Synthesis of Reactive Systems

Bernd Finkbeiner
Universität des Saarlandes

Abstract. These lecture notes trace the developments triggered by Church's classic synthesis problem from the early solutions in the 1960s to the practical tools that have come out in the past few years. The article gives an overview on the automata- and game-theoretic foundations of the synthesis problem. We then explore the spectrum of logics for the synthesis of reactive systems, from reduced logics, like GR(1), to advanced logics such as strategy and coordination logic. Finally, we discuss the ideas behind recent synthesis approaches, like bounded synthesis and symbolic and SAT-based methods.

Keywords. Church's problem, synthesis, reactive systems, automata over infinite words and trees, S1S, temporal logic, GR(1), strategy logic, coordination logic, incomplete information, synthesis of distributed systems, bounded synthesis, symbolic synthesis, SAT-based synthesis

1. Introduction

More than fifty years after its introduction by Alonzo Church [1], the synthesis problem is still one of the most intriguing challenges in the theory of reactive systems. Reactive systems are computer systems that maintain a continuous interaction with their environment. Hardware circuits, communication protocols, and embedded controllers are typical examples. Synthesis algorithms construct such systems automatically from logical specifications, such as formulas of a temporal logic. Because synthesis eliminates the need for a manual implementation, it has the potential to revolutionize the development process for reactive systems. And indeed, synthesis has, over the past few years, found applications in several areas of systems engineering, notably in the construction of circuits and device drivers and in the synthesis of controllers for robots and manufacturing plants. The quest, however, is far from over: the performance of the available algorithms still leaves much to be desired. Finding logics and algorithms that make the synthesis of reactive systems efficient and scalable remains a vigorously pursued research goal.

Church's problem statement in 1957 ignited research on several fundamental topics, notably on the connection between logics and automata, on algorithmic solutions of infinite games over finite graphs, and on the theory of automata over infinite objects. The basic underlying insight is that the synthesis problem can be understood as a game between the system, trying to satisfy the specification, and the environment, trying to expose an error. A winning strategy for the system player defines an implementation that is guaranteed to satisfy the specification. The games that result from formulas of the usual specification logics for reactive

systems are infinite in the sense that the plays have infinite duration, and yet finite in the sense that the games are played on a finite game arena. These games can be solved algorithmically, i.e., one can determine which player wins the game, and produce a winning strategy; the winner is guaranteed to have a strategy that only requires finite memory; and the winning strategies form a regular set, which can be recognized and manipulated with automata over infinite trees.

Progress in automata and game theory allowed Church's problem to be solved within a decade, in the sense that the decidability of the problem was established [2]. The path towards practical algorithms, however, turned out to be much longer. The main challenge to be overcome was the algorithmic complexity of the problem. In his 1974 thesis, Larry Stockmeyer discussed one of the corner stones of the solution of Church's problem, the translation of formulas of the monadic second-order logic of one successor (S1S) into Büchi automata. The translation from S1S to automata is nonelementary in the length of the formula. Stockmeyer predicted, somewhat darkly, that "any attempt to find an efficient algorithm for the problem is foredoomed" [3].

The expensive translation from formulas to automata was not to remain the only obstacle towards practical solutions of Church's problem. A very discouraging result was Rosner and Pnueli's discovery of the undecidability of the synthesis problem for distributed systems [4]. Most modern reactive systems are distributed in the sense that they consist of multiple processes with individual inputs, which they may or may not share with other processes. A key difficulty in the design of such systems is to decide how the processes should interact so that each process obtains the information needed to carry out its functionality. *Distributed synthesis*, i.e., the extension of Church's problem to the case that the implementation consists of several processes, is thus a particularly useful type of synthesis. However, when Pnueli and Rosner began, in the late 1980s, to investigate Church's problem for distributed systems, they quickly discovered that the problem is undecidable in even the most restricted settings, such as architectures with as few as *two* independent processes.

Today, reactive synthesis has matured, despite these challenges, into an area with not only well-understood foundations, but also a rich supply of practical algorithms. There is a growing landscape of tool implementations (cf. [5,6,7]). In 2014, the first *synthesis competition* took place at the annual CAV conference, where the synthesis tools were compared against each other using an initial collection of roughly 500 standard benchmark problems [8]. While synthesis may not yet have reached the same level of industrial acceptance as its twin brother, computer-aided verification, there is a growing number of impressive success stories from areas such as hardware circuits, device drivers, and robotics.

The historical developments on the synthesis problem can broadly be grouped into three big waves. The first wave, during the decade after Church's inception of the problem, brought an initial set of basic algorithms that established the principal decidability. In 1969, Büchi and Landweber gave the first game-theoretic solution [2]. Michael Rabin invented automata over infinite trees and provided an automata-theoretic approach to the solution of Church's problem [9], which paved the way for more advanced automata-based algorithms to be discovered later. The common basis for these early synthesis algorithms was a system specification

given as a formula of monadic second order logic (MSO). As a result, the early algorithms suffer, as observed by Stockmeyer, from nonelementary complexity.

The focus on MSO was disrupted in the 1980s, when the introduction of temporal logic for the specification of reactive systems by Amir Pnueli triggered a second wave of interest in Church's problem. The translation of linear-time temporal logic (LTL) to deterministic automata is doubly exponential, and, hence, while far from cheap, certainly much closer to practicality than MSO. At the same time, LTL-based model checking, which only requires PSPACE, took off as an industrial technique, in particular in the hardware industry. The need to detect unrealizable, and, hence, erroneous, specifications before verification — "a specification is useless if it cannot be realized by any concrete implementation" [10] — provided additional, practical motivation for the study of Church's problem. Synthesis algorithms were developed for the common linear and branching-time temporal logics.

The algorithmic advances led to broader interest in realizability and similar questions. Properties about the existence of strategies, including realizability and related properties, like the existence of Nash equilibria or recoverability from faults, began to be recognized as genuine system properties worthy of formal specification and analysis. Game-based logics like ATL [11], Strategy Logic [12], and Coordination Logic [13] extend temporal logic with quantification over strategies, and can express properties like realizability *within* the logic, as opposed to an external semantical condition, like the realizability of an MSO or LTL formula. The result is much greater flexibility: customized queries, such as realizability under specific assumptions or in specific system architectures, can be checked without the need to come up, every time, with customized synthesis algorithms. Once the desired variation is encoded in an appropriate logic, the actual synthesis work can be left to the model checking algorithm or decision procedure of the logic.

A third wave of inquiry into Church's problem, this time with the explicit goal of developing practical algorithms, began about a decade ago. The stage for the third wave had been set by the enormous advances in the performance of automatic verification during the 1980s and 1990s. To reduce the complexity of the synthesis problem, much attention focussed on "synthesis-friendly" reduced logics, such as *generalized reactivity GR(1)*, and on practically relevant variations of the basic synthesis problems, such as *bounded synthesis* [14]. Bounded synthesis focusses the search for an implementation to implementations with small (bounded) size. Motivated by the success of symbolic and SAT-based verification (cf. [15,85,86]), similar techniques were also developed for synthesis. Eventually, the advances in the performance of the synthesis algorithms and the growing availability of tools made it possible to tackle the first real-world design problems, such as the synthesis of an arbiter for the *AMBA AHB bus*, an open industrial standard for the on-chip communication and management of functional blocks in system-on-a-chip (SoC) designs [16] and the synthesis of device drivers like the Intel PRO/1000 ethernet controller [17].

2. Church's Problem

Church's problem [1,18] is concerned with the existence of *reactive systems*, which transform, in an online fashion, an infinite stream of inputs into an infinite stream of outputs. Reactive systems differ fundamentally from data transforming programs, which read some input and produce, upon termination, some output. The synthesis of data transforming programs is a fascinating subject with a rich history of its own (cf. [19,20,21]), which is, however, beyond the scope of this article.

In Church's problem, the *inputs* and *outputs* of the reactive system to be synthesized are valuations of boolean variables. Consider a set I of input variables and a set O of output variables; the reactive system must, given a finite sequence of valuations of the input variables $w \in (2^I)^*$ produce a valuation $f(w) \in 2^O$ of the output variables. A *specification* of a reactive system is a set $Spec \subseteq (2^{I \cup O})^\omega$ of *infinite* sequences of valuations of both the input and the output variables. The system is *correct* if for every infinite sequence of inputs, the input-output sequence that is obtained by computing the outputs according to f, is an element of $Spec$.

In the following, we make the definition of Church's problem precise, by fixing monadic second-order logic as the specification language and finite-state machines as implementations.

Monadic Second-Order Logic The input to Church's problem is a regular set of ω-words, given as a formula of the monadic second-order logic of one successor (S1S). Suppose, for example, we are interested in constructing an arbiter circuit. Arbiters are used when more than one client circuit needs access to some shared resource, such as a communication bus. To access the resource, the client sends a *request* signal R and waits until it receives a *grant* signal G from the arbiter. The task of the arbiter is to answer each request with a grant without giving grants to the two clients at the same time. In S1S, an arbiter with two clients can be specified as a conjunction of three properties:

$$\begin{aligned} &\forall x \,.\, \neg G_1(x) \vee \neg G_2(x) && \text{(mutual exclusion)} \\ &\forall x \,.\, R_1(x) \rightarrow \exists y \,.\, y \geq x \,\wedge\, G_1(y) && \text{(response 1)} \\ &\forall x \,.\, R_2(x) \rightarrow \exists y \,.\, y \geq x \,\wedge\, G_2(y) && \text{(response 2)} \end{aligned}$$

The *mutual exclusion* property states that at every point in time x, at most one grant signal can be set; the *response* properties state that if a request is made at time x, then there must exist a point in time $y \geq x$, where the corresponding grant signal is set. S1S formulas are based on two types of variables: first-order variables V_1, which range over natural numbers, and second-order variables V_2, which range over *sets* of natural numbers. We will distinguish the two types of variables here by using small letters $V_1 = \{z, y, x, \ldots\}$ for the first-order variables and capital letters $V_2 = \{Z, Y, X, \ldots\}$ for second-order variables. Terms are constructed from first-order variables, the constant 0, and the successor function: $t + 1$; formulas consist of the membership test $X(t)$, where t is a term and X a second-order variable, equality $t_1 = t_2$ on terms, first-order quantification $\exists x$ and second-order quantification $\exists X$, and boolean combinations. Greater-than-or-equal $x \geq y$ is

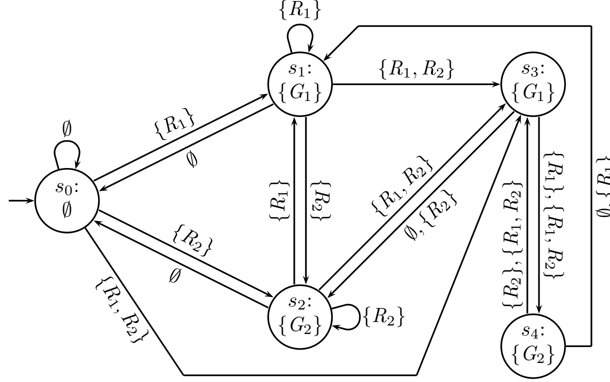

Figure 1. Moore machine implementing the arbiter specification.

definable in S1S as the requirement that every upward-closed set Z that contains y must also contain x: $\forall Z\ .\ (Z(y) \wedge (\forall y'\ .\ Z(y') \rightarrow Z(y'+1))) \rightarrow x \in Z$.

A model of an S1S formula can be interpreted as an infinite word from $(2^{V_1 \cup V_2})^\omega$, where each letter is a set of variables. A second-order variable evaluates to the set of positions where it occurs, first-order variables must occur at exactly one position and evaluate to this position. To specify the behavior of a reactive system in S1S, we use free second-order variables that are interpreted as the values of input and output signals at various points in time. In the arbiter example, R_1, R_2 are second-order variable representing the input, G_1, G_2 second-order variables representing the output.

Implementations. A solution to Church's problem is a circuit that satisfies the S1S formula for every possible input. The restriction to circuits means that we are interested in finite-state solutions, which we will formalize in the following as finite-state machines.

A *(finite-state) Moore machine* over input alphabet Σ and output alphabet Λ has the form $\mathcal{M} = (S, s_0, T, G)$, where S is a finite set of *states*, $s_0 \in S$ is an *initial state*, $T : S \times \Sigma \rightarrow S$ is a transition function, mapping a state and an input letter to the next state, and $G : S \rightarrow \Lambda$ is an *output function* mapping each state to an output letter.

For a given sequence of inputs $\alpha = \alpha(0)\,\alpha(1)\,\alpha(2)\ldots$, the Moore machine generates a sequence $\beta = s_0\, T(\beta(0), \alpha(0))\, T(\beta(1), \alpha(1)) \ldots$ of states and a sequence $\gamma = G(\beta(0))\, G(\beta(1))\, G(\beta(2)) \ldots$ of outputs.

For Church's problem, the input alphabet $\Sigma = 2^I$ consists of the valuations of the input variables I, and the output alphabet $\Lambda = 2^O$ consists of the valuations of the output variables O. The Moore machine is a realization of the S1S formula φ if for all possible inputs $\alpha \in (2^I)^\omega$, the combination $\alpha(0)\cup\gamma(0)\ \alpha(1)\cup\gamma(1)\ \alpha(2)\cup \gamma(2)\ldots$ satisfies φ.

Figure 1 shows a Moore machine that implements the arbiter specification. This implementation carefully answers every request with a grant at a later point in time. Note that this implementation is actually unnecessarily complicated. If a request comes in at a point in time where a grant is already being given out, there

is, according to our specification, no need for a further grant at a later point in time. Also, there is no requirement in our specification that grants must actually be preceded by requests. Another, and much simpler, solution for our specification would be to completely ignore the input and alternate between giving a grant to the first and to the second client, independently of whether there was a request or not.

3. Early Solutions

In 1960, Büchi [22] and Elgot [23] established the connection between logic and automata, by showing that formulas of monadic second-order logic over finite words and finite automata can be translated into each other. *Büchi's Theorem* [24] extends this connection to S1S and automata over infinite words. Büchi's Theorem establishes the decidability of S1S and also provides the first step for the solution of Church's problem: the translation of the logical specification to an automaton. This first step is common to both the game-based solution due to Büchi and Landweber, and the automata-based solution due to Rabin.

3.1. The Logic-Automata Connection

A (nondeterministic) *automaton* over infinite words over alphabet Σ has the form $\mathcal{A} = (Q, Q_0, T, Acc)$, where Q is a finite set of *states*, $Q_0 \subseteq Q$ is a set of *initial states*, $T \subseteq Q \times \Sigma \times Q$ is a set of *transitions*, and Acc is an *acceptance condition.* We assume that $\mathcal{A}$ is *complete*, i.e., for every $q \in Q$ and $\sigma \in \Sigma$ there is at least one $q' \in Q$ such that $(q, \sigma, q) \in T$. If, furthermore, Q_0 is singleton and for every $q \in Q$ and $\sigma \in \Sigma$ there is at most one $q' \in Q$ such that $(q, \sigma, q) \in T$, then $\mathcal{A}$ is *deterministic.* In this case, we also write T as a function $T : Q \times \Sigma \to Q$.

The language of automata over infinite words is defined with the following mechanism. A *run* of an automaton $\mathcal{A}$ on an infinite input word $\alpha = \alpha(0)\,\alpha(1)\,\alpha(2)\ldots \in \Sigma^\omega$ is an infinite sequence of states $r = r(0)\,r(1)\,r(2)\ldots \in Q^\omega$ such that the sequence starts with the initial state $r(0) = q_0$ and subsequently follows the transitions, i.e., for all $i \in \mathbb{N}$, $(r(i), \alpha(i), r(i{+}1)) \in T$. An automaton $\mathcal{A}$ *accepts* an infinite word α if there is a run of $\mathcal{A}$ on α that satisfies the acceptance condition. The *language* of $\mathcal{A}$ consists of all accepted words.

A basic acceptance condition is *safety*, which is given as a subset of $S \subseteq Q$ of *states*; a run is accepting if only safe states are visited. For the automata-theoretic representation of MSO it suffices to use a comparatively simple extension of the safety condition, the Büchi condition [24], which is given as a subset $F \subseteq S$ of *accepting states.* The Büchi condition requires that some state in F is visited infinitely often.

There are several other useful acceptance conditions. The *co-Büchi* condition is given as a set of *rejecting* states F and is satisfied if the states in F are only visited finitely often. The *Muller acceptance condition* consists of a set $\mathcal{F} \subset 2^Q$ of sets of states. A run is accepting if the set of states that occur infinitely often on the run is an element of $\mathcal{F}$. The *Rabin* condition consists of a set $\Omega = \{(E_1, F_1), \ldots, (E_k, F_k)\}$ of pairs of sets of states. A run is accepted if there

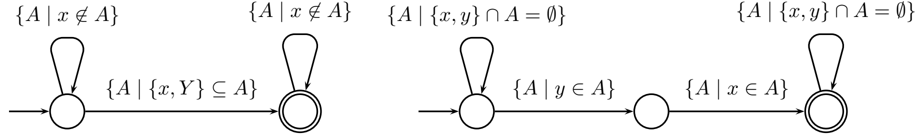

Figure 2. Büchi automata for S1S formulas. The automaton on the left accepts the models of the S1S formula $Y(x)$, the automaton on the right accepts the models of $x = y + 1$.

exists a pair $(E, F) \in \Omega$ such that none of the states in E occurs infinitely often and some state in F occurs infinitely often. A commonly used special case of the Rabin condition is the *Rabin chain* or *parity condition* [25]. Here, the acccepting pairs $(E_1, F_1), \ldots, (E_k, F_k)$ form a chain $E_1 \subset F_1 \subset E_2 \subset \ldots \subset E_k \subset F_k$ with respect to set inclusion. The states can thus be colored with natural numbers, where the states of E_1 are colored with 1, the states of $F_1 \smallsetminus E_1$ with 2, and so forth. A run is accepting if the least color that occurs infinitely often is even.

The construction of a Büchi automaton from an S1S formula follows the structure of the formula. The base cases are atomic formulas like $Y(x)$ and $x = y + 1$, which are translated into the automata shown in Fig. 2.

Automata for more complex formulas are built by applying automata transformations corresponding to the logical operators, i.e., disjunction is implemented by language union, conjunction by intersection, quantification by projection, negation by complementation. Language complementation is exponential in the number of states; the complexity of this translation is therefore, in general, nonelementary.

3.2. The Büchi-Landweber Theorem

In 1969, one decade after its inception, Church's problem was solved by Büchi and Landweber [2]. Büchi and Landweber stated Church's problem as a game between two players, one player representing the system and one player representing the environment. In each round of the game, the environment player first chooses a valuation of the inputs, then the system chooses the valuation of the outputs. The system player wins if the sequence of valuations produced in this way satisfies the given S1S formula. Realizability thus amounts to the existence of a winning strategy for the system player against all possible behaviors of the environment player.

Determinization. Büchi and Landweber's construction is based on the connection between logic and automata provided by Büchi's Theorem. Instead of checking the original S1S formula on every possible play, the game is played directly on a finite graph that is constructed from the the states of an automaton obtained from the formula.

The automata in Büchi's Theorem are, in general, nondeterministic, i.e., the set of initial states may consist of more than one state and there may be two transitions $(q, \sigma, q_1), (q, \sigma, q_2) \in T$ from the same source state q that lead, for some input symbol σ, to two different target states $q_1 \neq q_2$. This is a problem for the

construction of the game, because every sequence of choices of the two players must result in a unique play. An important preprocessing step, before the game can be constructed, is therefore to determinize the automaton.

Unlike automata over finite words, where each nondeterministic automaton can be translated into an equivalent deterministic automaton, deterministic Büchi automata are strictly weaker than nondeterministic Büchi automata. The determinization of a Büchi automaton therefore results in automata with more expressive acceptance conditions. Already in 1966, McNaughton [26] showed that every nondeterministic Büchi automaton can be translated into an equivalent deterministic automaton with Muller acceptance condition. In 1988, Safra gave a translation that produces a deterministic Rabin automaton with only $2^{O(n \log n)}$ states and $O(n)$ pairs in the acceptance condition, where n is the number of states of the nondeterministic automaton [27]. Michel showed that Safra's construction is in fact optimal [28]. More recently, Piterman gave an adaptation of Safra's construction that translates nondeterministic Büchi automata into deterministic parity automata [29].

The synthesis game. The treatment of Church's problem as an infinite game was first proposed by McNaughton [30].

A *game arena* is a graph $\mathcal{A} = (V, E)$, where the nodes V are partitioned into two disjoint sets $V = V_0 \cup V_1$, the *positions* of players 0 and 1, respectively. The edges E are a subset of $V \times V$ such that every every position $p \in V$ has at least one outgoing edge $(p, p') \in E$. A *game* $\mathcal{G} = (\mathcal{A}, \mathit{Win})$ consists of a game arena and a *winning condition* $\mathit{Win} \subseteq V^\omega$, given for example as a Büchi, Muller, or parity condition.

In the *synthesis game* of an S1S formula φ, Player 0 represents the system, Player 1 the environment. A round of the game consists of Player 0 choosing a valuation of the output variables, then Player 1 choosing a valuation of the input variables. The positions of the game keep track of the state of the deterministic automaton $\mathcal{A}_\varphi = (Q, \{q_0\}, T, \mathit{Acc})$ for φ: for Player 0, we set $V_0 = Q$. For Player 1, we set $V_1 = Q \times 2^O$. The game position of Player 1 stores the output selected by Player 0; a position p of Player 0 has edges to all positions (p, o) of Player 1 for some $o \in 2^O$. Player 1 then chooses some input and the game moves to the successor state of the automaton. A position (p, o) of Player 1 thus has edges to all positions $T(p, i \cup o)$ of Player 0, where $i \in 2^I$ is some input. The winning condition is derived from the acceptance condition of the automaton. For example, if the acceptance condition of the deterministic automaton is a parity condition $c : Q \to \mathbb{N}$, then the winning condition of the game is the parity condition $c' : Q \cup (Q \times 2^O) \to \mathbb{N}$ with $c'(q) = c(q)$ for $q \in Q$ and $c'(q, o) = c(q)$ for $q \in Q, o \in 2^O$.

A *play* $\pi \in V^\omega$ is an infinite sequence $\pi = \pi(0)\,\pi(1)\,\pi(2)\ldots$ of positions such that $\forall i \in \mathbb{N}\ .\ (\pi(i), \pi(i+1)) \in E$. The players take turns in choosing successor positions, i.e., $\pi(1)$ is chosen by Player 0 from the available successors of $\pi(0)$ in E, then $\pi(2)$ is chosen by Player 1, etc. A play π is won by Player 0 iff π satisfies the winning condition. A *strategy* for player σ is a function $f_\sigma : V^* \cdot V_\sigma \to V$ that maps a sequence of game positions, representing a history of a play, to a successor position such that $(p, p') \in E$ whenever $f_\sigma(u \cdot p) = p'$. A play π *conforms to* a strategy f_σ of player σ if $\forall i \in \mathbb{N}$. if $p_i \in V_\sigma$ then $\pi(i+1) = f_\sigma(\pi(0)\ldots\pi(i))$.

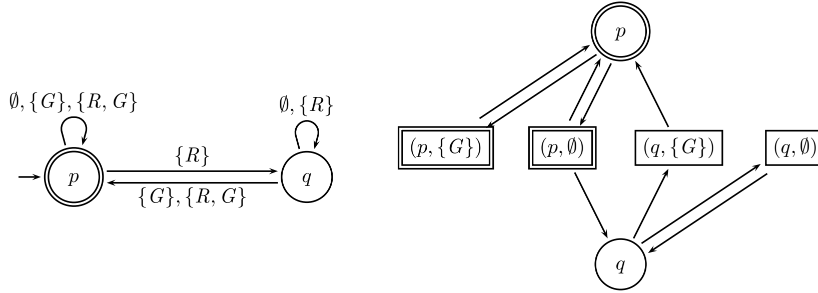

Figure 3. Deterministic Büchi automaton and synthesis game for the response property. Game positions depicted as circles belong to Player 0, game positions depicted as rectangles to Player 1. Double lines indicate that the game position or automaton state is accepting.

Strategies that do not depend on the history of the game except for the last position can be given as a function $g_\sigma : V_\sigma \to V$ (defining the strategy $f_\sigma(w \cdot v) = g_\sigma(v)$ for $w \in V^*$) and are called *memoryless.* A strategy f_σ is *p-winning* for player σ and position p if all plays that conform to f_σ and that start in p are won by Player σ.

In the synthesis game of an S1S formula φ, Player 0 has a q_0-winning strategy if and only if φ is realizable. As an example, consider the deterministic automaton and the synthesis game shown in Fig. 3 for the response property:

$$\forall x \ . \ R(x) \to \exists y \ . \ y \geq x \ \wedge \ G(y) \text{ (response)}$$

Note that this example is chosen so that the acceptance condition of the deterministic automaton, and, hence, the winning condition of the game, is a Büchi condition. In general, the result of the determinization would be an automaton with a more expressive acceptance condition such as parity.

Player 0 has a p-winning strategy. For example, if Player 0 always gives out the grant, the game moves back and forth between p and $(p, \{G\})$, without ever reaching q.

Game Solving. The *Büchi-Landweber Theorem* [2] says that for a game with Muller winning condition, one can decide whether Player 0 has a winning strategy and, if the answer is yes, construct a finite-state winning strategy. Starting with this fundamental result, the detailed analysis of the game solving problem, in particular with regard to the size of the required strategies and the complexity of determining the winning player, evolved into a research area that still flourishes today.

Gurevich and Harrington showed that the memory needed to win a Muller game can be limited to store a *latest appearance record*, which indicates the order in which the positions of the game arena were visited most recently [31]. The memory required for the latest appearance records is bounded by the factorial of the number of positions. A corresponding exponential lower bound for the memory required to win Muller games was shown by Lescow [32]. The situation is simpler for games with Rabin winning condition, where memoryless winning strategies suffice for Player 0 [33], although Player 1 may need exponential memory [32].

For parity games (and games with subsumed winning conditions like safety and Büchi), the winning player always has a memoryless winning strategy [34].

The complexity of determining the winner of a Muller game depends on the precise representation of the winning condition. For an explicit representation of the winning condition $\mathcal{F}$ as a list of sets of states, the winner can be determined in polynomial time [35]. For succinct representations, such as the *Emerson-Lei* representation [36], where $\mathcal{F}$ is given as a boolean formula whose variables are the positions of the game, the problem is PSPACE-complete [37].

Safety games can be solved in linear time in the size of the arena [38,39]; Büchi games in quadratic time in the number of positions [40]. Deciding the winner of a Rabin game is NP-complete [41]. An intriguing open question is the complexity of solving parity games. The problem is known to be in NP $\cap$ co-NP [42] (and in UP $\cap$ co-UP [43]). All currently known deterministic algorithms have complexity bounds that are either exponential in the number of colors [44,45,46], or in the square root of the number of game positions [47,48]. For some time, strategy improvement algorithms [47,49] were believed to be promising candidates for a polynomial-time solution; recently, however, a family of games with exponential running time was demonstrated for this class of algorithms as well [50].

A winning strategy for Player 0 in the synthesis game can be translated into an implementation. Suppose, for example, that the synthesis game is a parity game, derived from a deterministic parity automaton $\mathcal{A} = (Q, \{q_0\}, T, c)$, and a memoryless winning strategy $f_0 : V_0 \to V$. The strategy is implemented by the Moore machine $\mathcal{M} = (S, s_0, T', G)$, where $S = Q$, $s_0 = q_0$, $T'(q, i) = T(q, f_0(q) \cup i)$, and $G(q) = f_0(q)$, which always chooses the next output according to f_0.

3.3. Automata-Based Synthesis

In 1969, Rabin introduced automata over infinite trees [51]. Tree automata provide an alternative, and very elegant, solution to Church's problem [9]. Rabin's insight was to view implementations as infinite trees that branch according to the possible inputs and that are labeled with the outputs chosen by the system. Figure 4 shows such a tree representation for a strategy $f : (2^{\{i\}})^* \to 2^O$ with a single input variable i. Rabin's solution to Church's problem is to represent the set of all implementations that satisfy the specification as a tree automaton. The specification is thus realizable if and only if the language of the tree automaton is non-empty.

Automata on Infinite Trees. For a given set Υ of directions, the *infinite tree* is the set Υ^* of finite sequences over Υ. A Σ-*labeled* Υ-tree is a function $\Upsilon^* \to \Sigma$.

A *tree automaton* over Σ-labeled Υ-trees has the form $\mathcal{A} = (Q, q_0, T, Acc)$, where Q is a finite set of *states*, $q_0 \in Q$ is an *initial state*, $T \subseteq Q \times \Sigma \times (\Upsilon \mapsto Q)$ is a set of *transitions* and Acc is an acceptance condition $Acc \subseteq Q^\omega$.

A *run* of a tree automaton $\mathcal{A}$ on a Σ-labeled Υ-tree $v : \Upsilon^* \to \Sigma$ is a Q-labeled Υ-tree $r : \Upsilon^* \to Q$, where the root is labeled with the initial state, $r(\varepsilon) = q_0$, and every node $n \in \Upsilon^*$ satisfies some transition $(r(n), v(n), f) \in T$ in the sense that $f(v) = r(n \cdot v)$ for all directions $v \in \Upsilon$. A run r is *accepting* if all paths satisfy the acceptance condition Acc. The *language* of $\mathcal{A}$ consists of all accepted Σ-labeled Υ-trees.

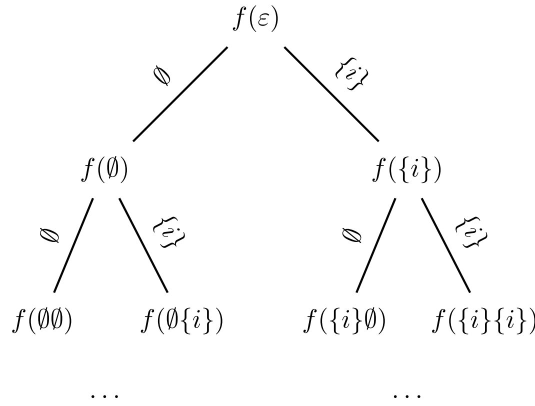

Figure 4. A strategy tree for a strategy $f_\sigma : (2^{\{i\}})^* \to 2^O$.

To build a tree automaton that accepts all implementations of a given S1S formula φ, we first translate φ into a deterministic word automaton $\mathcal{A}_\varphi$. The tree automaton then simply simulates the word automaton along every path of the input tree. Suppose $\mathcal{A}_\varphi$ is a deterministic parity word automaton $(Q, \{q_0\}, T, c)$. To simulate $\mathcal{A}_\varphi$, the tree automaton $\mathcal{A}'_\varphi = (Q, q_0, T', c)$, transitions, for every direction, to the successor state of the word automaton for the combination of the label and the direction, i.e., $T' = \{(q, \sigma, f) \mid q \in Q, \sigma \in \Sigma, f(\upsilon) = T(q, \sigma \cup \upsilon) \text{ for all } \upsilon \in \Upsilon\}$. An example of this construction is shown in Fig. 5. The tree automaton for the response property shown on the left of Fig. 5 was obtained from the deterministic word automaton shown on the left of Fig. 3. In this example, the tree automaton has a Büchi acceptance condition, because the automaton in Fig. 3 is a Büchi automaton. In general, the tree automaton might have a more expressive acceptance condition, such as a parity condition.

Tree Automata Emptiness. Rabin's original emptiness test for tree automata had nonelementary running time. In 1972, this was improved to algorithms with exponential running time in both the number of states and in the number of pairs of the Rabin condition [52,9]. Finally, in 1989, algorithms with polynomial running time in the number of states and exponential running time in the number of pairs were found [53,54].

In general, one can easily reduce the emptiness problem of a tree automaton with a certain acceptance condition to the problem of solving the game with the same type of condition as its winning condition. Following [31], the players in this game are often called *Automaton* and *Pathfinder*: Automaton (Player 0) tries to prove that an accepted tree exists, Pathfinder (Player 1) tries to disprove this by finding a path where the acceptance condition is violated.

For a tree automaton $\mathcal{A} = (Q, q_0, T, \omega)$, the *emptiness game* $((V, E), \mathit{Win})$ has positions $V = V_0 \cup V_1$ where Automaton's positions $V_0 = Q$ are the states of the automaton and Pathfinder's positions $V_1 = T$ are the transitions. The edges $E = E_0 \cup E_1$ correspond to Automaton choosing a transition originating from the present state $E_0 = \{(q, (q, \sigma, f)) \mid q \in Q, (q, \sigma, f) \in T\}$ and Pathfinder choosing

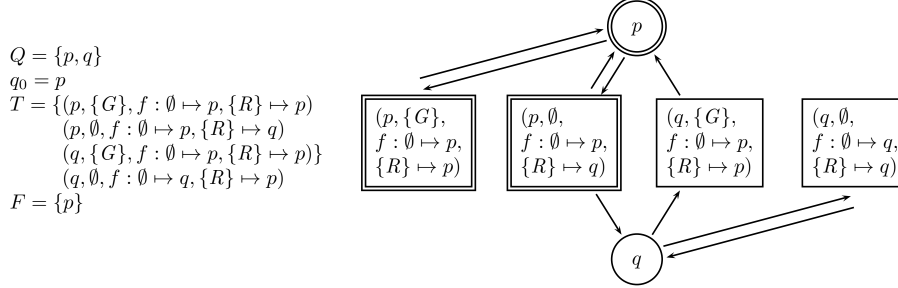

Figure 5. Büchi tree automaton and emptiness game for the response property. Game positions for player *Automaton* (Player 0) are depicted as circles, game positions for player *Pathfinder* (Player 1) as rectangles.

a direction $E_1 = \{((q, \sigma, f), f(\upsilon)) \mid \upsilon \in \Upsilon\}$. The winning condition checks if the sequence of states satisfies the acceptance condition of the automaton; e.g., if *Acc* is given as a parity condition $c : Q \to \mathbb{N}$, then *Win* is the parity condition $c' : V \to \mathbb{N}$ with $c'(q) = c(q)$ for $q \in Q$ and $c'((q, \sigma, f)) = c(q)$ for $(q, \sigma, f) \in T$. The tree automaton is non-empty iff Player Automaton has a q_0-winning strategy.

Figure 5 shows a Büchi tree automaton for the response property and its emptiness game. A p-winning strategy for Player Automaton is to move from p to $(p, \{G\}, f : \emptyset \mapsto p, \{R\} \mapsto p)$, corresponding to the implementation that always provides the grant.

The automata-based solution to Church's problem thus leads to an infinite game, just like the game-based approach. The automata-based approach is, however, an important generalization, because the representation as automata makes it possible to manipulate sets of implementations with automata transformations (see, for example, Section 4.3).

4. Synthesis from Temporal Logics

The invention of temporal logic and the discovery of temporal logic based model checking in the 1980s established computer-aided verification as a field with practical relevance especially for the hardware industry. The temporal logics turned out not only to be often more intuitive to use than S1S, but also to lead to significantly lower complexity for both verification and synthesis.

4.1. Linear-time Temporal Logic

In his seminal paper "The temporal logics of programs" [55], Pnueli introduced linear-time temporal logic (LTL) as a new specification language for the desired behavior of reactive systems. Similar to S1S, LTL is based on sequences. LTL differs from S1S in that there is no explicit mechanism for quantification. References to points in time, and quantification over time is done implicitly through modal operators. For a given set of atomic propositions AP, LTL is generated by the following grammar:

$$\varphi \quad ::= \quad p \;\mid\; \neg\varphi \;\mid\; \varphi \wedge \varphi \;\mid\; \bigcirc\varphi \;\mid\; \varphi \,\mathrm{U}\, \varphi$$

where $p \in AP$ is an *atomic proposition*, $\neg$ and $\wedge$ are Boolean connectives, and $\bigcirc$ and U are temporal operators: $\bigcirc$ is the *next* operator, U the *until* operator. Typically, the logic is extended with other Boolean operators and with derived temporal operators such as *eventually* $\Diamond \varphi \equiv \mathsf{true}\, \mathrm{U}\, \varphi$ and *globally* $\Box \varphi \equiv \neg \Diamond \varphi$.

Like S1S, models of LTL formulas are infinite words. For an infinite word $\alpha \in (2^{AP})^\omega$, $\alpha \models p$ iff $p \in \alpha(0)$; $\alpha \models \bigcirc \varphi$ iff $\alpha(1)\,\alpha(2)\,\alpha(3)\ldots \models \varphi$; and $\alpha \models \varphi_1 \mathrm{U} \varphi_2$ iff there is an $i \geq 0$ such that $\alpha(i)\,\alpha(i+1)\,\alpha(i+2)\ldots \models \varphi_2$ and for all $0 \leq j < i$, $\alpha(j)\,\alpha(j+1)\,\alpha(j+2)\ldots \models \varphi_1$.

The arbiter specification, which was given as a conjunction of S1S formulas in Section 2, can be equivalently stated in LTL as follows:

$$
\begin{array}{ll}
\Box(\neg G_1 \vee \neg G_2) & \text{(mutual exclusion)} \\
\Box(R_1 \rightarrow \Diamond G_1) & \text{(response 1)} \\
\Box(R_2 \rightarrow \Diamond G_2) & \text{(response 2)}
\end{array}
$$

For synthesis from LTL formulas, we assume that the atomic propositions $AP = I \cup O$ are partitioned into inputs I and outputs O. Given an LTL formula φ over AP of length n, one can build a Büchi word automaton with $2^{O(n)}$ states that recognizes the models of φ [56]. The basic idea of the translation is to store, in each state, a set of subformulas that are required to hold on the suffix of the input word. This idea has been optimized in various ways [57,58]. In practice, it is particularly important to simplify the automaton on-the-fly during the construction, for example by removing redundant transitions and by merging equivalent states [59].

From the nondeterministic Büchi automaton, one can obtain, via Safra's construction, a deterministic automaton as required by the game-based or automata-based synthesis approaches discussed in Sections 3.2 and 3.3. In terms of complexity, the LTL-to-automata translation results in a Büchi automaton with an exponential number of states in the length of the formula; from there, one obtains, via Safra's construction, a deterministic Rabin automaton with a doubly exponential number of states and a single-exponential number of pairs. The emptiness game of the corresponding tree automaton can be solved in polynomial running time in the number of states and in exponential running time in the number of pairs, resulting overall in a doubly exponential running time. The problem is, in fact, 2EXPTIME-complete [54].

4.2. Branching-time Temporal Logic

The temporal operators of LTL describe possible observations along a single time line. Branching-time temporal logics, by contrast, see time as a tree structure: at any point in time, there may be multiple futures. Path quantifiers make it possible to specify the existence and absence of futures with certain properties.

The standard branching-time temporal logics are CTL* [60] and its sublogic CTL [61]. The syntax of CTL* distinguishes state formulas Φ and path formulas φ, as generated by the following grammar:

$$
\begin{array}{lllllllllll}
\Phi & ::= & a & \mid & \neg\Phi & \mid & \Phi \wedge \Phi & \mid & \mathrm{A}\,\varphi & \mid & \mathrm{E}\,\varphi \\
\varphi & ::= & \Phi & \mid & \neg\varphi & \mid & \varphi \wedge \varphi & \mid & \bigcirc \varphi & \mid & \varphi\, \mathrm{U}\, \varphi
\end{array}
$$

As for LTL, CTL* is usually extended with additional Boolean connectives and temporal operators. CTL is the sublogic of CTL* where every temporal operator is immediately preceded by a path quantifier.

CTL* formulas can be interpreted over arbitrary Kripke structures. For the purposes of synthesis, the models of interest are again 2^O-labeled 2^I-trees. Let v be such a tree. A *path* γ in v is an infinite sequence $\gamma(0)\,\gamma(1)\,\gamma(2)\ldots \in ((2^I)^*)^\omega$ of nodes such that for each node $\gamma(i)$ in the sequence, the successor node $\gamma(i+1)$ is a child of $\gamma(i)$, i.e., $\gamma(i+1) = \gamma(i)\cdot \upsilon$ for some $\upsilon \in 2^I$. The set of paths originating in a node n is denoted by $\mathit{Paths}(n)$.

A node n satisfies an existentially quantified state formula $\mathrm{E}\,\varphi$ iff there is a path in $\mathit{Paths}(n)$ that satisfies the path formula φ; analogously, n satisfies a universally quantified state formula $\mathrm{E}\,\varphi$ iff all paths in $\mathit{Paths}(n)$ satisfy φ. A path γ satisfies an atomic proposition $p \in AP$ iff $p \in v(\gamma(0))$. The satisfaction of the temporal modalities is defined as for LTL, i.e., γ satisfies $\bigcirc\varphi$ iff $\gamma(1)\,\gamma(2)\,\gamma(3)\ldots$ satisfies φ, and γ satisfies $\varphi_1 \mathrm{U} \varphi_2$ iff there is an $i \geq 0$ such that $\gamma(i)\,\gamma(i+1)\,\gamma(i+2)\ldots$ satisfies φ_2 and for all $0 \leq j < i$ $\gamma(j)\,\gamma(j+1)\,\gamma(j=2)\ldots$ satisfies φ_1. The Boolean connectives are interpreted in the usual way.

Similar to the translation of LTL formulas into equivalent Büchi word automata, CTL formulas can be translated into Büchi tree automata. The size of the resulting automaton is exponential in the length of the formula [62]. Since Büchi games can be solved in polynomial time, the realizability of a CTL formula can thus be checked in exponential time. A matching lower bound follows from the satisfiability problem of CTL [63].

For CTL* formulas, we obtain a Rabin automaton with a doubly exponential number of states and a single exponential number of pairs. The synthesis problem can therefore be solved in doubly exponential time [64], and is, hence, 2EXPTIME-complete (the lower bound follows from the synthesis problem for LTL).

4.3. Synthesis under Incomplete Information

Church's problem is based on a specification that refers to the inputs and outputs of the implementation to be synthesized. Since the inputs are, by definition, observable by the implementation, this results in a game with *perfect information*. Often, however, one is interested in synthesizing a process within a larger system, where the process only observes a subset of the global inputs. This results in a game with *incomplete information*.

The classic solution to games with incomplete information is the translation to perfect-information games with a *knowledge-based subset construction* due to Reif [65]. Reif's construction simulates a given game $\mathcal{G}$ with incomplete information with a game $\mathcal{G}'$ with perfect information, where the positions of $\mathcal{G}'$ are sets of positions of $\mathcal{G}$. The set of positions of $\mathcal{G}$ reached after a certain sequence of moves in $\mathcal{G}'$ consists of those positions of $\mathcal{G}$ that would have been reached after some sequence of moves that is indistinguishable, by Player 0, from the sequence that actually occurred.

A similar idea can also be applied in a transformation on tree automata that recognize sets of strategy trees. Figure 6 shows the *widening* of the strategy tree

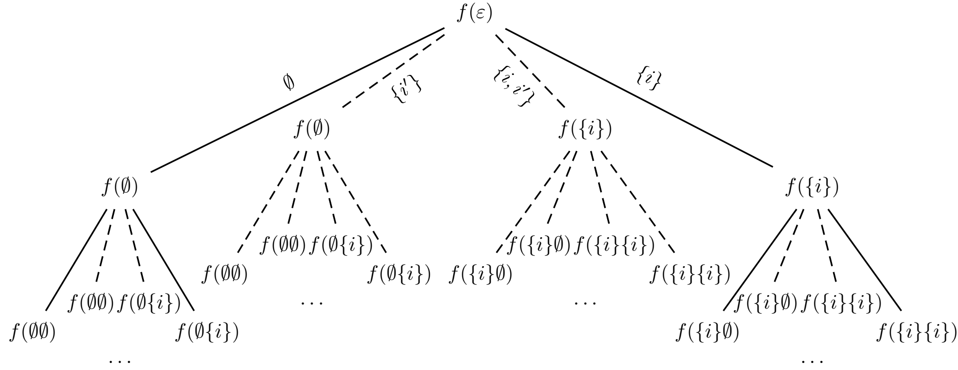

Figure 6. Widening of the strategy tree for strategy $f : (2^{\{i\}})^* \to 2^O$ over input variable i, shown in Fig. 4, to input variables i and i'.

from Fig. 4 for input variable i to the larger set of input variables consisting of both i and i'. Since the original strategy does not depend on i', the widened tree has identical labels on paths where the values of i are identical. Kupferman and Vardi defined a tree automata transformation called *narrowing* that transforms a given tree automaton on strategy trees to a tree automaton on strategies with a reduced set of inputs such that a strategy tree is accepted by the new automaton if and only if its widening is accepted by the original automaton [66]. To synthesize an implementation for a temporal specification, one first builds, as in standard synthesis, a tree automaton for the specification and then applies narrowing to reduce the inputs to the subset that is actually observable by the implementation. Realizability under incomplete information corresponds to non-emptiness of this automaton. Independently of the presence of incomplete information, the synthesis problems for LTL, CTL, and CTL* are complete for 2EXPTIME, EXPTIME, and 2EXPTIME, respectively.

4.4. Synthesis of Distributed Systems

The *distributed synthesis problem* is the generalization of the synthesis problem where we construct, instead of a single implementation, a set of implementations, one for each system process, that together must satisfy the specification. The system architecture is typically given as a directed graph, where the nodes represent processes, including the environment as a special process. The edges of the graph are labeled by variables indicating that data may be transmitted between two processes. The same variable may occur on multiple outgoing edges of a single node, allowing for the broadcast of data. Figure 7 shows several examples of system architectures.

The distributed synthesis problem was introduced by Pnueli and Rosner, who showed that the problem is decidable for *pipeline architectures* but undecidable in general [4]. In particular, the problem is already undecidable for the simple architecture A_0, which consists of the environment and two independent system processes. The decidability result was later generalized to one-way ring architectures [67] and, finally, to all architectures without information forks [68].

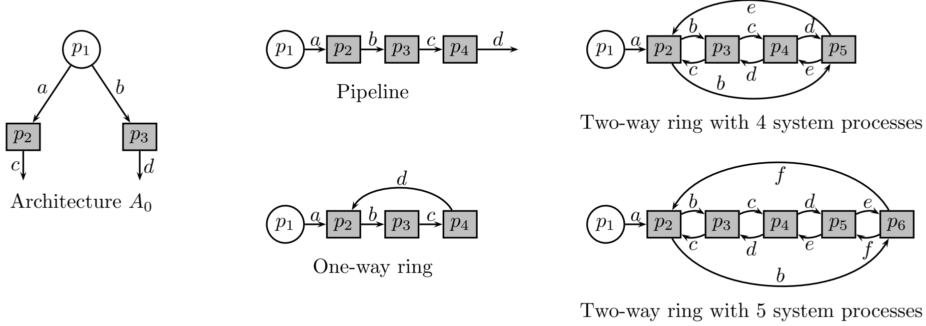

Figure 7. Distributed architectures

Information forks are a comprehensive criterion that characterizes all architectures with an undecidable synthesis problem. Intuitively, an information fork is a situation where two processes receive information from the environment (directly or indirectly), in such a way that they cannot completely deduce the information received by the other process. Consider, for example, the two-way ring with five system processes shown in Figure 7 on the bottom on the right side. The synthesis problem is undecidable because of the information fork in processes p_4 and p_5. The environment p_1 can transmit information through a, b, c to p_4 that remains secret to p_5, and vice versa, transmit information through a, b, f to p_5 that remains secret to p_4. The distributed synthesis problem becomes decidable if we eliminate one of the two processes, resulting in the two-way ring with four processes shown on the top on the right side.

For architectures without information forks, the synthesis problem is solved in two phases. First, the architecture is simplified, by grouping the system processes according to the information they possess about the environment's behavior. Groups of processes with the same level of information can simulate each other, and are therefore collapsed into a single process. Feedback edges from processes with a lower level of information to those with a higher level are eliminated, because they do not transmit new information. In the second phase, the synthesis problem is solved with an automata-based construction that, by repeatedly applying the narrowing construction, successively eliminates processes along the information order, starting with the best-informed process. The complexity of this construction is nonelementary in the number of processes.

4.5. Temporal Logics with Strategy Quantifiers

In 1997, Alur, Henzinger, and Kupferman extended the linear – branching-time spectrum of the temporal logics with the alternating-time temporal logics as a third type of temporal logic [11]. The alternating-time logic ATL* replaces the path quantifier of CTL* with the more general strategy quantifier. While the CTL* formula $\mathrm{E}\,\varphi$ means that there exists a path where φ holds, the ATL* formula $\langle\langle 0 \rangle\rangle \varphi$ means that Player 0 has a strategy to ensure φ. With alternating-time temporal logic, the realizability of a temporal property became itself a property expressible inside the logic, rather than as an externally defined semantical con-

dition. The added expressiveness is useful to describe the game-like behavior of multiprocess systems. For example in a cache-coherence protocol, one might require that every process has the *capability* to obtain memory access, irrespective of the behavior of the other processes.

The semantics of the alternating-time temporal logics refer to concurrent game structures, a game-based extension of Kripke structures. For a natural number k, a *k-player concurrent game structure* over a set of atomic propositions AP has the form $G = (S, L, d, \delta)$, where S is a set of states, $L : S \to 2^{AP}$ is a labeling function; $d_a : S \to \mathbb{N}$ assigns to each player $a \in \{1, 2, \ldots, k\}$ and each state $s \in S$ a number of moves available to player a, the resulting set of vectors $D(s) = \{1, \ldots, d_1(s)\} \times \{1, \ldots, d_2(s)\} \times \ldots \times \{1, \ldots, d_k(s)\}$ are called the move vectors of state s; δ is the transition function, which assigns to each state s and move vector $\langle j_1, j_2, \ldots, j_k\rangle \in D(s)$ a successor state $\delta(s, \langle j_1, j_2, \ldots j_k\rangle) \in S$. If a play reaches state s, each player chooses, concurrently, a number j_a beween 1 and $d_a(s)$, and the play continues with state $\delta(s, \langle j_1, j_2, \ldots j_k\rangle)$.

Since realizability of a temporal property φ can be stated as the ATL* property $\langle\langle 0\rangle\rangle\varphi$ of a generic game structure in which Player 0 gets to set the outputs and Player 1 the inputs, it is the model checking problem, not the realizability problem, that is most relevant for synthesis. ATL* model checking requires, like LTL synthesis, doubly exponential time. Model checking formulas of the restricted sublogic ATL requires the solution of games with Boolean combinations of Büchi conditions, which can be done in PSPACE.

Strategy Logic (SL) [12,69] generalizes alternating-time temporal logic by treating strategies as explicit first-order objects. In SL, the ATL* property $\langle\langle 1\rangle\rangle\varphi$ is expressed as $\exists x.\ \forall y.\ \varphi(x, y)$, i.e., there exists a strategy x for Player 1 such that for all strategies y of Player 2, φ is guaranteed to hold. The explicit quantification over strategies makes it possible to compare multiple strategies for the same player; Nash equilibria and similar properties can easily be expressed as SL formulas. The complexity of the model checking problem for SL formulas (which suffices for synthesis) is nonelementary; the satisfiability problem is undecidable.

A first attempt of a logical characterization of the *distributed synthesis* problem was already carried out within the setting of the alternating-time temporal logics. *Game structures under incomplete information* extend game structures with an *observability vector* $P = \{\Pi_a \mid 1 \leq a \leq k\}$, which identifies, for each player a, a set of atomic propositions that are observable by a. A strategy for player a may only depend on the history of its observable propositions, not on the history of the other, unobservable propositions. ATL* under incomplete information can thus express the existence of a distributed implementation. Not suprisingly, however, the model checking problem for ATL* (and even ATL) under incomplete information turned out to be undecidable [70].

A decidable logic for the distributed synthesis problem is *Coordination Logic* (CL) [13]. Like Strategy Logic, CL has explicit strategy quantifiers. The informedness of the strategies is not, as in ATL* under incomplete information, defined by the model, but is defined directly in the formula. CL uses two types of variables: *strategy variables* $\mathcal{S}$, which represent, like in SL, strategies (or *output*) and *coordination variables* $\mathcal{C}$, which represent information (or *input*). The syntax of CL is given by the grammar

$$\varphi ::= x \mid \neg x \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \mid \bigcirc \varphi \mid \varphi \,\mathcal{U}\, \varphi \mid \exists C \exists s.\varphi \mid \exists C \forall s.\varphi,$$

where $x \in \mathcal{C} \uplus \mathcal{S}$, $C \subseteq \mathcal{C}$, and $s \in \mathcal{S}$. The operators of CL consist of the usual LTL operators as well as the new *subtree quantifiers* $\exists C \exists s.\varphi \mid \exists C \forall s.\varphi$.

Coordination variables provide strategy variables with the information required for their decisions. Following the structure of a formula, a bound coordination variable c is *visible* to a bound strategy variable s, if s is in the scope of the subtree quantifier that introduced c or if c is a free coordination variable. The strategies the subtree quantifier ranges over are mappings from histories of valuations of the coordination variables in the scope of s, to a (Boolean) valuation of s.

CL can express all decidable distributed synthesis problems discussed in Section 4.4. In practice, many more synthesis problems can be expressed in CL (and are, hence, decidable), even though their system architecture does not fall into a decidable class. Suppose, for example, that the outputs c and d of processes p_1 and p_2, respectively, in the undecidable architecture A_0 from Fig. 7 are *independent* of each other and only depend on their respective inputs a and b. Then, the synthesis problem can be expressed as the CL formula

$$\exists \{a\} \exists c.\ \varphi(a, c) \ \wedge\ \exists \{b\} \exists d.\ \psi(b, d),$$

where the independence of the outputs is reflected by the fact that the two conjuncts do not have any shared variables and can, hence, be evaluated independently of each other.

5. Towards Practical Synthesis Algorithms

Even with the reduction in complexity from nonelementary for S1S to doubly exponential for LTL, the complexity challenge was not solved – for practical purposes, doubly exponential is still an intractable complexity. The quest for more practical synthesis methods since about a decade ago has, so far, been based on three main lines of research. The first line is concerned with the *input* to the synthesis problem, the specification logic. The goal is to find fragments of specification languages like LTL that allow for faster synthesis algorithms, while still being sufficiently expressive for the specification of relevant synthesis problems. The second line is concerned with the *output* of the synthesis problem, the implementation. Bounded synthesis restricts the size of the implementation and thus forces the synthesis algorithm to search for small, and, hence, easy to find, solutions. The third line of work towards practical algorithms is concerned with the *internal representation* of the synthesis problem. Symbolic and SAT-based approaches can, in certain cases, reduce the required memory exponentially.

5.1. Efficient Fragments

A fruitful starting point for "synthesis-friendly" specification logics is the observation that certain games, such as Büchi games and parity games with a constant number of colors, can be solved in polynomial time. Synthesis algorithms based

on such games are often referred to as "polynomial-time" even though, strictly speaking, the polynomial-time complexity is in the size of the game arena, which is typically at least exponential in its logical description.

A widely used synthesis-friendly fragment of LTL is *GR(1)*, short for *Generalized Reactivity (1)*, which was introduced in 2006 by Piterman, Pnueli, and Sa'ar [71]. GR(1) generalizes other fragments of LTL considered earlier [72,73]. For a given set of input variables I and output variables O, a GR(1) formula is an implication

$$A_1 \wedge A_2 \wedge \ldots \wedge A_m \ \rightarrow G_1 \wedge G_2 \wedge \ldots \wedge G_n,$$

of a conjunction of *assumptions* $A_i, i = 1 \ldots m$ to a conjunction of *guarantees* $G_i, i = 1 \ldots n$. The guarantees are restricted to the following types of formulas: (1) *initialization properties*, which are state formulas, i.e., formulas without temporal operators; these formulas are used to specify the initial state of the system; (2) *safety properties* of the form $\square(\varphi \rightarrow \bigcirc\psi)$, where φ and ψ are state formulas; these properties are used to describe the evolution of the system; and (3) *liveness properties* of the form $\square\Diamond\varphi$, where φ is a state formula; these properties describe objectives that need to be accomplished infinitely often. The assumptions have the same form, but are additionally restricted to refer to the environment in the sense that initialization properties are state formulas over I only, and safety properties are of the form $\square(\varphi \rightarrow \bigcirc\psi)$ where φ is a state formula over $I \cup O$ and ψ is a state formula over I.

GR(1) specifications define a game, where each position identifies a valuation of the input and output variables, the edges are defined by the safety assumptions and guarantees, and the winning condition for Player 0 is given by an LTL formula based on the liveness properties, i.e., a temporal formula of the form

$$(\square\Diamond\varphi_1 \wedge \square\Diamond\varphi_2 \wedge \ldots \square\Diamond\varphi_k) \ \rightarrow \ (\square\Diamond\psi_1 \wedge \square\Diamond\psi_2 \wedge \ldots \square\Diamond\psi_l),$$

where $\varphi_i, i = 1 \ldots k$, and $\psi_i, i = 1 \ldots l$, are state formulas. This game can be translated into a parity game with three colors [74], which can be solved in quadratic time [75]. Ehlers showed that the same principle also applies to an extension of GR(1) with *stability properties* of the form $\Diamond\square\varphi$, called Generalized Rabin (1) [76]. Generalized Rabin (1) specifications translate to parity games with five colors, which can also be solved in polynomial time.

A disadvantage of GR(1) and similar fragments of LTL is, however, that they often require an expensive pre-processing step known as *pre-synthesis* [77]. Pre-synthesis is needed to encode system properties that cannot be specified directly in GR(1). Consider, for example, the response property $\square(R \rightarrow \Diamond G)$, which cannot be expressed directly in GR(1). To encode the response property in GR(1), pre-synthesis adds an auxiliary output variable H which is true whenever there was a request that has not yet been followed by a grant:

$$(\neg H) \wedge (\square((R \vee H) \wedge \neg G) \rightarrow \bigcirc H) \wedge (\square((\neg R \wedge \neg H) \vee \neg G) \rightarrow \bigcirc \neg H) \wedge (\square\Diamond\neg H)$$

The initialization property $\neg H$ and the safety properties $\square((R \vee H) \wedge \neg G) \rightarrow \bigcirc H$ and $\square((\neg R \wedge \neg H) \vee \neg G) \rightarrow \bigcirc \neg H$ ensure that H is indeed *true* if and only if there

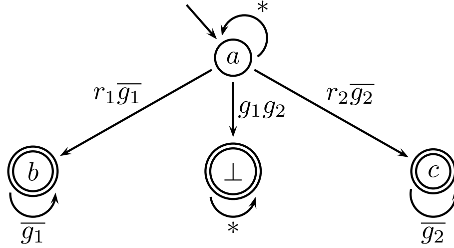

Figure 8. Universal co-Büchi automaton corresponding to the arbiter specification. The states depicted as double circles (b, $\bot$, and c) are the rejecting states in F. The abbreviations $r_1\overline{g_1}$, g_1g_2, $r_2\overline{g_2}$, $\overline{g_1}$, $\overline{g_2}$ are used to indicate, in Boolean notation, the input symbols of the transitions; e.g., $r_1\overline{g_1} = \{r_1, r_2, g_2\}$, $\{r_1, g_2\}$, $\{r_1, r_2\}$, $\{r_1\}$. $*$ denotes all subsets of $\{r_1, r_2, g_1, g_2\}$.

is an unanswered request; the liveness property $\Box\Diamond\neg H$ ensures that no request remains unanswered forever.

Despite the syntactic restrictions and the need for pre-synthesis, GR(1) has found many applications in practice.

5.2. Bounded Synthesis

The *bounded synthesis problem* [14] is a variation of the synthesis problem, where only implementations up to a given bound on the number of states are considered. The motivation for bounded synthesis is to focus on small implementations. In practice, realizable specifications often have reasonably small implementations, even though the theoretical bound on the smallest implementation is large, such as doubly exponential for LTL specifications.

Algorithmically, bounded synthesis is closely related to Kupferman and Vardi's *Safraless* approach [78], which avoids Safra's complicated determinization construction with an alternative automata transformation that only preserves realizability, not language-equivalence in the sense that all implementations remain represented. The Safraless construction goes via a universal co-Büchi automaton to a nondeterministic Büchi tree automaton that is non-empty if and only if the original specification is realizable. The Safraless approach is easier to implement than the standard construction, in particular using symbolic representations (see Section 5.3). Bounded synthesis improves the Safraless approach by constructing, instead of a single *Büchi* tree automaton, a *sequence* of increasingly larger *safety* tree automata, corresponding to an increasing bound on the size of the implementation, until a precomputed maximal bound is reached, at which point, an implementation, if it exists, must have been found.

The form $\mathcal{A} = (Q, q_0, T, Acc)$ of a *universal automaton over infinite words* is the same as that of a nondeterministic automaton (see Section 3.1): Q is a finite set of *states*, $q_0 \in Q$ is an *initial state*, $T \subseteq Q \times \Sigma \times Q$ is a set of *transitions*, and Acc is an *acceptance condition*. The transitions are, however, not interpreted existentially, which means that, in every run, some applicable transition is chosen; instead, the transitions are interpreted universally: all applicable transitions must be chosen. A run on an infinite input word $\sigma_0\sigma_1\sigma_2\ldots$ is an S-labeled Υ-tree $r : \Upsilon^* \to \Sigma$ for some set of directions Υ, such that the root is

labeled with the initial state, $r(\varepsilon) = q_0$, and for every node $n \in \Upsilon^*$ and every transition $(r(n), \sigma_{|n|}, q') \in T$, q' occurs on a child of n, i.e., there exists a direction $\upsilon \in \Upsilon$ such that $r(n \cdot \upsilon) = q'$. The run is accepting if every branch satisfies the acceptance condition Acc. Algorithms for the translation of LTL formulas to nondeterministic Büchi automata can also be used to translate the formulas to universal co-Büchi automata. For this purpose, one actually translates the negation of the formula of interest; the nondeterministic Büchi automaton obtained from the negation thus recognizes the complement of the intended language. The same automaton interpreted as a universal co-Büchi automaton is the exact dual of the nondeterministic automaton, and, hence, recognizes again the complement, i.e., the set of models of the original formula. Figure 8 shows an example of a universal co-Büchi automaton. The automaton accepts precisely the models of the arbiter specification.

In bounded synthesis, the universal co-Büchi automaton is approximated with a sequence of deterministic safety automata. The safety automata maintain a set of "currently active" states of the universal automaton, and, for each state of the universal automaton, a natural number, which indicates the maximal number of visits to rejecting states on some path in the run tree that leads to this state. The ith safety automaton in the sequence limits this number to i (and rejects if the number is exceeded).

5.3. Symbolic Synthesis

Symbolic synthesis algorithms are based on compact representations of sets of game positions, using data structures such as *binary decision diagrams* (BDDs) [79], *and-inverter graphs* (AIGs) [80], or *antichains* [81]. The symbolic data structures are used to represent the arena (V, E) and the sets of positions that occur in a fixed point iteration that computes the winning positions. For a safety game $((V, E), S)$, the fixed point computation iterates the *uncontrollable predecessors* operator, defined as

$$\begin{aligned} UPRE(X) = \{&v \in V_0 \mid \forall v' \in V \,.\, (v, v') \in E \rightarrow v' \in X\} \cup \\ \{&v \in V_1 \mid \exists v' \in V \,.\, (v, v') \in E \wedge v' \in X\}. \end{aligned}$$

The $UPRE$ operator collects all positions of Player 1 where some outgoing edge leads to the given set X of positions, and all positions of Player 0, where all outgoing edges lead to X. The least fixed point of the function $\tau : X \mapsto \overline{S} \cup UPRE(X)$, where $\overline{S}$ denotes the complement of the safe states S, consists of exactly the game positions from which Player 1 has a winning strategy [82].

For slightly more complex synthesis games, such as Büchi games and the synthesis games that result from GR(1) specifications (cf. Section 5.1), the winning positions are similarily computed in a nested fixed point iteration (cf. [71]). For full LTL specifications, the synthesis problem is often reduced, via the bounded synthesis approach described in Section 5.2, to the solution of safety games [81,83].

There is no general guarantee that the symbolic algorithms outperform algorithms based on an explicit representation of the game arena, and there are, in fact, well-known structures on which, for example, BDDs perform poorly [84]. In most practical situations, however, symbolic methods perform significantly better

than explicit methods. especially in synthesis problems with many input variables, which result in game arenas with a large number of edges.

5.4. SAT-based Synthesis

The advances in *Boolean satisfiability* (SAT) and *satisfiability modulo theories* (SMT) solving during the 1990s and the success of bounded model checking [85,86] and other satisfiability-based verification techniques inspired the development of synthesis techniques based on SAT and SMT solvers. The bounded synthesis approach (cf. Section 5.2) can, for example, be encoded as a constraint system, where the transition function and the output function of the Moore machine to be synthesized are represented as uninterpreted functions [87]. This approach generalizes naturally to the distributed synthesis problem, where each process has its own transition function, and other variations of the synthesis problem, such as the synthesis of symmetric processes or the synthesis of systems built from component libraries [88]. SMT-based bounded synthesis has been combined with symbolic verification in the *lazy synthesis* approach [89], where the SMT solver builds candidate implementations based on an incomplete constraint system; the symbolic verifier then compares the candidate against the specification and either proves the correctness of the candidate, which terminates the synthesis process, or finds counter examples, which are in turn used to refine the constraint system used by the SMT solver.

For safety games, satisfiability checking for *quantified boolean formulas* (QBF) combined with computational learning has been used to compute the winning region of Player 0 [90]. QBF solving has also been used to *refute* realizability, by unrolling the game arena for a bounded number of steps, quantifying universally over choices of Player 0 and existentially over choices of Player 1 [91]. This approach again generalizes to the distributed synthesis problem, by unrolling the game arena simultaneously along several paths that are indistinguishable from the perspective of some process [92].

6. Modern Applications

For a long time, reactive synthesis was considered a theoretician's exercise. It was only about five to ten years ago that serious case studies started to be carried out. With the advances towards practical synthesis algorithms and the growing availability of synthesis tools, it suddenly became clear that reactive synthesis was no longer limited to academic toy examples and should, instead, venture out to industrial applications. At the time of this writing, it is still too early to give a complete picture of the real-world applications of reactive synthesis. The following examples should give a reasonable idea, however, of the breadth and potential of the modern applications.

Hardware. The automatic construction of circuits is still one of the main targets of reactive synthesis. A good illustration of the current state of the art is the synthesis of the arbiter for the AMBA AHB bus. The *Advanced Microcontroller Bus Architecture* (AMBA) is an open-standard, on-chip interconnect specification for

the connection and management of functional blocks in system-on-a-chip (SoC) designs [93]. The *Advanced high-performance Bus (AHB)* is a bus protocol defined in the AMBA standard that connects up to 16 masters with up to 16 slaves. The role of the arbiter is to control the access to the bus in response to requests by the masters. In 2007, Bloem *et al.* gave a specification of the AMBA AHB bus as a GR(1) formula with four assumptions and eleven guarantees [16] and reported the synthesis of an arbiter for up to three masters; in follow-up work, the synthesis was scaled to the full set of 16 masters specified in the AMBA standard [94,95].

Device drivers. A device driver is a computer program that provides a software interface to a hardware device. In addition to the manufacturer of the hardware device, the operating systems vendors also care about the quality of the device drivers, and impose quality checks, because it impacts the reputation of the operating system. Manually developing device drivers is an error-prone and often tedious task. Synthesis can be used to construct device drivers automatically from formal specifications of the operating system (OS) interface and of the hardware device. The synthesis game begins with the environment making OS-to-driver requests. In response to these requests, the system selects commands to the device that cause the device to produce the correct response for the given OS request and to continue to operate correctly in the future. Examples of low-level drivers that have been synthesized successfully are an Intel PRO/1000 ethernet controller, a UART NS16450 Linux driver, and an SD Host controller [17].

Robotics. Reactive synthesis is used to generate controllers for robotic mission and motion planning for complex robot behaviors. The advantage of reactive synthesis over traditional hierarchical planning is that the robot is *guaranteed* to achieve the desired task if it is feasible. In the LTLMoP tool [96], the user defines a task for a robot by drawing a workspace map and then defining goals, such as search and rescue, coverage, or collision avoidance, that refer to the regions of the map. This task specification is translated into a GR(1) formula. The atomic propositions of this formula include propositions that refer to the robot's sensor readings, which are controlled by the environment. If the formula is realizable, the resulting strategy is combined with continuous control handlers to create a hybrid controller, which is then used to control a real or simulated robot.

7. Conclusions

This article went on a journey through more than 50 years of research on the synthesis problem; we traced the evolution of Church's problem from the theoretical challenge in 1957 to practical synthesis algorithms and modern applications. Each of the three waves of inquiry into Church's problem, the early solutions during the 1960s, the synthesis algorithms for the temporal logics starting in the 1980s, and the quest for practical algorithms over the past decade, has brought enormous progress. In terms of complexity, we have gone from "foredoomed" problems to algorithms with substantial but reasonable complexity; the running times are somewhere between polynomial and exponential, provided that certain realistic assumptions are satisfied, such as specifications given in a synthesis-friendly frag-

ment, like GR(1), or implementations that are sufficiently small implementation to be discovered quickly by SMT-based bounded synthesis. For distributed systems, we have gone from isolated decidability results in restricted system architectures to uniform algorithms and a comprehensive logical representation that covers all decidable cases. In terms of applications, academic toy examples have started to give way to real design problems from industry. In short, we have gone from an open theoretical problem to a field with modern applications, practical algorithms, and the potential to revolutionize the development process for reactive systems.

References

[1] A. Church. Applications of recursive arithmetic to the problem of circuit synthesis. In *Summaries of the Summer Institute of Symbolic Logic*, volume 1, pages 3–50. Cornell Univ., Ithaca, NY, 1957.

[2] J. Richard Büchi and Lawrence H. Landweber. Solving sequential conditions by finite-state strategies. *Transactions of the American Mathematical Society*, 138, 1969.

[3] L. Stockmeyer. *The Complexity of Decision Problems in Automata Theory and Logic*. PhD thesis, Massachusetts Institute of Technology, 1974.

[4] Amir Pnueli and Roni Rosner. Distributed reactive systems are hard to synthesize. In *Proc. FOCS'90*, pages 746–757, 1990.

[5] Aaron Bohy, Véronique Bruyère, Emmanuel Filiot, Naiyong Jin, and Jean-François Raskin. Acacia+, a tool for LTL synthesis. In P. Madhusudan and Sanjit A. Seshia, editors, *CAV*, volume 7358 of *LNCS*, pages 652–657. Springer, 2012.

[6] Roderick Paul Bloem, Hans-Jürgen Gamauf, Georg Hofferek, Bettina Könighofer, and Robert Könighofer. Synthesizing robust systems with RATSY. In Open Publishing Association, editor, *SYNT 2012*, volume 84, pages 47 – 53. Electronic Proceedings in Theoretical Computer Science, 2012.

[7] Rüdiger Ehlers. Unbeast: Symbolic bounded synthesis. In Parosh Aziz Abdulla and K. Rustan M. Leino, editors, *TACAS*, volume 6605 of *Lecture Notes in Computer Science*, pages 272–275. Springer, 2011.

[8] Roderick Bloem, Rüdiger Ehlers, and Swen Jacobs. The synthesis competition. `http://www.syntcomp.org/`, July 2014.

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

[10] Martín Abadi, Leslie Lamport, and Pierre Wolper. Realizable and unrealizable specifications of reactive systems. In *Proceedings of the 16th International Colloquium on Automata, Languages and Programming*, ICALP '89, pages 1–17, London, UK, UK, 1989. Springer-Verlag.

[11] Rajeev Alur, Thomas A. Henzinger, and Orna Kupferman. Alternating-time temporal logic. In Willem P. de Roever, Hans Langmaack, and Amir Pnueli, editors, *COMPOS*, volume 1536 of *Lecture Notes in Computer Science*, pages 23–60. Springer, 1997.

[12] Krishnendu Chatterjee, Thomas A. Henzinger, and Nir Piterman. Strategy logic. *Inf. Comput.*, 208(6):677–693, 2010.

[13] Bernd Finkbeiner and Sven Schewe. Coordination logic. In Anuj Dawar and Helmut Veith, editors, *CSL*, volume 6247 of *Lecture Notes in Computer Science*, pages 305–319. Springer, 2010.

[14] Sven Schewe and Bernd Finkbeiner. Bounded synthesis. In *Proc. ATVA*, pages 474–488. Springer-Verlag, 2007.

[15] J. R. Burch, E.M. Clarke, K. L. McMillan, D.L. Dill, and L. J. Hwang. Symbolic model checking: 1020 states and beyond. In *Logic in Computer Science, 1990. LICS '90, Proceedings., Fifth Annual IEEE Symposium on e*, pages 428–439, Jun 1990.

[16] Roderick Bloem, Stefan Galler, Barbara Jobstmann, Nir Piterman, Amir Pnueli, and Martin Weiglhofer. Automatic hardware synthesis from specifications: A case study. In

Proceedings of the Conference on Design, Automation and Test in Europe (DATE), pages 1188–1193, 2007.
[17] Mona Vij, John Keys, Arun Raghunath, Scott Hahn, Vincent Zimmer, Leonid Ryzhyk, Adam Christopher Walker, and Alexander Legg. Device driver synthesis. *Intel Technology Journal*, 17(2):136–157, dec 2013.
[18] Alonzo Church. Logic, arithmetic, and automata. In *Proc. Internat. Congr. Mathematicians (Stockholm, 1962)*, pages 23–35. Inst. Mittag-Leffler, Djursholm, 1963.
[19] Cordell Green. Application of theorem proving to problem solving. In *Proceedings of the 1st International Joint Conference on Artificial Intelligence*, IJCAI'69, pages 219–239, San Francisco, CA, USA, 1969. Morgan Kaufmann Publishers Inc.
[20] Zohar Manna and Richard Waldinger. A deductive approach to program synthesis. *ACM Trans. Program. Lang. Syst.*, 2(1):90–121, January 1980.
[21] Armando Solar-Lezama. Program sketching. *STTT*, 15(5-6):475–495, 2013.
[22] J. Richard Büchi. Weak second-order arithmetic and finite automata. *Mathematical Logic Quarterly*, 6(1-6):66–92, 1960.
[23] Calvin C. Elgot. Decision problems of finite automata design and related arithmetics. *Transactions of the American Mathematical Society*, 98(1):pp. 21–51, 1961.
[24] Julius R. Büchi. On a decision method in restricted second order arithmetic. In Ernest Nagel, Patrick Suppes, and Alfred Tarski, editors, *Proceedings of LMPS*, pages 1–11. Stanford University Press, June 1962.
[25] A.W. Mostowski. Regular expressions for infinite trees and a standard form of automata. In Andrzej Skowron, editor, *Computation Theory*, volume 208 of *Lecture Notes in Computer Science*, pages 157–168. Springer Berlin Heidelberg, 1985.
[26] Robert McNaughton. Testing and generating infinite sequences by a finite automaton. *Information and Control*, 9(5):521–530, 1966.
[27] S. Safra. On the complexity of omega -automata. In *Proceedings of the 29th Annual Symposium on Foundations of Computer Science*, SFCS '88, pages 319–327, Washington, DC, USA, 1988. IEEE Computer Society.
[28] M. Michel. Complementation is more difficult with automata on infinite words. Technical report, CNET, 1988.
[29] Nir Piterman. From nondeterministic Büchi and Streett automata to deterministic parity automata. In *LICS*, pages 255–264. IEEE Computer Society, 2006.
[30] R. McNaughton. Project MAC Rep. Technical report, MIT, 1965.
[31] Yuri Gurevich and Leo Harrington. Trees, automata, and games. In *Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing*, STOC '82, pages 60–65, New York, NY, USA, 1982. ACM.
[32] Helmut Lescow. On polynomial-size programs winning finite-state games. In Pierre Wolper, editor, *Computer Aided Verification*, volume 939 of *Lecture Notes in Computer Science*, pages 239–252. Springer Berlin Heidelberg, 1995.
[33] E.Allen Emerson. Automata, tableaux, and temporal logics. In Rohit Parikh, editor, *Logics of Programs*, volume 193 of *Lecture Notes in Computer Science*, pages 79–88. Springer Berlin Heidelberg, 1985.
[34] E.A Emerson and C.S. Jutla. Tree automata, mu-calculus and determinacy. In *Foundations of Computer Science, 1991. Proceedings., 32nd Annual Symposium on*, pages 368–377, Oct 1991.
[35] Florian Horn. Explicit Muller games are PTIME. In Ramesh Hariharan, Madhavan Mukund, and V. Vinay, editors, *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2008, December 9-11, 2008, Bangalore, India*, volume 2 of *LIPIcs*, pages 235–243. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2008.
[36] E. Allen Emerson and Chin-Laung Lei. Modalities for model checking: Branching time logic strikes back. *Sci. Comput. Program.*, 8(3):275–306, June 1987.
[37] Paul Hunter and Anuj Dawar. Complexity bounds for regular games. In Joanna Jdrzejowicz and Andrzej Szepietowski, editors, *Mathematical Foundations of Computer Science 2005*, volume 3618 of *Lecture Notes in Computer Science*, pages 495–506. Springer Berlin Heidelberg, 2005.

[38] Catriel Beeri. On the membership problem for functional and multivalued dependencies in relational databases. *ACM Trans. Database Syst.*, 5(3):241–259, September 1980.
[39] Neil Immerman. Number of quantifiers is better than number of tape cells. *Journal of Computer and System Sciences*, 22(3):384 – 406, 1981.
[40] Krishnendu Chatterjee and Monika Henzinger. An $O(n^2)$ time algorithm for alternating Büchi games. In *Proceedings of the Twenty-third Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '12, pages 1386–1399. SIAM, 2012.
[41] E. Allen Emerson and Charanjit S. Jutla. The complexity of tree automata and logics of programs. *SIAM J. Comput.*, 29(1):132–158, September 1999.
[42] E.A. Emerson, C.S. Jutla, and A.P. Sistla. On model-checking for fragments of -calculus. In Costas Courcoubetis, editor, *Computer Aided Verification*, volume 697 of *Lecture Notes in Computer Science*, pages 385–396. Springer Berlin Heidelberg, 1993.
[43] Marcin Jurdziński. Deciding the winner in parity games is in up ∩ co-up. *Information Processing Letters*, 68(3):119 – 124, 1998.
[44] E. Allen Emerson and Chin-Laung Lei. Efficient model checking in fragments of the propositional mu-calculus (extended abstract). In *LICS*, pages 267–278. IEEE Computer Society, 1986.
[45] Robert McNaughton. Infinite games played on finite graphs. *Annals of Pure and Applied Logic*, 65(2):149 – 184, 1993.
[46] Sven Schewe. Solving parity games in big steps. In V. Arvind and Sanjiva Prasad, editors, *FSTTCS 2007: Foundations of Software Technology and Theoretical Computer Science*, volume 4855 of *Lecture Notes in Computer Science*, pages 449–460. Springer Berlin Heidelberg, 2007.
[47] Walter Ludwig. A subexponential randomized algorithm for the simple stochastic game problem. *Inf. Comput.*, 117(1):151–155, February 1995.
[48] M. Jurdziński, M. Paterson, and U. Zwick. A deterministic subexponential algorithm for solving parity games. *SIAM Journal on Computing*, 38(4):1519–1532, 2008.
[49] Jens Vöge and Marcin Jurdziński. A discrete strategy improvement algorithm for solving parity games. In E.Allen Emerson and Aravinda Prasad Sistla, editors, *Computer Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*, pages 202–215. Springer Berlin Heidelberg, 2000.
[50] Oliver Friedmann. An exponential lower bound for the latest deterministic strategy iteration algorithms. *Logical Methods in Computer Science*, 7(3), 2011.
[51] M.O. Rabin. Decidability of second-order theories and automata on infinite trees. *Trans. Amer. Math. Soc.*, 141:1–35, 1969.
[52] R. Hossley and Charles Rackoff. The emptiness problem for automata on infinite trees. In *SWAT (FOCS)*, pages 121–124. IEEE Computer Society, 1972.
[53] E. Allen Emerson and Charanjit S. Jutla. The complexity of tree automata and logics of programs. In *FOCS*, pages 328–337. IEEE Computer Society, 1988.
[54] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '89, pages 179–190, New York, NY, USA, 1989. ACM.
[55] Amir Pnueli. The temporal logic of programs. In *Foundations of Computer Science, 1977., 18th Annual Symposium on*, pages 46–57, Oct 1977.
[56] Moshe Y. Vardi and Pierre Wolper. Reasoning about infinite computations. *Inf. Comput.*, 115(1):1–37, November 1994.
[57] Rob Gerth, Doron Peled, Moshe Y. Vardi, and Pierre Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Proceedings of the Fifteenth IFIP WG6.1 International Symposium on Protocol Specification, Testing and Verification XV*, pages 3–18, London, UK, UK, 1996. Chapman & Hall, Ltd.
[58] Fabio Somenzi and Roderick Bloem. Efficient Büchi automata from LTL formulae. In *CAV 2000, LNCS 1855:247263*. Springer-Verlag, 2000.
[59] Paul Gastin and Denis Oddoux. Fast LTL to Büchi automata translation. In Gérard Berry, Hubert Comon, and Alain Finkel, editors, *Proceedings of the 13th International Conference on Computer Aided Verification (CAV'01)*, volume 2102 of *Lecture Notes in Computer Science*, pages 53–65, Paris, France, July 2001. Springer.

[60] E.Allen Emerson and Joseph Y. Halpern. Decision procedures and expressiveness in the temporal logic of branching time. *Journal of Computer and System Sciences*, 30(1):1 – 24, 1985.

[61] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In Dexter Kozen, editor, *Logics of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer Berlin Heidelberg, 1982.

[62] Moshe Y Vardi and Pierre Wolper. Automata-theoretic techniques for modal logics of programs. *J. Comput. Syst. Sci.*, 32(2):183–221, April 1986.

[63] Michael J. Fischer and Richard E. Ladner. Propositional dynamic logic of regular programs. *Journal of Computer and System Sciences*, 18(2):194 – 211, 1979.

[64] E. Allen Emerson. Temporal and modal logic. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science (Vol. B)*, pages 995–1072. MIT Press, Cambridge, MA, USA, 1990.

[65] John H. Reif. The complexity of two-player games of incomplete information. *J. Comput. Syst. Sci.*, 29(2):274–301, 1984.

[66] Orna Kupferman and Moshe Y. Vardi. Synthesis with incomplete information. In *Proc. ICTL'97*, 1997.

[67] Orna Kupferman and Moshe Y. Vardi. Synthesizing distributed systems. In *Proc. LICS'01*, July 2001.

[68] Bernd Finkbeiner and Sven Schewe. Uniform distributed synthesis. In *IEEE Symposium on Logic in Computer Science*, pages 321–330, June 2005.

[69] Fabio Mogavero, Aniello Murano, and Moshe Y. Vardi. Reasoning about strategies. In Kamal Lodaya and Meena Mahajan, editors, *FSTTCS*, volume 8 of *LIPIcs*, pages 133–144. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2010.

[70] Rajeev Alur, Thomas A. Henzinger, and Orna Kupferman. Alternating-time temporal logic. *J. ACM*, 49(5):672–713, 2002.

[71] Nir Piterman, Amir Pnueli, and Yaniv Sa'ar. Synthesis of reactive(1) designs. In E.Allen Emerson and KedarS. Namjoshi, editors, *Verification, Model Checking, and Abstract Interpretation*, volume 3855 of *Lecture Notes in Computer Science*, pages 364–380. Springer Berlin Heidelberg, 2006.

[72] A Pnueli, E Asarin, O Maler, and J Sifakis. Controller synthesis for timed automata. *Proc. System Structure and Control. Elsevier*, 1998.

[73] Rajeev Alur and Salvatore La Torre. Deterministic generators and games for LTL fragments. *ACM Trans. Comput. Logic*, 5(1):1–25, January 2004.

[74] Roderick Bloem, Krishnendu Chatterjee, Karin Greimel, ThomasA. Henzinger, and Barbara Jobstmann. Robustness in the presence of liveness. In Tayssir Touili, Byron Cook, and Paul Jackson, editors, *Computer Aided Verification*, volume 6174 of *Lecture Notes in Computer Science*, pages 410–424. Springer Berlin Heidelberg, 2010.

[75] Luca de Alfaro and Marco Faella. An accelerated algorithm for 3-color parity games with an application to timed games. In Werner Damm and Holger Hermanns, editors, *Computer Aided Verification*, volume 4590 of *Lecture Notes in Computer Science*, pages 108–120. Springer Berlin Heidelberg, 2007.

[76] Rüdiger Ehlers. Generalized Rabin(1) synthesis with applications to robust system synthesis. In Mihaela Bobaru, Klaus Havelund, Gerard Holzmann, and Rajeev Joshi, editors, *Proceedings of the 3rd NASA Formal Methods Symposium*, volume 6617 of *Lecture Notes in Computer Science*, pages 101–115. Springer-Verlag, 2011.

[77] S. Sohail and F. Somenzi. Safety first: A two-stage algorithm for ltl games. In *Formal Methods in Computer-Aided Design, 2009. FMCAD 2009*, pages 77–84, Nov 2009.

[78] O. Kupferman and M.Y. Vardi. Safraless decision procedures. In *Foundations of Computer Science, 2005. FOCS 2005. 46th Annual IEEE Symposium on*, pages 531–540, Oct 2005.

[79] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comput.*, 35(8):677–691, August 1986.

[80] Andreas Kuehlmann, Malay K Ganai, and Viresh Paruthi. Circuit-based boolean reasoning. In *Design Automation Conference, 2001. Proceedings*, pages 232–237. IEEE, 2001.

[81] Emmanuel Filiot, Naiyong Jin, and Jean-Franois Raskin. An antichain algorithm for ltl

realizability. In Ahmed Bouajjani and Oded Maler, editors, *Computer Aided Verification*, volume 5643 of *Lecture Notes in Computer Science*, pages 263–277. Springer Berlin Heidelberg, 2009.

[82] Wolfgang Thomas. On the synthesis of strategies in infinite games. In Ernst W. Mayr and Claude Puech, editors, *STACS 95*, volume 900 of *Lecture Notes in Computer Science*, pages 1–13. Springer Berlin Heidelberg, 1995.

[83] Rüdiger Ehlers. Symbolic bounded synthesis. In T. Touili, B. Cook, and P. Jackson, editors, *22nd International Conference on Computer Aided Verification*, volume 6174 of *LNCS*, pages 365–379. Springer Verlag, 2010.

[84] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic model checking without bdds. In W.Rance Cleaveland, editor, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1579 of *Lecture Notes in Computer Science*, pages 193–207. Springer Berlin Heidelberg, 1999.

[85] F. Copty, L. Fix, E. Giunchiglia, G. Kamhi, A. Tacchella, and M. Vardi. Benefits of bounded model checking at an industrial setting. In *Proceedings of 13th International Conference on Computer Aided Verification (CAV 2001), 18–22 July, Paris, France*, LNCS, pages 436–453. Springer Verlag, 2001.

[86] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, and Yunshun Zhu. Bounded model checking. *Advances in Computers*, 58:118–149, 2003.

[87] Bernd Finkbeiner and Sven Schewe. SMT-based synthesis of distributed systems. In *Proc. AFM*, 2007.

[88] Bernd Finkbeiner and Sven Schewe. Bounded synthesis. *International Journal on Software Tools for Technology Transfer*, 15(5-6):519–539, 2013.

[89] Bernd Finkbeiner and Swen Jacobs. Lazy synthesis. In *13th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2012)*, pages 219–234. Springer Verlag, 2012.

[90] Bernd Becker, Rüdiger Ehlers, Matthew Lewis, and Paolo Marin. Allqbf solving by computational learning. In S. Chakraborty and M. Mukund, editors, *10th International Symposium on Automated Technology for Verification and Analysis (ATVA)*, volume 7561 of *LNCS*, pages 370–384. Springer Verlag, 2012.

[91] Marc Herbstritt, Bernd Becker, and Christoph Scholl. Advanced sat-techniques for bounded model checking of blackbox designs. *Fifth International Workshop on Microprocessor Test and Verification (MTV'04)*, 0:37–44, 2006.

[92] Bernd Finkbeiner and Leander Tentrup. Detecting unrealizable specifications of distributed systems. In Erika Ábrahám and Klaus Havelund, editors, *TACAS*, volume 8413 of *Lecture Notes in Computer Science*, pages 78–92. Springer, 2014.

[93] ARM Ltd. Amba$^{\text{TM}}$ specification (rev. 2). Available at `http://www.arm.com`, 1999.

[94] Roderick Bloem, Stefan Galler, Barbara Jobstmann, Nir Piterman, Amir Pnueli, and Martin Weiglhofer. Specify, compile, run: Hardware from psl. *Electron. Notes Theor. Comput. Sci.*, 190(4):3–16, November 2007.

[95] Yashdeep Godhal, Krishnendu Chatterjee, and ThomasA. Henzinger. Synthesis of amba ahb from formal specification: a case study. *International Journal on Software Tools for Technology Transfer*, 15(5-6):585–601, 2013.

[96] H. Kress-Gazit, G.E. Fainekos, and G.J. Pappas. Temporal-logic-based reactive mission and motion planning. *Robotics, IEEE Transactions on*, 25(6):1370–1381, Dec 2009.