Sandra Fruhmann, BSc

Bounded Assume-Guarantee Synthesis

MASTER'S THESIS

to achieve the university degree of

Diplom-Ingenieurin

Master's degree programme: Computer Science

submitted to

Graz University of Technology

Supervisor

Univ.-Prof. Ph.D. Roderick Bloem

Institute for Applied Information Processing and Communications

Graz, May 2016

Abstract (English)

Synthesis is the derivation of a system from a given specification. Consequently, the developer is not required to specify how the system should be implemented but what the system should be capable of.

Especially the area of concurrent systems is very error-prone, as it is very likely to create faulty implementations due to deadlocks and race-conditions. Synthesis ensures that these errors do not occur in the synthesized system.

Unlike other synthesis approaches, assume-guarantee synthesis does not only provide correct but also robust systems. This methodology allows to reuse generated processes from other synthesis tasks. Thus the overall computation can be reduced without sacrificing the correctness of the system.

In this work we propose an algorithm to synthesize processes according to their high-level specification, by means of assume-guarantee synthesis. This algorithm is based on the bounded cooperative co-synthesis approach for distributed and asynchronous systems.

For comparison, we implemented a prototype for our bounded assume-guarantee synthesis approach and the bounded cooperative co-synthesis approach for distributed systems. Using this prototype, we compare these approaches by means of robustness of the synthesized processes, total runtime and statistical values from the used SMT solver. This experimental comparison shows that the processes of cooperative co-synthesis are not necessary robust and that the assurance of robustness leads to a runtime overhead.

Keywords: Reactive Systems, Bounded Synthesis, Assume-Guarantee Synthesis

Abstract (German)

Synthese ist die automatische Erstellung eines Systems anhand einer gegeben Spezifikation. Somit muss der Entwickler nicht mehr spezifizieren wie ein System implementiert wird, sondern nur noch, was dieses erfüllen muss.

Speziell der Bereich der verteilten Systeme ist sehr fehleranfällig, da die Gefahr einer fehlerhaften Implementierung aufgrund von Deadlocks und Race-Conditions sehr hoch ist. Diese Fehler können durch die Anwendung von Synthese verhindert werden. Im Gegensatz zu anderen Synthesemethoden liefert assume-guarantee synthesis nicht nur korrekte, sondern auch robuste Systeme. Dadurch können bereits synthetisierte Prozesse in neuen Systemen wieder eingesetzt werden. Somit kann die Gesamtlaufzeit des Synthetisierens verringert werden, ohne die Korrektheit des Systems zu gefährden.

In dieser Arbeit stellen wir einen Algorithmus vor, der Prozesse anhand deren Spezifikation mit der Methode "assume-guarantee synthesis" erstellt. Dieser Algorithmus basiert auf dem Ansatz "bounded cooperative co-synthesis" für verteilte Systeme. Um assume-guarantee synthesis und Bounded Cooperative Co-Synthesis miteinander zu vergleichen haben wir einen Prototyp implementiert. Mit diesem Prototyp vergleichen wir die Ansätze anhand der Robustheit der synthetisierten Prozesse, deren Laufzeit, statistischen Werten des SMT solvers und dem benötigtem Speicher.

Dieser experimentelle Vergleich zeigt, dass die synthetisierten Prozesse von Bounded Cooperative Co-Synthesis nicht notwendigerweise robust sind und dass die Garantie der Robustheit zu einem Anstieg der Laufzeit und des Speicherverbrauches führt.

Keywords: Reaktive Systeme, Bounded Synthesis, Assume-Guarantee Synthesis

Acknowledgement

I would like to express my gratitude to all people supporting me during this journey of writing my master's thesis.

Foremost, I want to express my appreciation to my advisor Roderick Bloem. He has not only guided and assisted me throughout the thesis, but also aroused my fascination for computer science many years ago.

Secondly, I would like to thank Robert Könighofer. Without his guidance, assistance and patience, this thesis would have been never finished. I am very thankful for his ideas and our discussions during the implementation and writing.

I am also grateful for the support, patience and motivational speeches of my supervisors Manfred Aigner, Thomas Popp and Stefan Tillich.

Last but not least, I would like to thank my friends and family for supporting me throughout the time. Special thanks are addressed to my parents Ottilia and Emmerich Fruhmann, who gave me the possibility to study and supported me mentally and financially.

Sandra Fruhmann

Graz, 2016

Statutory Declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the used sources. The text document uploaded to TUGRAZonline is identical to the present master's thesis.

Graz, ______________________ ______________________

Date Signature

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe. Das in TUGRAZonline hochgeladene Textdokument ist mit der vorliegenden Masterarbeit identisch.

Graz, am ______________________ ______________________

Datum Unterschrift

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1. Background and Motivation

During the last decades, the influence of computer systems on our day to day life became higher and higher. Consequently, the reliability of these systems became more important. Malfunctions do not only lead to inconveniences but can be very dangerous and life-threatening. For example, on 4 June 1996, a software bug [25] caused the explosion of the Ariane 5 launcher about 37 seconds after liftoff. The material damage amounted to about 370 million US dollar. In another incident, a race condition in an alarm systems was the reason that a manageable local blackout cascaded into the Northeast blackout, in 2003 [20]. Because of this blackout, about 50 million people lost power for about two days. The costs ware estimated between $ 4 and $ 10 billion US dollar.

Such incidents are not only cost-expensive, but can also lead to a bad image for companies. So the area of producing correct and safe code is very important in research and in the industry. There are various methods to ensure better code. Informal methods, like testing, are widely used but cannot ensure the correctness of the code. To guarantee this correctness, there are two different approaches: Formally verifying an implementation according to its specification (verification) and deriving a correct-by-construction implementation from its formal specification automatically (synthesis). While verification is already largely researched and used in the industry [18], synthesis is, although already mentioned by Alonzo Church in 1962 [9], not yet mature enough to be widely used. This thesis contributes towards improving the applicability of synthesis in the context of distributed systems.

The advantage of synthesis is that it is not applied on existing code but only on the specification. So the developer only needs to specify *what* the system should be capable of but not *how* the system should be implemented. The implementation is then constructed automatically.

As stated above, Church [9] was the first who mentioned this problem and so it is often referred to as Church's problem. In his paper, Church asked for a procedure to build an automaton or a set of recursion equivalences according to a specification relating an infinite input-string and an infinite output-string. If such an automaton cannot be built, this procedure has to signal that the synthesis requirement is impossible. The specification is assumed to be a Monadic second-order (MSO) formula, where it is only possible to quantify over unary relations. Let $\varphi(I,O)$ be a MSO-specification where I defines the input and O the output. Church's problem asks to determine if there exists an operator F which satisfies the relation $\varphi(I,F(I))$ according to the specification for all inputs I. If such an operator exists, it should be defined how it is constructed.

McNaughton [24] was the first who proposed to treat Church's problem as an infinite game with two players, where player 1 defines the input and player 2 the output. The game is played for an infinite number of rounds. Player 1 starts the round by defining the input. Afterwards, player 2 defines output according to the input. Player 2 wins the game if it can produce a correct output sequence. If player 2 is not able to produce such output, player 1 wins.

Büchi and Landweber [7] used this idea and provided the first solution based on infinite games. Some years, later Rabin [29] solved Church's problem using tree automata.

During the next years, the synthesis problem was studied for single-process systems. In 1977, Pnueli [27] adopted a fragment of tense logic and proposed it as a verification tool for concurrent programs. In particular, he introduced the temporal operator **G** for invariances and the temporal operator **F** to describe eventualities.

Emerson and Clark [14] as well as Manna and Wolper [23] used temporal logic specification to synthesize synchronization parts for concurrent programs. The drawback of these solutions was that the system to be synthesized has to be *entire* or *closed*, which means that it is not possible for the system to interact with an environment.

Pnueli and Rosner [28] solved the synthesis problem for reactive systems in the context of Linear Temporal Logic (LTL) specifications. In such systems, the implementation has to react on external inputs. Pnueli and Rosner considered this synthesis problem as a two-player game where player 1 is the environment and player 2 (the system) has to find a winning strategy for all possible input-scenarios given by the environment. The algorithm to solve this problem has an double-exponential worst case complexity.

Furthermore, Pnueli and Rosner [26] reduced the synthesis problem for distributed reactive systems for certain architectures to the halting problem. Therefore, they showed that the synthesis problem for distributed reactive systems is undecidable in general. However, they also proved that hierarchical architectures are non-elementarily decidable.

Schewe and Finkbeiner showed [16] in 2005 that the *distributed synthesis problem* for LTL specifications is decidable if the architecture does not contain information forks (like pipline architectures). The architecture contains an information fork if the environment provides information to an individual black-box process and this information cannot be completely deduced by other processes. Two years later, in 2007, the same authors provided a procedure to semi-decide the synthesis problem for systems with information forks [31]. They showed that it is possible to decide this problem if the number of states in the system is restricted. Furthermore, they showed that this bounded synthesis problem can be reduced to a satisfiability (SAT) problem.

One challenge in synthesis of distributed systems is that the systems should be robust against changes of individual processes, to provide a modular system. Chatterjee and Henziger [8] showed, in 2007, that it is possible to build robust systems by computing secure-equilibrium strategies. They solved this assume-guarantee synthesis (AGS) problem using a 3-player game graph. The advantage of AGS is that parts of the system can be replaced without resynthesizing or verifying the rest of the system.

1.2. Problems Addressed in this Thesis

As stated above, Chatterjee and Henziger [8] introduced AGS and provided a game-theoretical solution for the distributed synthesis problem. However, this solution has never been implemented and is only applicable for distributed systems in a perfect information setting. Consequently, all processes have to be aware of all variables and inputs in the system and it is not possible for a process to use private variables, which is very unrealistic for distributed systems.

1.3. Outline of the Solution

In this work, we are going to present an algorithm to semi-decide the assume-guarantee synthesis problem for given LTL-specifications of two processes A and B, where the processes may only have partial information about inputs or each other's states. To solve the AGS problem it is necessary to fulfil three conditions containing the specifications of the processes. The first condition ensures the correctness of the system. The system is correct if the specifications of all processes are fulfilled. The other two conditions ensure that the processes are robust. A process is called robust if the other process can be exchanged with an arbitrary process as long as the new process still fulfils the specification of the exchanged process.

Our algorithm accomplishes the AGS conditions by extending the bounded synthesis approach for cooperative co-synthesis by Schewe and Finkbeiner [31].

The first step of our algorithm is to describe the AGS conditions as LTL formulas. These formulas are afterwards translated into universal co-Büchi automata.

These produced universal co-Büchi automata are later used to build a constraint system which describes the behaviour of the transition relations of three different state spaces. Two of these state spaces describe the processes and the third one describes the composition of the other state spaces.

If this constraint system is satisfiable, the model of the solution provides the requested robust system where the processes can be exchanged easily, without resynthesizing or verifying the whole system.

However, if the constraint system is not satisfiable, the system is not realisable with these process-specifications. Similar to the bounded synthesis approach, we allow the system designer to increase the bound by increasing the number of possible states. In our algorithm this is realized by adding fresh memory variables that are not restricted by the specifications. As these memory variables can be fully controlled by the algorithm, they allow the algorithm more possibilities to find a suitable transition relation.

By increasing the number of these memory variables iteratively, the algorithm can be used as a procedure to semi-decide unbounded AGS problems.

Unlike the solution proposed by Chatterjee and Henziger [8], our algorithm allows the designer to distinguish between local and global variables. Therefore, it is possible to define local and global process variables as well as global an local input variables. Consequently, a perfect information setting in the specification is no longer required.

1.4. Related Work

Our solution is based on the bounded synthesis approach by Schewe and Finkbeiner [31]. They did not only propose the constraint-based bounded synthesis approach in [31] but also an automata-theoretic approach. In this approach they are reducing the synthesis problem to an emptiness check on safety automata.

Filiot, Jin and Raskin [15] provided in 2011 an alternative algorithms to synthesize systems from LTL specifications. Like Schewe and Finkbeiner, they translate the specification into a co-Büchi automaton. But instead of using SMT-constraints and bounding the number of states in the implementation, they are using antichains and bound the number of visits of rejecting states in the co-Büchi automaton.

In 2010, Fisman, Kupferman and Lustig [17] proposed rational synthesis. In rational synthesis, the environment is not seen as an opponent, but as a set of rational components. This approches does not only provide a synthesized system but also strategies for the components of the environment. The correctness of the system can be guaranteed as long as the components follow the proposed strategy. Compared to AGS it is not possible to exchange processes without resynthesizing or verifying the whole system, even if the new processes would implement their local specifications.

Bloem, Chatterjee, Jacobs and Könighofer provided, in 2015, a solution of AGS problems for sketched systems, where parts of the implementation are already given, including systems with partial information [5]. Furthermore they analysed the complexity and decidability in different settings. This work is also based on bounded synthesis. However, while this work focuses on sketched programs, our solution is about the synthesis of complete distributed systems from scratch.

1.5. Structure of this Document

The rest of the thesis starts by giving an overview of the used theoretical background and by establishing notation. In particular, Chapter 2 gives an overview about Linear Temporal Logic and synthesis approaches. It also establishes notation for processes, process descriptions, scheduler and specifications.

We use this notations in Chapter 3 to introduce our algorithm to solve AGS problems. This chapter shows, how the bounded synthesis approach of Schewe and Finkbeiner [31] can be used to construct a system based on LTL-specifications that solves the AGS problem.

The next chapter (Chapter 4) gives an overview of the implementation of the prototype tool that solves the AGS problem. In particular, it defines the used input format and the structure of the synthesized implementation.

Chapter 5 discusses the performance of our AGS algorithm and compares it to the co-synthesis approach that does not involve robustness.

We conclude this thesis in Chapter 6 by giving an short conclusion and a discussion of future work.

Chapter 2

Preliminaries

In this section, we introduce the background of LTL, ω-automata (specifically Büchi automata) and synthesis. Furthermore, we introduce the notation for variables, valuations, traces, processes, process descriptions, schedulers and specifications.

2.1. Logic

2.1.1. Linear Temporal Logic

Temporal Logic was used since ancient times. For example, Aristotle used it in his famous sea-battle analogy to discuss the problem of future contingents [1, 22]. The problem is about the assignment of truth-values for statements involving future contingents. Aristotle stated that it is not possible to assign determinately truth-values to statements like ”Tomorrow there will be a sea-battle”.

The first one who formalized temporal logic was Prior in 1957 [2]. He referred to it as ”Tense-logic”. Nowadays the most important and popular types are Linear Time Logics (e.g. Linear Temporal Logic) and Branching Time Logics (e.g. Computation Tree Logic). Linear Temporal Logic (LTL), was first proposed by Pnueli [27] in 1977 as a possibility to describe programs, as it allows designers to formalize invariants and temporal implications. LTL formulas are constructed from a set of atomic propositions *Prop* using the Boolean Operators $\neg$ and $\vee$ and the temporal operators $\mathbf{U}$ (”until”) and $\mathbf{X}$ (”next”). Additionally, the Boolean operators $\wedge, \rightarrow, \leftrightarrow$, the constants ”true” and ”false” and the temporal operators $\mathbf{G}$ (”globally”), $\mathbf{R}$ (”Release”) and $\mathbf{F}$ (”eventually”) can be derived. The syntax of LTL can be formally defined in the following way [34]:

- An atomic proposition $p \in Prop$ is an LTL formula and
- if φ and ψ are LTL formulas then $\neg\varphi$, $\varphi \vee \psi, \varphi\ \mathbf{U}\ \psi$ and $\mathbf{X}\ \varphi$ are also LTL formuals.

An LTL formula is evaluated over the truth assignments of an infinite trace $\omega = \omega_0\omega_1\dots$, where each $\omega_i \subseteq$ Prop is a set of atomic propositions that are true at step $i \in \mathbb{N}$. For a trace ω and a point $i \in \mathbb{N}, \omega^i \models \varphi$

indicates that the formula φ holds in the point i of the trace ω. The semantics of the operators can be formally defined as follows:

- $\omega^i \models p$ for $p \in \text{Prop}$ iff $p \in \omega_i$,
- $\omega^i \models \neg\varphi$ iff $\omega^i \not\models \varphi$,
- $\omega^i \models \varphi \vee \psi$ iff $\omega^i \models \varphi$ or $\omega^i \models \psi$,
- $\omega^i \models \mathbf{X}\,\varphi$ iff $\omega^{i+1} \models \varphi$, and
- $\omega^i \models \varphi\,\mathbf{U}\;\psi$ iff there exists $j \geqslant i$ where $\omega^j \models \psi$ and forall $j > k \geqslant i$ the condition $\omega^i \models \varphi$ holds.

The additional operators and constants can be defined in the following way:

- $\varphi \wedge \psi \equiv \neg(\neg\varphi \vee \neg\psi)$,
- $\varphi \rightarrow \psi \equiv \neg\varphi \vee \psi$,
- $\varphi \leftrightarrow \psi \equiv (\varphi \rightarrow \psi) \wedge (\psi \rightarrow \varphi)$,
- true $\equiv \varphi \vee \neg\varphi$,
- false $\equiv \neg$ true,
- $\mathbf{F}\,\varphi \equiv$ true $\mathbf{U}\,\varphi$,
- $\mathbf{G}\,\varphi \equiv \neg\,\mathbf{F}\,\neg\varphi$, and
- $\varphi\,\mathbf{R}\,\psi \equiv \neg(\neg\psi\,\mathbf{U}\,\neg\varphi)$.

2.1.2. Propositional logic

Propositional logic is used to formalize statements and to reason about them. The syntax can be formally defined as follows:

- An atomic proposition $p \in \text{Prop}$ is a propositional formula and
- if φ and ψ are propositional formulas then $\neg\varphi, \varphi \vee \psi, \varphi \wedge \psi$ and $\varphi \rightarrow \psi$ are also propositional formulas.

The truth-value of a formula φ is evaluated over its interpretation function I. I assigns to any propositional formula ϕ a truth value $I(\phi) \in \{true, false\}$

In the following, we are going to describe the semantics using the interpretation function. However, instead of writing $I(\varphi) = \text{true}$, we will write $I \models \varphi$ (the interpretation function models ϕ):

- $I \models \neg\phi$ iff $I \not\models \phi$,
- $I \models \phi \wedge \psi$ iff $I \models \phi$ and $I \models \psi$,
- $I \models \phi \vee \psi$ iff $I \models \phi$ or $I \models \psi$,
- $I \models \phi \rightarrow \psi$ iff $I \models \neg\phi$ or $I \models \psi$,

a	b	$a \wedge b$
false	*false*	*false*
false	*true*	*false*
true	*false*	*false*
true	*true*	*true*

Table 2.1.: Truth table for the formula $a \wedge b$

2.1.2.1. Conjunctive Normal Form

A formula is in conjunctive normal form if its clauses (disjunctions of literals) are conjuncted, where literals are atomic propositions or their negation:

$$\bigwedge_i \bigvee_j l_{ij}$$

2.1.2.2. Decidability

Propositional logic is NP-complete and therefore decidable. Consequently, there exist decision procedures to determine if propositional formulas are satisfiable. An example for such a procedure is the truth table. The truth table shows the logical value of a formula by assigning all possible combinations of truth-values to the literals and computing the truth value of the formula using these assignments. An example can be found in Table 2.1. Another decision procedure is the DPLL algorithm, described in the following paragraph.

Davis–Putnam–Logemann–Loveland (DPLL) algorithm The DPLL algorithm was invented 1962 [11] by Davis, Putnam, Logemann and Loveland. It is a recursive backtracking algorithm to decide the satisfiability of CNF formulas. Listing 2.1 shows a pseudocode of the algorithm.

The algorithm starts by checking if the formula is already satisfied. If the formula is satisfied (all clauses evaluate to *true*), the algorithm returns *true*. Otherwise, the algorithm checks if there exists a conflict in the formula and returns *false* if such a conflict can be found. If the formula is not already satisfied and does not contain a conflict, the algorithm tries to identify pure literals and unit clauses to reduce the search space. A pure literal is a literal that has only one polarity throughout the whole formula. Therefore, the truth-value can be directly assigned and the clauses containing this literal are *true* and the rest of the formula is going to be checked recursively. A unit clause is a clause consisting of only a single unassigned literal and can be therefore only satisfied by assigning the according truth-value to this unassigned literal. Also here, the rest of the formula is checked recursively with this assignment. If the current formula does neither contain a pure literal nor an unit clause, the algorithm selects a literal, guesses the truth-value, assigns it to the literal and propagates the formula with this assignment. If the formula is satisfiable with this assignment, the algorithm returns *true*, otherwise the algorithm assign the other truth-value and returns the outcome of this propagation.

This algorithm was improved throughout the last decades by, for example, using and improving branching heuristics and conflict driven learning techniques [35] [19].

Listing 2.1: DPLL algorithm

```
input: CNF-formula φ
```

```

function DPLL(φ):
  if(satisfied(φ))
    return true;

  if(conflict(φ))
    return false;

  {l, value} <- findPureLiteral(φ)
  if({l, value} is not empty)
    return DPLL(φ[l|value]);

  {l, value} <- findUnitClause(φ)
  if({l, value} is not empty)
    return DPLL(φ[l|value]);

  {l, value} <- selectLiteral(φ)
  if(DPLL(φ[l|value])
    return true;

  return DPLL(φ[l|¬value]);
```

2.1.3. First-order logic

First-order logic extends propositional logic by variables, functions, predicates and quantifiers. Predicates and functions can be used to describe properties and relations. The arguments of predicate and functions are called terms. Using quantifiers and variables it is possible to describe more general statements. For example, in propositional logic, the Epimedes paradox [30] ”All Cretans are liars” can only be expressed as a single term p. Using predicate logic, this statement can be coded more detailed by defining the predicates C for Cretans and L for liar. With these predicates and the forall quantifier ($\forall$), we can formalize the following statement: $\forall x.C(x) \rightarrow L(x)$.

2.1.3.1. Syntax

First-order logic can be formally described by terms and formulas:

Terms

- Every variable is a term
- If $f \in \mathcal{F}$ is a 0-ary function then f is a term
- If $t_1, t_2, \ldots, t_n$ are terms and $f \in \mathcal{F}$ n-ary function with $n > 0$, then $f(t_1, \ldots, t_n)$ is a term.

Formulas

- If $P \in \mathcal{P}$ is an n-ary predicate symbol and $t_1, \dots, t_n$ are terms, then $P(t_1, \dots, t_n)$ is a formula
- If φ and ψ are formulas, then so are $\neg\varphi$, $\varphi \wedge \psi$, $\varphi \vee \psi$ and $\varphi \rightarrow \psi$.
- If φ is a formula and x is a variable then $\exists x.\varphi$ and $\forall x.\varphi$ are also formulas.

2.1.3.2. Semantics

Like in Section 2.1.2, the truth-value of a formula φ is evaluated over its interpretation function I. Therefore, ϕ is true if $I(\phi)$ is true.

The interpretation function consists of:

- A non-emtpy set of objects (D),
- an assignment of each function symbol f of arity n to an object. $I(f) : D^n \rightarrow D$ and
- an assignment of each predicate symbol P of arity n to an truth value. $I(P) : D^n \rightarrow \{\text{true}, \text{false}\}$.

In the following, we are going to describe the semantics using the interpretation function. However, instead of writing $I(\varphi) = \text{true}$, we will write $I \models \varphi$ (the interpretations functions models ϕ):

- $I \models P(t_1, \cdots, t_n)$ iff $I(P)$ applied to $< I(t_1), \dots, I(t_n) >$ is *true*,
- $I \models t_1 = t_2$ iff $I(t_1) = I(t_2)$,
- $I \models \neg\phi$ iff $I \not\models \phi$,
- $I \models \phi \wedge \psi$ iff $I \models \phi$ and $I \models \psi$,
- $I \models \phi \vee \psi$ iff $I \models \phi$ or $I \models \psi$,
- $I \models \phi \rightarrow \psi$ iff $I \models \neg\phi$ or $I \models \psi$,
- $I \models \forall x.\phi(x)$ iff $I \models \phi(d)$ for all $d \in D$,
- $I \models \exists x.\phi(x)$ iff $I \models \phi(d)$ for at least one $d \in D$.

A logical formula ϕ is *satisfiable* if there exists an interpretation function I such that $I \models \phi$. It is *valid* if ϕ is true for all interpretation functions.

2.1.3.3. Decidability

First-order logic is in general undecidable. This can be proved by reducing the Halting Problem to the SAT problem for FOL formulas. The reduction is done by encoding the instructions and the tape of the Turing-machine into FOL formulas. The tape is described as a successor-function s (the first cell is encoded as $s(0)$, the cell right of the first cell is encoded as $s(s(0))$ ...). As it is not possible to go left if the head is situated at the first cell, a constraint has to be added that the Turing-machine will stay at the leftmost cell. The instructions are encoded as transition functions describing movement between the cells. The Turing-machine halts if, given an arbitrary starting value, the leftmost state is reached. In FOL this question translates to: Does there exists a timestamp t where the position is 0.

2.1.3.4. Satisfiable modulo theories (SMT)

SMT extends first-order logic by additional theories. These extensions provide the possibility to add additional interpretations to function or predicate symbols. SMT formulas can be solved by combining the DPLL algorithm with additional algorithms to solve the theories. Simplified, the DPLL algorithm provides the skeleton to solve the formulas and the theory solver checks if the current assignment is valid.

To minimize the runtime, the theory solver returns the reason if a formula is not satisfiable. This reason can be used to make better decisions and therefore to decrease the runtime.

Example 2.1. We have the following formula: $x > 0 \land x < 0 \lor x = 0$. This formula can be satisfied if x is equal to zero. In the first step, the DPLL algorithm tries to set $x > 0$ and $x < 0$ to true and $x = 0$ to false. The theory solver checks this assignment and returns that $x > 0 \land x < 0$ is not consistent. The DPLL algorithm uses this information and finally finds a valid assignment by setting the term $x = 0$ to true and the other terms to false. □

2.1.3.5. Quantifier-free linear integer arithmetic logic

A quantifier-free linear integer arithmetic formula is a first-order logic formula describing equalities, inequalities and disequalities. The atoms are in the form of $r_1x_1 + r_2x_2 + \cdots \bowtie b$ where r_i are rational numbers and x_i represent integer variables. The symbol $\bowtie$ represents either an equality operator (=) or an inequality operator ($>, <, \leqslant, \geqslant$ or $\neq$).

Such formulas are commonly solved by extending the DPLL algorithm with the Simplex-algorithm. The simplex algorithm originates from the field of mathematical optimization and is used to solve linear optimization problems. The tableau-table of the simplex algorithm is used by the backtracking mechanism of DPLL. The first algorithms updated the tableau inclemently, which lead to a costly overhead during the backtracking. Newer algorithms avoid these costly updates to reduce this overhead. [12]

2.2. Automata

2.2.1. Finite ω-Automata

An ω-automaton [33] can be defined as a 5-tuple $A = (Q, 2^{\text{Prop}}, \Delta, Q_0, Acc)$ whose components are:

- A finite set Q, which defines the states,
- the finite input alphabet 2^{Prop},
- the transition relation $\Delta \subseteq Q \times 2^{\text{Prop}} \times Q$,
- a finite set of initial states $Q_0 \subseteq Q$, and
- the acceptance condition $\text{Acc} \subseteq Q^{\omega}$.

A finite ω-automaton is called *deterministic* if for every state and input letter there exists only one target state. Formally the automaton is deterministic iff its transition relation is deterministic [13]:

$$\forall q \in Q, \omega \in 2^{\text{Prop}}.|\{q' \in Q | (q, \omega, q') \in \Delta\}| \leqslant 1$$

A *run* on a given infinite input-string $\omega = \omega_0\omega_1 \ldots \in (2^{\text{Prop}})^\omega$ is an infinite sequence $r = r_0r_1 \ldots \in Q^\omega$ of states such that:

- $r_0 = q_0$ and
- $(r_i, \omega_i, r_{i+1}) \in \Delta$ for all $i \in \mathbb{N}$

For a deterministic automaton, a string is accepted if its run r fulfils the acceptance condition: $r \in$ Acc. For an non-deterministic automaton A, a string ω is accepted by A if there exists a run r of A on ω that fulfils the acceptance condition $r \in$ Acc. We will use another kind of automata, which are called universal automata. A universal automaton accepts a string if *all runs* fulfil the acceptance conditions.

Depending on the acceptance conditions, there exist various types of ω-automata. Popular types are Büchi automata, Muller automata, Rabin automata and Streett automata. In this thesis we only consider universal co-Büchi automata.

2.2.2. Büchi Automata

A Büchi automaton is an ω-automaton. Büchi invented this automata in 1960 as a decision method in restricted second order arithmetic [6].

To define the acceptance condition, a new set of *accepting states* (F) is introduced. A run r is accepted if at least one accepting state occurs infinitely often in this run. Let $\inf(r)$ be the set of all states which occur infinitely often in the run. Then the acceptance condition can be formulated [33] as

$$Acc = \{r \in Q^\omega | \inf(r) \cap F \neq \varnothing\}$$

co-Büchi automaton A co-Büchi automaton only differs from the Büchi automaton in the acceptance condition. While it is enough for the Büchi acceptance condition that at least one accepting state occurs infinitely often in the run, for the co-Büchi acceptance condition there has to exist a point i in the run, where the following states are only accepting states. This condition can be formalized in the following way:

$$Acc = \{r \in Q^\omega | \inf(r) \cap (Q \backslash F) = \varnothing\}$$

That is, non-accepting (*rejecting*) states must be visited only finitely often and therefore, there exists a point in the run where the following states are only visited finitely often.

2.3. Synthesis

Synthesis is the automatic construction of a system according to its formal specification. Due to the automatic construction, the system is correct-by-construction and does not need any further verification or testing.

During the last decades, synthesis has been studied for distributed and non-distributed systems, reactive and non-reactive systems, synchronous and asynchronous systems.

In distributed systems, the goal is to synthesize two or more processes to fulfil the requirements. Every process has its own state space and may interact with other processes in the system. Therefore, the synthesis method does not only have to model one state space representing the system but also the state space of each process.

Reactive systems react on external events, which are handled by the system as inputs. These events can be real inputs from users or actions from other processes. An often used input is the scheduler, which is relevant for distributed asynchronous systems. These systems are harder to synthesize then non-reactive systems, as the constructed system has to react on all possible inputs given by the environment.

In asynchronous systems the processes are not scheduled at the same time. A possibility to model this behaviour is to add the scheduler to the environment. In this way, it is possible to handle a scheduler as an input and the system has to react on its decisions. However, synthesis for asynchronous systems is undecidable [32] if the implementation of at least two processes is unknown.

In this thesis we focus on reactive, distributed, asynchronous systems consisting of two processes. The used language for the specification is LTL.

2.3.1. Definitions

2.3.1.1. Variables, Valuations, Traces

Let X be a finite set of Boolean variables. A valuation is a function v, which maps each $x \in X$ to a Boolean value $v : x \to \{\text{true}, \text{false}\}$. We use the notation $V = 2^X$ to describe the set of valuations of X. Let X' be a subset of X, we define $v \restriction_{X'}$ as the restriction of the valuation of v to the variables of X'. We distinguish between global, local and memory variables. Global variables can be set by all processes and local and memory variables can only be set by one process. While global and local variables occur within the specification, memory variables are used to extend the possible state space of the process.

A trace π of X is an infinite sequence $\pi = v_0 v_1 \cdots \in V^\omega$ of valuations. Using $\pi \restriction_{X'} = v_0 \restriction_{X'} v_1 \restriction_{X'} \ldots \in V^\omega$ we can also restrict the valuations of the variables in the trace.

2.3.1.2. Process

For $i \in \{1, 2\}$ a Process i is a tuple $P_i' = (S_i, s_{i_0}, I_i, \tau_i)$ consisting of the following components:

- $S_i = L_i \cup G_i \cup M_i$ is a finite set of state variables, where L_i defines the set of local variables, G the set of global variables and M_i the set of memory variables,
- $s_{i_0} \in 2^{S_i}$ is the initial state,
- I_i is a finite set of local and global input variables and disjoint from S_i, and
- τ_i is a transition function that maps the current state and the input variables to the next state: $2^{S_i} \times 2^{I_i} \to 2^{S_i}$

Given an input trace $X = x_0x_1x_1 \ldots \in (2^{I_i})^\omega$ the execution trace of a process P'_i is defined as $t(X,P'_i) = \omega = \omega_0\omega_1\omega_2\omega_3 \ldots \in 2^{(S_i \cup I_i)^\omega}$ where $\omega = x_j \cup s_{ij}$ and $s_{ij} = \tau_i(s_{ij+1}, x_j)$.

A process P'_i satisfies an LTL specification ϕ if and only if $\forall x \in (2^{I_i})^\omega : t(X,P'_i) \models \phi$.

2.3.1.3. Process Description

A process description $P_i = (S_i, s_{i_0}, I_i)$ is a process without the transition function τ_i, therefore it is much more general then the process itself, and S_i can be updated not-deterministically. We write $P'_i \leq P_i$ to define that a process P'_i is an implementation of the process description P_i.

2.3.1.4. System

A system is a tuple $U = (P'_1, P'_2, C)$ consisting of the following components:

- P'_1 representing the first process,
- P'_2 representing the second process and
- C representing the scheduler

2.3.1.5. Scheduler

A scheduler chooses for each computation step, which process can update its variables. If one process P_i is selected, the local variables and memory of the other process remain unchanged in the next state. Given a finite trace π^*, it can formally be defined as a function $C : \pi^* \to \{1,2\}$. A scheduler is a fair scheduler if it selects both processes infinitely often, i.e., for all possible traces $\pi = v_0v_1 \cdots \in V^\omega$ there exist infinitely many $j \geqslant 0$ and infinitely many $k \geqslant 0$ such that $C(v_0, \ldots, v_j) = 1$ and $C(v_0, \ldots, v_k) = 2$.

In the following we describe the sets of possible traces depending on processes, process descriptions and the scheduler. While the valuations of the variables depend on the transition relation of a process if a process was scheduled, the valuations of the variables is not restricted if a process description is scheduled:

Given the process descriptions P_1 and P_2, the combined state space $S = S_1 \cup S_2$ and the scheduler C, the set of possible traces can be defined as

$$[[P_1||P_2||C]] = \left\{ v_0v_1 \cdots \in V^\omega \;\middle|\; \begin{array}{c} \forall j \geqslant 0. C(v_0v_1 \ldots v_j) = i \in \{1,2\} \\ v_j \restriction_{(S\setminus S_i)} = v_{j+i} \restriction_{(S\setminus S_i)} \\ v_0 \restriction_{S_1} = s_{1_0} \wedge v_0 \restriction_{S_2} = s_{2_0} \end{array} \right\}$$

Given the process description P_1 and the process $P'_2 \leq P_2$ and the scheduler C, the set of possible traces can be defined as

$$[[P_1||P_2'||C]] = \left\{ v_0v_1\cdots \in V^{\omega} \;\middle|\; \begin{array}{c} \forall j \geqslant 0. C(v_0v_1\dots v_j) = i \in \{1,2\} \\ v_j \restriction_{(S\setminus S_i)} = v_{j+i} \restriction_{(S\setminus S_i)} \\ \exists x \in 2^{I_i} : v_{j+1} \restriction_{S_i} = \tau_i(v_j \restriction_{S_i}, x) \text{ iff } i = 2 \\ v_0 \restriction_{S_1} = s_{1_0} \wedge v_0 \restriction_{S_2} = s_{2_0} \end{array} \right\}.$$

Given the process $P_1' \preceq P_1$ and the process description P_2 and the scheduler C , the set of possible traces can be defined as

$$[[P_1'||P_2||C]] = \left\{ v_0v_1\cdots \in V^{\omega} \;\middle|\; \begin{array}{c} \forall j \geqslant 0. C(v_0v_1\dots v_j) = i \in \{1,2\} \\ v_j \restriction_{(S\setminus S_i)} = v_{j+i} \restriction_{(S\setminus S_i)} \\ \exists x \in 2^{I_i} : v_{j+1} \restriction_{S_i} = \tau_i(v_j \restriction_{S_i}, x) \text{ iff } i = 1 \\ v_0 \restriction_{S_1} = s_{1_0} \wedge v_0 \restriction_{S_2} = s_{2_0} \end{array} \right\}.$$

Given the processes $P_1' \preceq P_1$ and $P_2' \preceq P_2$ and the scheduler C, the set of possible traces can be defined as

$$[[P_1'||P_2'||C]] = \left\{ v_0v_1\cdots \in V^{\omega} \;\middle|\; \begin{array}{c} \forall j \geqslant 0. C(v_0v_1\dots v_j) = i \in \{1,2\} \\ v_j \restriction_{(S\setminus S_i)} = v_{j+i} \restriction_{(S\setminus S_i)} \\ \exists x \in 2^{I_i} : v_{j+1} \restriction_{S_i} = \tau_i(v_j \restriction_{S_i}, x) \\ v_0 \restriction_{S_1} = s_{1_0} \wedge v_0 \restriction_{S_2} = s_{2_0} \end{array} \right\}.$$

2.3.1.6. Specification

A specification ϕ is used by a program designer to specify the requirements of a system. In synthesis, this specification is eventually used to automatically construct the processes, which fulfils these requirements. As a specification language we use LTL.

2.3.2. Co-Synthesis

As stated above, the classical synthesis problem asks for the construction of a single process which fulfils its specification. The distributed synthesis problem extends this requirement such that the specification has to be fulfilled 2 constructed processes.

In the co-synthesis problem, there not only exists a specification for the system, but every process has its own specification, which has to be fulfilled. There are two classical approaches on how the processes should interact which each other in such an system: cooperative or competitive. The cooperative approach corresponds to the distributed synthesis problem. Here, the processes are cooperating to fulfil all specifications in the system. In contrast to this behaviour, the only objective for each of the processes in the competitive approach is to fulfil its own specification independent of the behaviour of other processes. This competitive behaviour can easily lead to unrealisable systems if resources are shared, while such system would be possible if the processes were to cooperate instead.

For instance, consider two processes P_1 and P_2. Let the specification of P_1 be that a shared bit a has to be set at minimum every second tick, and let the specification of P_2 be that a has to be set always. If the processes are cooperative, a possible system would be that P_1 and P_2 are always setting a. However, if the behaviour of these processes is competitive, such a system cannot be constructed, as process P_2 can assign false to a in every second step.

As you can see in this example, a drawback of the cooperative approach is that the constructed system is not robust in the sense that individual processes can be exchanged with another processes, even though this other process would fulfil the specification. Consequently, processes cannot be exchanged without any further verification of the correctness of the system. This drawback does not occur if the processes behave competitive. However, specifications with shared resources become easily unrealisable.

Chatterjeee and Henzinger [8] defined another approach, called assume-guarantee synthesis. This approach states that the processes do not have to be strictly competitive to construct a robust system. In assume-guarantee synthesis, a process only needs to fulfil its specification if the other process also fulfils its specification. As a result, the processes can be exchanged easily for processes that fulfil the specification but the objectives are not so complementary as in the competitive synthesis approach.

Below, you can find a more detailed description of these synthesis approaches. For all co-synthesis problems, we consider the two process descriptions P_1 and P_2 with $P_i = (S_i, s_0, I_i)$, two processes P_1' and P_2' with $P_i = (S_i, s_0, I_i, \tau_i)$ and $P_i' \preceq P_i$, their specifications ϕ_1 for P_1 and ϕ_2 for P_2.

2.3.2.1. Cooperative co-synthesis

As stated above, the objective of cooperative co-synthesis is to find a pair of processes that fulfil all specifications in the system. As a process is not responsible for the realisability of a specific specification, it is not possible to exchange a process with another process, without any further verification of the system. Furthermore the synthesized system should be valid for all fair schedulers.

The cooperative co-synthesis problem can be formally defined as follows: Do there exist two processes for the process descriptions $P_1' \preceq P_1$ and $P_2' \preceq P_2$, such that the following condition holds for all fair schedulers C:

$$[[P_1'||P_2'||C]] \upharpoonright (S \cup I) \models \phi_1 \wedge \phi_2?$$

2.3.2.2. Competitive co-synthesis

In contrast to cooperative co-synthesis, the objective of the processes in the competitive co-synthesis approach is not to fulfil all specifications in the system, but only to fulfil its own specification. Consequently, a process can be replaced by another one which fulfils its specification. As a process cannot rely on the transition function of the other process but only on the process description, the co-synthesis problem becomes easily unrealisable if the processes access shared resources. However, like in cooperative co-synthesis, the synthesized processes should be valid for all fair schedulers.

Formally the competitive co-synthesis problem is defined as follows: Do there exist two processes for the process descriptions $P_1' \preceq P_1$ and $P_2' \preceq P_2$, such that the following conditions hold for all fair schedulers C:

$$(i)\ [[P_1'||P_2||C]] \upharpoonright (S \cup I) \models \phi_1 \text{ and}$$
$$(ii)\ [[P_1||P_2'||C]] \upharpoonright (S \cup I) \models \phi_2$$

2.3.2.3. Assume-guarantee synthesis

The assume-guarantee synthesis approach is a compromise between the cooperative and the competitive approach. The synthesized system should be robust, but it should not be as strict as competitive co-synthesis, so that it is possible to synthesize more systems with shared resources. The idea is to take the specification of the other process into account and to assume that the other process is primarily going to fulfil its specification and thus does not behave strictly competitive. Therefore, the the synthesis procedure has only to find a strategy for the behaviours of the other process if this process fulfils its specification. Another objective for the processes is that they have to fulfil, like in the cooperative approach, all specifications in the system.

These objectives can be formally defined as follows: Do there exist two processes for the process descriptions $P_1' \preceq P_1$ and $P_2' \preceq P_2$, such that the following conditions hold for all fair schedulers C:

$$(i)\ [[P_1'||P_2'||C]] \upharpoonright (S \cup I) \models \phi_1 \wedge \phi_2,$$
$$(ii)\ [[P_1'||P_2||C]] \upharpoonright (S \cup I) \models \phi_2 \rightarrow \phi_1 \text{ and}$$
$$(iii)\ [[P_1||P_2'||C]] \upharpoonright (S \cup I) \models \phi_1 \rightarrow \phi_2?$$

2.3.3. Constraint-Based Bounded Synthesis

Schewe and Finkbeiner [31] defined in 2007 a constraint-based method to semi-decide the synthesis problem for reactive systems with LTL specifications. They described, based on the universal co-Büchi automata of the LTL-specification, a constraint-based transition system which is represented by uninterpreted functions.

To translate the LTL-specification into a universal co-Büchi automaton, the specification ϕ is first negated. Afterwards, $\neg\phi$ is translated into a non-deterministic Büchi automaton $A_1 = (Q, 2^{\text{Prop}}, \Delta, Q_0, F)$. This constructed Büchi automaton is interpreted as a universal co-Büchi automaton $A_2 = (Q, 2^{\text{Prop}}, \Delta, Q_0, Q \backslash F)$ by interpreting the final states of A_i as rejecting states and the non-final states as accepting states. Consequently, to fulfil the acceptance condition, the final states must not be visited infinitely often.

The transition relation Δ of the universal co-Büchi automaton is used to construct the constraint system. For this construction we define the SMT-encoding function $\text{SMT}(S, I, \tau, \Delta)$ that takes the following arguments:

- S is a set of states of the system. A state is defined by the valuation of the global, local and memory variables of the process
- I is a set of input variables
- τ is an uninterpreted transition function: $2^S \times 2^I \rightarrow 2^S$
- $\Delta \subseteq Q \times 2^{S \cup I} \times Q$ is the transition relation of the universal co-Büchi automaton

Additionally we define the uninterpreted functions

- $\lambda_B : 2^Q \times 2^S \to \mathbb{B}$, which maps a pair of states to true iff the state in the universal co-Büchi automaton corresponds to the state in the process and
- $\lambda_\sharp : 2^Q \times 2^S \to \mathbb{N}$, which assigns a natural number to a pair.

Furthermore we are using the $\rhd$ operator as a placeholder, which stands for " $>$ " if the endstate (q') of the transition in the universal co-Büchi automaton is rejecting and " $\geqslant$ " if the endstate is accepting.

Using these functions, components and the $\rhd$-operator, it is possible to define the SMT-encoding function $\mathrm{SMT}(S,I,\tau,\Delta)$ in the following way: $\mathrm{SMT}(S,I,\tau,\Delta) =$

$$\lambda_B(q_0,s_0) \wedge \bigwedge_{(q,\mathrm{cond},q')\in\Delta} \bigwedge_{s\in 2^S} \bigwedge_{i\in 2^I} \lambda_B(q,s) \wedge i \wedge s \wedge \mathrm{cond} \to \lambda_B(q',\tau(s,i)) \wedge \lambda_\#(q',\tau(s,i)) \rhd \lambda_\#(q,s)$$

Intuitively, the SMT-encoding assigns the states of the universal co-Büchi automaton to the states of the processes. The enumeration of the process-states ensures that each state a rejecting state of the universals co-Büchi automaton is assigned to, is not visited infinitely often during an execution of the process. It is only possible to visit a state infinitely often during an execution if the transition relation contains a cycle that contains this state. The $>$-operator ensures that such a cycle is not possible.

Chapter 3

Bounded Assume-Guarantee Synthesis

This chapter describes how the the AGS problem is solved using bounded synthesis.

3.1. Constraint based AGS

As described in Section 2.3.2.3 the AGS-problem is defined as following: Do there exist two processes for the given process descriptions $P_1' \preceq P_1$ and $P_2' \preceq P_2$, such that the following conditions are valid for all fair schedulers:

$$\begin{aligned}(i)\ [[P_1'||P_2'||C]] \upharpoonright_{(S\cup I)} &\models \phi_1 \wedge \phi_2,\\ (ii)\ [[P_1'||P_2||C]] \upharpoonright_{(S\cup I)} &\models \phi_2 \rightarrow \phi_1 \text{ and}\\ (iii)\ [[P_1||P_2'||C]] \upharpoonright_{(S\cup I)} &\models \phi_1 \rightarrow \phi_2\end{aligned}$$

Our constraint-based approach to solve the AGS-problem is based on the bounded synthesis encoding described in Section 2.3.3. As shown in Figure 3.1, we firstly negate the AGS conditions. These conditions are afterwards translated into universal co-Büchi automata.

These automata are afterwards SMT-encoded using the SMT-encoding function of the bounded synthesis approach. As this approach only handles systems containing a single process, we extend the algorithm by redefining the state space, the inputs and the τ-function to handle two processes and the scheduler. Furthermore, this refinement allows us to define private inputs and variables as well as global inputs and variables.

Consequently, the encoding of the AGS-constraints argues about three different state spaces: Two state spaces represent the synthesized process and the other state space represents the combined state space of the two processes. This state space defines the whole system given the two synthesized processes. Consistency throughout the various state spaces is maintained by utilizing the transition functions of each state space.

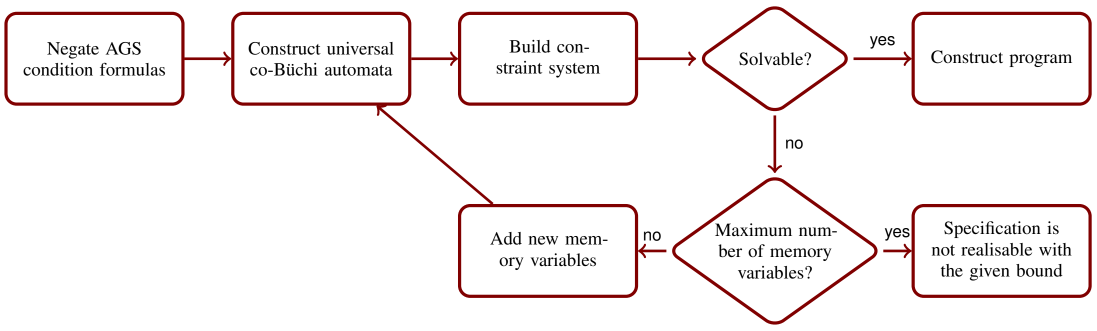
Negate AGS condition formulas
Construct universal co-Büchi automata
Build constraint system
Solvable?
yes
Construct program
no
Maximum number of memory variables?
yes
Specification is not realisable with the given bound
no
Add new memory variables

Figure 3.1.: AGS overview

The transition functions of each individual process describes the state-transitions for this process. The transition functions of the combined processes are, however, more elaborate. This function takes process scheduling into consideration: For each state within the combined process, transitions are defined for each possible scheduler decision.

If the resulting constraint system is satisfiable, a model of this solution can be used to construct automatically the code of the processes. In particular, the model of the transition function can be used to model the process logic.

However, if the resulting constraint system is not satisfiable, it is possible to add memory variables. These memory variables cause the extension of the state space but are not used in the process specification of the system designer. Because of the additional variables, the transition function can use these variables freely and can for example define counter variables. In contrast to cooperative co-synthesis, these memory variables have to be private for each process, as the specification does not argue about these variables. Therefore, the processes can set these variables freely and the other process cannot rely on the valuation of the variable.

Example 1. As an example to show the usage of the memory variables, consider the case where the specifications of both processes state that a bit has to be flipped in every second tick and stay the same otherwise. This system is only realisable if memory variables are added. A possible way to construct the system is to use a memory bit as a counter that only counts to 1. If the memory bit is 1, the other bit can be flipped at the next tick, otherwise the bit stays the same. A possible trace would be:

global variable	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	...
memory variable	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	...

Table 3.1.: Trace of example 1

The following section describes the redefined synthesizing method for each AGS condition in detail.

Example 2. As a running example, consider the following specification of a system: Let r be the global input, g the global output, m the private variable of the first process and p the private variable of the second process.

The specification of the first process is to eventually set the global variable g permanently to true if the system receives a request r.

In contrast to the global variable g, the private variable m of the first process is independent of the request. Eventually, this variable has to be set to false permanently.

The specification of the second process defines the global variable like the first process. Additionally, it specifies the private variable p that has also to be always true after a certain point in the execution if the system receives a request r.

The two process-specification are formulated with LTL as follows:

- Specification for the first process: $\mathbf{G}((r \rightarrow \mathbf{F}(\mathbf{G}g)) \wedge \mathbf{F}(\mathbf{G}\neg m))$
- Specification for the second process: $\mathbf{G}(r \rightarrow (\mathbf{F}(\mathbf{G}(g \wedge p))))$

Additionally, the scheduler is defined as an input variable c that is true if the first process is scheduled and false if the the second process is scheduled. The fairness condition to ensure that every process is scheduled infinitely often can be described in LTL as $\mathbf{G}(\mathbf{F}c \wedge \mathbf{F}\neg c)$. Furthermore all variables are initialized to false.

3.2. Conditions

3.2.1. First condition

The first condition $([[P_1'||P_2'||C]] \upharpoonright_{(S \cup I)} \models \phi_1 \wedge \phi_2)$ defines that the synthesized processes have to fulfil the given LTL specifications. This condition also defines the cooperative co-synthesis problem for this specification.

However, to construct the constraint system for this condition, we firstly have to translate the condition into a universal co-Büchi automaton. Since LTL-formulas can be translated into universal co-Büchi automata, we define the following formula, where ϕ_i defines the specifications of the processes and f the specification of the fair scheduler:

$$f \rightarrow (\phi_1 \wedge \phi_2)$$

This LTL-formula is translated into a universal co-Büchi automaton, which rejects a run if there exists a point in the execution of the automaton where at least one specification of a process is not fulfilled, even though the scheduler is fair.

Figure 3.2 shows the universal co-Büchi automaton for Example 2. As the fairness condition blows up the universal co-Büchi automaton without adding necessary information to understand the example, Figure 3.2 does not contain the fairness condition. The automaton that contains the fairness condition can be found in the Appendix in Figure A.1.

The left branch of the minimized automaton defines the behaviour of the variables which are influenced by the input, whereas the right branch defines the behaviour of the private variable of the first process, which is independent from the input. The automaton accepts an execution if the system never receives a request and m is set accordingly. If the system receives a request also the private variable p and the global variable g have to be eventually set to true, to fulfil the specification. The construction of the system is possible if the rejecting states in the universal co-Büchi automaton are not visited infinitely often. As it is not possible for

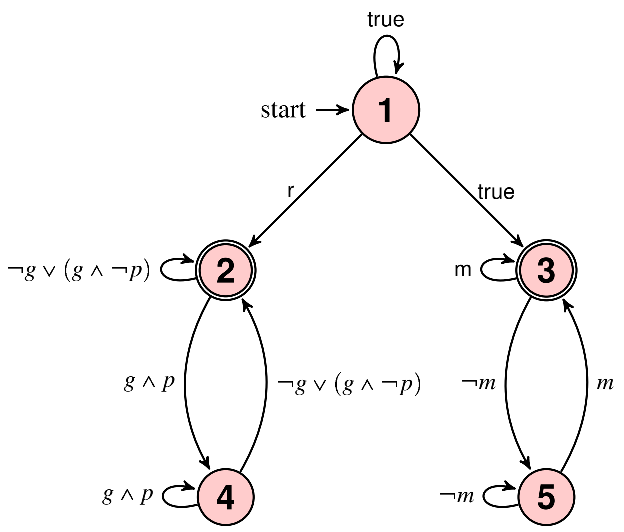

Figure 3.2.: Simplified universal co-Büchi automaton of the first AGS-constraint for Example 2

the system to influence the input variables or the scheduler, the private and global variables have to be set accordingly.

To translate the universal co-Büchi automaton into the constraint system we use the SMT-encoding function $\text{SMT}(S_1 \cup S_2, (I_g, I_1, I_2, C), \tau_{1'2'}, \Delta_1)$ of the bounded synthesis approach and define the arguments accordingly:

- S is the compound state space and S_1 and S_2 are the state spaces of the processes. A state is defined by the valuation of the global variables G, the local variables L_i and the memory variables M_i of each process.
- I_g is a set of global input variables and I_1 and I_2 are sets of private process-input variables, i.e. $S_i = G \cup L_i \cup M_i$
- c is the scheduler variable, which is seen by the system as an input variable but is handled in the transition function separately.
- $\tau_{1'2'}$ is an uninterpreted function which defines the transition function of the system. Based on the scheduler and the current valuation of the input variables, it defines for each state the subsequent states: $2^{S_1 \cup S_2} \times 2^I \times 2^C \to 2^{S_1 \cup S_2}$. To define these states, the function uses the uninterpreted functions τ_1 and τ_2, which define the transition relation of the processes of P_1' and P_2'. The function is defined as follows:

$$\tau_{1'2'}(s, (i_g, i_1, i_2, c)) = \begin{cases} \tau_1(s \restriction_{G \cup L_1 \cup M_1}, i_g \cup i_1) \cup (s \restriction_{L_2 \cup M_2}) & \text{if } c = 1 \\ \tau_2(s \restriction_{G \cup L_2 \cup M_2}, i_g \cup i_2) \cup (s \restriction_{L_1 \cup M_1}) & \text{if } c = 2 \end{cases}$$

Intuitively this function updates the global variables, the private variables and the memory variables of P_1 and copies the private and memory variables of the second process if the first process is scheduled, and vice versa if the second process is scheduled.

- Δ_1 is the transition relation of the universal co-Büchi automaton for $f \rightarrow (\phi_1 \wedge \phi_2)$

The combined state space in Example 2 is defined by the valuation of the variables p, m and g. The state space of the first process is defined by the valuation of the global variable g and the private variable m. The valuations of the global variable g and the private variable p define the state space of the second process. The input is defined by the variable r and the scheduler variable c.

Listing 3.1 shows the constraints for the this AGS condition for the minimized Example 2. The first line specifies a set of all possible valuations of the global input. The next line describes the combined state space of the system. The state spaces of the processes are implicitly defined by their transition functions. The third line defines the start evaluation. The lines 4 to 14 are the SMT-encoded transitions of the universal co-Büchi automaton.

Listing 3.1: Constraint-system of the first minimized universal co-Büchi automaton of Example 2

1 $I_g = \{r,\bar{r}\}$

2 $S = \{gmp, \bar{g}mp, g\bar{m}p, gm\bar{p}, \bar{g}\bar{m}p, g\bar{m}\bar{p}, \bar{g}m\bar{p}, \bar{g}\bar{m}\bar{p}\}$

3 $\lambda_B(q_1, \bar{g}\bar{m}\bar{p})$

4 $\forall s \in S. \forall i \in I_g.\ \lambda_B(q_1,s) \rightarrow \lambda_B(q_1,\tau_{1'2'}(s,(i,1))) \wedge \lambda_\#(q_1,\tau_{1'2'}(s,(i,1))) \geqslant \lambda_\#(q_1,s) \wedge$
$\lambda_B(q_1,\tau_{1'2'}(s,(i,2))) \wedge \lambda_\#(q_1,\tau_{1'2'}(s,(i,2))) \geqslant \lambda_\#(q_1,s)$

5 $\forall s \in S. \forall i \in I_g.\ \lambda_B(q_1,s) \wedge s \wedge i \wedge r \rightarrow \lambda_B(q_2,\tau_{1'2'}(s,(i,1))) \wedge \lambda_\#(q_2,\tau_{1'2'}(s,(i,1))) > \lambda_\#(q_1,s) \wedge$
$\lambda_B(q_2,\tau_{1'2'}(s,(i,2))) \wedge \lambda_\#(q_2,\tau_{1'2'}(s,(i,2))) > \lambda_\#(q_1,s)$

6 $\forall s \in S. \forall i \in I_g.\ \lambda_B(q_1,s) \rightarrow \lambda_B(q_3,\tau_{1'2'}(s,(i,1))) \wedge \lambda_\#(q_3,\tau_{1'2'}(s,(i,1))) > \lambda_\#(q_1,s) \wedge$
$\lambda_B(q_3,\tau_{1'2'}(s,(i,2))) \wedge \lambda_\#(q_3,\tau_{1'2'}(s,(i,2))) > \lambda_\#(q_1,s)$

7 $\forall s \in S. \forall i \in I_g.\ \lambda_B(q_2,s) \wedge s \wedge i \wedge (\neg g \vee (g \wedge \neg p)) \rightarrow \lambda_B(q_2,\tau_{1'2'}(s,(i,1))) \wedge \lambda_\#(q_2,\tau_{1'2'}(s,(i,1))) > \lambda_\#(q_2,s) \wedge$
$\lambda_B(q_2,\tau_{1'2'}(s,(i,2))) \wedge \lambda_\#(q_2,\tau_{1'2'}(s,(i,2))) > \lambda_\#(q_2,s)$

8 $\forall s \in S. \forall i \in I_g.\ \lambda_B(q_2,s) \wedge s \wedge i \wedge (g \wedge p)) \rightarrow \lambda_B(q_4,\tau_{1'2'}(s,(i,1))) \wedge \lambda_\#(q_4,\tau_{1'2'}(s,(i,1))) \geqslant \lambda_\#(q_2,s) \wedge$
$\lambda_B(q_4,\tau_{1'2'}(s,(i,2))) \wedge \lambda_\#(q_4,\tau_{1'2'}(s,(i,2))) \geqslant \lambda_\#(q_2,s)$

9 $\forall s \in S. \forall i \in I_g.\ \lambda_B(q_3,s) \wedge s \wedge i \wedge m \rightarrow \lambda_B(q_3,\tau_{1'2'}(s,(i,1))) \wedge \lambda_\#(q_3,\tau_{1'2'}(s,(i,1))) > \lambda_\#(q_3,s) \wedge$
$\lambda_B(q_3,\tau_{1'2'}(s,(i,2))) \wedge \lambda_\#(q_3,\tau_{1'2'}(s,(i,2))) > \lambda_\#(q_3,s)$

10 $\forall s \in S. \forall i \in I_g.\ \lambda_B(q_3,s) \wedge s \wedge i \wedge \neg m \rightarrow \lambda_B(q_5,\tau_{1'2'}(s,(i,1))) \wedge \lambda_\#(q_5,\tau_{1'2'}(s,(i,1))) \geqslant \lambda_\#(q_3,s) \wedge$
$\lambda_B(q_5,\tau_{1'2'}(s,(i,2))) \wedge \lambda_\#(q_5,\tau_{1'2'}(s,(i,2))) \geqslant \lambda_\#(q_3,s)$

11 $\forall s \in S. \forall i \in I_g.\ \lambda_B(q_4,s) \wedge s \wedge i \wedge (g \wedge p) \rightarrow \lambda_B(q_4,\tau_{1'2'}(s,(i,1))) \wedge \lambda_\#(q_4,\tau_{1'2'}(s,(i,1))) \geqslant \lambda_\#(q_4,s) \wedge$
$\lambda_B(q_4,\tau_{1'2'}(s,(i,2))) \wedge \lambda_\#(q_4,\tau_{1'2'}(s,(i,2))) \geqslant \lambda_\#(q_4,s)$

12 $\forall s \in S. \forall i \in I_g.\ \lambda_B(q_4,s) \wedge s \wedge i \wedge (\neg g \vee (g \wedge \neg p)) \rightarrow \lambda_B(q_2,\tau_{1'2'}(s,(i,1))) \wedge \lambda_\#(q_2,\tau_{1'2'}(s,(i,1))) > \lambda_\#(q_4,s) \wedge$
$\lambda_B(q_2,\tau_{1'2'}(s,(i,2))) \wedge \lambda_\#(q_2,\tau_{1'2'}(s,(i,2))) > \lambda_\#(q_4,s)$

13 $\forall s \in S. \forall i \in I_g.\ \lambda_B(q_5,s) \wedge s \wedge i \wedge \neg m \rightarrow \lambda_B(q_5,\tau_{1'2'}(s,(i,1))) \wedge \lambda_\#(q_5,\tau_{1'2'}(s,(i,1))) \geqslant \lambda_\#(q_5,s) \wedge$
$\lambda_B(q_5,\tau_{1'2'}(s,(i,2))) \wedge \lambda_\#(q_5,\tau_{1'2'}(s,(i,2))) \geqslant \lambda_\#(q_5,s)$

14 $\forall s \in S. \forall i \in I_g.\ \lambda_B(q_5,s) \wedge s \wedge i \wedge m \rightarrow \lambda_B(q_3,\tau_{1'2'}(s,(i,1))) \wedge \lambda_\#(q_3,\tau_{1'2'}(s,(i,1))) > \lambda_\#(q_5,s) \wedge$
$\lambda_B(q_3,\tau_{1'2'}(s,(i,2))) \wedge \lambda_\#(q_3,\tau_{1'2'}(s,(i,2))) > \lambda_\#(q_5,s)$

3.2.2. Second condition

The second constraint ($[[P_1' || P_2 || C]] \upharpoonright_{(S \cup I)} \models \phi_2 \rightarrow \phi_1$) ensures that P_1' is robust and is therefore able to react on all possible processes which fulfil the specification for the second process. This is ensured as the constraint does not argue about the second process itself but about the process description. Therefore the transition relation is not fixed and P_1' has to be able to react on all possible behaviours of the second process

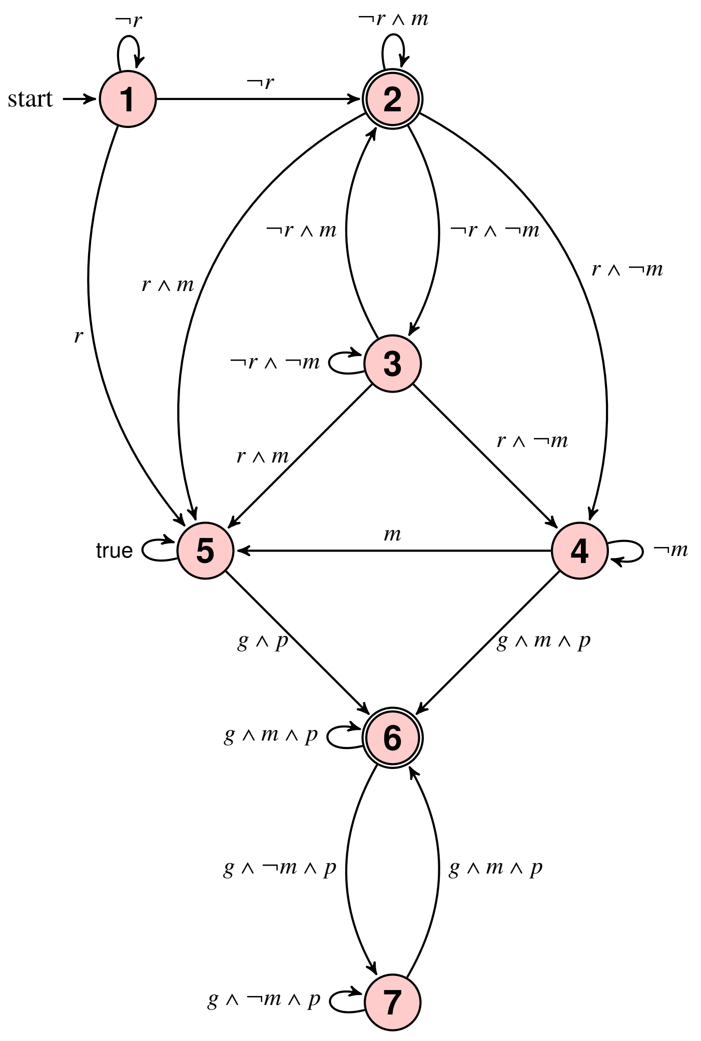

Figure 3.3.: Simplified universal co-Büchi automaton of the second AGS-constraint for Example 2

that fulfil its specification. This restriction is caused by the implication which ensures that the constraint is valid if the specification ϕ_2 is not fulfilled even though P_1' may not fulfil its specification. This condition allows the designer to exchange the second process in the system with another process that also fulfils the specification ϕ_2.

Like for the first condition, we translate the LTL specification into a universal co-Büchi automaton. Here, ϕ_i also defines the specifications of the processes and f defines the fair scheduler. The resulting LTL-formula is encoded as follows:

$$f \rightarrow (\phi_2 \rightarrow \phi_1)$$

Figure 3.3 shows the universal co-Büchi automaton for example 2 of this second constraint. Like above, to keep the figure simple, the universal co-Büchi automaton does not consider the fairness condition. A universal co-Büchi automaton including the fairness condition can be found in the Appendix in Figure A.2

As both processes have the same specification for the global variable, the universal co-Büchi automaton mainly argues about the private variable m. State 2 rejects a run if the system does not send a request and the private variable m is eventually not permanently set to false. The other rejecting state is state 6. Here the automaton rejects a run if the system receives a request and sets the variables g and p accordingly but the variable m is not eventually set permanently to false. Please note that due to the implication there does not exist a rejecting state where m is set correctly but g or p are not set accordingly.

To encode the universal co-Büchi automaton, we are also using the SMT-encoding function $\mathrm{SMT}(S,(I_g,I_1,S'_2,C),\tau_{1'2},\Delta_2)$ of the bounded synthesis approach. The components S, I_g, I_1 and C are defined as above. Δ_2 is the transition relation of the universal co-Büchi automaton for the formula $f \rightarrow (\phi_1 \rightarrow \phi_2)$. S'_2 are inputs defining the new values of the state variables of P_2. The function $\tau_{1'2}$, also defines for each state the subsequent state: $2^{S_1\cup S_2} \times 2^{I_g\cup I_1\cup S'_2\cup C} \rightarrow 2^{S_1\cup S_2}$. Using the uninterpreted function τ_1 it can be defined in the following way:

$$\tau_{1'2}(s,(i_g,i_1,s',c)) = \begin{cases} \tau_1(s\restriction_{G\cup L_1\cup M_1}, i_g\cup i_1)\cup s\restriction_{L_2\cup M_2} & \text{if } c=1 \\ s\restriction_{L_1\cup M_1}\cup s'\restriction_{G\cup L_2\cup M_2} & \text{if } c=2 \end{cases}.$$

Like the function of the first constraint, $\tau_{1'2}$ updates the global variables and the local variables of the first process if this process was scheduled. But in contrast to the $\tau_{1'2'}$ of the first constraint, the variables of the second process are not updated according to the transition system of the second process. Instead they are updated with fresh input variables. Therefore, the first process has to react on all possible valuations of s' if the valuation does not contradict the specification of the second process.

Listing 3.2 shows the SMT-encoding of the universal co-Büchi automaton in Figure 3.3. As in Listing 3.1, the first two lines specify all possible valuations of the input and the combined state space. The third line defines the fresh input variables for the transition function $\tau_{1'2}$. Line four defines the start valuation. The other lines are the SMT-encoded transitions of the universal co-Büchi automaton.

Listing 3.2: Constraint-system of the simplified universal co-Büchi automaton of Figure 3.3

1 $I_g = \{r,\bar{r}\}$

2 $S = \{gmp,\bar{g}mp,g\bar{m}p,gm\bar{p},\bar{g}\bar{m}p,g\bar{m}\bar{p},\bar{g}m\bar{p},\bar{g}\bar{m}\bar{p}\}$

3 $S' = \{gp,\bar{g}p,g\bar{p},\bar{g}\bar{p}\}$

4 $\lambda_B(q_1,\bar{g}\bar{m}\bar{p})$

5 $\forall s\in S.\forall s'\in S'.\forall i\in I_g.\ \lambda_B(q_1,s)\wedge s\wedge i\wedge\neg r\rightarrow\lambda_B(q_1,\tau_{1'2}(s,(i,s',1)))\wedge\lambda_\#(q_1,\tau_{1'2}(s,(i,s',1)))\geqslant\lambda_\#(q_1,s)\wedge$
$\lambda_B(q_1,\tau_{1'2}(s,(i,s',2)))\wedge\lambda_\#(q_1,\tau_{1'2}(s,(i,s',2)))\geqslant\lambda_\#(q_1,s)$

6 $\forall s\in S.\forall s'\in S'.\forall i\in I_g.\ \lambda_B(q_1,s)\wedge s\wedge i\wedge\neg r\rightarrow\lambda_B(q_2,\tau_{1'2}(s,(i,s',1)))\wedge\lambda_\#(q_2,\tau_{1'2}(s,(i,s',1)))>\lambda_\#(q_1,s)\wedge$
$\lambda_B(q_2,\tau_{1'2}(s,(i,s',2)))\wedge\lambda_\#(q_2,\tau_{1'2}(s,(i,s',2)))>\lambda_\#(q_1,s)$

7 $\forall s\in S.\forall s'\in S'.\forall i\in I_g.\ \lambda_B(q_1,s)\wedge s\wedge i\wedge r\rightarrow\lambda_B(q_5,\tau_{1'2}(s,(i,s',1)))\wedge\lambda_\#(q_5,\tau_{1'2}(s,(i,s',1)))\geqslant\lambda_\#(q_1,s)\wedge$
$\lambda_B(q_5,\tau_{1'2}(s,(i,s',2)))\wedge\lambda_\#(q_5,\tau_{1'2}(s,(i,s',2)))\geqslant\lambda_\#(q_1,s)$

8 $\forall s\in S.\forall s'\in S'.\forall i\in I_g.\ \lambda_B(q_2,s)\wedge s\wedge i\wedge(\neg r\wedge m)\rightarrow\lambda_B(q_2,\tau_{1'2}(s,(i,s',1)))\wedge\lambda_\#(q_2,\tau_{1'2}(s,(i,s',1)))>\lambda_\#(q_2,s)\wedge$
$\lambda_B(q_2,\tau_{1'2}(s,(i,s',2)))\wedge\lambda_\#(q_2,\tau_{1'2}(s,(i,s',2)))>\lambda_\#(q_2,s)$

9 $\forall s\in S.\forall s'\in S'.\forall i\in I_g.\ \lambda_B(q_2,s)\wedge s\wedge i\wedge(\neg r\wedge\neg m)\rightarrow\lambda_B(q_3,\tau_{1'2}(s,(i,s',1)))\wedge\lambda_\#(q_3,\tau_{1'2}(s,(i,s',1)))\geqslant\lambda_\#(q_2,s)\wedge$
$\lambda_B(q_3,\tau_{1'2}(s,(i,s',2)))\wedge\lambda_\#(q_3,\tau_{1'2}(s,(i,s',2)))\geqslant\lambda_\#(q_2,s)$

10 $\forall s\in S.\forall s'\in S'.\forall i\in I_g.\ \lambda_B(q_2,s)\wedge s\wedge i\wedge(r\wedge\neg m)\rightarrow\lambda_B(q_4,\tau_{1'2}(s,(i,s',1)))\wedge\lambda_\#(q_4,\tau_{1'2}(s,(i,s',1)))\geqslant\lambda_\#(q_2,s)\wedge$
$\lambda_B(q_4,\tau_{1'2}(s,(i,s',2)))\wedge\lambda_\#(q_4,\tau_{1'2}(s,(i,s',2)))\geqslant\lambda_\#(q_2,s)$

11 $\forall s \in S.\forall s' \in S'.\forall i \in I_g.\ \lambda_B(q_2,s) \wedge s \wedge i \wedge (r \wedge m) \rightarrow \lambda_B(q_5,\tau_{1'2}(s,(i,s',1))) \wedge \lambda_\#(q_5,\tau_{1'2}(s,(i,s',1))) \geqslant \lambda_\#(q_2,s) \wedge$
$\lambda_B(q_5,\tau_{1'2}(s,(i,s',2))) \wedge \lambda_\#(q_5,\tau_{1'2}(s,(i,s',2))) \geqslant \lambda_\#(q_2,s)$

12 $\forall s \in S.\forall s' \in S'.\forall i \in I_g.\ \lambda_B(q_3,s) \wedge s \wedge i \wedge (\neg r \wedge \neg m) \rightarrow \lambda_B(q_3,\tau_{1'2}(s,(i,s',1))) \wedge \lambda_\#(q_3,\tau_{1'2}(s,(i,s',1))) \geqslant \lambda_\#(q_3,s) \wedge$
$\lambda_B(q_3,\tau_{1'2}(s,(i,s',2))) \wedge \lambda_\#(q_3,\tau_{1'2}(s,(i,s',2))) \geqslant \lambda_\#(q_3,s)$

13 $\forall s \in S.\forall s' \in S'.\forall i \in I_g.\ \lambda_B(q_3,s) \wedge s \wedge i \wedge (\neg r \wedge m) \rightarrow \lambda_B(q_2,\tau_{1'2}(s,(i,s',1))) \wedge \lambda_\#(q_2,\tau_{1'2}(s,(i,s',1))) > \lambda_\#(q_3,s) \wedge$
$\lambda_B(q_2,\tau_{1'2}(s,(i,s',2))) \wedge \lambda_\#(q_2,\tau_{1'2}(s,(i,s',2))) > \lambda_\#(q_3,s)$

14 $\forall s \in S.\forall s' \in S'.\forall i \in I_g.\ \lambda_B(q_3,s) \wedge s \wedge i \wedge (r \wedge \neg m) \rightarrow \lambda_B(q_4,\tau_{1'2}(s,(i,s',1))) \wedge \lambda_\#(q_4,\tau_{1'2}(s,(i,s',1))) \geqslant \lambda_\#(q_3,s) \wedge$
$\lambda_B(q_4,\tau_{1'2}(s,(i,s',2))) \wedge \lambda_\#(q_4,\tau_{1'2}(s,(i,s',2))) \geqslant \lambda_\#(q_3,s)$

15 $\forall s \in S.\forall s' \in S'.\forall i \in I_g.\ \lambda_B(q_3,s) \wedge s \wedge i \wedge (r \wedge m) \rightarrow \lambda_B(q_5,\tau_{1'2}(s,(i,s',1))) \wedge \lambda_\#(q_5,\tau_{1'2}(s,(i,s',1))) \geqslant \lambda_\#(q_3,s) \wedge$
$\lambda_B(q_5,\tau_{1'2}(s,(i,s',2))) \wedge \lambda_\#(q_5,\tau_{1'2}(s,(i,s',2))) \geqslant \lambda_\#(q_3,s)$

16 $\forall s \in S.\forall s' \in S'.\forall i \in I_g.\ \lambda_B(q_4,s) \wedge s \wedge i \wedge \neg m \rightarrow \lambda_B(q_4,\tau_{1'2}(s,(i,s',1))) \wedge \lambda_\#(q_4,\tau_{1'2}(s,(i,s',1))) \geqslant \lambda_\#(q_4,s) \wedge$
$\lambda_B(q_4,\tau_{1'2}(s,(i,s',2))) \wedge \lambda_\#(q_4,\tau_{1'2}(s,(i,s',2))) \geqslant \lambda_\#(q_4,s)$

17 $\forall s \in S.\forall s' \in S'.\forall i \in I_g.\ \lambda_B(q_4,s) \wedge s \wedge i \wedge m \rightarrow \lambda_B(q_5,\tau_{1'2}(s,(i,s',1))) \wedge \lambda_\#(q_5,\tau_{1'2}(s,(i,s',1))) \geqslant \lambda_\#(q_4,s) \wedge$
$\lambda_B(q_5,\tau_{1'2}(s,(i,s',2))) \wedge \lambda_\#(q_5,\tau_{1'2}(s,(i,s',2))) \geqslant \lambda_\#(q_4,s)$

18 $\forall s \in S.\forall s' \in S'.\forall i \in I_g.\ \lambda_B(q_4,s) \wedge s \wedge i \wedge (g \wedge m \wedge p) \rightarrow \lambda_B(q_6,\tau_{1'2}(s,(i,s',1))) \wedge \lambda_\#(q_6,\tau_{1'2}(s,(i,s',1))) > \lambda_\#(q_4,s) \wedge$
$\lambda_B(q_6,\tau_{1'2}(s,(i,s',2))) \wedge \lambda_\#(q_6,\tau_{1'2}(s,(i,s',2))) > \lambda_\#(q_4,s)$

19 $\forall s \in S.\forall s' \in S'.\forall i \in I_g.\ \lambda_B(q_5,s) \rightarrow \lambda_B(q_5,\tau_{1'2}(s,(i,s',1))) \wedge \lambda_\#(q_5,\tau_{1'2}(s,(i,s',1))) \geqslant \lambda_\#(q_5,s) \wedge$
$\lambda_B(q_5,\tau_{1'2}(s,(i,s',2))) \wedge \lambda_\#(q_5,\tau_{1'2}(s,(i,s',2))) \geqslant \lambda_\#(q_5,s)$

20 $\forall s \in S.\forall s' \in S'.\forall i \in I_g.\ \lambda_B(q_5,s) \wedge s \wedge i \wedge (g \wedge p) \rightarrow \lambda_B(q_6,\tau_{1'2}(s,(i,s',1))) \wedge \lambda_\#(q_6,\tau_{1'2}(s,(i,s',1))) > \lambda_\#(q_5,s) \wedge$
$\lambda_B(q_6,\tau_{1'2}(s,(i,s',2))) \wedge \lambda_\#(q_6,\tau_{1'2}(s,(i,s',2))) > \lambda_\#(q_5,s)$

21 $\forall s \in S.\forall s' \in S'.\forall i \in I_g.\ \lambda_B(q_6,s) \wedge s \wedge i \wedge (g \wedge m \wedge p) \rightarrow \lambda_B(q_6,\tau_{1'2}(s,(i,s',1))) \wedge \lambda_\#(q_6,\tau_{1'2}(s,(i,s',1))) > \lambda_\#(q_6,s) \wedge$
$\lambda_B(q_6,\tau_{1'2}(s,(i,s',2))) \wedge \lambda_\#(q_6,\tau_{1'2}(s,(i,s',2))) > \lambda_\#(q_6,s)$

22 $\forall s \in S.\forall s' \in S'.\forall i \in I_g.\ \lambda_B(q_6,s) \wedge s \wedge i \wedge (g \wedge \neg m \wedge p) \rightarrow \lambda_B(q_7,\tau_{1'2}(s,(i,s',1))) \wedge \lambda_\#(q_7,\tau_{1'2}(s,(i,s',1))) \geqslant \lambda_\#(q_6,s) \wedge$
$\lambda_B(q_7,\tau_{1'2}(s,(i,s',2))) \wedge \lambda_\#(q_7,\tau_{1'2}(s,(i,s',2))) \geqslant \lambda_\#(q_6,s)$

23 $\forall s \in S.\forall s' \in S'.\forall i \in I_g.\ \lambda_B(q_7,s) \wedge s \wedge i \wedge (g \wedge \neg m \wedge p) \rightarrow \lambda_B(q_7,\tau_{1'2}(s,(i,s',1))) \wedge \lambda_\#(q_7,\tau_{1'2}(s,(i,s',1))) \geqslant \lambda_\#(q_7,s) \wedge$
$\lambda_B(q_7,\tau_{1'2}(s,(i,s',2))) \wedge \lambda_\#(q_7,\tau_{1'2}(s,(i,s',2))) \geqslant \lambda_\#(q_7,s)$

24 $\forall s \in S.\forall s' \in S'.\forall i \in I_g.\ \lambda_B(q_7,s) \wedge s \wedge i \wedge (g \wedge m \wedge p) \rightarrow \lambda_B(q_6,\tau_{1'2}(s,(i,s',1))) \wedge \lambda_\#(q_6,\tau_{1'2}(s,(i,s',1))) > \lambda_\#(q_7,s) \wedge$
$\lambda_B(q_6,\tau_{1'2}(s,(i,s',2))) \wedge \lambda_\#(q_6,\tau_{1'2}(s,(i,s',2))) > \lambda_\#(q_7,s)$

3.2.3. Third constraint

The third constraint $[[P_1||P_2'||C]] \upharpoonright (S \cup I) \models \phi_1 \rightarrow \phi_2$ ensures the robustness of P_2' like the second constraint for P_1'. Also here the process description is used to ensure the robustness and the implication restricts the freedom of the valuations.

As above, the LTL specification is translated into a universal co-Büchi automaton. Like before, f defines the fair scheduler and ϕ_i defines the specification of process i. Therefore, the LTL-formula can be defined as follows:

$$f \rightarrow (\phi_1 \rightarrow \phi_2)$$

Figure 3.4 shows the universal co-Büchi automaton for this constraint of example 2. Also here, we neglect the fairness condition to keep the automaton simple. The whole automaton can be found in the Appendix in Figure A.3.

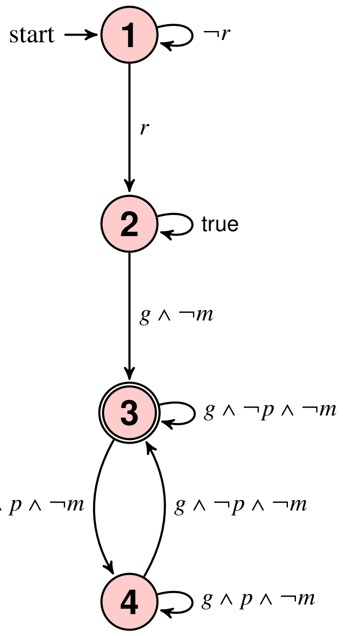

Figure 3.4.: Simplified universal co-Büchi automaton of the third AGS-constraint for Example 2

The simplified automaton only rejects a run if the private variable p is not set accordingly. Because of the implication, an invalid valuations of the global variable g and the private variable m do not lead to failing runs. Therefore the only rejecting state is state three, where all variables but the private variable p are set correctly.

Like the transition functions of the first and second constraint, the transition function $\tau_{12'}$ updates the variables in the combined state space by defining the relation.

$$\tau_{12'}(s,(i_g,i_2,s',c)) = \begin{cases} (s' \upharpoonright_{G\cup L_1\cup M_1}) \cup (s \upharpoonright_{L_2\cup M_2}) & \text{if } c = 1 \\ (s \upharpoonright_{L_1\cup M_1}) \cup \tau_2(s \upharpoonright_{G\cup L_2\cup M_2}, i_g \cup i_2) & \text{if } c = 2 \end{cases}$$

Intuitively this function updates the state space according to the transition function of the second process if this process was scheduled. Otherwise the variables of the second process stay the same and the variables of the first process and the global variables are set to the value of fresh input variables s', as long as they do not contradict the specification of the first process.

Using this function the transitions can be encoded using the SMT-encoding function $\mathrm{SMT}(S_{12}, I_g \cup I_2 \cup C \cup S'_1, \tau_{12'}, \Delta_3)$, where S_{12} is the combined state space, I_g is defined as the global input and I_2 is the input of the second process, S'_1 contains fresh inputs defining the new state of the first process and Δ_3 is a set of transitions of the current universal co-Büchi automaton.

The SMT-encoding of the simplified automaton can be found in Listing 3.3. As above, the first four sates define all possible valuations of the global input variable, the combined state space, all valuations of the fresh input variables and the start valuation. Compared to the listing above, the fresh input variables represent arbitrary valuations for the global variables and the private variable of the first process. Line 5 - 12 represent the SMT-encoding of the transitions of the universal co-Büchi automaton in Figure 3.4. As the forall-quantifier only argue about finite sets of assignments, the formula is decidable.

Listing 3.3: Constraint-system of the simplified universal co-Büchi automaton of Figure 3.4

1 $I_g = \{r, \bar{r}\}$

2 $S = \{gmp, \bar{g}mp, g\bar{m}p, gm\bar{p}, \bar{g}\bar{m}p, g\bar{m}\bar{p}, \bar{g}m\bar{p}, \bar{g}\bar{m}\bar{p}\}$

3 $S' = \{gm, \bar{g}m, g\bar{m}, \bar{g}\bar{m}\}$

4 $\lambda_B(q_1, \bar{g}\bar{m}\bar{p})$

5 $\forall s \in S. \forall s' \in S'. \forall i \in I_g.\ \lambda_B(q_1, s) \wedge s \wedge i \wedge \neg r \rightarrow \lambda_B(q_1, \tau_{12'}(s, (i, s', 1))) \wedge \lambda_\#(q_1, \tau_{12'}(s, (i, s', 1))) \geqslant \lambda_\#(q_1, s) \wedge$
$\lambda_B(q_1, \tau_{12'}(s, (i, s', 2))) \wedge \lambda_\#(q_1, \tau_{12'}(s, (i, s', 2))) \geqslant \lambda_\#(q_1, s)$

6 $\forall s \in S. \forall s' \in S'. \forall i \in I_g.\ \lambda_B(q_1, s) \wedge s \wedge i \wedge r \rightarrow \lambda_B(q_2, \tau_{12'}(s, (i, s', 1))) \wedge \lambda_\#(q_2, \tau_{12'}(s, (i, s', 1))) \geqslant \lambda_\#(q_1, s) \wedge$
$\lambda_B(q_2, \tau_{12'}(s, (i, s', 2))) \wedge \lambda_\#(q_2, \tau_{12'}(s, (i, s', 2))) \geqslant \lambda_\#(q_1, s)$

7 $\forall s \in S. \forall s' \in S'. \forall i \in I_g.\ \lambda_B(q_2, s) \rightarrow \lambda_B(q_2, \tau_{12'}(s, (i, s', 1))) \wedge \lambda_\#(q_2, \tau_{12'}(s, (i, s', 1))) \geqslant \lambda_\#(q_2, s) \wedge$
$\lambda_B(q_2, \tau_{12'}(s, (i, s', 2))) \wedge \lambda_\#(q_2, \tau_{12'}(s, (i, s', 2))) \geqslant \lambda_\#(q_2, s)$

8 $\forall s \in S. \forall s' \in S'. \forall i \in I_g.\ \lambda_B(q_2, s) \wedge s \wedge i \wedge (g \wedge \neg m) \rightarrow \lambda_B(q_3, \tau_{12'}(s, (i, s', 1))) \wedge \lambda_\#(q_3, \tau_{12'}(s, (i, s', 1))) > \lambda_\#(q_2, s) \wedge$
$\lambda_B(q_3, \tau_{12'}(s, (i, s', 2))) \wedge \lambda_\#(q_3, \tau_{12'}(s, (i, s', 2))) > \lambda_\#(q_2, s)$

9 $\forall s \in S. \forall s' \in S'. \forall i \in I_g.\ \lambda_B(q_3, s) \wedge s \wedge i \wedge (g \wedge \neg p \wedge \neg m) \rightarrow \lambda_B(q_3, \tau_{12'}(s, (i, s', 1))) \wedge \lambda_\#(q_3, \tau_{12'}(s, (i, s', 1))) > \lambda_\#(q_3, s) \wedge$
$\lambda_B(q_3, \tau_{12'}(s, (i, s', 2))) \wedge \lambda_\#(q_3, \tau_{12'}(s, (i, s', 2))) > \lambda_\#(q_3, s)$

10 $\forall s \in S. \forall s' \in S'. \forall i \in I_g.\ \lambda_B(q_3, s) \wedge s \wedge i \wedge (g \wedge p \wedge \neg m) \rightarrow \lambda_B(q_4, \tau_{12'}(s, (i, s', 1))) \wedge \lambda_\#(q_4, \tau_{12'}(s, (i, s', 1))) \geqslant \lambda_\#(q_3, s) \wedge$
$\lambda_B(q_4, \tau_{12'}(s, (i, s', 2))) \wedge \lambda_\#(q_4, \tau_{12'}(s, (i, s', 2))) \geqslant \lambda_\#(q_3, s)$

11 $\forall s \in S. \forall s' \in S'. \forall i \in I_g.\ \lambda_B(q_4, s) \wedge s \wedge i \wedge (g \wedge p \wedge \neg m) \rightarrow \lambda_B(q_4, \tau_{12'}(s, (i, s', 1))) \wedge \lambda_\#(q_4, \tau_{12'}(s, (i, s', 1))) \geqslant \lambda_\#(q_4, s) \wedge$
$\lambda_B(q_4, \tau_{12'}(s, (i, s', 2))) \wedge \lambda_\#(q_4, \tau_{12'}(s, (i, s', 2))) \geqslant \lambda_\#(q_4, s)$

12 $\forall s \in S. \forall s' \in S'. \forall i \in I_g.\ \lambda_B(q_4, s) \wedge s \wedge i \wedge (g \wedge \neg p \wedge \neg m) \rightarrow \lambda_B(q_3, \tau_{12'}(s, (i, s', 1))) \wedge \lambda_\#(q_3, \tau_{12'}(s, (i, s', 1))) > \lambda_\#(q_4, s) \wedge$
$\lambda_B(q_3, \tau_{12'}(s, (i, s', 2))) \wedge \lambda_\#(q_3, \tau_{12'}(s, (i, s', 2))) > \lambda_\#(q_4, s)$

3.3. Program construction

If the SMT solver is able find a model for the transition relations that fulfil the specification of the processes, this model can be used to construct the program. The transition function $\tau_{1'2'}$ can be used to construct the whole system at once. The more relevant transition functions are τ_1 and τ_2, as these functions define the transitions for the processes. Therefore, these transition functions can be translated into an arbitrary language.

Listing 3.4 shows a model and Figure 3.5 shows a graphical model for the transition function τ_1 of Example 2 in SMT-LIB format 2.0 [4]. Here the define-function routine defines the function named tau_p1 which takes the current state of the state space of process 1, the current valuation of the global input variables and the current valuation of the private input variables as parameters. As the specification defines no private input, the value for x!3 is empty in every case. The function mkStateP1 constructs a state with the valuations of the global variable g an the private variable p.

Intuitively the SMT-solver constructs a model where the private variable m is set to false in every case. Furthermore, it sets the global variable g to false if g is false in the current state and the system did not receive a request. As our initial specification was $\mathbf{G}((r \rightarrow \mathbf{F}(\mathbf{G}(g))) \wedge \mathbf{F}(\mathbf{G}(\neg m)))$ this model fulfils this specification. Another possible model would be, if g is set, independent from the request, to true and m is always set to false.

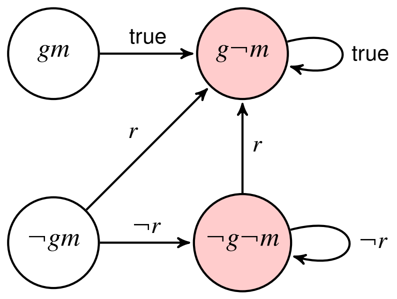

Figure 3.5.: Graphical model for the transition function τ_1. Only the red nodes are reachable by the system.

Listing 3.4: Model for the transition function τ_1

```
(define-fun tau_p1 ((x!1 StateP1) (x!2 GlobalInput) (x!3 P1Input)) StateP1
   (ite (and (= x!1 (mkStateP1 (g true) (m true) ))
             (= x!2 (r true))
             (= x!3 ()))
     (mkStateP1 (g true) (m false) )
   (ite (and (= x!1 (mkStateP1 (g true) (m true) ))
             (= x!2 (r false))
             (= x!3 ()))
     (mkStateP1 (g true) (m false) )
   (ite (and (= x!1 (mkStateP1 (g false) (m true) ))
             (= x!2 (r true))
             (= x!3 ()))
     (mkStateP1 (g true) (m true) )
   (ite (and (= x!1 (mkStateP1 (g false) (m true) ))
             (= x!2 (r false))
             (= x!3 ()))
     (mkStateP1 (g false) (m false) )
   (ite (and (= x!1 (mkStateP1 (g true) (m false) ))
             (= x!2 (r true))
             (= x!3 ()))
     (mkStateP1 (g true) (m false) )
   (ite (and (= x!1 (mkStateP1 (g true) (m false) ))
             (= x!2 (r false))
             (= x!3 ()))
     (mkStateP1 (g true) (m false) )
   (ite (and (= x!1
                (mkStateP1 (g false) (m false) ))
             (= x!2 (r true))
             (= x!3 ()))
     (mkStateP1 (g true) (m false) )
   (ite (and (= x!1
                (mkStateP1 (g false) (m false) ))
             (= x!2 (r false))
             (= x!3 ()))
     (mkStateP1 (g false) (m false) )
     (mkStateP1 (g true) (m false) )))))))))
)
```

Listing 3.5 shows a model and Figure 3.6 shows a graphical model for the transition function of the second process. This process should fulfil the specification $\mathbf{G}(r \rightarrow (\mathbf{F}(\mathbf{G}(g \wedge p))))$.

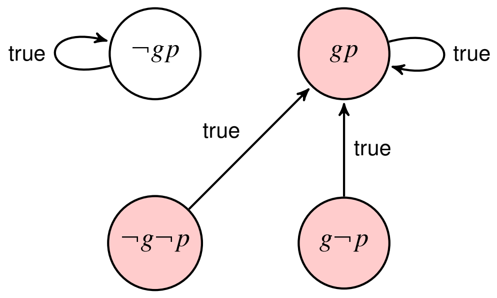

Figure 3.6.: Graphical model for the transition function τ_2. Only the red nodes are reachable by the system.

While the model for the first process is straightforward, this model is more interesting. Intuitively, the model sets the global variable g and the local variable p always to true except for two cases: If the global variable is false, the local variable is true and the system

- receives or
- does not receive

a request. If either the variables are set to true, or if the system does not receives a request, the specification is fulfilled directly by the model. The case that sets the private variable to true and the global variable to false if the system receives a request and the current global variable is set to false and the current local variable is set to true, does not fulfil the specification by its own. However, if we consider the specified start valuation, the specification of the other process and the remaining model of this process, this state can never be reached.

Listing 3.5: Model for the transition function τ_2

```
(define-fun tau_p2 ((x!1 StateP2) (x!2 GlobalInput) (x!3 P2Input)) StateP2
    (ite (and (= x!1 (mkStateP2 (g false) (p true) ))
              (= x!2 (r true))
              (= x!3 ()))
      (mkStateP2 (g false) (p true) )
    (ite (and (= x!1 (mkStateP2 (g false) (p true) ))
              (= x!2 (r false))
              (= x!3 ()))
      (mkStateP2 (g false) (p true) )
    (ite (and (= x!1 (mkStateP2 (g true) (p true) ))
              (= x!2 (r true))
              (= x!3 ()))
      (mkStateP2 (g true) (p true) )
    (ite (and (= x!1 (mkStateP2 (g true) (p true) ))
              (= x!2 (r false))
              (= x!3 ()))
      (mkStateP2 (g true) (p true) )
    (ite (and (= x!1 (mkStateP2 (g true) (p false) ))
              (= x!2 (r false))
              (= x!3 ()))
      (mkStateP2 (g true) (p true) )
    (ite (and (= x!1 (mkStateP2 (g true) (p false) ))
              (= x!2 (r true))
              (= x!3 ()))
      (mkStateP2 (g true) (p true) )
    (ite (and (= x!1
                    (mkStateP2 (g false) (p false) ))
                 (= x!2 (r true))
                 (= x!3 ()))
      (mkStateP2 (g true) (p true) )
    (ite (and (= x!1
                    (mkStateP2 (g false) (p false) ))
                 (= x!2 (r false))
                 (= x!3 ()))
      (mkStateP2 (g true) (p true) )
      (mkStateP2 (g true) (p true) ))))))))))
```

The Listings 3.6 and 3.7 show the deduced pseudo-code of the models above. The transition functions are used to construct the program logic of the processes and the mkState functions are used to define the assignments.

Listing 3.6: Model for the transition function τ_1

```
do{
  if(g && m && r)  {
    g = true;
    m = false;
  } else if(g && m && !r) {
    g = true;
    m = false;
  } else if(!g && m && r) {
    g = true;
    m = false;
  } else if(!g && m && !r) {
    g = false;
    m = false;
  } else if(g && !m && r) {
    g = true;
    m = false;
  } else if(g && !m && !r) {
    g = true;
    m = false;
  } else if(!g && !m && r) {
    g = true;
    m = false;
  } else if(!g && !m && !r) {
    g = false;
    m = false;
  } else {
    g = true;
    m = false;
  }
} while(true)
```

Listing 3.7: Model for the transition function τ_2

```
do{
  if(!g && p && r)  {
    g = false;
    p = true;
  }else if(!g && p && !r)  {
    g = false;
    p = true;
  }else if(g && p && r)  {
    g = true;
    p = true;
  }else if(g && p && !r)  {
    g = true;
    p = true;
  }else if(g && !p && !r)  {
    g = true;
    p = true;
  }else if(g && !p && r)  {
    g = true;
    p = true;
  }else if(!g && !p && r)  {
    g = true;
    p = true;
  }else if(!g && !p && !r)  {
    g = true;
    p = true;
  }else {
    g = true;
    p = true;
  }
} while(true)
```

Chapter 4

Implementation

This section describes our implemented prototype to solve assume-guarantee synthesis problems for LTL specifications with partial information.

4.1. Overview

The prototype is a commandline tool that synthesizes a pseudocode based on the input file defined by the system designer. The architecture of the prototype is modular and therefore the prototype can be easily extended with other synthesis approaches, input formats or output formats. A detailed description of the design can be found in chapter 4.3.

At the moment, the prototype offers implementations for assume-guarantee synthesis and cooperative co-synthesis. The user indicates the desired algorithm within the input file. Moreover, this input file is also used to specify the specifications of the processes and system parameters. The format of the input file is specified in detail in chapter 4.2.

4.1.1. Program flow

The program starts with analysing and processing the input file. The specifications of the processes are extracted and the contained information is used to built the co-Büchi automata for the assume-guarantee or cooperative co-synthesis approach. To construct these automata, the LTL3BA [3] [21] tool is used. LTL3BA uses the LTL specification as an input and returns the states and transitions of the built automaton as an output. These states and transitions are afterwards translated into constraint systems. The constraint system is written in an extended version of the SMT-LIB v2 [4], which can be solved by the Z3 SMT-ü solver. If Z3 is able to find a satisfying model for the constraint system, this model is used to construct a pseudocode of the processes.

Additional to the pseudocode, the prototype returns statistical information like the runtime of the prototype, the runtime of Z3, the number of decisions, the number of conflicts and the used memory. These values can be used to compare the synthesis approaches.

4.2. Input

The input file consists of the following two parts:

- The program configuration and the
- the system specification.

Program configuration. The program configuration is used to specify the path to the LTL3BA tool, Z3-solver and the paths and names for the output files. The LTL3BA tool is used to convert the AGS-conditions into Büchi automata. The resulting automata are translated by the prototype to a constraint system that is solved by the Z3 theory solver . The output of Z3 is then parsed and based on its outpu,t example-processes in pseudocode are produced.

The following mandatory tags are used by the program to specify the values for the program configuration.

- ”-ltl3ba”: path to the LTL3BA application,
- ”-z3”: path to the Z3 theorem solver,
- ”-syn”: synthesis approach to be executed (either ”BoundedSyntesis” or ”AssumeGuaranteeSynthesis”),
- ”-cons”: path of the produced constraint system,
- ”-ba”: path of the produced Büchi automata,
- ”-p1”: output-file for the pseudocode of the first process and
- ”-p2”: output-file for the pseudocode of the second process

System specification. The system specification defines the processes and the inputs of the system to be synthesized. By specifying the global, local and memory variables, the maximal possible states of the state space are fixed. To increase the number of possible states it is possible to add new memory variables.

The following parameters are used to describe the processes in the configuration file.

- ”-spec1”: LTL specification of the first process,
- ”-spec2”: LTL specification of the second process,
- ”-glob”: global system-variables with their start values,
- ”-l1”: local variables of the first process and their start values,
- ”-l2”: local variables of the second process and their start values,
- ”-mg”: number of global memory variables,
- ”-m1”: number of memory variables used by the first process,
- ”-m2”: number of memory variables used by the second process,
- ”-ig”: global input variables of the system,

- ”-i1”: local input variables of the first process and
- ”-i2”: local input variables of the second process.

The configuration file of example 2 would be:

```
-ltl3ba "path//ltl3ba.exe"
-z3 "path//z3.exe"
-syn BoundedSynthesis
-cons "path//cons.smt2"
-ba "path//automaton.dot"
-p1 "path//process1"
-p2 "path//process2"
-spec1 (r->F(G(g)))&&F(!m)
-spec2 r->(F(G(g))&&(F(G(p))))
-glob g:false
-l1 m:false
-l2 p:false
-ig r
```

4.3. Software Design

The prototype is written in C#. The design is modular and therefore allows the programmer to easily add new features and algorithms. Figure 4.1 shows a class diagram describing the structure of the program. In order to keep the diagram simple, only the most relevant classes and public methods are included. Moreover, class-attributes and subordinate helper classes are not displayed.

Config package The purpose of the config package is to parse the input file and to prepare the data. These tasks are executed by the two classes `Parser` and `Config`. The parser reads the input file and sets the appropriate attributes in the `Config` class. The `Config` file is implemented as singleton to be accessible throughout all classes. It is used to provide information like program paths or specification details for the other classes.

Automata package The automata package handles the co-Büchi automata. The class `UCTAutomaton` prepares, in particular negates, the provided specification formula and calls the LTL3BA application. Afterwards, the class parses the resulting automaton. The automaton is represented as a list containing the transitions as `UCTTransistion`. Furthermore, the `UCTAutomaton` class also provides methods to write the automaton and to create a DOT representation.

Synthesis The synthesis package comprises the program logic. In the prototype, this package contains the algorithms for assume-guarantee and cooperative co-synthesis. The synthesis algorithms implement the `Synthesis` interface. Consequently, it is possible to add new synthesis algorithms easily. The methods

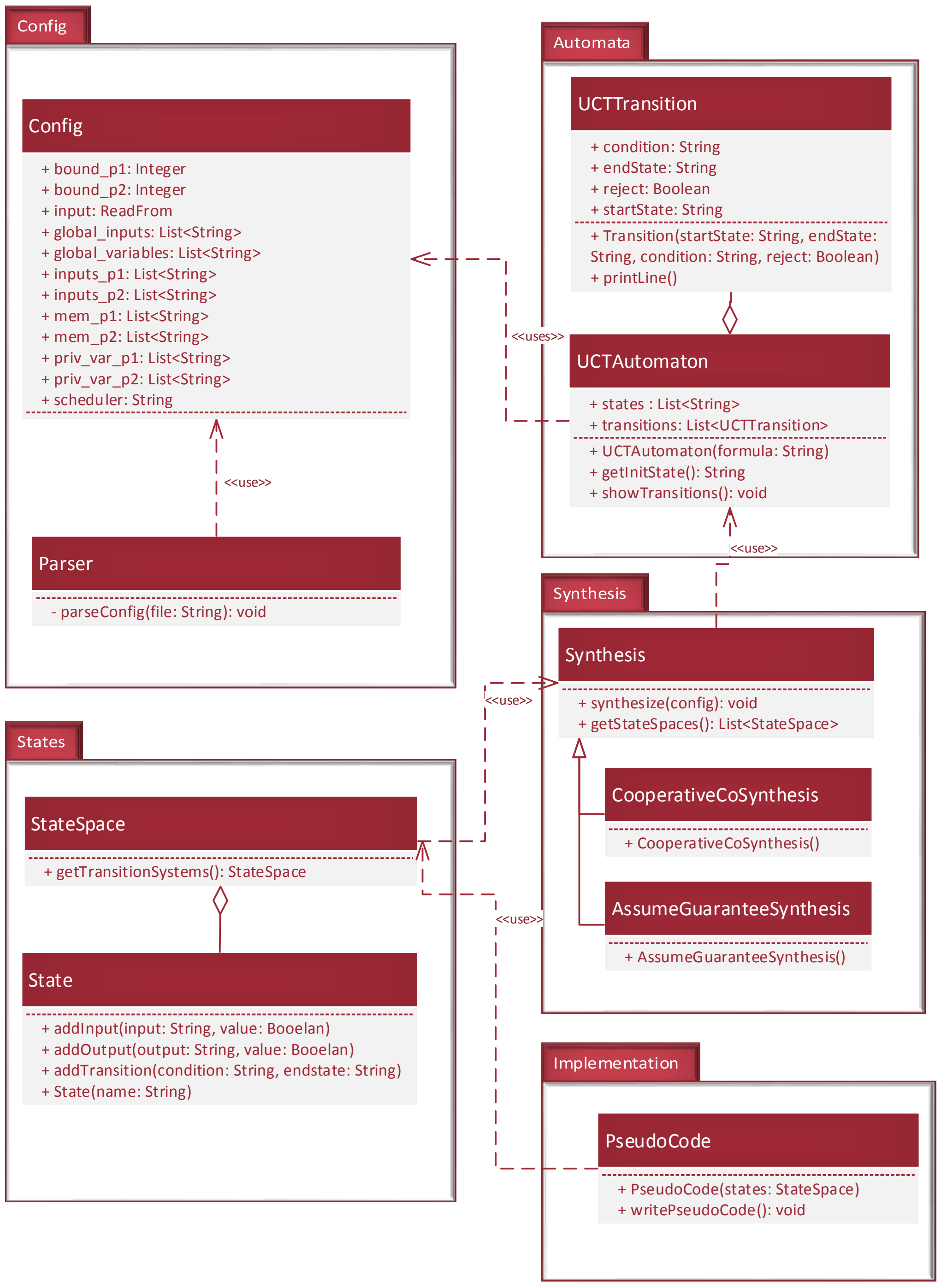
Config
Config
+ bound_p1: Integer
+ bound_p2: Integer
+ input: ReadFrom
+ global_inputs: List<String>
+ global_variables: List<String>
+ inputs_p1: List<String>
+ inputs_p2: List<String>
+ mem_p1: List<String>
+ mem_p2: List<String>
+ priv_var_p1: List<String>
+ priv_var_p2: List<String>
+ scheduler: String
<<use>>
Parser
- parseConfig(file: String): void
Automata
UCTTransition
+ condition: String
+ endState: String
+ reject: Boolean
+ startState: String
+ Transition(startState: String, endState: String, condition: String, reject: Boolean)
+ printLine()
<<uses>>
UCTAutomaton
+ states : List<String>
+ transitions: List<UCTTransition>
+ UCTAutomaton(formula: String)
+ getInitState(): String
+ showTransitions(): void
<<use>>
Synthesis
Synthesis
+ synthesize(config): void
+ getStateSpaces(): List<StateSpace>
<<use>>
CooperativeCoSynthesis
+ CooperativeCoSynthesis()
AssumeGuaranteeSynthesis
+ AssumeGuaranteeSynthesis()
States
StateSpace
+ getTransitionSystems(): StateSpace
State
+ addInput(input: String, value: Booelan)
+ addOutput(output: String, value: Booelan)
+ addTransition(condition: String, endstate: String)
+ State(name: String)
<<use>>
Implementation
PseudoCode
+ PseudoCode(states: StateSpace)
+ writePseudoCode(): void

Figure 4.1.: Class diagram of the prototype

to implement are `synthesize` and `getTransitionSystems`. The `synthesize` method uses the Automata package to build the co-Büchi automata, translates them into the constraint system, calls the Z3 tool to solve the constraint system and parses the result. The method `getTransitionSystems` returns this result.

Currently, the `Synthesis` interface is implemented by the classes `AssumeGuaranteeSynthesis` and `CooperativeCoSynthesis`.

The `AssumeGuaranteeSynthesis` class provides an implementation for the AGS algorithm. Therefore, it uses the specification of the two processes and converts them into the three AGS conditions. These conditions are afterwards transformed into universal co-Büchi automata. For this transformation the class `UCTAutomaton` is used. This class returns the transition relation of the automata. These conditions are then converted, as described in Section 3.2, into SMT constraints, using the input format of the Z3 theorem solver. The input format is an extended version of the SMT-LIB 2.0 [4] standard. The advantage of the approach in contrast of using the Z3 API is that the solver can be exchanged easily.

As the states are defined by the valuation of the global, local and memory variables we are using the datatype objects of Z3 to represent this. Another approach would be to define the variables within the transition function, as it was proposed in [31].

If Z3 is able to solve the constraint system, the class parses the model returned by the solver. In particular, it extracts the model for the transition functions τ_1 and τ_2 and uses the `StateSpace` class to construct models of the processes. These models can be received afterwards by the `getTransitionSystems` method, which returns the state spaces of the processes.

The class `CooperativeCoSynthesis` does the same for the cooperative co-synthesis approach. In particular, only the first AGS condition is used to construct the constraint system. The prototype uses the cooperative co-synthesis approach to compare the performance of AGS to cooperative co-synthesis.

States The `States` package is used to represent the state spaces of the processes. For this representation the class `StateSpace` is used, which contains the states and the transitions of the processes. To do so, it uses the a list of `State`-objects to construct the transition system. This state space can be later used to implement the processes in various programming languages.

Implementation The `Implementation` package uses the States package to implement the processes. Currently, the only class available is the `PseudoCode` class, which transfers the state spaces of the processes into a pseudocode. Other classes that produce output in languages such as C or Verilog can easily be added.

Chapter 5

Experiments

This chapter shows the assum- guarantee synthesis approach for some examples. The first part describes the used examples. The second part analyses the performance of the assume guarantee synthesis and cooperative co-synthesis approach and compares them. The performance is measured by the runtime of the prototype, the time consumed by the Z3 solver, the number of occurred conflicts, the number of taken decisions and the needed memory.

In the following experiments we are going to show the impact of the additional constraints needed for assume-guarantee synthesis compared to cooperative co-synthesis. Additionally, we show that the processes synthesized with cooperative co-synthesis are in contrast to assume guarantee synthesis, not robust.

5.1. Examples

In this section, we introduce the examples. The examples are grouped into three types: arbiter examples, memory examples and reader-writer problems.

5.1.1. Arbiter Examples

The first type of examples are arbiter examples. An arbiter is used to control the access to shared resources. For example, a memory arbiter is used to decide which CPU is allowed to access the shared memory.

In our examples, the system receives requests to access critical resources and the processes provide grants, depending on the specification.

5.1.1.1. Global Grant

Our first architecture describes a simple 2-process arbiter. The processes shall grant access infinitely often, by setting the global variables g_1 and g_2 to true.

```
g = false;

//process P1
do{
  if(g1 && g2)  {
    g1 = true;
    g2 = false;
  } else if(!g1 && g2) {
    g1 = true;
    g2 = false;
  } else if(g1 && !g2) {
    g1 = false;
    g2 = true;
  } else if(!g1 && !g2) {
    g1 = true;
    g2 = false;
  } else {
    g1 = true;
    g2 = false;
  }
} while(true);
```

```
//process P2
do{
  if(!g1 && g2)  {
    g1 = false;
    g2 = true;
  } else if(g1 && !g2) {
    g1 = false;
    g2 = true;
  } else if(g1 && g2) {
    g1 = true;
    g2 = false;
  } else if(!g1 && !g2) {
    g1 = false;
    g2 = true;
  } else {
    g1 = false;
    g2 = true;
  }
} while(true);
```

Code Listing 5.1: Solution with Cooperative Co-Synthesis

Specification. The LTL-specification for the first process is as follows: $\mathbf{G}(\mathbf{F}g_1)$ and the specification of the second process analogously: $\mathbf{G}(\mathbf{F}g_2)$ Both variables g_1 and g_2 are defined globally and their start valuation is false.

Result. As shown in Code Listing 5.1 the assume-guarantee synthesis approach delivered two robust processes. Whereas the second process grants access every time it is scheduled, the first process only grants access if it was not granted in the last step or g_2 was true.

Comparison with Cooperative Co-Synthesis. Code Listing 5.2 shows the pseudocode produced by our prototype for the cooperative co-synthesis approach. As it can be seen, the second process synthesized with cooperative co-synthesis is not robust. If the processes would be scheduled alternately and the global variables are set to false, the second process is setting g_1 to true and g_2 to false. If the first process is replaced with another process that fulfils its specification and sets both variables to false if g_1 is true and g_2 is false, the second process would not be able to fulfil its specification, as g_2 would be never set to true.

5.1.1.2. Global Request and Grant

Our second architecture describes a simple 2-process arbiter. If the system receives a request (r), a process grants the access by eventually setting the global variable g to true.

Specification. Both processes are specified by $\mathbf{G}(r \rightarrow \mathbf{F}g)$. This specification ensures that the access is granted, independent of the scheduling. The request r and the grant g are globally defined and are therefore accessible by both processes. The start valuation of the global variable g is false.

```
g = false;

//process P1
do{
  if(g1 && g2)  {
    g1 = true;
    g2 = false;
  } else if(!g1 && g2) {
    g1 = true;
    g2 = false;
  } else if(g1 && !g2) {
    g1 = true;
    g2 = false;
  } else if(!g1 && !g2) {
    g1 = true;
    g2 = false;
  } else {
    g1 = true;
    g2 = false;
  }
} while(true);
```

```
//process P2
do{
  if(!g1 && g2)  {
    g1 = true;
    g2 = false;
  } else if(g1 && !g2) {
    g1 = false;
    g2 = true;
  } else if(g1 && g2) {
    g1 = true;
    g2 = false;
  } else if(!g1 && !g2) {
    g1 = true;
    g2 = false;
  } else {
    g1 = true;
    g2 = false;
  }
} while(true);
```

Code Listing 5.2: Solution with Cooperative Co-Synthesis

```
g = false;

//process P1
do{
  if(g && r)  {
    g = false;
  } else if(g && !r) {
    g = false;
  } else if(!g && r) {
    g = true;
  } else if(!g && !r) {
    g = true;
  } else {
    g = false;
  }
} while(true);
```

```
//process P2
do{
  if(!g && r)  {
    g = true;
  } else if(!g && !r) {
    g = false;
  } else if(g && r) {
    g = false;
  } else if(!g && r) {
    g = false;
  } else {
    g = false;
  }
} while(true);
```

Code Listing 5.3: Solution with AGS

Result. Code Listing 5.3 shows the solution produced by our prototype. Even though the processes are specified in the same way, the SMT solver produces different process implementations. The first process sets g to true if g was false in the previous execution step and vice versa. The second process sets g only to true if g was false in the previous execution step and the system receives a request.

As both processes have the same specification, the AGS approach does not add any additional constraints compared to the bounded synthesis approach for cooperative co-synthesis. This is caused by the implications in the second ($[[P'_1||P_2||C]] \upharpoonright_{(S\cup I)} \models \phi_2 \rightarrow \phi_1$) and third ($[[P_1||P'_2||C]] \upharpoonright_{(S\cup I)} \models \phi_1 \rightarrow \phi_2$) AGS-condition. As ϕ_1 is equal to ϕ_2, the implications and therefore the conditions are always fulfilled.

5.1.1.3. Global Request and Grant in the following Execution Step

This example describes a stricter arbiter. As in the previous example, a 2-process arbiter is specified. Here access has to be granted in the next step.

```
g = false;

//process P1
do{
  if(g && r)  {
    g = true;
  } else if(g && !r) {
    g = false;
  } else if(!g && r) {
    g = true;
  } else if(!g && !r) {
    g = false;
  } else {
    g = true;
  }
} while(true);
```

```
//process P2
do{
  if(g && r)  {
    g = true;
  } else if(g && !r) {
    g = false;
  } else if(!g && r) {
    g = true;
  } else if(!g && !r) {
    g = false;
  } else {
    g = true;
  }
} while(true);
```

Code Listing 5.4: Solution with AGS

Specification. The LTL specification for both processes is: $\mathbf{G}(r \rightarrow \mathbf{X}g)$. It ensures that the access is granted in the next execution step. As above, the input variable r and the process variable g are defined public. Furthermore, the start valuation of g is false.

Result. As this specification is stricter than the previous one, the implementation of the previous example does not fulfil it, whereas this implementation fulfils the requirement for the previous example. Code Listing 5.4 shows the process-implementation produced by our prototype. In this case, our prototype produced the same implementation for both processes, where the processes set g to true if the system receives a request and to false otherwise. Another possible solution would be to set g always to true.

5.1.1.4. Global request and Grant only in the following Execution Step

In this example, the arbiter is even more restricted, in the sense that a grant is only allowed if the system receives a request in the next execution step.

Specification. As above, the processes have the same specification. The LTL-specification for both processes is: $\mathbf{G}((r \rightarrow \mathbf{X}g) \wedge (\neg r \rightarrow \mathbf{X}\neg g))$. The first part of the specification ensures that if the system receives a request, a grant is given at the next execution step. On the other hand, the second part of the specification states that the system must not give a grant if it does not receive a request.

Result. Even though this specification is stricter than the previous one, the previous implementation of the processes, as shown in Code Listing 5.4, also fulfils this specification. Our prototype already coincidentally provided an solution that is also valid even though this specification adds more restrictions. However, the other mentioned solution of the previous example, in which g is true in every case, does not fulfil this restricted specification.

```

//process P1
g1 = false;

do{
  if(g1 && r)  {
    g1 = true;
  } else if(g1 && !r) {
    g1 = true;
  } else if(!g1 && r) {
    g1 = true;
  } else if(!g1 && !r) {
    g1 = false;
  } else {
    g1 = true;
  }
} while(true);
```

```
//process P2
g2 = false;
do{
  if(g2 && r)  {
    g2 = true;
  } else if(g2 && !r) {
    g2 = true;
  } else if(!g2 && r) {
    g2 = true;
  } else if(!g2 && !r) {
    g2 = true;
  } else {
    g2 = true;
  }
} while(true);
```

Code Listing 5.5: Solution with AGS

5.1.1.5. Global Request and Private Grant

In this architecture, every process has its own private variable (g_1 and g_2) to grant the access if the system receives a global request r.

Specification. The LTL-specification for the first process is as follows: $\mathbf{G}(r \rightarrow \mathbf{F}\, g_1)$. The first process has to grant access by setting g_1 eventually to true if the system receives a request r. The second process is specified analogously: $\mathbf{G}(r \rightarrow \mathbf{F}\, g_2)$. Like the first process, if the system receives a request, the second process has to set the private variable g_2 eventually to true.

Result. As only one process is scheduled in each execution step, the processes cannot be sure if the system has received a request or not. Therefore the private variables of the processes have to be set infinitely often to true. Code Listing 5.5 shows the solution of our prototype. Here both variables are always set to true. Another possible solution is to flip the private variables whenever the process is scheduled.

If we add the restriction that a grant is only allowed if the system received a request before, the system would not be realisable with AGS without any adaptations. The LTL-specifications with this modification would be $((r\,\mathbf{R} \neg g_1) \wedge \mathbf{G}(r \rightarrow \mathbf{F}\, g_1))$ for the first process and $((r\,\mathbf{R} \neg g_2) \wedge \mathbf{G}(r \rightarrow \mathbf{F}\, g_2))$ for the second process. The second part of the specification is as above. The first part guarantees that the system does not grant access as long as it does not receive a request.

However, such a system would be realisable with cooperative co-synthesis. For example, a global memory variable could be introduced to mark if the system received a request or not. As a memory variable can be set freely, without any restrictions by the specification, it is still unrealisable for AGS.

5.1.1.6. Implicit Memory Variable

As mentioned in the last example, a global memory variable is rarely useful in AGS as its behaviour is not specified and therefore the processes cannot rely on the behaviour of the other processes. However, it is possible to explicitly define the behaviour of this ”memory” variable.

In this example, we explicitly define a global variable to mark if the system has received a request or not.

Specification. Let g_1 and g_2 be the private variable of process 1 and 2, r the global input variable that signals if the system receives a request or not, and the global variable a that indicates if the system has received a request or not. The LTL-specification for the processes are defined as follows:

- First process: $(r\mathbf{R}\neg a) \wedge \mathbf{G}(r \rightarrow \mathbf{X}(\mathbf{G}a)) \wedge \mathbf{G}(a \rightarrow \mathbf{F}g_1) \wedge (a\mathbf{R}\neg g_1)$
- Second process: $(r\mathbf{R}\neg a) \wedge \mathbf{G}(r \rightarrow \mathbf{X}(\mathbf{G}a)) \wedge \mathbf{G}(a \rightarrow \mathbf{F}g_2) \wedge (a\mathbf{R}\neg g_2)$

The subformula $(r\mathbf{R}\neg a)$ ensures that a is false as long as the system does not receive a request. The next subformula $\mathbf{G}(r \rightarrow \mathbf{X}(\mathbf{G}a))$ states that the global variable a has to be set always after the system receives a request. Therefore, under the assumption that the scheduler is fair, both processes know if the system has received a request. That both processes grant access only after the system receives a request is ensured by the subformuals $\mathbf{G}(a \rightarrow \mathbf{F}g_1) \wedge (a\mathbf{R}\neg g_1)$ and $\mathbf{G}(a \rightarrow \mathbf{F}g_2) \wedge (a\mathbf{R}\neg g_2)$.

Result. The synthesized pseudocode for this specification is shown in Code Listing 5.6. The first process sets a to true if a was true in the last execution step or the system receives a request r. The private variable g_1 is set to true the first time the process is scheduled after the system receives a request. Afterwards, g_1 is flipped every time the first process is scheduled. The second process also sets the global variable a to true after the system receives a request for the first time and if a was true in the last execution step. Like the first process, the second process sets its private variable g_2 to true the first time the process is scheduled after a was set to true, and flips it afterwards every time the process is scheduled.

5.1.1.7. Global Request, Private Grant in the Next Execution Step and Referring to the Scheduler

The last examples contained workarounds to handle the scheduling of the process. Another way to treat the scheduler is to explicitly specify that the invariants should only hold if the according process was scheduled.

In this example, access should be granted if the process is scheduled and the system receives a request. If the system does not receive a request and a process is scheduled, this process should not grant access.

Specification. Let r be the global output variable, g_1 and g_2 the private input variables and c the scheduler variable. The scheduler variable is treated as an input and is true if the first process is scheduled and false if the second process is scheduled. The processes are specified as follows:

- First process: $\mathbf{G}(((c \wedge r) \rightarrow \mathbf{X}g_1) \wedge ((c \wedge \neg r) \rightarrow \mathbf{X}\neg g_1))$
- Second process: $\mathbf{G}(((\neg c \wedge r) \rightarrow \mathbf{X}g_2) \wedge ((\neg c \wedge \neg r) \rightarrow \mathbf{X}\neg g_2))$

Result. Code Listing 5.7 shows the pseudocode produced by our prototype. As expected, g_1 and g_2 are only set to true if the system receives a request and the process is scheduled.

```
a = false;
//process P1
g1 = false;

do{
  if(r && a && g1)  {
    a = true;
    g1 = false;
  } else if(!r && a && g1) {
    a = true;
    g1 = false;
  } else if(r && !a && !g1) {
    a = true
    g1 = false;
  } else if(!r && !a && !g1) {
    a = false;
    g1 = false;
  } else if(r && a && !g1) {
    a = true;
    g1 = true;
  } else if(!r && a && !g1) {
    a = true;
    g1 = true;
  } else if(r && !a && g1) {
    a = true;
    g1 = true;
  } else if(!r && !a && g1) {
    a = false;
    g1 = false;
  }  else {
    a = true;
    g1 = false;
  }
} while(true);
```

```
//process P2
g2 = false;
do{
  if(r && a && g2)  {
    a = true;
    g2 = false;
  } else if(!r && a && g2) {
    a = true;
    g2 = false;
  } else if(r && a && !g2) {
    a = true;
    g2 = true;
  } else if(!r && a && !g2) {
    a = true;
    g2 = true;
  } else if(!r && !a && g2) {
    a = false;
    g2 = false;
  } else if(!r && !a && !g2) {
    a = false;
    g2 = false;
  } else if(r && !a && g2) {
    a = false;
    g2 = false;
  } else if(r && !a && !g2) {
    a = true;
    g2 = false;
  } else{
    a = false;
    g2 = false;
  }
} while(true);
```

Code Listing 5.6: Solution with AGS

```

//process P1
g1 = false;

do{
  if(g1 && r)  {
    g1 = true;
  } else if(g1 && !r) {
    g1 = false;
  } else if(!g1 && r) {
    g1 = true;
  } else if(!g1 && !r) {
    g1 = false;
  } else {
    g1 = true;
  }
} while(true);
```

```
//process P2
g2 = false;
do{
  if(g2 && r)  {
    g2 = true;
  } else if(g2 && !r) {
    g2 = false;
  } else if(!g2 && r) {
    g2 = true;
  } else if(!g2 && !r) {
    g2 = false;
  } else {
    g2 = true;
  }
} while(true);
```

Code Listing 5.7: Solution with AGS

```

//process P1
g1 = false;

do{
  if(g1 && r1)  {
    g1 = true;
  } else if(g1 && !r1) {
    g1 = false;
  } else if(!g1 && r1) {
    g1 = true;
  } else if(!g1 && !r1) {
    g1 = false;
  } else {
    g1 = true;
  }
} while(true);
```

```
//process P2
g2 = false;
do{
  if(g2 && r2)  {
    g2 = true;
  } else if(g2 && !r2) {
    g2 = false;
  } else if(!g2 && r2) {
    g2 = true;
  } else if(!g2 && !r2) {
    g2 = false;
  } else {
    g2 = true;
  }
} while(true);
```

Code Listing 5.8: Solution with AGS

5.1.1.8. Private Request, private Grant in the next Execution Step and referring to Scheduler

All previous examples used a global input variable. This example shows the usage of private input variables. Here, the process should only grant access if the system receives a private request for the process currently scheduled.

Specification. Let r_1 and r_2 be the private input variables and g_1 and g_2 the private variables of the first and second process. The LTL specification is as follows:

- First process: $\mathbf{G}(((c \wedge r_1) \rightarrow \mathbf{X} g_1) \wedge ((c \wedge \neg r_1) \rightarrow \mathbf{X} \neg g_1))$
- Second process: $\mathbf{G}(((\neg c \wedge r_2) \rightarrow \mathbf{X} g_2) \wedge ((\neg c \wedge \neg r_2) \rightarrow \mathbf{X} \neg g_2))$

Result. According to the specification the first process sets its private variable g_1 to true if it receives a request r_1 and sets g_1 to false if it does not receive a request r_1. Similarly, the second process grants access by setting g_2 to true if the process receives a private request.

5.1.1.9. Private Grants without Requests

This example describes an arbiter in which the processes should infinitely often grant access for the first and second resource without any request. Additionally, the access to both resources must not be granted simultaneously. A simple LTL-specification for the two processes would be $\mathbf{G}((\neg g_1 \vee \neg g_2) \wedge (\mathbf{F} g_i))$ for $i \in \{1,2\}$, where g_1 and g_2 are global variables. This specification is realisable, for example if the first process always sets the global variable g_1 to true and g_2 to false. Vice versa, the second process sets the global variable g_2 always to true and g_1 always to false. A disadvantage of this solution is that both processes have access to both variables. A solution could be to set both variables to private and to add a global variable as flag that indicates if g_1 is set or not. Such a system is realisable with cooperative co-synthesis but not with AGS as a valid implementation for the first process is to always set g_1 to true. Therefore the second process does not have the possibility to fulfil its specification by eventually setting g_2 to true. Thus, for AGS to succeed, a condition saying that g_1 and g_2 are not always true needs to be added.

```
a = false;
//process P1
g1 = false;

do{
  if(!a && !g1)  {
    a = false;
    g1 = false;
  } else if(a && g1) {
    a = false;
    g1 = false;
  } else if(!a && g1) {
    a = false;
    g1 = false;
  } else if(a && !g1) {
    a = true;
    g1 = true;
  } else {
     a = false;
     g1 = true;
  }
} while(true);
```

```
//process P2
g2 = false;
do{
  if(a && g2)  {
    a = true;
    g2 = false;
  } else if(!a && g2) {
    a = true;
    g2 = false;
  } else if(!a && !g2) {
    a = false;
    g2 = true;
  } else if(a && !g2) {
    a = true;
    g2 = false;
  } else {
    a = true;
    g2 = false;
  }
} while(true);
```

Code Listing 5.9: Solution with AGS

Specification. Let g_1 and g_2 be private variables and a a global variable. The LTL-specification for the processes is defined as follows:

- First process: $\mathbf{G}((\neg g_1 \vee a) \wedge \mathbf{F} g_1 \wedge \mathbf{F} \neg a \wedge (g_1 \rightarrow a) \wedge (\neg a \rightarrow \neg g_1))$
- Second process: $\mathbf{G}((\neg g_2 \vee \neg a) \wedge \mathbf{F} g_2 \wedge \mathbf{F} a \wedge (g_2 \rightarrow \neg a) \wedge (a \rightarrow \neg g_2))$

The first clause guarantees that the processes do not grant the access simultaneously. That the process grants access to the own client is guaranteed by the second clause. The remaining clauses ensure that the grant is given correctly and the other process has the possibility the grant access to its client.

Result. The pseudocode produced by our prototype is shown in Code Listing 5.9. Both processes grant and deny access as fast as possible.

5.1.2. Memory examples

The second type of examples are memory examples. These examples show the usage of memory variables. In general, memory variables are used to extend the state space of the processes if it is not possible to construct a program with the given specification.

5.1.2.1. Memory defined

In this example, the public variable b should be flipped in every second execution step. Additionally we define the global variable a, which is flipped in every execution step. This variable can be used as a counter to indicate when b has to be flipped.

```
a = false;
b = false;
//process P1

do{
  if(a && b)  {
    a = false;
    b = false;
  } else if(!a && b) {
    a = true;
    b = true;
  } else if(a && !b) {
    a = false;
    b = true;
  } else if(!a && !b) {
    a = true;
    b = false;
  } else {
     a = false;
     b = false;
  }
} while(true);
```

```

do{
  if(a && b)  {
    a = false;
    b = false;
  } else if(a && !b) {
    a = false;
    b = true;
  } else if(!a && b) {
    a = true;
    b = true;
  } else if(!a && !b) {
    a = true;
    b = false;
  } else {
     a = false;
     b = false;
  }
} while(true);
```

Code Listing 5.10: Solution with AGS

Specification. The specification for both processes is $\mathbf{G}((a \leftrightarrow \mathbf{X}\neg a) \wedge (b \leftrightarrow \mathbf{X}(\mathbf{X}\neg b)))$. The first part specifies that a has to be flipped in every execution step, the second part specifies the flipping behaviour of the global variable b.

Result. The implementation produced by our prototype is shown in Code Listing 5.10. Our prototype produced the same implementation for both processes. The public variable a is flipped every time and the public variable b is flipped if a and b have different truth-values in the last execution step. Below you can find a possible execution trace for this implementation.

scheduled process	1	2	2	2	1	2	1	1	2	2	1	1	2	1	2	1	1	1	2	...
a	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	...
b	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	...

Table 5.1.: Possible execution trace

5.1.2.2. Memory not defined

In this example, we do not specify the counter directly. Therefore, we only specify the global variable b. As above, the global variable b should be flipped in every second execution step. Additionally, we add a global memory variable m, which is not mentioned by the specification.

Specification. The LTL specification for both processes is $\mathbf{G}(a \leftrightarrow \mathbf{X}(\mathbf{X}\neg b))$. It specifies that b has to be flipped in every third execution step.

Result. Our prototype produced the same implementation as in the last example but the global variable is changed into a global memory variable.

5.1.3. Readers-Writers problems

The Readers-Writers problems were introduced by Courtois, Heyman and Parnas [10] as practically significant problems related to the mutual exclusion problem. They defined two classes of wishing to use the shared resource. The first class is the readers class and the second one is the writers class. While the processes of the writers class must have an exclusive access to the resource, the processes of the readers class may share the resources with other readers.

5.1.3.1. Exclusive access

This example is a modified version of the Readers-Writers problem. In our example, the processes should write/read infinitely often without asking for it. Furthermore, at every execution step, only one process is allowed to access the shared resource.

Specification. The processes have the following specification:

$\mathbf{G}(\mathbf{F}w_1 \wedge \mathbf{F}r_1 \wedge \mathbf{F}r_2 \wedge \mathbf{F}w_2 \wedge (w_1 \rightarrow (\neg w_2 \wedge \neg r_2 \wedge \neg r_1)) \wedge (w_2 \rightarrow (\neg w_1 \wedge \neg r_2 \wedge \neg r_1)) \wedge (r_2 \rightarrow (\neg w_1 \wedge \neg w_2 \wedge \neg r_1)) \wedge (r_1 \rightarrow (\neg w_1 \wedge \neg w_2 \wedge \neg r_2)))$.

All variables are defined globally. The first four subformulas ensure that every variable is set eventually to true. The rest of the formula specifies that only one process is allowed to access the shared resource.

Result. Too keep it simple, Code Listing 5.11 shows only the necessary conditions for the processes. As shown in Figure 5.1, the first process starts by setting w_1 to true. After w_1 the first process sets w_2 to true, followed by r_1. The last variable in the cycle is r_2. Afterwards, the first process sets all variables to false.

The second process behaves slightly different. Like the first process, the second process sets the variable w_1 to true if all variables were set to false in the previous execution set. If w_1 or r_1 is true, the second process does not change the valuations of the variables. Unlike the first process, the second process does not set all variables to false after r_2 was set to true but sets directly w_1 to true.

As you can see, the second process does not fulfil the specification by itself but relies on the first process. As the specifications of the processes are the same, the robustness conditions do not add any additional constraints.

```
r1 = false;
r2 = false;
w1 = false;
w2 = false;

//process P1

do{
  if(r1 && !r2 && !w1 && !w2) {
    r1 = false;
    r2 = true;
    w1 = false;
    w2 = false;
  }else if(!r1 && r2 && !w1 && !w2) {
    r1 = false;
    r2 = false;
    w1 = false;
    w2 = false;
  }else if(!r1 && !r2 && w1 && !w2) {
    r1 = false;
    r2 = false;
    w1 = false;
    w2 = true;
  }else if(!r1 && !r2 && !w1 && w2) {
    r1 = true;
    r2 = false;
    w1 = false;
    w2 = false;
  }else if(!r1 && !r2 && !w1 && !w2) {
    r1 = false;
    r2 = false;
    w1 = true;
    w2 = false;
  }else {
    r1 = false;
    r2 = false;
    w1 = false;
    w2 = false;
  }
} while(true);
```

```

//process P2

do{
  if(r1 && !r2 && !w1 && !w2) {
    r1 = true;
    r2 = false;
    w1 = false;
    w2 = false;
  }else if(!r1 && r2 && !w1 && !w2) {
    r1 = false;
    r2 = false;
    w1 = true;
    w2 = false;
  }else if(!r1 && !r2 && w1 && !w2) {
    r1 = false;
    r2 = false;
    w1 = true;
    w2 = false;
  }else if(!r1 && !r2 && !w1 && w2) {
    r1 = true;
    r2 = false;
    w1 = false;
    w2 = false;
  }else if(!r1 && !r2 && !w1 && !w2) {
    r1 = false;
    r2 = false;
    w1 = true;
    w2 = false;
  }else {
    r1 = false;
    r2 = false;
    w1 = false;
    w2 = false;
  }
} while(true);
```

Code Listing 5.11: Simplified solution with AGS

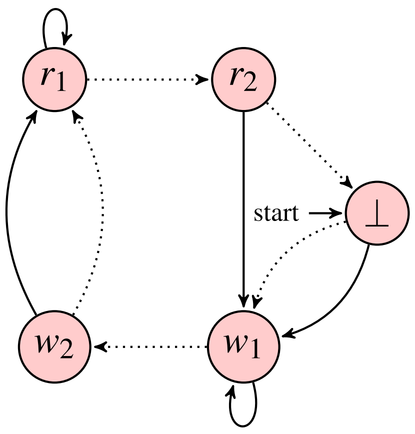

Figure 5.1.: Execution graph for the Read-Write problem. The execution of the first process is visualized using dotted edges. The execution of the second process is visualized using solid edges.

5.2. Performance Evaluation

All our experiments were preformed on a ordinary notebook equipped with a Intel® Core^{TM} 2 Duo CPU T9550 @ 2.66GHz, 4 GB RAM. Windows 8.1 served as operating system. To generate the universal co-Büchi automata, we used ltl3ba version 1.1.2. To solve the SMT constraints, the SMT solver Z3 version 4.3.2 was used.

Example	Total Time (sec)	Time Z3 (sec)	# Decisions	# Conflicts	Memory Z3 (MB)
Arbiter					
5.1.1.1	0.58	0.25	101	8	4.52
5.1.1.2	0.46	0.13	28	3	4.28
5.1.1.3	0.43	0.14	120	33	4.34
5.1.1.4	0.40	0.18	137	14	4.44
5.1.1.5	0.50	0.26	118	8	4.64
5.1.1.6	2.03	1.67	8,419	895	11.04
5.1.1.7	0.59	0.34	416	44	5.02
5.1.1.8	0.55	0.30	340	50	5.06
5.1.1.9	0.99	0.71	1,746	272	6.61
Memory					
5.1.2.1	1.56	0.93	1,349	334	5.46
5.1.2.2	0.47	0.26	976	147	4.54
R/W Problem					
5.1.3.1	2.9	2.56	58,772	4,080	19.83

Table 5.2.: Performance of the examples with cooperative co-synthesis

Example	Total Time (sec)	Time Z3 (sec)	# Decisions	# Conflicts	Memory Z3 (MB)
Arbiter					
5.1.1.1	2.38	1.75	219	10	6.48
5.1.1.2	0.43	0.12	28	3	4.28
5.1.1.3	0.44	0.15	120	33	4.34
5.1.1.4	0.48	0.18	139	39	4.44
5.1.1.5	2.03	1.54	147	10	9.27
5.1.1.6	220.69	219.88	23,277	491	19.39
5.1.1.7	21.06	20.54	891	32	9.20
5.1.1.8	63.57	63.10	1,117	33	9.49
5.1.1.9	49.52	48.91	514	40	10.37
Memory					
5.1.2.1	1.49	0.53	1139	195	4.97
5.1.2.2	0.67	0.29	1026	178	4.57
R/W Problem					
5.1.3.1	2.77	1.74	32,216	2,594	16.19

Table 5.3.: Performance of the examples with AGS

We compared our assume-guarantee synthesis approach to the cooperative co-synthesis. Table 5.2 shows the result for the cooperative co-synthesis approach and Table 5.3 shows the results for the assume-guarantee approach. The first column shows the runtime of the our prototype. As the performance heavily depends on the SMT solver, columns 2-4 show Z3 specific data. The second column shows the time Z3 needs to solve the constraint system. The number of decisions made by Z3 is shown in the third column and the number of conflicts is shown in the fourth column. The fifth column shows the memory consumed by Z3.

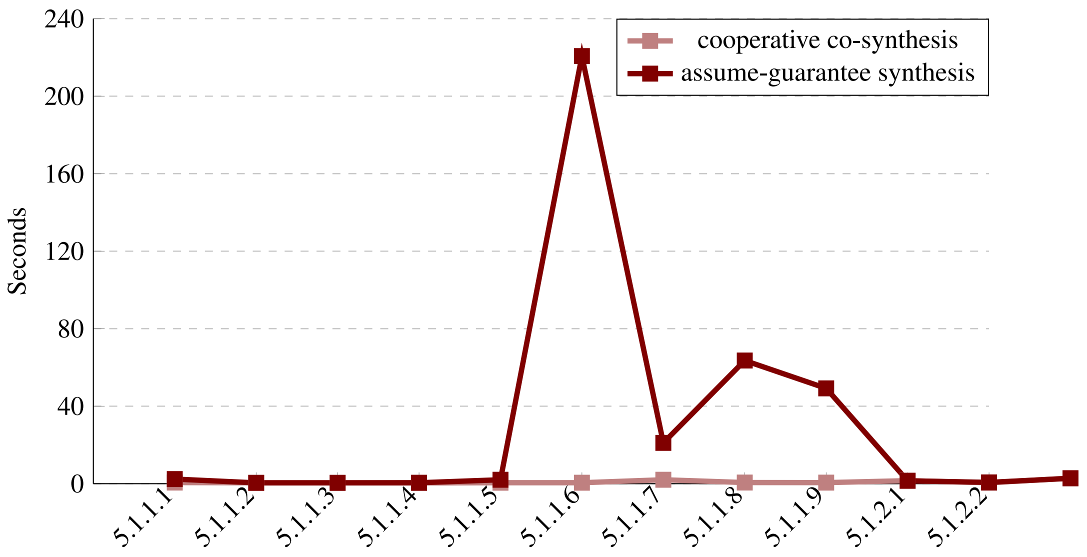

Figure 5.2.: Total time consumed by the prototype

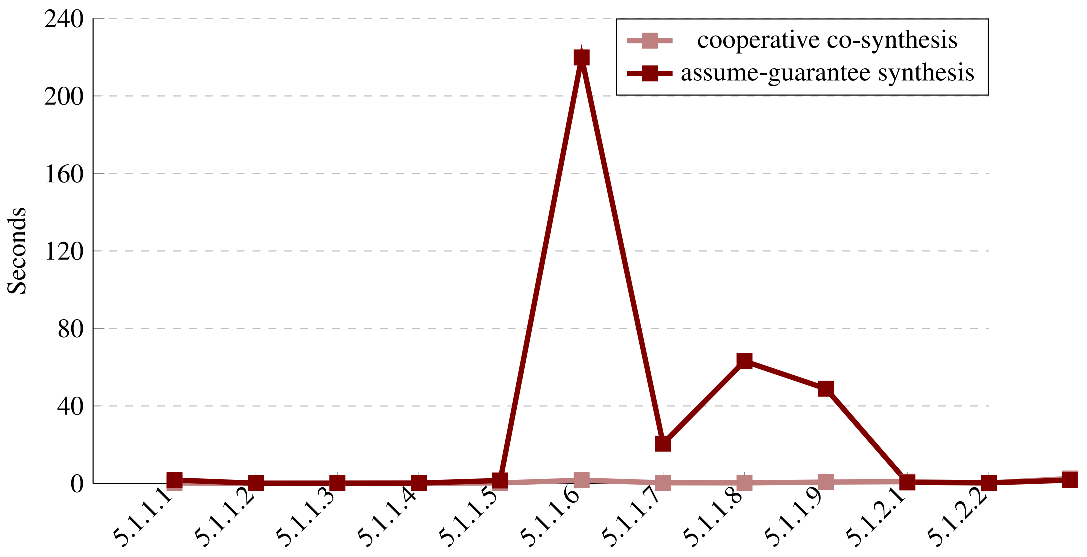

Figure 5.3.: Time consumed by the Z3 SMT solver

Figure 5.2 and Figure 5.3 illustrate the total runtime and the time spent by Z3 graphically. As it can be seen, in most our examples, the robustness conditions of AGS lead t plo a significant overhead. Whereas the time needed to realize the system is almost equal if the processes are defined equally, AGS consumes a lot more time to realize the specification. This is caused by the robustness conditions. If the specifications of the processes are equal, the robustness conditions are always true and consequently, the robustness conditions do not add any additional constraints.

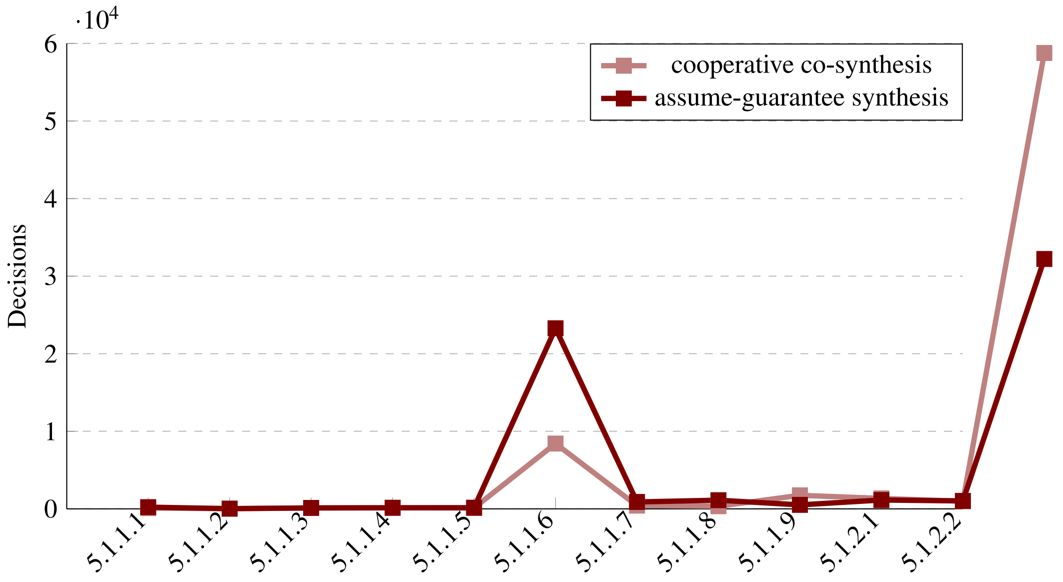

Figure 5.4.: Number of decision

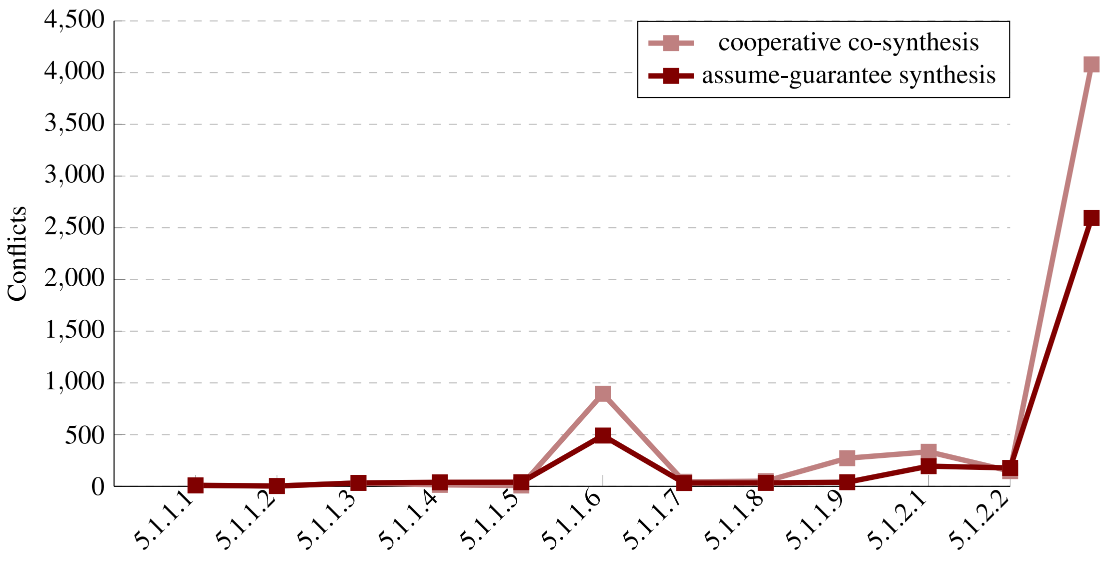

Figure 5.5.: Number of conflicts

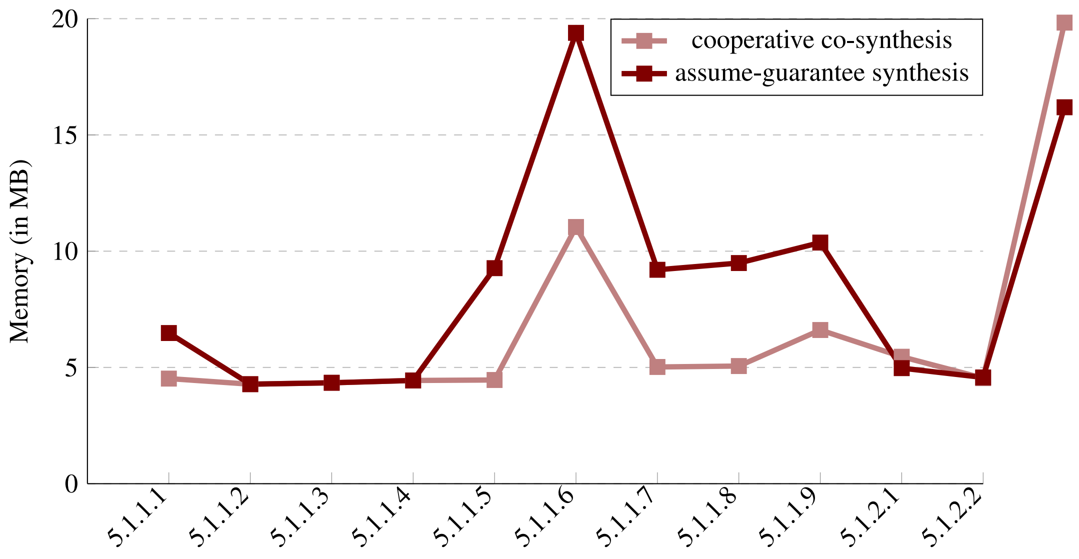

Figure 5.6.: Consumed memory

Figure 5.6 illustrates that the increase in memory consumption is quite moderate.

In some cases the robustness conditions gives the theory solver more information so that the solver converges faster and needs less time to find a model. This can be seen in example 5.1.3.1. Here the cooperative co-synthesis approach need 2.9 seconds, whereas AGS needs 2.77 seconds. Although the time saved is not a lot, the decrease in the number of decisions and conflicts are noticeable. AGS need about 20,000 fewer decisions and produces about 1,500 lesser conflicts than cooperative co-synthesis.

Chapter 6

Conclusion and Future Work

6.1. Summary

In this thesis, we presented an approach to solve assume-guarantee synthesis problem for two processes described by their LTL specifications. Furthermore, we described the implementation of our prototype and finally provided experimental results.

We extend the bounded synthesis approach by Schewe and Finkbeiner [31] to be able to construct a robust system that fulfils the three AGS conditions. Our approach starts by defining the AGS conditions based on the LTL specifications of the processes. These conditions are converted into universal co-Büchi automata. Afterwards, these automata are used to construct a constraint system. The constraint system defines the unknown transition functions of the processes. If an SMT solver is able to solve this constraint system the model of the transition functions can be used to implement the processes.

If the SMT solver is not able to find a solution, the system designer can increase the possible state space by adding memory variables that are not restricted by the specification. By increasing the state space, the SMT solver has more possibilities to find a transition function. If the number of memory variables is increased iteratively, our approach can be used to semi-decide the unbounded AGS problem.

Additionally, our approach supports partial information. This allows the processes to define private variables which cannot be read or altered by the other processes.

6.2. Conclusion

Our approach provides a system designer with a method to automatically construct correct and robust systems consisting of two processes based on their LTL specification.

Robustness has the advantage that it gives implementations that require less assumptions about the other processes and therefore the processes can be exchanged with other processes that fulfil the specification and the robustness conditions.

The support of partial information lead to a more flexible system as the processes can be decoupled form each other.

However, the experiments showed that the robustness condition of AGS lead, compared to cooperative co-synthesis, to a significant overhead. Additionally there are also less specifications realisable as the robustness conditions are very strict. This is especially true if additional memory variables need to be introduced because these variables are not constrained by the specification and can thus be set arbitrarily by an adversarial process.

6.3. Future Work

In this thesis, we defined an approach that handles exactly two processes and their LTL specifications. A future topic would be to eliminate the restrictions on the number of allowed processes to increase the number of possible applications of assume-guarantee synthesis.

Another restriction in our approach is the specification language. An interesting topic would be to adapt this approach for other formal specification languages.

In our work, the states are defined by the global, local and memory variables and the transitions are defined through the valuation of the inputs or the state has only one specific successor. Another possibility would be to enumerate the possible states and to label the transitions with the process and input variables.

Another topic concerning the variables would be to introduce variables that can be written by a single process and read by all processes. In our approach, this behaviour can be reached by defining a global variable and explicitly specifying that this variable should not be changed by the other process. However, this new type would simplify the specification and possibly decrease the runtime.

As the most time consuming part is the solving of the constraint system, another interesting topic would be to evaluate the performance of different SMT solvers for AGS problems. Another possibility would be to transfer the constraint system into a logical programming language.

Bibliography

[1] ARISTOTLE. De interpretatione. (Cited on page 5.)

[2] ARTHUR, N. 1957. Prior. time and modality. (Cited on page 5.)

[3] BABIAK, T., KŘETÍNSKỲ, M., ŘEHÁK, V., AND STREJČEK, J. 2012. Ltl to büchi automata translation: Fast and more deterministic. In *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 95–109. (Cited on page 33.)

[4] BARRETT, C., STUMP, A., AND TINELLI, C. 2010. The SMT-LIB Standard: Version 2.0. Tech. rep., Department of Computer Science, The University of Iowa. Available at `www.SMT-LIB.org`. (Cited on pages 28, 33, and 37.)

[5] BLOEM, R., CHATTERJEE, K., JACOBS, S., AND KÖNIGHOFER, R. 2015. Assume-guarantee synthesis for concurrent reactive programs with partial information. In *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*. 517–532. (Cited on page 4.)

[6] BÜCHI, J. R. 1960. *On a decision method in restricted second order arithmetic*. na. (Cited on page 11.)

[7] BUCHI, J. R. AND LANDWEBER, L. H. 1990. *Solving sequential conditions by finite-state strategies*. Springer. (Cited on page 2.)

[8] CHATTERJEE, K. AND HENZINGER, T. A. 2007. Assume-guarantee synthesis. In *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 261–275. (Cited on pages 2, 3, and 15.)

[9] CHURCH, A. 1962. Logic, arithmetic and automata. In *Proceedings of the international congress of mathematicians*. 23–35. (Cited on page 1.)

[10] COURTOIS, P.-J., HEYMANS, F., AND PARNAS, D. L. 1971. Concurrent control with “readers” and “writers”. *Communications of the ACM 14,* 10, 667–668. (Cited on page 49.)

[11] DAVIS, M., LOGEMANN, G., AND LOVELAND, D. 1962. A machine program for theorem-proving. *Communications of the ACM 5,* 7, 394–397. (Cited on page 7.)

[12] DUTERTRE, B. AND DE MOURA, L. 2006. A fast linear-arithmetic solver for DPLL (T). In *Computer Aided Verification*. Springer, 81–94. (Cited on page 10.)

[13] EILENBERG, S. AND TILSON, B. 1974. *Automata, languages, and machines*. Vol. 76. Academic press New York. (Cited on page 11.)

[14] EMERSON, E. A. AND CLARKE, E. M. 1982. Using branching time temporal logic to synthesize synchronization skeletons. *Science of Computer programming 2,* 3, 241–266. (Cited on page 2.)

[15] FILIOT, E., JIN, N., AND RASKIN, J.-F. 2011. Antichains and compositional algorithms for LTL synthesis. *Formal Methods in System Design 39,* 3, 261–296. (Cited on page 4.)

[16] FINKBEINER, B. AND SCHEWE, S. 2005. Uniform distributed synthesis. In *Logic in Computer Science, 2005. LICS 2005. Proceedings. 20th Annual IEEE Symposium on*. IEEE, 321–330. (Cited on page 2.)

[17] FISMAN, D., KUPFERMAN, O., AND LUSTIG, Y. 2010. Rational synthesis. In *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 190–204. (Cited on page 4.)

[18] KAIVOLA, R., GHUGHAL, R., NARASIMHAN, N., TELFER, A., WHITTEMORE, J., PANDAV, S., SLOBODOVÁ, A., TAYLOR, C., FROLOV, V., REEBER, E., ET AL. 2009. Replacing testing with formal verification in Intel® CoreTM i7 Processor Execution Engine Validation. In *Computer Aided Verification*. Springer, 414–429. (Cited on page 1.)

[19] LAGOUDAKIS, M. G. AND LITTMAN, M. L. 2001. Learning to select branching rules in the dpll procedure for satisfiability. *Electronic Notes in Discrete Mathematics 9*, 344–359. (Cited on page 7.)

[20] LISCOUSKI, B. AND ELLIOT, W. 2004. Final report on the august 14, 2003 blackout in the united states and canada: Causes and recommendations. *A report to US Department of Energy 40,* 4. (Cited on page 1.)

[21] LLC, M. 1999. MS Windows NT kernel description. (Cited on page 33.)

[22] MACFARLANE, J. 2003. Future contingents and relative truth. *The philosophical quarterly 53,* 212, 321–336. (Cited on page 5.)

[23] MANNA, Z. AND WOLPER, P. 1984. Synthesis of communicating processes from temporal logic specifications. *ACM Transactions on Programming Languages and Systems (TOPLAS) 6,* 1, 68–93. (Cited on page 2.)

[24] MCNAUGHTON, R. 1965. Finite-state infinite games. *Project MAC Rep*. (Cited on page 2.)

[25] NUSEIBEH, B. 1997. Ariane 5: who dunnit? *IEEE Software* 3, 15–16. (Cited on page 1.)

[26] PNEULI, A. AND ROSNER, R. 1990. Distributed reactive systems are hard to synthesize. In *Foundations of Computer Science, 1990. Proceedings., 31st Annual Symposium on*. IEEE, 746–757. (Cited on page 2.)

[27] PNUELI, A. 1977. The temporal logic of programs. In *Foundations of Computer Science, 1977., 18th Annual Symposium on*. IEEE, 46–57. (Cited on pages 2 and 5.)

[28] PNUELI, A. AND ROSNER, R. 1989. On the synthesis of a reactive module. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 179–190. (Cited on page 2.)

[29] RABIN, M. O. 1972. *Automata on infinite objects and Church's problem*. Vol. 13. American Mathematical Soc. (Cited on page 2.)

[30] RUSSELL, B. 1908. Mathematical logic as based on the theory of types. *American journal of mathematics 30,* 3, 222–262. (Cited on page 8.)

[31] SCHEWE, S. AND FINKBEINER, B. 2007a. Bounded synthesis. In *Automated Technology for Verification and Analysis*. Springer, 474–488. (Cited on pages 2, 3, 4, 16, 37, and 55.)

[32] SCHEWE, S. AND FINKBEINER, B. 2007b. Synthesis of asynchronous systems. In *Logic-Based Program Synthesis and Transformation*. Springer, 127–142. (Cited on page 12.)

[33] THOMAS, W. 1997. *Languages, automata, and logic*. Springer. (Cited on pages 10 and 11.)

[34] VARDI, M. Y. 2008. From church and prior to psl. In *25 years of model checking*. Springer, 150–171. (Cited on page 5.)

[35] ZHANG, L., MADIGAN, C. F., MOSKEWICZ, M. H., AND MALIK, S. 2001. Efficient conflict driven learning in a boolean satisfiability solver. In *Proceedings of the 2001 IEEE/ACM international conference on Computer-aided design*. IEEE Press, 279–285. (Cited on page 7.)

Appendix A

co-Büchi Automata

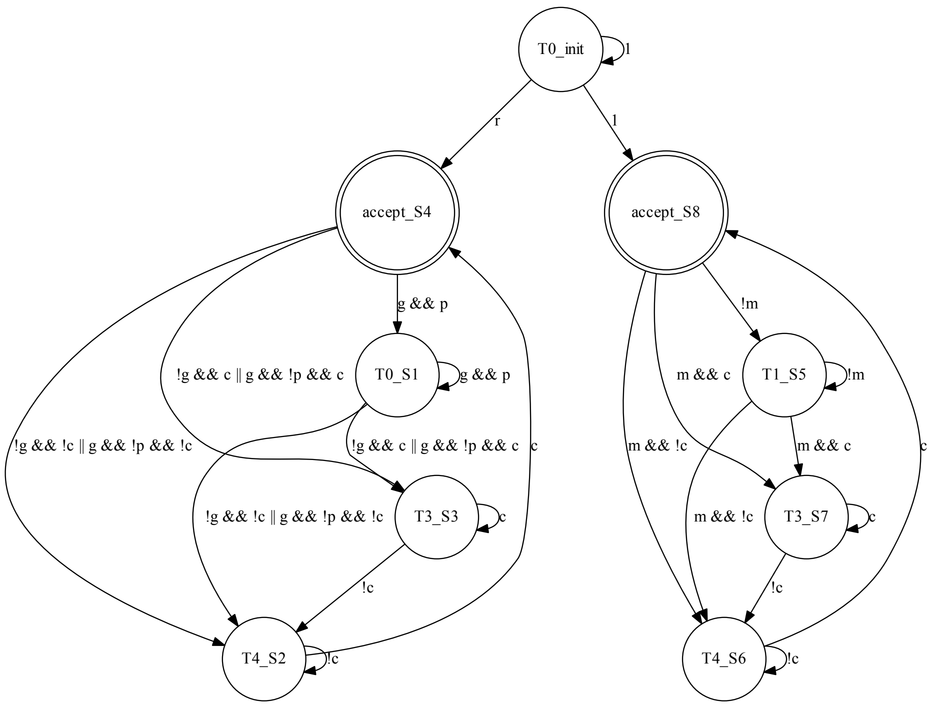

Figure A.1.: co-Büchi automaton of the first AGS-condition

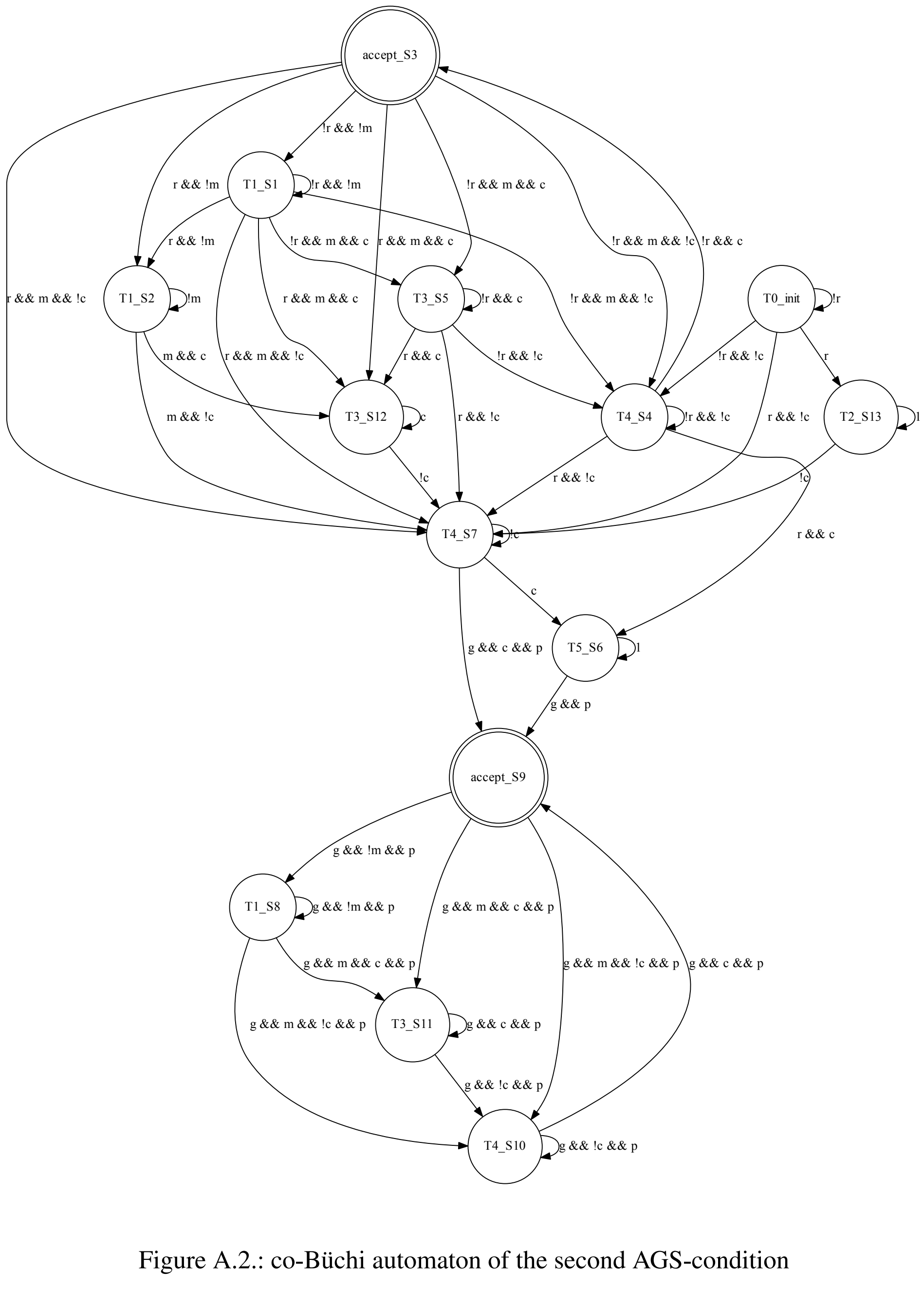

Figure A.2.: co-Büchi automaton of the second AGS-condition

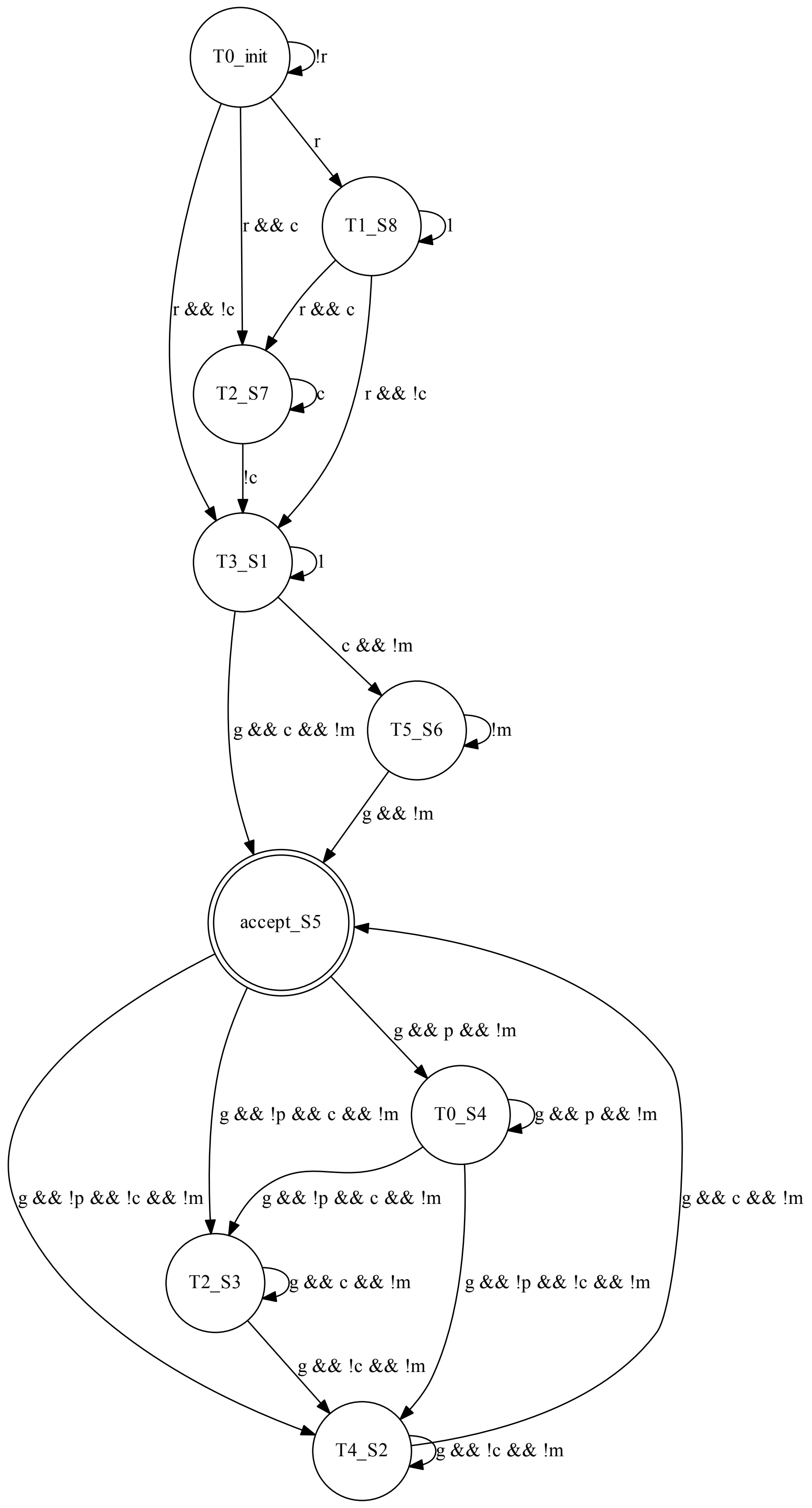

Figure A.3.: co-Büchi automaton of the second AGS-condition