ACTL ∩ LTL Synthesis[⋆]

Rüdiger Ehlers

Reactive Systems Group
Saarland University

Abstract. We study the synthesis problem for specifications of the common fragment of ACTL (computation tree logic with only universal path quantification) and LTL (linear-time temporal logic). Key to this setting is a novel construction for translating properties from LTL to very-weak automata, whenever possible. Such automata are structurally simple and thus amenable to optimizations as well as symbolic implementations.
Based on this novel construction, we describe a synthesis approach that inherits the efficiency of generalized reactivity(1) synthesis [27], but is significantly richer in terms of expressivity.

1 Introduction

Synthesizing reactive systems from functional specifications is an ambitious challenge. It combines the correctness assurance that systems obtain after model checking with the advantage to skip the manual construction step for the desired system. As a consequence, a rich line of research has emerged, witnessed by the fact that recently, off-the-shelf tools for this task have become available.

A central question in synthesis is: *what is the right specification language that allows us to tackle the synthesis problem for its members efficiently, while still having enough expressivity to capture the specifications that system designers want to write?*

Some recent approaches focused on supporting full linear-time temporal logic as the specification language. While the synthesis problem for such specifications was shown to be 2EXPTIME-complete, by focusing on specifications of the form that engineers tend to write, significant progress could recently be obtained for full LTL [17, 13]. Still, it is not hard to write small specifications that cannot be tackled by such tools.

At the same time, there are numerous techniques that trade the high expressivity of logics such as LTL against the computational advantages of only having to deal with structurally simpler specifications. A prominent approach of this kind is *generalized reactivity(1) synthesis* [27]. It targets specifications that consist of some set of assumptions (which we can assume the environment of the system to fulfill) and some set of guarantees that the system needs to fulfill. Both assumptions and guarantees can contain only safety properties that relate the input and output in one computation cycle with the input and output in the next computation cycle and basic liveness properties over current input and output. In order to encode more complex properties, the output of the system

[⋆] This work was supported by the DFG as part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS).

to be designed can be widened and the additional bits can be used to stitch together more complex properties. Getting such an encoding right and efficient is manual and cumbersome work, which is why Somenzi and Sohail coined the term "*pre-synthesis*" for such an operation [28, 11].

It is apparent that there is a desperate need for a sweet spot between the high expressivity but low performance that full LTL synthesis approaches offer, and fast but low-level synthesis approaches such as generalized reactivity(1) synthesis, where currently, pre-synthesis is crucial to its performance.

In this paper, we present *ACTL ∩ LTL synthesis* as a solution to this problem. Our approach targets specifications of the form $\bigwedge_{a \in \mathrm{Assumptions}} a \rightarrow \bigwedge_{g \in \mathrm{Guarantees}} g$, where all assumptions and guarantees are written in LTL, with the restriction that they must also be representable in ACTL, i.e., computation tree logic with only universal path quantification. We reduce the synthesis problem for such specifications to solving symbolically represented three-color parity games, which is the reasoning framework from which also generalized reactivity(1) synthesis takes its good efficiency. In particular, such games can be solved in time quadratic in the number of positions (see, e.g., [1]).

The reason why ACTL ∩ LTL is such an interesting fragment for synthesis is the fact that the fragment has *universal very-weak automata* as the characterizing automaton class. These automata do not only allow the application of simple, yet effective minimization algorithms, but give rise to a straight-forward efficient symbolic encoding into binary decision diagrams (BDDs), without the need for pre-synthesis. Alternatively, other symbolic data structures such as *anti-chains* [16] can also be used, but for the simplicity of the initial evaluation of the approach in this paper, we use BDDs.

For best performance in solving the parity games that we build in our approach, we present a novel construction that defers choosing the assumption and guarantee parts to be satisfied next to the system player and the environment player, respectively. This keeps the number of iterations that need to be performed in the fixed-point based game solving process small and leads to short computation times of the game solving process.

The contribution of this paper is threefold. First of all, it describes a new efficient synthesis workflow for the common fragment of ACTL and LTL. Secondly, it describes the first algorithm for translating an LTL formula that lies in this common fragment into its characterizing automaton class, i.e., universal very-weak automata. As a corollary, we obtain a translation algorithm from LTL to ACTL, whenever possible. Third, we introduce a technique to speed up the game solving process for generalized reactivity(1) games by letting the two players in the game choose the next obligation for the respective other player instead of using counters as in previous approaches.

We start with preliminaries in Sect. 2, where we discuss the basic properties of very-weak automata. Then, we describe the construction to obtain universal very-weak automata from LTL formulas that are also representable in ACTL. Afterwards, we present the smart reduction of our synthesis problem to three-color parity games in Sect. 4. Section 5 then discusses the twists and tricks for solving parity games symbolically in an efficient way and describes how a winning strategy that represents an implementation satisfying the specification can be extracted. Finally, Sect. 6 contains an experimental evaluation of the approach using a prototype toolset for the overall workflow. We conclude in Sect. 7.

2 Preliminaries

Basics: Given a (finite) alphabet Σ, we denote the sets of finite and infinite words of Σ as Σ^* and Σ^ω, respectively. Sets of words are called *languages*. A useful tool for representing languages over finite words are regular expressions, and ω-regular expressions are regular expressions that are enriched by the $(\cdot)^\omega$ operator, which denotes infinite repetition. This way, languages over infinite words can be expressed.

Given some monotone function $f : 2^X \rightarrow 2^X$ for some finite set X, we define $\mu^0.f = \emptyset$, $\nu^0.f = X$ and for every $i > 0$, set $\mu^i.f = (f \circ \mu^{i-1}.f)$ and $\nu^i.f = (f \circ \nu^{i-1}.f)$. For a monotone function f and finite X, it is assured that the series $\mu^0.f, \mu^1.f, \mu^2.f \ldots$ and $\nu^0.f, \nu^1.f, \nu^2.f \ldots$ converge to some limit functions, which we denote by $\mu.f$ and $\nu.f$, respectively.

Automata: For reasoning about (ω-)regular languages, *automata* are a suitable tool. In this paper, we will be concerned with *deterministic*, *non-deterministic*, *non-deterministic very-weak* and *universal very-weak* automata over finite and infinite words. For all of these types, the automata are described by tuples $\mathcal{A} = (Q, \Sigma, Q_0, \delta, F)$ with the set of states Q, the alphabet Σ, the set of initial states $Q_0 \subseteq Q$, and the transition function $\delta : Q \times \Sigma \rightarrow 2^Q$. For non-deterministic or deterministic automata, $F \subseteq Q$ is called the set of *accepting states*, whereas for universal automata, $F \subseteq Q$ denotes the set of *rejecting* states. For deterministic automata, we require that $|Q_0| = 1$ and that for every $(q, x) \in Q \times \Sigma$, we have $|\delta(q, x)| \leq 1$. For very-weak automata, we require them to have an order $f : Q \rightarrow \mathbb{N}$ on the states such that for every transition from a state q to a state q' for some some $x \in \Sigma$ (i.e., $q' \in \delta(q, x)$), if $q' \neq q$, then $f(q') > f(q)$. Figure 1 contains examples of very-weak automata. Intuitively, the order requires the automaton to be representable in a figure such that all non-self-loop transitions lead from top to bottom.

Given a word $w = w_0 w_1 w_2 \ldots w_n \in \Sigma^*$, we say that $\pi = \pi_0 \pi_1 \ldots \pi_{n+1}$ is a finite run for $\mathcal{A}$ and w if $\pi_0 \in Q_0$ and for $0 \leq i \leq n$, $\pi_{i+1} \in \delta(\pi_i, w_i)$. Likewise, for a word $w = w_0 w_1 w_2 \ldots \in \Sigma^\omega$, we say that $\pi = \pi_0 \pi_1 \ldots$ is an infinite run for $\mathcal{A}$ and w if $\pi_0 \in Q_0$ and for all $i \in \mathbb{N}$, $\pi_{i+1} \in \delta(\pi_i, w_i)$.

A non-deterministic (NFA), non-deterministic very-weak (NVWF) or deterministic (DFA) automaton over finite words accepts all finite words that have some run that ends in an accepting state. A universal automaton over finite words accepts all finite words for which all runs do not end in a rejecting state. A non-deterministic automaton over infinite words accepts all infinite words that have some run that visits accepting states infinitely often. A universal very-weak automaton over infinite words (UVW) accepts all infinite words for which all runs visit rejecting states only finitely often.

We say that two automata are equivalent if they accept the same set of words. This set of words is also called their *language*. We define the *language of a state* q to mean the language of the automaton that results from setting the initial states to $\{q\}$. The functions $\hat{\delta} : 2^Q \times 2^X \rightarrow 2^Q$ and $\hat{\delta}^* : 2^Q \times 2^X \rightarrow 2^Q$ with $\hat{\delta}(Q', X) = \bigcup_{\{q' \in Q', x \in X\}} \delta(q', x)$ and $\hat{\delta}^*(Q', X) = \{q' \in Q \mid \exists k \in \mathbb{N}, x_1, x_2, \ldots, x_k \in X, q_1, q_2, \ldots, q_{k+1} \in Q.(q_1 \in Q' \wedge q_k = q' \wedge \forall 1 \leq i \leq k.\, q_{i+1} \in \delta(q_i, x_i))\}$ will simplify the presentation in Sect. 3. Deterministic automata over finite words also appear as *distance automata* in this paper. The only difference to non-distance automata is the fact that for these, we have $\delta : Q \times \Sigma \rightarrow 2^{Q \times \{0,1\}}$. We assign with each of their runs the *accumulated cost*, obtained by adding all of the second components of the transition target tuples for the transitions along the run. The cost of a word is the minimal cost of an accepting run.

Labeled parity games: A parity game is defined as a tuple $\mathcal{G} = (V_0, V_1, \Sigma_0, \Sigma_1, E_0, E_1, v_0, c)$ with the game position sets V_0 and V_1 for player 0 and player 1, respectively, the *action sets* Σ_0 and Σ_1, the edge functions $E_0 : V_0 \times \Sigma_0 \rightarrow V_1$ and $E_1 : V_1 \times \Sigma_1 \rightarrow V_0$, the initial position $v_0 \in V_0$, and the coloring function $c : (V_0 \uplus V_1) \rightarrow \mathbb{N}$.

A decision sequence in $\mathcal{G}$ is a sequence $\rho = \rho^0_0 \rho^1_0 \rho^0_1 \rho^1_1 \ldots$ such that for all $i \in \mathbb{N}$, $\rho^0_i \in \Sigma_0$ and $\rho^1_i \in \Sigma_1$. A decision sequence ρ induces an infinite play $\pi = \pi^0_0 \pi^1_0 \pi^0_1 \pi^1_1 \ldots$ if $\pi^0_0 = v_0$ and for all $i \in \mathbb{N}$ and $p \in \{0, 1\}$, $E_p(\pi^p_i, \rho^p_i) = \pi^{1-p}_{i+p}$.

Given a play $\pi = \pi^0_0 \pi^1_0 \pi^0_1 \pi^1_1 \ldots$, we say that π is winning for player 1 if $\max\{c(v) \mid v \in V_0 \uplus V_1, v \in \inf(\pi)\}$ is even for the function $\inf$ mapping a sequence onto the set of elements that appear infinitely often in the sequence. If a play is not winning for player 1, it is winning for player 0.

Given some parity game $\mathcal{G} = (V_0, V_1, \Sigma_0, \Sigma_1, E_0, E_1, v_0, c)$, a strategy for player 0 is a function $f_0 : (\Sigma_0 \times \Sigma_1)^* \rightarrow \Sigma_0$. Likewise, a strategy for player 1 is a function $f_1 : (\Sigma_0 \times \Sigma_1)^* \times \Sigma_0 \rightarrow \Sigma_1$. In both cases, a strategy maps prefix decision sequences to an action to be chosen next. A decision sequence $\rho = \rho^0_0 \rho^1_0 \rho^0_1 \rho^1_1 \ldots$ is said to be in correspondence to f_p for some $p \in \{0, 1\}$ if for every $i \in \mathbb{N}$, we have $\rho^p_i = f_p(\rho^0_0 \rho^1_0 \ldots \rho^{1-p}_{i+p-1})$. A strategy is winning for player p if all plays in the game that are induced by some decision sequence that is in correspondence to f_p are winning for player p. It is a well-known fact that for parity games, there exists a winning strategy for precisely one of the players (see, e.g., [26, 22]).

Labeled parity games for synthesis: Parity games are a computation model for systems that interact with their environment. For the scope of this paper, let us assume that player 0 represents the *environment* of a system that we want to synthesize, and player 1 represents the system itself. The action set of player 0 corresponds to the inputs to the system and the action set of player 1 corresponds to the output. Given a language L over infinite words for the desired properties of a system, the main idea when building a parity game for synthesis is to ensure that the decision sequences that induce winning plays are the ones that, when read as words, are in L. If the game is then found to be winning for the system player, we can take a *strategy* for that player to win the game and read it as a *Mealy* automaton that is guaranteed to satisfy the specification. Note that all constructions in this paper can equally be used for a *Moore* automaton computation model. The two players then swap roles in this case.

Linear-time temporal logic: Linear-time temporal logic (LTL) is a popular formalism to describe properties of systems to be synthesized or verified. LTL formulas are built inductively from atomic propositions in some set AP and sub-formulas using the Boolean operators $\neg$, $\vee$, $\wedge$, and the temporal operators X, F, G, and U. Given an infinite word $w = w_0 w_1 w_2 \in (2^{\mathrm{AP}})^\omega$, a LTL formula over AP either holds on w or not. The words for which an LTL formula holds are also called its *models*. A full definition of LTL can be found in [12, 16, 18].

Properties of very-weak automata

As foundation for the constructions of the sections to come, we discuss some properties of very-weak automata over finite and infinite words here. Given two automata, we call computing a third automaton that represents the set of words that are accepted by both automata *taking their conjunction*, while *taking their disjunction* refers to computing a third automaton that accepts all words that are accepted by either of the two input automata.

Proposition 1. *Universal and non-deterministic very-weak automata over infinite and finite words are closed under disjunction and conjunction. Given two very-weak automata $\mathcal{A}$ and $\mathcal{A}'$ with state sets Q and Q', we can compute their disjunctions and conjunctions in polynomial time, with the following state counts of the results:*

1. *for universal automata and taking the conjunction: $|Q| + |Q'|$ states,*
2. *for non-deterministic automata and taking the disjunction: $|Q| + |Q'|$ states,*
3. *for universal automata and taking the disjunction: $|Q| \cdot |Q'|$ states, and*
4. *for non-deterministic automata and taking the conjunction: $|Q| \cdot |Q'|$ states.*

Proof. For the first two cases, the task can be accomplished by just merging the state sets and transitions. For cases 3 and 4, a standard product construction can be applied, with defining those states in the product as rejecting/accepting for which both corresponding states in the factor automata are rejecting/accepting, respectively [20]. □

Proposition 2. *Every very-weak automaton has an equivalent one of the same type for which no accepting/rejecting state has a non-self-loop outgoing edge (called the separated form of the automaton henceforth).*

Proof. Duplicate every accepting/rejecting state in the automaton and let the duplicate have the same incoming edges. Then, mark the original copy of the state as non-accepting/non-rejecting. The left part of Fig. 1 shows an example of such a state duplication. □

The fact that every automaton has a separated form allows us to decompose it into a set of so-called *simple chains*:

Definition 1. *Given an alphabet Σ, we call a subset Q' of states of an automaton over Σ a simple chain if there exists a transition order on Q', i.e., a bijective function $f : Q' \to \{1, \ldots, |Q'|\}$ such that:*

- *only the state q with $f(q) = 1$ is initial,*
- *only the state q with $f(q) = |Q'|$ is accepting/rejecting,*
- *there is no transition in the automaton between a state in Q' and a state not in Q',*
- *for every transition from q to q' in the automaton, $f(q) \leq f(q') \leq f(q) + 1$.*

Furthermore, regular expressions that are an unnested concatenation of elements of the form A, A^*, and A^ω for $A \subseteq \Sigma$ are called vermicelli.

As an example, the right-most sequence of states in Fig. 1 is a simple chain and can equivalently be represented as the vermicelli $\Sigma^* a(\overline{b})^* b(\overline{c})^\omega$. Note that every vermicelli can be translated to a language-equivalent set of simple chains.

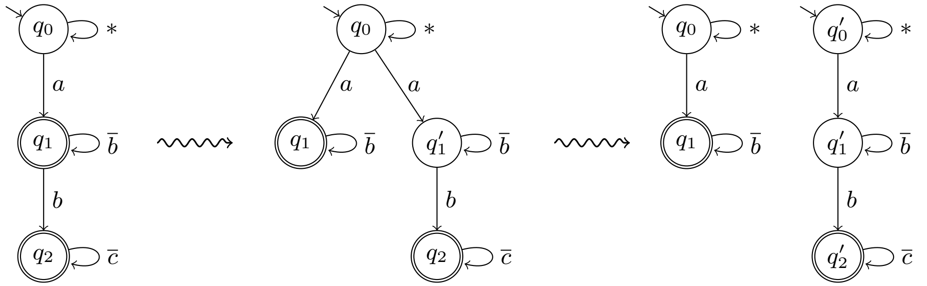

Fig. 1. Example for converting a UVW into separated form and subsequently decomposing it into simple chains. The automata in this example are equivalent to the LTL formula $\mathsf{G}(a \rightarrow \mathsf{XF}(b \wedge \mathsf{XF}c))$. We use Boolean combinations of atomic propositions and their negation as edge labels here. For example, $\overline{b}$ refers to all elements $x \in \Sigma = 2^{\mathrm{AP}}$ for which $b \notin x$. Rejecting states are doubly-circled.

Proposition 3. *Every very-weak automaton can be translated to a form in which it only consists of simple chains.*

Proof. Convert the very-weak automaton into separated form and enumerate all paths to leaf nodes along with the self-loops that might possibly be taken. For every of these paths, construct a simple chain. □

3 Translating LTL formulas into UVWs

Universal very-weak automata (UVW) were identified as a characterizing automaton class for the intersection of ACTL and LTL by Maidl [25]. She also described an algorithm to check for a given ACTL formula if it lies in the intersection. For the LTL case, Maidl defined a syntactic fragment of it, named $\mathrm{LTL}^{\mathrm{det}}$, whose expressivity coincides with that of ACTL $\cap$ LTL. However, she did not show how to translate an LTL formula into this fragment whenever possible, and the fragment itself is cumbersome to use, as it essentially requires the specifier to describe the structure of a UVW in LTL, and is not even closed under disjunction, although UVW are. Thus, for all practical means, the question how to check for a given LTL formula if it is contained in ACTL $\cap$ LTL remained open.

When synthesizing a system, the designer of the system specifies the desired sequence of events, for which linear-time logics are more intuitive to use than branching-time logics. Thus, to use the advantage of universal very-weak automata in actual synthesis tool-chains, the ability to translate from LTL to UVW is highly desirable.

Recently, Bojańczyk [4] gave an algorithm for testing the membership of the set of models of an LTL formula in ACTL $\cap$ LTL after the LTL formula has been translated to a deterministic parity automaton. However, the algorithm cannot generate a universal very-weak automaton (UVW) from the parity automaton in case of a positive answer. The reason is that the algorithm is based on searching for so-called *bad patterns* in the automaton. If none of these are present, the deterministic parity automaton is found to be convertible, but we do not obtain any information about how a UVW for the property might look like. Here, we reduce the problem of constructing a UVW for a given ω-regular language to a sequence of problems over automata for finite words. We modify a procedure by Hashiguchi [19] that builds a distance automaton to check if a given language over finite words can be decomposed into a set of vermicelli (see Def. 1). Our modification adds a component to keep track of vermicelli already found. This way, by iteratively searching for vermicellis of increasing length in the language, we eventually find them all and obtain a full language decomposition.

Since Maidl [25, Lemma 11] described a procedure to translate a UVW to an equivalent ACTL formula, we obtain as a corollary also a procedure to translate from LTL to ACTL, whenever possible.

3.1 The case of automata over infinite words

We have seen that every UVW can be translated to a separated UVW. In a separated UVW, we can distinguish rejecting states by the set of alphabet symbols for which the states have self-loops. If two rejecting states have the same set, we can merge them without changing the language of the automaton. As a corollary, we obtain that a UVW can always be modified such that it is in separated form and has at most $2^{|\Sigma|}$ rejecting states. We will see in this section that obtaining a UVW for a given language L over some alphabet Σ can be done by finding a suitable *decomposition* of the set of words that are not in L among these up to $2^{|\Sigma|}$ rejecting states, and then constructing the rest of the UVW such that words that are mapped to some rejecting state in the decomposition induce runs that eventually enter that rejecting state and stay there forever.

Definition 2. *Given a language L over infinite words from the alphabet Σ, we call a function $f : 2^\Sigma \to 2^{\Sigma^*}$ an end-component decomposition of L if $L = \Sigma^\omega \setminus \bigcup_{X \subseteq \Sigma}(f(X) \cdot X^\omega)$. We call f a maximal end-component decomposition of L if for every $X \subseteq \Sigma$, $f(X) = \{w \in \Sigma^* \mid w \cdot X^\omega \cap L = \emptyset\}$.*

Definition 3. *Given a separated UVW $\mathcal{A} = (Q, \Sigma, Q_0, \delta, F)$ and an end-component decomposition f, we say that f corresponds to $\mathcal{A}$ if for $(q_1, X_1), \ldots, (q_m, X_m)$ being the rejecting states and alphabet symbols under which they have self-loops, we have:*

- *for all $i \neq j$, $X_i \neq X_j$;*
- *for all $1 \leq i \leq m$: $f(X_i) = \{w_0 w_1 \ldots w_k \in \Sigma^* \mid q_i \in \hat{\delta}(\ldots \hat{\delta}(\hat{\delta}(Q_0, \{w_0\}), \ldots), \{w_k\})\}$;*
- *for all $X \subseteq \Sigma$ with $X \notin \{X_1, \ldots, X_m\}$, we have $f(X) = \emptyset$.*

As an example, the end-component decomposition that corresponds to the UVW in the middle part of Fig. 1 is a function f with $f(\overline{b}) = \Sigma^* a (\overline{b})^*$, $f(\overline{c}) = \Sigma^* a(\overline{b})^* b(\overline{c})^*$, and $f(X) = \emptyset$ for $X \neq \overline{b}$ and $X \neq \overline{c}$. The decomposition is not maximal as, for example, the word $\{a\}\emptyset^\omega$ is not in the language of the automaton, but we have $\{a\} \notin f(\{\emptyset\}) = \emptyset$.

By the definition of corresponding end-components, every separated UVW has one unique corresponding end-component decomposition. On the other hand, every language has one maximal end-component decomposition. The key result that allows us to reduce finding a UVW for a given language to a problem on finite words combines these two facts:

Lemma 1. *Let L be a language that is representable by a universal very-weak automaton. Then, L is also representable as a separated UVW whose corresponding end-component decomposition is the maximal end-component decomposition of L.*

Proof. Let a UVW be given whose end-component decomposition f is not maximal. The decomposition can be made maximal by taking $f'(X) = \bigcup_{X' \supseteq X} f(X')$ for every $X \subseteq \Sigma$, without changing the language. Building a corresponding UVW only requires taking disjunctions of parts of the original UVW. Since we know that UVW are closed under disjunction, it is assured that there also exists a UVW that corresponds to f'. □

Thus, in order to obtain a UVW for a given language $L \subset \Sigma^\omega$, we can compute the maximal end-component decomposition f' of L, and for every end component $X \subseteq \Sigma$, compute a non-deterministic very-weak automaton over finite words for $f'(X)$.

Starting with an LTL formula, we can thus translate it to a UVW (if possible) as follows: first of all, we translate the LTL formula to a deterministic Büchi automata (see, e.g., [9] for an overview). Note that as the expressivities of LTL and deterministic Büchi automata are incomparable, this is not always possible. If no translation exists, we however know that there also exists no UVW for the LTL formula, as all languages representable by UVW are also representable by deterministic Büchi automata. After we have obtained the Büchi automaton, we compute for every possible end-component $X \subseteq \Sigma$ from which states S in the automaton every word ending with X^ω is rejected. This is essentially a model checking problem over an automaton with Büchi acceptance condition. This way, for each end component, a deterministic automaton over finite words with S as the set of accepting states then represents the prefix language.

3.2 Decomposing a language over finite words into a non-deterministic very-weak automaton

This problem of deciding whether there exists a non-deterministic very-weak automaton for a language over finite words is widely studied in the literature. However, constructive algorithms that compute such an automaton are unknown. Hashiguchi studied a more general version of the problem in [19]. His solution is based on computing the maximal distance of an accepted word in a distance automaton. Bojańczyk [4] recently gave a simpler algorithm.

Here, we build on the classical construction by Hashiguchi and modify it in order to be constructive. We describe an iterative algorithm that successively searches for vermicelli in the language to be analyzed. In a nutshell, this is done by searching for accepting words of minimal distance in a distance automaton. Whenever a new vermicelli is found, the automaton is modified in order not to accept words that are already covered by vermicelli that have been found before. At the same time, the new vermicelli can be read from the state sequence in the accepting run. The distance automaton is built as follows.

Definition 4. *Given a DFA $\mathcal{A} = (Q^\mathcal{A}, \Sigma, Q_0^\mathcal{A}, \delta^\mathcal{A}, F^\mathcal{A})$ for the language to be analyzed and a NVWF $\mathcal{B} = (Q^\mathcal{B}, \Sigma, Q_0^\mathcal{B}, \delta^\mathcal{B}, F^\mathcal{B})$ for the vermicelli already found, the non-deterministic vermicelli-searching distance automaton over finite words $\mathcal{D} = (Q, \Sigma, Q_0, \delta, F)$ is defined as follows:*

$$
\begin{aligned}
Q &= 2^{Q^{\mathcal{A}}} \times 2^{\Sigma} \times \mathbb{B} \times 2^{Q^{\mathcal{B}}} \\
Q_0 &= \{(Q_0^{\mathcal{A}}, \emptyset, \mathbf{false}, Q_0^{\mathcal{B}})\} \\
F &= \{(S, X, b, R) \mid S \subseteq F^{\mathcal{A}}, (R \cap F^{\mathcal{B}}) = \emptyset\} \\
\delta((S, X, b, R), x) &= \{((S, X, b, R'), 0) \mid R' = \hat{\delta}^{\mathcal{B}}(R, \{x\}), x \in X, b = \mathbf{true}\} \\
&\cup \{((S', X', \mathbf{true}, R'), 1) \mid R' = \hat{\delta}^{\mathcal{B}}(R, \{x\}), x \in X', S' = \hat{\delta}^{\mathcal{A},*}(S, X')\} \\
&\cup \{((S', X', \mathbf{false}, R'), 1) \mid R' = \hat{\delta}^{\mathcal{B}}(R, \{x\}), x \in X', S' = \hat{\delta}^{\mathcal{A}}(S, X')\} \\
&\textit{for all } (S, X, b, R) \in Q, x \in \Sigma
\end{aligned}
$$

The states in a vermicelli-searching automaton $\mathcal{D}$ are four-tuples (S, X, b, R) such that X and b represent an element in a vermicelli, where b tells us if the current vermicelli element X is starred. During a run, we track in S in which states in $\mathcal{A}$ we can be in after reading some word that is accepted by a vermicelli represented by the vermicelli elements observed in the X and b state components along the run of $\mathcal{D}$ so far. Whenever we have $S \subseteq F^{\mathcal{A}}$, then we know that all these words are accepted by $\mathcal{A}$. At the same time, the R component simulates all runs of the NVWF $\mathcal{B}$, and the definition of F ensures that no word that is in the language of $\mathcal{B}$ is accepted by $\mathcal{D}$. Thus, $\mathcal{D}$ can only find vermicelli that contain some word that is not accepted by $\mathcal{B}$ in their language. Transitions with cost 1 represent moving on to the next vermicelli element.

Theorem 1. *Let $\mathcal{A}$ be an DFA, $\mathcal{B}$ be a NVWF and $\mathcal{D}$ be the corresponding vermicelli-searching distance automaton. We have:*

- $\mathcal{L}(\mathcal{D}) = \mathcal{L}(\mathcal{A}) \setminus \mathcal{L}(\mathcal{B})$
- *Let $\mathcal{L}(\mathcal{A})$ contain a vermicelli $V = A_1 \ldots A_k$, where every A_i is either of the form X^* or X for some $X \subseteq \Sigma$. If V is not covered by $\mathcal{L}(\mathcal{B})$, then $\mathcal{D}$ accepts some word w that is a model of V with a run of distance k. Along this run, the first three state components only change during transitions with a cost of 1, and the second and third component in between changes describe the alphabet symbol sets in the vermicelli and whether the vermicelli elements are starred or not.*

As a consequence, since every UVW of size n can be described by a set of vermicelli in which each vermicelli is of length at most n, we can compute a UVW representation of $\mathcal{A}$ by using Algorithm 1. Note that the algorithm does not terminate if $\mathcal{A}$ cannot be represented as a very-weak automaton. Since we can however apply the algorithm by Bojańczyk [4] beforehand to verify the translatability, this imposes no problem.

4 Reduction of the Synthesis Problem to Parity Games

In this section, we explain how to reduce the synthesis problem for specifications of the form $\bigwedge_{a \in \text{Assumptions}} a \rightarrow \bigwedge_{g \in \text{Guarantees}} g$ (or shorter, in assumptions→guarantees form), for which each of the assumptions and guarantees is in the common fragment of ACTL and LTL, to solving a parity game. We have discussed in the previous section

Algorithm 1 Translating a DFA $\mathcal{A}$ into a non-deterministic very-weak automaton $\mathcal{B}$.

```
$\mathcal{B} = (\emptyset, \Sigma, \emptyset, \emptyset, \emptyset)$
**repeat**
    $\mathcal{D} =$ vermicelli searching automaton for $\mathcal{A}$ and $\mathcal{B}$
    $r =$ accepting run of minimal distance in $\mathcal{D}$
    **if** $r$ was found **then**
        Add $r$ as vermicelli to $\mathcal{B}$
    **end if**
**until** $\mathcal{L}(\mathcal{D}) = \emptyset$
```

how one assumption or guarantee can be converted to a UVW. As the conjunction of two UVW can be taken by just merging the state sets and the initial states, we also know how to compute *one* UVW for *all* of the assumptions and *one* UVW for *all* of the guarantees. So it remains to be discussed how we combine these two UVW into a game that captures the overall specification.

Bloem et al. [1] describe a way to translate a specification of the assumptions→guarantee form, in which all assumptions and guarantees are in form of deterministic Büchi automata, into a three-color parity game. Essentially, the construction splits the process by converting the assumptions and guarantees to a so-called generalized reactivity(1) game, and then modifying the game structure and adjusting the winning condition to three-color parity. When converting the game, assumption and guarantee pointers represent which assumption and guarantee is observed next for satisfaction. The pointers increment one-by-one, which makes the game solving process a tedious task; for example, if it is the last guarantee (in some assumed order) that the system cannot satisfy, then during the game solving process, this information has to be propagated through all the other pointer values before the process can terminate.

As a remedy, we describe an improved construction here, and let the two players set the pointers. This way, the winning player can set the assumption or guarantee pointer to the problematic assumption or guarantee early in the play, which reduces the time needed for game solving. The game only has colors other than 0 for positions of the environment player, and the states are described as six-tuples. The first two tuple components describe in which states the assumption and guarantee UVW are, followed by the assumption and guarantee pointers that are updated by the system and environment players, respectively. The last two components are Boolean flags that describe whether recently, the assumption (guarantee) state that the respective pointer points to has been left, or the system (environment) player has changed her pointer value, respectively, which is then reflected in the color of the game position. On a formal level, the parity game is built as follows:

Definition 5. *Let $\mathcal{A}^a = (Q^A, \Sigma, Q^A_0, \delta^A, F^A)$ and $\mathcal{A}^g = (Q^G, \Sigma, Q^G_0, \delta^G, F^G)$ be two UVW that represent assumptions and guarantees, and Σ_I and Σ_O be sets such that $\Sigma = \Sigma_I \times \Sigma_O$. Without loss of generality, let furthermore $F^A = \{1, \ldots, m\}$ and $F^G = \{1, \ldots, n\}$. We define the induced synthesis game as a parity game $\mathcal{G} = (V_0, V_1, \Sigma_I, \Sigma_O, E_0, E_1, v_0, c)$ with:*

$$V_0 = 2^{Q^A} \times 2^{Q^G} \times \{1, \ldots, m\} \times \{1, \ldots, n\} \times \mathbb{B} \times \mathbb{B}$$

$$
\begin{aligned}
V_1 &= V_0 \times \Sigma_I \\
E_0 &= \{((S^A, S^G, d^A, d^G, b^A, b^G), x) \mapsto (S^A, S^G, d^A, d'^G, \mathbf{false}, b'^G, x) \mid x \in \Sigma_I, \\
&\qquad (d^G = d'^G) \vee (b'^G = \mathbf{true})\} \\
E_1 &= \{((S^A, S^G, d^A, d^G, b^A, b^G, x), y) \mapsto (S'^A, S'^G, d'^A, d^G, b'^A, b'^G) \mid y \in \Sigma_O, \\
&\qquad S'^A = \delta^A(S^A, (x, y)),\ S'^G = \delta^G(S^G, (x, y)), \\
&\qquad b'^G = (b^G \vee (d^G \notin S'^G) \vee (d^G \notin \delta^G(d^G, (x, y)))), \\
&\qquad b'^A = ((d^A \neq d'^A) \vee (d^A \notin S^A) \vee (d^A \notin \delta^A(d^A, (x, y))))\} \\
v_0 &= (Q_0^A, Q_0^G, 1, 1, \mathbf{false}, \mathbf{false}) \\
c &= \{V_1 \cup \{(q^A, q^G, d^A, d^G, b^A, b^G) \mid \neg b^A \wedge \neg b^G\} \mapsto 0, \{(q^A, q^G, d^A, d^G, \\
&\qquad b^A, b^G) \mid b^A \wedge \neg b^G\} \mapsto 1, \{(q^A, q^G, d^A, d^G, b^A, b^G) \mid b^G\} \mapsto 2\}
\end{aligned}
$$

For the central correctness claim of this construction, we need some more notation. Given a play $\pi = \pi_0^0 \pi_0^1 \pi_1^0 \pi_1^1 \ldots$ for a decision sequence $\rho = \rho_0^0 \rho_0^1 \rho_1^0 \rho_1^1 \ldots$ in the game, we say that a state $q \in Q^A$ is *left* at position $k \in \mathbb{N}$ if for $\pi_k^1 = (S_1^A, S_1^G, d_1^A, d_1^G, b_1^A, b_1^G, \rho_{k-1}^0)$ and $\pi_{k+1}^0 = (S_2^A, S_2^G, d_2^A, d_2^G, b_2^A, b_2^G)$, we have $q \notin S_1^A$ or $q \notin \delta(q, (\rho_k^0, \rho_k^1))$. The construction of $\mathcal{G}$ assures that this is the case whenever any run of the assumption automaton corresponding to the first k choice pairs in the decision sequence leaves state q in the $k+1$th round or is not in state q in the kth round. The case for the guarantee automaton is analogous. We say that a player *rotates* through the possible pointer values if whenever the state that the pointer refers to is left, the player increases it to the next possible value. In case the highest value is reached, the pointer is set to 1 instead.

Theorem 2. *Let $\mathcal{A}^a$ and $\mathcal{A}^g$ be two UVWs over the alphabet $\Sigma = \Sigma_I \times \Sigma_O$, and $\mathcal{G}$ be the induced synthesis game by Def. 5. The winning strategies for player 1 ensure that along any decision sequence that corresponds to the strategy and in which the input player rotates though the guarantee pointer values, either the sequence is not accepted by $\mathcal{A}^a$, or the sequence is accepted by $\mathcal{A}^g$. Furthermore, every Mealy machine with the input Σ_I and output Σ_O for which along any of its runs, the run is either rejected by $\mathcal{A}^a$ or accepted by $\mathcal{A}^g$ induces a winning strategy in $\mathcal{G}$ by having player 1 rotate through the possible assumption pointer values.*

The main message of Theorem 2 is that the games built according to Def. 5 are suitable for solving the synthesis task. Note that there are plays in the game that are winning for the system player, but do not correspond to words that are models of the specification. The reason is that the environment player is not forced to iterate infinitely often over every possible pointer value for the final states of the guarantee automaton. Thus, a winning strategy for the system player in this game does not correspond one-to-one to a Mealy machine that satisfies the specification. For obtaining an implementation for the specification, we need to apply some post-processing to a system player's winning strategy in the parity game.

The post-processing step is however not difficult: observe that the worst case for the system player is that the environment player cycles through the guarantee pointer values. This way, the system player can only win if the decision sequence in the game represents a model of the guarantees, or the system player is able to eventually point out a rejecting state of the assumption automaton that is never left again. In both cases, the specification is met. Thus, if we attach a round-robin counter for the assumption pointer to a system player's strategy, we obtain a valid result for our synthesis problem.

5 Solving Parity Games Symbolically

For an efficient implementation of the synthesis approach in this paper, the ability to perform the *symbolic* solution of the parity game built according to the construction of the previous section is imperative.

For the scope of this paper, we use a simple parity game solving algorithm that is based on a fixed-point characterization of the winning set of positions in the game, i.e., the positions from which, if the game is started there, the system player can win the game. This approach has three advantages over the classical parity game solving algorithms by Jurdzinski [22] or McNaughton [26]. First of all, it is simpler. Second, it allows applying a nested fixed-point computation acceleration method by Browne et al. [5] that essentially reduces the solution complexity to quadratic time (in the number of game positions), which speeds up the game solving process in contrast to McNaughton's algorithm. Finally, the three-color parity game acceleration method for Jurszinski's algorithm by de Alfaro and Faella [8] is in some sense included for free. Their technique searches for gaps in counter values for visits to positions with color 1. These counters are an artifact that is introduced by Jurzinski's algorithm. The gaps witness the case that the game solving process can be terminated before the convergence of the counter values. As we do not need such counters here, our algorithm can terminate early automatically without the need to search for such gaps. At the same time, we still have a quadratic complexity of the game solving process. This advantage would also generalize to more than three colors, which the acceleration method in [8] does not.

For the special case of the games in this paper (with only player 0 having colors other than 0 and having only three colors in total), a characterization of the winning positions in a parity game by Emerson and Jutla [14] reduces to the following fixed-point equation:

$$W_0 = \nu X_2.\mu X_1.\nu X_0.(V_1 \cap \Diamond X_0) \cup (V_0 \cap C_0 \cap \Box X_0) \cup (V_0 \cap C_1 \cap \Box X_1) \cup (V_0 \cap C_2 \cap \Box X_2)$$

In this formula, C_i represents the set of positions with color i (for every $0 \leq i < 2$), and $\Box Y$ and $\Diamond Y$ describe, for every $Y \subseteq V$, the set of positions of player 0/player 1 from which player 1 can ensure that after the next move, a position in Y is reached, respectively. All of the operations needed to evaluate this formula can be performed symbolically [6]. Also, encoding the state space of the game into BDDs is not difficult: we can simply assign one bit to every state in the assumption and guarantee automata, one bit for every input or output atomic proposition, two bits for the "recently visited" flags in the game, and $\lceil \log_2 m \rceil + \lceil \log_2 n \rceil$ bits for the pointers.

It remains to be discussed how a winning strategy can be computed symbolically after the sets of winning positions for the two players have been identified. First of all, for $\psi = (V_1 \cap \Diamond X_0) \cup (V_0 \cap C_0 \cap \Box X_0) \cup (V_0 \cap C_1 \cap \Box X_1) \cup (V_0 \cap C_2 \cap \Box X_2)$, we compute a sequence of prefixed points $Y_i = \nu X_2.\mu^i X_1.\nu X_0.\psi$ for $i \in \mathbb{N}$. Then, we take the transition function E_1 of the game and restrict it such that only actions that result in ensuring that the successor position is in the set Y_i for the lowest possible value of i are taken. Any positional strategy that adheres to the restricted transition function is guaranteed to be winning for the system player.

6 Experimental Results

To evaluate the new synthesis approach presented in this paper, it has been implemented in a prototype tool-chain, written in C++ and Python. For the symbolic computation steps, the BDD library CUDD v.2.4.2 [29] was employed. The first step in the tool-chain is to apply the LTL-to-Büchi converter `ltl2ba v.1.1` [18] to the negation of all assumptions and guarantees of the specification. If the result happens to be very-weak, we already have a UVW for the specification part. All remaining assumptions and guarantees are first converted to deterministic Rabin automata using `ltl2dstar v.0.5.1` [23], then translated to equivalent deterministic Büchi automata (if possible), and finally, after a quick check with the construction by Bojańczyk [4] that they represent languages in the common fragment of ACTL and LTL, translated to sets of vermicelli using the construction from Sect. 3. Whenever one of these translations is found to be not possible for some assumption or guarantee, the specification is known not to lie in the supported specification fragment and rejected. The construction from Sect. 3 is performed symbolically, using BDDs and dynamic variable reordering for the BDD variables. The UVW for the individual assumptions and guarantees are then merged and some simulation-based automaton minimization steps are applied. In contrast to general bisimulation-based minimization techniques for non-deterministic Büchi automata (see, e.g., [15]), we make use of the fact that the automata are very-weak, which allows applying more optimizations. The optimization steps are:

- States that are reached by the same set of prefix words are merged (unless this would introduce a loop).
- States with the same language are merged.
- For every pair of states (q, q') in the automaton, if q is reached by at least as many prefix words as q', but q' has a greater language than q, we remove q'.

Finally, we perform symbolic parity game solving for the synthesis game build using the minimized UVW for the assumptions and guarantees as described in Sect. 4 and Sect. 5. In case of realizability, we use an algorithm by Kukula and Shiple [24] to compute a circuit description of the implementation. The prototype tool also checks for which input/output bits it makes sense to encode the last values into the game as an additional component. This can happen if there are many states in the UVW for which it only depends on the last input and output whether we are in that state at a certain time. Then, we can save the BDD bits for these states. For checking the resulting implementations for correctness, we use `NuSMV v.2.5.4` [7].

All computation times given in the following were obtained on an Intel Core 2 Duo (1.86 Ghz) computer running Linux. All tools considered are single-threaded. We restricted the memory usage to 2 GB and set a timeout of 3600 seconds. We compare our new approach against `Acacia+ v.1.2` [16, 17] and `Unbeast v.0.6` [13], both using ltl2ba. Both synthesis tools implement semi-algorithms, i.e., we need to test for realizability and unrealizability separately and only give the computation time of the invocation that terminated for comparison. We could not compare against tools that implement generalized reactivity(1) synthesis such as Anzu [21] as due to the non-standard semantics (see [11], p. 4 for details) used there, the results would not be meaningful.

Benchmarks

First of all, we consider the load balancer from [10]. This benchmark is for synthesis tools that are capable of handling full LTL, and consists of 10 scalable specifications. Out of these, we found 6 to be contained in the supported fragment by our approach, including the final specification of the load balancer. Table 1 summarizes the results. It can be observed that the two synthesis tools for full LTL are clearly outperformed on the supported specifications.

As a second benchmark, we use the non-pre-synthesized AMBA high performance bus arbiter specification described in [2], which is again scalable in the number of clients. Here, our tool-chain is able to synthesize the two-client version in 151 seconds, while the three-client version takes 1422 seconds. In both cases, most of the time is spent on the symbolic game solving step. Neither Unbeast nor Acacia+ can handle any of these two cases within 1 hour of computation time. According to [3], with the pre-synthesized version of the specification of [2], the generalized reactivity(1) tool used in the experimental evaluation of [3] could only handle up to four clients. Thus, our approach comes close in terms of efficiency, but without the need of pre-synthesis. For completeness, it must be added, however, that a (manual) rewriting of the specification was later able boost the generalized reactivity(1) synthesis performance [3] on this benchmark.

7 Conclusion

In this paper, we have proposed ACTL $\cap$ LTL as a specification fragment that combines expressivity and efficiency for the synthesis of reactive systems. We gave novel algorithms and constructions for the individual steps in the synthesis workflow. In particular, we gave the first procedure to obtain universal very-weak automata from LTL formulas (if possible) and described a novel procedure for building a parity game from assumption and guarantee properties that speeds up the game solving process by letting the two players choose the next obligations to the respective other player in the game.

We did not fully exploit the favorable properties of UVW in the paper, and only see the experimental evaluation herein as a start. For example, since in the structure of the game built from UVWs, we keep track of in which assumption and guarantee states we could be in, the game lends itself to the symbolic encoding of the prefixed points in the game solving process using anti-chains [16].

Also, the approach can easily be extended to support properties whose *negation* is in the common fragment of ACTL and LTL. This would allow using *persistence* properties like “the system must eventually signal readiness forever”. We recently described in

Table 1. Running times of the synthesis tools Acacia (“A”), Unbeast (“U”) and a prototype tool for the approach presented in this paper (“B”) for the load balancer benchmark, using setting labels from [10]. For each combination of assumptions and guarantees, it is reported whether the specification was realizable (+/-) and how long the computation took (in seconds).

Tool	Setting / # Clients	2	3	4	5	6	7	8	9
B	1	+ 0.3	+ 0.4	+ 0.4	+ 0.4	+ 0.5	+ 0.5	+ 0.6	+ 0.6
U		+ 0.0	+ 0.0	+ 0.6	+ 0.0	+ 0.0	+ 0.0	+ 0.1	+ 0.2
A		+ 0.3	+ 0.3	+ 0.3	+ 0.3	+ 0.4	+ 0.4	+ 0.4	+ 0.5
B	$1 \wedge 2$	+ 0.4	+ 0.4	+ 0.4	+ 0.5	+ 0.6	+ 0.9	+ 2.2	+ 6.9
U		+ 0.7	+ 0.0	+ 0.1	+ 0.1	+ 0.1	+ 0.1	+ 0.2	+ 0.3
A		+ 0.3	+ 0.4	+ 1.2	+ 0.3	+ 0.4	+ 0.7	+ 1.8	+ 5.5
B	$1 \wedge 2 \wedge 3$	- 0.5	- 0.6	- 0.7	- 0.9	- 1.2	- 1.7	- 3.4	- 7.6
U		- 0.0	- 0.0	- 0.1	- 0.1	- 0.2	- 1.3	- 11.5	- 145.4
A		- 0.3	- 0.3	- 0.4	- 2.9	timeout	timeout	timeout	timeout
B	$6 \wedge 7 \rightarrow 1 \wedge 2 \wedge 5 \wedge 8$	+ 0.6	+ 0.8	+ 0.9	+ 1.2	+ 1.6	+ 2.2	+ 4.0	+ 9.7
U		+ 0.1	+ 0.4	+ 1.4	+ 39.9	timeout	timeout	timeout	timeout
A		+ 2.1	+ 1.3	timeout	timeout	timeout	timeout	timeout	timeout
B	$6 \wedge 7 \rightarrow 1 \wedge 2 \wedge 5 \wedge 8 \wedge 9$	- 0.7	- 0.9	- 1.2	- 1.6	- 2.1	- 3.2	- 5.5	- 11.5
U		- 0.0	- 0.1	- 0.2	- 1.4	- 28.5	- 886.4	timeout	timeout
A		- 0.4	- 0.4	- 2.6	timeout	timeout	timeout	timeout	timeout
B	$6 \wedge 7 \wedge 10 \rightarrow 1 \wedge 2 \wedge 5 \wedge 8 \wedge 9$	+ 0.8	+ 1.0	+ 1.3	+ 2.3	+ 2.5	+ 3.3	+ 5.7	+ 11.8
U		+ 0.3	+ 2.2	+ 23.7	+ 632.5	timeout	timeout	timeout	timeout
A		+ 0.9	+ 0.8	+ 16.3	timeout	timeout	timeout	timeout	timeout

[12] how generalized reactivity(1) synthesis can be extended to handle such properties, resulting in five-color parity games. The constructions in this paper are easy to extend accordingly.

Acknowledgements

The author wants to thank Bernd Finkbeiner and Sven Schewe for the interesting discussions in which the idea of Sect. 4 was developed. Furthermore, the author wants to thank Armin Biere for pointing out the common fragment of ACTL and LTL as potentially interesting for synthesis.

References

1. Bloem, R., Chatterjee, K., Greimel, K., Henzinger, T.A., Jobstmann, B.: Robustness in the presence of liveness. In Touili, T., Cook, B., Jackson, P., eds.: CAV. Volume 6174 of LNCS., Springer (2010) 410–424
2. Bloem, R., Galler, S.J., Jobstmann, B., Piterman, N., Pnueli, A., Weiglhofer, M.: Interactive presentation: Automatic hardware synthesis from specifications: a case study. In Lauwereins, R., Madsen, J., eds.: DATE, ACM (2007) 1188–1193
3. Bloem, R., Galler, S.J., Jobstmann, B., Piterman, N., Pnueli, A., Weiglhofer, M.: Specify, compile, run: Hardware from PSL. Electr. Notes Theor. Comput. Sci. **190**(4) (2007) 3–16
4. Bojańczyk, M.: The common fragment of ACTL and LTL. In Amadio, R.M., ed.: FoSSaCS. Volume 4962 of LNCS., Springer (2008) 172–185
5. Browne, A., Clarke, E.M., Jha, S., Long, D.E., Marrero, W.R.: An improved algorithm for the evaluation of fixpoint expressions. Theor. Comput. Sci. **178**(1-2) (1997) 237–255

6. Burch, J.R., Clarke, E.M., McMillan, K.L., Dill, D.L., Hwang, L.J.: Symbolic model checking: 10^{20} states and beyond. Inf. Comput. **98**(2) (1992) 142–170
7. Cimatti, A., Clarke, E.M., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: NuSMV 2: An opensource tool for symbolic model checking. In Brinksma, E., Larsen, K.G., eds.: CAV. Volume 2404 of LNCS., Springer (2002) 359–364
8. de Alfaro, L., Faella, M.: An accelerated algorithm for 3-color parity games with an application to timed games. In Damm, W., Hermanns, H., eds.: CAV. Volume 4590 of LNCS., Springer (2007) 108–120
9. Ehlers, R.: Minimising deterministic Büchi automata precisely using SAT solving. In Strichman, O., Szeider, S., eds.: SAT. Volume 6175 of LNCS., Springer (2010) 326–332
10. Ehlers, R.: Symbolic bounded synthesis. In Touili, T., Cook, B., Jackson, P., eds.: CAV. Volume 6174 of LNCS., Springer (2010) 365–379
11. Ehlers, R.: Experimental aspects of synthesis. In Reich, J., Finkbeiner, B., eds.: iWIGP. Volume 50 of EPTCS. (2011) 1–16
12. Ehlers, R.: Generalized Rabin(1) synthesis with applications to robust system synthesis. In Bobaru, M.G., Havelund, K., Holzmann, G.J., Joshi, R., eds.: NASA Formal Methods. Volume 6617 of LNCS., Springer (2011) 101–115
13. Ehlers, R.: Unbeast: Symbolic bounded synthesis. In Abdulla, P.A., Leino, K.R.M., eds.: TACAS. Volume 6605 of LNCS., Springer (2011) 272–275
14. Emerson, E.A., Jutla, C.S.: Tree automata, mu-calculus and determinacy (extended abstract). In: FOCS, IEEE Computer Society (1991) 368–377
15. Etessami, K., Wilke, T., Schuller, R.A.: Fair simulation relations, parity games, and state space reduction for Büchi automata. In Orejas, F., Spirakis, P.G., van Leeuwen, J., eds.: ICALP. Volume 2076 of LNCS., Springer (2001) 694–707
16. Filiot, E., Jin, N., Raskin, J.F.: An antichain algorithm for LTL realizability. In Bouajjani, A., Maler, O., eds.: CAV. Volume 5643 of LNCS., Springer (2009) 263–277
17. Filiot, E., Jin, N., Raskin, J.F.: Compositional algorithms for LTL synthesis. In Bouajjani, A., Chin, W.N., eds.: ATVA. Volume 6252 of LNCS., Springer (2010) 112–127
18. Gastin, P., Oddoux, D.: Fast LTL to Büchi automata translation. In Berry, G., Comon, H., Finkel, A., eds.: CAV. Volume 2102 of LNCS., Springer (2001) 53–65
19. Hashiguchi, K.: Representation theorems on regular languages. J. Comput. Syst. Sci. **27**(1) (1983) 101–115
20. Janin, D., Lenzi, G.: On the relationship between monadic and weak monadic second order logic on arbitrary trees, with applications to the mu-calculus. Fundam. Inform. **61**(3-4) (2004) 247–265
21. Jobstmann, B., Galler, S.J., Weiglhofer, M., Bloem, R.: Anzu: A tool for property synthesis. In Damm, W., Hermanns, H., eds.: CAV. Volume 4590 of LNCS., Springer (2007) 258–262
22. Jurdzinski, M.: Small progress measures for solving parity games. In Reichel, H., Tison, S., eds.: STACS. Volume 1770 of LNCS., Springer (2000) 290–301
23. Klein, J., Baier, C.: Experiments with deterministic ω-automata for formulas of linear temporal logic. Theor. Comput. Sci. **363**(2) (2006) 182–195
24. Kukula, J.H., Shiple, T.R.: Building circuits from relations. In Emerson, E.A., Sistla, A.P., eds.: CAV. Volume 1855 of LNCS., Springer (2000) 113–123
25. Maidl, M.: The common fragment of CTL and LTL. In: FOCS. (2000) 643–652
26. McNaughton, R.: Infinite games played on finite graphs. Ann. Pure Appl. Logic **65**(2) (1993) 149–184
27. Piterman, N., Pnueli, A., Sa'ar, Y.: Synthesis of reactive(1) designs. In Emerson, E.A., Namjoshi, K.S., eds.: VMCAI. Volume 3855 of LNCS., Springer (2006) 364–380
28. Sohail, S., Somenzi, F.: Safety first: A two-stage algorithm for LTL games. In: FMCAD, IEEE (2009) 77–84
29. Somenzi, F.: CUDD: CU decision diagram package, release 2.4.2 (2009)

A Appendix

A.1 Notes to Lemma 1

Let a f be an end component decomposition. In the proof of Lemma 1, it is said that the end component decomposition f' with $f'(X) = \bigcup_{X' \supseteq X} f(X')$ for every $X \subseteq \Sigma$ is then a maximal end-component decomposition for the same language. For the sake of completeness, let us derive here that (1) f' is an end component decomposition for the same language, and that (2) f' is maximal.

1. Consider an end-component decomposition $\hat{f}$ that we obtain from f by setting $\hat{f}(X_2) = f(X_1) \cup f(X_2)$ for some $X_1 \subseteq \Sigma$ and $X_2 \subseteq \Sigma$ with $X_2 \subseteq X_1$, and $\hat{f}(X') = f(X')$ for all $X' \neq X_2$. Thus, $\hat{f}$ is obtained from f by merging $f(X_1)$ into $f(X_2)$. As f' with $f'(X) = \bigcup_{X' \supseteq X} f(X')$ for every $X \subseteq \Sigma$ can be obtained from f by applying finitely many such merging operations, if we can prove that the languages that f and $\hat{f}$ are end-component decompositions of the same language, then we also know this for f and f'.
   As $X_2 \subseteq X_1$, we have that:
   $$\hat{f}(X_2) \cdot X_2^\omega = (f(X_1) \cup f(X_2)) \cdot X_2^\omega \subseteq f(X_1) \cdot X_1^\omega \cup f(X_2) \cdot X_2^\omega$$
   By taking the union with $f(X_1) \cdot X_1^\omega$ in every part of this equation (and removing the middle part), we obtain:
   $$f(X_1) \cdot X_1^\omega \cup \hat{f}(X_2) \cdot X_2^\omega \subseteq f(X_1) \cdot X_1^\omega \cup f(X_2) \cdot X_2^\omega$$
   As by the definition of $\hat{f}(X_2)$, we have that $\hat{f}(X_2) \cdot X_2^\omega \supseteq f(X_2) \cdot X_2^\omega$, it follows that $f(X_1) \cdot X_1^\omega \cup \hat{f}(X_2) \cdot X_2^\omega = f(X_1) \cdot X_1^\omega \cup f(X_2) \cdot X_2^\omega$. We can now deduce that f and $\hat{f}$ are end-component decompositions of the same language:
   $$\begin{aligned} &\Sigma^\omega \setminus \bigcup_{X \subseteq \Sigma} (f(X) \cdot X^\omega) \\ &= \Sigma^\omega \setminus \left(\left(\bigcup_{X \subseteq \Sigma, X \neq X_1, X \neq X_2} (f(X) \cdot X^\omega) \right) \cup (f(X_1) \cdot X_1^\omega) \cup (f(X_2) \cdot X_2^\omega) \right) \\ &= \Sigma^\omega \setminus \left(\left(\bigcup_{X \subseteq \Sigma, X \neq X_1, X \neq X_2} (f(X) \cdot X^\omega) \right) \cup (f(X_1) \cdot X_1^\omega) \cup (\hat{f}(X_2) \cdot X_2^\omega) \right) \\ &= \Sigma^\omega \setminus \bigcup_{X \subseteq \Sigma} (\hat{f}(X) \cdot X^\omega) \end{aligned}$$

2. Let us now prove that f' is a maximal decomposition. Let $w \in \Sigma^*$ and $T = \{X_1, \ldots, X_n\}$ be the subsets of Σ such that for all $1 \leq i \leq n$, $w \cdot X_i^\omega \cap L = \emptyset$, for L being the language such that f' is an end-component decomposition of it. Observe the following facts:

- First of all, T is a downwards-closed set, i.e., for all $X \in T$ and $X' \subseteq X$, we have $X' \in T$. This follows directly from the fact that $w \cdot X^\omega \supseteq w \cdot X'^\omega$ for all $X' \subseteq X$.
- Furthermore, for all maximal elements $Y \in T$ (i.e. there exists no $X \supset Y$ with $X \in T$), we have that $w \in f(Y)$. This follows from the fact that for $Y = \{x_1, \ldots x_m\}$, we have $w(x_1 x_2 \ldots x_m)^\omega \notin L$ by definition, and thus for some $X \supseteq Y$, we must have $w \in f(X)$ in order for the end component decomposition not to miss that $w(x_1 x_2 \ldots, x_m)^\omega \notin L$. However, as for all $X \supset Y$, by the definition of T, we also have $w \cdot X_i^\omega \cap L \neq \emptyset$, having $w \in f(X)$ would lead to $L \neq \Sigma^\omega \setminus \bigcup_{X' \subseteq \Sigma} (f(X') \cdot X'^\omega)$. Thus, the only way for f to be a valid end-component decomposition for L is to have $w \in f(Y)$.

As we have established that T is a downward-closed set and for all maximal elements Y, we have $w \in f(Y)$, we know that for all $X \subseteq \Sigma$, we have $X \in T$ (and thus $w \cdot X_i^\omega \cap L = \emptyset$) if and only if $w \in \bigcup_{X' \supseteq X} f(X')$. Since this line of reasoning holds for all $w \in \Sigma^*$, it follows that $\{w \in \Sigma^* \mid w \cdot X^\omega \cap L = \emptyset\} = \bigcup_{X' \supseteq X} f(X')$, and thus, f' is a maximal end-component decomposition.

A.2 On the relationships of UVW and deterministic Büchi automata

Proposition 4. *Every UVW is representable as a deterministic Büchi automaton.*

Proof. Let $\mathcal{A} = (Q, \Sigma, \delta, Q_0, F)$ be a UVW. Without loss of generality, we assume that $F = \{1, \ldots, m\}$ for some $m \in \mathbb{N}$. We can construct an equivalent deterministic Büchi automaton $\mathcal{A}' = (Q', \Sigma, \delta', Q'_0, F')$ as follows:

$$\begin{aligned}
Q' &= 2^Q \times \{0, \ldots, m\} \\
\forall (S, 0) \in Q', x \in \Sigma : \delta'((S,0), x) &= (\hat{\delta}(S, \{x\}), 1) \\
\forall (S, j) \in Q', x \in \Sigma \text{ with } j \geq 1 \text{ and } & \\
j \notin \delta(j, x) \vee j \notin S : \delta'((S,j), x) &= (\hat{\delta}(S, \{x\}), (j+1) \bmod m) \\
\forall (S, j) \in Q', x \in \Sigma \text{ with } j \geq 1 \text{ and } & \\
j \in \delta(j, x) \wedge j \in S : \delta'((S,j), x) &= (\hat{\delta}(S, \{x\}), j) \\
Q'_0 &= \{(Q_0, 0)\} \\
F &= \{(S, 0) \mid S \subseteq Q\}
\end{aligned}$$

Assume that for a given word $w \in \Sigma^\omega$, there exists some rejecting state q in $\mathcal{A}$ that is eventually entered along some run and never left from that point onwards. The construction ensures that then, the counter value cannot be increased from q to $q+1$ (or to 0 if $q = m$) in the corresponding run of $\mathcal{A}'$ anymore. As a consequence, $\mathcal{A}'$ also rejects the word as the counter value 0 cannot occur along the run more than once from that time onwards.

On the other hand, if a word is accepted by the UVW, then every state in the UVW along any of its runs is left (see the Definition of "left" on page 11) infinitely often. Since the counter values increase whenever this happens (note that a state is always left for all runs of the UVW at the same time), we get a value of 0 infinitely often, leading to visiting accepting states in the deterministic Büchi automaton infinitely often. □

A.3 Proof of Theorem 1

Proof. For the first claim, note that during reading a finite input word, the fourth component keeps track of in which states a run in $\mathcal{B}$ could be in. Only if there is no accepting state in which $\mathcal{B}$ could be in, $\mathcal{D}$ can accept a word. At the same time, for any word in the language of $\mathcal{A}$, we can always take transitions that reset the Boolean flag to **false**, and set the third component to the set that that contains only the current alphabet symbol. This way, the first component only contains a singleton state in every instance of time and represents a path to the accepting state in $\mathcal{A}$.

For the second claim, given a word that is not accepted by $\mathcal{B}$ and a vermicelli that contains the word, the construction of $\mathcal{D}$ ensures that we can build an accepting run in which the first component of the states represents a sequence that contains an accepting run of $\mathcal{A}$ on the word. During the run, the second and third components are used to represent the vermicelli, and updated whenever for the word chosen, we move to the next vermicelli element. The Boolean flag represents whether the current factor in the vermicelli has a * or not, and the second component represents the subset of the alphabet in the vermicelli element. Since a transition has a cost of 1 precisely at the points at which we change these elements, the run has a cost of k. Along the way, the first component is updated to represent all of the states in which $\mathcal{A}$ can be after having read any of the words in the vermicelli so far. Since by assumption, we have a vermicelli whose models are contained in the language of $\mathcal{A}$, all these runs need to be in a final state at the end of the run. At the same time, since the word is not accepted by $\mathcal{B}$ and the last component in the states track all possible runs in $\mathcal{B}$, the state set in the fourth component does not intersect with F at the end. Thus, the run outlined here is accepting. □

A.4 Proof of Theorem 2

Proof. The construction of the game ensures that along any play $\pi = \pi^0_0 \pi^1_0 \pi^1_0 \ldots$ that corresponds to some decision sequence $\rho = \rho^0_0 \rho^1_0 \rho^0_1 \ldots$, the first component in every position in $\pi^0 = \pi^0_0 \pi^0_1 \pi^0_2 \ldots$ (the sequence of positions of player 0 along the run) always represents in which states the automaton $\mathcal{A}^a$ could be in after having read a prefix of $\tilde{\rho} = (\rho^0_0, \rho^1_0)(\rho^0_1, \rho^1_1), \ldots$. The same applies for the second component and $\mathcal{A}^g$. If we have a winning strategy f for player 1 given, then it ensures that the highest color that can occur along a play infinitely often is even. There are only two ways in which this can happen: (1) if either eventually the assumption pointer is set to and left with a value that refers to a state in the assumption UVW that is never left from that point onwards, or (2) if every guarantee UVW state that the guarantee pointer is set to is eventually left. In the latter case, since the input player rotates through all of the pointer values, this means that they are all left infinitely often and thus the UVW $\mathcal{A}^g$ accepts the word.

For the second claim in the theorem, note that the construction ensures that if the system player always cycles through the assumptions, then a position in the game with color 1 can only be visited infinitely often if $\mathcal{A}^a$ accepts the decision sequence. Thus, the run is winning for the system player if the assumptions are not fulfilled. On the other hand, if the guarantees are fulfilled along the run, then every rejecting state in $\mathcal{A}^g$ is left infinitely often along the run, and thus color 2 is visited infinitely often regardless of how the system player chooses the guarantee pointer values.