Optimizations for LTL Synthesis

Barbara Jobstmann and Roderick Bloem
Graz University of Technology

Abstract— **We present an approach to automatic synthesis of specifications given in Linear Time Logic. The approach is based on a translation through universal co-Büchi tree automata and alternating weak tree automata [1]. By careful optimization of all intermediate automata, we achieve a major improvement in performance.**

We present several optimization techniques for alternating tree automata, including a game-based approximation to language emptiness and a simulation-based optimization. Furthermore, we use an incremental algorithm to compute the emptiness of nondeterministic Büchi tree automata. All our optimizations are computed in time polynomial in the size of the automaton on which they are computed.

We have applied our implementation to several examples and show a significant improvement over the straightforward implementation. Although our examples are still small, this work constitutes the first implementation of a synthesis algorithm for full LTL. We believe that the optimizations discussed here form an important step towards making LTL synthesis practical.

I. Introduction

Writing both a specification and an implementation and subsequently checking whether the latter satisfies the former seems wasteful. A much more attractive approach would be to automatically construct the implementation from the specification, leaving the designer with only the task of ensuring that the specification describes the intended behavior. The benefit is even more pronounced when one takes into effect the cost for debugging the manual implementation, and of redesigning it when the specification changes.

LTL synthesis was proposed in [2]. The key to the solution is the observation that a program with input signals I and output signals O can be seen as a complete Σ-labeled D-tree with $\Sigma = 2^O$ and $D = 2^I$: the label of node $t \in D^*$ gives the output after input sequence t. The solution proposed in [2] is to build a nondeterministic Büchi word automaton for the specification and then to convert this automaton to a deterministic Rabin automaton that recognizes all complete trees satisfying the specification. A witness to the nonemptiness of the automaton is an implementation of the specification. A specification is called *realizable* if such an implementation exists.

There are two reasons that this approach has not been followed by an implementation. The first reason is that synthesis of LTL properties is 2EXPTIME-complete [3]. The second is that the solution uses an intricate determinization construction [4] that is hard to implement and very hard to optimize. The first reason should not prevent one from implementing the approach. After all, the bound is a lower bound and a manual implementation is also subject to it. (Cf. [5].) Thus, the worst case complexity of verifying the specification on a manual implementation is also 2EXPTIME in terms of the (full) specification. In combination with the second reason, however, the argument gains strength. For many specifications, a doubly-exponential blow up is not necessary, but can only be avoided through careful use of optimization techniques, which is hard to achieve in combination with Safra's algorithm.

This work was supported in part by the European Commission under contract 507219 (Prosyd).

Kupferman and Vardi [1] recently proposed an alternative to this approach. Starting from a specification φ over they generate, through the nondeterministic Büchi word automaton for $\neg\varphi$, a universal co-Büchi tree automaton that accepts all trees satisfying φ. From this automaton they construct an alternating weak tree automaton accepting at least one (regular) tree satisfying φ (or none, if φ is not realizable). Finally, the alternating automaton is converted to a nondeterministic Büchi tree automaton with the same language. A witness for the nonemptiness of this automaton is an implementation of φ. The approach is applicable to any linear logic that is closed under negation and can be compiled to nondeterministic Büchi word automaton.

This approach allows for optimizations at all steps. First of all, to generate the nondeterministic Büchi word automaton, we use the optimizations present in Wring [6]. The conversion to the universal co-Büchi tree automaton is relatively simple. If we consider the universal co-Büchi tree automaton as a game between the environment (which drives I) and the system (which drives O), states that are winning for the environment represent unrealizable specifications and can be removed. On the weak alternating tree automaton that is created next, we can perform the same optimization. Furthermore, we extend the concept of simulation to alternating tree automata and use it to optimize the automaton. Next,we compute the states of the nondeterministic automaton in a breadth-first manner and compute the game at every step. Thus, we may avoid expanding many of the states of the nondeterministic automaton. Finally, we use a simulation-based optimization to minimize the size of the resulting finite state machine. As suggested in [1], we perform the construction of the weak alternating automaton and the nondeterministic Büchi automaton incrementally. We build an increasingly large part of the weak alternating automaton and reuse results obtained using the smaller automata for the larger ones.

Our tool, Lily (Linear Logic Synthesizer), takes as input an LTL formula and a partition of the atomic propositions into input and output signals. If the specification is realizable, it delivers a Verilog file as output. We present an experimental evaluation of our implementation in Section V. Previous work on LTL synthesis focuses on restricted subsets of LTL [7], [8], [9]. Our implementation is the first to handle the complete language. It does not impose any syntactic requirements on

the specification.

The flow of the paper is as follows. In the next section, we will introduce the necessary concepts. In Section III we describe a game-based and a simulation-based optimization that can be used on any tree automaton. In Section IV, we recall the construction of Kupferman and Vardi [1] and discuss how it can be implemented efficiently. We show experimental results and conclude in Section V.

II. Preliminaries

We assume that the reader is familiar with the μ-calculus and linear time temporal logic (LTL). (See [10].) We will use LTL to specify the behavior of a system. Properties will use the set $I \cup O$ of atomic propositions, where I and O are disjoint sets denoting the input and output signals, respectively.

A Σ-labeled D-*tree* is a tuple (T, τ) such that $T \subseteq D^*$ is prefix-closed and $\tau : T \to \Sigma$. The tree is *complete* if $T = D^*$. The set of all Σ-labeled D-trees is denoted by $T_{\Sigma,D}$.

We will use Σ-labeled D-trees to model programs with input alphabet D and output alphabet Σ. In order to establish a link with the specification, we will assume that $D = 2^I$ and $\Sigma = 2^O$. Thus, a path of a Σ-labeled D-tree can be seen as a word over $(\Sigma \cup D)^\omega$: we merge the label of the node with the direction edge following it in the path. Given a word language $L \in (\Sigma \cup D)^\omega$, let $\Lambda(L) \subseteq T_{\Sigma,D}$ be the set of trees T such that all paths of T are in L. For a word automaton A we will write $\Lambda(A)$ for $\Lambda(L(A))$. Similarly, we will write $\Lambda(\varphi)$ for the set of trees T such that every path of T satisfies the LTL formula φ.

A *Moore machine* with output alphabet Σ and input alphabet D is a tuple $M = (\Sigma, D, S, s_0, f, g)$ such that S is a finite set of states, $s_0 \in S$ is the initial state, $f : S \times D \to S$ is the transition function, and $g : S \to \Sigma$ is the output function. We extend f to the domain $S \times \Sigma^*$ in the usual way. The *input/output language* of M is $L(M) = \{\pi \in (\Sigma \cup D)^\omega \mid \pi = (\sigma_0 \cup d_0, \sigma_1 \cup d_1, \dots), \sigma_i = g(f(q_0, d_0 ... d_{i-1}))\}$.

Every Moore machine corresponds to a complete Σ-labeled D-tree for which every node $t \in D^*$ is labeled with $g(f(q_0, t))$. Thus, every tree language $L \subseteq T_{\Sigma,D}$ defines a set $\mathcal{M}(L)$ of Moore machines: those machines M for which $\Lambda(L(M)) \in L$. (Note that not every tree can be defined by a Moore machine and thus there are tree languages L for which $\bigcup\{\Lambda(L(M)) \mid M \in \mathcal{M}(L)\} \neq L$.)

An *alternating tree automaton* for Σ-labeled D-trees is a tuple $A = (\Sigma, D, Q, q_0, \delta, \alpha)$ such that Q is a finite set of *states*, $q_0 \in Q$ is the *initial state*, $\delta : Q \times \Sigma \to 2^{2^{D \times Q}}$ is the *transition relation* (an element $C \in 2^{D \times Q}$ is a *transition*) and $\alpha \subseteq Q$ is the *acceptance condition*. We denote by A^q, for $q \in Q$, the automaton A with the initial state q.

A run (R, ρ) of A on a Σ-labeled D-tree (T, τ) is a $T \times Q$-labeled $\mathbb{N}$-tree satisfying the following constraints:

1) $\rho(\varepsilon) = (\varepsilon, q_0)$.
2) If $r \in R$ is labeled (t, q), then there is a set $\{(d_1, q_1), \dots, (d_k, q_k)\} \in \delta(q, \tau(t))$ such that r has k children labeled $(t \cdot d_1, q_1), \dots, (t \cdot d_k, q_k)$.

Intuitively, the nodes of the run on a tree T are labeled with pairs (t, q) meaning that A is in state q in node t of T. Because A is alternating, for a given t there can be multiple q_i and nodes labeled (t, q_i) in R. The automaton starts at the root node in state q_0. If it is in state q in node t of T, and t is labeled σ, then $\delta(q, \sigma)$ tells A what to do next. The automaton can nondeterministically choose a transition $C \in \delta(q, \sigma)$. Then, for all $(d', q') \in C$, A moves to node $t \cdot d'$ in state q'. (The transition relation $\delta(q, \sigma)$ can be considered as a DNF formula over $D \times Q$.) Note that there are no runs with a node (t, q) for which $\delta(q, \tau(t)) = \emptyset$. On the other hand, a run that visits a node t needs not visit all of its children; there are no restrictions on the subtrees rooted in a node that is not visited. In particular, a node (t, q) such that $\delta(q, \tau(t)) = \{\emptyset\}$ does not have any children and there are no restrictions on the subtree rooted in t.

We have two acceptance conditions: Büchi and co-Büchi. A run (R, ρ) of a Büchi (co-Büchi) automaton is accepting if all *infinite* paths of (R, ρ) have infinitely many states in α (only finitely many states in α, resp.). The language $L(A)$ of A is the set of trees for which there exists an accepting run.

An alternating tree automaton induces a graph. The states of the automaton are the nodes of the graph and there is an edge from q to q' if (d', q') occurs in $\delta(q, \sigma)$ for some $\sigma \in \Sigma$ and $d' \in D$. A Büchi automaton is *weak* if each strongly connected component (SCC) contains either only states in α or only states not in α.

An automaton is *universal* if $|\delta(q, \sigma)| = 1$. A universal automaton has at most one run for a given input. An automaton is *nondeterministic* if for all $q \in Q, \sigma \in \Sigma, C \in \delta(q, \sigma)$ and $(d_i, q_i), (d_j, q_j) \in C$ we have $d_i = d_j$ implies $q_i = q_j$. That is, the automaton can only send one copy in each direction and every run is isomorphic to the input tree. An automaton is *deterministic* if it is both universal and nondeterministic.

An automaton is a *word automaton* if $|D| = 1$. In that case, we can leave out D altogether.

We will abbreviate alternating/nondeterministic/universal/deterministic Büchi/co-Büchi/weak tree/word automaton as a three letter acronym: A/N/U/D B/C/W T/W.

III. Simplifying tree automata

In this section we discuss two optimizations that can be used for any tree automaton.

A. Simplification Using Games

We define a sufficient (but not necessary) condition for language emptiness of A^q. Our heuristic views the alternating automaton as a game which is played in rounds. In each round, starting at a state q, the protagonist decides the label $\sigma \in \Sigma$ and a set $C \subseteq \delta(q, \sigma)$ and the antagonist chooses a pair $(d, q') \in C$. The next round starts in q'. If $\delta(q, \sigma)$ or C are empty the play is finite and the player who has to choose from an empty set loses the game. If a play is infinite the winner is determined by the acceptance condition. For an ABT (ACT), the protagonist wins the play if the play visits the set of accepting states α infinitely often (only finitely often, resp.). A *strategy* s maps a finite sequence of states $q_0, \dots, q_k$ to a set $C \subseteq \delta(q_k, \sigma)$ for some a label $\sigma \in \Sigma$. A play $q_1, q_2, \dots$ adheres to a strategy s if for every k, $s(q_0, \dots, q_k) = C$

implies that there is a pair $(d, q_{k+1}) \in C$. The game A^q is *won* (and q is *winning*) if there is a strategy such that all plays starting at q that adhere to the strategy are won. The set of all winning states is the *winning region*.

If the game is lost, $L(A^q)$ is empty. In the case of an NBT or NCT the converse holds as well, but in the alternating case it does not: A counterexample is a word automaton such that (1) $\delta(q_0, \sigma) = q_1 \wedge q_2$ for all σ, (2) $L(A^{q_1}) \cap L(A^{q_2}) = \emptyset$, and (3) the games A^{q_1} and A^{q_2} are won. (Computing a necessary and sufficient condition in polynomial time is not possible as it would give us an EXPTIME algorithm for deciding realizability.)

The game is computed as follows. For $S \subseteq Q$, let

$$\langle \mathsf{P} \rangle \mathsf{X}(S) = \{q \mid \exists \sigma \in \Sigma, C \in \delta(q, \sigma) \forall (d, q') \in C : q' \in S\},$$
$$W_B(S) = \nu Y. \langle \mathsf{P} \rangle \mathsf{X}(\mu Z. Y \wedge (S \vee \langle \mathsf{P} \rangle \mathsf{X}\ Z)), \text{ and}$$
$$W_C(S) = \mu Y. \langle \mathsf{P} \rangle \mathsf{X}(\nu Z. Y \vee (S \wedge \langle \mathsf{P} \rangle \mathsf{X}\ Z)).$$

In an ABT (ACT) with acceptance condition α, we can discard the states outside of $W_B(\alpha)$ ($W_C(\alpha)$, resp.).

Theorem 1: Given an ABT (ACT) $A = (\Sigma, D, Q, q_0, \delta, \alpha)$, let $W = W_B(\alpha)$. ($W = W_C(\alpha)$, resp.) If $q_0 \in W$, let the ABT (ACT) $A' = (\Sigma, D, Q', q_0, \delta', \alpha')$ with $Q' = Q \cap W$, $\alpha' = \alpha \cap W$, and $\delta'(q, \sigma) = \{C \in \delta(q, \sigma) \mid \forall (d, q') \in C, q \in W\}$. If $q_0 \notin W$, let A' consist of a single non-accepting state.

We have $L(A^q) = L(A'^q)$ for all $q \in Q'$ and thus $L(A) = L(A')$. □

B. Simplification Using Simulation Relations

The second optimization uses (direct) simulation minimization on alternating tree automata. Our construction generalizes that for alternating word automata [11], [12], [13].

Let $A = (\Sigma, D, Q, q_0, \delta, \alpha)$ be an ABT. The *direct simulation relation* $\preceq\ \subseteq Q \times Q$ is the largest relation such that $u \preceq v$ implies that (1) $u \in \alpha \rightarrow v \in \alpha$ and (2) $\forall \sigma \in \Sigma, C_u \in \delta(u, \sigma)\ \exists C_v \in \delta(v, \sigma)\ \forall d' \in D, (d', v') \in C_v\ \exists (d', u') \in C_u : u' \preceq v'$.

If $u \preceq v$, we say that u is *simulated by* v. If additionally, $u \succeq v$, we say that u and v are *simulation equivalent*, denoted $u \simeq v$.

Lemma 2: If $u \preceq v$ then $L(A^u) \subseteq L(A^v)$. □

The following theorems are tree-automaton variants of those presented in [13] for optimizing alternating word automata. The first theorem allows us to restrict the state space of an ABT to a set of representatives of every equivalence class under $\simeq$.

Theorem 3: Let $A = (\Sigma, D, Q, q_0, \delta, \alpha)$ be an ABT, let $u, v \in Q$, and suppose $u \simeq v$. Let $A' = (\Sigma, D, Q \setminus \{u\}, q_0', \delta', \alpha)$, where $q_0' = v$ if $q_0 = u$ and $q_0' = q_0$ otherwise, and δ' is obtained from δ by replacing u by v everywhere. Then, $L(A) = L(A')$. □

The following two theorems allow us to simplify the transition relation of an ABT. The first theorem tells us that we can drop states from a transition if they are not minimal with respect to the simulation relation and the second theorem tells us that we can drop entire transitions if there are other transitions that allow for a larger language.

Theorem 4: Let $A = (\Sigma, D, Q, q_0, \delta, \alpha)$ be an ABT. For $C \subseteq D \times Q$, let $C' = \{(d, u) \in C \mid \neg \exists v : v \neq u, v \preceq u, (d, v) \in C\}$. Let $A' = (\Sigma, D, Q, q_0, \delta', \alpha)$, where for all q and σ we have $\delta'(q, \sigma) = \{C' \mid C \in \delta(q, \sigma)\}$. We have $L(A) = L(A')$. □

Theorem 5: Let $A = (\Sigma, D, Q, q_0, \delta, \alpha)$ be an ABT. Suppose $C, C' \in \delta(q, \sigma)$, $C \neq C'$, and for all d and $(d, q') \in C'$ there is a $(d, q) \in C$ such that $q \preceq q'$. Let $A = (\Sigma, D, Q, q_0, \delta', \alpha)$ be an ABT for which δ' equals δ except that $\delta'(q, \sigma) = \delta(q, \sigma) \setminus C$. We have $L(A) = L(A')$. □

We can simplify an ABT by repeated application of the last two theorems and removal of states that are no longer reachable from the initial state. The simulation relation can be computed in polynomial time, as can the optimizations. (Application of the theorems does not alter the simulation relation.)

IV. Optimizations for Synthesis

A. Synthesis Algorithm

The goal of synthesis is to find a Moore machine M implementing an LTL specification φ (or to prove that no such M exists). Our approach follows that of [1], introducing optimizations that make synthesis much more efficient. The flow is as follows.

1) Construct an NBW A_{NBW} with $L(A_{\text{NBW}}) = \{w \in (\Sigma \cup D)^\omega \mid w \not\models \varphi\}$. Let n' be the number of states of A_{NBW}. Note that in the worst case n' is exponential in $|\varphi|$ [14].
2) Construct a UCT A_{UCT} with $L(A_{\text{UCT}}) = T_{\Sigma,D} \setminus \Lambda(A_{\text{NBW}}) = \Lambda(\varphi)$. Let n be the number of states of A_{UCT}; we have $n \leq n'$,
3) Perform the following steps for increasing k, starting with $k = 0$.
   a) Construct an AWT $A_{\text{AWT}k}$ such that $L(A_{\text{AWT}k}) \subseteq L(A_{\text{UCT}})$ and $A_{\text{AWT}k}$ has at most $n \cdot k$ states.
   b) Construct an NBT $A_{\text{NBT}k}$ such that $L(A_{\text{NBT}k}) = L(A_{\text{AWT}k})$; $A_{\text{NBT}k}$ has at most $(k+1)^{2n}$ states.
   c) Check for the nonemptiness of $L(A_{\text{NBT}k})$. If the language is nonempty, proceed to Step 4.
   d) If $k = 2n^{2n+2}$, stop: φ is not realizable. Otherwise, proceed with the next iteration of the loop. (The bound on k follows from [15].)
4) Compute a witness for the nonemptiness of $A_{\text{NBT}k}$ and convert it to a Moore machine.

If the UCT constructed in Step 2 is weak, synthesis is much simpler: we complement the acceptance condition of A_{UCT} turning it into a UWT, a special case of an AWT. Then, we convert the UWT into an NBT A_{NBT} as in Step 3b. If $L(A_{\text{NBT}})$ is nonempty, the witness is a Moore machine satisfying φ, if it is empty, φ in unrealizable. In this case, we avoid increasing k and the size of the NBT is at most 2^{2n}.

It turns out that in practice, for realizable specifications, the algorithm terminates with very small k, often around three. It should be noted that it is virtually impossible to prove the specification unrealizable using this approach, because of the high bound on k. The one exception is if the UCT is weak, because in that case we avoid the dependence on k altogether, as explained above.

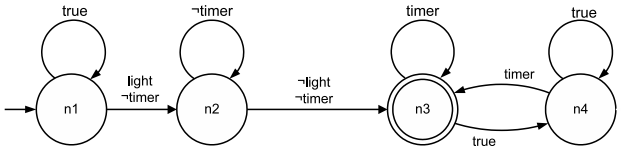

Fig. 1. NBW for $\neg\varphi = \mathsf{G}(\mathsf{F}(\text{timer})) \wedge \mathsf{F}(\text{light} \wedge (\neg\text{light}\,\mathsf{R}\,\neg\text{timer}))$

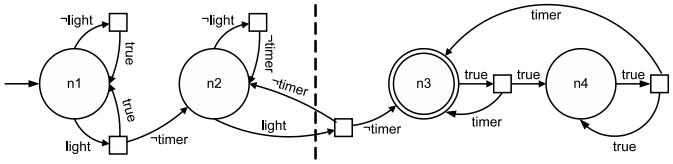

Fig. 2. UCT for $\varphi = \mathsf{G}(\mathsf{F}(\text{timer})) \rightarrow \mathsf{G}(\text{light} \rightarrow (\text{light}\,\mathsf{U}\,\text{timer}))$

In the following, we will describe the individual steps, discuss the optimizations that we use at every step, and show how to reuse information gained in one iteration of the loop for the following iterations.

B. NBW & UCT

We use Wring [6] to construct a nondeterministic generalized Büchi automaton for the negation of the specification. We then use the classic counting construction and the optimizations available in Wring to obtain a small NBW A_{NBW} with $L(A_{\text{NBW}}) = (D \cup \Sigma)^\omega \setminus L(\varphi)$.

We construct a UCT A_{UCT} over Σ-labeled D-trees with $L(A_{\text{UCT}}) = \Lambda((\Sigma \cup D)^\omega \setminus L(A_{\text{NBW}}))$.

Definition 6: [1] Given an NBW $A_{\text{NBW}} = (\Sigma, D, Q, q_0, \delta, \alpha)$, let UCT $A_{\text{UCT}} = (\Sigma, D, Q, q_0, \delta', \alpha)$, with for every $q \in Q$ and $\sigma \in \Sigma$

$$\delta'(q, \sigma) = \big\{\{(d, q') \mid d \in D, q' \in \delta(q, d \cup \sigma)\}\big\}.$$

□

We have $L(A_{\text{UCT}}) = T_{\Sigma,D} \setminus \Lambda(A_{\text{NBW}})$.

We can reduce the size of $L(A_{\text{UCT}})$ using game-based simulation and Theorem 1. Optimizing the UCT reduces the time spent optimizing the AWT and, most importantly, it may make the UCT weak, which means that we avoid the expensive construction of the AWT discussed in the next section. Because the UCT is small in comparison to the AWT and the NBT, optimization comes at a small cost.

Specifications are often of the form $\varphi \rightarrow \psi$, where φ is an assumption on the environment and ψ describes the allowed behavior of the system. When the system assertion ψ has been violated, it may still be possible to satisfy the specification by a violation of the environment assumption. However, since the system cannot control the environment, states that check that φ is violated once the system assertion ψ has been violated are not necessary. Such states, among others, are removed by the game-based optimization.

Example 7: Let $\varphi = \mathsf{G}\,\mathsf{F}\,\text{timer} \rightarrow \mathsf{G}(\text{light} \rightarrow (\text{light}\,\mathsf{U}\,\text{timer}))$. This formula is part of the specification of a traffic light controller and states that if the timer signal is set regularly, the light does not make a transition to zero unless the timer is high. The atomic propositions are partitioned into $I = \{\text{timer}\}$ and $O = \{\text{light}\}$. Fig. 1 shows a minimal NBW A_{NBW} accepting all words in $\neg\varphi$. The edges in the figure (and in the implementation) are labeled with cubes over the set of atomic propositions $I \cup O$. (A cube is a conjunct consisting of possibly negated atomic propositions.) An edge labeled with the cube c summarizes a set of edges, each labeled with a letter $w \subseteq I \cup O$ that is compatible with c.

The UCT A_{UCT} that accepts all 2^O-labeled 2^I-trees not in $T(A_{\text{NBW}})$ is shown in Fig. 2. Circles denote states and boxes denote transitions. We label edges starting at circles with cubes over O ($\Sigma = 2^O$)) and edges from boxes with cubes over I ($D = 2^I$). The transition corresponding to a box C consists of all pairs (d, q) for which there is an edge from C to q such that d satisfies the label on the edge. In particular, if d satisfies none of the labels, the branch in direction d is finite, e.g., in state n_2 with light=0 and timer=1. Recall that finite branches are accepting.

Even though the NBW is optimized, the UCT is not minimal: The tree languages $L({A_{\text{UCT}}}^{n3})$ and $L({A_{\text{UCT}}}^{n4})$ are empty. Our algorithm finds both states and replaces them by transitions to false, removing the part of A_{UCT} to the right of the dashed line. Note that the optimizations cause the automaton to become weak. □

C. AWT

From the automaton A_{UCT} we construct an AWT $A_{\text{AWT}k}$ such that $L(A_{\text{AWT}k}) \subseteq L(A_{\text{UCT}})$

Definition 8: [1] Let $A_{\text{UCT}} = (\Sigma, D, Q, q_0, \delta, \alpha)$, let $n = |Q|$ and let $k \in \mathbb{N}$. Let $[k]$ denote $\{0, \ldots, k\}$. We construct $A_{\text{AWT}k} = (\Sigma, D, Q', q_0', \delta', \alpha')$ with

$$\begin{aligned}
Q' &= \{(q, i) \in Q \times [k] \mid q \notin \alpha \text{ or } i \text{ is even}\},\\
q_0' &= (q_0, k),\\
\delta'((q, i), \sigma) &= \big\{\{(d_1, (q_1, i_1)), \ldots, (d_k, (q_k, i_k))\} \mid\\
&\qquad \{(d_1, q_1), \ldots, (d_k, q_k)\} \in \delta(q, \sigma),\\
&\qquad i_1, \ldots, i_k \in [i], \forall j : (q_j, i_j) \in Q'\big\}\\
\alpha' &= Q \times \{1, 3, \ldots, 2k-1\}.
\end{aligned}$$

We call i the *rank* of an AWT state (q, i). □

If $k = 2n^{n+2}$ we have $L(A_{\text{AWT}k}) = \emptyset$ implies $L(A_{\text{UCT}}) = \emptyset$ [1], [15].

We improve this construction in three ways: by using games, by merging directions, and by using simulation relations.

1) Game Simulation: We can use Theorem 1 to remove states from $A_{\text{AWT}k}$.

Example 9: Consider the UCT in Fig. 3 and the corresponding AWT in Fig. 4, using $k = 5$. The UCT (an artificial example) has been optimized using the techniques discussed in Section IV-B, and the AWT has been optimized in three ways: We have removed states that are not reachable from the initial state, we have merged directions, and we have removed edges. (The last two optimizations are explained in the next subsections). Still, there is ample room for improvement of the AWT.

Application of Theorem 1 removes the 12 states below the dashed line on the bottom left and the incident edges. This is a

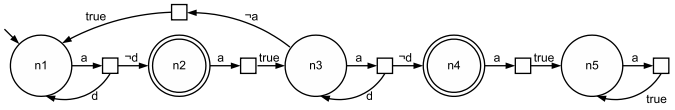

Fig. 3. UCT that requires rank 5. Edges that are not shown (for instance from n_4 with label $\neg a$) correspond to labels that are not allowed.

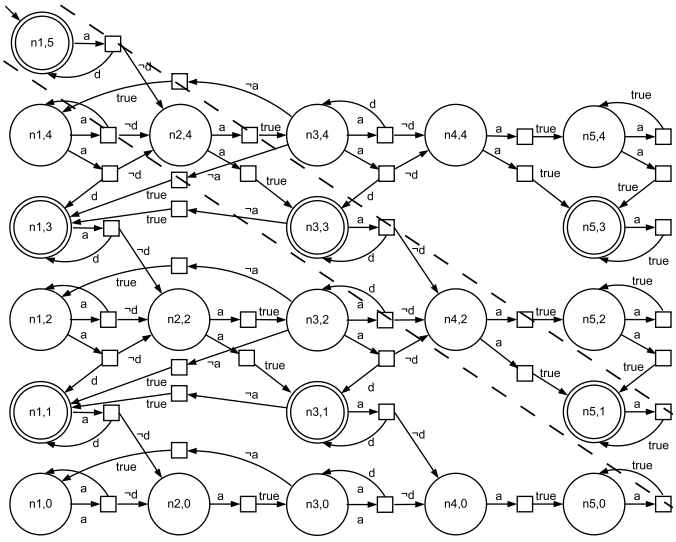

Fig. 4. AWT for UCT in Figure 3.

typical situation: each UCT state has an associated minimum rank. □

It should be noted that $A_{\text{AWT}k}$ has a layered structure: there are no states with rank j with a transition back to a state with a rank $i > j$. Furthermore, $A_{\text{AWT}k+1}$ consists of $A_{\text{AWT}k}$ plus one layer of states with rank $k+1$. This implies that game information computed for $A_{\text{AWT}k}$ can be reused for $A_{\text{AWT}k+1}$. A play is won (lost) in $A_{\text{AWT}k+1}$ if it reaches a states that is won (lost) in $A_{\text{AWT}k}$. Furthermore, if (q, j) is won, then so is (q, i) for $i > j$ when i is odd or j is even, which allows us to reuse some of the information computed for states with rank k when adding states with rank $k+1$. This follows from the fact that (q, i) simulates (q, j), as will be discussed in Subsection IV-C.3.

2) Merging Directions: Note that δ' may be drastically larger than δ: a single transition $C \in \delta(q, \sigma)$ yields $i^{|C|}$ transitions out of state $(q, i) \in Q'$. However, it turns out that it is not necessary to include conjuncts that send a copy to a (q, j) and (q, j') for $j \neq j'$. This is fortunate because it allows us to treat edges labeled with cubes over I as if they were labeled with directions.

Theorem 10: Let $A''_{\text{AWT}k} = (\Sigma, D, Q', q'_0, \delta'', \alpha')$ be as in Definition 8, but with

$$\delta''((q,i),\sigma) = \{C \in \delta'((q,i),\sigma) \mid \forall (d,(q,j)),(d',(q,j')) \in C, \text{ we have } j = j'\}.$$

We have $L(A''_{\text{AWT}k}) = L(A_{\text{AWT}k})$. □

Proof: Because $\delta''(q, \sigma) \subseteq \delta'(q, \sigma)$, any tree accepted by $A''_{\text{AWT}k}$ is also accepted by $A_{\text{AWT}k}$.

Let R be a run of $A_{\text{AWT}k}$, we will build a run R'' of $A''_{\text{AWT}k}$. Run R'' is isomorphic to R, using a bijection that maps node v of R to node v'' of R''. Run R'' has the same labels as R with the following exception. If node v in R is labeled $(t, (q, i))$ and has children $(t', (q', i'))$ and $(t'', (q', i''))$ with $i' > i''$, then the corresponding children of node v'' of R'' are labeled $(t', (q', i'))$ and $(t'', (q', i'))$.

Because in $A_{\text{AWT}k}$ state (q', i') has all transitions that (q', i'') has, R'' is a run of $A_{\text{AWT}k}$, and because it satisfies the extra condition on δ'' it is also a run of $A''_{\text{AWT}k}$. If R is accepting, then every infinite path π in R gets stuck in an odd rank w from some level l onwards. So starting from l, all children of nodes on π have rank at most w. That implies that the nodes on π in R'' have rank w starting at rank $l+1$ at the latest. Thus, π is still accepting, and since π is arbitrary, R'' is accepting as well. ■

This theorem is key to an efficient implementation as it allows us to represent a set of pairs $\{(d_1, q), \ldots, (d_k, q)\}$ as $(\{d_1, \ldots, d_k\}, q)$ whenever $\{d_1, \ldots, d_k\}$ can efficiently be represented by a cube over the input signals I.

3) Simulation minimization: We compute the simulation relation on $A_{\text{AWT}k}$ and use Theorems 3, 4, and 5 to optimize the automaton. We would like to point out one optimization in particular.

Lemma 11: For $(q, i), (q, j) \in Q'$ with $i \geq j$ such that i is odd or j is even, we have $(q, i) \succeq (q, j)$. □

Thus, for any σ, if i is even, we can remove all transitions $C \in \delta((q, i), \sigma)$ that include a pair (q', j) for $j \leq i - 2$. If i is odd we can additionally remove all transitions that contain a pair (q', j) with $q \notin \alpha$ and $j = i - 1$. That is, odd states become deterministic and for even states there are at most two alternatives to choose from.

Theorem 12: Let $A'_{\text{AWT}k} = (\Sigma, D, Q', q'_0, \delta'', \alpha')$ as in Definition 8, but with

$$\delta''((q,i),\sigma) = \big\{C \in \delta(q,\sigma) \mid \forall (d',(q',i')) \in C : i' \in \{i-1, i\}, (i \text{ is even} \vee q' \in \alpha \vee i' = i), \forall (d'',(q',i'')) \in C : i' = i''\}.$$

then $L(A'_{\text{AWT}k}) = L(A_{\text{AWT}k})$. □

Example 13: States $(n_4, 4)$, $(n_5, 4)$, and $(n_5, 3)$ (top right) are simulation equivalent with $(n_4, 2)$, $(n_5, 2)$, and $(n_5, 1)$, respectively. Using Theorem 3, we can remove states $(n_4, 4)$, $(n_5, 4)$, and $(n_5, 3)$, and redirect incoming edges to equivalent states.

Furthermore, the previous removal of the states on the bottom left implies that $(n_3, 4) \preceq (n_3, 3)$. Since $(n_2, 4)$ has identical transitions to $(n_3, 4)$ and $(n_3, 3)$, Theorem 5 allows us to remove the transition to $(n_3, 4)$. Thus, $(n_3, 4)$ becomes unreachable and can be removed. The same holds for $(n_5, 2)$ for a similar reason. (This optimization also allows us to remove states $(n_4, 4)$, $(n_5, 4)$, and $(n_5, 3)$, but Theorem 5 is not in general stronger than Theorem 3.)

The optimization of the edges due to Theorem 12 is already shown in Fig. 4. Consider, for instance, the transition from $(n_2, 4)$ to $(n_3, 4)$.

Altogether, we have reduced the number of states in the AWT from 22 to 5. The removal of edges is equally important as it reduces nondeterminism and makes the translation to an NBT more efficient. □

D. NBT

The next step is to translate $A_{\text{AWT}k}$ to an NBT $A_{\text{NBT}k}$ with the same language. Assume that $A_{\text{AWT}k} = (\Sigma, D, Q, q_0, \delta, \alpha)$. We first need some additional notation. For $S \subseteq Q$ and $\sigma \in \Sigma$ let

$$sat(S, \sigma) = \{C \in 2^{D \times Q} \mid C \text{ is a minimal set such that } \forall q \in S \, \exists C_q \in \delta(q, \sigma) : C_q \subseteq C\}.$$

For $(S, O) \in 2^Q \times 2^Q$, let

$$sat((S, O), \sigma) = \{(S', O') \in 2^Q \times 2^Q \mid S' \in sat(S, \sigma), O' \in sat(O, \sigma), O' \subseteq S'\}.$$

Furthermore, let $S_d = \{s \mid (d, s) \in S\}$, let $O_d = \{s \mid (d, s) \in O\}$. Let $C_N(S, O) = \{(d, (S_d, O_d \setminus \alpha)) \mid d \in D\}$ and let $C_\emptyset(S) = \{(d, (S_d, S_d \setminus \alpha)) \mid d \in D\}$.

Definition 14: [1], [16] Let $A_{\text{NBT}k} = (\Sigma, D, 2^Q \times 2^{Q \setminus \alpha}, (\{q_0\}, \emptyset), \delta', 2^Q \times \emptyset)$ with

$$\delta'((S, O), \sigma) = \begin{cases} \{C_N(S', O') \mid (S', O') \in sat((S, O), \sigma)\} & \text{if } O' \neq \emptyset \\ \{C_\emptyset(S') \mid S' \in sat(S, \sigma)\} & \text{otherwise} \end{cases}$$

□

We have $L(A_{\text{NBT}k}) = L(A_{\text{AWT}k})$.

We improve this construction in three ways. First, we make use of the simulation relation on the AWT to reduce the size of the NBT. Second, we remove *inconsistent states*, and third, we compute the NBT on the fly.

1) Simulation-Based Optimization: We can use the simulation relation that we have computed on $A_{\text{AWT}k}$ to approximate the simulation relation on $A_{\text{NBT}k}$. This is a simple extension of Fritz' result for word automata [17].

Given a direct simulation relation $\preceq_{\text{AWT}}$ for $A_{\text{AWT}k}$, we define the simulation relation $\preceq' \subseteq Q' \times Q'$ on $A_{\text{NBT}k}$ as

$$(S_1, O_1) \preceq' (S_2, O_2) \text{ iff } \forall q_2 \in S_2 \, \exists q_1 \in S_1 : q_1 \preceq_{\text{AWT}} q_2 \wedge (q_2 \in O_2 \rightarrow q_1 \in O_1).$$

Note that $\preceq'$ is a subset of the full (direct) simulation relation on $A_{\text{NBT}k}$ and thus, the following lemma holds.

Lemma 15: $(S_1, O_1) \preceq' (S_2, O_2)$ implies $L(A^{(S_1, O_1)}) \subseteq L(A^{(S_2, O_2)})$. □

In particular, for a state $(S, O) \in Q'$, if $q, q' \in S$, $q \preceq_{\text{AWT}} q'$, and $q' \in O \rightarrow q \in O$, then $(S, O) \simeq (S \setminus \{q'\}, O \setminus \{q'\})$. Thus, by Theorem 3, we can remove q' from such sets. Likewise, if $A_{\text{NBT}k}$ contains two simulation equivalent states (S, O) and (S', O') we keep only one (preferring the one with smaller cardinality). Finally, we can use Theorem 5 to remove states that have a simulating sibling.

2) Removing Inconsistent States: In [1], it is shown that it is not necessary to include states (S, O) such that (q, i) and $(q, j) \in S$ with $i \neq j$. This implies that we can use the following optimization.

Theorem 16: Let $A'_{\text{NBT}k} = (\Sigma, D, Q'', (\{q_0\}, \emptyset), \delta'', 2^Q \times \emptyset)$ be as in Definition 14, with $Q'' = Q \setminus \{(S, O) \mid \exists (q, i), (q, j) \in S : i \neq j\}$. The transition relation δ'' is obtained from δ' by replacing, for all $C \in \delta'(q, \sigma)$ and all $(S, O) \in C$, state (S, O) by (S', O') where S' is obtained from S by removing all states (q, j) with j not minimal and O' is obtained from O by replacing $(q, j) \in O$ by (q, j') if $(q, j) \notin S'$ and $(q, j) \in S'$.

We have $L(A'_{\text{NBT}k}) = L(A_{\text{NBT}})$. □

This is an important theorem as it reduces the number of states in the NBT to $(k+1)^{2n}$ instead of 2^{nk}, where n is the number of states in A_{UCT}.

3) On-the-Fly Computation: Suppose $A_{\text{NBT}k} = (\Sigma, D, Q, q_0, \delta, \alpha)$. Instead of building $A_{\text{NBT}k}$ in full, we construct an NBT $A'_{\text{NBT}}[k] = (\Sigma, D, Q', q_0, \delta', \alpha \cap Q')$ such that $q_0 \in Q' \subseteq Q$ and for $q \in Q'$, either $\delta'(q, \sigma) = \delta(q, \sigma)$ for all σ or $\delta'(q, \sigma) = \emptyset$ for all σ. Thus, $L(A'_{\text{NBT}k}) \subseteq L(A_{\text{NBT}k})$. If $L(A'_{\text{NBT}k}) \neq \emptyset$, the witness of nonemptiness of $L(A'_{\text{NBT}k})$ is a witness of nonemptiness of $L(A_{\text{NBT}k})$. Otherwise, we select a state $q \in Q'$ with $\delta'(q, \sigma) = \emptyset$ and *expand* it, setting $\delta'(q, \sigma) = \delta(q, \sigma)$, introducing the necessary states to Q'.

Our current heuristic expands states in a breadth first manner, which is quite effective. It may be beneficial to expand certain state first, say states with a low cardinality or with high ranks.

E. Moore Machine

We use the game defined in Section III-A to compute language emptiness on the $A_{\text{NBT}k}$. Since $A_{\text{NBT}k}$ is nondeterministic, all states in the winning region have a nonempty language. If the initial state is in the winning region, the language of $A_{\text{NBT}k}$ is not empty and we extract a witness.

Since $A_{\text{NBT}k}$ is a subset of $A_{\text{NBT}k+1}$, we can reuse all results obtained when computing language emptiness on $A_{\text{NBT}k}$ to compute language emptiness on $A_{\text{NBT}k+1}$.

Moreover, it follows from Miyano and Hayashi's construction that if $L(A^{(S,O)}) \neq \emptyset$ and $S \subseteq S'$, then $L(A^{(S',O')}) \neq \emptyset$. We may use this fact to further speed up the computation of language emptiness, and especially to reuse information obtained computing language emptiness on $A_{\text{NBT}k}$ for larger k.

A witness for nonemptiness corresponds to a winning *attractor strategy* [18]. The winning strategy follows the μ-iterations of the final ν-computation of $W_B(\alpha)$: From a state $q \notin \alpha$ we go to a state q' from which the protagonist can force a shorter path to an accepting state. In an accepting state we move back to an arbitrary state in the winning region.

If a strategy exists, it corresponds to a complete Σ-labeled D-tree and thus to a Moore machine M. The states of M are the states of $A_{\text{NBT}k}$ that are reachable when the strategy is followed, and the edges are given by the strategy.

To minimize the strategy, we compute the simulation relation and apply Theorem 3, which is equivalent to using the classical FSM minimization algorithm [19]. Thus, the optimized strategy is guaranteed to be minimal with respect to its given I/O language. The output of our tool is a state machine described in Verilog that implements this strategy.

V. Experimental Results

Our tool, Lily[1], is an implementation of the synthesis algorithm and the optimizations described in this paper. It is

[1] http://www.ist.tugraz.at/staff/jobstmann/lily/

```
module tlc(hl,fl,clk,car,timer);
  input  clk,car,timer;
  output hl,fl;
  wire clk,hl,fl,car,timer;
  reg [0:0] state;

  assign hl = (state == 0);
  assign fl = (state == 1);

  initial begin
    state = 0;
  end
  always @(posedge clk) begin
    case(state)
    0: begin
      if (timer==0) state = 0;
      if (timer==1 && car==1) state = 1;
      if (car==0) state = 0;
    end
    1: begin
      if (timer==1) state = 0;
      if (timer==0) state = 1;
    end
    endcase
  end
endmodule //tlc
```

Fig. 5. Generated design for a simple traffic light

implemented on top of Wring [6], [20] and written in Perl. We have run our experiments on a Linux machine with a 2.8 GHz Pentium 4 CPU and 2 GB of RAM.

Example 17: Consider the following formula.

$$\begin{aligned}&\mathsf{G}\,\mathsf{F}\,\mathit{timer} \rightarrow \\ &\quad \big(\mathsf{G}(\mathit{hl} \rightarrow (\mathit{hl}\,\mathsf{U}\,\mathit{timer})) \wedge \mathsf{G}(\mathit{fl} \rightarrow (\mathit{fl}\,\mathsf{U}\,\mathit{timer})) \wedge \\ &\quad \mathsf{G}(\neg \mathit{hl} \vee \neg \mathit{fl}) \wedge \mathsf{G}(\mathit{car} \rightarrow \mathsf{F}\,\neg \mathit{car} \vee \mathit{fl}) \wedge \mathsf{G}\,\mathsf{F}\,\mathit{hl} \wedge \\ &\quad\qquad \mathsf{G}(\mathit{hl} \rightarrow (\mathit{hl}\,\mathsf{W}\,\mathit{car}))\big).\end{aligned}$$

The formula specifies a small traffic light system consisting of two lights. The highway light is green iff $hl = \mathsf{true}$, and similarly for the crossing farm road and fl. Signals hl and fl form the output. The input signal car indicates that a car is waiting at the farm road and $timer$ represents the expiration of a timer. The specification assumes that the timer expires regularly. It requires that a green light stay green until the timer expires. Furthermore, one of the lights must always be red, every car at the farm road is eventually allowed to drive on (unless it disappears), the highway light is regularly set to green, and it does not become red until there is a car that wants to cross the highway. The specification is realizable and the design generated by Lily is shown in Figure 5. □

We show the effectiveness of the various optimizations by synthesizing 20 handwritten formulas, mostly different arbiters and some traffic light controllers. Our examples are small, but we show a significant improvement over the straightforward implementation.

For realizable formulas, we have verified the output of our tool with a model checker. We can verify that a formula is unrealizable by synthesizing an environment that forces the specification to be violated. Since a system is a Moore machine, an environment is a Mealy machine. Note that φ can be realized by a Mealy machine iff φ' can be realized by a Moore machine, where φ' is obtained from φ by replacing all occurences of an output o by $\mathsf{X}\,o$. This means that we can apply our approach to check that the environment is realizable.

We show our results in Table I. (The specification in example 17 has number 9.) In the column labeled *T,B,AP*, we provide the number of temporal operators, the number of Boolean operators, and the number of atomic propositions in the formula. We also give the strength in Column *strength* and the number of states and edges of the optimized NBW in Column *NBW* using the format states(edges).

The next six columns report time used with different combinations of optimizations. We write "> mem" if the run has exceeded the memory limit and "$> 3600s$" if it did not finish within one hour.

In Column *Plain* we give the time using only one optimization: transitions from a state with rank i go only to states with ranks $i-1$ and $i-2$, not to smaller ranks. Without this optimization, synthesis is impossible on most examples. Column *Plain+dm* shows the time used if we apply Theorem 10, which allows us to merge directions. In Column *UCT+dm*, we give the time usage of runs in which we applied game optimization (Theorem 1) on the UCT and we merged the directions. We show the results for applying all the optimizations suggested in Section IV-C on the AWT in Column *AWT+dm*. Column *UCT+dm* shows the time used if we apply the NBT optimizations and merge directions.

In the Column *All* we give the results for combining all optimizations. For realizable formulas, the number of states and edges of the design generated during those runs is given in the column labeled with *Witness*. We write n.r. if a formula is not realizable. The generated designs are minimized as described in Section IV-E.

In Column *NBT-all* we give the size (states(edges)) of the NBT using all optimizations. In contrast, in Column *NBT-plain+dm*, we show the size of the NBTs generated by the runs where we used only direction merging.

Our examples show that if the NBW for $\neg\varphi$ is strong and we have to construct the AWT, the straightforward implementation often fails to complete. (See Column *Plain*.) Five examples exceed the memory limit due to the expensive construction of the transition relation of the AWT. The optimizations due to Theorem 10, which allows us to merge directions, are necessary to overcome this limit. (See Column *Plain+dm*.)

If we optimize the UCT according to Theorem 1, we speed up about half of the examples by a factor of two or more. (Compare Column *Plain+dm* and *UCT+dm*, e.g., Line 2,3, or 9). The game based minimization is very effective if the formula or part of it is not realizable and is very cheap to compute. In particular for Examples 3 the difference is huge because the language of the optimized UCT is empty.

None of the optimizations on the AWT was very effective on their own. For example we still had two timeouts with game based optimization. The same holds for simulation based optimization. Further simplification of δ' according to Theorem 12 resulted in one timeout. Nevertheless, these optimizations are very efficient when combined as can be seen in Column *AWT+dm*.

TABLE I

No	*T,B,AP*	*Strength*	*NBW*	*Plain*	*Plain+dm*	*UCT+dm*	*AWT+dm*	*NBT+dm*	*All*	*NBT-plain+dm*	*NBT-all*	*Witness*
1	12,5,4	weak	8(14)	2.94 s	1.96 s	0.71 s	0.73 s	2.53 s	0.38 s	48(192)	0(0)	n. r.
2	6,3,4	strong	7(15)	> mem	1689.13 s	575.70 s	2.20 s	2.43 s	1.25 s	3943(973764)	6(28)	2(3)
3	4,4,4	strong	12(44)	> mem	> mem	3.73 s	>3600 s	> 3600 s	3.95 s	-	0(0)	n. r.
4	9,5,4	strong	6(11)	12.90 s	3.14 s	2.63 s	0.61 s	0.74 s	0.66 s	95(1104)	8(37)	6(11)
5	9,6,4	weak	7(19)	0.61 s	0.62 s	0.68 s	0.64 s	0.62 s	0.69 s	14(65)	14(65)	10(25)
6	15,12,4	weak	9(30)	3.43 s	2.94 s	3.01 s	2.97 s	3.26 s	3.15 s	58(502)	58(502)	43(145)
7	13,5,5	strong	7(15)	69.14 s	20.07 s	12.83 s	1.23 s	3.42 s	1.24 s	384(9564)	26(164)	15(31)
8	20,9,7	strong	9(22)	> 3600 s	258.81 s	294.29 s	6.41 s	13.32 s	5.98 s	1810(75264)	80(811)	41(109)
9	11,9,4	strong	9(19)	> mem	113.08 s	9.29 s	14.90 s	5.57 s	8.90 s	1079(81700)	101(840)	2(5)
10	12,5,4	weak	8(12)	3.64 s	1.28 s	0.62 s	0.70 s	1.31 s	0.35 s	28(116)	0(0)	n. r.
11	23,21,5	strong	24(90)	> mem	> mem	> 3600 s	17.57 s	94.29 s	15.91 s	-	31(167)	6(13)
12	40,24,8	weak	14(84)	201.18 s	219.38 s	58.82 s	61.02 s	37.95 s	32.41 s	251(88016)	26(456)	5(41)
13	10,9,4	strong	24(134)	> mem	> 3600 s	> 3600 s	522.21 s	> 3600 s	46.37 s	360(440093)	23(127)	17(75)
14	14,9,4	weak	13(34)	7.15 s	6.09 s	4.08 s	4.44 s	4.54 s	2.39 s	82(730)	21(92)	7(22)
15	16,10,10	weak	24(68)	90.63 s	68.49 s	13.60 s	13.10 s	17.00 s	8.29 s	618(8326)	27(142)	n. r.
16	16,13,4	weak	18(50)	8.65 s	6.33 s	6.45 s	7.23 s	7.71 s	3.93 s	142(1492)	17(112)	8(31)
17	19,13,4	weak	25(69)	24.52 s	21.88 s	11.28 s	12.97 s	14.46 s	8.18 s	368(3716)	25(162)	12(54)
18	16,17,4	weak	17(45)	11.14 s	9.14 s	4.58 s	5.51 s	6.05 s	4.14 s	118(730)	13(53)	8(23)
19	28,14,9	strong	11(30)	> mem	> mem	> 3600 s	72.78 s	483.38 s	39.26 s	755(83106)	242(3834)	124(444)
20	8,6,2	strong	7(14)	3.21 s	2.98 s	1.24 s	1.13 s	0.82 s	0.73 s	112(1187)	5(10)	2(3)

The simulation-based optimizations for the NBT (explained in Section IV-D) typically reduce the size of the resulting NBT between 60% and 90%. For example, in Example 9 the size of the NBT is reduced from about 1000 states to 300. For this example, the on-the-fly game computation further reduces the number of NBT-states to about 100. Only in the small examples is the entire state space of the NBT needed to compute a witness. (Cf. Column *NBT-plain+dm,NBT-all,Witness*).

In our examples, the UCT optimizations were crucial once to turn a strong UCT into a weak one (in this case with an empty language). In all other cases, they are outperformed by the AWT optimizations. The results of the NBT optimization are mixed: they can fail (Example 3 and 13) or perform better than any other optimization (Example 9 and 12). The combination of all optimizations is needed to finish all examples.

VI. Summary

The paper described the first implementation of a synthesis tool for full LTL. We have presented a set of optimizations for tree automata and have shown how these make a major difference in the efficiency of the implementation.

Further work is concerned with further increases in efficiency, debugging of specifications (especially of unrealizable ones), and with ways to effectively combine specifications with hand-written HDL code.

Acknowledgments: The authors would like to thank Orna Kupferman for stimulating discussions and patient explanations.

References

[1] O. Kupferman and M. Vardi, "Safraless decision procedures," in *Symposium on Foundations of Computer Science (FOCS'05)*, 2005, pp. 531–542.

[2] A. Pnueli and R. Rosner, "On the synthesis of a reactive module," in *Proc. Symposium on Principles of Programming Languages (POPL '89)*, 1989, pp. 179–190.

[3] R. Rosner, "Modular synthesis of reactive systems," Ph.D. dissertation, Weizmann Institute of Science, 1992.

[4] S. Safra, "On the complexity of ω-automata," in *Symposium on Foundations of Computer Science*, Oct. 1988, pp. 319–327.

[5] M. Vardi, "A game-theoretic approach to automated program generation," 2005, Presentation at IFIP Working Group 2.11 Second Meeting. Available from http://www.cs.rice.edu/ taha/wg2.11/m-2/.

[6] F. Somenzi and R. Bloem, "Efficient Büchi automata from LTL formulae," in *Twelfth Conference on Computer Aided Verification (CAV'00)*, E. A. Emerson and A. P. Sistla, Eds. Berlin: Springer-Verlag, July 2000, pp. 248–263, LNCS 1855.

[7] N. Wallmeier, P. Hütten, and W. Thomas, "Symbolic synthesis of finite-state controllers for request-response specifications," in *Proceedings of the International Conference on the Implementation and Application of Automata*. Springer-Verlag, 2003.

[8] A. Harding, M. Ryan, and P. Schobbens, "A new algorithm for strategy synthesis in LTL games," in *Tools and Algorithms for the Construction and the Analysis of Systems (TACAS'05)*, 2005, pp. 477–492.

[9] N. Piterman, A. Pnueli, and Y. Sa'ar, "Synthesis of reactive(1) designs," in *Proc. Verification, Model Checking, and Abstract Interpretation (VMCAI'06)*, 2006, pp. 364–380.

[10] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model Checking*. Cambridge, MA: MIT Press, 1999.

[11] R. Alur, T. A. Henzinger, O. Kupferman, and M. Y. Vardi, "Alternating refinement relations," in *Proc. 9th Conferance on Concurrency Theory*. Nice: Springer-Verlag, Sept. 1998, pp. 163–178, LNCS 1466.

[12] C. Fritz and T. Wilke, "State space reductions for alternating Büchi automata," in *Foundations of Software Technology and Theoretical Computer Science*. Kanpur, India: Springer-Verlag, Dec. 2002, pp. 157–168, LNCS 2556.

[13] S. Gurumurthy, O. Kupferman, F. Somenzi, and M. Y. Vardi, "On complementing nondeterministic Büchi automata," in *Correct Hardware Design and Verification Methods (CHARME'03)*. Berlin: Springer-Verlag, Oct. 2003, pp. 96–110, LNCS 2860.

[14] P. Wolper, M. Y. Vardi, and A. P. Sistla, "Reasoning about infinite computation paths," in *Proceedings of the 24th IEEE Symposium on Foundations of Computer Science*, 1983, pp. 185–194.

[15] N. Piterman, "From nondeterministic Büchi and Streett automata to deterministic parity automata," in *21st Symposium on Logic in Computer Science (LICS'06)*, 2006, To appear.

[16] S. Miyano and T. Hayashi, "Alternating finite automata on ω-words," *Theoretical Computer Science*, vol. 32, pp. 321–330, 1984.

[17] C. Fritz, "Constructing Büchi automata from linear temporal logic using simulation relations for alternating Büchi automata," in *Conference on Implementation and Application of Automata (CIAA'03)*, O. H. Ibarra and Z. Dang, Eds., 2003, pp. 35–48, LNCS 2759.

[18] W. Thomas, "On the synthesis of strategies in infinite games," in *Proc. 12th Annual Symposium on Theoretical Aspects of Computer Science*. Springer-Verlag, 1995, pp. 1–13, LNCS 900.

[19] J. E. Hopcroft and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*. Reading, MA: Addison-Wesley, 1979.

[20] S. Gurumurthy, R. Bloem, and F. Somenzi, "Fair simulation minimization," in *Fourteenth Conference on Computer Aided Verification (CAV'02)*, E. Brinksma and K. G. Larsen, Eds. Berlin: Springer-Verlag, July 2002, pp. 610–623, LNCS 2404.