k)$ to recognize the programs over B with k-bounded delay. We only consider the construction of $\mathcal{B}_{\text{sat}}(B,k,\mathcal{A}_{\bar{R}})$ here, as the other two DTAs can be constructed in a very similar way.

The DTA $\mathcal{B}_{\text{sat}}(B,k,\mathcal{A}_{\bar{R}})$ evaluates a given program p in a bottom-up manner, thereby assigning one of its states to each node of the program tree. The state reached at the root node must provide enough information to decide whether $\langle\langle p \rangle\rangle_k \subseteq R$, or equivalently, whether $\langle\langle p \rangle\rangle_k \cap \mathcal{L}(\mathcal{A}_{\bar{R}}) = \emptyset$. To that end, we are interested in the possible runs of $\mathcal{A}_{\bar{R}}$ on the input/output sequences generated by the program. Thus, we consider pairs of program computations and corresponding runs of $\mathcal{A}_{\bar{R}}$, which we call *co-executions*. Intuitively, $\mathcal{B}_{\text{sat}}(B,k,\mathcal{A}_{\bar{R}})$ inductively computes a representation of the possible co-executions of a given program and $\mathcal{A}_{\bar{R}}$. We define these representations, called *co-execution signatures*, in the following.

The beginning and end of a co-execution can be indicated by a valuation of the program variables and a state of $\mathcal{A}_{\bar{R}}$. However, we have to consider the following: The input sequence of a computation might be longer or shorter than its output sequence, but a run of $\mathcal{A}_{\bar{R}}$ only consumes input and output sequences of the same length. The suffix of the input/output sequence after the end of the shorter sequence, called the *overhanging suffix*, is hence still waiting to be consumed by $\mathcal{A}_{\bar{R}}$. Thus, we indicate the start and end of a co-execution by tuples of the form $\gamma = (\sigma, s, u, v)$, called *co-configurations*, where σ is a variable valuation, s is a state of $\mathcal{A}_{\bar{R}}$ and $(u,v) \in (\mathbb{B}^* \times \{\varepsilon\}) \cup (\{\varepsilon\} \times \mathbb{B}^*)$ is an overhanging suffix. Since we are only interested in k-bounded computations, we only consider co-configurations with $|u| \leq k$ and $|v| \leq k$. The set of these co-configurations for a given set of variables B and a given NBA $\mathcal{A}_{\bar{R}}$ is denoted by $CoCfg_k(B,\mathcal{A}_{\bar{R}})$.

A finite co-execution is called *complete* if the program terminates at the end of the computation. The *finite co-execution signature* $cosig^{\text{fin}}(p,\mathcal{A}_{\bar{R}},k)$ of a program p (with respect to $\mathcal{A}_{\bar{R}}$) is a relation consisting of tuples of the form (γ, f, γ') with $f \in \mathbb{B}$, which indicate that there exists a complete k-bounded co-execution that starts with the co-configuration γ and ends with γ' such that the corresponding run of

$\mathscr{A}_{\bar{R}}$ visits a final state iff $f = 1$. The *infinite co-execution signature* $cosig^{\infty}(p, \mathscr{A}_{\bar{R}}, k)$ of p is a set of co-configurations with $\gamma \in cosig^{\infty}(p, \mathscr{A}_{\bar{R}}, k)$ iff there exists an infinite k-bounded co-execution starting with γ such that the run of $\mathscr{A}_{\bar{R}}$ visits a final state infinitely often. We use pairs consisting of a finite and infinite co-execution signature as states of the DTA $\mathscr{B}_{\text{sat}}(B, k, \mathscr{A}_{\bar{R}})$. The size of the DTA is hence determined by the number of possible co-execution signatures, which is doubly exponential in the number of variables and k and exponential in the size of $\mathscr{A}_{\bar{R}}$. For a fixed k, this matches the complexity of Madhusudan's construction [10].

If σ_0 is the initial variable valuation (where all variables have the value 0) and s_0 is the initial state of $\mathscr{A}_{\bar{R}}$, then $(\sigma_0, s_0, \varepsilon, \varepsilon) \in cosig^{\infty}(p, \mathscr{A}_{\bar{R}}, k)$ iff there is an initial k-bounded computation of p such that some corresponding run of $\mathscr{A}_{\bar{R}}$ visits a final state infinitely often, so $cosig^{\infty}(p, \mathscr{A}_{\bar{R}}, k)$ is indeed sufficient to decide whether $\langle\langle p \rangle\rangle_k \subseteq R$. It remains to be shown that the co-execution signatures can be computed inductively. Exemplarily, we consider the case of programs of the form $p =$ "`while` e `do` p_1". First, we construct a representation $cosig_e^*(p_1, \mathscr{A}_{\bar{R}}, k)$ of all finite sequences of consecutive co-executions of p_1 that are compatible with the loop condition e. To that end, we consider only those tuples (γ, f, γ') in $cosig^{\text{fin}}(p_1, \mathscr{A}_{\bar{R}}, k)$ where the variable valuation in γ satisfies the loop condition e, and compute the reflexive transitive closure of the resulting relation. Formally, we have $cosig_e^*(p_1, \mathscr{A}_{\bar{R}}, k) = closure(C)$ with $C = \left\{ ((\sigma, s, u, v), f, \gamma') \in cosig^{\text{fin}}(p_1, \mathscr{A}_{\bar{R}}, k) \mid \sigma \in [\![e]\!] \right\}$. Here, $[\![e]\!]$ denotes the set of variable valuations that satisfy e, and $closure(C)$ is the smallest relation $D \subseteq CoCfg_k(B, \mathscr{A}_{\bar{R}}) \times \mathbb{B} \times CoCfg_k(B, \mathscr{A}_{\bar{R}})$ such that

- $(\gamma, 0, \gamma) \in D$ for all $\gamma \in CoCfg_k(B, \mathscr{A}_{\bar{R}})$, and
- $(\gamma, f_1, \gamma') \in D$, $(\gamma', f_2, \gamma'') \in C$ implies $(\gamma, \max\{f_1, f_2\}, \gamma'') \in D$.

Using $cosig_e^*(p_1, \mathscr{A}_{\bar{R}}, k)$, the co-execution signatures for p can be computed by the following reasoning: A finite co-execution of $p =$ "`while` e `do` p_1" (and $\mathscr{A}_{\bar{R}}$) can be decomposed into a finite sequence of co-executions of p_1. An infinite co-execution of p can either eventually stay inside a loop iteration forever or traverse infinitely many iterations. It can therefore be decomposed either into a finite sequence of co-executions of p_1 followed by an infinite co-execution of p_1, or into a finite sequence of co-executions of p_1 followed by a cycle of co-executions of p_1, leading back to a previous co-configuration. Thus, we obtain the following formal representation of the co-execution signatures for p:

- $(\gamma, f, (\sigma', s', u', v')) \in cosig^{\text{fin}}(p, \mathscr{A}_{\bar{R}}, k)$ iff $(\gamma, f, (\sigma', s', u', v')) \in cosig_e^*(p_1, \mathscr{A}_{\bar{R}}, k)$ and $\sigma' \notin [\![e]\!]$.
- $\gamma \in cosig^{\infty}(p, \mathscr{A}_{\bar{R}}, k)$ iff at least one of the following holds:
  - There exist $\gamma' = (\sigma', s', u', v') \in CoCfg_k(B, \mathscr{A}_{\bar{R}})$ and $f \in \mathbb{B}$ such that $(\gamma, f, \gamma') \in cosig_e^*(p_1, \mathscr{A}_{\bar{R}}, k)$, $\sigma' \in [\![e]\!]$ and $\gamma' \in cosig^{\infty}(p_1, \mathscr{A}_{\bar{R}}, k)$.
  - There exist $\gamma' = (\sigma', s', u', v') \in CoCfg_k(B, \mathscr{A}_{\bar{R}})$ and $f \in \mathbb{B}$ such that $(\gamma, f, \gamma') \in cosig_e^*(p_1, \mathscr{A}_{\bar{R}}, k)$, $\sigma' \in [\![e]\!]$ and $(\gamma', 1, \gamma') \in cosig_e^*(p_1, \mathscr{A}_{\bar{R}}, k)$.

4 Lower Bound for the Size of the Tree Automata

We show the following lower bound for the size of any nondeterministic tree automaton (NTA) recognizing the desired set of programs:

Theorem 2. *Let B be a set of n Boolean variables, let $k \in \mathbb{N}$ and let $R \subseteq (\mathbb{B} \times \mathbb{B})^{\omega}$ be a specification that is realizable by some program over B with k-bounded delay. Let $\mathscr{C}$ be an NTA that accepts a tree p iff p is a reactive program over B with k-bounded delay and $\langle\langle p \rangle\rangle \subseteq R$. Then $\mathscr{C}$ has at least $2^{2^{n-1}}$ states.*

For a sketch of the proof, consider a set of Boolean variables $B = \{b_1, \ldots, b_n\}$. There are $2^{2^{n-1}}$ functions of the type $\mathbb{B}^{n-1} \to \mathbb{B}$. Each of these functions can be implemented by a program that checks the values of $b_1, \ldots, b_{n-1}$ and sets b_n to the corresponding function value. An NTA as in Theorem 2 must be able to distinguish all of these programs. Otherwise, let p_i and p_j be two such programs that cannot be distinguished by the NTA. We could then construct a program that satisfies the specification and contains p_i as a subprogram, but runs into a non-reactive infinite loop if this subprogram is replaced by p_j. The NTA would accept both variants, including the non-reactive program, which contradicts the premise.

5 Conclusion

The contributions of this paper are threefold, advancing the study of structured reactive programs: We introduced a formal semantics for structured reactive programs in the sense of [10]. Furthermore, we presented a new synthesis algorithm for structured reactive programs with bounded delay, using the elementary concept of deterministic bottom-up tree automata. Finally, we showed a lower bound for the size of any nondeterministic tree automaton that recognizes the set of specification-compliant programs, emphasizing the importance of choosing a small yet still sufficient set of program variables. Estimating the number of Boolean variables that are needed to realize a given specification is a major open problem. While [13] implies an exponential upper bound for the required number of variables in the case of LTL specifications, a corresponding lower bound is still to be determined.

Acknowledgments. The author would like to thank Wolfgang Thomas for his helpful advice and Marcus Gelderie for fruitful discussions.

References

[1] Benjamin Aminof, Fabio Mogavero & Aniello Murano (2012): *Synthesis of Hierarchical Systems*. In Farhad Arbab & Peter Csaba Ölveczky, editors: *Formal Aspects of Component Software*, *Lecture Notes in Computer Science* 7253, Springer Berlin Heidelberg, pp. 42–60, doi:10.1007/978-3-642-35743-5_4.

[2] J. Richard Büchi & Lawrence H. Landweber (1969): *Solving Sequential Conditions by Finite-State Strategies*. *Transactions of the American Mathematical Society* 138, pp. 295–311, doi:10.2307/1994916.

[3] Roderick Bloem, Stefan Galler, Barbara Jobstmann, Nir Piterman, Amir Pnueli & Martin Weiglhofer (2007): *Specify, Compile, Run: Hardware from PSL*. *Electronic Notes in Theoretical Computer Science* 190(4), pp. 3 – 16, doi:10.1016/j.entcs.2007.09.004.

[4] Rüdiger Ehlers (2010): *Symbolic Bounded Synthesis*. In Tayssir Touili, Byron Cook & Paul Jackson, editors: *Computer Aided Verification*, *Lecture Notes in Computer Science* 6174, Springer Berlin Heidelberg, pp. 365–379, doi:10.1007/978-3-642-14295-6_33.

[5] Marcus Gelderie (2012): *Strategy Machines and Their Complexity*. In Branislav Rovan, Vladimiro Sassone & Peter Widmayer, editors: *Mathematical Foundations of Computer Science 2012*, *Lecture Notes in Computer Science* 7464, Springer Berlin Heidelberg, pp. 431–442, doi:10.1007/978-3-642-32589-2_39.

[6] Marcus Gelderie & Michael Holtmann (2011): *Memory Reduction via Delayed Simulation*. In Johannes Reich & Bernd Finkbeiner, editors: *iWIGP*, *EPTCS* 50, pp. 46–60, doi:10.4204/EPTCS.50.4.

[7] Michael Holtmann, Lukasz Kaiser & Wolfgang Thomas (2010): *Degrees of Lookahead in Regular Infinite Games*. In Luke Ong, editor: *Foundations of Software Science and Computational Structures*, *Lecture Notes in Computer Science* 6014, Springer Berlin Heidelberg, pp. 252–266, doi:10.1007/978-3-642-12032-9_18.

[8] Orna Kupferman & Moshe Y. Vardi (1999): *Church's Problem Revisited*. *The Bulletin of Symbolic Logic* 5(2), pp. 245–263, doi:10.2307/421091.

[9] Yoad Lustig & Moshe Y. Vardi (2009): *Synthesis from Component Libraries*. In Luca Alfaro, editor: *Foundations of Software Science and Computational Structures*, *Lecture Notes in Computer Science* 5504, Springer Berlin Heidelberg, pp. 395–409, doi:10.1007/978-3-642-00596-1_28.

[10] Parthasarathy Madhusudan (2011): *Synthesizing Reactive Programs*. In Marc Bezem, editor: *Computer Science Logic (CSL'11) - 25th International Workshop/20th Annual Conference of the EACSL*, *Leibniz International Proceedings in Informatics (LIPIcs)* 12, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, pp. 428–442, doi:10.4230/LIPIcs.CSL.2011.428.

[11] Amir Pnueli & Roni Rosner (1989): *On the Synthesis of a Reactive Module*. In: *POPL*, pp. 179–190. Available at `http://doi.acm.org/10.1145/75277.75293`.

[12] Michael Oser Rabin (1972): *Automata on Infinite Objects and Church's Problem*. American Mathematical Society, Boston, MA, USA.

[13] Roni Rosner (1992): *Modular Synthesis of Reactive Systems*. Ph.D. thesis, Weizmann Institute of Science.

[14] Sven Schewe & Bernd Finkbeiner (2007): *Bounded Synthesis*. In Kedar S. Namjoshi, Tomohiro Yoneda, Teruo Higashino & Yoshio Okamura, editors: *Automated Technology for Verification and Analysis*, *Lecture Notes in Computer Science* 4762, Springer Berlin Heidelberg, pp. 474–488, doi:10.1007/978-3-540-75596-8_33.

[15] Wolfgang Thomas (1997): *Languages, Automata, and Logic*. In Grzegorz Rozenberg & Arto Salomaa, editors: *Handbook of Formal Languages*, Springer Berlin Heidelberg, pp. 389–455, doi:10.1007/978-3-642-59126-6_7.