Symbolic Bounded Synthesis*

Rüdiger Ehlers

Reactive Systems Group
Saarland University

Abstract. Synthesis of finite state systems from full linear time temporal logic (LTL) specifications is gaining more and more attention as several recent achievements have significantly improved its practical applicability. Many works in this area are based on the Safraless synthesis approach. Here, the computation is usually performed either in an explicit way or using symbolic data structures other than binary decision diagrams (BDDs). In this paper, we close this gap and consider Safraless synthesis using BDDs as state space representation. The key to this combination is the application of novel optimisation techniques which decrease the number of state bits in such a representation significantly. We evaluate our approach on several practical benchmarks, including a new load balancing case study. Our experiments show an improvement of several orders of magnitude over previous approaches.

1 Introduction

Ensuring the correctness of a system is a difficult task. Bugs in manually constructed hard- or software are often missed during testing. To remedy this problem, two lines of research have emerged. The first one deals with the verification of systems that have already been built and spans topics such as process calculi and model checking. The second line concerns the automatic derivation of systems that are correct by construction, also called *synthesis*. In both cases, the specification of the system needs to be given, but we can save the work of constructing the actual system in the case of synthesis.

Unfortunately, the complexity of synthesis has been proven to be rather high. For example, when given a specification in form of a property in linear time temporal logic (LTL), the synthesis task has a complexity that is doubly-exponential in the size of the specification [17]. Recently, it has been argued that this is however not a big problem [18] as *realisable* practical specifications typically have implementations that are small, which can be exploited. This observation is used in the context of *bounded synthesis* [18, 8], which builds upon the *Safraless synthesis* principle [14]. Here, the LTL specification is converted to a universal co-Büchi word or tree automaton, which is then, together with a bound $b \in \mathbb{N}$, used

* This work was supported by the German Research Foundation (DFG) within the program "Performance Guarantees for Computer Systems" and the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS).

for building a safety game such that winning strategies in the game correspond to implementations satisfying the specification. The bound in this setting describes the maximum allowed number of visits to rejecting states in the co-Büchi automaton. If there exists an implementation satisfying a given specification, then there exists some bound such that the resulting game is winning.

In practice, the bound required is usually rather small, often much smaller than the number of states in the smallest implementation. This leads to improved running times of implementations following this approach. Consequently, all modern tools for full LTL synthesis publicly available nowadays build upon Safraless synthesis. The first of these, named Lily [11], performs the realisability check in an explicit way. Recently, a symbolic algorithm based on antichains has been presented [8], showing a better performance on larger specifications. Surprisingly, the usage of binary decision diagrams (BDDs), a technique that has skyrocketed the size of the systems that can be handled by model checking tools [15], seems to be unconsidered in this context so far. A possible explanation for this is that the safety games constructed in the bounded synthesis context contain a lot of *counters* with dependencies between them in the transition relation. It has been observed that this can tremendously blow-up the size of BDDs [22, 19, 3]. Thus, for success using this technique, it is a central requirement that efficient techniques for reducing the number of counters are being used. In this paper we investigate this problem and present such techniques. By taking them together, we can improve upon the performance of previous approaches to full LTL synthesis by several orders of magnitude.

In particular, we show how to split a specification, consisting of assumptions about the environment and guarantees that the system needs to fulfill, into safety and non-safety parts, which can be handled separately in the synthesis game. As for safety properties, no counters are necessary, this reduces the computation time significantly and allows utilising a major strength of BDDs: efficient dealing with automata that run in parallel. Since it has been argued that typical specifications found in practice are mostly of the form $\bigwedge_{a \in A} a \rightarrow \bigwedge_{g \in G} g$ for some sets of assumptions A and guarantees G [2, 20], both containing LTL formulas, we design our technique to be adapted to this case. As a second contribution, we discuss the efficient encoding of the safety and non-safety parts in BDD-based games. Finally, we show how to adapt the techniques presented to checking the *unrealisability* of a given specification in an efficient way. We evaluate our approach on the benchmarks from [11, 8] and also present a new, more complex *load balancing* benchmark that allows for a more meaningful discussion of the practical applicability of our approach.

This paper is structured as follows. In the next section, we briefly discuss the preliminaries and give suitable references for those readers who are not familiar with the fundamentals of bounded synthesis. Then, we show how a specification can be split into safety and non-safety parts without losing soundness or completeness of the synthesis procedure. Section 4 describes how to efficiently encode both parts in a symbolic state space. In Section 5, we continue with the explanation of how the unrealisability of a specification can also be checked with our

approach. Section 6 contains the experimental results of running our prototype tool on the benchmarks from [11] and [8] as well as on a novel load-balancing system case study. We conclude with a summary.

2 Preliminaries

This section describes the fundamentals of the bounded synthesis approach. We choose the notation in a way such that it fits best to the presentation of the new concepts in the remaining sections.

Mealy automata: For the representation of systems to be synthesized, a suitable computation model is required. In this work, we use *Mealy automata* [16]. Formally, a Mealy automaton $\mathcal{M} = (S, I, O, \delta, s_{\text{in}})$ is defined as a 5-tuple with the set of states S, the input set I, the output set O, the transition function $\delta : (S \times I) \to (S \times O)$ and the initial state $s_{\text{in}} \in S$. For the scope of this paper, we assume that the sets S, I and O are finite. We set $I = 2^{\text{AP}_I}$ for some input proposition set AP_I and $O = 2^{\text{AP}_O}$ for some output proposition set AP_O as this facilitates the description of properties of Mealy automata with temporal logic.

Given some input stream $d = d_1 d_2 \ldots \in I^\omega$ to a Mealy automaton, we define the computation of the automaton *induced* by d as $\pi = s_0 s_1 s_2 \ldots \in S^\omega$ s.t. $s_0 = s_{\text{in}}$ and for all *rounds* $j \in \mathbb{N}_0$, we have $\delta(s_j, d_{j+1}) = (s_{j+1}, o)$ for some $o \in O$. Furthermore, the output of $\mathcal{A}$ over d is defined as $\rho = \rho_1 \rho_2 \ldots$ such that for all $j \in \mathbb{N}_0$, we have $\delta(s_j, d_{j+1}) = (s_{j+1}, \rho_{j+1})$. We furthermore say that $w = (d_1 \cup \rho_1)(d_2 \cup \rho_2) \ldots$ is a word induced by $\mathcal{M}$.

Linear time temporal logic (LTL) & universal co-Büchi word automata: For the specification of a system to be synthesized, some description logic is necessary. *Linear time temporal logic (LTL)* has been the predominantly used such logic in previous works. It allows the usage of the *Safraless synthesis approach*, which circumvents the need for constructing deterministic automata from the specification that occurs in other synthesis methodologies.

Due to space restrictions, we do not define LTL and its semantics here but rather refer to [7]. Formulas in LTL can use the temporal operators "G" (globally), "F" (finally), "X" (next time) and "U" (until). We say that some automaton $\mathcal{M}$ satisfies an LTL formula ψ if for all words $w = w_1 w_2 \ldots$ induced by $\mathcal{M}$, we have $w \models \psi$. Some LTL formulas are also called *safety properties*; this is the case if for every word w not satisfying the property, there exists some prefix w' of w such that no word having the same prefix satisfies the property.

Formulas in LTL can be transformed into equivalent *universal co-Büchi word automata* (UCW), i.e., given an LTL formula ψ, a UCW $\mathcal{A}$ of size at most exponential in $|\psi|$ can be obtained such that for every Mealy automaton $\mathcal{M}$, all runs induced by $\mathcal{M}$ are accepted by $\mathcal{A}$ if and only if all words induced by $\mathcal{M}$ satisfy ψ.

We define universal co-Büchi word automata as five-tuples $\mathcal{A} = (Q, \Sigma, \delta, q_{\text{in}}, F)$ with a set of states Q, an alphabet Σ, a transition function $\delta : Q \times \Sigma \to 2^Q$, an initial state $q_{\text{in}} \in Q$ and some set of rejecting states $F \subseteq Q$. Given a word

$w = w_1 w_2 \ldots \in \Sigma^\omega$, we say that a sequence $\pi = \pi_0 \pi_1 \pi_2 \ldots \in Q^\omega$ is a run of $\mathcal{A}$ over w if $\pi_0 = q_{\text{in}}$ and for all $j \in \mathbb{N}_0$, $\pi_{j+1} \in \delta(\pi_j, w_{j+1})$. A word w is *accepted* by $\mathcal{A}$ if for all runs π of $\mathcal{A}$ over w, we have $\inf(\pi) \cap F = \emptyset$ for inf denoting the function that maps a sequence onto the set of elements that occurs infinitely often in it. We say that a Mealy automaton $\mathcal{M}$ is accepted by $\mathcal{A}$ if all words induced by $\mathcal{M}$ are accepted by $\mathcal{A}$. Due to the finiteness of Mealy automata, if $\mathcal{M}$ is accepted by $\mathcal{A}$, there exists a finite upper bound $b(\mathcal{M}, \mathcal{A})$ on the number of rejecting states visited on the runs of $\mathcal{A}$ on any word induced by $\mathcal{M}$. This bound is always at most $|F| \cdot |S|$ for S being the state set of the Mealy automaton [18].

Safety games: Given some universal co-Büchi word automaton $\mathcal{A} = (Q, \Sigma, \delta, q_{\text{in}}, F)$ with $\Sigma = I \times O$ and some bound $b \in \mathbb{N}$, we can build a two-player *safety game* $\mathcal{G}$ such that player 1 wins the game if and only if there exists some Mealy automaton $\mathcal{M}$ over the inputs I and outputs O with $b(\mathcal{M}, \mathcal{A}) \leq b$ [18].

Formally, we define safety games as tuples $\mathcal{G} = (V, \Sigma_0, \Sigma_1, \delta, v_{\text{in}}, v_F)$ with some vertex set (also called *state space* in the context of synthesis) V, some action set Σ_0 for player 0, some action set Σ_1 for player 1, some total edge function $\delta : V \times \Sigma_0 \times \Sigma_1 \to V$, some initial vertex v_{in} and some final vertex v_F. We require that v_F is *absorbing*, i.e., for all $x \in \Sigma_0 \times \Sigma_1$, $\delta(v_F, x) = v_F$. A *decision sequence* is an infinite sequence $\rho = \rho_0 \rho_0' \rho_1 \rho_1' \ldots$ such that for all $j \in \mathbb{N}_0$, $\rho_j \in \Sigma_0$ and $\rho_j' \in \Sigma_1$. Such a decision sequence induces an infinite *play* $\pi = \pi_0 \pi_1 \ldots$ in $\mathcal{G}$ such that $\pi_0 = v_{in}$ and for all $j \in \mathbb{N}_0$, we have $\delta(\pi_j, \rho_j, \rho_j') = \pi_{j+1}$. We call plays winning for player 1 (the *system player*) if there does not exist some $j \in \mathbb{N}$ such that $\pi_j = v_F$. For the scope of this paper, we also need *reachability games*; in these, player 1 wins a play if there exists some $j \in \mathbb{N}$ such that $\pi_j = v_F$.

Safety games are *memoryless determined*, i.e., if and only if player 1 wins the game, there exists some function $f : V \times \Sigma_0 \to \Sigma_1$ such that for all decision sequences $\rho = \rho_0 \rho_0' \rho_1 \rho_1' \ldots$ with corresponding plays $\pi = \pi_0 \pi_1 \ldots$, if $\rho_j' = f(\pi_j, \rho_j)$ for all $j \in \mathbb{N}_0$, then π is winning for player 1. The situation for player 0 is dual.

Given some bound $b \in \mathbb{N}$, some input and output alphabets Σ_0 / Σ_1 and some universal co-Büchi word automaton $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ with $\Sigma = \Sigma_0 \times \Sigma_1$, the corresponding (classical) *synthesis game* is defined as $\mathcal{G} = (V, \Sigma_0, \Sigma_1, \delta, v_{\text{in}}, v_F)$ with a vertex set V comprising all functions mapping the states in Q onto $\{\bot, 0, 1, \ldots, b\}$. The vertices of the game encode in which states of $\mathcal{A}$ a run of the automaton corresponding to the input/output played by the players so far could be. All such states have a numeral value assigned, whereas the others are mapped to $\bot$. The numeral value represents how many rejecting states have been visited at most along such a run so far (the so-called *counters*). For details of this approach, the reader is referred to [18].

We have defined safety games in a way such that we can efficiently extract a Mealy automaton $\mathcal{M}$ satisfying $\mathcal{A}$ from a winning strategy f. We define the *winning region* of $\mathcal{G}$ to be the largest subset of vertices in V such that for setting v_{in} to any of these, the game is winning for player 1.

Binary decision diagrams: For representing sets of vertices and the transition relation in safety games symbolically, we use *reduced ordered binary decision diagrams* (BDDs) [4, 5], which represent characteristic functions $f : 2^V \to \mathbb{B}$

for some finite set of variables V. Since they are well-established in the context of formal verification, we do not describe their details here but rather treat them on an abstract level and state the operations on them that we use. For a comprehensive overview, see [5]. Given two BDDs f and f', we define their conjunction and disjunction as $(f \wedge f')(x) = f(x) \wedge f'(x)$ and $(f \vee f')(x) = f(x) \vee f'(x)$ for all $x \subseteq V$. The negation of a BDD is defined similarly. Given some set of variables $V' \subseteq V$ and a BDD f, we define $\exists V'.f$ as a function that maps all $x \subseteq V$ to **true** for which there exists some $x' \subseteq V'$ such that $f(x' \cup (x \setminus V')) =$ **true**. Dually, we define $\forall V'.f \equiv \neg(\exists V'.\neg f)$. Given two ordered lists of variables $L = l_1, \ldots, l_n$ and $L' = l'_1, \ldots l'_n$ of the same length, we furthermore denote by $f[L/L']$ the BDD for which some $x \subseteq V$ is mapped to true if and only if $f(x \setminus \{l'_1, \ldots, l'_n\} \cup \{l_i \mid \exists 1 \leq i \leq n : l'_i \in x\}) =$ **true**.

2.1 Differences to Other Works

In contrast to previous works on Safraless synthesis, we give a simplified presentation here, which relies on universal co-Büchi word automata (UCW) instead of co-Büchi tree automata [14, 18] or transition-based UCWs [8].

Furthermore, the definition of safety games differs from the one used when synthesizing Moore automata. First of all, we assume that player 0 (the environment) does the first move instead of player 1 (the system player). This way, the game model corresponds to the behaviour of Mealy automata. This slightly changes the semantics of the LTL formulas for synthesis. For example, the specification $G(r \leftrightarrow g)$ for the input atomic proposition (AP) set $\{r\}$ and the output AP set $\{g\}$ is realisable, whereas for the reversed order of input and output used in previous works, it is unrealisable. The intuition of this change is that this reduces the number of next-time LTL operators necessary for practical specifications, thus reducing the size of the UCW for the specification and the synthesis time needed in total. Nevertheless, the techniques presented in this paper are equally applicable to Moore automata synthesis.

Additionally, the fact that we do not have vertex sets for both players 0 and 1 in the game allows us to simplify the game solving process and also saves bits for the state sets in a symbolic game solving process. Given a safety game $\mathcal{G} = (V, \Sigma_0, \Sigma_1, \delta, v_{in}, v_F)$, we build a BDD B^δ corresponding to δ over the four lists of variables $\{pre, in, out, post\}$ such that for all $q, q' \in Q$, $i \in \Sigma_1$ and $o \in \Sigma_2$, by abuse of notation, $B^\delta(q, i, o, q') =$ **true** if and only if $\delta(q, i, o) = q'$ (for some encoding of the states, inputs and outputs into the BDD variables). Using B^δ and some BDD B^F over the variables in *post* mapping only v_F to **true**, we can compute the winning region of $\mathcal{G}$ as $\nu X.X \wedge (\forall in.\exists out, post.(B^\delta \wedge X[post/pre] \wedge (\neg B^F)))$ for ν denoting the greatest fixed point operator.

3 Safety and Non-safety: Splitting the Specification

In this section, we explain how to decompose an LTL specification being subject to synthesis in a way such that non-safety and safety properties can be treated in

parallel. Recall that we assume that the specification is written in the form $\psi = \bigwedge_{a \in A} a \rightarrow \bigwedge_{g \in G} g$. In the classical bounded synthesis approach, ψ is transformed to a UCW which in turn is converted to its induced safety game for some given bound. Here, we propose a slightly different approach. Instead of building one single game from the specification, we split the latter into parts, build individual games for each of the parts and then take their parallel composition to obtain a *composite game*. This has several advantages:

1. It has been observed [8] that the time to compute a UCW from an LTL formula is a significant part of the overall realisability checking time. By splitting the specification beforehand, building a monolithic UCW is avoided, resulting in a lower total computation time.
2. Taking the parallel composition of multiple game structures can be done in a relatively efficient way when using BDDs for solving the composite game.
3. The state spaces of games corresponding to safety properties do not need the counters that are employed in the bounded synthesis approach. Thus, by decomposing the specification into safety and non-safety parts, we can save counters, which in turn reduces the computation time further.

In order to obtain a valid decomposition scheme, the resulting game must be winning for player 1 (the system player) in the same cases as before, i.e., if and only if either a safety or non-safety assumption is violated or all guarantees are fulfilled. The technique presented in the following does not preserve the smallest bound b such that the specification is fulfillable (as the bound depends on the syntactic structure of the UCW). However, the method proposed is still sound and complete, i.e., if and only if there exists a bound b such that the safety game induced by the UCW for the overall specification and b is winning for player 1, there exists some bound for the non-safety part of the specification and the technique presented in this section such that the resulting game is winning for player 1.

In [20], the authors propose a method to solve a generalised parity game for a specification of the form $\bigwedge_{a \in A} a \rightarrow \bigwedge_{g \in G} g$ as stated above successively. They first build games for the safety assumptions and guarantees, strip the non-winning parts (for the system player) from them and compose them with games for the remaining parts of the specification. For completeness of this methodology, the non-safety assumptions however must not have any effect on the fulfillability of the safety guarantees. In general, we cannot assume this; we thus propose a different method here that is based on introducing some kind of *signal* into the game that links the safety guarantees and the non-safety part of the specification.

We start by splitting the specification $\psi = \bigwedge_{a \in A} a \rightarrow \bigwedge_{g \in G} g$ into four sets of LTL formulas: the safety assumptions A_s, the safety guarantees G_s, the non-safety assumptions A_n, and the non-safety guarantees G_n. Then, we build a reachability game $\mathcal{G}_1$ for the safety assumptions that is won by player 1 if some assumption in A_s is violated. For the next step, we add one bit to the output atomic proposition set of the system to be synthesized; let its name be safe_g. We build a safety game $\mathcal{G}_2$ from the safety guarantees G_s that is won

by player 1 if safe_g always represents whether one of the safety guarantees has already been violated. For the non-safety part, we take the *modified specification* $\psi' = (\bigwedge_{a \in A_n} a) \rightarrow (\bigwedge_{g \in G_n} g \wedge G(\text{safe}_g))$ and convert it to a UCW $\mathcal{A}$. Given a bound $b \in \mathbb{N}$ and having prepared $\mathcal{G}_1$, $\mathcal{G}_2$ and $\mathcal{A}$, we can now build the composite game $\mathcal{G}$:

1. We take $\mathcal{A}$ and b and build the corresponding bounded synthesis safety game. Let its name be $\mathcal{G}_3$.
2. We define the overall synthesis game $\mathcal{G}$ as the parallel composition of $\mathcal{G}_1$, $\mathcal{G}_2$ and $\mathcal{G}_3$, i.e., the vertex set is the product of the individual vertex sets and the transition relation is defined such that the games $\mathcal{G}_1$, $\mathcal{G}_2$ and $\mathcal{G}_3$ are played in parallel (over the same inputs and outputs). We say that $\mathcal{G}_1$, $\mathcal{G}_2$ and $\mathcal{G}_3$ are *components* of $\mathcal{G}$.
3. Let q_F^1, q_F^2 and q_F^3 be the final vertices of the games $\mathcal{G}_1$, $\mathcal{G}_2$ and $\mathcal{G}_3$, respectively. We define a play π in $\mathcal{G}$ to be winning for the system player if either q_F^1 is visited at some point on π or q_F^2 and q_F^3 are never visited.

We obtain the following result:

Theorem 1. *For every LTL specification* $\psi = \bigwedge_{a \in A} a \rightarrow \bigwedge_{g \in G} g$*, there exists some bound* $b \in \mathbb{N}$ *such that the composite game* $\mathcal{G}$ *built from* ψ *and* b *as defined above is won by the system player* 1 *if and only if there exist some bound* $b' \in \mathbb{N}$ *such that the (classical) safety synthesis game induced by the UCW corresponding to* ψ *and* b' *is winning for player* 1*.*

Proof. By examining the possible causes for winning/losing the synthesis games, the correctness of the claim can easily be seen. □

Let B_1^F be a BDD over the set of variables *pre* representing the final vertices of the game $\mathcal{G}_1$ and B_2^F and B_3^F be BDDs over *post* for the final vertices of $\mathcal{G}_2$ and $\mathcal{G}_3$, respectively. For B^δ being the BDD representing the transition relation of $\mathcal{G}$, we can obtain the winning region of player 1 by computing (for μ denoting the least fixed point operator):

$$V = \mu Y.Y \vee B_1^F \vee (\forall in.\exists out, post.B^\delta \wedge Y[post/pre])$$
$$W = \nu X.V \vee (X \wedge (\forall in.\exists out, post.B^\delta \wedge X[post/pre] \wedge (\neg B_2^F) \wedge (\neg B_3^F)))$$

In these equations, V represents the states that are winning due to the fact that the system player can choose a sequence of decisions such that some safety assumption is not fulfilled; W is the winning region for player 1 in $\mathcal{G}$.

We can simplify the computation by taking $V = B_1^F$, making the composite game essentially a safety game. To see this, consider a state in the game in which some guarantee has just been violated but that is still winning as from that state onwards, the system player can force the other player into a state in which also some safety assumption is violated. As the game is finite, there is an upper bound of k steps for some $k \in \mathbb{N}$ on the length of such a bridging path in the game. By increasing the bound used for building $\mathcal{G}_3$ by k, it can be made sure that q_1^F is reached before q_3^F is visited, making the game also winning with

the modified definition for V. We use this simplification for our implementation to be described in Section 6 as it facilitates the extraction of winning strategies from the game.

4 Encoding Bounded Synthesis in BDDs

The efficiency of solving games using BDDs heavily depends on a smart encoding of the state space into the BDD bits. As already stated, for a symbolic solution of a safety game, four groups of BDD variables are needed: two groups for the game vertices (*pre* and *post*), one for the input to the system and one for the output. As we defined the input as $I = 2^{\mathrm{AP}_I}$ and the output as $O = 2^{\mathrm{AP}_O}$ for the scope of this paper, a straight-forward boolean encoding of I and O for usage in the BDDs exists: we allocate one BDD bit for each element of AP_I and AP_O. It remains to find a suitable encoding for the state space of the game.

First of all, if the state space is the product of some smaller state spaces, we can parallelise the problem; for example, if $V = V_1 \times V_2 \times \ldots \times V_m$ for some $m \in \mathbb{N}$, we can find good encodings for each of the state spaces $V_1, \ldots, V_m$ individually. We are thus able to handle the state space encodings of $\mathcal{G}_1$, $\mathcal{G}_2$ and $\mathcal{G}_3$ (as defined in the previous section) separately.

4.1 The Non-safety Part

Recall that in the context of bounded synthesis, the safety game induced by a UCW for a given bound b has a certain property: the state space consists of all functions mapping the states of the UCW onto $\{\bot, 0, 1, \ldots, b\}$ for b being the chosen bound. For each state, we can encode the value the function maps to individually. For the scope of this paper, we define the following encoding for this counter set $\{\bot, 0, 1, \ldots, b\}$: we use $\lceil \log_2(b+1) \rceil + 1$ bits. One bit is used for representing whether the value equals $\bot$, the remaining bits represent the standard binary encoding of the numeral (if given). Taking an extra bit for the $\bot$ value has the advantage of obtaining smaller BDDs in most cases as this value appears very often in the definition of the transition relation.

We also use and propose two additional tricks. First of all, the games defined in the previous section are built in a way such that they permit one type of non-determinism: we can allow the system player to choose a successor state from a set of possible ones. If the system player can do this in a greedy way, i.e., the non-determinism can be resolved after each input/output cycle without losing completeness, the game semantics remain unchanged. For bounded synthesis, we can thus relax the transition relation slightly: we allow the system player to increase her counters in addition to the counter increases imposed by visits to rejecting states. We also allow her to set some counters from $\bot$ to some arbitrary other value. This *non-minimality* [1] of the transition relation typically decreases the size of its symbolic encoding. A similar idea was also pursued by Henzinger et al. [10] for simplifying the process of automaton determinisation.

As a second trick, we can use some automata-theoretic argument for not having to store counters for certain states. Let a *strongly connected component* (SCC) in a UCW be a maximal set of states such that there exist sequences of transitions between all pairs of states in the SCC. It is well-known that every infinite run of a UCW $\mathcal{A}$ enters a strongly connected component in $\mathcal{A}$ after a finite number of steps that it never leaves again. It is accepting if and only if in this last SCC, rejecting states are visited only finitely often. This fact gives rise to an optimisation idea: for transient states or states in SCCs without rejecting states, we do not really need counters: we can assume that the counter corresponding to such a state is always reset to 0. We call those states in $\mathcal{A}$ *transient* that can only be visited once on every run of the automaton. Thus, only one bit is needed for such states instead of $\lceil \log_2(b+1) \rceil + 1$ bits. This modification does not alter the soundness or completeness of the overall synthesis procedure. Additionally, as some counters are now reset on some transitions, in practice we often have the situation that for realisable specifications, the number of counter bits per remaining state necessary for finding out that the specification is realisable is also less.

4.2 The Safety Part

For the encoding of the game components corresponding to safety assumptions and guarantees, we state two different, straight-forward methods, which we explain in the following. The first method only works for *locally checkable properties* and is usually more efficient than the second one in this case, whereas the latter method is capable of handling arbitrary safety properties.

Smart encoding of locally checkable properties: If an LTL property is of the form $\psi = G(\phi)$ with a formula ϕ in which the only temporal operator occurring is X, then ψ is a *locally checkable property* [12]. Let k be the deepest nesting of the X operator in ϕ. For checking the satisfaction of such a property along a trace, it suffices to store whether the property has already been violated, the last k input/outputs (also called *history*) and the current round number (with the domain $\{0, 1, \ldots, k-1, \geq k\}$). Then, in every round with a number $\geq k$, we update whether the specification is already falsified with the input and output in the last k rounds and the current round. For encoding the round number in a symbolic way, we use a binary representation.

Encoding such a property in this way has some advantages: First of all, the encoding proposed is canonical. Furthermore, multiple properties can share the information stored in the game state space this way, so we can recycle the stored information for all such locally checkable safety properties. Note that it is possible to reduce the number of bits necessary for storage by leaving out the history bits not needed for checking the given properties.

The general method: Safety properties have equivalent *syntactically safe* UCW, i.e., in the UCW, all rejecting states are absorbing. In this case, the UCW can be determinised by the power set construction. Thus, we can assign to each state in the universal automaton a state bit which is set to 1 whenever

there is a run from the initial state to the respective state encoded by the bit for the input/output played by the players during the game so far.

This method is applicable to all safety properties but requires the computation of a universal co-Büchi automaton having the property stated above. While it has been observed that checking if a property is safety is not harder than building an equivalent universal co-Büchi automaton [13], it is not guaranteed that typical procedures for constructing UCWs from LTL properties yield automata that have this property. For conciseness, we use a simplified approach in our actual implementation. If the procedure employed for converting an LTL formula into a UCW yields a UCW for which all rejecting states are absorbing or transient, we declare the property as being safety and otherwise treat it as a non-safety property. While we may miss safety properties this way, the soundness of the overall approach is preserved.

5 Checking Unrealisability

So far, we have only dealt with the case that we want to prove realisability of a specification. If a specification is unrealisable, then for no bound $b \in \mathbb{N}$, the safety game induced by the bound and the specification is won for the system player. Thus, an implementation of our approach, which would typically increase the bound successively until the induced safety game is winning for the system player, does not terminate in this case. In [8], it is described how the bounded synthesis approach can be used for detecting unrealisability quickly anyway: we simply run the synthesis procedure both on the original specification as well as on the negated specification with swapped input and output in parallel. One of these runs is guaranteed to terminate. Whenever this happens, we can abort the other run. This results in an decision procedure for the overall problem.

When applying the optimisations from this paper, this idea is not directly usable, as when negating the specification, the result is not again of the form $\bigwedge_{a \in A} a \rightarrow \bigwedge_{g \in G} g$ for some sets of assumptions A and guarantees G. Instead, checking if the environment player wins can be done by swapping input and output, negating only the modified specification, and making the final states of $\mathcal{G}_1$ losing for player 1 instead of winning. Then, player 1 (which is now the environment player) wins only if the safety assumptions are fulfilled, the safe_g bit always represents if a safety guarantee has already been violated, and the negated modified specification is fulfilled (with respect to the given bound).

Using the notations from Section 3, after replacing $\mathcal{G}_3$ with a game corresponding to the negated modified specification, we can compute the set of winning states for the environment player by:

$$W = \nu X.X \wedge (\neg B_1^F) \wedge (\exists in. \forall out. \exists post. B^\delta \wedge X[post/pre] \wedge (\neg B_2^F) \wedge (\neg B_3^F))$$

6 Experimental Results

We implemented our symbolic bounded synthesis approach in C++ with the BDD library CUDD v.2.4.2 [21], using dynamic variable reordering. The pro-

totype tool assumes that the individual guarantees and assumptions are given separately. The first step in the computation is to split non-safety properties from safety ones. For this, the tool calls the LTL-to-Büchi converter LTL2BA v.1.1 [9] on the negations of the properties to obtain equivalent universal co-Büchi word automata. As described in Section 4.2, we then check if the automata obtained are syntactically safe. Locally checkable properties are converted to games using the procedure specialised in this case, all other safety properties are treated by the general procedure given. The UCW corresponding to the modified non-safety part of the specification (as described in Section 3) is again computed by calling LTL2BA on it. The last step for realisability checking is to solve the composite games built for a successively increasing number of counter bits per state in the UCW until the game is winning for the system player. We always start with two bits.

We always check for realisability and unrealisabilty of the given specification simultaneously, as described in the previous section. In case of realisability, we extract an implementation that fulfills the specification. We do this in a fully symbolic way: the first step is to compute the winning region of the game and identify state bits that have a fixed value throughout all winning plays. These state bits are removed. Then, we restrict the transition relation to moves by the system player for which the lexicographically minimal next winning state is chosen (for some order of the state bits). We do the same for the output bits, i.e., for some order of the output bits, we restrict the resulting transition relation to lexicographically minimal output bit valuations (with respect to the remaining choices for the system player). As a result, the transition relation is weakened in a way such that there is precisely one combination of next state and output bit valuation left for every reachable state and input variable assignment, making the behaviour of player 1 deterministic. The remaining game graph is, together with the specification, converted to a NuSMV [6] model. This allows running NuSMV to verify the correctness of the models produced.

All computation times given in the following are obtained on a Sun XFire computer with 2.6 Ghz AMD Opteron processors running an x64-version of Linux. All tools considered are single-threaded. We restricted the memory usage to 2 GB and set a timeout of 3600 seconds. The running times for our tool always include the computation times of LTL2BA.

6.1 Performance Comparison on the Examples from [11, 8]

We compare our prototype implementation with the only other currently publicly available tools for full LTL synthesis, namely Lily v.1.0.2 [11] and Acacia v.0.3.9 [8]. In the following, for Acacia as well as our prototype tool, we only give running times for the non-realisability check if the property is not realisable and the realisability check and model synthesis if the property is realisable.

The 23 mutex variations used as examples in [11, 8] are a natural starting point for our investigation. For usage with our tool, we adapted these examples to the Mealy-type computation model used in this work (as described in Section 2.1) by prefixing all references to input variables with a next-time operator. For

Table 1. Comparison of the running times of Acacia and our prototype tool (in seconds) on the scalable example no. 3 from [8]. In this table, we denote timeouts by "t/o" and running out of memory by "m/o". As Lily performs worse than Acacia on this benchmark, we did not include Lily in this comparison.

# of Clients:	1	2	3	4	5	6	7	10	14	15	20	21	22
Acacia running times:	0.9	2.0	4.0	9.8	47.3	506.5	m/o	m/o	m/o	m/o	m/o	m/o	m/o
Prototype running times:	0.3	0.7	0.6	1.9	0.9	4.6	3.0	651.5	491.0	t/o	1909.0	t/o	t/o

these 23 examples, Lily needed 54.35 seconds of computation time (of which 44.25 seconds were devoted to computing the automata from the given specifications). Acacia in turn finished the task in 53.71 seconds (including 42.2 seconds for building the automata). Our prototype implementation had a total running time of about 19.41 seconds. As computing the automata from the specification parts is not a pure preprocessing step in our prototype, we do not split up the total running time here.

In [8], the authors also modify one of these examples in order to be scalable. Table 1 contains the respective results for this example.

6.2 A Load Balancing System

For evaluating the techniques presented in this paper in a more practical context, we present an example concerning a *load balancing unit* distributing requests to a fixed number of servers. Such a unit typically occurs as a component of a bigger system which in turn utilises it for scheduling internal requests. We demonstrate how a synthesis procedure can be used in the early development process of the bigger system in order to systematically engineer the requirements of the load balancer. Using a synthesis tool in this context makes it possible detect errors in the specification that result in unrealisability as early as possible. We start by stating the fundamental properties of the load balancing system and finally tune it towards serving requests to the first server in a prioritised way. After each added specification/assumption, we run our example implementation in order to check if the specification is still realisable.

The following list contains the parts of the specification. Table 2 gives the running times of our tool and Acacia for the respective sets of assumptions and guarantees and some numbers of clients $n \in \{2, \ldots, 9\}$. The system to be synthesized uses the input bits $r_0, \ldots, r_{n-1}$ for receiving the information whether some server is sufficiently under-utilised to accommodate another task and the output bits $g_0, \ldots, g_{n-1}$ for the task assignments. An additional input *job* reports on an incoming job to be assigned. For usage with Acacia, all occurrences of output variables in the specification have been prefixed with a next-time operator to take into account the different underlying computation model.

1. *Guarantee:* Non-ready servers are never bothered: $\bigwedge_{0 \leq i < n} G(g_i \rightarrow r_i)$
2. *Guarantee:* A task is only assigned to one server: $\bigwedge_{0 \leq i < n} G(g_i \rightarrow (\bigwedge_{j \in \{1,\ldots,n\} \setminus \{i\}} \neg g_j))$

3. *Guarantee:* Every server is used infinitely often: $\bigwedge_{0 \leq i < n} GF(g_i)$

Note that the guarantees 1,2 and 3 cannot be fulfilled at the same time as some server might not report when it is ready. Therefore, we replace the third part of the specification and continue:

4. *Guarantee:* Liveness of the system: $\bigwedge_{0 \leq i < n} GF(r_i) \rightarrow GF(g_i)$
5. *Guarantee:* Only jobs that actually exist are assigned: $G((\bigvee_{0 \leq i < n} g_i) \rightarrow job)$.

Again, the guarantees 1, 2, 4 and 5 are unrealisable in conjunction as the *job* signal might never be given. We add the assumption that this is not the case:

6. *Assumption:* There are always incoming jobs: $GFjob$

At this point, the system designer gets to know that this added requirement does not fix the unrealisability problem, either. The reason is that the clock cycles in which *job* is set and the cycles in which some server is ready might occur in an interleaved way. We therefore add:

7. *Assumption:* The job signal stays set until the job has been assigned: $G(job \wedge (\bigwedge_{0 \leq i < n} \neg g_i) \rightarrow X(job))$

Note that the specification is still not realisable. The reason is that the ready signal of one server i might always be given after a job assignment to another server j has been given (for some $i \neq j$). If server i then always immediately withdraws its ready signal, the controller can never schedule a job to server i, contradicting guarantee 4 if both servers i and j are ready infinitely often. We therefore modify guarantee 4 to not consider these cases:

8. *Guarantee:* Every ready signal is either withdrawn or eventually handled: $\bigwedge_{0 \leq i < n} \neg(FG(r_i \wedge \neg g_i))$

We continue by adding a priority to the first server. Note that this breaks realisability again, as server 0 can block the others. As an example, we solve this problem by adding the assumption that server 0 works sufficiently long after it obtains a new job before signalling ready again.

9. *Guarantee*: Server 0 gets a job whenever a job is given and it is ready: $G((\bigvee_{1 \leq i < n} g_i) \rightarrow \neg r_0)$
10. *Assertion*: Server 0 does not report being ready when it gets a task until after an incoming job has been reported on for the next time: $G(g_0 \rightarrow ((\neg job \wedge \neg r_0)\ U\ (job \wedge \neg r_0)))$.

7 Conclusion & Outlook

In this paper, we described the steps necessary to make the bounded synthesis approach work well with symbolic data structures such as BDDs. The key requirement was to reduce the number of counters in the safety games that occur

Table 2. Running times of Acacia ("A") and our prototype tool ("P") for the subproblems defined in Section 6.2 for $n \in \{2, \ldots, 9\}$. For each combination of assumptions and guarantees, it is reported whether the specification was satisfiable (+/-), how many counter bits per state in the UCW were involved at the end of the computation (only for our prototype tool) and how long the computation took (in seconds). We left out the Lily tool as it is not competitive on the load balancing example.

Tool	Specification / # Clients	2	3	4	5	6	7	8	9
P	1	+ 2 0.6	+ 2 0.6	+ 2 0.2	+ 2 1.3	+ 2 0.2	+ 2 0.3	+ 2 0.2	+ 2 0.3
A		+ 0.3	+ 0.4	+ 0.6	+ 0.9	+ 1.5	+ 2.7	+ 5.3	+ 12.1
P	$1 \wedge 2$	+ 2 0.4	+ 2 0.3	+ 2 0.6	+ 2 0.6	+ 2 0.7	+ 2 0.6	+ 2 0.6	+ 2 0.7
A		+ 0.3	+ 0.3	+ 0.4	+ 0.4	+ 0.6	+ 0.9	+ 1.6	+ 3.1
P	$1 \wedge 2 \wedge 3$	- 2 0.5	- 2 0.5	- 2 0.5	- 2 0.5	- 2 0.7	- 2 1.0	- 2 6.9	- 2 73.9
A		- 19.2	- 475.6	timeout	timeout	timeout	timeout	timeout	timeout
P	$1 \wedge 2 \wedge 4$	+ 2 0.3	+ 3 0.4	+ 3 0.9	+ 4 65.5	+ 4 104.6	+ 4 990.3	timeout	timeout
A		+ 0.6	+ 1.3	+ 8.7	+ 277.9	timeout	timeout	timeout	timeout
P	$1 \wedge 2 \wedge 4 \wedge 5$	- 2 0.2	- 2 0.7	timeout	timeout	timeout	timeout	timeout	timeout
A		- 163.4	timeout	timeout	timeout	timeout	timeout	timeout	timeout
P	$6 \rightarrow 1 \wedge 2 \wedge 4 \wedge 5$	- 2 0.2	- 2 0.7	- 2 3244.1	timeout	timeout	timeout	timeout	timeout
A		- 175.3	timeout	timeout	timeout	timeout	timeout	timeout	timeout
P	$6 \wedge 7 \rightarrow 1 \wedge 2 \wedge 4 \wedge 5$	- 2 0.5	- 2 1.1	timeout	timeout	timeout	timeout	timeout	timeout
A		- 190.7	timeout	timeout	timeout	timeout	timeout	timeout	timeout
P	$6 \wedge 7 \rightarrow 1 \wedge 2 \wedge 5 \wedge 8$	+ 2 0.3	+ 3 0.6	+ 3 2.4	+ 4 20.7	+ 4 368.6	timeout	timeout	timeout
A		+ 7.5	+ 69.0	+ 357.4	timeout	timeout	timeout	timeout	timeout
P	$6 \wedge 7 \rightarrow 1 \wedge 2 \wedge 5 \wedge 8 \wedge 9$	- 2 0.3	- 2 0.2	- 2 0.3	- 2 1.0	- 2 16.8	- 2 449.1	timeout	timeout
A		- 48.8	- 2133.5	timeout	timeout	timeout	timeout	timeout	timeout
P	$6 \wedge 7 \wedge 10 \rightarrow 1 \wedge 2 \wedge 5 \wedge 8 \wedge 9$	+ 2 0.4	+ 2 0.8	+ 3 118.7	timeout	timeout	timeout	timeout	timeout
A		+ 26.9	+ 295.8	timeout	timeout	timeout	timeout	timeout	timeout

in this approach as much as possible. We performed this task by splitting the specification into safety and non-safety parts and presented an additional trick that allowed stripping some counters from the game component corresponding to the non-safety specification conjuncts. We also discussed efficient encodings of the safety part of the specification into games. Experimental results show a huge speed-up compared to previous works.

One particular issue we did not address in this paper is the extraction of small implementations in the synthesis process for the case that the specification is realisable. Similarly to the observations made in the context of generalised reactivity(1) synthesis, where the expressivity of full LTL is traded against the possibility to use more efficient algorithms for performing the synthesis process, the models produced are often non-optimal [2], i.e., unnecessarily large. Thus, further work will deal with the more effective extraction of winning strategies. While the techniques presented here are already suitable for requirements engineering and prototype extraction, the problem of how to obtain small implementations which can directly be converted to suitable hardware circuits is still open.

References

1. Bloem, R., A.Cimatti, Pill, I., Roveri, M.: Symbolic implementation of alternating automata. International Journal of Foundations of Computer Science **18**(4) (2007)

727–743
2. Bloem, R., Galler, S., Jobstmann, B., Piterman, N., Pnueli, A., Weiglhofer, M.: Specify, compile, run: Hardware from PSL. Electr. Notes Theor. Comput. Sci. **190**(4) (2007) 3–16
3. Bozga, M., Maler, O., Pnueli, A., Yovine, S.: Some progress in the symbolic verification of timed automata. In Grumberg, O., ed.: CAV. Volume 1254 of LNCS., Springer (1997) 179–190
4. Bryant, R.E.: Graph-based algorithms for boolean function manipulation. IEEE Trans. Computers **35**(8) (1986) 677–691
5. 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
6. 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: CAV. Volume 2404 of LNCS., Springer (2002) 359–364
7. Clarke, E.M., Grumberg, O., Peled, D.: Model Checking. MIT Press (1999)
8. Filiot, E., Jin, N., Raskin, J.F.: An antichain algorithm for LTL realizability. In: CAV. Volume 5643 of LNCS., Springer (2009) 263–277
9. Gastin, P., Oddoux, D.: Fast LTL to Büchi automata translation. In: CAV. Volume 2102 of LNCS., Springer (2001) 53–65
10. Henzinger, T.A., Piterman, N.: Solving games without determinization. In Ésik, Z., ed.: CSL. Volume 4207 of LNCS., Springer (2006) 395–410
11. Jobstmann, B., Bloem, R.: Optimizations for LTL synthesis. In: FMCAD, IEEE Computer Society (2006) 117–124
12. Kupferman, O., Lustig, Y., Vardi, M.: On locally checkable properties. In: Logic for Programming, Artificial Intelligence, and Reasoning. (2006) 302–316
13. Kupferman, O., Vardi, M.Y.: Model checking of safety properties. In Halbwachs, N., Peled, D., eds.: CAV. Volume 1633 of LNCS., Springer (1999) 172–183
14. Kupferman, O., Vardi, M.Y.: Safraless decision procedures. In: FOCS, IEEE (2005) 531–542
15. McMillan, K.L.: Symbolic Model Checking. Kluwer Academic Publishers (1993)
16. Müller, S.M., Paul, W.J.: Computer architecture: complexity and correctness. Springer (2000)
17. Pnueli, A., Rosner, R.: On the synthesis of an asynchronous reactive module. In Ausiello, G., Dezani-Ciancaglini, M., Rocca, S.R.D., eds.: ICALP. Volume 372 of LNCS., Springer (1989) 652–671
18. Schewe, S., Finkbeiner, B.: Bounded synthesis. In Namjoshi, K.S., Yoneda, T., Higashino, T., Okamura, Y., eds.: ATVA. Volume 4762 of LNCS., Springer (2007) 474–488
19. Schneider, K., Logothetis, G.: Abstraction of systems with counters for symbolic model checking. In Mutz, M., Lange, N., eds.: Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen, Braunschweig, Germany, Shaker (1999) 31–40
20. Sohail, S., Somenzi, F.: Safety first: A two-stage algorithm for LTL games. In: FMCAD, IEEE Computer Society (2009) 77–84
21. Somenzi, F.: CUDD: CU decision diagram package, release 2.4.2 (2009)
22. Wegener, I.: Branching Programs and Binary Decision Diagrams. SIAM (2000)

A Notes on the terminology in this paper

Liveness and safety properties: In this paper, we use the term *non-safety property* rather than liveness property. This is to avoid confusion as the latter is often considered to be equivalent with the term *pure liveness properties*. Our non-safety properties can however have impact on the finitary behaviour of a system under consideration. Consider for example the property $\psi = G(a\ U\ b) \vee G(a \leftrightarrow \neg b)$ over a set of variables $\{a, b\}$. While ψ is not a safety property, it does not allow extending the prefix word $\{\}\{a, b\}\{a, b\}\ldots$ in a way that it satisfies ψ.

As many readers directly skip to sections of their interest and might thus miss notes on the usage of terms (especially in the preliminaries), we preferred not to use the term "liveness properties" in our work, even though "non-safety" sounds less catchy.

Extracting small strategies: In this paper, we focussed on obtaining results from the synthesis process as fast as possible. Consequently, the strategies extracted are not necessarily minimal in size. Indeed, as the problem of obtaining minimal strategies is even NP-complete for the one-player case[1], for the usage cases considered in this paper (feasibility check and prototype extraction), spending too much time on obtaining small results seems to be overkill. Consequently, as also observed in [2] in the context of generalised reactivity synthesis, the resulting strategies are often much larger than necessary. We leave the problem of finding suitable heuristics for strategy reduction open.

B Notes on the experimental evaluation:

The impact of the LTL-to-Büchi tool on our experimental evaluation: The tools "Lily" and "Acacia" we compare against in our experimental evaluation both use "Wring" as the tool to convert LTL formulas into non-deterministic Büchi automata (or dually, into universal co-Büchi automata). Wring produces Büchi automata in which the states are labelled with the input/output in the last round of the run rather than the edges. This disallowed us from using Wring as the LTL-to-Büchi tool for our synthesis tool as well: the optimisations presented in this paper are geared towards reducing the state space representation of the synthesis game as much as possible. Introducing the last input/output as needed in such a case into the state space would contradict this idea; thus, we use a tool for creating Büchi automata in which the edges are labelled with the last input/output.

[1] Obtaining a minimal positional strategy in a safety game has been proven to be NP-hard (see: Krishnendu Chatterjee, Luca de Alfaro, Rupak Majumdar: The Complexity of Coverage. APLAS 2008: 91-106). The fact that obtaining a Mealy automaton with a minimal number of states is NP-hard can be derived from this result. It actually even holds if we restrict the input/output width to just one bit. This is a yet unpublished result by us.

Automatic variable ordering: As already mentioned, we used automatic BDD variable-reordering in our implementation. For fairness of the comparison, we did not use a hand-crafted initial order. We only changed three of the parameters of the CUDD BDD library:

- Maximum BDD growth during sifting a variable was restricted to 10% (instead of the default value: 20%). This way, less automatic reordering is performed.
- The CUDD variable DD_MAX_CACHE_TO_SLOTS_RATIO is set to 16. This allocates more of the available memory to the cache, reducing the amount of re-computation of previous results.
- The maximum memory usage by the CUDD library was set to 3.5GB (instead of the default, 128MB). Although this is beyond the memory restriction we imposed when running the synthesis tools (namely, 2GB), we observed that due to intensive automatic reordering, the timeout of 3600 seconds was always reached before the memory was exceeded. Thus, our prototype tool never ran out of memory in our experimental evaluation.

Complete results for the load balancing example: Due to space restrictions, we could not give benchmarking results for the load balancing example for all tools considered in the paper. For completeness, they are given here in the appendix.

Table 3 shows the complete results for our prototype tool, Table 4 does the same for the Acacia tool. Finally, 5 contains running times for the tool Lily.

Table 3. Complete results (running times) of our prototype tool on the load balancing benchmark.

Setting / # Clients	**2**			**3**			**4**			**5**			**6**			**7**			**8**			**9**		
1	+	2	0.6	+	2	0.6	+	2	0.2	+	2	1.3	+	2	0.2	+	2	0.3	+	2	0.2	+	2	0.3
$1 \wedge 2$	+	2	0.4	+	2	0.3	+	2	0.6	+	2	0.6	+	2	0.7	+	2	0.6	+	2	0.6	+	2	0.7
$1 \wedge 2 \wedge 3$	-	2	0.5	-	2	0.5	-	2	0.5	-	2	0.5	-	2	0.7	-	2	1.0	-	2	6.9	-	2	73.9
$1 \wedge 2 \wedge 4$	+	2	0.3	+	3	0.4	+	3	0.9	+	4	65.5	+	4	104.6	+	4	990.3	timeout			timeout		
$1 \wedge 2 \wedge 4 \wedge 5$	-	2	0.2	-	2	0.7	timeout			timeout			timeout			timeout			timeout			timeout		
$6 \rightarrow 1 \wedge 2 \wedge 4 \wedge 5$	-	2	0.2	-	2	0.7	-	2	3244.1	timeout			timeout			timeout			timeout			timeout		
$6 \wedge 7 \rightarrow 1 \wedge 2 \wedge 4 \wedge 5$	-	2	0.5	-	2	1.1	timeout			timeout			timeout			timeout			timeout			timeout		
$6 \wedge 7 \rightarrow 1 \wedge 2 \wedge 5 \wedge 8$	+	2	0.3	+	3	0.6	+	3	2.4	+	4	20.7	+	4	368.6	timeout			timeout			timeout		
$6 \wedge 7 \rightarrow 1 \wedge 2 \wedge 5 \wedge 8 \wedge 9$	-	2	0.3	-	2	0.2	-	2	0.3	-	2	1.0	-	2	16.8	-	2	449.1	timeout			timeout		
$6 \wedge 7 \wedge 10 \rightarrow 1 \wedge 2 \wedge 5 \wedge 8 \wedge 9$	+	2	0.4	+	2	0.8	+	3	118.7	timeout			timeout			timeout			timeout			timeout		

The version of the Acacia tool used in the paper: For obtaining the benchmarks, we used the version "Acacia'10" downloaded on the 5$^{\text{th}}$ on January 2010 from its homepage. In order to be comparable to [8], we did not use the new realisability checking methods provided, but rather used the default method "0" offered by the tool.

As far as the example specifications other than the load balancing example used in the experimental evaluation section are concerned, we have taken them

Table 4. Complete results (running times) of the Acacia tool on the load balancing benchmark.

Setting / # Clients	2	3	4	5	6	7	8	9
1	+ 0.3	+ 0.4	+ 0.6	+ 0.9	+ 1.5	+ 2.7	+ 5.3	+ 12.1
$1 \wedge 2$	+ 0.3	+ 0.3	+ 0.4	+ 0.4	+ 0.6	+ 0.9	+ 1.6	+ 3.1
$1 \wedge 2 \wedge 3$	- 19.2	- 475.6	timeout	timeout	timeout	timeout	timeout	timeout
$1 \wedge 2 \wedge 4$	+ 0.6	+ 1.3	+ 8.7	+ 277.9	timeout	timeout	timeout	timeout
$1 \wedge 2 \wedge 4 \wedge 5$	- 163.4	timeout	timeout	timeout	timeout	timeout	timeout	timeout
$6 \rightarrow 1 \wedge 2 \wedge 4 \wedge 5$	- 175.3	timeout	timeout	timeout	timeout	timeout	timeout	timeout
$6 \wedge 7 \rightarrow 1 \wedge 2 \wedge 4 \wedge 5$	- 190.7	timeout	timeout	timeout	timeout	timeout	timeout	timeout
$6 \wedge 7 \rightarrow 1 \wedge 2 \wedge 5 \wedge 8$	+ 7.5	+ 69.0	+ 357.4	timeout	timeout	timeout	timeout	timeout
$6 \wedge 7 \rightarrow 1 \wedge 2 \wedge 5 \wedge 8 \wedge 9$	- 48.8	- 2133.5	timeout	timeout	timeout	timeout	timeout	timeout
$6 \wedge 7 \wedge 10 \rightarrow 1 \wedge 2 \wedge 5 \wedge 8 \wedge 9$	+ 26.9	+ 295.8	timeout	timeout	timeout	timeout	timeout	timeout

Table 5. Complete results (running times) of the Lily tool on the load balancing benchmark.

Setting / # Clients	2	3	4	5	6	7	8	9
1	+ 0.3	+ 0.7	+ 5.2	+ 93.9	+ 2296.6	timeout	timeout	timeout
$1 + 2$	+ 0.2	+ 0.5	+ 4.0	+ 84.3	+ 2542.6	timeout	timeout	timeout
$1 + 2 + 3$	- 0.4	- 1.8	- 19.4	- 294.8	timeout	timeout	timeout	timeout
$1 + 2 + 4$	+ 5.5	timeout	timeout	timeout	timeout	timeout	timeout	timeout
$1 + 2 + 4 + 5$	timeout	timeout	timeout	timeout	timeout	timeout	timeout	timeout
$6 \rightarrow 1 + 2 + 4 + 5$	timeout	timeout	timeout	timeout	timeout	timeout	timeout	timeout
$6 + 7 \rightarrow 1 + 2 + 4 + 5$	timeout	timeout	timeout	timeout	timeout	timeout	timeout	timeout
$6 + 7 \rightarrow 1 + 2 + 5 + 8$	timeout	timeout	timeout	timeout	timeout	timeout	timeout	timeout
$6 + 7 \rightarrow 1 + 2 + 5 + 8 + 9$	timeout	timeout	timeout	timeout	timeout	timeout	timeout	timeout
$6 + 7 + 10 \rightarrow 1 + 2 + 5 + 8 + 9$	timeout	timeout	timeout	timeout	timeout	timeout	timeout	timeout

from the download archive of the '09-version of the Acacia tool as some of them are no longer contained in the Acacia'10 download archive.

Complete results for demo-v3: For completeness, we also give a full table for the running times of Acacia and our prototype tool on the scalable version of demo no. 3 from [8]. As it has been shown in [8] that Acacia clearly outperforms Lily here, we left out Lily from the comparison. The results for 1–33 clients are given in Table 6. For this evaluation, we set the timeout to 2 hours. The maximum allowed memory consumption was still set to 2GB. It can be observed that the running times of our prototype tool do not strictly increase with the number of clients. A similar behaviour was also observed in [2] in the context of generalized reactivity synthesis. We conjecture that this effect might be a consequence of using a dynamic BDD variable reordering heuristic. Since it heuristically chooses variables to sift, a small change in the BDD can have large effects on the ordering and thus also on the computation time.

The examples 22.1 to 22.6 from [8]: In [8], the authors give some additional examples (named 22.1 to 22.6). For conciseness of the presentation, we did use them for the first part of the experimental evaluation.

However, we did try our implementation on them and the accumulated running time was 4.4 seconds. Acacia needed 61.36 seconds in total for them to finish. Lily in turn needed 34.05 seconds for all of them.

Table 6. The complete results for demo-v3 for our approach and Acacia. All running times are given in seconds.

# Cl.	Running times		# Cl.	Running times		# Cl.	Running times	
	Acacia	**Our ap.**		**Acacia**	**Our ap.**		**Acacia**	**Our ap.**
1	0.9	0.3	2	2.0	0.7	3	4.0	0.6
4	9.8	1.9	5	47.3	0.9	6	506.5	4.6
7	memout	3.0	8	timeout	15.3	9	memout	6.9
10	memout	651.5	11	memout	754.5	12	memout	160.9
13	memout	54.8	14	memout	491.0	15	memout	3852.2
16	timeout	3290.2	17	memout	513.4	18	memout	3352.4
19	memout	6881.6	20	memout	1909.0	21	memout	timeout
22	memout	timeout	23	memout	timeout	24	memout	timeout
25	memout	timeout	26	memout	3828.0	27	memout	timeout
28	memout	timeout	29	memout	timeout	30	memout	timeout
31	memout	timeout	32	memout	timeout	33	memout	timeout

Checking correctness of the models produced: As written in the main part of the paper, our implementation produced NuSMV models for the cases in which the specification is realisable. These were in turn usable for verifying the result of the synthesis process. We did this for the following cases:

- The 19 realisable specifications from the 23 basic benchmarks used in [8].
- The 5 realisable specifications from the modification of specification 22 given in [8].
- The scalable version of specification 3 given in [8] for the numbers of clients $\{1, 2, 3\}$.
- The models of the first two rows and the left-most column in the benchmark table for the load balancing example.

Due to the non-minimality of the obtained strategies, in other cases, the models were too large to be verified by NuSMV (NuSMV crashed or did not finish within a couple of hours). We never observed NuSMV to falsify a model.

Switching off features: For completeness, we also give benchmark results for our prototype tool and the load balancing benchmark with the optimisations from Section 3 and Section 4 switched off. Table 7 contains the results with specification splitting turned off, but non-safety counter reduction switched on. Table 8 contains the results of switching the non-safety counter reduction off, but keeping the specification splitting turned on. Finally, for Table 9, both features are switched off.

Table 7. Benchmark results for our prototype tool and the load balancing benchmark with specification splitting turned off and non-safety counter reduction switched on

Setting / # Clients	2			3			4			5			6			7			8			9		
1	+	2	0.1	+	2	0.1	+	2	0.2	+	2	0.2	+	2	0.3	+	2	0.7	+	2	2.6	+	2	8.9
$1 \wedge 2$	+	2	0.1	+	2	0.1	+	2	0.1	+	2	0.1	+	2	0.1	+	2	0.1	+	2	0.1	+	2	0.1
$1 \wedge 2 \wedge 3$	-	2	0.1	-	2	0.1	-	2	0.2	-	2	3.9	timeout			timeout			timeout			timeout		
$1 \wedge 2 \wedge 4$	+	2	0.2	+	3	0.3	+	3	0.7	+	4	5.8	+	4	55.5	timeout			timeout			timeout		
$1 \wedge 2 \wedge 4 \wedge 5$	-	2	0.1	-	2	0.3	-	2	63.8	timeout			timeout			timeout			timeout			timeout		
$6 \rightarrow 1 \wedge 2 \wedge 4 \wedge 5$	-	2	0.1	-	2	0.8	-	2	23.1	timeout			timeout			timeout			timeout			timeout		
$6 \wedge 7 \rightarrow 1 \wedge 2 \wedge 4 \wedge 5$	-	2	0.1	-	2	0.7	-	2	165.1	timeout			timeout			timeout			timeout			timeout		
$6 \wedge 7 \rightarrow 1 \wedge 2 \wedge 5 \wedge 8$	+	2	0.3	+	3	3.7	+	3	1299.3	timeout			timeout			timeout			timeout			timeout		
$6 \wedge 7 \rightarrow 1 \wedge 2 \wedge 5 \wedge 8 \wedge 9$	-	2	0.1	-	2	0.2	-	2	3.2	timeout			timeout			timeout			timeout			timeout		
$6 \wedge 7 \wedge 10 \rightarrow 1 \wedge 2 \wedge 5 \wedge 8 \wedge 9$	+	2	0.3	+	2	7.6	timeout			timeout			timeout			timeout			timeout			timeout		

Table 8. Benchmark results for our prototype tool and the load balancing benchmark with specification splitting turned on and non-safety counter reduction switched off

Setting / # Clients	2			3			4			5			6			7			8			9		
1	+	2	0.3	+	2	0.1	+	2	0.2	+	2	0.2	+	2	0.3	+	2	0.2	+	2	0.4	+	2	0.2
$1 \wedge 2$	+	2	0.1	+	2	0.1	+	2	0.1	+	2	0.2	+	2	0.1	+	2	0.2	+	2	0.1	+	2	0.2
$1 \wedge 2 \wedge 3$	-	2	0.2	-	2	0.2	-	2	0.2	-	2	0.2	-	2	0.3	-	2	1.0	-	2	7.0	-	2	73.4
$1 \wedge 2 \wedge 4$	+	2	0.2	+	3	2.7	+	3	67.6	timeout			timeout			timeout			timeout			timeout		
$1 \wedge 2 \wedge 4 \wedge 5$	-	2	0.1	-	2	8.3	timeout			timeout			timeout			timeout			timeout			timeout		
$6 \rightarrow 1 \wedge 2 \wedge 4 \wedge 5$	-	2	0.3	-	2	207.9	timeout			timeout			timeout			timeout			timeout			timeout		
$6 \wedge 7 \rightarrow 1 \wedge 2 \wedge 4 \wedge 5$	-	2	0.3	-	2	485.6	timeout			timeout			timeout			timeout			timeout			timeout		
$6 \wedge 7 \rightarrow 1 \wedge 2 \wedge 5 \wedge 8$	+	2	0.3	+	3	0.5	+	3	1.3	+	4	339.1	+	4	3596.8	timeout			timeout			timeout		
$6 \wedge 7 \rightarrow 1 \wedge 2 \wedge 5 \wedge 8 \wedge 9$	-	2	0.2	-	2	0.2	-	2	0.3	-	2	1.1	-	2	16.8	-	2	448.7	timeout			timeout		
$6 \wedge 7 \wedge 10 \rightarrow 1 \wedge 2 \wedge 5 \wedge 8 \wedge 9$	+	2	0.4	+	2	1.7	+	3	1051.5	timeout			timeout			timeout			timeout			timeout		

Table 9. Benchmark results for our prototype tool and the load balancing benchmark with specification splitting turned off and non-safety counter reduction switched off

Setting / # Clients	2			3			4			5			6			7			8			9		
1	+	2	0.1	+	2	0.2	+	2	0.1	+	2	0.2	+	2	0.3	+	2	0.7	+	2	2.7	+	2	8.2
$1 \wedge 2$	+	2	0.1	+	2	0.1	+	2	0.1	+	2	0.1	+	2	0.1	+	2	0.1	+	2	0.1	+	2	0.2
$1 \wedge 2 \wedge 3$	-	2	0.1	-	2	0.1	-	2	0.2	-	2	3.9	timeout			timeout			timeout			timeout		
$1 \wedge 2 \wedge 4$	+	2	0.3	+	3	2.6	+	3	91.1	timeout			timeout			timeout			timeout			timeout		
$1 \wedge 2 \wedge 4 \wedge 5$	-	2	0.1	-	2	1.2	-	2	508.3	timeout			timeout			timeout			timeout			timeout		
$6 \rightarrow 1 \wedge 2 \wedge 4 \wedge 5$	-	2	0.5	-	2	687.7	timeout			timeout			timeout			timeout			timeout			timeout		
$6 \wedge 7 \rightarrow 1 \wedge 2 \wedge 4 \wedge 5$	-	2	0.7	-	2	817.9	timeout			timeout			timeout			timeout			timeout			timeout		
$6 \wedge 7 \rightarrow 1 \wedge 2 \wedge 5 \wedge 8$	+	2	0.4	+	3	320.7	timeout			timeout			timeout			timeout			timeout			timeout		
$6 \wedge 7 \rightarrow 1 \wedge 2 \wedge 5 \wedge 8 \wedge 9$	-	2	0.1	-	2	0.2	-	2	3.2	timeout			timeout			timeout			timeout			timeout		
$6 \wedge 7 \wedge 10 \rightarrow 1 \wedge 2 \wedge 5 \wedge 8 \wedge 9$	+	2	0.6	+	2	11.7	timeout			timeout			timeout			timeout			timeout			timeout		