Symbolic Bounded Synthesis[*]

Rüdiger Ehlers
Reactive Systems Group, Universität des Saarlandes

Synthesizing finite-state systems from full linear-time temporal logic (LTL) is an ambitious way to tackle the challenge of constructing correct-by-construction systems. One particularly promising approach in this context is *bounded synthesis*, originally proposed by Schewe and Finkbeiner, which in turn builds upon *Safraless synthesis*, as described by Kupferman and Vardi. Previous implementations of these approaches performed the computation either in an explicit way or used symbolic data structures other than binary decision diagrams (BDDs). In this paper, we reconsider BDDs as state space representation and use it as data structure for bounded synthesis. The key to this construction is the application of two novel optimisation techniques that decrease the number of state bits in such a representation significantly. The first technique uses *signalling* bits to connect sub-games representing the safety- and non-safety parts of the specification. The second technique is based on a closer analysis of the step of building a safety game from a universal automaton and uses a sufficient condition to remove some so-called counters from the state space of the game.

We evaluate our approach on several benchmark suites and show that the new approach leads to a computation time improvement of several orders of magnitude.

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 [31]. Recently, it has been argued that this is however not a big problem [32] as *realisable* specifications typically have implementations that are small, which can be exploited. This observation is used in the context of *bounded synthesis* [32, 14], which builds upon the *Safraless synthesis* principle [26]. 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 for building a safety game such that winning strategies in the game correspond to implementations that satisfy the specification. The bound in this setting describes the maximally allowed number of visits to rejecting states in the co-Büchi automaton along some run of the automaton that corresponds to the input/output observed along the prefix play in the game so far. If there exists an implementation satisfying a given specification, then there exists some bound such that the resulting game is winning.

[*]This is an extended version of the paper "Symbolic Bounded Synthesis", presented at the 22nd International Conference on Computer Aided Verification (CAV 2011). 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).

In practice, the bound value required is usually rather small [32, 16, 14, 15], 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 [21], performs the realisability check in an explicit way. Recently, a symbolic algorithm based on antichains has been presented [14], 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 [27, 12] 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 [40, 33, 7]. 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 the safety and non-safety parts of a specification can be handled separately in the synthesis game. As for safety properties, no counters are necessary, this reduces the computation time significantly and allows for 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 [5, 35, 4, 6, 18], both containing LTL formulas, we design our technique to be adapted to this case. As a second contribution, we present a technique that allows for a further reduction of the number of counters in the non-safety part of the synthesis games.

Obviously, the optimisations described in this paper cannot circumvent the 2EXPTIME-completeness of the LTL synthesis problem, but are rather targeted towards improving the applicability of LTL synthesis from specifications that are deemed to be typical for the practice. As representatives for such specifications, we use the benchmarks from [21, 14] to evaluate our approach. We also consider a new *load balancing* benchmark that mimics the specification design phases of a scalable system.

This paper consists of four parts. In the first part, we recapitulate the basics of the bounded synthesis approach. Then, we discuss the main contributions of this work. The third part provides information on additional minor techniques needed for an efficient BDD-based implementation of the contributions presented. The fourth part then consists of experimental results and a conclusion.

To be more precise, in the next section, we briefly state the preliminaries. Then, we discuss the solution process of obligation games, which forms the basis of the bounded synthesis approach, which is in turn described in Section 4.

With respect to the second part, we first explain the new specification splitting and signalling technique for bounded synthesis in Section 5, and then discuss in Section 6 how some counters that are introduced in the bounded synthesis process can be stripped away for increased efficiency of the overall approach.

In the third part, we start by discussing how the game that is solved in the bounded synthesis process can efficiently be encoded into binary decision diagrams in Section 7. In Section 8, we then show how the techniques presented in this paper can also be used for detecting unrealisable specifications, and for those that are realisable, we discuss the fully symbolic extraction of prototype implementations satisfying the specification in Section 9.

Section 10 contains the experimental results of running a tool that implements the techniques introduced in this paper on the benchmarks from [21] and [14] as well as on a novel load-balancing system case study. We conclude with a summary.

2 Preliminaries

2.1 Basics

We denote by $\mathbb{N}$ the natural numbers, including 0, and $\mathbb{B} = \{\mathbf{false}, \mathbf{true}\}$ are the Boolean values. Given some *alphabet* Σ, we denote by Σ^ω and Σ^* the sets of infinite and finite words over Σ, respectively. Subsets of Σ^ω are also called *ω-languages*. For a word $w = w_0 w_1 \ldots$, we denote the set of elements occurring infinitely often in w by $\inf(w)$ and for some $i \in \mathbb{N}$, denote by $w^i = w_i w_{i+1} w_{i+2} \ldots$ the suffix of w from the ith element. By $(A \rightarrow B)$

we denote the set of all functions with domain A and co-domain B. For elements (a, b) of some set $A \times B$, we denote its projections by $(a, b)|_A = a$ and $(a, b)|_B = b$.

2.2 Word Automata

An *automaton* is a tuple $\mathcal{A} = (Q, \Sigma, \delta, Q_{init}, \mathcal{F})$ where Q is a finite set of *states*, Σ is a finite *alphabet*, $\delta : Q \times \Sigma \to 2^Q$ is a *transition function*, $Q_{init} \subseteq Q$ is the set of *initial states* and $\mathcal{F} : Q \to \mathbb{N}$ is a so-called *colouring function*. For the scope of this paper, we need four types of automata: universal co-Büchi word automata (UCW), deterministic parity automata, deterministic safety automata and deterministic co-safety automata. Deterministic safety and co-safety automata and universal co-Büchi automata are parity automata with domain $\{0, 1\}$ for the colouring function. The set of states with colour 1 in a safety automaton is absorbing, and for a co-safety automaton the set of states with colour 0 is absorbing, i.e., all transitions from any state in such a set leads back into the set. For deterministic automata, we require that for all $(q, x) \in Q \times \Sigma$, we have $|\delta(q, x)| = 1$, and that the set of initial states contains only one state. We call a subset of states in $\mathcal{A}$ a *strongly connected component (SCC)* if every state in the subset is reachable by some sequence of transitions from any other state in the subset. An SCC $S \subseteq Q$ is furthermore called a *maximal strongly connected component* if there is no SCC that is a strict superset of S. By definition, we assume that every state is reachable from itself, so we can partition the state space of an automaton into its set of maximal SCCs $\mathcal{D}(\mathcal{A})$.

Given an infinite word $w = w_0 w_1 \ldots \in \Sigma^\omega$, we say that some infinite sequence $\pi = \pi_0 \pi_1 \ldots \in Q^\omega$ is a *run* of $\mathcal{A}$ over w if $\pi_0 \in Q_{init}$ and for all $i \in \mathbb{N}$, $\pi_{i+1} \in \delta(\pi_i, w_i)$. We say that π is an *accepting run* for w if $\max(\inf(\mathcal{F}(\pi_0)\, \mathcal{F}(\pi_1) \ldots))$ is even. A word w is *accepted* by $\mathcal{A}$ if all runs for w are accepting. The set of all words accepted by $\mathcal{A}$ is called its *language*, denoted by $\mathcal{L}(\mathcal{A})$.

2.3 Finite-state Machines (FSMs)

Formally, we distinguish two types of FSMs: *Mealy* and *Moore machines*. A Mealy machine is a tuple $\mathcal{M} = (S, I, O, \delta, s_{init})$, while a Moore machine is described by a tuple $\mathcal{M} = (S, I, O, \delta, s_{init}, L)$. In both cases, S is a set of *states*, I is a set of *inputs symbols*, O is a set of *output symbols* and $s_{init} \in S$ is an *initial state*. For Mealy machines, $\delta : Q \times I \to Q \times O$ is the *transition function*, while for Moore machines, this function is defined as $\delta : Q \times I \to Q$ and $L : Q \to O$ is the *labelling* function.

We say that some word $w = w_0 w_1 w_2 \ldots \in (I \times O)^\omega$ is in the *language* of a Mealy machine $\mathcal{M}$, written as $\mathcal{L}(\mathcal{M})$, if there exists some run $\pi = \pi_0 \pi_1 \ldots \in S^\omega$ of the machine with $\pi_0 = s_{init}$ and for all $i \in \mathbb{N}$, $(\pi_{i+1}, w_i|_O) = \delta(\pi_i, w_i|_I)$. Likewise, w is in the language of a Moore machine if there exists some run $\pi = \pi_0 \pi_1 \ldots \in S^\omega$ of the machine with $\pi_0 = s_{init}$ and for all $i \in \mathbb{N}$, $\delta(\pi_i, w_i|_I) = \pi_{i+1}$ and $w_i|_O = L(\pi_i)$. Such a word is also called a *trace* of $\mathcal{M}$.

We say that an FSM $\mathcal{M}$ over the input set I and the output set O *realizes* a word automaton over the alphabet $I \times O$ if the language of $\mathcal{M}$ is contained in the language of the automaton.

2.4 Linear-time Temporal Logic (LTL)

Before a system that is correct with respect to its specification can be synthesized, the specification has to be formally stated. For such a task, *linear-time temporal logic* [30] (LTL) is a commonly used logic. Syntactically, LTL formulas are defined inductively as follows (over some set of atomic propositions AP):

- For all atomic propositions $x \in \mathsf{AP}$, x is an LTL formula.
- Let ϕ_1 and ϕ_2 be LTL formulas. Then, $\neg\phi_1$, $(\phi_1 \vee \phi_2)$, $(\phi_1 \wedge \phi_2)$, $\mathsf{X}\phi_1$, $\mathsf{F}\phi_1$, $\mathsf{G}\phi_1$, and $(\phi_1 \mathsf{U} \phi_2)$ are also LTL formulas.

The validity of an LTL formula ϕ over AP is defined inductively with respect to an infinite trace $w = w_0 w_1 \ldots \in (2^{\mathsf{AP}})^\omega$. Let ϕ_1 and ϕ_2 be LTL formulas. We set:

- $w \models p$ if and only if (iff) $p \in w_0$ for $p \in \mathsf{AP}$
- $w \models \neg\psi$ iff not $w \models \psi$

- $w \models (\phi_1 \vee \phi_2)$ iff $w \models \phi_1$ or $w \models \phi_2$
- $w \models (\phi_1 \wedge \phi_2)$ iff $w \models \phi_1$ and $w \models \phi_2$
- $w \models \mathsf{X}\phi_1$ iff $w^1 \models \phi_1$
- $w \models \mathsf{G}\phi_1$ iff for all $i \in \mathbb{N}$, $w^i \models \phi_1$
- $w \models \mathsf{F}\phi_1$ iff there exists some $i \in \mathbb{N}$ such that $w^i \models \phi_1$
- $w \models (\phi_1 \mathsf{U} \phi_2)$ iff there exists some $i \in \mathbb{N}$ such that for all $0 \leq j < i$, $w^j \models \phi_1$ and $w^i \models \phi_2$

We use the usual precedence rules for LTL formulas (unary temporal operators bind stronger than binary temporal operators) in order to be able to omit unnecessary braces and also allow the abbreviations typically used for Boolean logic, e.g., that $a \rightarrow b$ is equivalent to $\neg a \vee b$ for all formulas a, b.

Given an LTL formula ψ over AP, we can convert ψ to an equivalent universal co-Büchi word automaton $\mathcal{A}$ over the alphabet 2^{AP} such that precisely the words over 2^{AP} that satisfy ϕ are also in the language of $\mathcal{A}$ [26, 39]. The size of the UCW is at most exponential in the length of ψ.

The LTL synthesis problem is to determine, for a given LTL formula ψ over some set of atomic propositions $\mathsf{AP} = \mathsf{AP}_I \uplus \mathsf{AP}_O$, whether there exists a Mealy or Moore machine with input symbol set 2^{AP_I} and output symbol set 2^{AP_O} whose language is a subset of the words that satisfy ψ, and to compute such a machine whenever existing. Whenever such a machine exists, we call the specification *realisable*, otherwise it is *unrealisable*. We will mostly be concerned with Mealy machine synthesis in this paper.

We call an LTL formula over some set of atomic proposition AP a *safety property* [34] if for every word $w = w_0 w_1 \ldots \in (2^{\mathsf{AP}})^\omega$ that does not satisfy the formula, there exists some $i \in \mathbb{N}$ such that every other infinite word starting with $w_0 w_1 \ldots w_i$ does also not satisfy the formula.

2.5 Obligation Games

In this paper, we reduce the synthesis problem to determining the winner (and obtaining a winning strategy) in so-called *obligation games* [37], which contain reachability and safety games as special cases. They are sometimes also called *weak Muller games* in the literature.

Intuitively, when solving the synthesis problem by reducing it to determining the winner in an obligation game, the two players (player 0 and 1) in the game represent the input and output of a finite-state machine. Player 0 provides the input to the system, while player 1 produces the output. The game needs to be designed in a way such that whenever player 1 has a winning strategy in the game, the strategy represents a finite-state machine that satisfies the original specification.

Formally, a *game* is a tuple $\mathcal{G} = (V, \Sigma_0, \Sigma_1, E, v_{init}, \mathcal{F})$, where V is a finite set of *positions* (also called the *vertices* of the game), Σ_0 and Σ_1 are the finite sets of *actions* for the two players, $E : V \times \Sigma_0 \times \Sigma_1 \rightarrow V$ is an *edge function*, $v_{init} \in V$ denotes the *initial position* and $\mathcal{F}$ is a *winning condition* in form of a Boolean formula in which subsets of V are used as atomic propositions.[1] For a winning condition $\mathcal{F}$, we denote by $\mathcal{S}(\mathcal{F})$ the set of all subsets of V that occur in the formula $\mathcal{F}$. We also call the sub-tuple $(V, \Sigma_0, \Sigma_1, E, v_{init})$ of $\mathcal{G}$ the *transition structure* of $\mathcal{G}$.

A *decision sequence* $\rho = \rho_0^0 \rho_0^1 \rho_1^0 \rho_1^1 \rho_2^0 \rho_2^1 \ldots$ is an infinite sequence such that for every $i \in \mathbb{N}$, $\rho_i^0 \in \Sigma_0$ and $\rho_i^1 \in \Sigma_1$. We say that ρ induces a *play* $\pi = \pi_0 \pi_1 \ldots$ if $\pi_0 = v_{init}$ and for every $i \in \mathbb{N}$, $\pi_{i+1} = E(\pi_i, (\rho_i^0, \rho_i^1))$.

We categorize plays in $\mathcal{G}$ by whether they are winning for player 0 or 1. Given an infinite play $\pi = \pi_0^0 \pi_0^1 \ldots$, we define π as being *winning* for player 1 if $\{s \in \mathcal{S}(\mathcal{F}) \mid s \cap \mathrm{Occ}(\pi) \neq \emptyset\} \models \mathcal{F}$, where $\mathrm{Occ}(\pi)$ denotes the set of positions occurring along π, i.e., $\mathrm{Occ}(\pi) = \{v \in V \mid \exists i \in \mathbb{N} : \pi_i = v\}$.[2] Whenever a play is not winning for

[1]While this representation of the winning condition is rather uncommon in the literature, it will allow us later to discuss the composition of games in a simplified way.

[2]In other words, whether a play is winning for player 1 can be determined by evaluating which of the sets of $\mathcal{S}(\mathcal{F})$ contain a position that is visited along the play, and then substituting all sets in $\mathcal{F}$ for which some position is visited along the play by $\mathbf{true}$ and the other ones by $\mathbf{false}$. If the resulting Boolean formula evaluates to $\mathbf{true}$, the play is winning for player 1. While this definition of the acceptance condition is uncommon for obligation games in the literature, we use it here as it is very helpful to describe the game compositions in the following chapters in a simple and intuitive way.

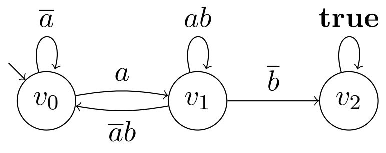

Figure 1: Graphical representation of an obligation game $\mathcal{G} = (V, \Sigma_0, \Sigma_1, E, v_{init}, \mathcal{F})$ with $\Sigma_0 = 2^{\{a\}}$, $\Sigma_1 = 2^{\{b\}}$, $v_{init} = v_0$ and $\mathcal{F} = \neg\{v_2\} \wedge \{v_1\}$. The edge labels represent the constraints on the transitions, e.g., the label $\overline{a}b$ on the edge from v_1 to v_0 describes that $E(v_1, \emptyset, \{b\}) = v_0$ and the self-loop of v_2 describes that $E(v_2, A, B) = v_2$ for every $A \subseteq \{a\}$ and $B \subseteq \{b\}$.

player 1, we define the play to be winning for player 0. The Occ function is defined likewise for finite plays. We also say that some decision sequence is winning for some player if the induced play is winning.

It is well-known that in obligation games, there exists a *winning strategy* for precisely one of the players, i.e., for some player $j \in \{0, 1\}$, there exists a function $f : (\Sigma_0 \uplus \Sigma_1)^* \rightarrow \Sigma_j$ such that all decision sequences that are in correspondence to f are winning for player j. A decision sequence $\rho = \rho_0^0\rho_0^1\rho_1^0\rho_1^1\rho_2^0\rho_2^1 \ldots$ is said to be *in correspondence* to f if and only if for all $i \in \mathbb{N}$, $\rho_i^j = f(\rho_0^0\rho_0^1 \ldots \rho_{i-1+j}^{1-j})$.

Given a game $\mathcal{G}$ over the action sets Σ_0 and Σ_1, we say that some Moore machine $\mathcal{M} = (S, \Sigma_0, \Sigma_1, \delta, s_{init}, L)$ induces a strategy f for player 0 in $\mathcal{G}$ with $f(\epsilon) = L(s_{init})$ and $f(\rho_0^0\rho_0^1 \ldots \rho_k^1) = L(\delta(\delta(\ldots\delta(s_{init}, \rho_0^1), \rho_1^1), \ldots, b\rho_k^1))$ for every prefix decision sequence $\rho_0^0\rho_0^1 \ldots \rho_k^1$. Likewise, a Mealy machine $\mathcal{M} = (S, \Sigma_0, \Sigma_1, \delta, s_0)$ induces a strategy f for player 1 by choosing $f(\rho_0^0\rho_0^1 \ldots \rho_k^0) = \delta(\delta(\ldots\delta(s_{init}, \rho_0^0)|_S, \rho_1^0)|_S, \ldots, \rho_k^0)|_O$ for every prefix decision sequence $\rho_0^0\rho_0^1 \ldots \rho_k^0$. For obligation games, it is assured that whenever there exists a winning strategy for player 0/1, there also exists a Moore/Mealy automaton inducing a winning strategy, respectively.

In this paper, we mostly take the view of player 1. Thus, we simply call a decision sequence or play winning if it is winning for player 1. We say that a position is winning for a player $p \in \{0, 1\}$ if changing v_{init} to the position makes player p having a winning strategy in the game. We say that a game is winning for a player if the player has a winning strategy.

Henceforth, we use the names of the game tuple components to extract these from a game. For example, for some game $\tilde{\mathcal{G}} = (\tilde{V}, \tilde{\Sigma}_0, \tilde{\Sigma}_1, \tilde{E}, \tilde{v}_{init}, \tilde{\mathcal{F}})$, we define $v_{init}(\tilde{\mathcal{G}}) = \tilde{v}_{init}$. For automata tuples, we apply the same notational concept.

2.6 Example for Obligation Games

To clarify the connection between the synthesis problem and obligation games, we give an example here. Figure 1 shows an obligation game $\mathcal{G} = (V, \Sigma_0, \Sigma_1, E, v_{init}, \mathcal{F})$ for which every decision sequence that induces a winning play represents a word that satisfies the LTL specification $\psi = \mathsf{F}a \wedge \mathsf{G}(\neg a \vee \mathsf{X}b)$. The winning condition $\mathcal{F} = \neg\{v_2\} \wedge \{v_1\}$ of $\mathcal{G}$ is satisfied by the plays that never visit v_2 but eventually visit v_1. The game in the example also has the property that *precisely* the decision sequences that induce a winning play for player 1 in $\mathcal{G}$ are the ones that satisfy ψ. In general, LTL specifications do not always permit to build obligation games with this property. We will however see in Section 4 that they are still a suitable computation model for synthesis when applying the *bounded synthesis* approach.

2.7 Important Special Cases of Winning Conditions in Obligation Games

Let $\mathcal{G} = (V, \Sigma_0, \Sigma_1, E, v_{init}, \mathcal{F})$ be a game. In this paper, we consider two prominent special cases of winning conditions:

- If $\mathcal{F} = \neg V'$ for some $V' \subseteq V$, we call $\mathcal{G}$ a safety game.
- If $\mathcal{F} = V'$ for some $V' \subseteq V$, we call $\mathcal{G}$ a co-safety or reachability game.

Thus, the reachability winning condition is the dual case of the safety winning condition. Note that if a winning condition $\mathcal{F}$ consists of a conjunction of safety winning objectives or a disjunction of reachability objectives, we can use the following simplification rules:

$$\bigwedge_{V' \in \{V'_1,\ldots,V'_n\}} \neg V' \equiv \neg(V'_1 \cup \ldots \cup V'_n) \tag{1}$$

$$\bigvee_{V' \in \{V'_1,\ldots,V'_n\}} V' \equiv (V'_1 \cup \ldots \cup V'_n) \tag{2}$$

2.8 Safety Automata and Safety/co-safety Games

Given a deterministic safety automaton $\mathcal{A}$ over the alphabet $\Sigma = \Sigma_I \times \Sigma_O$, we can easily build a safety game $\mathsf{ToGame}_1(\mathcal{A}, \Sigma_I, \Sigma_O)$ that is winning for player 1 if and only if there exists a Mealy machine reading Σ_I and writing to Σ_O that realizes $\mathcal{A}$. Likewise, we can build a co-safety game $\mathsf{ToGame}_0(\mathcal{A}, \Sigma_O, \Sigma_I)$ that is winning for player 0 if and only if there exists a Moore machine reading Σ_I and writing to Σ_O that realizes $\mathcal{A}$.

Definition 1. *Given a deterministic safety automaton $\mathcal{A} = (Q, \Sigma, \delta, \{q_{init}\}, F)$ with $\Sigma = \Sigma_a \times \Sigma_b$, we define the game $\mathsf{ToGame}_p(\mathcal{A}, \Sigma_a, \Sigma_b) = (V, \Sigma_a, \Sigma_b, E, v_{init}, \mathcal{F})$ for $p \in \{0,1\}$ as follows:*

- $V = Q$
- *For all $v \in V$, $x \in \Sigma_a$, $y \in \Sigma_b$: $E(v,x,y) = \delta(v,(x,y))$*
- $v_{init} = q_{init}$
- $\mathcal{F} = \neg\{v \in V \mid F(v) = 1\}$ *if $p = 1$ and $\mathcal{F} = \{v \in V \mid F(v) = 1\}$ otherwise.*

2.9 Parallel Composition of Games

Given two games $\mathcal{G} = (V, \Sigma_0, \Sigma_1, E, v_{init}, \mathcal{F})$ and $\mathcal{G}' = (V', \Sigma_0, \Sigma_1, E', v'_{init}, \mathcal{F}')$, for some $\odot \in \{\wedge, \vee\}$, we define the *parallel composition* $\mathcal{G}||_\odot\mathcal{G}'$ of the two games as the (synchronized product) game $\mathcal{G}^p = (V^p, \Sigma_0, \Sigma_1, E^p, v^p_{init}, \mathcal{F}^p)$ with:

- $V^p = V \times V'$
- $v^p_{init} = (v_{init}, v'_{init})$
- For all $v \in V$, $v' \in V'$, $x \in \Sigma_0$ and $y \in \Sigma_1$, $E^p((v,v'),x,y) = (E(v,x,y), E'(v',x,y))$.
- $\mathcal{F}^p = \mathcal{F}_1 \odot \mathcal{F}_2$, where:
  1. $\mathcal{F}_1$ is obtained by taking the Boolean formula $\mathcal{F}$ and replacing every occurrence of some atomic proposition $s \in \mathcal{S}(\mathcal{F})$ in the formula by $s \times Q'$, and
  2. $\mathcal{F}_2$ is obtained by taking $\mathcal{F}'$ and replacing every occurrence of some atomic proposition $s \in \mathcal{S}(\mathcal{F}')$ by $Q \times s$.

The conjunctive parallel composition $\mathcal{G}||_\wedge\mathcal{G}'$ has the property that precisely the decision sequences $\rho \in (\Sigma_0\Sigma_1)^\omega$ that are winning for player 1 in $\mathcal{G}$ and $\mathcal{G}'$ are also winning for player 1 in $\mathcal{G}^p$. Likewise, the disjunctive parallel composition $\mathcal{G}||_\vee\mathcal{G}'$ has the property that precisely the decision sequences $\rho \in (\Sigma_0\Sigma_1)^\omega$ that are winning for player 1 in $\mathcal{G}$ or $\mathcal{G}'$ are also winning for player 1 in $\mathcal{G}^p$.

Note that when taking the conjunctive parallel composition of two safety games, the resulting game is still a safety game, using the simplification given in Equation 1.

2.10 Binary Decision Diagrams

For representing sets of vertices and the transition relation in safety games symbolically, we use *reduced ordered binary decision diagrams* (BDDs) [8, 9], which represent characteristic functions $f : 2^{\mathcal{V}} \rightarrow \mathbb{B}$ for some finite set of variables $\mathcal{V}$. We start by treating them on an abstract level and state the operations on them that we use. For a comprehensive overview, see [9]. 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 $x \subseteq \mathcal{V}$. The negation of a BDD is defined similarly. Given some set of variables $V' \subseteq \mathcal{V}$ and a BDD f, we define $\exists V'.f$ as a function that maps all $x \subseteq \mathcal{V}$ to $\mathbf{true}$ for which there exists some $x' \subseteq V'$ such that $f(x' \cup (x \setminus V')) = \mathbf{true}$. 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 $x \subseteq \mathcal{V}$ is mapped to $\mathbf{true}$ if and only if $f(x \setminus \{l'_1, \ldots, l'_n\} \cup \{l_i \mid \exists 1 \leq i \leq n : l'_i \in x\}) = \mathbf{true}$. By abuse of notation, we say that $f =$ **false** or $f =$ **true** if $f(x) =$ **false** or $f(x) =$ **true** for all $x \subseteq \mathcal{V}$, respectively.

In the sequel, we will also be concerned with operations on BDDs over different variable sets. We assume the standard semantics for such cases, e.g., for a BDD f over $\mathcal{V}$ and a BDD f' over $\mathcal{V}'$, we define $(f \wedge f')(x) = f(x \cap \mathcal{V}) \wedge f'(x \cap \mathcal{V}')$ for all $x \subseteq \mathcal{V} \cup \mathcal{V}'$.

We also define least and greatest fixed points of monotone functions over BDDs. Given two BDDs f and g over the set of variables $\mathcal{V}$, we write $f \geq g$ if for all $a \subseteq \mathcal{V}$, we have that $f(a) = \mathbf{true}$ implies $f(b) = \mathbf{true}$. Let $g : (2^{\mathcal{V}} \rightarrow \mathbb{B}) \rightarrow (2^{\mathcal{V}} \rightarrow \mathbb{B})$ be a monotone function that maps a BDD over some set of variables $\mathcal{V}$ onto another BDD over the same set of variables. We call g monotone if for all BDDs f and f', if $f \geq f'$, then $g(f) \geq g(f')$. We define $\mu^0.g =$ **false**, $\mu^{i+1}.g = g(\mu^i.g)$ for all $i \in \mathbb{N}$ and for j being the least element of $\mathbb{N}$ such that $\mu^{j+1}.g = \mu^j.g$, the least fixed point is defined as $\mu.g = \mu^j.g$. The greatest fixed point $\nu.g$ of a monotone function g is computed likewise, i.e., we define $\nu^0.g = \mathbf{true}$, $\nu^{i+1}.g = g(\nu^i.g)$ for all $i \in \mathbb{N}$ and for j being the least element of $\mathbb{N}$ such that $\nu^{j+1}.g = \nu^j.g$, we define $\nu.g = \nu^j.g$. In both cases, we call $\mu^i.g$ and $\nu^i.g$ prefixed points of g. Checking whether a computed prefixed point is already the fixed point is simple in practice by using the fact that equivalence checks on BDDs can be performed in constant time in modern BDD libraries [12].

For most parts of this paper, it is only of importance what BDDs are good for and not how they actually represent Boolean functions. In Section 9, however, we will need this information.

Intuitively, a BDD is an acyclic graph with a root node and two sink nodes, labelled $\mathbf{true}$ and $\mathbf{false}$. Every node in the graph, except for the sink nodes, is labelled by a variable and has two successor nodes, connected by so-called *then* and *else* edges. Given a set of variables $\mathcal{V}$ and Boolean function $f : 2^{\mathcal{V}} \rightarrow \mathbb{B}$, we say that some set $v \subseteq \mathcal{V}$ (which we also call a *variable valuation* in this context) induces $f(v) = \mathbf{true}$ if and only if there exists a path in the BDD from the root node to $\mathbf{true}$ where in every node, we take the $then$-successor whenever the variable the node is labelled with is contained in v, and take the $else$-successor otherwise. We consider only ordered BDDs here, for which there exists a total order on the variables, and along every path of a BDD, every variable can only occur at most once and in the given order.

3 Solving Obligation Games

In this section, we discuss the solution process for obligation games, i.e., how the winner of a game can be obtained.[3] Obligation games are determined, i.e., there exists a winning strategy for exactly one of the players. In general, the winning player needs to take the history of the play into account when making the next decision in order to win a play. However, as we will see below, this does not make solving obligation games hard, as it in fact suffices for the winning player to keep track of which vertices have already been visited along a prefix play. Even more, as vertices that occur only in combination in the vertex sets of the winning condition need not be distinguished, the amount of information that needs to be stored is even smaller. Formally, let $\mathcal{G} = (V, \Sigma_0, \Sigma_1, E, v_{init}, \mathcal{F})$ be an obligation game. Given a set of vertices $V' \subseteq V$, we denote by $\mathcal{C}(V', \mathcal{F})$ the set of atomic propositions of $\mathcal{F}$ that denote position sets that intersect with V', i.e., $\mathcal{C}(V', \mathcal{F}) = \{V'' \in \mathcal{S}(\mathcal{F}) \mid V'' \cap V' \neq \emptyset\}$. We use $\mathcal{Z}(\mathcal{G})$ to

[3]The solution process described here is related to solving so-called games with a weak winning condition (see [13] for a definition and details) and a reformulation of a procedure for solving games with a weak transition structure (with uniform treatment of all game positions in a strongly connected component, see [19] for a definition and details).

represent the set of subsets of $\mathcal{S}(\mathcal{F})$ that need to be distinguished when tracking which vertices have been visited so far in a prefix game, i.e.,

$$\mathcal{Z}(\mathcal{G}) = \{S \subseteq \mathcal{S}(\mathcal{F}) \mid \exists V' \subseteq V : S = \{V'' \in \mathcal{S}(\mathcal{F}) \mid V'' \cap V' \neq \emptyset\}\}$$

For a finite game $\mathcal{G}$, the set $\mathcal{Z}(\mathcal{G})$ is finite. Together with set inclusion, $\mathcal{Z}(\mathcal{G})$ forms a lattice, where $\emptyset$ is the minimal and $\mathcal{S}(\mathcal{F})$ is the maximal element. The following definition and the subsequent lemma formalize this intuition that $\mathcal{Z}(\mathcal{G})$ is useful as lattice of information about the past to remember in obligation games.

Definition 2. *Let $\rho = \rho_0^0\rho_0^1\ldots$ be a decision sequence in an obligation game $\mathcal{G} = (V, \Sigma_0, \Sigma_1, E, v_{init}, \mathcal{F})$ and $\pi = \pi_0\pi_1\ldots$ be the corresponding play. We define $\eta = \eta_0\eta_1\ldots$ to be the corresponding path through $\mathcal{Z}(\mathcal{G})$, i.e., for all $i \in \mathbb{N}$, we have $\eta_i = \mathcal{C}(\mathrm{Occ}(\pi_0\ldots\pi_i), \mathcal{F})$. If for some $j \in \mathbb{N}$, we have $\eta_j = \eta_{j+1} = \ldots$, we define the limit value η_∞ as $\eta_\infty = \eta_j$.*

Lemma 3. *Let $\rho = \rho_0^0\rho_0^1\ldots$ be a decision sequence in an obligation game $\mathcal{G} = (V, \Sigma_0, \Sigma_1, E, v_{init}, \mathcal{F})$, $\pi = \pi_0\pi_1\ldots$ be the corresponding play and $\eta = \eta_0\eta_1\ldots$ be the corresponding path through $\mathcal{Z}(\mathcal{G})$. For all $i \in \mathbb{N}$, we have $\eta_i \subseteq \eta_{i+1}$. Furthermore, η has a limit value η_∞ and ρ is winning if and only if $\eta_\infty \models \mathcal{F}$.*

As an example, consider the obligation game $\mathcal{G}$ in Figure 1. Here, the winning condition is $\mathcal{F} = \neg\{v_2\} \wedge \{v_1\}$. Obviously, for this game, we have $\mathcal{Z}(\mathcal{G}) = \{\{\emptyset\}, \{\{v_1\}\}, \{\{v_2\}\}, \{\{v_1\}, \{v_2\}\}\}$. For a play $\pi = v_0v_1v_0v_1v_2\ldots$, the corresponding path $\eta = \eta_0\eta_1\eta_2\eta_3\eta_4\ldots$ has $\eta_0 = \emptyset$, $\eta_1 = \eta_2 = \eta_3 = \{\{v_1\}\}$, and $\eta_i = \{\{v_1\}, \{v_2\}\}$ for all $i \geq 4$, i.e., it is continuously updated which elements of $\mathcal{S}(\mathcal{F})$ are state sets that have are already been visited along the play.

The main idea of keeping track of the path in $\mathcal{Z}(\mathcal{G})$ along a play is that while staying in some element of $\mathcal{Z}(\mathcal{G})$, the winning player has a memoryless strategy (which only depends on the current vertex in a play of the game and not on the history) in $\mathcal{G}$; only when a new vertex is visited and thus the play moves to a different element in $\mathcal{Z}(\mathcal{G})$, the strategy might have to change. Since we know that $\mathcal{Z}(\mathcal{G})$ is finite, this allows us to solve the game in a bottom-up fashion, where we start from the maximal elements in $\mathcal{Z}(\mathcal{G})$ and iteratively compute the states that are winning for successively shrinking sets of states visited so far, up to the set of state sets of $\mathcal{S}(\mathcal{F})$ that contain the initial vertex.

We use some standard notation for this task, starting with the enforceable predecessor operator. Given a game $\mathcal{G} = (V, \Sigma_0, \Sigma_1, E, v_{init}, \mathcal{F})$, we define the $\mathsf{EnfPre} : 2^V \to 2^V$ operator as follows:

$$\mathsf{EnfPre}(V') = \{v \in V \mid \forall x \in \Sigma_0 \exists y \in \Sigma_1 : E(v, x, y) \in V'\}$$

When solving an obligation game and working upwards in the $\mathcal{Z}(\mathcal{G})$ lattice, we might have already identified some vertices as losing (as visiting them causes a transition in the $\mathcal{Z}(\mathcal{G})$ lattice) and some as winning. Let us assume that we have two (distinct) sets of positions $W_0, W_1 \subseteq V$ given and player 1 wins whenever some position in W_1 is reached before any position in W_0 is reached, and furthermore player 1 wins if no position in $W_0 \cup W_1$ is ever visited. The set of winning positions for this player and winning condition can easily be obtained by computing a greatest fixed point:

$$\mathsf{Win}_1(W_0, W_1) = \nu X.X \cap (\mathsf{EnfPre}(X \setminus W_0) \cup W_1)$$

Likewise, we can compute the set of states from which player 1 can enforce that no position in W_0 is visited before (possibly) some position in W_1 is reached, assuming that player 1 loses whenever no position in $W_0 \cup W_1$ is ever visited:

$$\mathsf{Win}_0(W_0, W_1) = \mu X.X \cup W_1 \cup \mathsf{EnfPre}(X \setminus W_0)$$

Using these prerequisites, we can finally devise a game solving algorithm for obligation games. For all $S \in \mathcal{Z}(\mathcal{G})$, we define $T(S) = \mathsf{Win}_b(W_0, W_1)$, where:

- $b = 1$ if and only if $S \models \mathcal{F}$
- $W_0 = \bigcup_{S' \in \mathcal{Z}(\mathcal{G}), S \neq S'} \{v' \notin T(S') \mid S' = S \cup \mathcal{C}(\{v'\}, \mathcal{F})\}$

- $W_1 = \bigcup_{S' \in \mathcal{Z}(\mathcal{G}), S \neq S'} \{v' \in T(S') \mid S' = S \cup \mathcal{C}(\{v'\}, \mathcal{F})\}$

Theorem 4 (see, e.g., [38]). *Let $\mathcal{G} = (V, \Sigma_0, \Sigma_1, E, v_{init}, \mathcal{F})$ be an obligation game. For every prefix play $\pi = \pi_0\pi_1 \ldots \pi_n$ in $\mathcal{G}$, we have $\pi_n \in T(\mathcal{C}(\mathrm{Occ}(\pi), \mathcal{F}))$ if and only if player 1 has a winning suffix strategy after the prefix play π.*

Corollary 5. *Let $\mathcal{G} = (V, \Sigma_0, \Sigma_1, E, v_{init}, \mathcal{F})$ be an obligation game. Player 1 has a winning strategy in $\mathcal{G}$ if and only if $v_{init} \in T(\mathcal{C}(\{v_{init}\}, \mathcal{F}))$.*

As the values of $\{T(S)\}_{S \in \mathcal{Z}(\mathcal{G})}$ can be computed using a topological order on $\mathcal{Z}(\mathcal{G})$, the overall computation can be done in time linear in $|\Sigma_0| \cdot |\Sigma_1| \cdot |V| \cdot |\mathcal{Z}(\mathcal{G})|$ (cf. [19, 2, 11]).

3.1 Binary Decision Diagrams for Solving Games

Binary decision diagrams are a suitable data structure for the symbolic solution of obligation games (see, e.g., [1]). In order to apply them, we need to encode the set of positions in the game and the sets of actions for the two players into Boolean variables. Then, the edge function can be represented in form of a binary decision diagram (BDD).

Formally, we need four sets of variables $\mathcal{V}_{pre}$, $\mathcal{V}_0$, $\mathcal{V}_1$, and $\mathcal{V}_{post}$ in order to do so. The sets $\mathcal{V}_{pre}$ and $\mathcal{V}_{post}$ are used to represent predecessor and successor positions for the edges in games, while $\mathcal{V}_0$ and $\mathcal{V}_1$ are used for the action sets of player 0 and 1, respectively. A combined variable valuation for all of these sets is consequently capable of representing a single edge, and as a BDD maps variable valuations to $\mathbf{true}$ and $\mathbf{false}$, a BDD over $\mathcal{V}_{pre} \uplus \mathcal{V}_{post} \uplus \mathcal{V}_0 \uplus \mathcal{V}_1$ can represent the whole edge function (by mapping variable valuations that represent existing edges to $\mathbf{true}$ and the others to $\mathbf{false}$).

We assume that $\mathcal{V}_{pre}$ and $\mathcal{V}_{post}$ are totally ordered and use these sets and the lists induced by the sets and the ordering interchangeably. Encoding the set of positions V of a game $\mathcal{G} = (V, \Sigma_0, \Sigma_1, E, v_{init}, \mathcal{F})$ into $\mathcal{V}_{pre}$ and $\mathcal{V}_{post}$ is done using the encoding operators $\llbracket\cdot\rrbracket : 2^V \to (2^{\mathcal{V}_{pre}} \to \mathbb{B})$ and $\llbracket\cdot\rrbracket' : 2^V \to (2^{\mathcal{V}_{post}} \to \mathbb{B})$. For the techniques presented in this paper, any such operators can be used, provided that they fulfil some side-constraints:

1. For all $v, v' \in V$: if $v \neq v'$, then $\llbracket\{v\}\rrbracket \wedge \llbracket\{v'\}\rrbracket = \mathbf{false}$.
2. For all $V', V'' \subseteq V$: $\llbracket V'\rrbracket \vee \llbracket V''\rrbracket = \llbracket V' \cup V''\rrbracket$.
3. $\llbracket V\rrbracket = \mathbf{true}$ and $\llbracket\emptyset\rrbracket = \mathbf{false}$.
4. For all $v \in V$, $\llbracket\{v\}\rrbracket' = \llbracket\{v\}\rrbracket[\mathcal{V}_{post}/\mathcal{V}_{pre}]$.

Similarly, we encode Σ_0 and Σ_1 into $\mathcal{V}_0$ and $\mathcal{V}_1$, and overload the operator $\llbracket\cdot\rrbracket$ for this purpose as this does not introduce ambiguities. The first three of the properties above are also assumed to hold in this case. We defer a description of the encodings actually used in this work to the later sections. The edge function can now be encoded into a BDD B_E by taking:

$$B_E = \bigvee_{v \in V, x \in \Sigma_0, y \in \Sigma_1} \left(\llbracket\{v\}\rrbracket \wedge \llbracket\{x\}\rrbracket \wedge \llbracket\{y\}\rrbracket \wedge \llbracket\{E(v,x,y)\}\rrbracket'\right)$$

Obviously, B_E is a BDD over the set of variables $\mathcal{V} = \mathcal{V}_{pre} \uplus \mathcal{V}_{post} \uplus \mathcal{V}_0 \uplus \mathcal{V}_1$. This encoding has the advantage that it is transparent with respect to taking the parallel composition of two games $\mathcal{G}^1 = (V^1, \Sigma_0, \Sigma_1, E^1, v^{init}, \mathcal{F}^1)$ and $\mathcal{G}^2 = (V^2, \Sigma_0, \Sigma_1, E^2, v^{init}, \mathcal{F}^2)$. If we define $\llbracket\{(v,v')\}\rrbracket = \llbracket\{v\}\rrbracket \wedge \llbracket\{v'\}\rrbracket$ for all positions (v, v') in $\mathcal{G}^1||_\odot\mathcal{G}^2$ for some $\odot \in \{\wedge, \vee\}$, then we compute the BDD for the edge function of $\mathcal{G}^1||_\odot\mathcal{G}^2$ by taking $B_{E^1} \wedge B_{E^2}$.

Given a BDD B_E encoding an edge function E and a BDD $B_{V'}$ encoding a set of positions (over the variables $\mathcal{V}_{pre}$), we can compute the EnfPre operator as follows:

$$\mathsf{EnfPre}(B_{V'}) = \forall\mathcal{V}_0.\exists\mathcal{V}_1.\exists\mathcal{V}_{post}.(B_{V'}[\mathcal{V}_{post}/\mathcal{V}_{pre}] \wedge B_E)$$

This equation is a reformulation of the EnfPre function defined earlier in this section, with the modification that it uses BDDs. The replacement of every $\mathcal{V}_{pre}$ variable by the corresponding variable in $\mathcal{V}_{post}$ is necessary to make

$B_{V'}$ represent the successor positions when taking the conjunction with B_E (and thus applying the edge function). The existential abstraction of the variables in $\mathcal{V}_{post}$ lets the BDD forget information about successor positions. The subsequent universal and existential abstractions of $\mathcal{V}_0$ and $\mathcal{V}_1$ finally quantify over the possible moves of the players and at the same time only let information about the predecessor vertices in the game remain in the result of taking $\mathsf{EnfPre}(B_{V'})$. The rest of the computation process for solving obligation games can be performed as described at the beginning of this section.

4 Bounded Synthesis

The bounded synthesis approach [32, 14] is a conceptually simple technique to check the realizability of a given specification, given in form of a universal co-Büchi word automaton (UCW) for the scope of this paper (which are expressive enough to encode any LTL formula), and to compute a Mealy or Moore machine realizing it in case of a positive answer. We briefly recapitulate the basics of that approach here.

The main observation used in the bounded synthesis approach is that whenever a specification is realizable, there also exists one with a certain maximal number of states, which is exponential in the size of the specification UCW. Furthermore, if some Mealy or Moore FSM realizes a specification, there also exists a bound $b \in \mathbb{N}$ on the number of visits to rejecting states in the UCW along every of its runs for some word in the language of the FSM. By taking both facts together, one can derive a worst-case value of b that needs to be considered when searching for an implementation. For the scope of this section, for a UCW $\mathcal{A}$, we let $|\mathcal{F}(\mathcal{A})|$ represent the number of rejecting states in the UCW, i.e, the ones with colour 1.

Theorem 6 ([28])**.** *Given a universal co-Büchi automaton with n states, we can construct an equivalent deterministic parity automaton $\mathcal{P}$ with $2n^n n!$ states and $2n$ colours.*

Theorem 7 ([32])**.** *Let $\mathcal{M}$ be a Mealy or Moore machine with n states and $\mathcal{A}$ be a universal co-Büchi word automaton. The language of $\mathcal{M}$ is a subset of the language of $\mathcal{A}$ if and only if the maximum number of visits to rejecting states in $\mathcal{A}$ for some run and some word in the language of $\mathcal{M}$ is less than or equal to $|\mathcal{F}(\mathcal{A})| \cdot n$.*

Corollary 8 ([32])**.** *Given a universal co-Büchi word automaton $\mathcal{A}$ over the alphabet $\Sigma_I \times \Sigma_O$ with n states, there exists a Mealy or Moore machine over Σ_I/Σ_O realizing $\mathcal{A}$ if and only if there exists one for which the maximum number of visits to rejecting states in $\mathcal{A}$ for some run and some word in the language of $\mathcal{M}$ is less than or equal to $|\mathcal{F}(\mathcal{A})| \cdot 2n^n n!$.*

This observation gives rise to the following idea: from a UCW, we build a safety automaton that checks if all runs of the UCW for some given word do not exceed some bound value $b \in \mathbb{N}$. Provided that the bound value is chosen high enough, the unrealisability of the resulting safety automaton implies the unrealisability of the original specification.

Definition 9. *Let $\mathcal{A} = (Q, \Sigma, \delta, Q_{init}, F)$ be a UCW. We define the deterministic safety automaton* $\mathsf{Bound}(\mathcal{A}, b) = (Q', \Sigma, \delta', \{q'_{init}\}, F')$ *for some $b \in \mathbb{N}$ by:*

- *$Q' = (Q \to \{-\infty, 0, \ldots, b\}) \cup \{\bot\}$*
- *For all $f \in (Q \to \{-\infty, 0, \ldots, b\})$, $x \in \Sigma$, we have $\delta'(f, x) \in (Q \to \{-\infty, 0, \ldots, b\})$ and*

$$\delta'(f,x)(q) = \max\{f(q') \mid q' \in Q, q \in \delta(q', x)\} + \begin{cases} 1 & \text{if } q \in F \\ 0 & \text{otherwise} \end{cases}$$

  if for all $q \in Q$, we have

$$b \geq \max\{f(q') \mid q' \in Q, q \in \delta(q', x)\} + \begin{cases} 1 & \text{if } q \in F \\ 0 & \text{otherwise} \end{cases},$$

  and $\delta'(f, x) = \bot$ otherwise. In both equations, we assume that $\max(\emptyset) = -\infty$ and that $-\infty + 1 = -\infty$.

- *For all $x \in \Sigma$, $\delta'(\bot, x) = \{\bot\}$*
- $q'_{init} = \{Q_{init} \mapsto 0, Q \setminus Q_{init} \mapsto -\infty\}$
- $F' = \{Q' \setminus \{\bot\} \mapsto 0, \{\bot\} \mapsto 1\}$

Theorem 10 ([32, 14])**.** *Let $\mathcal{A} = (Q, \Sigma, \delta, Q_{init}, F)$ be a UCW with n states and Σ_I and Σ_O be sets such that $\Sigma = \Sigma_I \times \Sigma_O$. There exists a Mealy machine over Σ_I and Σ_O realizing $\mathcal{A}$ if and only if the game $\mathsf{ToGame}_1(\mathsf{Bound}(\mathcal{A}, b), \Sigma_I, \Sigma_O)$ is winning for player 1 with $b = |\mathcal{F}(\mathcal{A})| \cdot 2n^n n!$. Furthermore, there exists a Moore machine over Σ_I and Σ_O realizing $\mathcal{A}$ if and only if the game $\mathsf{ToGame}_0(\mathsf{Bound}(\mathcal{A}, b), \Sigma_O, \Sigma_I)$ is winning for player 0 and $b = |\mathcal{F}(\mathcal{A})| \cdot 2n^n n!$. Winning strategies for the respectively players represent FSMs realizing $\mathcal{A}$.*

In order to speed up bounded synthesis in practice, we can apply a simple strategy: we start with a bound of $b = 1$ and successively increase b while checking whether the resulting game $\mathsf{ToGame}_0(\mathrm{Bound}(\mathcal{A}, b), \Sigma_O, \Sigma_I)$ is winning for player 0 in the Moore setting or whether the game $\mathsf{ToGame}_1(\mathrm{Bound}(\mathcal{A}, b), \Sigma_I, \Sigma_O)$ is winning for player 1 in the Mealy setting. Typically, in practice, realizable specifications only require a small bound value in order to be identified as such [32, 16, 14, 15]. In order to also detect unrealisable specifications quickly, a procedure described in [21, 14] can be applied: using a universal co-Büchi automaton $\overline{\mathcal{A}}$ representing the complement language of $\mathcal{A}$ (i.e., both automata have the same alphabet Σ and we have $\mathcal{L}(\mathcal{A}) = \Sigma^\omega \setminus \mathcal{L}(\overline{\mathcal{A}})$), we execute two copies of the bounded synthesis process. In the first process, we successively increase a bound value b until the game $\mathsf{ToGame}_0(\mathrm{Bound}(\overline{\mathcal{A}}, b), \Sigma_I, \Sigma_O)$ is winning for player 0. In the second copy, we successively increase a bound value of b' until $\mathsf{ToGame}_1(\mathrm{Bound}(\mathcal{A}, b'), \Sigma_I, \Sigma_O)$ is winning for player 1. Once one of the two processes terminates, we know whether $\mathcal{A}$ is realizable over Σ_I/Σ_O (in the Mealy setting) or not. The corresponding procedure for the Moore setting is analogous to the Mealy case.

4.1 Compositional Bounded Synthesis

When performing synthesis, it is sometimes desirable to search for implementations that satisfy several specifications. For the scope of this paper, we only consider a special case of this situation: we search for a Mealy machine for which each word in its language is either accepted by:

- a UCW $\mathcal{A}$, and
- a deterministic safety automaton $\mathcal{A}_s$,

or by

- a deterministic co-safety automaton $\mathcal{A}_c$.

Details on the compositional bounded synthesis approach for more general cases can be found in [15].

Lemma 11. *Given a universal co-Büchi automaton $\mathcal{A}$ with n states, a deterministic safety automaton $\mathcal{A}_s$ with n_s states, and a deterministic co-safety automaton $\mathcal{A}_c$ with n_c states, we can construct a deterministic parity automaton $\mathcal{P}$ with $n_s \cdot n_c \cdot 2n^n n!$ states and $2n$ colours whose language is $(\mathcal{L}(\mathcal{A}) \cap \mathcal{L}(\mathcal{A}_s)) \cup \mathcal{L}(\mathcal{A}_c)$, and a deterministic parity automaton $\mathcal{P}'$ of equal size whose language is $(\mathcal{L}(\mathcal{A}) \cup \mathcal{L}(\mathcal{A}_s)) \cap \mathcal{L}(\mathcal{A}_c)$.*

Proof. We first convert $\mathcal{A}$ to a deterministic parity automaton (Theorem 6) and then use a standard product construction. If the language of $\mathcal{P}$ should be $(\mathcal{L}(\mathcal{A}) \cap \mathcal{L}(\mathcal{A}_s)) \cup \mathcal{L}(\mathcal{A}_c)$, as the states with colour 1 in a safety automaton and the states with colour 0 in a co-safety automaton need to be absorbing, it suffices to assign a state (q, q', q'') in $\mathcal{P}$ with $q \in Q(\mathcal{A}_s)$, $q' \in Q(\mathcal{A}_c)$ and $q'' \in Q(\mathcal{A})$ the colour 0 if $\mathcal{F}(\mathcal{A}_c)(q') = 0$, colour 1 if $\mathcal{F}(\mathcal{A}_s)(q) = \mathcal{F}(\mathcal{A}_c)(q') = 1$, and colour $\mathcal{F}(\mathcal{A})(q'')$ otherwise.

The other case in which $\mathcal{P}'$ should have the language $(\mathcal{L}(\mathcal{A}) \cup \mathcal{L}(\mathcal{A}_s)) \cap \mathcal{L}(\mathcal{A}_c)$ is analogous. □

Corollary 12. *Let $\mathcal{A} = (Q, \Sigma, \delta, Q_{init}, F)$ be a UCW with n states, $\mathcal{A}_s$ be a safety automaton with n_s states, $\mathcal{A}_c$ be a co-safety automaton with n_c states and Σ_I and Σ_O be sets such that $\Sigma = \Sigma_I \times \Sigma_O$. There exists a Moore machine over Σ_I and Σ_O whose induced words are all in $(\mathcal{L}(\mathcal{A}) \cap \mathcal{L}(\mathcal{A}_s)) \cup \mathcal{L}(\mathcal{A}_c)$ if and only if the game $(\mathsf{ToGame}_0(\mathrm{Bound}(\mathcal{A}, b), \Sigma_O, \Sigma_I)||_\vee \mathsf{ToGame}_0(\mathcal{A}_s, \Sigma_O, \Sigma_I))||_\wedge \mathsf{ToGame}_0(\mathcal{A}_c, \Sigma_O, \Sigma_I)$ is winning for* *player* 0 *and* $b = |\mathcal{F}(\mathcal{A})| \cdot n_s \cdot n_c \cdot 2n^n n!$. *Furthermore, there exists a Mealy machine whose induced words are all in* $(\mathcal{L}(\mathcal{A}) \cap \mathcal{L}(\mathcal{A}_s)) \cup \mathcal{L}(\mathcal{A}_c)$ *if and only if the game*

$$(\mathsf{ToGame}_1(\mathrm{Bound}(\mathcal{A}, b), \Sigma_I, \Sigma_O)||_\wedge \mathsf{ToGame}_1(\mathcal{A}_s, \Sigma_I, \Sigma_O))||_\vee \mathsf{ToGame}_0(\mathcal{A}_c, \Sigma_I, \Sigma_O)$$

is winning for player 1 *and* $b = |\mathcal{F}(\mathcal{A})| \cdot n_s \cdot n_c \cdot 2n^n n!$.

Proof. Adapt Corollary 8 to the compositional case using Lemma 11. □

5 Safety and Non-safety: Splitting the Specification

We now turn towards explaining the main novelties of this work. In the previous sections, we have seen how bounded synthesis works in general: we construct a UCW from a specification and build a family of safety automata for a successively increasing bound value. These safety automata are converted to realisability safety games and solved. Once the game is found to be winning for the system player (player 1 if our aim is to synthesize an implementation in form of a Mealy automaton or player 0 for a Moore automaton), the process terminates. To simplify the presentation, for the rest of this paper, we assume that we search for a Mealy-type implementation. The techniques presented here are however of course also applicable for the Moore-type.

We have also seen how to encode the process of safety game solving (as a special case of obligation game solving) in a way that allows using binary decision diagrams (BDDs) as data structure for this task. When doing so, we however face one main problem: when building the safety automata from the UCW and a given bound value, we introduce a lot of *counters* into the structure of the word automaton. Recall that in Definition 9, the state space is defined as $(Q \to \{-\infty, 0, \ldots, b\}) \cup \{\bot\}$, where Q is the set of states of the UCW. Thus, for every state in the original UCW, we introduce a counter ranging from 0 to b with $-\infty$ as an additional value. Also, the definition of the transition function for the safety automata makes use of these counter values. It has been noticed that encoding such number comparisons into BDDs, which is necessary for the solution of the bounded synthesis problem using BDDs in the context of this paper, often leads to huge BDDs in practice [40, 33, 7]. To solve this problem, we introduce two techniques in this paper, of which the first one is described in this section.

In particular, 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 from the introduction of this paper that we assume that the specification is written in the form $\psi = \bigwedge_{a \in A} a \to \bigwedge_{g \in G} g$ for some set of assumptions A and some set of guarantees G, each consisting of safety and non-safety LTL properties. 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* (see Section 2). This has several advantages:

1. It has been observed [14] 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. Additionally, winning strategies for the composite game must also be valid solutions to the synthesis problem.

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

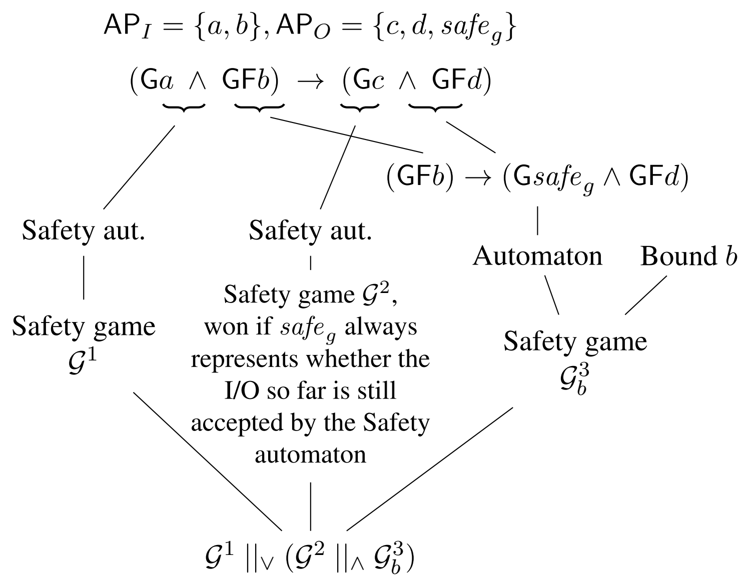
$\mathsf{AP}_I = \{a, b\}, \mathsf{AP}_O = \{c, d, \mathit{safe}_g\}$
$(\mathsf{G}a \wedge \mathsf{GF}b) \rightarrow (\mathsf{G}c \wedge \mathsf{GF}d)$
$(\mathsf{GF}b) \rightarrow (\mathsf{G}\mathit{safe}_g \wedge \mathsf{GF}d)$
Safety aut.
Safety aut.
Automaton
Bound b
Safety game $\mathcal{G}^1$
Safety game $\mathcal{G}^2$, won if safe_g always represents whether the I/O so far is still accepted by the Safety automaton
Safety game $\mathcal{G}^3_b$
$\mathcal{G}^1 \,||_\vee\, (\mathcal{G}^2 \,||_\wedge\, \mathcal{G}^3_b)$

Figure 2: Graphical representation of the specification splitting approach proposed in Section 5 on an example specification consisting of non-safety and safety assumptions and guarantees.

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. Essentially, Corollary 12 establishes this fact.

In [35], the authors propose a method to solve a generalized 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 (i.e., the assumptions and guarantees that happen to be safety properties), strip the non-winning parts (for the system player) from them and compose them with games for the remaining parts of the specification. For the completeness of this methodology, the non-safety assumptions however must not have any effect on the fulfillability of the safety guarantees.[4] In general, we cannot assume this; we thus propose a different method here that is based on introducing a *signalling bit* into the game that links the safety guarantees and the non-safety part of the specification.

We start by splitting the specification $\psi = \bigwedge_{a \in A} a \rightarrow \bigwedge_{g \in G} g$ over the input atomic proposition AP_I and the output atomic propositions AP_O 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* $\psi' = (\bigwedge_{a \in A_n} a) \rightarrow (\bigwedge_{g \in G_n} g \wedge \mathsf{G}(\mathrm{safe}_g))$ 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 $\mathcal{G}$ by defining $\mathcal{G}^3_b = \mathsf{ToGame}_1(\mathsf{Bound}(\mathcal{A}, b), 2^{\mathsf{AP}_I}, 2^{\mathsf{AP}_O})$ and:

$$\mathcal{G} = \mathcal{G}^1 ||_\vee (\mathcal{G}^2 ||_\wedge \mathcal{G}^3_b)$$

Figure 2 visualizes the construction of the composite game. We obtain the following result:

Theorem 13. *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 built from the UCW corresponding to ψ and b' built using the constructions from Definition 1 and Definition 9 is winning for player 1.*

[4]The same problem also occurs in the context of generalized reactivity(1) synthesis [29, 22], an approach that trades the full expressivity of LTL against the possibility to use a simpler and more efficient algorithm solving the synthesis problem.

Proof. Assume that ψ is realisable over $\mathsf{AP}_I/\mathsf{AP}_O$. Then, there exists a Mealy automaton/strategy for player 1 realizing it with k states for some $k \in \mathbb{N}$. Thus, the construction of the game makes Corollary 12 applicable.

On the other hand, it there exists no winning strategy, then by the construction of the game, player 0 can prevent player 1 from playing a winning strategy in $\mathcal{G}$ for every value of b. □

Since $\mathcal{G}$ is the outcome of taking the parallel composition between the games as defined above, its winning condition is of the form $\mathcal{F}(\mathcal{G}) = V' \vee (\neg V'' \wedge \neg V''')$ with $V' = V_1 \times V(\mathcal{G}^2) \times V(\mathcal{G}^3_b)$ for $\mathcal{F}(\mathcal{G}^1) = V_1$, $V'' = V(\mathcal{G}^1) \times V_2 \times V(\mathcal{G}^3_b)$ for $\mathcal{F}(\mathcal{G}^2) = \neg V_2$, and $V''' = V(\mathcal{G}^1) \times V(\mathcal{G}^2) \times V_3$ for $\mathcal{F}(\mathcal{G}^3_b) = \neg V_3$.

Solving the composite game $\mathcal{G}$ is simple using the procedure presented in Section 3. We first simplify the winning condition of the game to $\mathcal{F}(\mathcal{G}) = V' \vee (\neg(V'' \cup V'''))$, and then compute $T(\{V', V'' \cup V'''\})$, $T(\{V'\})$, $T(\{V'' \cup V'''\})$ and $T(\emptyset)$. If the last of these sets contains $v_{init}(\mathcal{G})$, then the game is winning for player 1 and thus the original specification is realisable.

In the context of bounded synthesis, there is however a way to simplify the computation. Let $B_1^F = [\![V']\!]$, $B_2^F = [\![V'']\!]$ and $B_3^F = [\![V''']\!]$, and EnfPre be the enforceable predecessor operator that uses the edge function of the composite game. We can obtain the set of winning positions of player 1 in $\mathcal{G}$ by computing the following BDD:

$$W \;=\; \nu X. X \wedge (B_1^F \vee (\mathsf{EnfPre}(X \wedge (\neg B_2^F) \wedge (\neg B_3^F))))$$

This equation represents operations on BDDs whose application results in a BDD that encodes $\mathsf{Win}_1(V'' \cup V''', V')$. We have $v_{init}(\mathcal{G}) \in \mathsf{Win}_1(V'' \cup V''', V')$ if player 1 has a strategy to either eventually reach one of the vertices in V', or to stay away from $V'' \cup V'''$ forever. Clearly, $\mathsf{Win}_1(V'' \cup V''', V')$ is an under-approximation of the set of vertices in $\mathcal{G}$ from which player 1 can win. However, $\mathsf{Win}_1(V'' \cup V''', V')$ does not classify those vertices as winning from which player 1 cannot avoid visiting $V'' \cup V'''$, but can ensure that eventually V' is visited afterwards (in this case player 1 can also win the game). Since $\mathcal{G}$ is a finite game, there exists an upper bound u on the number of moves that player 1 may require to do so. At the same time, the only way to visit V'' is that player 1 does not choose the right valuation for the $safe_g$ output bit (which can always be avoided), and the only way to visit V''' is to have some counter in $\mathcal{G}^3_b$ exceeding the bound.

Thus, whenever we have $v_{init}(\mathcal{G}) \notin \mathsf{Win}_1(V'' \cup V''', V')$, but v_{init} is winning for player 1 in $\mathcal{G}$, by increasing the bound value b used to construct the game $\mathcal{G}^3_b$ by u, we ensure that in the game $\mathcal{G}$ that we then obtain, player 1 can reach V' before any vertex in $V'' \cup V'''$ is reached from the positions in the game with the smaller bound that $\mathsf{Win}_1(V'' \cup V''', V')$ misclassified as losing for player 1, and thus we have $v_{init} \in \mathsf{Win}_1(V'' \cup V''', V')$ for the increased bound value. So it suffices to use the equation above to compute a set of winning states in $\mathcal{G}$ that eventually contains $v_{init}(\mathcal{G})$ when successively increasing the bound value. Since we do so anyway in the bounded synthesis process, we use this simplification for our implementation to be described in Section 10 as it facilitates the extraction of winning strategies from the game. For the scope of that section, we will refer to W is the set of winning positions.

6 Observations on the Bounded Synthesis Approach

In this section, we describe the second optimisation technique proposed in this paper for performing bounded synthesis efficiently using binary decision diagrams (BDDs).

Consider a UCW as depicted in Figure 3. For a bound of $b = 2$, the corresponding safety automaton $\mathsf{Bound}(\mathcal{A}, b)$ has $5^4 + 1$ states (including the ones that are not reachable from the initial state). While the LTL formula corresponding to the UCW can be decomposed, we describe here a technique that even works without this decomposition, and can be used orthogonally.

In the bounded synthesis approach, any other procedure Bound' to convert a UCW and a bound value b to a safety automaton can be used that makes sure that for every universal co-Büchi word automaton $\mathcal{A}$ and bound value $b \in \mathbb{N}$, we have $\mathcal{L}(\mathsf{Bound}(\mathcal{A}, b)) \subseteq \mathcal{L}(\mathsf{Bound}'(\mathcal{A}, b)) \subseteq \mathcal{L}(\mathcal{A})$. In such a case, the soundness and completeness arguments given in the previous sections still hold. We make use of this fact by deriving such a modified procedure from the following observation: when decomposing the UCW into maximal strongly connected components (SCCs), any run through the UCW can visit every maximal strongly connected component at most once (by the definition of SCCs) and thus, it is possible to interpret the bound value in a modified way: instead of bounding

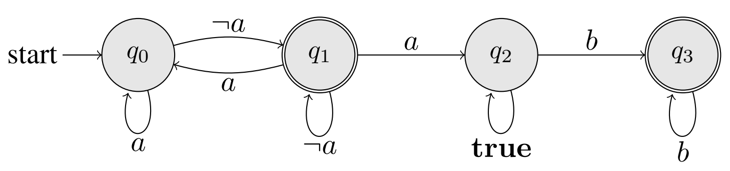

Figure 3: Example UCW for the LTL formula $\mathsf{FG}a \wedge \mathsf{G}((\neg a \wedge \mathsf{X}a) \rightarrow \mathsf{XXGF}\neg b)$ over the set of atomic propositions $\{a, b\}$. Rejecting states are doubly-circled. The part of the automaton consisting of q_0 and q_1 checks that eventually a stops occurring in the input word. Whenever a letter is read that does not contain a, but the subsequent letter contains a, the automaton branches universally from q_1 into q_0 and q_2 at the same time. A suffix run starting from state q_2 then checks if we have $\mathsf{GF}\neg b$ for the following suffix word.

the number of allowed visits to rejecting states along a run, we bound this number for every maximal SCC separately, i.e., we reset the counter whenever a maximal SCC is left. As every maximal SCC can be visited at most once, we do not need to keep track of the counters for SCCs already left in a run. At the same time, whenever a maximal SCC does not have rejecting states, the counter values for the current SCC can only be 0 or $-\infty$ with this modification. We formally define:

Definition 14. *Let $\mathcal{A} = (Q, \Sigma, \delta, Q_{init}, F)$ be a UCW. We define D to be the equivalence relation over Q such that for every $q, q' \in Q$, $(q, q') \in D$ if and only if q and q' are in the same maximal SCC. We furthermore define $K \subseteq Q$ to be the set of states that are not in the same maximal SCC as a rejecting state.*

The deterministic safety automaton $\mathsf{Bound}'(\mathcal{A}, b) = (Q', \Sigma, \delta', \{q'_{init}\}, F')$ for some $b \in \mathbb{N}$ is defined by:

- $Q' = \big(((Q \setminus K) \rightarrow \{-\infty, 0, \ldots, b\}) \times (K \rightarrow \{-\infty, 0\})\big) \cup \{\bot\}$
- *For all $f \in \big(((Q \setminus K) \rightarrow \{-\infty, 0, \ldots, b\}) \times (K \rightarrow \{-\infty, 0\})\big)$, $x \in \Sigma$, we have $\delta'(f, x) \in \big(((Q \setminus K) \rightarrow \{-\infty, 0, \ldots, b\}) \times (K \rightarrow \{-\infty, 0\})\big)$ and*

$$\begin{aligned}\delta'(f,x)(q) = \quad \max \quad & \big(\{f(q') \mid q' \in Q, (q,q') \in D, q \in \delta(q',x)\} \\ \cup \quad & \{0 \mid \exists q' \in Q, (q,q') \notin D, q \in \delta(q',x)\}\big) + \begin{cases} 1 & \text{if } q \in F \\ 0 & \text{otherwise} \end{cases}\end{aligned}$$

if for all $q \in Q$, we have

$$\begin{aligned}b \geq \quad \max \quad & \big(\{f(q') \mid q' \in Q, (q,q') \in D, q \in \delta(q',x)\} \\ \cup \quad & \{0 \mid \exists q' \in Q, (q,q') \notin D, q \in \delta(q',x)\}\big) + \begin{cases} 1 & \text{if } q \in F \\ 0 & \text{otherwise} \end{cases}\end{aligned}$$

and $\delta'(f, x) = \bot$ otherwise. In both equations, we assume that $\max(\emptyset) = -\infty$ and that $-\infty + 1 = -\infty$.

- *For all $x \in \Sigma$, $\delta'(\bot, x) = \{\bot\}$*
- $q'_{init} = \{Q_{init} \mapsto 0, Q \setminus Q_{init} \mapsto -\infty\}$
- $F' = \{Q' \setminus \{\bot\} \mapsto 0, \{\bot\} \mapsto 1\}$

7 Encoding Bounded Synthesis in BDDs

After the discussion of the main ideas presented in this paper, we turn towards filling the remaining blanks in the BDD-based approach to synthesis. In particular, it needs to be discussed how to efficiently symbolically encode the counters in the game $\mathsf{ToGame}_1(\mathsf{Bound}(\mathcal{A}, b), 2^{\mathsf{AP}_I}, 2^{\mathsf{AP}_O})$ and how to deal with $\mathcal{G}^1$ and $\mathcal{G}^2$.

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 game, four groups of BDD variables are needed: two groups for the predecessor and successor game positions in its edge function ($\mathcal{V}_{pre}$ and $\mathcal{V}_{post}$), one for the input to the system ($\mathcal{V}_0$) and one for the output ($\mathcal{V}_1$). 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.

Since our overall game $\mathcal{G}$ is the product of some smaller state spaces, we parallelise the problem and search for state space encodings of $\mathcal{G}^1$, $\mathcal{G}^2$ and $\mathcal{G}^3_b = \mathsf{ToGame}_1(\mathsf{Bound}(\mathcal{A}, b), 2^{\mathsf{AP}_I}, 2^{\mathsf{AP}_O})$ separately.

7.1 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 $\{-\infty, 0, 1, \ldots, b\}$ (or $\{-\infty, 0\}$ for some states when using the optimisation technique from the previous section) 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 $\{-\infty, 0, 1, \ldots, b\}$: we use $\ulcorner \log_2(b+1) \urcorner + 1$ bits. One bit is used for representing whether the value equals $-\infty$, the remaining bits represent the standard binary encoding of the numeral (if given). Taking an extra bit for the $-\infty$ value has the advantage of obtaining smaller BDDs in most cases as this value appears very often in the definition of the transition function. Encoding the range $\{-\infty, 0\}$ on the other hand is trivial as we only need one bit for doing so.

We also propose and use one additional trick. 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 while ensuring that the decision sequence for the play is still winning in the unmodified game, the game semantics remain unchanged. For bounded synthesis, we can thus relax the transition relation (encoded by the BDD B_E) 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 $-\infty$ to some arbitrary other value. This *non-minimality* [3] of the transition relation typically decreases the size of its symbolic encoding.[5]

7.2 The Safety Part

For the encoding of the game components that correspond to safety assumptions and safety 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.

7.2.1 Smart Encoding of Locally Checkable Properties

If an LTL property is of the form $\psi = \mathsf{G}(\phi)$ with a formula ϕ in which the only temporal operator occurring is X, then ψ is a *locally checkable property* [24]. Let k be the deepest nesting of the X operator in ϕ. For checking the satisfaction of such a property when observing a trace, in every round, it suffices to store whether the property has already been violated, the last k inputs/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.

[5]A similar idea was also pursued by Henzinger et al. [20] for simplifying the process of automaton determinisation.

7.2.2 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 given 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 [25], it is not guaranteed that typical procedures for constructing UCW from LTL properties yield automata that have this property. 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.

8 Checking Unrealisability

So far, we have only dealt with the case that we want to prove the realisability of a specification. If a specification is unrealisable, then for no bound $b \in \mathbb{N}$, the game $\mathcal{G} = \mathcal{G}^1 ||_\vee (\mathcal{G}^2 ||_\wedge \mathcal{G}^3_b)$ for $\mathcal{G}^3_b = \mathsf{ToGame}_1(\mathsf{Bound}(\mathcal{A}, b), 2^{\mathsf{AP}_I}, 2^{\mathsf{AP}_O})$ is won by the system player. Thus, an implementation of the approach presented in this paper, which would typically increase the bound successively until the induced game is winning for the system player, would have to increase the bound all the way up to the worst case bound established in Corollary 12. In [14, 21], 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. Then, while in the original realizability question, we search for a Mealy automaton satisfying the specification, we search for a Moore automaton for the environment that witnesses the unrealisability of the specification. 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). Note that this makes the resulting game a safety game, as in this case, we build the composite game by taking $\mathcal{G}' = \mathcal{G}^1 ||_\wedge \mathcal{G}^2 ||_\wedge \mathcal{G}^3_b$. We can thus easily solve $\mathcal{G}'$ symbolically using the procedure from Section 3.

9 Extracting an Implementation in Case of Realizability

Before we discuss the experimental results in the next section, it remains to be described how implementations that realize a given specification can be extracted in case the game $\mathcal{G}$ is found to be winning for the system player during realizability checking.

From a theoretical point of view, extracting a winning strategy from a simple obligation game is not difficult. Recall that at the end of Section 5, we have seen that the set of winning positions in $\mathcal{G}$ can be obtained by evaluating a simple fixed point equation. We can extract a winning strategy by using the position set of the game as state set of the Mealy automaton and choose one successor for every position/input combination that does not lead to leaving the set of winning positions. For positions what witness the violation of a safety assumption or any non-winning position, we can use any successor. From a technical point of view, doing so in a fully symbolic manner, i.e., without enumerating all reachable positions explicitly, is however difficult.

In the former part of this paper, the concrete definition of BDDs was not of relevance, i.e., it sufficed to view them as a data structure for Boolean functions. For the task of extracting an implementation, we need to deviate from this. Assume that V' is the set of winning positions in the game $\mathcal{G}$ and B_E is a BDD representing the edge function of $\mathcal{G}$. Then, a BDD representation of the set of transitions that the Mealy automaton may perform can be obtained by computing $B'_E = \llbracket V' \rrbracket \wedge B_E \wedge \llbracket V' \rrbracket'$ (recall that $\llbracket \cdot \rrbracket$ encodes a set of vertices into $\mathcal{V}_{pre}$ and that $\llbracket \cdot \rrbracket'$ encodes a set of vertices into $\mathcal{V}_{post}$). If in the order of the BDD, the variables in $\mathcal{V}_{pre}$ and $\mathcal{V}_0$ occur first, then the BDD can be thought of as a decision tree with collapsed branches. Such a tree can easily be converted to a switching circuit, using one gate per node in the tree, and one flip-flop per state bit.

However, in practice, the variables can be ordered arbitrarily, as a large share of the power of modern BDD libraries is rooted in the fact that they can perform dynamic variable reordering to keep the sizes of the BDDs small. As a remedy, the algorithm proposed by Kukula and Shiple [23] can be used instead. It is directly applicable to the BDD B'_E, and it generates a circuit whose size is linear in the number of nodes in the BDD. Intuitively, the algorithm works as follows: for every node in the BDD, some switching logic unit is synthesized. These units are connected by the same edges as the BDD nodes. Whenever a new input bit valuation is fed into the Mealy machine, the switching logic starts to propagate which node in the BDD is reachable from the root node for some output. After the $\mathbf{true}$ node has been reached, along such a path, a token is back-propagated in the BDD to the root node, while every switching logic part corresponding to an output variable feeds the chosen value to the output bits and every switching logic part corresponding to a $\mathcal{V}_{post}$ bit feeds the output to a set of flip-flops that preserve the state for the next computation cycle.

In the actual implementation of the approach in this paper, whenever there are multiple output or post-state bit valuations possible in the back-propagation phase, the bit is set to $\mathbf{false}$.

10 Experimental Results

We implemented the symbolic bounded synthesis approach presented in this paper in C++ with the BDD library CUDD v.2.4.2 [36], using dynamic variable reordering. The resulting tool UNBEAST (v.0.6) 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 [17] on the negations of the properties to obtain equivalent universal co-Büchi word automata. As described in Section 7.2.2, 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 5) 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 check for realisability and unrealisabilty of the given specification simultaneously, as described in Section 8. In case of realisability, we extract an implementation that fulfils the specification. We do this in a fully symbolic way, as described in the previous section. The remaining game graph is, together with the specification, converted to a NUSMV [10] model. This allows running NUSMV to verify the correctness of the implementations 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.

10.1 Performance Comparison on the Examples from [21, 14]

We compare UNBEAST with the only other currently publicly available tools for full LTL synthesis, namely Lily v.1.0.2 [21] and ACACIA 2010 [14, 15]. In the following, for ACACIA as well as UNBEAST, 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 [21, 14] 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 by prefixing all references to input variables with a next-time operator. For 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 52.43 seconds (including 40.79 seconds for building the automata). The version of UNBEAST with the features described in this paper had a total running time of about 41.24 seconds. As computing the automata from the specification parts is not a pure preprocessing step in UNBEAST, we do not split up the total running time here.

Interestingly, UNBEAST spends 39.5 out of the 41.24 seconds of overall computation time on showing the unrealisability of a single specification, namely specification no. 4 from [21]. Lily spent only 1.95 seconds here, whereas Acacia needs 2.11 seconds.

In addition to what has been described in this paper, UNBEAST v.0.6 uses one special trick that is common in BDD-based model checking: by grouping the variables in $\mathcal{V}_{pre}$ together with their respective copies in $\mathcal{V}_{post}$, the search space for suitable variable orderings can easily be pruned in a reasonable way. For the fourth specification from [21], however, this optimisation is very malicious: it increases the computation time of UNBEAST to 198.76 seconds, and the computation time for all specifications together to 204.6 seconds. We currently have no explanation for this huge difference, but use this variable grouping feature for the benchmarks given in the next sub-section anyway, as this optimisation is typically non-malicious.

As a summary, the implementation of the approach presented in this paper is typically much faster on the benchmarks from this suite, with the exception of specification number 4, for which the BDD-based approach is not competitive.

10.2 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* that distributes 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 1 gives the running times of our tool and ACACIA for the respective sets of assumptions and guarantees and some numbers of clients $n \in \{2, \ldots, 9\}$. We did not use the compositional techniques introduced in the 2010 version of ACACIA as they are targeted towards specifications that consist of a conjunction of guarantees (and at the same time have no assumptions, or only assumptions that are to be interpreted locally to some guarantees).

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: $\bigwedge_{0 \leq i < n} \mathsf{G}(g_i \rightarrow r_i)$
2. *Guarantee:* A task is only assigned to one server: $\bigwedge_{0 \leq i < n} \mathsf{G}(g_i \rightarrow (\bigwedge_{j \in \{1,\ldots,n\} \setminus \{i\}} \neg g_j))$
3. *Guarantee:* Every server is used infinitely often: $\bigwedge_{0 \leq i < n} \mathsf{GF}(g_i)$

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: $\bigwedge_{0 \leq i < n} \mathsf{GF}(r_i) \rightarrow \mathsf{GF}(g_i)$

Table 1: Running times of ACACIA ("A") and UNBEAST ("U") for the sub-problems defined in Section 10.2 for $n \in \{2, \dots, 9\}$. For each combination of assumptions and guarantees, it is reported whether the specification was realisable (+/-), how many counter bits per state in the UCW were involved at the end of the computation (only for UNBEAST) and how long the computation took (in seconds). For realisable specifications and UNBEAST, we consider the case that an implementation is to be extracted ("+S") and the case that implementation extraction is turned off "-S"). It can easily be seen that implementation extraction appears to be very costly in this approach. We left out the Lily tool as it is not competitive on the load balancing example.

Tool	Setting / # Clients	2	3	4	5	6	7	8	9
A	1	+ 0.3	+ 0.4	+ 0.5	+ 0.9	+ 1.5	+ 2.7	+ 5.0	+ 12.4
U+S		+ 2 0.0	+ 2 0.0	+ 2 0.0	+ 2 0.0	+ 2 0.0	+ 2 0.1	+ 2 0.1	+ 2 0.1
U−S		+ 2 0.1	+ 2 0.0	+ 2 0.0	+ 2 0.0	+ 2 0.0	+ 2 0.0	+ 2 0.1	+ 2 0.1
A	$1 \wedge 2$	+ 0.3	+ 0.3	+ 0.4	+ 0.4	+ 0.6	+ 0.9	+ 1.6	+ 3.1
U+S		+ 2 0.0	+ 2 0.0	+ 2 0.0	+ 2 0.0	+ 2 0.0	+ 2 0.0	+ 2 0.1	+ 2 0.1
U−S		+ 2 0.0	+ 2 0.0	+ 2 0.0	+ 2 0.0	+ 2 0.0	+ 2 0.0	+ 2 0.0	+ 2 0.0
A	$1 \wedge 2 \wedge 3$	- 21.9	- 565.6	timeout	timeout	timeout	memout	memout	timeout
U		- 2 0.1	- 2 0.1	- 2 0.1	- 2 0.1	- 2 0.3	- 2 0.9	- 2 6.7	- 2 74.5
A	$1 \wedge 2 \wedge 4$	+ 0.6	+ 1.3	+ 9.2	+ 274.0	memout	memout	memout	timeout
U+S		+ 2 0.1	+ 3 0.3	+ 3 1.1	+ 4 39.0	+ 4 187.7	timeout	timeout	timeout
U−S		+ 2 0.1	+ 3 0.2	+ 3 0.3	+ 4 2.1	+ 4 2.9	+ 4 10.7	+ 4 42.9	+ 5 387.9
A	$1 \wedge 2 \wedge 4 \wedge 5$	- 164.9	timeout	timeout	timeout	memout	memout	memout	timeout
U		- 2 0.1	- 2 0.6	- 2 886.1	timeout	timeout	timeout	timeout	timeout
A	$6 \rightarrow 1 \wedge 2 \wedge 4 \wedge 5$	- 176.4	timeout	timeout	timeout	memout	memout	memout	timeout
U		- 2 0.1	- 2 0.7	- 2 782.4	timeout	timeout	timeout	timeout	timeout
A	$6 \wedge 7 \rightarrow 1 \wedge 2 \wedge 4 \wedge 5$	- 198.6	memout	memout	timeout	timeout	timeout	timeout	timeout
U		- 2 0.1	- 2 1.6	- 2 780.9	timeout	timeout	timeout	timeout	timeout
A	$6 \wedge 7 \rightarrow 1 \wedge 2 \wedge 5 \wedge 8$	+ 7.4	+ 46.0	memout	memout	timeout	timeout	timeout	timeout
U+S		+ 2 0.1	+ 3 0.3	+ 3 1.2	+ 4 56.7	+ 4 1213.3	timeout	timeout	timeout
U−S		+ 2 0.1	+ 3 0.3	+ 3 0.4	+ 4 3.8	+ 4 4.2	+ 4 16.7	+ 4 77.6	+ 5 961.8
A	$6 \wedge 7 \rightarrow 1 \wedge 2 \wedge 5 \wedge 8 \wedge 9$	- 45.4	- 1087.1	timeout	memout	timeout	timeout	timeout	timeout
U		- 2 0.1	- 2 0.1	- 2 0.2	- 2 0.9	- 2 15.8	- 2 428.8	timeout	timeout
A	$6 \wedge 7 \wedge 10 \rightarrow 1 \wedge 2 \wedge 5 \wedge 8 \wedge 9$	+ 23.2	+ 212.1	memout	timeout	timeout	timeout	timeout	timeout
U+S		+ 2 0.2	+ 2 1.9	+ 3 26.6	+ 3 971.3	timeout	timeout	timeout	timeout
U−S		+ 2 0.2	+ 2 0.7	+ 3 10.9	+ 3 17.2	+ 4 1313.2	timeout	timeout	timeout

5. *Guarantee:* Only jobs that actually exist are assigned:
$\mathsf{G}((\bigvee_{0 \leq i < n} g_i) \rightarrow job)$.

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: $\mathsf{GF}job$

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: $\mathsf{G}(job \wedge (\bigwedge_{0 \leq i < n} \neg g_i) \rightarrow \mathsf{X}(job))$

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: $\bigwedge_{0 \leq i < n} \neg(\mathsf{FG}(r_i \wedge \neg g_i))$

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.

Table 2: Benchmark results for UNBEAST 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	0.0	+	2	0.1	+	2	0.1	+	2	0.1	+	2	0.2	+	2	0.5	+	2	2.3	+	2	9.6
$1 \wedge 2$	+	2	0.0	+	2	0.0	+	2	0.0	+	2	0.0	+	2	0.0	+	2	0.1	+	2	0.1	+	2	0.1
$1 \wedge 2 \wedge 3$	-	2	0.0	-	2	0.0	-	2	0.1	-	2	3.7	timeout			timeout			timeout			timeout		
$1 \wedge 2 \wedge 4$	+	2	0.1	+	3	0.3	+	3	1.1	+	4	12.8	+	4	476.2	timeout			timeout			timeout		
$1 \wedge 2 \wedge 4 \wedge 5$	-	2	0.1	-	2	0.3	-	2	9.6	-	2	1371.2	timeout			timeout			timeout			timeout		
$6 \rightarrow 1 \wedge 2 \wedge 4 \wedge 5$	-	2	0.1	-	2	0.8	-	2	39.1	timeout			timeout			timeout			timeout			timeout		
$6 \wedge 7 \rightarrow 1 \wedge 2 \wedge 4 \wedge 5$	-	2	0.1	-	2	0.7	-	2	29.9	timeout			timeout			timeout			timeout			timeout		
$6 \wedge 7 \rightarrow 1 \wedge 2 \wedge 5 \wedge 8$	+	2	0.3	+	3	17.8	timeout			timeout			timeout			timeout			timeout			timeout		
$6 \wedge 7 \rightarrow 1 \wedge 2 \wedge 5 \wedge 8 \wedge 9$	-	2	0.1	-	2	0.1	-	2	2.8	timeout			timeout			timeout			timeout			timeout		
$6 \wedge 7 \wedge 10 \rightarrow 1 \wedge 2 \wedge 5 \wedge 8 \wedge 9$	+	2	0.4	+	2	63.2	timeout			timeout			timeout			timeout			timeout			timeout		

Table 3: Benchmark results for UNBEAST 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	0.0	+	2	0.0	+	2	0.0	+	2	0.0	+	2	0.0	+	2	0.1	+	2	0.1	+	2	0.1
$1 \wedge 2$	+	2	0.0	+	2	0.0	+	2	0.0	+	2	0.0	+	2	0.0	+	2	0.0	+	2	0.1	+	2	0.1
$1 \wedge 2 \wedge 3$	-	2	0.1	-	2	0.1	-	2	0.1	-	2	0.2	-	2	0.2	-	2	0.9	-	2	6.8	-	2	71.9
$1 \wedge 2 \wedge 4$	+	2	0.1	+	3	3.1	+	3	36.2	timeout			timeout			timeout			timeout			timeout		
$1 \wedge 2 \wedge 4 \wedge 5$	-	2	0.1	-	2	6.2	timeout			timeout			timeout			timeout			timeout			timeout		
$6 \rightarrow 1 \wedge 2 \wedge 4 \wedge 5$	-	2	0.2	-	2	110.5	timeout			timeout			timeout			timeout			timeout			timeout		
$6 \wedge 7 \rightarrow 1 \wedge 2 \wedge 4 \wedge 5$	-	2	0.4	-	2	556.7	timeout			timeout			timeout			timeout			timeout			timeout		
$6 \wedge 7 \rightarrow 1 \wedge 2 \wedge 5 \wedge 8$	+	2	0.1	+	3	0.9	+	3	2.1	+	4	345.7	timeout			timeout			timeout			timeout		
$6 \wedge 7 \rightarrow 1 \wedge 2 \wedge 5 \wedge 8 \wedge 9$	-	2	0.1	-	2	0.1	-	2	0.2	-	2	1.1	-	2	16.1	-	2	432.8	timeout			timeout		
$6 \wedge 7 \wedge 10 \rightarrow 1 \wedge 2 \wedge 5 \wedge 8 \wedge 9$	+	2	0.4	+	2	1.9	timeout			timeout			timeout			timeout			timeout			timeout		

9. *Guarantee*: Server 0 gets a job whenever a job is given and it is ready: $\mathsf{G}((\bigvee_{1 \leq i < n} g_i) \rightarrow \neg r_0)$

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: $\mathsf{G}(g_0 \rightarrow ((\neg job \wedge \neg r_0) \ \mathsf{U} \ (job \wedge \neg r_0)))$.

10.3 The Effect of the Two Main Techniques Proposed in this Paper

To show the effect of the specification splitting technique and the counter reduction technique proposed in this paper (Section 5 and 6, respectively), we also give benchmark results for UNBEAST and the load balancing benchmark with these optimisations switched off. Table 2 contains the results with specification splitting turned off, but non-safety counter reduction switched on. Table 3 contains the results of switching the non-safety counter reduction off, but keeping the specification splitting turned on. Finally, for Table 4, both features are switched off. In all cases, the computation times for realisable specifications include the implementation extraction time. It can be seen that the optimisation techniques proposed are most effective in conjunction.

Table 4: Benchmark results for UNBEAST 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	0.0	+	2	0.0	+	2	0.1	+	2	0.1	+	2	0.2	+	2	0.5	+	2	2.2	+	2	8.5
$1 \wedge 2$	+	2	0.0	+	2	0.0	+	2	0.0	+	2	0.1	+	2	0.1	+	2	0.1	+	2	0.1	+	2	0.1
$1 \wedge 2 \wedge 3$	-	2	0.0	-	2	0.0	-	2	0.1	-	2	3.2	timeout			timeout			timeout			timeout		
$1 \wedge 2 \wedge 4$	+	2	0.1	+	3	3.3	+	3	37.5	timeout			timeout			timeout			timeout			timeout		
$1 \wedge 2 \wedge 4 \wedge 5$	-	2	0.1	-	2	0.9	-	2	181.1	timeout			timeout			timeout			timeout			timeout		
$6 \rightarrow 1 \wedge 2 \wedge 4 \wedge 5$	-	2	0.4	-	2	1614.4	timeout			timeout			timeout			timeout			timeout			timeout		
$6 \wedge 7 \rightarrow 1 \wedge 2 \wedge 4 \wedge 5$	-	2	1.1	-	2	1377.3	timeout			timeout			timeout			timeout			timeout			timeout		
$6 \wedge 7 \rightarrow 1 \wedge 2 \wedge 5 \wedge 8$	+	2	0.3	+	3	409.5	timeout			timeout			timeout			timeout			timeout			timeout		
$6 \wedge 7 \rightarrow 1 \wedge 2 \wedge 5 \wedge 8 \wedge 9$	-	2	0.1	-	2	0.1	-	2	2.7	timeout			timeout			timeout			timeout			timeout		
$6 \wedge 7 \wedge 10 \rightarrow 1 \wedge 2 \wedge 5 \wedge 8 \wedge 9$	+	2	2.2	+	2	24.4	timeout			timeout			timeout			timeout			timeout			timeout		

11 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 in this approach as much as possible. We performed this task by splitting the specification into safety and non-safety parts and presented a counter number reduction technique for the game component corresponding to the non-safety specification conjuncts. We also discussed efficient encodings of the safety part of the specification into games. The experimental results show a huge speed-up compared to previous works.

We only briefly discussed the problem of extracting small implementations 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 [5], 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] Alur, R., Madhusudan, P., Nam, W.: Symbolic computational techniques for solving games. STTT **7**(2), 118–128 (2005)

[2] Andersen, H.R.: Model checking and Boolean graphs. Theor. Comput. Sci. **126**(1), 3–30 (1994)

[3] Bloem, R., A.Cimatti, Pill, I., Roveri, M.: Symbolic implementation of alternating automata. International Journal of Foundations of Computer Science **18**(4), 727–743 (2007)

[4] Bloem, R., Chatterjee, K., Greimel, K., Henzinger, T.A., Jobstmann, B.: Robustness in the presence of liveness. In: T. Touili, B. Cook, P. Jackson (eds.) CAV, *LNCS*, vol. 6174, pp. 410–424. Springer (2010)

[5] 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), 3–16 (2007)

[6] Bloem, R., Galler, S.J., Jobstmann, B., Piterman, N., Pnueli, A., Weiglhofer, M.: Interactive presentation: Automatic hardware synthesis from specifications: a case study. In: R. Lauwereins, J. Madsen (eds.) DATE, pp. 1188–1193. ACM (2007)

[7] Bozga, M., Maler, O., Pnueli, A., Yovine, S.: Some progress in the symbolic verification of timed automata. In: O. Grumberg (ed.) CAV, *LNCS*, vol. 1254, pp. 179–190. Springer (1997)

[8] Bryant, R.E.: Graph-based algorithms for Boolean function manipulation. IEEE Trans. Computers **35**(8), 677–691 (1986)

[9] 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), 142–170 (1992)

[10] 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: E. Brinksma, K.G. Larsen (eds.) CAV, *LNCS*, vol. 2404, pp. 359–364. Springer (2002)

[11] Cleaveland, R., Steffen, B.: A linear-time model-checking algorithm for the alternation-free modal μ-calculus. In: K.G. Larsen, A. Skou (eds.) CAV, *Lecture Notes in Computer Science*, vol. 575, pp. 48–58. Springer (1991)

[12] Drechsler, R., Sieling, D.: Binary decision diagrams in theory and practice. STTT **3**(2), 112–136 (2001)

[13] Farwer, B.: ω-automata. In: E. Grädel, W. Thomas, T. Wilke (eds.) Automata, Logics, and Infinite Games, *Lecture Notes in Computer Science*, vol. 2500, pp. 3–20. Springer (2001)

[14] Filiot, E., Jin, N., Raskin, J.F.: An antichain algorithm for LTL realizability. In: CAV, *LNCS*, vol. 5643, pp. 263–277. Springer (2009)

[15] Filiot, E., Jin, N., Raskin, J.F.: Compositional algorithms for LTL synthesis. In: A. Bouajjani, W.N. Chin (eds.) ATVA, *LNCS*, vol. 6252, pp. 112–127. Springer (2010)

[16] Finkbeiner, B., Schewe, S.: SMT-based synthesis of distributed systems. In: Automated Formal Methods (AFM) (2007)

[17] Gastin, P., Oddoux, D.: Fast LTL to Büchi automata translation. In: CAV, *LNCS*, vol. 2102, pp. 53–65. Springer (2001)

[18] Godhal, Y., Chatterjee, K., Henzinger, T.: Synthesis of AMBA AHB from formal specification: a case study. International Journal on Software Tools for Technology Transfer (STTT) DOI 10.1007/s10009-011-0207-9

[19] Helmert, M., Mattmüller, R., Schewe, S.: Selective approaches for solving weak games. In: S. Graf, W. Zhang (eds.) ATVA, *LNCS*, vol. 4218, pp. 200–214. Springer (2006)

[20] Henzinger, T.A., Piterman, N.: Solving games without determinization. In: Z. Ésik (ed.) CSL, *LNCS*, vol. 4207, pp. 395–410. Springer (2006)

[21] Jobstmann, B., Bloem, R.: Optimizations for LTL synthesis. In: FMCAD, pp. 117–124. IEEE Computer Society (2006)

[22] Klein, U., Pnueli, A.: Revisiting synthesis of GR(1) specifications. In: HVC, *LNCS*, vol. 6504 (2010)

[23] Kukula, J.H., Shiple, T.R.: Building circuits from relations. In: E.A. Emerson, A.P. Sistla (eds.) CAV, *LNCS*, vol. 1855, pp. 113–123. Springer (2000)

[24] Kupferman, O., Lustig, Y., Vardi, M.: On locally checkable properties. In: Logic for Programming, Artificial Intelligence, and Reasoning, pp. 302–316 (2006). DOI 10.1007/11916277_21

[25] Kupferman, O., Vardi, M.Y.: Model checking of safety properties. In: N. Halbwachs, D. Peled (eds.) CAV, *LNCS*, vol. 1633, pp. 172–183. Springer (1999)

[26] Kupferman, O., Vardi, M.Y.: Safraless decision procedures. In: FOCS, pp. 531–542. IEEE (2005)

[27] McMillan, K.L.: Symbolic Model Checking. Kluwer Academic Publishers (1993)

[28] Piterman, N.: From nondeterministic Büchi and Streett automata to deterministic parity automata. Logical Methods in Computer Science **3**(3) (2007)

[29] Piterman, N., Pnueli, A., Sa'ar, Y.: Synthesis of reactive(1) designs. In: E.A. Emerson, K.S. Namjoshi (eds.) VMCAI, *LNCS*, vol. 3855, pp. 364–380. Springer (2006)

[30] Pnueli, A.: The temporal logic of programs. In: FOCS, pp. 46–57. IEEE (1977)

[31] Pnueli, A., Rosner, R.: On the synthesis of an asynchronous reactive module. In: G. Ausiello, M. Dezani-Ciancaglini, S.R.D. Rocca (eds.) ICALP, *LNCS*, vol. 372, pp. 652–671. Springer (1989)

[32] Schewe, S., Finkbeiner, B.: Bounded synthesis. In: K.S. Namjoshi, T. Yoneda, T. Higashino, Y. Okamura (eds.) ATVA, *LNCS*, vol. 4762, pp. 474–488. Springer (2007)

[33] Schneider, K., Logothetis, G.: Abstraction of systems with counters for symbolic model checking. In: M. Mutz, N. Lange (eds.) Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen, pp. 31–40. Shaker, Braunschweig, Germany (1999)

[34] Sistla, A.P.: On characterization of safety and liveness properties in temporal logic. In: PODC, pp. 39–48 (1985)

[35] Sohail, S., Somenzi, F.: Safety first: A two-stage algorithm for LTL games. In: FMCAD, pp. 77–84. IEEE Computer Society (2009)

[36] Somenzi, F.: CUDD: CU decision diagram package, release 2.4.2 (2009)

[37] Thomas, W.: Infinite games and verification (extended abstract of a tutorial). In: E. Brinksma, K.G. Larsen (eds.) CAV, *LNCS*, vol. 2404, pp. 58–64. Springer (2002)

[38] Thomas, W.: Solution of Church's problem: A tutorial. In: K. Apt, R.V. Rooij (eds.) New Perspectives on Games and Interaction. Amsterdam University Press (2008)

[39] Vardi, M.Y., Wolper, P.: Reasoning about infinite computations. Inf. Comput. **115**(1), 1–37 (1994)

[40] Wegener, I.: Branching Programs and Binary Decision Diagrams. SIAM (2000)