

*FP6-IST-507219*

**PROSYD:**

*Property-Based System Design*

Instrument: Specific Targeted Research Project

Thematic Priority: Information Society Technologies

**Research Report on Property Realizability  
(Deliverable 1.2/7)**

Due date of deliverable: April 30, 2006

Actual submission date: May 1, 2006

Start date of project: January 1, 2004

Duration: Three years

Organisation name of lead contractor for this deliverable: ITC-irst

Revision 1.0

<b>Project co-funded by the European Commission within the Sixth Framework Programme (2002-2006)</b>		
<b>Dissemination Level</b>		
<b>PU</b>	Public	<input checked="" type="checkbox"/>
<b>PP</b>	Restricted to other programme participants (including the Commission Services)	<input type="checkbox"/>
<b>RE</b>	Restricted to a group specified by the consortium (including the Commission Services)	<input type="checkbox"/>
<b>CO</b>	Confidential, only for members of the consortium (including the Commission Services)	<input type="checkbox"/>

## **Notices**

For information, contact Marco Roveri [roveri@irst.itc.it](mailto:roveri@irst.itc.it).

This document is intended to fulfil the contractual obligations of the PROSYD project concerning deliverable 1.2/7 described in contract number 507219.

The information in this document is provided "as is", and no guarantee or warranty is given that the information is fit for any particular purpose. The user thereof uses the information at its sole risk and liability.

© Copyright PROSYD 2006. All rights reserved.

# Table of Revisions

Version	Date	Description and reason	By	Affected sections
0.1	February, 2006	Creation	M. Roveri	All
0.2	February, 2006	Initial version of section 2 on definition and state of the art	M. Roveri	Section 2
0.3	February, 2006	Initial version of section 3 on condition for unrealizability	A. Tchaltsev	Section 3
0.4	March, 2006	Initial version of section 4 on condition for realizability	A. Tchaltsev	Section 4
0.5	March, 2006	Revised section 2 and 3	M. Roveri	Section 2, 3
0.6	March, 2006	Initial version section 5	A. Tchaltsev	Section 5
0.7	March, 2006	Revised section 3	M. Roveri	Section 3
0.8	March, 2006	Revised section 3	A. Tchaltsev	Section 3
0.9	March, 2006	Revised sections 3 and 5	M. Roveri	Section 3 and 5
0.10	March, 2006	Revised sections 4 and 5	A. Tchaltsev	Section 4 and 5
0.11	March, 2006	Initial version section 1	M. Roveri	Section 1
0.12	March, 2006	Revised section 5	M. Roveri	Section 5
0.13	March, 2006	Revised section 2	M. Roveri	Section 2
0.14	April, 2006	Global revision and submission to the consortium for feedback	M. Roveri	All
0.15	April, 2006	Global revision to integrate feedback from the consortium	M. Roveri	All
1.0	April, 2006	Final approval	C. Eisner	All

## Authors

Alessandro Cimatti  
Marco Roveri  
Andrei Tchaltsev

## Executive Summary

This document describes the theory of property realizability. Property realizability is the preliminary check before property synthesis which in turn deals with automatically synthesizing a design from a specification given in the linear-time fragment of PSL. In this document we review the theory of realizability and the techniques used so far to tackle this problem. Then we identify sufficient conditions for checking the realizability or unrealizability of a specification and we provide a description of the we identified to verify those conditions. The algorithms can then be implemented using BDD or QBF techniques, both approaches

are discussed. Finally we consider the problem of showing the designer useful debugging information.

## Purpose

The purpose of this document is to describe the research done on property realizability in the PROSYD project. The document provides an overview of different methods aiming to prove realizability or unrealizability of a set of properties without trying to synthesize a design which is out of the scope of this document.

## Intended Audience

This document is intended for researchers working on PSL or a similar specification languages, who are interested in automatic synthesis and in particular in techniques aiming to detect the realizability or unrealizability of a set of properties. It is assumed that readers are familiar with the notions and terms related to PSL, Temporal Logics and their semantics, have a good understanding of model checking, symbolic model checking, of game theory, and of automata theory, including tree automata and alternating automata on infinite words.

## Background

Realizability of linear-time formulas was formalized by Pnueli and Rosner [13]. In their approach there is the need to determinize the automata, which is very expensive and intricate [16]. In this document we propose techniques that avoid the complete methods that assume the specification being realizable and thus aim to synthesize a program. We identify sufficient conditions to prove realizability or unrealizability of a given set of properties and we provide algorithms to check these conditions.

# Contents

Table of Revisions .....	iii
Authors .....	iii
Executive Summary .....	iii
Purpose .....	iv
Intended Audience.....	iv
Background .....	iv
Contents .....	v
Table of Figures .....	vii
Table of Algorithms .....	viii
Glossary .....	ix
1 Introduction .....	1
2 The Realizability Problem .....	3
2.1 Realizability as two player game .....	5
Considerations about initial conditions .....	6
Building the game structure.....	7
2.2 Solving games .....	8
Generalized Reactivity (1) games.....	8
Complete approach.....	9
2.3 Problems .....	9
3 Checking Unrealizability by bounded search .....	11
3.1 Sufficient condition for Unrealizability .....	11
3.2 Checking Unrealizability of Games using BDDs.....	12
3.3 Checking Unrealizability of Games using QBF.....	13
3.4 Approximation in checking Unrealizability using QBF .....	14
3.5 Checking Unrealizability of Büchi Games.....	16
3.6 Condition on initial states.....	17
4 Checking Realizability of Safety Games by induction.....	19
4.1 Environment-deadlock states .....	19
4.2 Fixed point approach to Realizability of Safety Games .....	19
4.3 Approximation in Realizability of Safety Games.....	20
Approximation A .....	21
Approximations B and C.....	22
Approximation D .....	23
4.4 Condition on initial states.....	25

5	Debugging information .....	27
5.1	Useful debugging information.....	27
5.2	Computing the Minimal Unrealizable Core .....	28
5.3	Changing initial conditions.....	29
5.4	Computing environment strategy for unrealizable games.....	30
	The general case .....	30
	Strategy obtained by approximation algorithms .....	30
5.5	Other useful debugging information .....	32
	Input determinism .....	32
	Output causality and requirements redundancy .....	33
	Deadlock of the system and the environment .....	34
6	Conclusions .....	37
7	References.....	39

# Table of Figures

Figure 1 - Example of input and output signals. ....	4
--	---

# Table of Algorithms

Algorithm 1 - Unrealizability check of games using BDDs. ....	12
Algorithm 2 - Unrealizability check of safety games using QBF. ....	13
Algorithm 3 - Unrealizability check using QBF. ....	14
Algorithm 4 - Approximated unrealizability check using QBF. ....	15
Algorithm 5 - Unrealizability check of Büchi games using BDDs. ....	16
Algorithm 6 - Approximated unrealizability check of Büchi games using QBF. ....	17
Algorithm 7 - Realizability of a safety game. ....	20
Algorithm 8 - Check for safe state condition. ....	21
Algorithm 9 - Realizability of a safety game: approximation A. ....	21
Algorithm 10 - Realizability of a safety game: approximation B. ....	23
Algorithm 11 - Realizability of a safety game: approximation C. ....	23
Algorithm 12 - Realizability of a safety game: approximation D. ....	24
Algorithm 13 - Computing the Minimal Unrealizable Core. ....	29
Algorithm 14 - Cone Of Influence of a realizability specification. ....	34
Algorithm 15 - Reachability of deadlock states. ....	35

# Glossary

## **Alternating Tree Automaton**

An automaton with an arbitrary branching mode running on trees.

## **Atomic Proposition**

An atomic proposition of a formula in a propositional logic corresponds to signals in a design or implementation.

## **AWT**

Alternating Weak Tree Automaton is an alternating tree automaton whose states are partitioned into partially ordered sets. Each set is classified as accepting or rejecting. The transition function is restricted so that in each transition, the automaton either stays at the same set or moves to a set smaller in the partial order.

## **BDD**

Binary Decision Diagram. A data structure upon which many model checkers are based.

## **Branching Mode**

The branching mode is a way to classify automata. We distinguish between four branching modes: Deterministic, nondeterministic, universal, and alternating. In a deterministic automaton, the transition function maps from state and letter to a single state. The transition functions of nondeterministic and universal automata map to sets of states. The automata differ in the way they accept an input word or tree. In a nondeterministic automaton the suffix of the word or tree should be accepted by one of the states in the set. In the universal automaton all states in the set have to accept the suffix. An alternating automaton can have nondeterministic and universal edges.

## **BMC**

Bounded model checking. A method of model checking in which a limited number of cycles is examined. Typically, a bounded model checker can falsify, but not verify, a design.

## **Conjunctive Normal Form (CNF)**

A boolean formula is in Conjunctive Normal Form (CNF) if it is a conjunction of clauses, where a clause is a disjunction of literals. A literal is an atomic proposition or its negation.

## **Design**

A hardware netlist (a list of logic gates and their interconnections which make up a circuit) representing the design phase of a chip.

## **Generalized Reactivity (1) formula – GR(1)**

Any PSL or LTL formula of the form

$$\bigwedge_i^m (\mathbf{always\ eventually!} p_i) \Rightarrow \bigwedge_j^n (\mathbf{always\ eventually!} q_j)$$

where  $p_i$  and  $q_j$  are Boolean formulas.

### **Infinite Game**

A finite state machine on which two players, the protagonist and the antagonist, determine the run, by each determining part of the input. The game comes with a winning condition and the task of the protagonist is to make sure that the run satisfies this condition.

### **Language Emptiness**

The language of an automaton is empty iff the automaton accepts no input object (word or tree). That means there is no accepting run for this automaton.

### **LTL**

Linear Temporal Logic or Linear-time temporal logic. LTL is a temporal logic for property specification in formal verification [12].

### **LTL/PSL Game**

An infinite game where the winning condition is given as LTL/PSL formula. All plays in which the sequence of states visited fulfill the given formula are winning for the protagonist. Otherwise the antagonist wins.

### **NBT**

Nondeterministic Büchi Tree Automaton. An alternating tree automaton with Büchi acceptance condition and nondeterministic branching mode.

### **NBW**

Nondeterministic Büchi Word Automaton. An alternating automaton with Büchi acceptance condition and nondeterministic branching mode. The automaton runs on words.

### **PSL**

Property Specification Language[14], the language for specification of designs upon which PROSYD is based.

### **Quantified Boolean Formula (QBF)**

Quantified Boolean Formula Problem (QBF) is a generalization of the Boolean Satisfiability Problem (SAT) in which both existential quantifiers and universal quantifiers can be applied to each atomic proposition.

### **QBF Solver**

QBF Solver is a solver for deciding quantified boolean formulas (QBFs).

### **Realizability**

A given PSL or LTL formula  $\phi$  over a sets of input  $\mathcal{E}$  and output  $\mathcal{S}$  signals is realizable if there exists a strategy  $f : (2^{\mathcal{E}})^* \rightarrow 2^{\mathcal{S}}$  such that all the computations of the system generated by  $f$  satisfy  $\phi$ . Intuitively, a specification is realizable if there exists a system that can respond in such a way that independent of the input values chosen by the environment the combination of inputs and outputs always fulfills the given formula.

### **Safety Property**

A safety property states that something bad should not happen. For instance, “a is never 1 in two consecutive clock ticks.”

## **SAT**

The Boolean Satisfiability Problem (SAT) is a decision problem represented by a boolean expression written using only AND, OR, NOT, atomic propositions, and parentheses. The question is: is there some assignment of TRUE and FALSE values to the atomic propositions that will make the entire expression true?

### **SAT Solver**

A SAT Solver is a solver for deciding the satisfiability of a Boolean Satisfiability Problem (SAT).

### **Synthesis**

The process of automatically generating a design from a given specification. Formally, check if the given specification is realizable and find a witness.

## **UCT**

Universal co-Büchi Word Automaton. An alternating tree automaton with co-Büchi acceptance condition and universal branching mode.

### **Winning Strategy**

A recipe with which a player is guaranteed to win an infinite game, no matter what the other player does. A finite state strategy may depend on a finite memory of the past, i.e., the move the strategy suggests can depend on previous moves of the two players. A memoryless strategy depends only on the current state of the game.



# 1 Introduction

One of the most ambitious and challenging problems in reactive systems construction is the automatic synthesis of programs and (digital) designs from logical specifications, e.g. LTL [12] or PSL [14]. As already discussed in [7] the prevalent approaches to solving the synthesis problem suggest to reduce it to the emptiness problem of tree automata, and view it as the solution of a two-player game.

Pnueli and Rosner in [13] propose a method that starts from a given LTL specification  $\varphi$  and constructs a Büchi automaton  $B_\varphi$ . The Büchi automaton  $B_\varphi$  is then determinized into a deterministic Rabin automaton [16]. This double translation may cause a complexity that is doubly exponential in the size of  $\varphi$ . Once the Rabin automaton is obtained, the game can be solved in time  $n^k$ , where  $n$  is the number of states of the automaton and  $k$  is the number of accepting pairs. The high complexity established in [13] and the intricacy of Safra's determinization construction have caused the synthesis process to be identified as hopelessly intractable and discouraged many practitioners from ever attempting to implement it.

In the PROSYD deliverable [7] it has been thoroughly discussed how to tackle the synthesis of a design from a logical specification given in the linear fragment of PSL. All the approaches described in [7] solve the realizability problem by attempting to synthesize a winning strategy for the system, and during the expensive process of synthesis possibly discovering that the specification is not realizable.

Given the high complexity result of [13] it appears to be of extreme importance to be able to detect in advance, before starting the synthesis process, whether the specification is realizable or not. In this document we aim to identify sufficient conditions for the realizability of a given specification that will allow us to detect the realizability or unrealizability of a specification without starting the real synthesis process.

In Chapter 2 we provide formal definitions of realizability, and we revise the most significant approaches to synthesis from the literature. In Chapter 3 we identify some conditions for unrealizability and we propose some algorithms based on existing technologies to check whether a specification is unrealizable without solving the whole game problem. In Chapter 4 we identify some conditions for realizability and we propose some related algorithms. Finally, in Chapter 5 we discuss the problem of providing the user useful debugging information to allow him to correct the specification or to improve the quality of the specification.



# 2 The Realizability Problem

Given the disjoint sets  $\mathcal{E}$  and  $\mathcal{S}$  of input and output signals respectively, a formula  $\varphi$  expressed in a temporal logic (e.g. LTL or PSL) over atomic propositions on  $\mathcal{E} \cup \mathcal{S}$  specifies the valid behaviors of the component we would like to design. The *realizability problem* consists of checking whether there exists a *program*  $P$  that satisfies  $\varphi$  [13].

Let  $D$  be some non-empty data domain. According to [13] a program  $P$  can be represented as a function  $f_P$  mapping non-empty sequences of  $D$  elements into elements of  $D$ , i.e.  $f_P : D^+ \mapsto D$ . The intuition is that  $f_P$  represents a program with for instance an input variable  $e$  ranging over  $D$  and an output variable  $s$  also ranging over  $D$ , such that at each step  $i = 0, 1, \dots$  the program outputs (assign to  $s$ ) the value  $f_P(e_0, e_1, \dots, e_i)$ , being  $e_j$  the value assumed by variable  $e$  over steps  $j = 0 \dots i$ . In the following we do not distinguish between  $P$  and  $f_P$ .

We define a behavior of the program  $f_P$  as the infinite sequence

$$\sigma : \langle e_0, s_0 \rangle, \langle e_1, s_1 \rangle, \dots$$

such that for every  $i \geq 0$ ,  $e_i, s_i \in D$ , and  $s_i = f_P(e_0, \dots, e_i)$ .

We say that a program  $P$  satisfies a temporal property  $\varphi(e, s)$ , written  $P \models \varphi(e, s)$ , iff every behavior  $\sigma$  of  $P$  satisfies  $\varphi$ , i.e.  $\sigma \models \varphi(e, s)$ .

Given these notions the realizability problem can be clearly stated as follows.

**Definition 1 (Realizability [13])** *Given two distinct sets  $\mathcal{E}$  and  $\mathcal{S}$  of input and output signals respectively, and an LTL formula  $\varphi$  defined over  $(\mathcal{E} \cup \mathcal{S})$ , we say that  $\varphi$  is realizable iff there exists a program  $P$  such that  $P \models \varphi$ .*

Specifications for which such a program exists are called *realizable* or *implementable*. Dually, specification for which such a program does not exist are called *not realizable* or *unrealizable*.

It is common practice to specify the formal specification of a component with two disjoint sets of properties  $\Gamma_A$  and  $\Gamma_G$ .  $\Gamma_A$  specifies the assumptions on the environment, while  $\Gamma_G$  represents what the system has to guarantee if the environment behaves as assumed in  $\Gamma_A$ . Thus, the formula  $\varphi$  to be used is such that:

$$\varphi = \left( \bigwedge_{\varphi_A \in \Gamma_A} \varphi_A \right) \longrightarrow \left( \bigwedge_{\varphi_G \in \Gamma_G} \varphi_G \right)$$

The main question is to find conditions which are necessary and sufficient for the realizability of a given specification.

The satisfiability of the temporal formula  $\varphi$ , that's its *consistency* [15], is a necessary but not sufficient condition for the realizability of  $\varphi$ .

Let us consider the following set of requirements expressed in PSL, where  $\mathcal{E} = \{\text{go}, \text{req}, \text{cancel}\}$  and  $\mathcal{S} = \{\text{grant}\}$ ,  $\Gamma_A = \emptyset$  and  $\Gamma_G = \{(1), (2), (3)\}$ , being (1), (2) and (3) the PSL formulae reported below.

- (1). **always** ( req -> **next\_e**[1:3] grant)
- (2). **always** ( grant -> **next** (!grant))
- (3). **always** ( cancel -> (!grant) **until** go)

This set of requirements is definitely satisfiable. A path that satisfy all of them is in Figure 1.a.

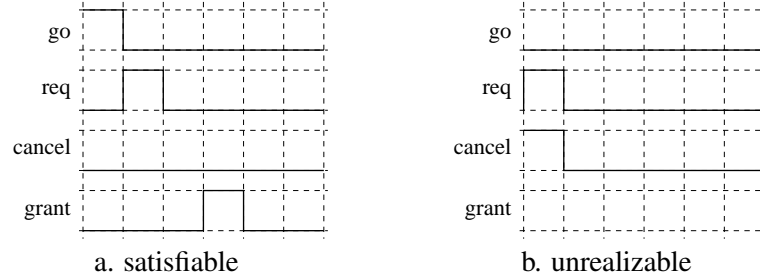


Figure 1: Example of input and output signals.

However, this set of properties is unrealizable, because no matter how we build the program, the environment can enforce a behavior where the signal `go` is always false, the signal `cancel` is issued to true whenever `req` is true, thus violating the above specification (Figure 1.b). That's there is a conflict among formula (1) and (3) when `req` and `cancel` are asserted together and the signal `go` does not cooperate by remaining always false.

This is a consequence of the fact that in constructing the program  $P$ , we cannot control the behavior of the environment.

Below we report Theorem 1 in [13].

**Theorem 1 (Realizability [13])** *Let  $\varphi$  be an LTL specification over the distinct sets of signals  $\mathcal{E}$  and  $\mathcal{S}$ , the following conditions are equivalent:*

- $\varphi$  is realizable.
- The formula  $\forall \mathcal{E}. \exists \mathcal{S}. A\varphi$  is valid over all tree-models.
- The formula  $A\varphi$  holds over a full-x-tree.
- The formula  $A\varphi \wedge AG(\forall e \in \mathcal{E}. \forall v \in D_e. EX(e = v))$  is satisfiable ( $D_e$  is the set of possible data for signal  $e$ ).

Thus, the fact that logical consistency of the formula  $\varphi$  is a necessary condition for the realizability of  $\varphi$  is captured by the following theorem.

**Theorem 2** *If an LTL formula  $\varphi$  is realizable then  $\varphi$  is logically consistent.*

The proof relies on the fact that  $\varphi$  is logically consistent iff  $\exists \mathcal{E}. \exists \mathcal{S}. E\varphi$  is valid, that's if all the tree-models are such that there is one full-path satisfying  $\varphi$ . It is then trivial to show that if  $\forall \mathcal{E}. \exists \mathcal{S}. A\varphi$  is valid, then also  $\exists \mathcal{E}. \exists \mathcal{S}. E\varphi$  is valid.

The prevalent approaches to solve the realizability problem consist in reducing it to the check for language emptiness of a tree automata, and in interpreting it as the existence of a program satisfying the specification. This approach allows to formalize the realizability problem as a two player game among the system we are going to realize and the environment: the system plays against the environment and wins if it produces a correct behaviors. In this framework, checking for realizability amounts to check for the existence of a winning strategy for the system in the corresponding game. The intuition being that a set of requirements is realizable iff it can be implemented for each known environment.

In the following we review the approaches to solve the problem.

---

## 2.1 Realizability as two player game

Using an approach similar to the one proposed in [7], we can formally define a *game structure* as follows.

A *game structure*  $G$  is a tuple  $G = (\mathcal{E}, \mathcal{S}, Q, Q_0, \rho_{\mathcal{E}}, \rho_{\mathcal{S}}, \varphi)$ , where:  $\mathcal{E}$  and  $\mathcal{S}$  are two distinct sets of typed variables representing respectively the variables controlled by the environment and the variables controlled by the system; a *state*  $q$  is an interpretation of  $\mathcal{E} \cup \mathcal{S}$  assigning to each variable  $v \in \mathcal{E} \cup \mathcal{S}$  a value belonging to the corresponding type;  $Q$  is the set of all states; an *assertion* is a Boolean formula over  $\mathcal{E} \cup \mathcal{S}$ ; a state  $q$  satisfies an assertion  $\varphi$  denoted  $q \models \varphi$ , if  $q[\varphi] = \mathbf{true}$ ;  $Q_0$  is an assertion characterizing the initial states;  $\rho_{\mathcal{E}}(\mathcal{E}, \mathcal{S}, \mathcal{E}')$  is the transition relation of the environment. This is an assertion relating a state in  $Q$  to a possible input value in  $\mathcal{E}$ , by referring to unprimed copies of  $\mathcal{E}$ ,  $\mathcal{S}$  and prime copies of  $\mathcal{E}$ ;  $\rho_{\mathcal{S}}(\mathcal{E}, \mathcal{S}, \mathcal{E}', \mathcal{S}')$  is the transition relation of the system. This is an assertion relating a state in  $Q$  to a possible input value in  $\mathcal{E}$  and output value in  $\mathcal{S}$ , by referring to primed and unprimed variables of  $\mathcal{E}$  and  $\mathcal{S}$ ;  $\varphi$  is an LTL or PSL formula identifying the winning condition. In the following we refer to a state  $q$  as a pair  $q = \langle e, s \rangle$  being  $e$  and  $s$  the interpretation of  $\mathcal{E}$  and  $\mathcal{S}$  variables respectively. Hereafter, when clear from the context since a state  $q$  is an interpretation of the inputs and output variables, we write  $\rho(q, q')$  instead of  $\rho(\mathcal{E}, \mathcal{S}, \mathcal{E}', \mathcal{S}')$ .

A *play*  $\sigma$  of  $G$  is a maximal sequence of states  $\sigma : q_0, q_1, \dots$  such that  $q_0 \models Q_0$ , and for each  $j \geq 0$ ,  $q_{j+1}$  is a successor of  $q_j$  (i.e.  $(q_j, q_{j+1}) \models \rho_{\mathcal{E}} \wedge \rho_{\mathcal{S}}$ , where  $(q, q')$  is the joint interpretation which interprets  $u \in \mathcal{E} \cup \mathcal{S}$  as  $s[u]$  and  $u \in \mathcal{E}' \cup \mathcal{S}'$  as  $s'[u]$ ). Let  $G$  be a game structure and  $\sigma$  be a play of  $G$ . From a state  $q$ , the environment chooses an input  $e' \in \mathcal{E}$  such that  $\rho_{\mathcal{E}}(q, e') = 1$  and the system chooses an output  $s' \in \mathcal{S}$  such that  $\rho_{\mathcal{S}}(q, e', s') = \rho_{\mathcal{S}}(q, q') = 1$ .

A play  $\sigma$  is *winning for the system* if it is infinite and if it satisfies  $\varphi$  (i.e.  $\sigma \models \varphi$ ). Otherwise  $\sigma$  is *winning for the environment*.

A *strategy* for the system is a partial function  $f : Q^+ \times \mathcal{E} \mapsto \mathcal{S}$  such that if  $\sigma = q_0, \dots, q_n$  then for every  $e' \in \mathcal{E}$  such that  $\rho_{\mathcal{E}}(q_n, e') = 1$  we have  $\rho_{\mathcal{S}}(q_n, e', f(\sigma \cdot e')) = 1$ . A strategy can be seen as a program that given the history and the current assignments to input variables produces the next assignments for output variables. In the following we use program and strategy as synonyms.

A play  $\sigma = q_0, q_1, \dots$  is said to be *compliant* with strategy  $f$  if for all  $i \geq 0$  we have  $f(q_0, \dots, q_i, q_{i+1}[\mathcal{E}]) = q_{i+1}[\mathcal{S}]$ , where  $q_{i+1}[\mathcal{E}]$  and  $q_{i+1}[\mathcal{S}]$  are the restrictions of  $q_{i+1}$  to variable sets  $\mathcal{E}$  and  $\mathcal{S}$ , respectively.

Strategy  $f$  is *winning* for the system from a state  $q$  if all  $q$ -plays (plays departing from  $q$ ) which are compliant with  $f$  are winning for the system. We denote by  $W_S$  the set of states from which there exists a winning strategy for the system. Dually are defined the notion of *strategy*, *winning strategy*, and the *winning set*  $W_E$  for the environment

A game structure  $G$  is said to be *winning* for the system if the initial states are not empty and all initial states are winning for the system:

$$\emptyset \neq Q_0 \subseteq W_S$$

Otherwise, the game is said to be winning for the environment.

A *safety game* is a game where the winning condition is a safety property, that's property stating that something bad should not happen.

## Considerations about initial conditions

In the definition above, a game structure  $G$  is winning for the system if *all* the initial states are winning for the system. Intuitively this means that initial states are chosen by the environment, i.e. if there exists at least one state satisfying initial condition and not winning for the system then the environment can always start a play from this state and not allow the system to win.

If the system is granted the right to choose an initial state, then the definition of a winning game has to be changed. Namely, a game structure  $G$  is *winning* for the system if *at least one* of the initial states is winning for the system.

In an alternative setting, both the system and the environment can be able to control initial values of only their own variables, respectively. In this situation there should be two initial conditions  $Q_0^{\mathcal{E}}(e)$  and  $Q_0^{\mathcal{S}}(e, s)$ . Choosing initial values for its variables the environment moves first and has to satisfy  $Q_0^{\mathcal{E}}$ . After that the system chooses initial values for the system variables such that  $Q_0^{\mathcal{S}}$  is satisfied. The definition of a winning game will then become the following: a game structure  $G$  is *winning* for the system if for all the initial values of the environment variables satisfying  $Q_0^{\mathcal{E}}$  there exist such values for the system variables that the obtained initial state satisfies  $Q_0^{\mathcal{S}}$  and is winning for the system.

It has to be noticed that, the later setting requires a modification of the game structure itself to consider the two initial conditions in its body,  $Q_0^{\mathcal{E}}$  for the environment and  $Q_0^{\mathcal{S}}$  for the system, i.e.  $G = (\mathcal{E}, \mathcal{S}, \mathcal{Q}, Q_0^{\mathcal{E}}, Q_0^{\mathcal{S}}, \rho_{\mathcal{E}}, \rho_{\mathcal{S}}, \phi)$ . In this last setting we

require that  $\forall e.(Q_0^E(e) \rightarrow \exists s.Q_0^S(e,s))$ , thus capturing the case where the system has no possible initial states given the choice of the environment.

Even though from practical point of view all these definitions of game structure and thus of winning conditions can be useful, for the sake of presentation unless explicitly stated we assume a game structure with one unique assertion specifying the initial condition, we assume the system being winning if all initial states are winning, and finally we assume the environment moves first. We will discuss how the proposed techniques to solve the realizability problem can be adapted to capture the different setting here discussed.

## Building the game structure

Given an LTL specification  $\varphi = (\bigwedge_{\phi_A \in \Gamma_A} \phi_A) \longrightarrow (\bigwedge_{\phi_G \in \Gamma_G} \phi_G)$ , being  $\Gamma_A$  the assumptions on the environment, and  $\Gamma_G$  representing what the system will guarantee if the environment behaves as assumed in  $\Gamma_A$ , and two distinct sets  $\mathcal{E}$  and  $\mathcal{S}$  of typed variables respectively controlled by the environment and by the system, we can construct a game structure as follows (see also [7]).

Let us assume the sets  $\Gamma_A$  (similarly  $\Gamma_G$ ) can be further decomposed in three disjoint subsets  $\Gamma_A^I, \Gamma_A^T, \Gamma_A^W$  such that  $\Gamma_A = \Gamma_A^I \cup \Gamma_A^T \cup \Gamma_A^W$ ,  $\phi \in \Gamma_A^I$  iff  $\phi$  is a propositional formula characterizing the initial states of the design;  $\phi \in \Gamma_A^T$  iff  $\phi$  is a formula of the form  $\phi = \mathbf{always} \psi$  where  $\psi$  is a Boolean expressions over variables in  $\mathcal{E} \cup \mathcal{S}$  and expressions of the form  $\mathbf{next} v$  where  $v \in \mathcal{E}$  (in the case of  $\Gamma_G^T$ , we require  $v \in \mathcal{E} \cup \mathcal{S}$ ); finally  $\Gamma_A^W = \Gamma_A \setminus (\Gamma_A^I \cup \Gamma_A^T)$ , and  $\Gamma_G^W = \Gamma_G \setminus (\Gamma_G^I \cup \Gamma_G^T)$ .

The game structure  $G_\varphi = (\mathcal{E}, \mathcal{S}, \mathcal{Q}, Q_0^E, Q_0^S, \rho_E, \rho_S, \varphi_G)$ , can thus be constructed starting from  $\varphi$  in such a way that  $Q_0^E = \bigwedge_{\psi \in \Gamma_A^I} \psi$ ;  $Q_0^S = \bigwedge_{\psi \in \Gamma_G^I} \psi$ ;  $\rho_E = \bigwedge_{\psi \in \Gamma_A^T} \tau(\psi)$ ;  $\rho_S = \bigwedge_{\psi \in \Gamma_G^T} \tau(\psi)$ ;  $\varphi_G = (\bigwedge_{\alpha \in \Gamma_A^W} \alpha) \rightarrow (\bigwedge_{\alpha \in \Gamma_G^W} \alpha)$ . Where  $\tau$  is a translation replacing each instance of  $\mathbf{next} v$  by  $v'$ .

The game structure  $G_\varphi = (\mathcal{E}, \mathcal{S}, \mathcal{Q}, Q_0, \rho_E, \rho_S, \varphi_G)$  with one unique initial condition can be easily obtained from  $G_\varphi = (\mathcal{E}, \mathcal{S}, \mathcal{Q}, Q_0^E, Q_0^S, \rho_E, \rho_S, \varphi_G)$  with separated initial condition by enforcing  $Q_0 = (Q_0^E \wedge Q_0^S)$ .

The construction can even more be pushed by substituting each formula  $\phi \in \Gamma_A^W$  with a corresponding deterministic Büchi automaton, thus adding the initial condition and the transition relation of the Büchi automaton to  $Q_0$  and  $\rho_E$  respectively, and using the acceptance condition as formula to replace  $\phi$ . For instance, for “**always**( $p \rightarrow \mathbf{eventually!} q$ )” we add  $x \leftrightarrow 1$  to  $Q_0$ , we add  $x' \leftrightarrow (q \vee x \wedge \neg p)$  to  $\rho_E$ , and we replace “**always**( $p \rightarrow \mathbf{eventually!} q$ )” with “**always eventually! x**”. Similar consideration apply also to any formula in  $\Gamma_G^W$ . As claimed in [8] it is not difficult to prove that this is a sound transition (see [8] for more details).

Once the game structure has been built, it is necessary to choose the interpretation regarding the initial states and as consequence use the corresponding algorithm to solve the problem.

## 2.2 Solving games

The prevalent approaches to solve the realizability problem are those that aim to solve the synthesis problem, which consists in constructing a winning strategy for the system in a two player game. These approaches assume that the system is realizable and attempt to build a winning strategy. If they fail in building it, then the specification the game refers to is not realizable. Here we list some of the approaches described in [7]. Then we analyze the problems that affect these approaches.

### Generalized Reactivity (1) games

Generalized Reactivity (1) games are a subset of LTL and PSL games where the winning condition is of the form:

$$\varphi = \left( \bigwedge_{i=1}^m \text{always eventually! } J_i^1 \right) \rightarrow \left( \bigwedge_{j=1}^n \text{always eventually! } J_j^2 \right)$$

For such kind of games, the set of winning states can be computed with the following  $\mu$ -calculus formula (We refer the reader to [7, 8] for more details):

$$W_S = \nu \begin{bmatrix} Z_1 \\ Z_2 \\ \vdots \\ \vdots \\ Z_n \end{bmatrix} \left[ \begin{array}{c} \mu Y \left( \bigvee_{i=1}^m \nu X (J_i^2 \wedge \textcircled{Z}_2 \vee \textcircled{Y} \vee \neg J_i^1 \wedge \textcircled{X}) \right) \\ \mu Y \left( \bigvee_{i=1}^m \nu X (J_i^2 \wedge \textcircled{Z}_3 \vee \textcircled{Y} \vee \neg J_i^1 \wedge \textcircled{X}) \right) \\ \vdots \\ \mu Y \left( \bigvee_{i=1}^m \nu X (J_n^2 \wedge \textcircled{Z}_1 \vee \textcircled{Y} \vee \neg J_i^1 \wedge \textcircled{X}) \right) \end{array} \right]$$

Where,  $j \oplus 1 = (j \bmod n) + 1$ , and  $\textcircled{A} = \{ \langle e, s \rangle \mid \forall e'. \rho_{\mathcal{E}}(e, s, e') \rightarrow \exists s'. \rho_{\mathcal{S}}(e, s, e', s') \wedge \langle e', s' \rangle \in A \}$ , i.e. a state  $q = \langle s, e \rangle$  is included in  $\textcircled{A}$  if the system can force the play to reach a state in  $A$  in one step, that's regardless of how the environment moves in  $q$ , the system can choose an appropriate move leading into  $A$ .

As claimed in [7, 8] this approach can solve realizability of LTL formulas in the form that we discussed before in polynomial (cubic) time. In particular, given sets of variables  $\mathcal{E}, \mathcal{S}$  whose set of possible valuations is  $\Sigma$  and an LTL formula  $\varphi$  with  $m$  and  $n$  conjuncts, it is possible to determine using a symbolic algorithm whether  $\varphi$  is realizable in time proportional to  $(nm \mid \Sigma \mid)^3$ .

## Complete approach

The *complete* approach able to deal with a generic LTL or PSL formula  $\varphi$  has been proposed in [13]. In [13], first a nondeterministic Büchi automaton  $B_\varphi$  which recognizes all the  $\omega$ -words satisfying the LTL formula  $\varphi$  is constructed. This automaton is then determinized into a deterministic Rabin automata using the Safra construction [16]. Finally, the resulting automaton is interpreted as a game and it is checked for language emptiness: if the language is not empty the computation provides a witness, which corresponds to a correct implementation of the given specification. The game is solved in time  $n^k$  being  $k$  the number of accepting pairs, and  $n$  the number of states of the Rabin automaton. The double translation results in a complexity which is double exponential in the size of the original formula  $\varphi$  [13]. Apart from the complexity results described in [13], the main difficulty in this approach is the intricacy of the Safra determinization construction.

Recently in [11] and also discussed in [7] has been presented a novel approach avoiding the Safra construction to solve the synthesis problem. In this new automata based approach first, a nondeterministic Büchi word automaton for  $\neg\varphi$  is constructed and translated into a universal co-Büchi tree automaton that recognizes all trees containing only paths that satisfy  $\varphi$ . This translation demands the first exponent of the complexity bound. Then, the tree automaton is translated into an alternating weak tree automaton, from which a nondeterministic Büchi tree automaton is built. The latter translation causes the second exponential blow-up. Finally, language emptiness for this nondeterministic Büchi automaton is computed. If the language is not empty the computation provides a witness, which corresponds to a correct implementation of the given specification.

---

## 2.3 Problems

The approaches described previously and also presented in [7] can tackle the problem of realizability and synthesis for the linear fragment of PSL. They are complete, in the sense that if the specification is not realizable, then they can detect this since they are not able in this case to synthesize a winning strategy and thus a corresponding digital design. They are monolithic, that's they consider the whole specification without even considering that the problem of unrealizability can for instance reside in particular parts of the specification (e.g. in the constraints specifying the initial conditions). These approaches assume the specification being realizable and try to synthesize a winning strategy since those algorithms have been thought with synthesis in mind. Moreover, even though the Generalized Reactivity (1) approach [8] is cubic in the size of the specification, and the complete approach can avoid the use of the intricate Safra construction [16] the doubly exponential complexity is still there.

Thus, it turns out to be of extreme importance to be able to detect a priori, before starting the synthesis process, whether the specification is realizable or unrealizable. These checks can also be incomplete, in the sense that they can provide the

specification writer a better understanding of the possibility the specification being realizable, not the certainty. Once he or her is confident the specification being realizable, he or she can start the computationally heavy process of synthesis.

For these reasons we investigate in the following sufficient conditions for realizability and unrealizability that will be of help to the specification writer to detect early and efficiently whether the specification is realizable or unrealizable without applying the complete approaches aiming to synthesize the design.

Last but not least, whenever a specification turns out to be unrealizable, the approaches aiming to synthesize the design are usually not able to provide the specification writer useful debugging information on the reason why the specification is unrealizable. They can be used to synthesize a winning strategy for the environment that can be used in an interactive simulation by the specification writer. However, since the starting point is a specification, the specification would benefit of debugging information in terms of specification, as to make it easy to detect where the problem resides. In the following we will also investigate this aspect, and we will try to identify what would be possible useful debugging information.

# 3 Checking Unrealizability by bounded search

The previous section showed that the realizability problem has the same complexity as the synthesis problem: in the worst case it is doubly exponential. Thus, it is of great importance to find sufficient conditions that will allow to detect that the specification is unrealizable before really starting the synthesis.

In this section we provide some sufficient conditions for unrealizability.

---

## 3.1 Sufficient condition for Unrealizability

Let us consider a game structure  $G = (\mathcal{E}, \mathcal{S}, \mathcal{Q}, Q_0, \rho_{\mathcal{E}}, \rho_{\mathcal{S}}, \varphi)$ . As we have seen in previous section all the plays induced by a winning strategy must be infinite. Thus, the environment can make the system lose by enforcing finite runs, e.g. by forcing the system to reach a state from which the system cannot move anymore, that's the environment can force the system to a deadlock state. This intuition can be formalized and generalized by saying that a game  $G$  is *unrealizable* if the set of initial states  $Q_0$  is empty, or if for *some* initial state  $\langle e_0, s_0 \rangle \in Q_0$  the environment has a *strategy* to force the system to reach a *system-deadlock* state in  $k$  steps.

**Definition 2 (System-Deadlock States)** For a game structure  $G = (\mathcal{E}, \mathcal{S}, \mathcal{Q}, Q_0, \rho_{\mathcal{E}}, \rho_{\mathcal{S}}, \varphi)$ , the set of system-deadlock states  $DL_{\mathcal{S}}$  is the set of those states from which the environment can choose such values for its variables that the system will not be able to satisfy its transition relation  $\rho_{\mathcal{S}}$ .

$$DL_{\mathcal{S}} = \{ \langle e, s \rangle \mid \exists e'. \rho_{\mathcal{E}}(e, s, e') \wedge \forall s'. \rho_{\mathcal{S}}(e, s, e', s') \rightarrow \emptyset \}$$

The following theorem captures the fact that the ability of the environment to force a system-deadlock state in a finite number of steps is sufficient condition for unrealizability.

**Theorem 3** A game  $G = (\mathcal{E}, \mathcal{S}, \mathcal{Q}, Q_0, \rho_{\mathcal{E}}, \rho_{\mathcal{S}}, \varphi)$  is unrealizable if  $Q_0 = \emptyset$  or if there exists  $k \geq 0$  such that for some initial state the environment has a strategy to force the system to reach a system-deadlock state in  $k$  steps. That's one among (1) and (2) conditions below is satisfied.

$$\forall e_0 s_0. \langle e_0, s_0 \rangle \notin Q_0 \tag{1}$$

$$\begin{aligned}
& \exists k \geq 0. \exists e_0 s_0. \langle e_0, s_0 \rangle \in Q_0, \text{ such that} \\
& \exists e_1. \rho_E(e_0, s_0, e_1) \wedge \forall s_1. \rho_S(e_0, s_0, e_1, s_1) \rightarrow \\
& \dots \\
& \exists e_k. \rho_E(e_{k-1}, s_{k-1}, e_k) \wedge \forall s_k. \rho_S(e_{k-1}, s_{k-1}, e_k, s_k) \rightarrow \\
& \langle e_k, s_k \rangle \in DL_S
\end{aligned} \tag{2}$$

If  $k = 0$  this means that one of the initial state is a system-deadlock state.

---

## 3.2 Checking Unrealizability of Games using BDDs

The unrealizability problem of a game  $G$  can be solved using BDD techniques. If all the constraints and transition relations are represented in the form of BDD then the algorithm to check unrealizability of a game  $G$  can be written as a function:

```

1 function IsUnrealizableBdd( $G$ )
2   if ( $Q_0 = \emptyset$ ) then return true;
3   else
4      $Deadlock = \emptyset$ ;
5     while ( $Deadlock \wedge Q_0 = \emptyset$ ) do
6        $Deadlock =$ 
7          $Deadlock \vee \exists e'. \rho_E(e, s, e') \wedge \forall s'. \rho_S(e, s, e', s') \rightarrow Deadlock'$ 
8     done
9     return true;
10  fi;
11 end

```

Algorithm 1: Unrealizability check of games using BDDs.

This function returns *true* if the specification is unrealizable. In the body the first condition to be checked is the emptiness of initial states, which is the first condition that makes the problem unrealizable. *Deadlock* is a predicate over variables  $e$  and  $s$ , representing the states from which the system can be forced to reach system-deadlock states in  $DL_S$ , i.e. from which the system cannot guarantee to run infinitely long. *Deadlock'* is obtained from *Deadlock* by substituting all variables  $e$  and  $s$  by  $e'$  and  $s'$ , respectively. Initially *Deadlock* is empty set. At very iteration a new state  $\langle e, s \rangle$  is added to *Deadlock* if there are such values for environment variables that for any values of system variables satisfying  $\rho_S$  the system always reaches *Deadlock* states computed so far. Note that after first iteration the value of *Deadlock* is  $DL_S$ . The algorithm stops as soon as *Deadlock* states include any of the initial states  $Q_0$ .

As it can be seen from definition (1) and (2) the above algorithm returns *true* only if the problem is unrealizable because the environment can force the system into a system-deadlock state.

If the environment cannot force the system into a system-deadlock state the algorithm as described does not terminate. This situation is very similar to the Bounded Model Checking [5] where the design is assumed to be buggy and the search aims to prove this by looking for a counterexample. However, if no counterexample is found BMC cannot infer whether the design is not buggy.

To make the algorithm terminate it is necessary to check at each iteration whether *Deadlock* changes. If no new states are added to *Deadlock* then we can exit from the while loop and return *unknown* to indicate that we cannot say that the game is unrealizable.

For a safety game the above algorithm can be slightly modified to make it complete. If at line 4 the initial value for the *Deadlock* is  $\neg A$ , where  $A$  is the safety condition, then the algorithm will compute all the states from which the environment can eventually force a step to  $\neg A$  or  $DL_S$  states. Therefore, the algorithm will return *true* if and only if the game is unrealizable.

```

1 function IsSafetyUnrealizableBdd( $G$ )
2   if ( $Q_0 = \emptyset \vee Q_0 \not\subseteq A$ ) then return true;
3   else
4      $Deadlock = \neg A$ ;
5     while ( $Deadlock \wedge Q_0 = \emptyset$ ) do
6        $Deadlock =$ 
7          $Deadlock \vee \exists e'. \rho_E(e, s, e') \wedge \forall s'. \rho_S(e, s, e', s') \rightarrow Deadlock'$ 
8     done
9     return true;
10  fi;
11 end

```

Algorithm 2: Unrealizability check of safety games using QBF.

---

### 3.3 Checking Unrealizability of Games using QBF

The algorithm described in the previous section can be written in the form suitable for being submitted to a QBF solver such as YQUAFFLE [17], SKIZZO [2], QUANTOR [4], QUBE [9].

Let's define the formulas  $Deadpath[k](e_0, s_0, e_1, s_1, \dots, e_k, s_k)$  as

$$\begin{aligned}
Deadpath[1](e_0, s_0, e_1, s_1) &= \rho_E(e_0, s_0, e_1) \wedge (\rho_S(e_0, s_0, e_1, s_1) \rightarrow \emptyset) \\
Deadpath[2](e_0, s_0, e_1, s_1, e_2, s_2) \\
&= \rho_E(e_0, s_0, e_1) \wedge (\rho_S(e_0, s_0, e_1, s_1) \rightarrow Deadpath[1](e_1, s_1, e_2, s_2)) \\
&\dots \\
Deadpath[k](e_0, s_0, \dots, e_k, s_k) \\
&= \rho_E(e_0, s_0, e_1) \wedge (\rho_S(e_0, s_0, e_1, s_1) \rightarrow Deadpath[k-1](e_1, s_1, \dots, e_k, s_k))
\end{aligned}$$

Intuitively,  $Deadpath[k](e_0, s_0, e_1, s_1, \dots, e_k, s_k)$  represents a  $k$ -step run of a game from a state  $\langle e_0, s_0 \rangle$  to a state where the system violates  $\rho_S$ .

To solve unrealizability problem it is necessary to check whether the initial states are empty, i.e.

$$\forall e_0 s_0. \neg Q_0(e_0, s_0) \quad (3)$$

or whether the environment can force the system to reach the system-deadlock state from some of the initial states  $Q_0$ . So at every iteration  $k$  (beginning from 1) the following formula is computed:

$$\begin{aligned} &\exists e_0 s_0. \exists e_1 \forall s_1. \exists e_2 \forall s_2. \dots \exists e_k \forall s_k. \\ &Q_0(e_0, s_0) \wedge Deadpath[k](e_0, s_0, e_1, s_1, \dots, e_k, s_k) \end{aligned} \quad (4)$$

The operation is performed until the formula becomes *true* which means that the problem is unrealizable.

Note that the above formula defines exactly the same formulas as formulas (1) and (2) on page 12 with the only difference that all quantifiers are moved outside of the propositional part of the formulas and put at the beginning.

Thus the algorithm described in previous section can be implemented using a QBF solver instead of BDDs as in algorithm 3.

```

1 function IsUnrealizableQbf(G)
2   if ( $\forall e_0, s_0. \neg Q_0$ ) then return true; fi
3   foreach (k in 0, 1, ...) do
4     if ( $\exists e_0 s_0.$ 
5        $\exists e_1 \forall s_1.$ 
6          $\exists e_2 \forall s_2.$ 
7           ...
8              $\exists e_k \forall s_k$ 
9              $Q_0(e_0, s_0) \wedge Deadpath[k](e_0, s_0, \dots, e_k, s_k)$ 
10          ) then return true; fi
11   done
12 end

```

Algorithm 3: Unrealizability check using QBF.

---

## 3.4 Approximation in checking Unrealizability using QBF

In QBF solving usually the more alternations of quantifiers are in a formula – the more difficult it is to solve the formula. Trying to optimize checking the unrealizability, one of the possible approaches can be to approximate the formula (4) by moving all existential quantifiers outside. So the formula become

$$\begin{aligned} &\exists e_0 s_0. \exists e_1 \exists e_2 \dots \exists e_k. \forall s_1 \forall s_2 \dots \forall s_k. \\ &Q_0(e_0, s_0) \wedge Deadpath[k](e_0, s_0, \dots, e_k, s_k) \end{aligned} \quad (5)$$

If this formula is *true* then the environment have a sequence  $e_1, e_2, \dots, e_k$  of values such that independent of the system moves the environment at every step  $k$  can consecutively choose values  $e_k$  for its variables thereby forcing the system to system-deadlock states.

Solving the formula (5) can be easier than solving (4). Moreover, (5) can be converted to a simpler SAT problem. Assume that  $\phi(x, y)$  is boolean formula in CNF form. Any formula of the form  $\forall y. \phi(x, y)$  is equivalent to  $\phi'(x)$ , where  $\phi'$  is obtained from  $\phi$  by removing all literals  $y$  and  $\neg y$  from all clauses (note that an empty clause is false). Therefore, (5) can be easily converted to a pure SAT problem with  $k$  less variables.

The formula (5) is a coarse approximation of (4) because the environment has to make all its moves without knowing what moves the system has already made. If the approximation does not give an answer, a finer (but more difficult to solve) approximation can be tried. For that let's allow the environment to choose in advance only  $k/2$  moves instead of all  $k$  moves as in (5). So the formula becomes:

$$\exists e_0 s_0. \exists e_1 \exists e_2 \dots \exists e_{k/2}. \forall s_1 \forall s_2 \dots \forall s_{k/2}. \exists e_{k/2+1} \dots \exists e_k. \forall s_{k/2+1} \dots \forall s_k. \quad (6)$$

$$Q_0(e_0, s_0) \wedge Deadpath[k](e_0, s_0, \dots, e_k, s_k)$$

If the formula is still not *true* the number of quantifier alternations can be increased. In such a way the formula may degenerate to the original form (4).

This alternative algorithm with approximation to solving the unrealizability problem using QBF solvers can be represented as algorithm 4.

```

1 function IsUnrealizableQbfApproximation(G)
2   foreach (k in 0, 1, ...) do
3     foreach (i in k, k/2, k/4, ..., 1) do
4       if ( $\exists e_0 s_0. \exists e_1 \dots \exists e_i. \forall s_1 \dots \forall s_i.$ 
5          $\exists e_{i+1} \dots \exists e_{2*i}. \forall s_{i+1} \dots \forall s_{2*i}.$ 
6         ...
7          $\exists e_{k-i+1} \dots \exists e_k. \forall s_{k-i+1} \dots \forall s_k.$ 
8          $Q_0(e_0, s_0) \wedge Deadpath[k](e_0, s_0, \dots, e_k, s_k)$ 
9       ) then return true; fi;
10    done
11  done
12 end

```

Algorithm 4: Approximated unrealizability check using QBF.

The external loop increases the number of steps the game can make to reach the system-deadlock state. The internal loop tries to solve different approximations of formula (4), beginning from the most coarse and easy to solve (the formula (5)) and gradually increasing the number of quantifier alternations. In the worst case the original formulas (4) has to be solved.

### 3.5 Checking Unrealizability of Büchi Games

Let us consider a Büchi game  $G = (\mathcal{E}, \mathcal{S}, \mathcal{Q}, \mathcal{Q}_0, \rho_{\mathcal{E}}, \rho_{\mathcal{S}}, \varphi)$  where  $\varphi$  defines a set of fairness conditions  $F_0, \dots, F_n$ , each of which should be satisfied infinitely many times during the game plays. The system can lose the Büchi game if from all states of at least one of the fairness conditions  $F_i$  the environment can force the system to reach a system-deadlock state. This sufficient condition for unrealizability can be captured with the following theorem.

**Theorem 4** *A Büchi game  $G = (\mathcal{E}, \mathcal{S}, \mathcal{Q}, \mathcal{Q}_0, \rho_{\mathcal{E}}, \rho_{\mathcal{S}}, \varphi)$  where  $\varphi$  defines a set of fairness conditions  $F_0, \dots, F_n$  is unrealizable if*

$$\begin{aligned}
 & \exists k \geq 0. \\
 & \exists j. 0 \leq j < n. \\
 & \forall e_0 s_0. F_j(e_0, s_0) \rightarrow \\
 & \quad \exists e_1. \rho_{\mathcal{E}}(e_0, s_0, e_1) \wedge \forall s_1. \rho_{\mathcal{S}}(e_0, s_0, e_1, s_1) \rightarrow \\
 & \quad \dots \\
 & \quad \exists e_k. \rho_{\mathcal{E}}(e_{k-1}, s_{k-1}, e_k) \wedge \forall s_k. \rho_{\mathcal{S}}(e_{k-1}, s_{k-1}, e_k, s_k) \rightarrow \\
 & \quad \langle e_k, s_k \rangle \in DL_{\mathcal{S}}
 \end{aligned} \tag{7}$$

All the algorithms previously introduced can be easily modified for checking the truth of the formula (7). It is just necessary to add a most internal loop that for a given  $k$  at every iteration  $j$  will check whether all states satisfying  $F_j$  may lead to system-deadlock states. The BDD algorithm 1 described on page 12 adapted to deal with the above theorem is algorithm 5.

```

1 function IsBuchiUnrealizableBdd(G)
2   if ( $Q_0 = \emptyset$ ) then return true;
3   else
4     Deadlock =  $\emptyset$ ;
5     while ( $Deadlock \wedge Q_0 = \emptyset$ ) do
6       foreach (i in  $0, \dots, n$ ) do
7         if ( $F_i \wedge Deadlock = F_i$ ) then
8           return true;
9         fi
10      done
11      Deadlock =
12         $Deadlock \vee \exists e'. \rho_{\mathcal{E}}(e, s, e') \wedge \forall s'. \rho_{\mathcal{S}}(e, s, e', s') \rightarrow Deadlock'$ 
13    done
14    return true;
15  fi;
16 end

```

Algorithm 5: Unrealizability check of Büchi games using BDDs.

While, the corresponding algorithm based on QBF approximation described on page 14 will be algorithm 6.

```

1 function IsBuchiUnrealizableQbfApproximation(G)
2   foreach (k in 0, 1, ...) do
3     foreach (i in k, k/2, k/4, ..., 1) do
4       if ( $\exists e_0 s_0. \exists e_1 \dots \exists e_i. \forall s_1 \dots \forall s_i.$ 
5          $\exists e_{i+1} \dots \exists e_{2^*i}. \forall s_{i+1} \dots \forall s_{2^*i}.$ 
6         ...
7          $\exists e_{k-i+1} \dots \exists e_k. \forall s_{k-i+1} \dots \forall s_k.$ 
8          $Q_0(e_0, s_0) \wedge \text{Deadpath}[k](e_0, s_0, \dots, e_k, s_k)$ 
9       ) return true; fi
10    foreach (j in 0, 1, ..., n) do
11      if ( $\forall e_0 s_0. \exists e_1 \dots \exists e_i. \forall s_1 \dots \forall s_i.$ 
12         $\exists e_{i+1} \dots \exists e_{2^*i}. \forall s_{i+1} \dots \forall s_{2^*i}.$ 
13        ...
14         $\exists e_{k-i+1} \dots \exists e_k. \forall s_{k-i+1} \dots \forall s_k.$ 
15         $F_j(e_0, s_0) \wedge \text{Deadpath}[k](e_0, s_0, \dots, e_k, s_k)$ 
16      ) then return true; fi
17    done
18  done
19 done
20 end

```

Algorithm 6: Approximated unrealizability check of Büchi games using QBF.

---

### 3.6 Condition on initial states

In the framework considered so far, the environment chooses an initial state since a game is unrealizable for the system if at least from one initial state the environment can force the system to reach a system-deadlock state. Thus in the condition at the initial states in formula (2) the quantifiers of both the system and the environment variables are existential.

In an alternative setting, as explained in section 2.1, the system may be able to choose an initial state. In this case, the problem becomes unrealizable only if from *all* initial states the environment can force the system to reach the system-deadlock states. In this setting, in the formula (2) the line specifying the condition on the initial state

$$\exists k \geq 0. \exists e_0 s_0. \langle e_0, s_0 \rangle \in Q_0, \text{ such that}$$

will become

$$\exists k \geq 0. \forall e_0 s_0. \langle e_0, s_0 \rangle \in Q_0, \text{ implies}$$

Similarly, the condition at line 5 in the function *IsUnrealizableBdd* at page 12 will become **while** ( $\text{Deadlock} \wedge Q_0 \neq Q_0$ ).

In the QBF approach, the quantifiers for  $e_0$  and  $s_0$  will be fixed to  $\forall e_0 \forall s_0$ . Thus, in the approximated QBF approach these quantifiers have to be shifted together with all other universal quantifiers in the formula. The propositional part of the formulas will be modified to

$$Q_0(e_0, s_0) \rightarrow \text{Deadpath}[k](e_0, s_0, \dots, e_k, s_k)$$

An alternative definition of the game structure may assume the initial condition  $Q_0$  being splitted in two constraints  $Q_0^E(e)$  and  $Q_0^S(e,s)$  specifying respectively the initial condition for the environment and for the system. In this setting, at first, the environment chooses the initial values for its variables  $e$  such that  $Q_0^E(e)$  is satisfied. Then, the system chooses its part of the initial state such that  $Q_0^S(e,s)$  is satisfied. In this case the quantifiers on initial values  $e_0$  and  $s_0$  will be  $\exists e_0 \forall s_0$  and the condition on the initial state will be  $Q_0^E \wedge (Q_0^S \rightarrow \text{Deadpath})$ . Again all the above algorithms can be easily modified to capture this definition. For instance, the condition at line 5 in the function *isUnrealizableBdd* at page 12 will become **while**  $(\neg \exists e. Q_0^E(e) \wedge (\forall s. Q_0^S(e,s) \rightarrow \text{Deadlock}))$ .

# 4 Checking Realizability of Safety Games by induction

In this section we focus on safety games, and we provide specialized algorithms to answer the realizability problem for all those specifications that can be reduced to safety games.

By the definition a safety game  $G = (\mathcal{E}, \mathcal{S}, Q, Q_0, \rho_{\mathcal{E}}, \rho_{\mathcal{S}}, \varphi)$  with the winning condition  $\varphi$  specified by the safety condition  $A$  is a game such that each state of plays has to satisfy  $A$ .

---

## 4.1 Environment-deadlock states

In this section we assume that the environment transition relation  $\rho_{\mathcal{E}}$  is a partial function, and therefore the environment may have *environment-deadlock* states which are the counterpart of the notion of system deadlock states. The formal definition of environment-deadlock states is as follows.

**Definition 3 (Environment-Deadlock)** For a game  $G = (\mathcal{E}, \mathcal{S}, Q, Q_0, \rho_{\mathcal{E}}, \rho_{\mathcal{S}}, \varphi)$ , the set of environment-deadlock states  $DL_{\mathcal{E}}$  is defined as

$$DL_{\mathcal{E}} = \{\langle e, s \rangle \mid \neg \exists e'. \rho_{\mathcal{E}}(e, s, e')\}$$

Intuitively this means that in an environment-deadlock state  $\langle e, s \rangle \in DL_{\mathcal{E}}$  the environment cannot make a move, i.e. it cannot find an  $e'$  such that  $\rho_{\mathcal{E}}(e, s, e')$  holds.

We assume that all the environment-deadlock states are winning for the system, since the environment “dies” in these states. As result the definition of a play *winning for the system* given on page 5 has to be extended. In this setting a play is also winning for the system if it is *finite*, satisfies winning condition  $\varphi$  and its last state is an environment-deadlock state.

---

## 4.2 Fixed point approach to Realizability of Safety Games

The algorithm 2 on page 12 can be easily adapted to check for realizability of safety game  $G$  as to obtain algorithm 7.

```

1 function IsSafetyGameRealizableBdd(G)
2   if ( $Q_0 = \emptyset \vee Q_0 \not\subseteq A$ ) return false; fi
3   Safe = A;
4   while ( $Q_0 \rightarrow \textit{Safe}$ ) do
5     Safeprevious = Safe;
6     Safe =  $\textit{Safe} \wedge \forall e'. \rho_{\mathcal{E}}(e, s, e') \rightarrow \exists s'. \rho_{\mathcal{S}}(e, s, e', s') \wedge \textit{Safe}'$ 
7     if ( $\textit{Safe}_{\textit{previous}} = \textit{Safe}$ ) return true; fi
8   done
9   return false;
10 end

```

Algorithm 7: Realizability of a safety game.

In this algorithm, the variable *Safe* is the complement of *Deadlock* in the original algorithm, and initially equal to *A*, i.e. a set of allowed states. If at some iteration some of the initial states  $Q_0$  is not in *Safe* then for this initial states the environment can force the system to reach the system-deadlock states or states satisfying  $\neg A$ . This means that the game is unrealizable, and therefore *false* is returned. Otherwise, at some iteration the value of *Safe* will not change with respect to the previous iteration and thus a fixed point has been reached. At the end of the loop the value of *Safe* will include all the states which are environment-deadlock or from which regardless of the environment moves the system can always force a step back to *Safe* states, i.e. stay infinitely long in states satisfying *A*. In this case the game is realizable, and therefore, *true* is returned.

For safety games the algorithm 7 will always terminate and report whether the game is realizable or unrealizable.

---

### 4.3 Approximation in Realizability of Safety Games

The algorithm 7 relies on the computation of a fixed point and it computes *all* states where the system wins, i.e. *all* those states where the system can avoid system-deadlock and  $\neg A$  states infinitely long or force the environment to environment-deadlock states.

To check the realizability of a safety game there is no need to compute the whole set of system winning states. Indeed, a subset may be enough. Let us define the *safe states set*  $\textit{SafeStates} \subseteq A$  such a subset of states that independent of the environment moves the system can stay in this subset infinitely long, or force the environment to environment-deadlock states.

**Definition 4 (Safe States)** For a game  $G = (\mathcal{E}, \mathcal{S}, \mathcal{Q}, Q_0, \rho_{\mathcal{E}}, \rho_{\mathcal{S}}, \Phi)$ , the system safe states set *SafeStates* is defined as

$$\textit{SafeStates} = \{ \langle e, s \rangle \in A \mid \forall e'. \rho_{\mathcal{E}}(e, s, e') \rightarrow \exists s'. \rho_{\mathcal{S}}(e, s, e', s') \wedge \langle e', s' \rangle \in \textit{SafeStates} \}$$

If the set of states  $SafeStates$  exists and includes all initial states  $Q_0$  then the game is realizable.

**Theorem 5** For a safety game  $G_A$ , if exists  $SafeStates \neq \emptyset$  and  $Q_0 \subseteq SafeStates$ , then  $G_A$  is realizable.

The set of safe states can potentially be given from outside: a user may provide this set as a guess; or some external tool using specific knowledge about the problem may infer the set of states. An externally provided candidate for  $SafeStates$  can easily be checked for being an actual system safe states set by the following function:

```

1 function IsSafeStates( $G, SafeStates$ )
2   return  $SafeStates \subseteq A \wedge$ 
3      $\forall e, s. SafeStates \rightarrow \forall e'. (\rho_E(e, s, e') \rightarrow \exists s'. \rho_S(e, s, e', s') \wedge SafeStates')$ 
4 end

```

Algorithm 8: Check for safe state condition.

where  $SafeStates'$  is obtained from  $SafeStates$  by substituting variables  $s'$  and  $e'$  for  $s$  and  $e$ , respectively.

The algorithm 7 computes the *greatest SafeStates*. In the rest of this section we describe and analyze several algorithms aiming to compute *some* such  $SafeStates$  set.

## Approximation A

The first approximation algorithm (algorithm 9) builds the set  $SafeStates$  by adding at every iteration  $k$  such states that regardless of environment moves the system go back to the starting state in  $k$  steps. In this algorithm  $Safe$  plays the role of the

```

1 function IsSafetyGameRealizableBdd_A( $G$ )
2   if ( $Q_0 = \emptyset \vee Q_0 \not\subseteq A$ ) return false; fi
3    $Safe = \emptyset$ ;
4    $Path = e^* = e \wedge s^* = s$ ;
5   while ( $Q_0 \not\subseteq Safe$ ) do
6      $Path_{previous} = Path$ ;
7      $Path = A \wedge \forall e'. \rho_E(e, s, e') \rightarrow \exists s'. \rho_S(e, s, e', s') \wedge (Safe' \vee Path')$ 
8     if ( $Path_{previous} = Path$ ) return unknown; fi
9      $Safe = Safe \vee \exists \langle e^*, s^* \rangle. e^* = e \wedge s^* = s \wedge Path$ 
10  done
11  return true;
12 end

```

Algorithm 9: Realizability of a safety game: approximation A.

candidate safe states set. The candidate set of system safe states is initialized to be the empty set. At every iteration  $k$  the set  $Path$  is a set of pairs of states  $\langle\langle e, s \rangle, \langle e^*, s^* \rangle\rangle$  satisfying the safety condition  $A$  and such that state  $\langle e, s \rangle$  is the environment-deadlock state, or independent of the environment moves the system can go from state  $\langle e, s \rangle$  to state  $\langle e^*, s^* \rangle$  in  $k$  steps, or in fewer number of steps reach already computed  $Safe$  sets. The set  $Safe$  is a set of states  $\langle e, s \rangle$ . At every iteration  $Safe$  is augmented by such states that the system can go back to the same state in  $k$  steps or in one step reach  $Safe$  computed at previous iteration. The sets  $Safe'$  and  $Path'$  are obtained from  $Safe$  and  $Path$ , respectively, by substituting variables  $s'$  and  $e'$  for  $s$  and  $e$ , respectively.

As soon as  $Safe$  includes initial states  $Q_0$  the algorithm terminates with *true* since the game is realizable. Otherwise at some iteration the fixed point in the computation of  $Path$  will be reached, which means that the algorithm was not able to find a system safe states set big enough to encompass initial states, and it is unknown whether the game is realizable or not.

The apparent weakness of this algorithm is that when the loops in the game plays are computed it is assumed then the system has to return *exactly* to the same state the loop starts from. In specifications the system does not usually have the full control and thus it cannot guarantee which *individual* states are reached in a next step, but only which *subset* of states can be reached. As result, the algorithm can often be too weak to find a big enough set of safe states.

## Approximations B and C

The Theorem 5 says that a safety game  $G_A$  is realizable if the found set of safe states encompasses all the initial states. We can exploit this theorem considering the set of initial states as a candidate for a safe states set. Then iteratively some new states can be added to the candidate until the set built in this way satisfies the condition of being a safe states set.

Different heuristics can be thought of for adding new states to the candidate system safe states set. Among those, one of the possible way consists in adding those states from which the system for sure reaches the initial states. The intuition behind this approach is that if a safe states set is augmented with new states from which the system can reach the given safe states set then the set remains to be a system safe states set. The algorithm that uses this heuristic is the algorithm 10 below.

The weakness of this algorithm is that it can show the realizability of a game only if starting in any of the initial states the system can always go back to initial states. This is quite a strong constraint for specification, and therefore the algorithm may not be able to deal with many problems.

Algorithm 10 can be strengthened by considering new states for the candidate not only from those states from which the system can reach the initial states, but also those states that allow the system to make a one step loop. The line 5 in algorithm 10 must be modified as follows

$$NewStates = A \wedge \forall e'. \rho_E(e, s, e') \rightarrow \exists s'. \rho_S(e, s, e', s') \wedge (Safe' \vee (s = s' \wedge e = e'));$$

```

1 function IsSafetyGameRealizableBdd_B(G)
2   if ( $Q_0 = \emptyset \vee Q_0 \not\subseteq A$ ) return false; fi
3   Safe =  $Q_0$ ;
4   while ( $\neg \text{IsSafeStates}(G, \text{Safe})$ ) do
5     NewStates =  $A \wedge \forall e'. \rho_{\mathcal{E}}(e, s, e') \rightarrow \exists s'. \rho_{\mathcal{S}}(e, s, e', s') \wedge \text{Safe}'$ ;
6     Safeprevious = Safe;
7     Safe =  $\text{Safe} \vee \text{NewStates}$ ;
8     if ( $\text{Safe}_{\text{previous}} = \text{Safe}$ ) return unknown; fi
9   done
10  return true;
11 end

```

Algorithm 10: Realizability of a safety game: approximation B.

An even stronger solution can be obtained by the combination of the methods “A” and “B”. At each iteration  $k$  the candidate for a safe states set will be augmented with states from which the system can reach the initial states in  $k$  steps or make a loop in  $k$  steps.

```

1 function IsSafetyGameRealizableBdd_C(G)
2   if ( $Q_0 = \emptyset \vee Q_0 \not\subseteq A$ ) return false; fi
3   Safe =  $Q_0$ ;
4   Path =  $e^* = e \wedge s^* = s$ ;
5   while ( $\neg \text{IsSafeStates}(G, \text{Safe})$ ) do
6     Pathprevious = Path;
7     Path =  $A \wedge \forall e'. \rho_{\mathcal{E}}(e, s, e') \rightarrow \exists s'. \rho_{\mathcal{S}}(e, s, e', s') \wedge (\text{Safe}' \vee \text{Path}')$ 
8     if ( $\text{Path}_{\text{previous}} = \text{Path}$ ) return unknown; fi
9     NewStates =  $\exists \langle e^*, s^* \rangle. e^* = e \wedge s^* = s \wedge \text{Path}$ 
10    Safe =  $\text{Safe} \vee \text{NewStates}$ ;
11  done
12  return true;
13 end

```

Algorithm 11: Realizability of a safety game: approximation C.

This method still have the drawback inherent in method A, i.e. to perform a loop run outside of initial states the system has to be able to go back exactly to the same state the loop starts from, which may be too constrictive to determine realizability of most practical specifications.

## Approximation D

A further alternative to the previous methods consists in constructing a safe states set “optimistically” and then in removing the states which violates the definition of a safe states set.

Let’s starts with the initial states as a candidate *Safe* for the system safe states set. At every iteration this set will be augmented with all possible successors. It is

quite likely that the obtained set will not be a safe states set. Then it is necessary to removes “bad” states, i.e. the states from which the environment can force a step outside or system-deadlock states:

$$Bad = Safe \wedge \exists e'. \rho_E(e, s, e') \wedge \forall s'. \rho_S(e, s, e', s') \rightarrow \neg Safe'$$

It is also necessary to remove all the states from which the environment can reach the bad states. If the obtained set still has all the initial states then the game is realizable. Otherwise the removed states are restored and the iteration is repeated. The algorithm implementing this heuristic is the algorithm 12 below.

```

1 function IsSafetyGameRealizableBdd_D(G)
2   if ( $Q_0 = \emptyset \vee Q_0 \not\subseteq A$ ) return false; fi
3   Safe =  $Q_0$ ;
4   Safeprevious =  $\emptyset$ ;
5   while ( $Safe \neq Safe_{previous}$ ) do
6      $Bad = Safe \wedge \exists e'. \rho_E(e, s, e') \wedge \forall s'. \rho_S(e, s, e', s') \rightarrow \neg Safe'$ 
7     Badprevious =  $\emptyset$ ;
8     while ( $Bad \neq Bad_{previous}$ ) do
9       Badprevious = Bad;
10       $NewBad = Safe \wedge \exists e'. \rho_E(e, s, e') \wedge \forall s'. \rho_S(e, s, e', s') \rightarrow Bad'$ ;
11      Bad =  $Bad \vee NewBad$ ;
12    done;
13    if ( $Bad \wedge Q_0 = \emptyset$ ) return true; fi
14     $NewSafe = \exists e', s'. Safe' \wedge \rho_E(e', s', e) \wedge \rho_S(e', s', e, s) \wedge A$ ;
15    Safeprevious = Safe;
16    Safe =  $Safe \vee NewSafe$ ;
17  done
18  return false;
19 end

```

Algorithm 12: Realizability of a safety game: approximation D.

As in previous programs, *Bad'* and *Safe'* are obtained from *Bad* and *Safe*, respectively, by substituting *e'* and *s'* for *e* and *s*. Note that when the successors for *Safe* are computed at line 14 of algorithm 12 current  $\langle e, s \rangle$  and next states  $\langle e', s' \rangle$  are switched in the transition relations  $\rho_E$  and  $\rho_S$ . The internal loop computes all the “bad” states for a given *Safe*, and when the fixed point is reach and bad states still do not include any of initial states then the game is realizable. The external loop increases the *Safe*, i.e. the states reachable from initial states  $Q_0$ . If the fixed point has been reached then all reachable states have been checked and there is no safe states set big enough. As result for some initial states the environment always can force the system to a system-deadlock state of a state satisfying  $\neg A$ , i.e. the game is unrealizable.

This algorithm always terminates and returns *true* if the game is realizable, and *false* if the game is unrealizable similarly to the algorithm 7 on page 20.

However, the two algorithms differ. Algorithm 12 performs the search of the system safe states set forward, while the algorithm 7 proceeds backwards.

Let’s assume that for a particular safety game there exists a relatively small safe states set around, or equal to, the initial states, and both algorithms try to find

out whether the game is realizable. The algorithm 7 will have to compute all the system safe states and the number of iterations will be equal to the path length to a system-deadlock or  $\neg A$  state from the farthest state. The algorithm 12, on the other hand, computes reachable states and potentially reaches the safe states set instantly or in a few number of iterations. Then the internal loop removes all the states which violate the definition of a safe states set; operation that can potentially be performed in a small number of steps.

For these reasons, we can claim that the algorithm 12 can be more efficient than the algorithm 7 in determining the realizability of a safety game if it is expected that having a big number of states in total there exists a relatively small set of safe states reachable from the initial states in small number of steps.

---

## 4.4 Condition on initial states

The same way as in section 3 the algorithms described in this section assume that the initial states are chosen by the environment, i.e. the game is realizable only if from all the initial states not leaving  $A$  the system can survive infinitely long or force the environment to reach the environment-deadlock states. As it is explained in section 2.1, depending on the specification the system also may have the rights to choose the initial states or the system and the environment each chooses subparts of initial states. The original algorithm 7, algorithms “A” and “D” of this section can be easily adapted for any of such definitions.

If it is the system who chooses the initial states then condition on initial states (line 2 in the algorithms) will be

**if**  $(Q_0 \wedge A = \emptyset)$  **return** *false*; **fi**

The condition controlling the loop in the original algorithm 7 on page 20 will become

**while**  $(Q_0 \wedge Safe \neq \emptyset)$  **do**

Similar way the loop condition of the algorithm “A” on page 21 will be

**while**  $(Q_0 \wedge Safe = \emptyset)$  **do**

In algorithm “D” it will be necessary to change the line returning *true*:

**if**  $(Q_0 \wedge \neg Bad \neq \emptyset)$  **return** *true*; **fi**

Similar way, if the initial states are constrained by two constraints  $Q_0^E(e)$  and  $Q_0^S(e, s)$  (i.e. at first, the environment chooses the initial values for its variables  $e$  such that  $Q_0^E(e)$  is satisfied, and then the system chooses its part of the initial state such that  $Q_0^S(e, s)$  is satisfied) then the algorithms change the following way.

In all the algorithms the condition on the initial states (line 2) will be

**if**  $(\exists e.Q_0^E \wedge \forall s.\neg(Q_0^S \wedge A))$  **return false; fi**

The loop condition in the original algorithm 7 will be

**while**  $(\forall e.Q_0^E \rightarrow \exists s.Q_0^S \wedge Safe)$  **do**

In algorithm “A” the condition on the loop will become

**while**  $(\exists e.Q_0^E \wedge \forall s.Q_0^S \rightarrow \neg Safe)$  **do**

In algorithm “D” the line returning *true* has to be changed to:

**if**  $(\forall e.Q_0^E \rightarrow \exists s.Q_0^S \wedge \neg Bad)$  **return true; fi**

and  $Q_0$  will mean  $Q_0^E \wedge Q_0^S$ .

# 5 Debugging information

In Verification and in particular in Model Checking, whenever a design does not satisfy a specification a counterexamples is produced. The counterexamples is an execution of the design that shows why the system does not satisfy the corresponding property, and thus can be used by the designer to correct the design.

In previous sections we described several algorithms to check for (un)realizability, however the problem of producing and then showing the designer useful debugging information that can be used to correct the specification is still open. It has to be noticed that it is very important to show a specification writer the reason why the specification he wrote is unrealizable.

The aim of this section is to identify possible debugging information for the case of realizability and describe how to show them to the user in the most effective way.

---

## 5.1 Useful debugging information

For a generic game structure deriving from a given specification  $\Gamma$ , to demonstrate that the specification is unrealizable, that's there is no winning strategy for the system to win the corresponding game, we can generate a strategy for the environment. That's we can run any of the algorithms developed for solving the synthesis problem with the goal of synthesizing a strategy for the environment. Once this strategy for the environment has been synthesized, it can be interactively simulated by asking the designer to set the values for the system variables, or alternatively this strategy can be used to generate traces that satisfy the environment but that violate the system specification. Thus, in this setting the strategy plays the role of counterexamples in verification.

The major drawback of this approach is that the results, i.e. the strategy for the environment, may be not useful for helping the designer understand where the problem is in the specification. Indeed, the specification is a set of properties, and the designer may expect the debugging information being at the property level.

In Propositional satisfiability when a formula is unsatisfiable, it is often required to find a propositional *unsatisfiable core* – that is, a small unsatisfiable subset of the formula's clauses. Localizing a small unsatisfiable core is necessary to determine the underlying reasons for the failure. A *minimal unsatisfiable core* is a such unsatisfiable core that it becomes satisfiable whenever any one of its clauses is removed.

In Section 2 it has been shown that a necessary condition for realizability is that the specification must be logically consistent. Thus, if the specification is not consistent then similarly to the pure propositional case we can identify an *unsatisfiable core* and a *minimal unsatisfiable core*. For an inconsistent specification  $\varphi$  given in the form of assumptions on the environment  $\Gamma_A$  and guarantees of the system  $\Gamma_G$ , i.e.

$$\varphi = \left( \bigwedge_{\varphi_A \in \Gamma_A} \varphi_A \right) \longrightarrow \left( \bigwedge_{\varphi_G \in \Gamma_G} \varphi_G \right)$$

an *unsatisfiable core* is a such set  $\Gamma'_G \subseteq \Gamma_G$  that a specification  $\varphi'$

$$\varphi' = \left( \bigwedge_{\varphi_A \in \Gamma_A} \varphi_A \right) \longrightarrow \left( \bigwedge_{\varphi_G \in \Gamma'_G} \varphi_G \right)$$

is still inconsistent. A *minimal unsatisfiable core* is an unsatisfiable core such that removing any  $\varphi_G \in \Gamma'_G$  from  $\Gamma'_G$  would make  $\varphi'$  consistent. Intuitively,  $\Gamma'_G$  represents a subset of the specification that cause unsatisfiability and thus also unrealizability.

Note that an unsatisfiable core consists only of the components of  $\Gamma_G$  and does not take into account  $\Gamma_A$ . This is done so because weakening the assumptions on the environment, i.e. removing components of  $\Gamma_A$ , cannot make an unsatisfiable (unrealizable) specification  $\varphi$  be a satisfiable (realizable) one. As an example, consider an unsatisfiable (unrealizable) specification  $\varphi = \left( \bigwedge_{\varphi_A \in \Gamma_A} \varphi_A \right) \longrightarrow \left( \bigwedge_{\varphi_G \in \Gamma_G} \varphi_G \right)$ . If weakening the assumptions on the environment is done to an extreme level, i.e.  $\Gamma_A$  is substituted by a constant *true*, the specification  $\varphi = \text{true} \longrightarrow \left( \bigwedge_{\varphi_G \in \Gamma_G} \varphi_G \right)$  nevertheless remains unsatisfiable (unrealizable).

Dually to unsatisfiability, the specification  $\varphi$  can be trivially realizable if the assumptions on the environment are unsatisfiable. Indeed, in this case the specification reduces to the constant *true*. This case must be identified, and the user must be informed. In this case, what we are interested in is identifying the minimal unsatisfiable core of the assumptions  $\Gamma_A$ .

The concept of unsatisfiable core can be generalized to realizability, defining the concept of *unrealizable core*, that's a small unrealizable subset  $\Gamma'_G$  of the specification, and the *minimal unrealizable core* as the unrealizable core  $\Gamma'_G$  such that removing any of its component  $\varphi_G$  the resulting specification  $\varphi$  becomes realizable. The (minimal) unrealizable core will inform the designer of the (minimal) subset of the specification that is the cause of the unrealizability, thus he can concentrate on this subset and possibly fix it and make the whole specification realizable.

---

## 5.2 Computing the Minimal Unrealizable Core

The most straightforward algorithm to find a minimal unrealizable (unsatisfiable) core of an unrealizable (unsatisfiable) specification  $\varphi$  is to iteratively remove com-

ponents of the system guarantee  $\Gamma_G$  such that the specification remains unrealizable. As soon as any further removing would make the specification realizable (satisfiable) a minimal unrealizable core is found. Algorithm 13 depicts a schematic algorithm for identifying an unrealizable core of a given specification  $\varphi = \bigwedge_{\varphi_A \in \Gamma_A} \varphi_A \longrightarrow \bigwedge_{\varphi_G \in \Gamma_G} \varphi_G$ . In this algorithm the input parameters are the set of

```

1 function FindMinimalUnrealizableCore( $\Gamma_A, \Gamma_G$ )
2    $\Gamma_G' = \emptyset$ ;
3   while ( $\Gamma_G' \neq \Gamma_G$ ) do
4      $\Gamma_G' = \Gamma_G$ ;
5     foreach  $\varphi_G \in \Gamma_G$  do
6        $\Gamma_G = \Gamma_G \setminus \varphi_G$ ;
7       if (IsRealizable( $\Gamma_A, \Gamma_G$ ))  $\Gamma_G = \Gamma_G \cup \varphi_G$ ; fi
8     done;
9   done;
10  return  $\Gamma_G$ ;
11 end

```

Algorithm 13: Computing the Minimal Unrealizable Core

assumption on the environment  $\Gamma_A$  and set of the guarantees of the system  $\Gamma_G$ . The returned result is the minimal unrealizable core. Note that for a given specification there can be several minimal unrealizable (unsatisfiable) core. In this document we do not study the possibility of finding the *smallest* unrealizable (unsatisfiable) core.

The algorithm 13 can be used also to compute a minimal unrealizable core for a single set of requirements, i.e. if the specification does not contain assumptions on the environment  $\Gamma_A$ . In this case it suffices to set parameter  $\Gamma_A$  of the algorithm to  $\emptyset$ .

Section 2.1 explains how the set  $\Gamma_G$  is decomposed in three disjoint subsets  $\Gamma_G^I, \Gamma_G^T, \Gamma_G^\Psi$  which correspond to the constraints on initial states, the constraints on the transition relation and all the remaining constraints, respectively. During computation of the minimal unrealizable core, only one of these constraints sets can be taken into account, i.e. components of only one constraint set are tried to be removed. This modification of the algorithm can be useful if a specification writer would like to check only one of these subset of  $\Gamma_G$  for being the cause of unrealizability and is confident in the remaining subsets. For example, for a Büchi game a specification writer may be curious which of the fairness conditions results in unrealizability, having the constraints on the initial states and the transition relation intact.

---

## 5.3 Changing initial conditions

According to the definition of winning game structure given on page 6, a game structure is *not* winning if *some* initial states are not winning for the system. In

this case the specification can be unrealizable because its initial states are under-constrained. Adding additional constraints to the set of initial states may make the specification realizable. Such an assumption can be easily checked by changing the definition of a winning game and allowing the system to choose initial states. If after such a change the game structure becomes realizable the algorithms can return the subset of initial states which are winning for the system. Having this information a specification writer may be able to add to the original specification the additional constraints on the initial states and thus fix the problem.

---

## 5.4 Computing environment strategy for unrealizable games

### The general case

The algorithms described in Section 2.2 generate a winning strategy for realizable games. The same algorithms can be easily modified to synthesize a winning strategy for the environment in the case of unrealizable games.

In the general approach, we negate the specification, invert the role of system and environment, generate the game and look in the new built game for a winning strategy for the environment aiming to satisfy the negation of the specification.

In the Generalized Reactivity (1) approach, a strategy for the environment can be extracted by changing the quantification schedule in the generalized image computation used in the presented algorithm. Then store the intermediate results of the computation and use these intermediate results to build the strategy for the environment.

We are not going to present more details on how to extract the strategy for the environment in the general since it has been thoroughly discussed in [7]. For additional details we refer the reader to [7].

### Strategy obtained by approximation algorithms

In Section 3 we showed several algorithms which could identify that a game  $G$  is unrealizable if it is forced to reach a system deadlock state. Thus, to demonstrate the specification being unrealizable we can generate the strategy for the environment to reach the system-deadlock states.

For the algorithm 1 based on BDD representation on page 12 (and its adaptation for realizability of safety games algorithm 2 on page 13 and Büchi games

algorithm 5 on page 16) the strategy can be constructed if the consecutive values of *Deadlock* (respectively, negations of *Safe*) at every iteration are remembered. Such a sequence  $Deadlock_0, \dots, Deadlock_{n-1}$  has a property that  $Deadlock_i \subset Deadlock_{i+1}$ . For every state in  $Deadlock_i$  ( $0 < i < n$ ) the environment has such a move that there is no valid move for the system or the system will inevitably move to states in  $Deadlock_{i-1}$ .  $Deadlock_0$  is empty or  $\neg A$  for the safety condition  $A$  of a safety game.  $Deadlock_{n-1} = Deadlock$  includes initial states or states of a fairness condition for a Büchi game. To obtain the strategy from sequence  $Deadlock_0, \dots, Deadlock_{n-1}$  it is necessary to construct a sequence  $NewDeadlock_1, \dots, NewDeadlock_{n-1}$  such that for  $0 < i < n$ ,  $NewDeadlock_i = Deadlock_i \wedge \neg Deadlock_{i-1}$ , i.e. the sequence of the new states with respect to the previous  $Deadlock_i$ . The new sequence has a property that for every state from  $NewDeadlock_i$  the environment can reach a state from  $Deadlock_{i-1}$  in one step and all  $NewDeadlock_i$  are disjoint. Therefore the strategy can be given as a formula over variables  $e, s, e'$ :

$$Strategy_{\mathcal{E}} = \bigvee_{i=1}^{n-1} NewDeadlock_i \wedge \rho_{\mathcal{E}}(e, s, e') \wedge \forall s'. \rho(e, s, e', s') \rightarrow Deadlock'_{i-1}$$

For any state  $\langle e, s \rangle \in Deadlock$  the environment should choose a move  $e'$  such that  $Strategy_{\mathcal{E}}(e, s, e')$  holds, which is always possible by the construction of  $Strategy_{\mathcal{E}}$ . In this case the system in a finite number of steps will inevitably reach a situation when there is no system move  $s'$  satisfying  $\rho_{\mathcal{S}}$  or the state  $\langle e', s' \rangle \in \neg A$  for a safety game with safety condition  $A$ .

For the algorithms that use QBF solvers the strategy cannot be constructed because typically QBF solvers do not return explicit representations of *Deadlock* states or solutions of QBF problems. The only available information is the number of iterations, i.e. number of steps required for the environment to reach the system-deadlock states.

It is possible that a game is expected to be unrealizable but the algorithm shows that the system can avoid the system-deadlock states. Or for any other reason a specification writer would like to know how the system can avoid the system-deadlock states. In this case for the BDD based algorithms a strategy for the system can be reported:

$$Strategy_{\mathcal{S}} = Safe \wedge \rho_{\mathcal{E}}(e, s, e') \wedge \rho(e, s, e', s') \wedge Safe'$$

where  $Safe = \neg Deadlock$ . For the given state  $\langle e, s \rangle \in Safe$ , and the environment move  $e'$  satisfying  $\rho_{\mathcal{E}}$  the system should choose a move  $s'$  satisfying  $Strategy_{\mathcal{S}}(e, s, e', s')$ . In that case the system will be able to avoid the system-deadlock states infinitely long.

---

## 5.5 Other useful debugging information

### Input determinism

For a given requirements specification it can be important for a specification writer to know whether the specification suffers from *input nondeterminism* [10] or not, i.e. whether for every winning play satisfying the specification every move of the system is not completely determined by the environment moves. Input nondeterminism of a realizable specification may result in the existence of more than one winning strategy. And it might be the case that some of these strategies are less desirable for the specification writer than the others. The need to choose one among the possible winning strategies can be due, for example, to some non-functional requirements such as the difficulty in implementing some of the strategies, or the difficulty of specifying, synthesizing and implementing other components which are connected to the specified one. Thus, it turns out to be important to inform the specification writer that the specification he or she wrote suffers from input nondeterminism.

Formally, a requirements specification  $\phi$  (a game structure  $G$ ) is *input deterministic* iff for every two plays  $\sigma_0 : \langle e_0^{\sigma_0}, s_0^{\sigma_0} \rangle, \langle e_1^{\sigma_0}, s_1^{\sigma_0} \rangle, \dots$  and  $\sigma_1 : \langle e_0^{\sigma_1}, s_0^{\sigma_1} \rangle, \langle e_1^{\sigma_1}, s_1^{\sigma_1} \rangle, \dots$  satisfying the specification (winning for the system) the equality of all the corresponding environment moves of these two plays implies the equality of all the corresponding system moves, i.e.

$$\phi \models \forall \sigma_0 \sigma_1. \forall n \geq 0. \left( \left( \bigwedge_{i=0}^n e_i^{\sigma_0} = e_i^{\sigma_1} \right) \longrightarrow s_n^{\sigma_0} = s_n^{\sigma_1} \right)$$

A specification (a game structure) is *input nondeterministic* if it is not input deterministic.

Different algorithms can be thought of to perform checking for input determinism. The problem can be performed using classical CTL model checking algorithms over the Fair Kripke structure recognizing the same language of  $\phi$  obtained for instance with the conversion specified in [1].

Knowing that the specification is input nondeterministic the specification writer can try to remove nondeterminism by strengthening the system guarantee of the specification. It is worth noticing that, these additional constraints can make the search for a winning strategy even harder (remember that in the worst case the synthesis algorithms are doubly exponential on the size of the specification). However, if the additional constraints are simple enough the algorithms that extract a winning strategy can possibly easily be modified to apply these additional constraints to the strategy, without thus impacting too much on the overall performances.

Note also that if the strategy covering all the possible winning behavior of the system is available then the check for existence of multiple strategies (this is exactly the main purpose of checking for nondeterminism) can be easily performed on the

strategy itself. It is just necessary to check that for some reachable state  $\langle e, s \rangle$ , the condition

$$\forall e', s'_0, s'_1. ((Strategy_S(e, s, e', s'_0) \wedge Strategy_S(e, s, e', s'_1)) \rightarrow s'_0 = s'_1)$$

holds. If it does not hold and the required additional constraints on the system guarantee are simple enough then these constraints are added directly to the strategy thus making it input deterministic. In this respect, it has to be noticed that all the algorithms described in Section 2.2 as well as some of the new algorithms described in this document (such as algorithms 7 and 12) can build the strategy that include all the winning behaviors of the system.

## Output causality and requirements redundancy

Another debugging information that could be of help while writing a requirements specification is the identification of *redundant requirements* that do not add additional constraints to the specification and requirements without the *output causality*, i.e. the requirements that do not affect the behavior of output signals. Note that output causality is a weaker property than redundancy, since any redundant requirement does not affect the behavior of the system's signals.

To detect whether a requirement  $\varphi_i \in \varphi$  is redundant we can simply check whether

$$(\varphi \setminus \{\varphi_i\}) \leftrightarrow \varphi$$

is valid. In this case the requirement  $\varphi_i$  does not add further constraints to  $\varphi$  and thus it is redundant.

The check for output causality, i.e. that a particular constraint  $\varphi_i \in \varphi$  does not affect the behavior of output signals, can be performed by comparison of the strategies. If for realizable specifications  $\varphi$  and  $\varphi \setminus \{\varphi_i\}$  the strategies are the same then constraint  $\varphi_i$  does not affect the behavior of the system. We remark that the strategy can be constructed by any of the algorithms described in Section 2.2.

Constructing a strategy is a very hard problem and using incomplete but more efficient approaches can be more practical. For instance we could adapt the Cone Of Influence (COI) technique used in model checking [3] as to use it in this setting. A schematic representation of COI method is provided in algorithm 14. Given the original specification  $\varphi$  and a set of input  $\mathcal{E}$  and output  $\mathcal{S}$  signals the algorithm returns an over-approximation of the constraints which influence the behavior of the system, i.e. signals  $\mathcal{S}$ . *Vars* is a set of signals that can potentially influence the output signals. Its initial value is a set of all output signals. *COI* is a set of constraints that influence the output signals. At the beginning the set is empty. At every iteration the new constraints to be added to *COI* are those which reference the signals from *Vars*. The signals mentioned in these newly found constraints ( $Dep_s(\varphi_i)$ ) are added to *Vars*. The algorithm always terminates by one of the following two reasons: all the constraints in  $\varphi$  are found to be potentially influencing the output signals (condition at line 4), or the unprocessed constraints in  $\varphi$  refer

```

1 function FindRealizabilityCOI( $\varphi, \mathcal{E}, \mathcal{S}$ )
2    $Vars = \mathcal{S}$ ;
3    $COI = \emptyset$ ;
4   while ( $\varphi \neq \emptyset$ ) do
5      $COI_{new} = \{\varphi_i \in \varphi \mid \exists v \in Vars. \text{ and } v \in Deps(\varphi_i)\}$ ;
6      $Vars_{new} = \{v \mid \exists \varphi_i \in COI_{new}. \text{ and } v \in Deps(\varphi_i)\}$ ;
7     if ( $COI_{new} = \emptyset$ ) break; fi
8      $\varphi = \varphi \setminus COI_{new}$ ;
9      $COI = COI \cup COI_{new}$ ;
10     $Vars = Vars \cup Vars_{new}$ ;
11  done;
12  return  $COI$ ;
13 end

```

Algorithm 14: Cone Of Influence of a realizability specification.

only to the signals which cannot influence and cannot be influenced by the output signals (condition at line 7). The output value  $COI$  can be used as a substitute for  $\varphi$  but with proper care. First, if a removed constraint  $\varphi_i \in (\varphi \setminus COI)$  belongs to the environment assumptions  $\Gamma_{\mathcal{E}}$  then this constraint can potentially cause an environment-deadlock. Without this constraint  $\varphi_i$  the realizable specification may become unrealizable. The second reason is that if a constraint  $\varphi_i \in (\varphi \setminus COI)$  belongs to the system guarantee  $\Gamma_{\mathcal{S}}$  then removing the constraint may make unrealizable specification realizable. The reason is that if a constraint is in a set  $(\varphi \setminus COI)$  then the system cannot influence on this constraint, i.e. cannot control system guarantee. Quite likely the presence of such constraints indicate a bug in the specification.

## Deadlock of the system and the environment

Even though the emptiness of initial states set as well as deadlocks of the system or the environment are not typically intended by a specification writer they may be the cause of realizability or unrealizability of a game. Therefore such situations should be checked and reported to the specification writer explicitly.

As it is explained in Section 2.1 for a realizable game the initial states set has to be not empty, i.e.  $Q_0 \neq \emptyset$ . Violation of this requirement should be reported to the user. This requirement is for the case when only the system or only the environment chooses an initial state. In the case when they both control corresponding subparts of an initial state the requirement changes. It is necessary to check that for any environment initial move satisfying  $Q_0^{\mathcal{E}}$  the system can always choose such initial values for its variables that  $Q_0^{\mathcal{S}}$  is satisfied, i.e.  $\forall e. (Q_0^{\mathcal{E}} \rightarrow \exists s. Q_0^{\mathcal{S}})$ . Of course, if this condition is violated the game becomes unrealizable because the environment can always “kill” the system at the initial states, choosing such values for its variables that the system will not be able to make a move and satisfy  $Q_0^{\mathcal{S}}$ .

In the Definition 2 on page 11 of the system-deadlock states we showed that unrealizability of a game may be due to the system being in deadlock, i.e. the en-

vironment may force the system to reach a system-deadlock state  $q \in DL_S$  where with a proper environment move the system will not be able to make its own move thereby making the game unrealizable. Similarly, the realizability of a game can be due to the environment deadlock states  $DL_E$  (See Definition 3 on page 19). If the deadlocks of the system or of the environment are not expected by a specification writer it is worth to check whether the deadlocks can be forced by the other player. For the system-deadlock states this check can be done by performing unrealizability check of a safety game with safety condition  $\phi = True$ . For the environment-deadlock states it is necessary to check realizability of so called reachability game with *False* goal condition, i.e.  $\phi = False$ . These checks can be done by the algorithms (or slight modification of algorithms) described in Sections 3 and 4.

If both checks fail it does not mean that the deadlock states are always avoided. It just means that the system (the environment) can have a strategy to avoid its deadlock states. Nevertheless, a deadlock can potentially be reached. To be sure that deadlock states can never be reached it is necessary to perform a usual model checking analysis of reachability of  $DL_E \cup DL_S$  states. This can be done, for example, with the following algorithms:

```

1 function IsGameDeadlockFree(G)
2    $DL = DL_E \vee DL_S$ ;
3    $DL_{previous} = \emptyset$ ;
4   while ( $DL_{previous} \neq DL$ ) do
5      $DL_{previous} = DL$ ;
6      $DL = DL \vee (\exists e's'. (\rho_E(e, s, e') \wedge \rho_S(e, s, e', s') \wedge DL'))$ ;
7     if ( $DL \wedge Q_0 \neq \emptyset$ ) return unknown; fi
8   done
9   return true;
10 end

```

Algorithm 15: Reachability of deadlock states.



# 6 Conclusions

We have presented several approaches aiming to solve the realizability and unrealizability problem by checking sufficient conditions for realizability or unrealizability, thus avoiding the application of the complete approaches based on synthesis that assume the specification being realizable and try to synthesize a digital design.

All the approaches discussed in this document form a palette of functionalities to be integrated in the RAT [6] Requirements Analysis Tool developed within the PROSYD project (<http://rat.itc.it>).

The tool, possibly guided by heuristics or by the specification writer will apply the most appropriate technique. Whenever the specification result to be unrealizable, then the tool will provide useful debugging information to help the specification writer to correct the specification.

Once there is evidence that the specification is realizable, then the synthesis algorithms described in [7] can be applied to synthesize a digital design satisfying the specification.



# 7 References

- [1] S. Ben-David, R. Bloem, D. Fisman, A. Griesmayer, I. Pill, and S. Ruah. Automata construction algorithms optimized for PSL, September 2005. Prosyd Deliverable 3.2/4.
- [2] M. Benedetti. sKizzo: a Suite to Evaluate and Certify QBFs. In *Proc. of 20th International Conference on Automated Deduction (CADE05)*, 2005.
- [3] S. Berezin, S. V. A. Campos, and E. M. Clarke. Compositional reasoning in model checking. In *COMPOS'97: Revised Lectures from the International Symposium on Compositionality: The Significant Difference*, pages 81–102, London, UK, 1998. Springer-Verlag.
- [4] A. Biere. Resolve and expand. In H. H. Hoos and D. G. Mitchell, editors, *SAT*, volume 3542 of *Lecture Notes in Computer Science*. Springer, 2004.
- [5] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In W.R. Cleaveland, editor, *Tools and Algorithms for the Construction and Analysis of Systems. 5th International Conference, TACAS'99*, volume 1579 of *Lecture Notes in Computer Science*, pages 193–207. Springer-Verlag, July 1999.
- [6] R. Bloem, R. Cavada, C. Eisner, I. Pill, M. Roveri, and S. Semprini. Manual for property simulation and property assurance tool, November 2005. Prosyd Deliverable D1.2/4-5.
- [7] R. Bloem, B. Jobstmann, and A. Pnueli. Property-based logic synthesis for rapid design prototyping, September 2005. Prosyd Deliverable D2.2/1.
- [8] E. A. Emerson and K. S. Namjoshi, editors. *Verification, Model Checking, and Abstract Interpretation, 7th International Conference, VMCAI 2006, Charleston, SC, USA, January 8-10, 2006, Proceedings*, volume 3855 of *Lecture Notes in Computer Science*. Springer, 2006.
- [9] E. Giunchiglia, M. Narizzano, and A. Tacchella. Qube: A system for deciding quantified boolean formulas satisfiability. In *IJCAR '01: Proceedings of the First International Joint Conference on Automated Reasoning*, pages 364–369, London, UK, 2001. Springer-Verlag.
- [10] F. Korf and R. Schlör. Interface controller synthesis from requirement specifications. In *EDAC-ETC-EUROASIC*, pages 385–394, 1994.
- [11] O. Kupferman and M.Y. Vardi. Safrless decision procedures. In *Proc. 46th IEEE Symp. on Foundations of Computer Science*, pages 531–540, Pittsburgh, October 2005.
- [12] A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, pages 46–57. IEEE, 1977.
- [13] A. Pnueli and R. Rosner. On the synthesis of reactive modules. In *Proc. 16<sup>th</sup> ACM Symposium on Principles of Programming Languages*, pages 179–190, January 1989.
- [14] Accellera, Property Specification Language - Reference Manual - Version 1.01. [http://www.eda.org/vfv/docs/psl\\_lrm-1.01.pdf](http://www.eda.org/vfv/docs/psl_lrm-1.01.pdf), April 2003.
- [15] M. Roveri and S. Semprini. Novel techniques for property assurance, August 2004. Prosyd Deliverable D1.2/2.

- [16] S. Safra. On the complexity of  $\omega$ -automata. In *Proc. 29th IEEE Symp. Found. of Comp. Sci.*, pages 319–327, 1988. An extended version to appear in *J. Comp. Sys. Sci.*
- [17] L. Zhang and S. Malik. Conflict driven learning in a quantified boolean satisfiability solver. In L. T. Pileggi and A. Kuehlmann, editors, *ICCAD*, pages 442–449. ACM, 2002.