Open Implication

A New Relation on Specifications Given in Linear Temporal Logic

by

Karin Greimel

Diploma Thesis

in Computer Sciences

Written at the Institute of Software Technology
at the Graz University of Technology

under the supervision of
Prof. Franz Wotawa, Dipl.-Ing. Dr.techn. and
Univ.Ass. Roderick Bloem, Ph.D.

November 21, 2007

Abstract

The wish to automatically construct correct systems from specifications has been around for about half a century. Church was the first one to describe this idea in 1962. Our work is closely related to this idea. We will consider systems which react to the environment denoted as open modules and our specification language is linear temporal logic (LTL).

Synthesis is the problem of automatically constructing correct open modules from an LTL specification. In 1992, it was shown to be 2EXP complete. This result discouraged many researchers to continue working on the problem since it seemed to be intractable. Only in recent years the topic was picked up again and new results leaded to new ideas.

The core idea of this work is a new relation on LTL formulas. An LTL formula *open implies* another formula, if every open module realizing the first formula also realizes the second formula. Hence open implication can be used to improve specifications or to synthesize smaller solutions.

Compared to trace inclusion, open implication is harder to calculate and weaker. Thus there are formulas such that one formula open implies the other but trace inclusion does not hold.

In this work we will formally define open implication. Then we will give an algorithm to calculate open implication for full LTL. The algorithm meets the lower bound of 2EXP in the first argument and PSPACE in the second. Subsequently, we explain an efficient algorithm for a subset of LTL, that of General Reactivity of Rank 1 (GR(1)). We implemented the algorithm in a tool called Anzu which will also be discussed together with the results of our case study.

Kurzfassung

Das Bestreben aus einer Spezifikation automatisch ein korrektes System zu generieren gibt es schon lange. Church war der erste, der diese Idee 1962 formulierte. In der vorliegenden Arbeit werden in diesem Zusammenhang offene Systeme betrachtet, das sind Systeme die mit ihrer Umgebung interagieren. Für die Spezifikation dieser Systeme verwenden wir eine temporale Logik namens Linear Temporal Logic (LTL).

Synthese ist das Problem, ein korrektes offenes System aus einer LTL Spezifikation automatisch zu erzeugen. Im Jahr 1992 wurde gezeigt, dass Synthese 2EXP-vollständig ist. Dieses Ergebnis hat viele Wissenschaftler entmutigt weiter an diesem Problem zu arbeiten da es als praktisch unlösbar galt. In den letzten Jahren wurde das Thema jedoch wiederbelebt und neue Ergebnisse führten zu neuen Ideen.

Der Kern dieser Arbeit ist eine neue Relation für LTL Spezifikationen. Eine LTL Formel *offen-impliziert* eine andere, wenn alle offenen Systeme welche die erste Formel realisieren auch die zweite Formel realisieren. Daher können mit Hilfe der offenen Implikation Spezifikationen verbessert und kleinere Systeme synthetisiert werden.

Verglichen zur "trace inclusion" ist die offene Implikation schwerer zu berechnen und schwächer. D.h. es gibt Formeln, sodass eine Formel die andere offen-impliziert aber "trace inclusion" nicht gilt.

Im Zuge dieser Arbeit werden wir offene Implikation formal definieren. Dann werden wir einen Algorithmus vorstellen der offene Implikation für LTL entscheidet. Der Algorithmus ist 2EXP im ersten Argument und PSPACE im zweiten, das entspricht der unteren Komplexitätsgrenze. Zusätzlich zeigen wir auch einen effizienten Algorithmus für Formeln aus General Reactivity of Rank 1 (GR(1)), einer Teilmenge von LTL. Diesen Algorithmus haben wir in dem Programm Anzu implementiert und an einem industriellen Beispiel gestestet.

Acknowledgments

First of all I want to thank my supervisor Roderick Bloem who encouraged me to work on this topic and who “infected” me with his enthusiasm. I really enjoyed working on my thesis and I learned a lot in this last year of my studies. I am very grateful for the many interesting discussions and the patience Roderick had for my questions and technical problems. I could not have wished for a better supervisor.

Second I want to thank Barbara Jobstmann for her personal and technical input. Although Barbara had to finish her dissertation she always had time to answer my questions. She also helped me with the implementation and the case study. I think without her ideas and help this work would not have come into existence the way it is now.

I am also grateful to Prof. Franz Wotawa, head of the Institute of Software Technology for his interest in my work and for finishing all the paperwork so promptly.

Thanks also to Martin Weiglhofer and Stefan Galler. They did a wonderful job implementing and documenting Anzu which was very helpful. They were also very forthcoming, answering my questions and solving problems with the case study.

Furthermore I am very grateful to my parents for supporting me in every way and who encouraged me during all of my years of study. Special thanks also go to my boyfriend Wolfgang Aigner who is always there for me.

Contents

Chapter 1

Introduction

High level *synthesis* is the automated process of deriving a correct system from a logic specification. This idea was first introduced by Church in 1962 [Chu62]. A very appealing property of synthesis is the correctness by construction. Especially for safety critical systems such as air traffic control or banking networks this would be a great benefit.

We consider specifications given in *linear temporal logic* (LTL) [Pnu77] with a set of variables divided into input and output variables. LTL is a specification language which is well suited to describe the behavior of a reactive program because of its temporal character. The specifications we consider describe the behavior of *open modules*, modules where the system reacts to the environment by reading the input variables and setting the output variables. Such specifications are hard to define using Hoare logic because open modules might run infinitely (e.g. an operating system). Using Hoare logic it is only possible to reason about the relation of the state before execution and after execution.

An example of an LTL specification is $\mathsf{G}(r \rightarrow \mathsf{X}\, a)$, where r is an input variable denoting a request and a is an output variable denoting an acknowledgment. The specification requires an acknowledgment after each request. Strictly speaking, always after a request an acknowledgment must follow in the next time step.

The classical method for synthesis introduced in [PR89] is to construct a Büchi word automaton from the specification. Then through determinization the automaton is converted into a parity tree automaton which is checked for emptiness. A witness for the non emptiness of the parity tree automaton represents an implementation of the specification. If the parity tree automaton is empty the specification is not *realizable*, i.e., it can not be synthesized. The problem of synthesis and deciding realizability are closely related, they are both 2EXP complete for full LTL [Ros92]. This very high complexity stems from the so called *state explosion problem.* Converting the formula into a word automaton leads to the first exponential blow up. The second exponential blow up is due to the determinization of the automaton. In Example 29 in Section 4.1 we show why the automaton has to be determinized and Example 7 in Section 2.6 illustrates the classical method for synthesis.

The high time complexity is one reason why the dream of synthesis was given up for some time. Another reason was the hard to understand and hard to implement algorithm for determinization by Safra [Saf88]. In recent years some work was done to revive the idea of synthesis and to start putting it to practice. The work presented in [KV05, KPV06, JB06] pursues the idea of Safraless synthesis. Another option to make the problem more feasible is to only consider a subset of LTL [PPS06, BGJ$^+$07a, BGJ$^+$07b].

In this work we take a closer look at the following aspect of synthesis. In the case of non emptiness of the tree automaton in the classical method for synthesis there can be many different witnesses and consequently many different implementations. Thus every specification has a set of open modules realizing it. We call two specifications with the same set of realizing open modules *open equivalent*. Fittingly, we say a specification *open implies* another specification if the set of open modules realizing the first specification is a subset of the set of open modules realizing the second specification. For an illustration of the idea of open implication see Example 8 in Chapter 3.

Open implication can help to synthesize smaller and better solutions from big specifications. Consider two specifications φ and ψ where φ open implies ψ. If synthesizing φ results in a better solution than synthesizing ψ, the better solution can also be used to realize ψ. This can be done because open implication assures that all open modules realizing the first specification φ also realize the second specification ψ. See Section 5.6 for the results of our experiments.

Unfortunately, due to the high complexity results, the notion of open implication for full LTL is mainly of theoretical interest right now. The aim of this work is twofold. First we want to analyze the problem and give an optimal algorithm deciding open implication. Second we want to describe our efficient implementation of open implication for a subset of LTL.

The following chapters will explain the tools and theories used to decide open implication and thus also open equivalence. First of all the notation and conventions which will be used further on will be explained. Then we give a short overview of related topics, and the first attempts to handle the task at hand are shown. Subsequently a proof for the lower bound and the general idea of the algorithms presented later on is discussed. Chapter 4 contains the actual algorithm to decide open implication for full LTL, which meets the lower bound derived in Section 3.5. In Chapter 5 an implementation for a smaller class of formulas, that of generalized reactivity of rank 1 (GR(1)), introduced by Piterman, Pnueli and Sa'ar in [Pit06] is described. Last but not least we will summarize and shortly discuss our conclusions.

Chapter 2

Preliminaries

2.1 Open Modules

An open module is a module which interacts with the environment. In order to enable this interaction, the variables used by the module are partitioned into input and output variables. The module reacts to the environment by reading the values of the input variables and assigning values to the output variables. Therefore, open modules are also sometimes referred to as reactive modules.

We will use two different kinds of open modules. Mealy machines react immediately to the input of the environment and Moore machines react one time step later.

Given a finite set of input variables I and a finite set of output variables O, a *Mealy machine* is defined as follows:

$$M = (\Sigma, D, S, s_0, \delta, \lambda)$$

Where $\Sigma = 2^O$ denotes the output alphabet, $D = 2^I$ the input alphabet, S a finite set of states, $s_0 \in S$ the initial state, $\delta : S \times D \to S$ the transition function and $\lambda : S \times D \to \Sigma$ the output function. To make notation easier we will extend the definition of the transition function to $\delta : S \times D^* \to S$, where $\delta(s, \epsilon) = s$ and $\delta(s, aw) = \delta(\delta(s, a), w)$ for a in D and w in D^*.

A *Moore machine* is defined alike, with the exception of the output function $\lambda : S \to \Sigma$. Hence a Moore machine is a special case of a Mealy machine, with the output function not depending on the input. Formally, a Mealy machine is a Moore machine if for all s in S and all d_1, d_2 in D the label $\lambda(s, d_1)$ is equal to $\lambda(s, d_2)$.

A *path* of an open module $M = (\Sigma, D, S, s_0, \delta, \lambda)$ is a sequence of states starting at the initial state and following the transition function. Thus a path is $\pi = (s_0)(\delta(s_0, d_0))(\delta(s_0, d_0 d_1))$ where $d_0, d_1 \ldots$ are inputs chosen by the environment. The corresponding *word* of an open module is an infinite sequence of (output, input) tuples in $\Sigma \times D$ defined by the output function. Formally, for $d_0, d_1 \ldots$ in D a word σ over $\Sigma \times D$ is defined as follows

1. for a Mealy machine: $\sigma = (\lambda(s_0, d_0), d_0)(\lambda(\delta(s_0, d_0), d_1), d_1) \dots$
2. for a Moore machine: $\sigma = (\lambda(s_0), d_0)(\lambda(\delta(s_0, d_0)), d_1) \dots$

The set of infinite sequences over $\Sigma \times D$ is denoted by $(\Sigma \times D)^\omega$. The ω stems from the notion of ω-words which are infinite words. For a short introduction on ω-words and ω-languages refer to [Tho90].

Two open modules are equivalent if they produce the same output given any input, thus they are equivalent if they define the same set of words. Clearly there exists an equivalent Mealy machine for any Moore machine, because every Moore machine is also a Mealy machine. On the other hand, it is not possible to construct an equivalent Moore machine for any arbitrary Mealy machine, see Example 2 below. But there always exists a modified Moore machine which defines the same set of words as an arbitrary Mealy machine. Let $M = (\Sigma, D, S, s_0, \delta, \lambda)$ be a Moore machine then a word σ of the modified Moore machine $M^{[o_i \leftarrow o_{i+1}]}$ over $\Sigma \times D$ is defined as follows:

$$\sigma = (\lambda(\delta(s_0, d_0)), d_0)(\lambda(\delta(s_0, d_0 d_1)), d_1) \dots .$$

Thus the first output of the original Moore machine is ignored.

Claim 1. *Given a Mealy machine $M_e = (\Sigma, D, S, s_0, \delta, \lambda_e)$. Let $M_o = (\Sigma, D, S, s_0, \delta, \lambda_o)$ be the Moore machine with the same input alphabet, the same output alphabet and the same states as M_e. The labeling function λ_o is defined as follows: for the initial state s_0 choose an arbitrary label in Σ, for all other states $\delta(s, d)$ for s in S and d in D set $\lambda_o(\delta(s, d))$ equal to $\lambda_e(s, d)$.*

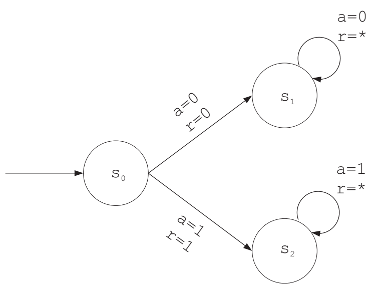
The modified Moore machine $M_o^{[o_i \leftarrow o_{i+1}]}$ defines the same set of words as the Mealy machine M_e.

Proof. A word $\sigma = (\lambda_e(s_0, d_0), d_0)(\lambda_e(\delta(s_0, d_0), d_1), d_1) \dots$ of the Mealy machine M_e is equal to a word $\sigma = (\lambda_o(\delta(s_0, d_0)), d_0)(\lambda_o(\delta(s_0, d_0 d_1)), d_1) \dots$ of the modified Moore machine $M_o^{[o_i \leftarrow o_{i+1}]}$ because $\lambda_o(\delta(s, d))$ is equal to $\lambda_e(s, d)$ for all d in D. □

Thus a modified Moore machine $M^{[o_i \leftarrow o_{i+1}]}$ can react to inputs in the same way as Mealy machines, only one time step later.

Example 2. This example shows a Mealy machine for which no equivalent Moore machine exists and the Moore machine M_0 such that $M_0^{[o_i \leftarrow o_{i+1}]}$ defines the same set of words as the Mealy machine.

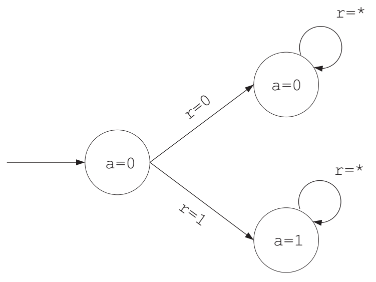
The Mealy machine is $M = (\Sigma, D, S, s_0, \delta, \lambda)$ with the output alphabet $\Sigma = \{(a = 0), (a = 1)\}$, the input alphabet $D = \{(r = 0), (r = 1)\}$, the set of states $S = \{s_0, s_1, s_2\}$ and the transitions $\delta(s_0, r = 0) = s_1$, $\delta(s_0, r = 1) = s_2 \dots$. The complete transition function is defined by the graph in Figure 2.1. The transitions are denoted as arrows pointing from the current state to the next state, depending on the input, which is written above each arrow. Transition arrows with $r = *$ mean that the transition does not depend on the input. The labeling function is defined by the table in Figure 2.1 and also in the graph. The labels in the graph are written above the transition arrows to illustrate the dependence on the input.



state, input	label
$s_0, r = 0$	$a = 0$
$s_0, r = 1$	$a = 1$
$s_1, r = *$	$a = 0$
$s_2, r = *$	$a = 1$

Figure 2.1: Mealy machine

There exists no equivalent Moore machine because the labeling function of a Moore machine is of the form $\lambda : S \rightarrow \Sigma$ and thus it can not react to the first input like the Mealy machine. But it is possible to find a Moore machine M_0 such that $M_0^{[o_i \leftarrow o_{i+1}]}$ defines the same set of words as the Mealy machine. This Moore machine is shown in Figure 2.2. The input alphabet, the output alphabet, the set of states and the transition function are the same as for the Mealy machine M above. The labeling function is defined in the table in Figure 2.2. In the graph the labels are written into the states, to illustrate the independence from the input.



state	label
s_0	$a = 0$
s_1	$a = 0$
s_2	$a = 1$

Figure 2.2: Moore machine

The Moore machine accepts all words either starting with the sequence $\binom{a=0}{r=0}\binom{a=0}{r=*}\binom{a=0}{r=*}\ldots$ or with the sequence $\binom{a=0}{r=1}\binom{a=1}{r=*}\binom{a=1}{r=*}\ldots$. If the first output is ignored the resulting set of words consists of all words either starting with the sequence $\binom{a=0}{r=0}\binom{a=0}{r=*}\ldots$ or with the sequence $\binom{a=1}{r=1}\binom{a=1}{r=*}\ldots$ which is the set of words defined by the Mealy machine M above. □

2.2 LTL

Linear Temporal Logic is a convenient language to describe the behavior of reactive modules. It was introduced by Pnueli in [Pnu77] to describe properties of infinite computations where the specification of an end state is not possible. Consequently, LTL specifications focus on the ongoing behavior of possibly infinitely running systems.

We will give a short introduction to the syntax and semantics of LTL future formulas. For further reading we recommend [MP91].

The *syntax of an LTL formula* over a set of atomic propositions P is defined as follows.

1. Every atomic proposition p in P is an LTL formula,
2. if φ and ψ are LTL formulas then $\neg\varphi$ and $\varphi \wedge \psi$ are LTL formulas
3. if φ and ψ are LTL formulas then $\mathsf{X}\,\varphi$ and $\varphi\,\mathsf{U}\,\psi$ are LTL formulas

The operators used in the definition are denoted as the elementary operators. Other connectives can be seen as abbreviations:

$$\begin{array}{ll} \varphi \vee \psi = \neg(\neg\varphi \wedge \neg\psi) & \mathsf{F}\,\varphi = \mathsf{true}\,\mathsf{U}\,\varphi \\ \varphi \rightarrow \psi = \neg\varphi \vee \psi & \mathsf{G}\,\varphi = \neg\,\mathsf{F}\,\neg\varphi \\ \varphi \leftrightarrow \psi = (\varphi \wedge \psi) \vee (\neg\varphi \wedge \neg\psi) & \varphi\,\mathsf{W}\,\psi = (\varphi\,\mathsf{U}\,\psi) \vee \mathsf{G}\,\varphi \end{array}$$

The *semantics of an LTL formula* over a set of atomic propositions P are recursively defined over infinite sequences $\sigma = \sigma_0\sigma_1\sigma_2\ldots$ where σ_i is in 2^P. The infinite subsequence σ^i of σ denotes the suffix of σ starting at σ_i, thus $\sigma^i = \sigma_i\sigma_{i+1}\ldots$.

1. Atomic Propositions:
   - $p \in P : \sigma^i \models p$ iff $p \in \sigma_i$
2. Boolean Operators:
   - $\sigma^i \models \neg\varphi$ iff not $\sigma^i \models \varphi$
   - $\sigma^i \models \varphi \wedge \psi$ iff $\sigma^i \models \varphi$ and $\sigma^i \models \psi$
3. Temporal Operators:
   - $\sigma^i \models \mathsf{X}\,\varphi$ iff $\sigma^{i+1} \models \varphi$
   - $\sigma^i \models \varphi\,\mathsf{U}\,\psi$ iff $\exists j \geq i : \sigma^j \models \psi$ and $\forall k, i \leq k < j : \sigma^k \models \varphi$

We will often refer to X as next and to U as until. The semantics of the operators globally (G), eventually (F) and weak until (W) can be derived from the semantics of the elementary operators.

- $\sigma^i \models \mathsf{G}\,\varphi$ iff $\forall j \geq i : \sigma^j \models \varphi$

- $\sigma^i \models \mathsf{F}\,\varphi$ iff $\exists j \geq i : \sigma^j \models \varphi$
- $\sigma^i \models \varphi\,\mathsf{W}\,\psi$ iff $\sigma^i \models \varphi\,\mathsf{U}\,\psi$ or $\sigma^i \models \mathsf{G}\,\varphi$

We use $\sigma \models \varphi$ instead of $\sigma^0 \models \varphi$ denoting that the sequence σ *satisfies* the formula φ.

A word $\sigma = (c_0, d_0)(c_0, d_1)\ldots$ in $(2^O \times 2^I)^\omega$ of a Moore machine (respectively Mealy Machine) $M = (2^O, 2^I, S, s_0, \delta, \lambda)$ satisfies an LTL formula over atomic propositions $(O \cup I)$ if the sequence $\sigma' = (c_0 \wedge d_0)(c_1 \wedge d_1)\ldots$ satisfies the LTL formula.

A Moore machine (respectively Mealy machine) M *realizes* an LTL formula φ ($M \models \varphi$) if all possible words of the Moore Machine (respectively Mealy machine) satisfy the formula. An LTL formula is *Moore realizable* (respectively *Mealy realizable*) if and only if there exists a Moore machine (respectively Mealy machine) that realizes the formula. The difference between Mealy and Moore realizability will be discussed in Section 3.2. If not stated otherwise we will use realizability short for Moore realizability.

Example 3. This example shows how the Moore machine M_0 defined in Example 2 in Section 2.1 realizes the specification $(r \wedge \mathsf{X}\,a) \vee (\neg r \wedge \mathsf{X}\,\neg a)$.

All words of the Moore machine M_0 either start with the sequence $\binom{a=0}{r=0}\binom{a=0}{r=*}\binom{a=0}{r=*}\ldots$ or with the sequence $\binom{a=0}{r=1}\binom{a=1}{r=*}\binom{a=1}{r=*}\ldots$. Words starting with the sequence $\binom{a=0}{r=0}\binom{a=0}{r=*}\binom{a=0}{r=*}\ldots$ satisfy the $(\neg r \wedge \mathsf{X}\,\neg a)$ part of the specification and words starting with the sequence $\binom{a=0}{r=1}\binom{a=1}{r=*}\binom{a=1}{r=*}\ldots$ satisfy the $(r \wedge \mathsf{X}\,a)$ part of the specification. Consequently all words of the Moore machine satisfy the specification and the specification is realizable.

□

For specifications given in LTL we define two related problems. The first one is *realizability* which is the problem of finding out if a given formula is realizable. The second one is *synthesis* which is the problem of finding an open module which realizes the formula.

We finish this section with one more definition. The *size* of an LTL formula φ ($|\varphi|$) is the number of symbols in φ. When we state complexity bounds for a problem concerning LTL formulas, the bound is usually given in the size of the formula.

2.3 Kripke Structures

We define Kripke structures in order to be able to define the semantics of the temporal logic defined in Section 2.4. A Kripke structure is a closed model; that is, we do not distinguish between input and output variables. We define a *Kripke structure* as follows:

$$(S, T, S_0, A, L),$$

where S denotes a finite set of states, $T \subseteq S \times S$ a complete transition relation, $S_0 \subseteq S$ the set of initial states, A a finite alphabet and $L : S \to A$ the labeling function. A transition relation is complete if every state has at least one successor.

As for open modules the *path* of a Kripke structure is a sequence of states. The sequence starts with an initial state s_0 in S_0 and it satisfies the transition relation. The corresponding *word* is defined by the labeling function. Formally a path is $\pi = s_0 s_1 s_2 \dots$ where (s_0, s_1), $(s_1, s_2) \dots$ are in T. The corresponding word over A is $\sigma = L(s_0)L(s_1)L(s_2) \dots$.

It is possible to translate any open module into an equivalent Kripke structure, so that they both define the same set of words. We will show how a Mealy machine can be converted into a Kripke structure. The construction for Moore machines follows automatically.

Let $M = (\Sigma, D, S_M, s_0, \delta, \lambda)$ be a Mealy machine, then the corresponding Kripke structure is $K = (S_K, T, S_0, A, L)$ with

- $S_K = S_M \times D$
- $((s_M, d), (s'_M, d'))$ is in T iff $\delta(s_M, d) = s'_M$
- $S_0 \subseteq S_K$ are all states (s_M, d) where $s_M = s_0$
- $A = \Sigma \cup D$
- $L : S_K \to A$, where $L((s_M, d)) = \lambda(s_M, d) \cup d$

Claim 4. *The Kripke structure K defines the same set of words as the Mealy machine M.*

Proof. We prove Claim 4 by showing that any word of the Mealy machine is also a word of the Kripke structure and vice versa. Assume that the sequence $\sigma = (\lambda(s_0, d_0), d_0)(\lambda(\delta(s_0, d_0), d_1), d_1) \dots$ with $d_0, d_1 \dots$ in D is a word of the Mealy machine M then it is also a word of the Kripke structure K because the sequence $(\lambda(s_0, d_0), d_0)(\lambda(\delta(s_0, d_0), d_1), d_1) \dots$ is equal to the sequence $L((s_0, d_0))L((s_1, d_1)) \dots$ where $((s_0, d_0), (s_1, d_1))$ is in T. The same argument can be used to show that every word of the Kripke structure K is also a word of the Mealy machine M. $\square$

Example 5. This example shows the equivalent Kripke structure to the Mealy machine introduced in Example 2 in Section 2.1. Figure 2.3 shows the Mealy machine $M = (\Sigma, D, S, s_0, \delta, \lambda)$ again and Figure 2.4 shows the equivalent Kripke structure. The Kripke structure has $|S| \cdot |D|$ states and $|D|$ initial states. The initial states are the states $(s_0, r = 0)$ and $(s_0, r = 1)$, the transitions are denoted as arrows and the labels are in the boxes in the graph.

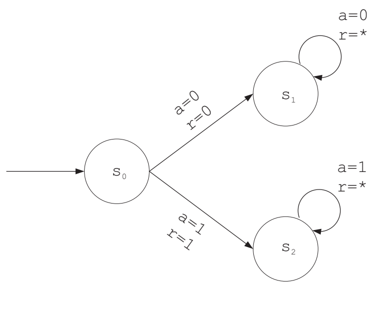


Figure 2.3: Mealy Machine

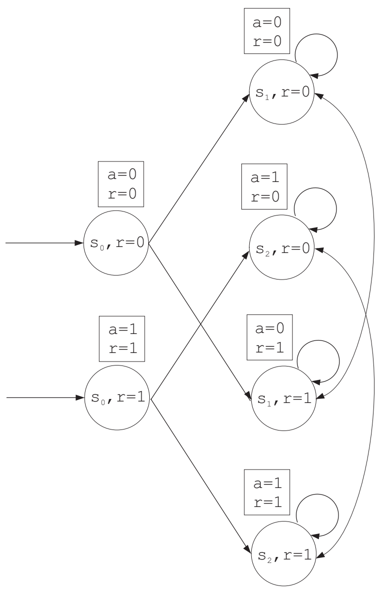


Figure 2.4: Kripke Structure

It is easy to see that the two graphs generate the same set of words. In both cases a word either starts with the sequence $\binom{a=0}{r=0}\binom{a=0}{r=*}\binom{a=0}{r=*}\ldots$ or with the sequence $\sigma = \binom{a=1}{r=1}\binom{a=1}{r=*}\binom{a=1}{r=*}\ldots$. □

2.4 CTL*

CTL* was introduced by Emerson and Halpern in [EH86]. Like LTL, it is also a temporal logic, in fact, it is an extension of LTL. In LTL, an open module realizes a specification, if all words of the realizing module satisfy the formula. Thus in LTL we can not define modules with at least one word satisfying a specification. Further on we often want to specify an open module where not all words satisfy an LTL formula φ, which is the same as saying at least one word satisfies $\neg\varphi$. Thus we want to quantify over words of modules.

Notice that "an open module M does not realize φ" ($M \not\models \varphi$) is not the same as "the open module M realizes $\neg\varphi$" ($M \models \neg\varphi$). For example, the Mealy machine shown in Figure 2.3 in Example 5 above does not realize the specification $\varphi = a$, but it does also not realize its negation $\neg\varphi = \neg a$, because not all words of the Mealy machine satisfy φ or $\neg\varphi$.

In order to define the *syntax of an CTL* formula* over a set of atomic propositions P we must define state and path formulas as follows.

1. Every atomic proposition p in P is a state formula,

2. if φ and ψ are state formulas then $\neg\varphi$ and $\varphi \wedge \psi$ are state formulas

3. if φ is a path formula then $\mathsf{E}\,\varphi$ is a state formula
4. Every state formula is also a path formula
5. if φ and ψ are path formulas then $\neg\varphi$ and $\varphi \wedge \psi$ are path formulas
6. if φ and ψ are path formulas then $\mathsf{X}\,\varphi$ and $\varphi\,\mathsf{U}\,\psi$ are path formulas

The set of state formulas generated by the above rules define the language CTL*. Again the abbreviations from Section 2.2 are commonly used, additionally $\mathsf{A}\,\varphi$ abbreviates $\neg\,\mathsf{E}\,\neg\varphi$.

The *semantics of an CTL* formula* over a set of atomic propositions P are recursively defined over states and paths of Kripke structures. Let $K = (S, T, S_0, 2^P, L)$ be a Kripke structure, s in S a state and $\pi = s_0 s_1 s_2 \ldots$ a path of the Kripke structure.

- $p \in P : s \models p$ iff $p \in L(s)$
- $s \models \neg\varphi$ iff not $s \models \varphi$
- $s \models \varphi \wedge \psi$ iff $s \models \varphi$ and $s \models \psi$
- $s \models \mathsf{E}\,\varphi$ iff there exists a path π starting at state s and $\pi \models \varphi$
- $\pi^i \models \varphi$ iff $s_i \models \varphi$ for a state formula φ
- $\pi^i \models \neg\varphi$ iff not $\pi^i \models \varphi$
- $\pi^i \models \varphi \wedge \psi$ iff $\pi^i \models \varphi$ and $\pi^i \models \psi$
- $\pi^i \models \mathsf{X}\,\varphi$ iff $\pi^{i+1} \models \varphi$
- $\pi^i \models \varphi\,\mathsf{U}\,\psi$ iff $\exists j \geq i : \pi^j \models \psi$ and $\forall k, i \leq k < j : \pi^k \models \varphi$

We will often refer to E as the existential path quantifier. The semantics of the universal path quantifier A can be derived from the semantics of the existential path quantifier and the negation.

$$s \models \mathsf{A}\,\varphi \text{ iff } \varphi \text{ holds for every path } \pi \text{ starting at state } s$$

An open module M *satisfies* a CTL* state formula φ if all initial states of the corresponding Kripke structure $K = (S, T, S_0, A, L)$ satisfy the formula:

$$M \models \varphi \Leftrightarrow \forall s_0 \in S_0 : s_0 \models \varphi$$

A CTL* formula is *Moore satisfiable* (respectively *Mealy satisfiable*) if there exists a Moore (respectively Mealy) machine which satisfies the formula.

Thus an open module realizes an LTL formula φ if and only if the open module satisfies the CTL* formula $\mathsf{A}\,\varphi$ and an open module M does not realize an LTL formula φ if and only if the module satisfies the CTL* formula $\mathsf{E}\,\neg\varphi$.

For a more detailed discussion of CTL* and other temporal logics see [Eme90].

2.5 Trees

A Σ-*labeled* D-*tree* is defined as follows:

$$t = (T, \tau)$$

Where $T \subseteq D^*$ is prefix-closed and $\tau : T \to \Sigma$ is the labeling function. A tree is *complete* if $T = D^*$.

The elements t_i of T are the *nodes* of the tree. The root of the tree $t_0 = \epsilon$ and t_{i+1} is a *successor* of t_i if $t_{i+1} = t_i d_i$ for d_i in D. A *path* of a tree is a sequence of nodes and the corresponding *word* of the tree is a sequence of tuples of letters from the alphabet Σ and directions in D. Formally a path is $\pi = t_0 t_1 t_2 \ldots$, where $t_0, t_1, t_2 \ldots$ are elements of T and $t_{i+1} = t_i d_i$. The corresponding word is $\sigma = (\tau(\epsilon), d_0)(\tau(t_1), d_1)(\tau(t_2), d_2) \ldots$.

A tree is Moore- respectively Mealy-*regular* if there exists a Moore respectively Mealy machine generating the same words as the tree. If not stated otherwise, regular denotes Moore-regular. Thus every regular tree can be represented by a Moore machine and vice versa.

2.6 Automata

In the context of synthesis or realizability, automata are often used to represent a set of infinite computations. We usually define such a set of computations by a specification in a temporal logic and then convert the specification into an automaton, which is easier to work with. At the end of this section we will give an example of synthesis, which shows very nicely how automata are used in this context.

In this section we follow the notation used by Kupferman and Vardi [KV05] and Jobstmann and Bloem [JB06].

An *alternating tree automaton* is defined as follows:

$$A = (\Sigma, D, Q, q_0, \delta, Acc)$$

Where Σ denotes a finite alphabet, D a finite set of directions, Q a finite set of states, $q_0 \in Q$ the initial state, δ the transition function and Acc the acceptance component.

Let $\mathcal{B}^+(D \times Q)$ specify the set of positive Boolean formulas over $D \times Q$. This includes the formulas false, true and all Boolean formulas built using the elements of $D \times Q$ and the operators $\vee$ and $\wedge$. The transition function of an automaton is of the form $\delta : Q \times \Sigma \to \mathcal{B}^+(D \times Q)$.

For now, let us assume the formulas in δ are written in disjunctive normal form. An automaton is *nondeterministic* if each element of D occurs at most once in each disjunction. An automaton is *deterministic* if it is nondeterministic and the formulas in δ are only conjunctions. The transition function of a deterministic automaton can also be written in the form $\delta : Q \times \Sigma \times D \to Q$.

We will use three different acceptance components:

- *Büchi acceptance*: Acc is a subset of Q called the accepting states.
- *Parity acceptance*: Acc is a coloring function $c : Q \to \{0, \dots, k\}$, which assigns a color between 0 and k (the *index*) to each state of the automaton.
- *Street acceptance*: $Acc = ((G_1, R_1) \dots (G_n, R_n))$, where all G_i and R_i are subsets of Q for $i = 1 \dots n$.

If $|D| = 1$ the automaton is a *word* automaton else it is a *tree* automaton. When we use word automata we often omit the set of directions from the definition.

The following abbreviations will be used further on: A, N or D for alternating, nondeterministic or deterministic; B, P or S for Büchi, parity or Street acceptance and W or T for word or tree automata (i.e. an NBW is a nondeterministic Büchi word automaton).

A *run* of a tree automaton $A = (\Sigma, D, Q, q_0, \delta, Acc)$ on a tree $t = (T, \tau)$ is a $(T \times Q)$-labeled $\mathbb{N}$-tree (T^r, τ^r). Every node in the run tree corresponds to a node in the input tree. The tree can be defined recursively starting at the root, the label of the root is $\tau^r(\epsilon) = (\epsilon, q_0)$. Let t_i^r be a node of the run tree T^r with $\tau^r(t_i^r) = (t_i, q_i)$. A set of successor nodes $\{t_{i+1,1}^r \dots t_{i+1,n}^r\}$ has labels $\tau^r(t_{i+1,1}^r) = (t_i d_1, q_1) \dots \tau^r(t_{i+1,n}^r) = (t_i d_n, q_n)$ such that $t_i d_1 \dots t_i d_n$ are successors of t_i in t and $(q_1, d_1) \wedge \dots \wedge (q_n, d_n) \models \delta(q_i, \tau(t_i))$.

A run is *accepting* if all infinite words starting at the root of the run tree satisfy the acceptance condition.

- A word satisfies the Büchi condition if a state in Acc occurs infinitely often.
- A word satisfies the parity condition if the maximal color occurring infinitely often is even. The colors are determined by applying the coloring function c to the states in the word.
- A word satisfies the Street condition if for all i, a state in the word occurring infinitely often is an element of R_i then also a state from G_i occur infinitely often.

Intuitively, when a tree automaton reads an input tree, it reads the input of the current node and moves according to the transition function into the next states. If all sequences of states visited by the automaton satisfy the acceptance condition, the tree is accepted.

The *language* of an automaton is the set of all trees accepted by the automaton. A tree is accepted if there exists at least one accepting run.

Theorem 6. (Rabin Basis Theorem [Rab72]) *The emptiness of the language of a tree automaton can be calculated and if it is not empty, there exists a regular tree which is accepted by the tree automaton.*

As already mentioned at the beginning of this section, we want to be able to represent a set of infinite computations which is specified by an LTL or CTL* formula, with the help of automata. For any LTL formula one can construct an ABW with a linear number of states in the size of the formula, or an NBW with an exponential number of states in the size of the formula, which accepts all words satisfying the formula. For proofs, algorithms and examples see [Var96]. Furthermore, it is possible to construct a DPT that accepts all trees where all words satisfy the same LTL formula. This construction will be explained in more detail in Section 4.1. For any CTL* formula one can construct a DST with a double exponential number of states in the size of the formula, which accepts all trees satisfying the formula, this was shown by Emerson and Sistla in [ES84]. The tree language L_φ is defined as the set of all trees accepted by the corresponding tree automaton for φ.

Next we give an example showing the classical method to solve realizability and synthesis for LTL specifications. This method heavily relies on automata theory, it was introduced by Pnueli and Rosner in [PR89], which also proved that realizability and synthesis are both 2EXP complete in the size of the formula.

Example 7. The specification $\varphi = \mathsf{X}\, a \vee \mathsf{X}\, r$ with input variables $I = \{r\}$ and output variables $O = \{a\}$ denotes all sequences with an a or an r in the second position. The Büchi word automaton representing the same language is defined as follows: $A^w_\varphi = (\Sigma, Q, q_0, \delta, Acc)$ with $\Sigma = 2^I \times 2^O$, $Q = \{A, B, C, D\}$ and $q_0 = A$. The transition function is defined by the arrows in Figure 2.5. Arrows labeled with T are transitions which do not depend on the values of the variables. The accepting state C is marked with an additional circle in the graph.

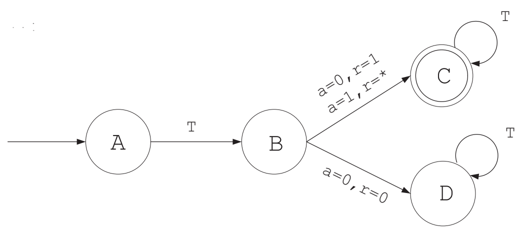


Figure 2.5: DBW for $\mathsf{X}\, a \vee \mathsf{X}\, r$

The next step would be to determinize the automaton (see Example 29 in Section 4.1), but A^w_φ is already deterministic so we can skip this step.

In order to decide realizability or to synthesize the specification, the tree automaton $A^t_\varphi = (\Sigma, D, Q, q_0, \delta, Acc)$ with $\Sigma = \{(a = 0), (a = 1)\}$ and

$D = \{(r = 0), (r = 1)\}$ is constructed. The automaton accepts all trees where all words satisfy φ. Since $|D| = 2$ it is a binary tree automaton. The set of states and the initial state are the same as for A^w_φ, also the transitions stay the same except that we have to distinguish between input and output variables. This is visualized in Figure 2.6 by small boxes. The arrows before the boxes describe the possible choices of the system, they are labeled with the letters of the alphabet of the automaton, e.g. the output variables. The arrows after the boxes describe the possible choices of the environment, they are labeled with directions, e.g. input variables. For example the transition $\delta(B, a = 0) = ((r = 0) \wedge D) \vee ((r = 1) \wedge C)$ is illustrated by an arrow starting at state B labeled with $a = 0$ and pointing at a box and two more arrows starting at the box, labeled with $r = 0$ respectively $r = 1$ and pointing to the states D respectively C.

The Büchi acceptance condition of the DBW is converted into a parity acceptance condition with the following coloring:

$$c(q) = \begin{cases} 2 & \text{for } q = C \\ 1 & \text{else} \end{cases}$$

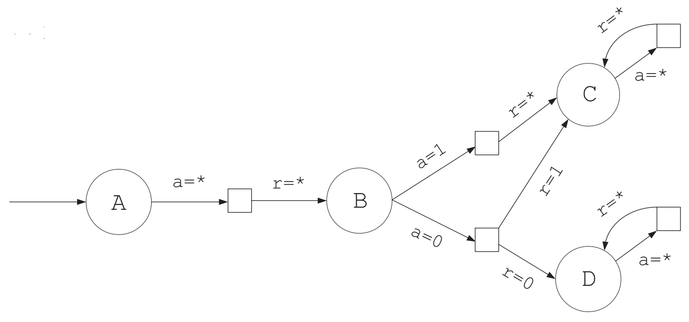
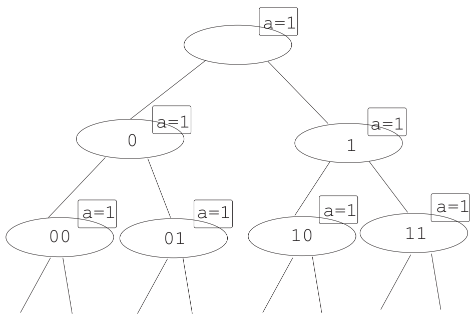


Figure 2.6: DPT for $\mathsf{X}\,a \vee \mathsf{X}\,r$

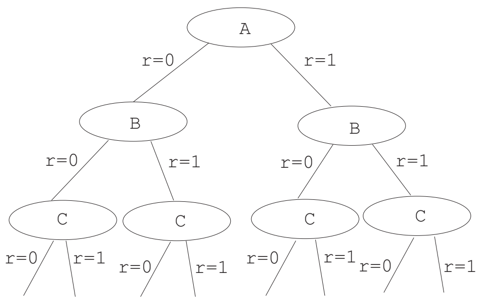
Subsequently we must check if the DPT has a non-empty language. This is usually done using game theory, see Section 4.3 for a short explanation, but in this case it is easy to find an input tree with an accepting run by hand. The input tree $t = (T, \tau)$ with $T = \{0, 1\}^*$ and $\tau(x) = (a = 1)$ for all x in T and its accepting run on the Parity tree automaton are shown in Figure 2.7. The run is accepting, because the maximal color occurring infinitely often on each path of the run tree is $c(C) = 2$ which is even.

In Figure 2.8 the Moore machine corresponding to the input tree is depicted. The set of words of the Moore machine is equal to the set of words of the input tree.

Since the input tree t can be represented by a Moore machine, it is regular. Now we know that a regular tree exists, which is accepted by the DPT and consequently we know that the corresponding Moore machine



(a) Input Tree 1



(b) Run Tree on DPT for $\mathsf{X}\,a \vee \mathsf{X}\,r$

Figure 2.7: Trees

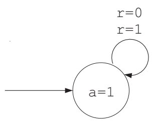


Figure 2.8: Moore Machine M_1

realizes the specification. Thus the specification is realizable and doing synthesis we found the Moore machine M_1.

Again, in our example it was easy to find the Moore machine M_1 by hand. In general, the realizing Moore machines are derived from the proof of non emptiness of the DPT. □

2.7 Mu-calculus

The μ-calculus is also a specification language. In fact it subsumes most temporal logics used for reasoning about reactive systems. Additionally, μ-calculus formulas can easily be translated into symbolic programs which are often more efficient for model checking and synthesis than programs using explicit state representation. Thus it is a very powerful and practical language. A very nice and more detailed discussion on the μ-calculus can be found in [Eme97].

Here we will define fixpoint formulas over the set of states of a DBW. Let $A = (\Sigma, Q, q_0, \delta, Acc)$ be a DBW. The *syntax of a μ-calculus formula* over a set of states Q and a set of variables V is defined as follows.

1. Every state q in Q is a μ-calculus formula,
2. every variable Y in V is a μ-calculus formula,
3. if φ is a μ-calculus formula then $\neg\varphi$ is a μ-calculus formula,
4. if φ and ψ are μ-calculus formulas then so is $\varphi \wedge \psi$,

5. if φ is a μ-calculus formula then $\mathsf{EX}\,\varphi$ is a μ-calculus formula,
6. if Y is in V and φ is a μ-calculus formula then $\mu Y\,.\,\varphi$ and $\nu Y\,.\,\varphi$ are μ-calculus formulas

If all variables are bound by a fixpoint operator the formula is *closed*. We will only consider closed fixpoint formulas here. The *semantics of a closed μ-calculus formula* over a DBW $A = (\Sigma, Q, q_0, \delta, Acc)$ are defined inductively. A formula φ is interpreted as the set of states in Q in which φ is true. We write such set of states as $[\![\varphi]\!]^e$ where $e : V \to 2^Q$ is an *environment*. The environment assigns to each variable in V a subset of Q. We denote by $e[Y \leftarrow S]$ the environment such that $e[Y \leftarrow S](Y) = S$ and $e[Y \leftarrow S](Z) = e(Z)$ for $Y \neq Z$. The set $[\![\varphi]\!]^e$ is defined as follows:

1. States:
   - $[\![q]\!]^e = \{q\}$
2. Variables:
   - $[\![Y]\!]^e = e(Y)$
3. Boolean Operators:
   - $[\![\neg\varphi]\!]^e = Q \setminus [\![\varphi]\!]^e$
   - $[\![\varphi \wedge \psi]\!]^e = [\![\varphi]\!]^e \cap [\![\psi]\!]^e$
4. Branching Time Temporal Operator:
   - $[\![\mathsf{EX}\,\varphi]\!]^e = \{q \in Q \mid \exists s \in \Sigma \text{ and } q' \in [\![\varphi]\!]^e : \delta(q,s) = q'\}$
     This denotes all states from which a state where φ is true can be reached in one step.
5. Fixpoint Operators:
   - Least Fixpoint:
     $$[\![\mu Y\,.\,\varphi]\!]^e = \bigcup_i S_i \text{ where } S_0 = \emptyset \text{ and } S_{i+1} = [\![\varphi]\!]^{e[Y \leftarrow S_i]}$$
   - Greatest Fixpoint:
     $$[\![\nu Y\,.\,\varphi]\!]^e = \bigcap_i S_i \text{ where } S_0 = Q \text{ and } S_{i+1} = [\![\varphi]\!]^{e[Y \leftarrow S_i]}$$

For closed formulas the initial environment is not relevant, we can omit the e and simply write $[\![\varphi]\!]$.

The complexity of a μ-calculus formula is measured by its *alternation depth*. Intuitively the alternation depth of a μ-calculus formula is the number of alternations of greatest and least fixpoints. For a definition of alternation depth and its relation to the complexity of evaluating a μ-calculus formula see [EL86].

Important examples of fixpoint formulas are

1. $\mu Y \,.\, \varphi \vee \mathsf{EX}\, Y$ (*reachability*) which is the same as the CTL* formula $\mathsf{E}\,\mathsf{F}\,\varphi$ denoting all states from which a state where φ is true can be reached and

2. $\nu Y \,.\, \varphi \wedge \mathsf{EX}\, Y$ (*safety*) which is the same as the CTL* formula $\mathsf{E}\,\mathsf{G}\,\varphi$ denoting all the states from which it is possible to stay in states where φ is true.

Both formulas are closed and have alternation depth one.

Another operator which we will use later on is $\mathsf{E}(\varphi \,\mathsf{U}\, \psi)$ which is an abbreviation of $\mu Y \,.(\psi \vee (\varphi \wedge \mathsf{EX}(Y))$. It denotes all states from which a path starting at this state and fulfilling $\varphi \,\mathsf{U}\, \psi$ exists. Hence all states where a path exists such that φ holds until ψ holds.

Chapter 3

Some Theory and More

In order to motivate our work we will start this chapter with an example. The example shows that given two different LTL formulas, there exists a Moore machine which realizes both specifications and there exists another Moore machine which realizes one specification but not the other. This observation led us to the following question:

> Is the set of Moore machines realizing one specification a subset of the set of Moore machines realizing another specification?

Example 8. The two specifications we will consider are $\varphi = \mathsf{X}\, a \vee \mathsf{X}\, r$ from Example 7 in Section 2.6 and $\psi = \mathsf{X}\, a \vee \mathsf{F}\, r$, with input variables $I = \{r\}$ and output variables $O = \{a\}$.

The specification ψ denotes all sequences with an a occurring in the second position or an r occurring at least once. The corresponding Büchi word automaton is shown in Figure 3.1.

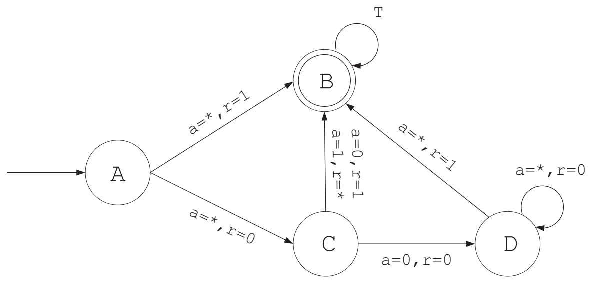


Figure 3.1: DBW for $\mathsf{X}\, a \vee \mathsf{F}\, r$

Note that the Büchi word automaton is already deterministic, thus we can skip the determinization step again and we construct the parity tree automaton A^t_ψ shown in Figure 3.2 with coloring

$$c(q) = \begin{cases} 2 & \text{for } q = B \\ 1 & \text{else} \end{cases}$$

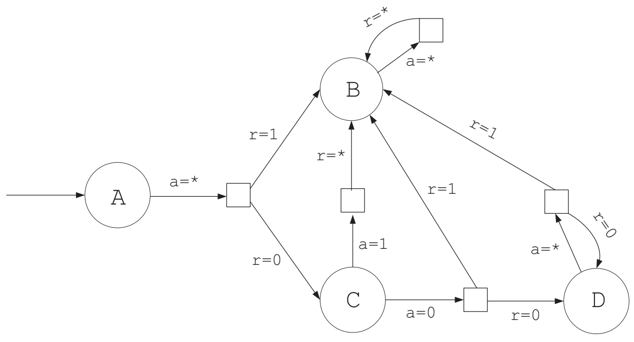
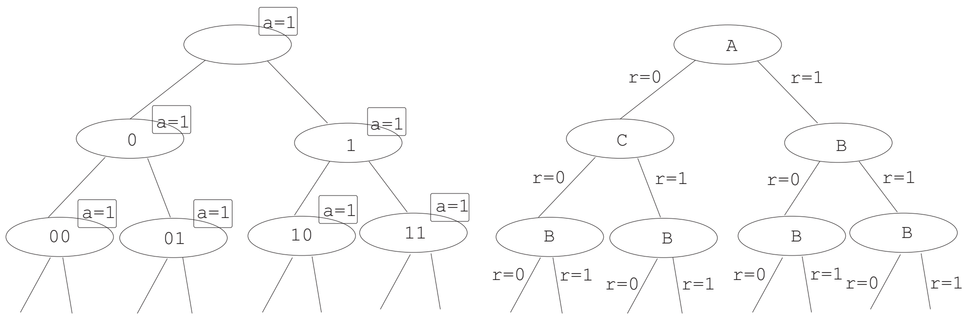


Figure 3.2: DPT for $\mathsf{X}\,a \vee \mathsf{F}\,r$

All regular trees accepted by the DPT represent Moore machines which realize the specification. In Figure 3.3 we show an accepting run on the DPT A^t_ψ for the input tree t introduced in Example 7 in Section 2.6. This shows that the Moore machine M_1 which realizes $\mathsf{X}\,a \vee \mathsf{X}\,r$ also realizes $\mathsf{X}\,a \vee \mathsf{F}\,r$. Figure 3.4 shows the Moore machine M_1 from Example 7 again. M_1 corresponds to the input tree t .



(a) Input tree 1 (b) Run tree on DPT for $\mathsf{X}\,a \vee \mathsf{F}\,r$

Figure 3.3: Trees

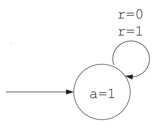
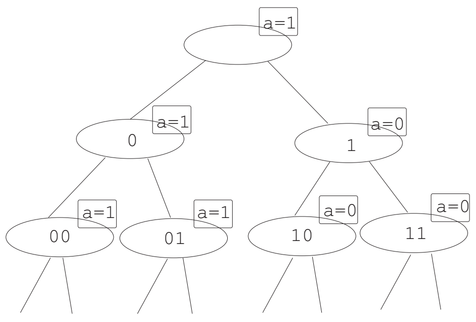
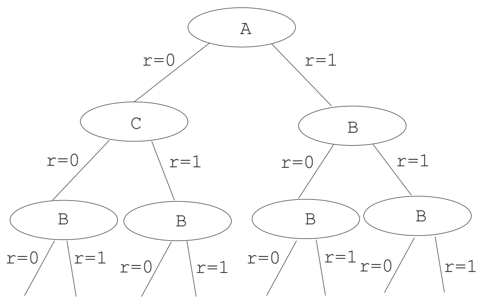


Figure 3.4: Moore Machine M_1

Now let us consider another input tree shown in Figure 3.5. This input tree also has an accepting run on the DPT A^t_ψ, and there exists a corresponding Moore machine M_2 shown in Figure 3.6, which realizes the specification.



(a) Input Tree 2



(b) Run Tree on DPT for $\mathsf{X}\, a \vee \mathsf{F}\, r$

Figure 3.5: Trees

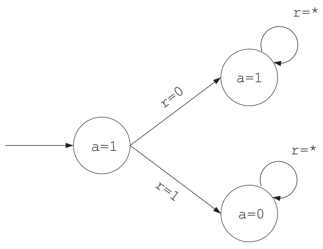


Figure 3.6: Moore Machine M_2

It can easily be seen that the Moore machine M_2 does not realize the specification $\mathsf{X}\, a \vee \mathsf{X}\, r$ discussed in Example 7 in Section 2.6 because a sequence starting with $\binom{a=1}{r=1}\binom{a=0}{r=0}$ does not satisfy $\mathsf{X}\, a \vee \mathsf{X}\, r$. Recall that all words of the Moore machine have to satisfy the specification in order to realize it.

To sum up, there exists a Moore machine M_1 which realizes both specifications and a Moore machine M_2 which realizes ψ but not φ. Consequently the set of Moore machines realizing ψ is not a subset of the set of Moore machines realizing φ because there is a Moore machine M_2 which realizes ψ but not φ. On the other hand, all Moore machines realizing φ also realize ψ because if all words of a Moore machine satisfy $\mathsf{X}\, a \vee \mathsf{X}\, r$, they also satisfy $\mathsf{X}\, a \vee \mathsf{F}\, r$.

Using the notation we will introduce later on in this chapter we can state that $\psi \not\to_o \varphi$, but $\varphi \to_o \psi$. $\square$

Finding out if a specification can be realized by the same set or a subset of open modules realizing another specification is a totally new aspect of synthesis. It is only comparable to trace inclusion, which is stronger (see below). Open implication suffices to find out if a module realizing one specification can be used to realize another specification.

In the remainder of this chapter we will give a formal definition of open implication and show some properties of the relation. We will discuss the difference between Mealy and Moore machines in the context of realizability and open implication. Then we compare open implication to the well known notion of trace inclusion. After that, our first attempts to solve the problem are shown and the time and space complexity of open implication is proved. From this prove we derive the idea for the algorithms solving open implication and we give a not quite optimal algorithm using satisfiability of CTL*. We conclude this chapter with a discussion on minimal LTL formulas.

3.1 What is Open Implication

Definition 9. Let φ and ψ be two LTL formulas over the set of input variables I and the set of output variables O and $\mathcal{M}_{I,O}$ the set of Moore machines with input variables I and output variables O. Then *open implication* is defined as follows:

$$\varphi \rightarrow_o \psi \Leftrightarrow \{M \in \mathcal{M}_{I,O} \mid M \models \varphi\} \subseteq \{M \in \mathcal{M}_{I,O} \mid M \models \psi\}.$$

Where I and O are clear from context they are omitted.

Claim 10. *Open implication is a partial order.*

Proof. The following three properties hold because the subset relation on sets is also a partial order:

- reflexivity: $\varphi \rightarrow_o \varphi$
- transitivity: if $\varphi \rightarrow_o \psi$ and $\psi \rightarrow_o \gamma$ then $\varphi \rightarrow_o \gamma$
- antisymmetry: if $\varphi \rightarrow_o \psi$ and $\psi \rightarrow_o \varphi$ then $\varphi \leftrightarrow_o \psi$

$\square$

Mealy open implication is defined alike. A specification is said to Mealy open imply another one, if the set of Mealy machines realizing the first specification is a subset of the Mealy machines realizing the second one. The difference between open implication, often also denoted as Moore open implication, and Mealy open implication will be discussed later on. Unless noted otherwise we will use Moore open implication as in Definition 9.

Using the definition of open implication in both directions we obtain a definition for open equivalence:

Definition 11. Again let φ and ψ be two LTL formulas over the set of input variables I and the set of output variables O and $\mathcal{M}_{I,O}$ the set of all Moore machines with input variables I and output variables O. Then *open equivalence* is defined as follows:

$$\varphi \leftrightarrow_o \psi \Leftrightarrow \{M \in \mathcal{M}_{I,O} \mid M \models \varphi\} = \{M \in \mathcal{M}_{I,O} \mid M \models \psi\}.$$

Claim 12. *Open equivalence is an equivalence relation.*

Proof. The following three properties hold because the equivalence of sets is also an equivalence relation:

- reflexivity: $\varphi \leftrightarrow_o \varphi$
- symmetry: $\varphi \leftrightarrow_o \psi \Rightarrow \psi \leftrightarrow_o \varphi$
- transitivity: $\varphi \leftrightarrow_o \psi$ and $\psi \leftrightarrow_o \gamma \Rightarrow \varphi \leftrightarrow_o \gamma$

□

Again Mealy open equivalence is defined in the same way. Two specifications are Mealy open equivalent if they Mealy open imply each other.

Further properties Next we show that open equivalence is invariant to extensions with "and". In contrast, "negation" and "or" have no such property. Thus open implication is not a congruence.

Claim 13.
$$\varphi \leftrightarrow_o \varphi' \Rightarrow \varphi \wedge \psi \leftrightarrow_o \varphi' \wedge \psi$$

Proof. Looking at the definition of open equivalence we have to prove:

$$\{M \in \mathcal{M} \mid M \models \varphi\} = \{M \in \mathcal{M} \mid M \models \varphi'\} \Rightarrow$$

$$\{M \in \mathcal{M} \mid M \models \varphi \wedge \psi\} = \{M \in \mathcal{M} \mid M \models \varphi' \wedge \psi\}$$

First we prove that $M \models (\varphi \wedge \psi) \Leftrightarrow (M \models \varphi) \wedge (M \models \psi)$ for an arbitrary M in $\mathcal{M}$:

$$M \models (\varphi \wedge \psi) \Leftrightarrow$$

$$\forall \text{ words } \sigma \text{ of } M : \sigma \models \varphi \wedge \sigma \models \psi \Leftrightarrow$$

$$\forall \text{ words } \sigma \text{ of } M : \sigma \models \varphi \text{ and } \forall \text{ words } \sigma \text{ of } M : \sigma \models \psi \Leftrightarrow$$

$$(M \models \varphi) \wedge (M \models \psi)$$

Now we can conclude that

$$M \models (\varphi \wedge \psi) \Leftrightarrow (M \models \varphi) \wedge (M \models \psi) \Leftrightarrow (M \models \varphi') \wedge (M \models \psi) \Leftrightarrow M \models (\varphi' \wedge \psi)$$

□

- The other direction does not hold.

  $$\varphi \wedge \psi \leftrightarrow_o \varphi' \wedge \psi \not\Rightarrow \varphi \leftrightarrow_o \varphi'$$

  A counter example is $a \wedge \mathsf{false} \leftrightarrow_o \neg a \wedge \mathsf{false}$ but $a \not\leftrightarrow_o \neg a$, where $O = \{a\}$.

- Negation has no such property.

$$\varphi \leftrightarrow_o \varphi' \not\Rightarrow \neg\varphi \leftrightarrow_o \neg\varphi'$$

  A counter example with $I = \{r\}$ and $O = \{a\}$ is $a \leftrightarrow_o a \vee r$. The two specifications are open equivalent because a Moore machine can not set the input variable r but only the output variable a. Thus in order to satisfy either specification it has to set $a = 1$ in the initial state. On the other hand $\neg a \not\leftrightarrow_o \neg a \wedge \neg r$ because $\neg a \wedge \neg r$ is not realizable but $\neg a$ is. The specification $\neg a$ can be realized by a Moore machine with $a = 0$ in the initial state, but $\neg a \wedge \neg r$ can not be realized because a Moore machine can not set the input variables and thus it can not guarantee $\neg r$.

- Or also has no such property.

$$\varphi \leftrightarrow_o \varphi' \not\Rightarrow \varphi \vee \psi \leftrightarrow_o \varphi' \vee \psi$$

  For example, $a\wedge r \leftrightarrow_o a\wedge\neg r$ but $(a\wedge r)\vee r \not\leftrightarrow_o (a\wedge\neg r)\vee r$, where $I = \{r\}$ and $O = \{a\}$. Neither specification is realizable (see arguments above) thus they are realizable equivalent because $\{M \in \mathcal{M} \mid M \models a \wedge r\} = \emptyset$ and $\{M \in \mathcal{M} \mid M \models a \wedge \neg r\} = \emptyset$. On the other hand $(a \wedge r) \vee r = r$ is not realizable but $(a \wedge \neg r) \vee r = a \vee r$ is realizable by a Moore machine with $a = 1$ in the initial state. Thus $(a \wedge r) \vee r$ and $(a \wedge \neg r) \vee r$ are not realizable equivalent.

3.2 Comparison Mealy and Moore

In Section 2.1 we explained the difference between Mealy and Moore machines. In this section we will first discuss the difference between Mealy and Moore realizability and then the difference between Mealy and Moore open equivalence.

If an LTL formula is Moore realizable then it is also Mealy realizable because the realizing Moore machine is also a Mealy machine. On the other hand, specifications realizable by a Mealy machine must not always be realizable by a Moore machine, see Example 15 below. But if the specification is modified by exchanging all occurrences of output variables o in the formula with $\mathsf{X}\, o$ then it is also Moore realizable.

Theorem 14. *Given an LTL formula φ over the set of input variables I and the set of output variables O. Let $\varphi^{[o \leftarrow \mathsf{X}\, o]}$ denote the formula φ where all occurrences of an output variable o in O is changed to $\mathsf{X}\, o$. If φ is Mealy realizable then $\varphi^{[o \leftarrow \mathsf{X}\, o]}$ is Moore realizable.*

Proof. We assume that the Mealy machine M_e realizes the given LTL formula φ. Let M_o be a Moore machine such that $M_o^{[o_i \leftarrow o_{i+1}]}$ defines the same set of words as the Mealy machine M_e. Such a Moore machine can always be found, see Claim 1 in Section 2.1. If the sequence $\sigma = (o_1, i_0), (o_2, i_1) \dots$ with $o_1, o_2 \dots$ in O and $i_0, i_1 \dots$ in I is a word of the Mealy machine M_e then $\sigma^{[o_i \leftarrow o_{i+1}]} = (o_0, i_0), (o_1, i_1) \dots$ for o_0 in O is a word of the Moore machine M_o. This follows from the definition of $M_o^{[o_i \leftarrow o_{i+1}]}$.

Because M_e realizes φ, σ satisfies φ. Looking at the definition of the semantics of LTL we know $\sigma^i \models o$ if and only if $o \in \sigma^i$ for all occurrences of an output variable o in φ. When we exchange all o with $\mathsf{X}\, o$ in φ then $\sigma^i \models \mathsf{X}\, o$ if and only if $o \in \sigma^{i+1}$. This holds for the word $\sigma^{[o_i \leftarrow o_{i+1}]}$ and consequently $\sigma^{[o_i \leftarrow o_{i+1}]}$ satisfies $\varphi^{[o \leftarrow \mathsf{X}\, o]}$.

We showed, for any word σ of the Mealy machine M_e, the corresponding word $\sigma^{[o_i \leftarrow o_{i+1}]}$ of the Moore machine M_o satisfies $\varphi^{[o \leftarrow \mathsf{X}\, o]}$. Thus M_o realizes $\varphi^{[o \leftarrow \mathsf{X}\, o]}$. □

Example 15. First we give a specification φ which is Mealy realizable but not Moore realizable, then we show that $\varphi^{[o \leftarrow \mathsf{X}\, o]}$ is Moore realizable.

Consider the specification $\varphi = (r \wedge a) \vee (\neg r \wedge \neg a)$ with input variable r and output variable a. The Mealy machine M shown in Example 2 in Section 2.1 realizes φ. The specification is not Moore realizable because a Moore machine can not react to the input immediately. It can only set either a or $\neg a$ in the beginning and thus it can never satisfy φ.

Next consider the specification $\varphi^{[o \leftarrow \mathsf{X}\, o]} = (r \wedge \mathsf{X}\, a) \vee (\neg r \wedge \mathsf{X}\, \neg a)$, in Example 3 in Section 2.2 we showed that it is Moore realizable. □

If two LTL specifications are Mealy open equivalent then they are also Moore open equivalent because every Moore machine is also a Mealy machine. Formally, if $\{M_e \in \mathcal{M}_e \mid M_e \models \varphi\}$ is equal to $\{M_e \in \mathcal{M}_e \mid M_e \models \psi\}$ where $\mathcal{M}_e$ is the set of all Mealy machines, then $\{M_e \in \mathcal{M}_e \mid M_e \models \varphi\} \cap \mathcal{M}_o$ is equal to $\{M_e \in \mathcal{M}_e \mid M_e \models \psi\} \cap \mathcal{M}_o$ where $\mathcal{M}_o$ is the set of all Moore machines which is a subset of the set of all Mealy machines $\mathcal{M}_e$.

On the other hand, if two specifications are Moore open equivalent, they are not necessarily Mealy open equivalent, see Example 18 below. But two specifications can be modified such that the original specifications are Mealy open equivalent if the modified specifications are Moore open equivalent.

Theorem 16. *If two specifications with all occurrences of output variables o replaced by next o ($\mathsf{X}\, o$) are Moore open equivalent then the original specifications are also Mealy open equivalent.*

Proof. This can be shown by contradiction. We assume that $\varphi^{[o \leftarrow \mathsf{X}\, o]}$ and $\psi^{[o \leftarrow \mathsf{X}\, o]}$ are Moore open equivalent but there exists a Mealy machine which realizes φ but not ψ. Then we can convert this Mealy machine into a Moore machine M_o such that $M_o^{[o_i \leftarrow o_{i+1}]}$ defines the same set of words as the Mealy machine. Consequently M_o realizes $\varphi^{[o \leftarrow \mathsf{X}\, o]}$ but not $\psi^{[o \leftarrow \mathsf{X}\, o]}$ (use same arguments as in proof of Theorem 14), which contradicts our assumption. □

Thus differences between open equivalence using Mealy machines and open equivalence using Moore machines may occur. In general this does not affect the theory.

Corollary 17. *The problem of calculating Mealy open equivalence can be reduced to the problem of calculating Moore open equivalence.*

Proof. Assume we can calculate Moore open equivalence. The formula φ is open Mealy equivalent to ψ if the modified formula $\varphi^{[o \leftarrow \mathsf{X}\, o]}$ is open Moore equivalent to $\psi^{[o \leftarrow \mathsf{X}\, o]}$. Additionally φ is not open Mealy equivalent to ψ if φ is also not open Moore equivalent to ψ. $\square$

Consequently we only need a decision procedure for Moore open equivalence. Though the difference between Mealy and Moore open equivalence does not affect the theory, one must be aware of the difference when applying the theory.

Example 18. First we will give two specifications $\varphi^{[o \leftarrow \mathsf{X}\, o]}$ and $\psi^{[o \leftarrow \mathsf{X}\, o]}$ which are Moore open equivalent but not Mealy open equivalent, then we show that the original specifications φ and ψ are also Mealy open equivalent.

The two specifications $\varphi^{[o \leftarrow \mathsf{X}\, o]} = \mathsf{X}\, a \vee \mathsf{X}\, r$ and $\psi^{[o \leftarrow \mathsf{X}\, o]} = \mathsf{X}\, a$ with input variable r and output variable a are Moore open equivalent, because the Moore machine can not control the input variables and it can not react to the input of the environment immediately. Consequently it has to set $a = 1$ in the second time step in order to realize the first specification and this also realizes the second specification.

Looking at the same specifications using Mealy machines, we can find a Mealy machine that realizes $\mathsf{X}\, a \vee \mathsf{X}\, r$ but not $\mathsf{X}\, a$, see Figure 3.7. The words of the Mealy machine shown in Figure 3.7 either start with $\binom{a=0}{r=*}\binom{a=1}{r=0}$ or with $\binom{a=0}{r=*}\binom{a=0}{r=1}$. The first sequence satisfies $\mathsf{X}\, a$ and the second sequence satisfies $\mathsf{X}\, r$, thus the Mealy machine realizes $\mathsf{X}\, a \vee \mathsf{X}\, r$. But the second sequence does not satisfy $\mathsf{X}\, a$ and thus the Mealy machine does not realize $\mathsf{X}\, a$.

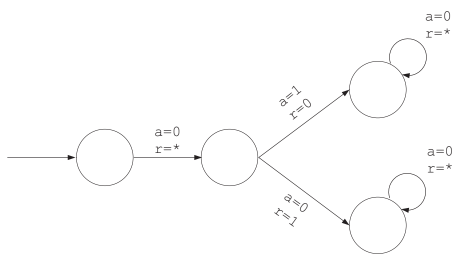


Figure 3.7: Mealy Machine

It is easy to see that the original specifications $\varphi = a \vee \mathsf{X}\, r$ and $\psi = a$ are Mealy open equivalent because even Mealy machines do not know the future and thus a Mealy machine has to set $a = 1$ in the first time step to realize φ and this also realizes ψ. □

3.3 Comparison to Trace Inclusion

Open implication can be compared to trace inclusion ($\varphi \rightarrow \psi$), which states that all sequences satisfying φ also satisfy ψ. This is the same as language inclusion on the corresponding NBWs. Trace inclusion is often used to verify if a module satisfies a specification. This form of verification is widely used in practice and thus well studied.

Trace inclusion amounts to calculating if the formula $\varphi \rightarrow \psi$ is a tautology, meaning that all sequences satisfy $\varphi \rightarrow \psi$. This can be done by deciding if the formula $\neg(\varphi \rightarrow \psi)$ is satisfiable. If it is satisfiable then $\varphi \not\rightarrow \psi$.

For LTL, satisfiability is PSPACE complete [SC85]. We will see that open implication is a lot harder to calculate, but it is weaker than trace inclusion. Consequently trace inclusion is a sufficient but not necessary condition for open implication.

Claim 19. *Given two LTL formulas φ and ψ with input variables I and output variables O, then the following implication holds:*

$$\varphi \rightarrow \psi \Rightarrow \varphi \rightarrow_o \psi$$

Proof. Set $\Sigma = 2^O$ and $D = 2^I$. If $\{\sigma \in (\Sigma \times D)^\omega \mid \sigma \models \varphi\}$ is a subset of $\{\sigma \in (\Sigma \times D)^\omega \mid \sigma \models \psi\}$ then $\{M \in \mathcal{M} \mid M \models \varphi\}$ is a subset of $\{M \in \mathcal{M} \mid M \models \psi\}$ because the set of words a Moore machine realizing φ respectively ψ defines is a subset of the words satisfying φ respectively ψ. □

Contrarily, the other direction is not true.

Claim 20. *Given two LTL formulas with input variables I and output variables O, then $\varphi \rightarrow_o \psi$ does not imply $\varphi \rightarrow \psi$.*

Proof. A counter example is $a \vee r \rightarrow_o a$ where r is an input variable and a an output variable. Since a Moore machine can not control the input variable r the two specifications are open equivalent, but $a \vee r \not\rightarrow a$ because they define different sequences. The specification $a \vee r$ is satisfied with an r and no a in the first position of the sequence, but the specification a is not. □

3.4 First Attempts

In this section we will explain our first attempts to solve open implication. To start with, we tried to use realizability but unfortunately this did not work. Then we used tree languages which led to a correct solution but a very high time complexity.

3.4.1 Using Realizability

First we tried to solve the problem using realizability:

$$(\varphi \rightarrow \psi) \text{ realizable} \overset{?}{\Rightarrow} \varphi \rightarrow_o \psi$$

The following counter example shows that the implication in question does not hold.

$$(\mathsf{X}\,a \vee \mathsf{F}\,r \rightarrow \mathsf{X}\,a) \text{ is realizable but } \mathsf{X}\,a \vee \mathsf{F}\,r \not\rightarrow_o \mathsf{X}\,a$$

with input variable r and output variable a. On the one hand, the Moore machine in Figure 2.8 from Example 7 in Section 2.6 realizes the formula $(\mathsf{X}\,a \vee \mathsf{F}\,r \rightarrow \mathsf{X}\,a)$. On the other hand, the specification $\mathsf{X}\,a \vee \mathsf{F}\,r$ does not open imply $\mathsf{X}\,a$ because the Moore machine in Figure 3.6 from Example 8 realizes $\mathsf{X}\,a \vee \mathsf{F}\,r$ but not $\mathsf{X}\,a$. Thus $\{M \in \mathcal{M} \mid M \models \mathsf{X}\,a \vee \mathsf{F}\,r\}$ is not a subset of $\{M \in \mathcal{M} \mid M \models \mathsf{X}\,a\}$.

3.4.2 Using Tree Languages

Next we took a closer look at tree languages. Every Moore Machine corresponds to a regular tree, see Section 2.5. Therefore, the obvious solution to the open implication problem is to compare tree languages. For a definition of tree languages see Section 2.6.

Lemma 21. *Let φ and ψ be two LTL formulas and L_φ and L_ψ the corresponding tree languages then*

$$L_\varphi \subseteq L_\psi \Leftrightarrow L_\varphi \setminus L_\psi = \emptyset \Leftrightarrow L_\varphi \cap L_\psi^c = \emptyset \Leftrightarrow \varphi \rightarrow_o \psi$$

Proof. Showing that $L_\varphi \subseteq L_\psi \Rightarrow \varphi \rightarrow_o \psi$ is trivial because if all trees satisfying φ also satisfy ψ we know that all regular trees satisfying φ also satisfy ψ. Thus all Moore machines realizing φ also realize ψ.

In order to prove the other direction we have to show:

> If all regular trees satisfying φ satisfy ψ then all trees satisfying φ satisfy ψ.

This is shown by contradiction. Assume there is a tree satisfying φ but not ψ, that is the same as assuming that the tree language for the CTL* formula $\mathsf{A}\,\varphi \wedge \neg\,\mathsf{A}\,\psi$ is not empty. But according to the Rabin Basis Theorem, see Section 2.6, a nonempty tree language must contain a regular tree, which contradicts our assumption that all regular trees satisfying φ satisfy ψ. □

Unfortunately we know of no efficient algorithm deciding if one tree language is a subset of another. First we would have to build the tree automata for φ and ψ which is already in 2EXP for each formula, see Section 4.1. Then we would have to calculate the complement of the tree automaton for ψ and intersect it with the tree automaton for φ. Last we would have to calculate the emptiness of the resulting tree language. So in theory we could calculate if $L_\varphi \cap L_\psi^c$ is empty but it would lead to a very high time and space complexity.

3.5 Lower Bound

The aim of this section is to find a lower bound for the complexity of deciding if φ open implies ψ.

Realizability of an LTL formula φ can be reduced to open implication as follows.

$$\varphi \text{ is not realizable} \Leftrightarrow \{M \in \mathcal{M} \mid M \models \varphi\} = \emptyset \Leftrightarrow$$

$$\{M \in \mathcal{M} \mid M \models \varphi\} \subseteq \{M \in \mathcal{M} \mid M \models \mathsf{false}\} \Leftrightarrow \varphi \rightarrow_o \mathsf{false}$$

Hence we can decide realizability of φ if we can decide open implication. Since the problem of LTL-Realizability is 2EXP-complete [Ros92] we can conclude that open implication is also 2EXP-hard.

A lower bound in the second argument ψ can be found with the following lemma.

Lemma 22.

$$\neg\psi \textit{ is not satisfiable} \Leftrightarrow \psi \textit{ is a tautology} \Leftrightarrow \mathsf{true} \rightarrow_o \psi$$

Proof. The specification $\neg\psi$ is not satisfiable if there is no sequence satisfying $\neg\psi$, which is equivalent to "all sequences satisfy ψ" and thus ψ is a tautology. If ψ is a tautology then $\psi \leftrightarrow \mathsf{true}$ and consequently $\mathsf{true} \rightarrow_o \psi$, see Claim 19 in Section 3.3.

The other direction ($\mathsf{true} \rightarrow_o \psi \Rightarrow \psi$ is a tautology) is a little harder to show. We have to prove:

> If $\{M \in \mathcal{M} \mid M \models \mathsf{true}\} \subseteq \{M \in \mathcal{M} \mid M \models \psi\}$ then all sequences satisfy ψ.

We show this by contradiction. Assume not all sequences satisfy ψ, that is the tree language for $\neg \mathsf{A}\,\psi = \mathsf{E}\,\neg\psi$ is not empty. Then, according to the Rabin basis theorem, a regular tree (Moore machine) exists with a word satisfying $\neg\psi$. This contradicts our assumption that all regular trees (Moore machines) realizing true also realize ψ. ☐

Deciding satisfiability/tautology is PSPACE-complete [SC85].

To sum up, we proved that the lower bound for the complexity to decide if φ open implies ψ is 2EXP-TIME in $|\varphi|$ and PSPACE in $|\psi|$ which is a subset of EXP-TIME.

3.6 The Idea

The main idea of the algorithms presented further on is to find a Moore machine (regular tree) where all words satisfy φ and one word satisfies $\neg\psi$. If we can find a Moore machine (regular tree) like this we know that $\varphi \not\to_o \psi$ because there exists an open module which satisfies φ but not ψ. On the other hand, if no such machine (tree) exists we know that $\varphi \to_o \psi$.

Let T be the set of all trees, by $\{t \in T \mid \exists\sigma \in t : \sigma \models \neg\psi\}$ we denote all trees with a word satisfying $\neg\psi$, thus all trees in $L_{(\mathsf{E}\,\neg\psi)}$. We claim that φ open implies ψ if and only if the tree language of φ intersected with the set of all trees with a word satisfying $\neg\psi$ is empty. Thus if and only if there exists no Moore machine realizing φ and satisfying $\neg\psi$ at the same time.

Claim 23.

$$\varphi \to_o \psi \Leftrightarrow L_\varphi \cap \{t \in T \mid \exists\sigma \in t : \sigma \models \neg\psi\} = \emptyset$$

Proof.

$$\begin{aligned}
&\varphi \to_o \psi \Leftrightarrow \\
&\{M \in \mathcal{M} \mid M \models \varphi\} \subseteq \{M \in \mathcal{M} \mid M \models \psi\} \Leftrightarrow \\
&\nexists M : M \models \varphi \textit{ and } M \not\models \psi \Leftrightarrow \\
&\nexists\, t \in L_\varphi : t \not\models \psi \Leftrightarrow \\
&L_\varphi \cap \{t \in T \mid \exists\sigma \in t : \sigma \not\models \psi\} = \emptyset \Leftrightarrow \\
&L_\varphi \cap \{t \in T \mid \exists\sigma \in t : \sigma \models \neg\psi\} = \emptyset
\end{aligned}$$

□

3.7 Algorithm using CTL*

It is possible to solve open implication deciding satisfiability of the CTL* formula $\mathsf{A}\,\varphi \wedge \mathsf{E}\,\neg\psi$ because

$$L_\varphi \cap \{t \in T \mid \exists\sigma \in t : \sigma \models \neg\psi\} = \emptyset \Leftrightarrow A\varphi \wedge E\neg\psi \;\text{ is not satisfiable}$$

We apply the algorithm from Emerson and Jutla proposed in [ES84] to decide satisfiability of CTL*. First we build a DST for $\mathsf{A}\,\varphi$. The number of states of the DST is double exponential and the number of pairs is exponential in $|\varphi|$. Next a DST for $\mathsf{E}\,\psi$ is build, here the number of states is exponential and the number of pairs is linear in $|\psi|$. Then the automata are combined using a cross product construction. The resulting Street automaton has a double exponential number of states in the size of φ and an exponential number of states in the size of ψ.

Last the emptiness of the resulting Street automaton is calculated. Jurdziński [Jur00] proposes an algorithm for the emptiness of Street automata with running time bound $O(m \cdot n^{2k}/(k/2)^k)$, where n is the number of vertices and m is the number of edges in the corresponding game graph, k is the number of pairs. Thus this algorithm is 2EXP in $|\varphi|$ and EXP in $|\psi|$ which meets the lower bound for the time complexity, but not the space complexity.

In the next chapter we will introduce an algorithm which meets the lower bound of the time and the space complexity shown in Section 3.5. It is not clear if our solution of open implication could be used to improve the algorithm calculating satisfiability of CTL*.

3.8 Minimal LTL formula

Another interesting question is that of finding the minimal LTL formula open implying a given formula. A really tough aspect of this question is to define minimal properly. If we want to find a minimal formula in the context of synthesis, we want a formula which is easier to synthesize and which open implies the original formula. If we can find such a formula we can avoid synthesizing the big formula because every open module realizing the minimal formula also realizes the original formula. Now we still have an open question: How do we know which formula is easier to synthesize? We will not give a general solution to this problem, but in Section 5.6 we show some examples.

Even if we do not have a proper definition of minimal, we can prove a lower bound for the problem of finding a minimal open equivalent LTL formula for a given formula. It is at least as hard as LTL-Realizability. We assume that the minimal open equivalent LTL formula for all LTL formulas which are not realizable is false. Consequently if we find the minimal open equivalent LTL formula we can decide LTL-Realizability. Since LTL-Realizability is 2EXP-complete the minimal open equivalent problem is also 2EXP-hard.

An upper bound for the problem can be given, if we define minimal as the shortest LTL formula. Given an LTL formula φ, the shortest LTL formula open implying φ can be found by testing all possible LTL formulas shorter than $|\varphi|$. There are at most $|\varphi|^{|\varphi|}$ such formulas and testing open implication is in 2EXP. To sum up, calculating the shortest open implying LTL formula can be done in 2EXP space and 3EXP time.

Chapter 4

The Algorithm

In this chapter we will introduce a nondeterministic algorithm to decide open implication. When the algorithm is determinized using Savitchs determinization construction it meets the lower bound shown in Section 3.5. The algorithm uses the idea shown in Section 3.6, hence we need to find out if a regular tree with all words satisfying φ and at least one word satisfying $\neg\psi$ exists. This can be decided by combining an algorithm calculating realizability for φ and one calculating satisfiability for $\neg\psi$.

Going back to the automata-theoretic solution for realizability (see Example 7 in Section 2.6) we build a DPT from the specification φ. The specification is realizable if and only if we can find a regular input tree which is accepted by the DPT. If such an input tree can be found, then there exists a Moore machine which realizes the specification.

For satisfiability we only need a word with an accepting run on the corresponding ABW. A word σ is *ultimately periodic* if it can be written in the following form $\sigma = u(v)^\omega$ where u and v are finite words.

Claim 24. *If there exists a word with an accepting run on an ABW then there also exists an ultimately periodic word with an accepting run.*

Proof. Every ABW B with n states can be transformed into an NBW C with $2^{O(n)}$ states such that B and C recognize the same language ([Var96]). Given an ABW B with a nonempty language, then we can construct an NBW C with the same nonempty language. If the language of the NBW is nonempty then there exists a word with a run such that an accepting state is visited infinitely often. Since the set of states of the NBW is finite, at least one accepting state s must be visited more than once. If the part of the word which leads from s to s again is repeated infinitely often then there exists an accepting run because s can be visited infinitely often and the word is ultimately periodic. Since there is an ultimately periodic word in the language of the NBW there must also be one in the language of the ABW. □

Consequently we only need an ultimately periodic word with an accepting run on the ABW for $\neg\psi$.

The nondeterministic algorithm introduced in this chapter works as follows. Given two LTL formulas φ and ψ with input variables I and output variables O, construct a DPT A which accepts all trees such that all words of a tree satisfy φ, construct an ABW B which accepts all words satisfying $\neg\psi$, guess an ultimately periodic word σ over $(2^O \times 2^I)$ and simultaneously

1. check if there exists a regular input tree t such that σ is a word of t and t is accepted by A

2. check if σ is accepted by B

An ABW which accepts all sequences satisfying $\neg\psi$ and a DPT which accepts all trees such that all words of the trees satisfy φ can always be found, see Section 2.6. In Section 4.4 we will explain how the ABW is built and in Section 4.1 we will show how the DPT is built. How to check (1) will be shown in Section 4.2 and how to check (2) in Section 4.5.

Lemma 25. *(1) and (2) hold for a given ultimately periodic word σ if and only if there exists a regular tree such that all words of the tree satisfy φ and one word of the tree satisfies $\neg\psi$.*

Proof. (1) holds if and only if there exists a regular tree such that all words satisfy φ and one word of the tree is σ. (2) holds if and only if σ satisfies $\neg\psi$. Thus if and only if both (1) and (2) hold there exists a regular tree such that all words satisfy φ and one word of the tree satisfies $\neg\psi$. □

Corollary 26. *If (1) and (2) hold for a given ultimately periodic word σ then $\varphi \not\to_o \psi$.*

Proof. If (1) and (2) hold for a given ultimately periodic word σ then there exists a regular tree such that all words satisfy φ and one word satisfies $\neg\psi$ (Lemma 25). If such a tree exists then $\varphi \not\to_o \psi$, see Claim 23 in Section 3.6. □

When we determinize the algorithm, we do not guess a word σ but we check all possible words.

Corollary 27. *If and only if an ultimately periodic word σ exists such that (1) and (2) hold then $\varphi \not\to_o \psi$.*

Proof. This is again true because of Lemma 25 above and Claim 23 in Section 3.6. □

In the next Section we will give a short overview on how the DPT for φ is built. Then we show how to check if (1) holds, thus we will show how to calculate if a tree exists such that one word of the tree is a given ultimately periodic word σ and the tree is accepted by a certain DPT. Subsequently we will give a short overview on how to determine the set of nonempty states Q^n. The set Q^n is defined in Section 4.2 below and will be needed to check if (1) from above holds. Following this we shortly describe how an ABW is built from an LTL formula. In Section 4.5 we show how to check if (2) holds. Thus Section 4.5 contains an algorithm to check if a given ultimately periodic word is accepted by an ABW.

The remainder of the chapter contains a detailed description of the algorithm deciding open implication, also giving the pseudocode of the non-deterministic algorithm shortly described above. Last but not least we will analyze the time and space complexity of the determinized algorithm which meets the lower bound of 2EXP in φ and PSPACE in ψ.

4.1 Building the DPT for φ

In this section we will show how a DPT can be build from an LTL formula such that the language of the DPT consists of all trees where all words satisfy the LTL formula. An example for an LTL formula and its corresponding NBW and DPT can be found in Example 7 in Section 2.6. The construction is done in tree steps. It was first proposed for synthesis by Pnueli and Rosner in [PR89]. It starts by constructing an NBW from the LTL formula φ.

Theorem 28. ([VW86, VW94]) *Given an LTL formula φ over a set of atomic propositions P, one can build an NBW $A = (\Sigma, Q, q_0, \delta, Acc)$ with $\Sigma = 2^P$ and $|Q| = 2^{O(|\varphi|)}$, such that a word is accepted by A if and only if it satisfies the formula φ.*

Next the NBW needs to be determinized. If the word automaton is not determinized, the resulting tree automaton will not be able to accept all necessary trees, see Example 29 for an explanation.

Example 29. This example shows why a word automaton needs to be determinized before converting it into a tree automaton.

Consider the specification $\varphi = (a \wedge \mathsf{X}\, b) \vee (a \wedge \mathsf{X}\, c)$ with $I = \{r\}$ and $O = \{a, b, c\}$, a corresponding NBW is shown in Figure 4.1.

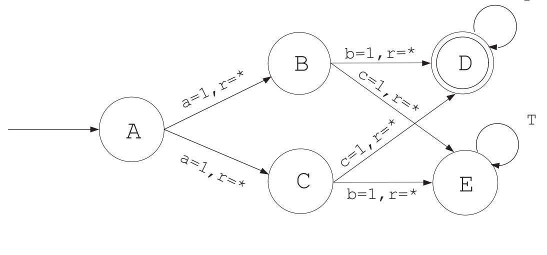


Figure 4.1: NBW for $(a \wedge \mathsf{X}\, b) \vee (a \wedge \mathsf{X}\, c)$

The matching NPT is shown in Figure 4.2 with coloring:

$$c(q) = \begin{cases} 2 & \text{for } q \in \{D\} \\ 1 & \text{else} \end{cases}$$

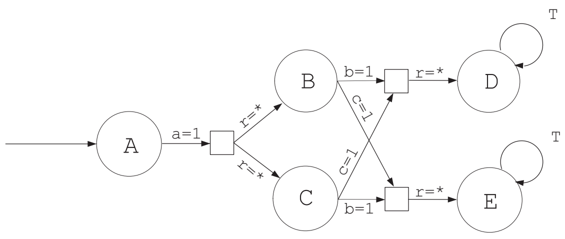


Figure 4.2: NPT for $(a \wedge \mathsf{X}\, b) \vee (a \wedge \mathsf{X}\, c)$

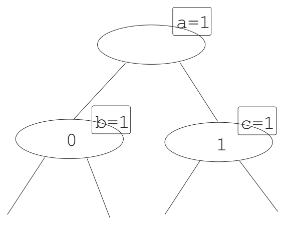


Figure 4.3: Input Tree

Although all the words of the input tree in Figure 4.3 satisfy the specification, there is no accepting run on the NPT in Figure 4.2. If the DPT decides to go into state B in the second step, it can not accept the right part of the input tree, and if it decides to go into state C, it can not accept the left part of the input tree. This does not happen if the word automaton is first determinized. □

The most well-known determinization construction is that of Safra. In 1988 Safra showed how to transform an NBW into an equivalent deterministic word automaton with either Rabin or Muller acceptance [Saf88]. Note that we used deterministic word automata with Büchi acceptance in our examples (Example 7 in Section 2.6, Example 8 in Chapter 3) but it is not always possible to convert an NBW into a DBW. DBWs are less powerful than NBWs, see Example 30.

Example 30. The nondeterministic automaton shown in Figure 4.4 can not be converted into a deterministic automaton with Büchi acceptance. The automaton accepts all words satisfying $\mathsf{F}\,\mathsf{G}(a \wedge \neg b)$ [Tho90]. □

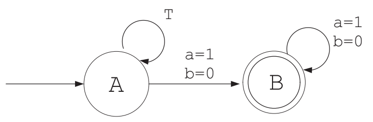


Figure 4.4: NBW with no equal DBW

In our case we will use parity acceptance, the determinization construction from NBWs to DPWs was shown by Piterman in 2006 [Pit06].

Theorem 31. ([Pit06]) *For every NBW A with n states there exists a DPW with at most n^{2n+2} states and index $2n$ that is equal to A.*

The last step from the specification to the DPT is to convert the word automaton into a tree automaton by dividing the atomic propositions into input and output variables.

Claim 32. *Given a DPW $B = (2^P, Q, q_0, \delta, c)$ and a partition of the atomic propositions P into input and output variables ($P = I \cup O$), the DPT $A = (2^O, 2^I, Q, q_0, \delta, c)$ accepts a tree if and only if all words starting at the root of the tree are also accepted by B.*

Proof. The transition function for the DPT is the same as the transition function for the DPW, because for deterministic automata δ is of the form $\delta : Q \times \Sigma \times D \to Q$ and $Q \times 2^P$ is equal to $Q \times 2^O \times 2^I$.

The DPT A accepts all trees where all words starting at the root of the tree are accepted by B because both automata have the same acceptance condition and a DPT accepts only those trees where all words satisfy the acceptance condition. □

With the help of Claim 32 we can construct a DPT accepting all trees where all words satisfy φ from a DPW which accepts all words satisfying φ.

Putting it all together, we build an NBW which accepts all words satisfying φ. Then we determinize the NBW and obtain a DPW which also accepts all words satisfying φ. Finally, we convert this DPW into a DPT which accepts all trees with all words satisfying φ. The number of states of the DPT is double exponential in φ and the size of the index is exponential in φ.

4.2 Acceptance of the DPT

This section explains how to check if there exists an input tree t such that a word of the tree is a given ultimately periodic word σ and t is accepted by a given DPT.

We start by defining the *run of a DPT on an infinite word.* Given a word $\sigma = (o_0, d_0)(o_1, d_1)\dots$ in $(\Sigma \times D)^\omega$ and a DPT $A = (\Sigma, D, Q_A, q_{A,0}, \delta_A, c)$. The run $\rho = q_0q_1q_2\dots$ of A on σ is defined as follows:

1. The run ρ starts with the initial state $q_{A,0}$.

2. Let q_i be the i-th state in the run, then q_{i+1} is the next state in the run if $\delta_A(q_i, o_i, d_i) = q_{i+1}$.

The word σ is accepted by the DPT A if and only if the highest color of a state occurring infinitely often in the run $\rho = q_0q_1q_2\dots$ is even. Note that there is only one possible run ρ on σ because the transition function of the DPT is deterministic.

We need one more definition. The *nonempty states set* Q^n of a DPT is the set of states such that starting at a state in Q^n there exists an input tree with an accepting run. We also say the states in the set Q^n have a nonempty language. If a state of the DPT is not in Q^n then there is no input tree with an accepting run. Thus all the states occurring in an accepting run tree of an input tree t are in Q^n because there exists an input tree with an accepting run tree for every such state. To find such an input tree for any state in the accepting run of the input tree t, look at the subtrees of t.

Claim 33. *Let $A = (\Sigma, D, Q_A, q_{A,0}, \delta_A, c)$ be a DPT and Q^n_A its set of nonempty states. Given an ultimately periodic word $\sigma = (o_0, d_0)(o_1, d_1)\dots$ over $(\Sigma \times D)$, there exists a regular input tree t such that σ is a word of t and t is accepted by A if and only if*

1. *the run $\rho = q_0q_1q_2\dots$ of A on σ is accepting and*
2. *for all i and for all d in D the next states $\delta_A(q_i, o_i, d)$ of the run ρ are in Q^n_A.*

Proof. We will show the two directions (if, only if) separately. We start with the easy one, the only if direction. Assume t is a regular tree such that a given ultimately periodic word σ is a word of t and t is accepted by A. Because t is accepted by A and σ is a word of t the run ρ of A on σ is accepting. Furthermore, all the states occurring in an accepting run tree are in Q^n_A because if they were not in Q^n_A the run tree could not be accepting. Consequently (2) also holds.

On the other hand, we want to show: If σ is accepted by A and all the next states of the states occurring in the run of σ are nonempty then there exists an accepting input tree with the word σ.

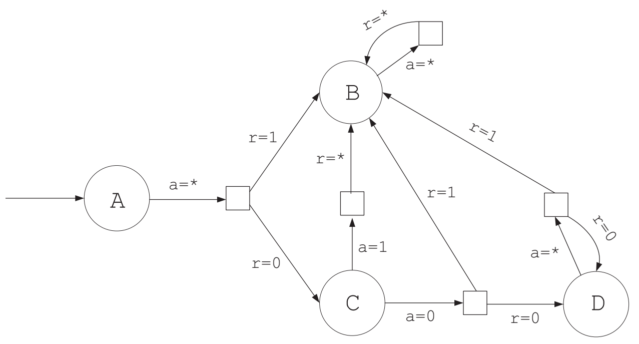
Starting at q_0 we know that all the next states for o_0 are in Q^n_A. First we will consider the next states which are not the next state in the accepting run ρ of σ. For those states we only need an arbitrary accepting input tree, such a tree exists because the states are in Q^n_A. For the next state in the accepting sequence ρ, we apply the same argument as for q_0 recursively. Thus we showed that there exists an input tree where all words except of σ are accepted and because of (1) also σ is accepted. Consequently there exists an input tree consisting of accepting words only and one of those words is σ. Clearly there also exists a regular tree because the word σ is ultimately periodic and we can always choose the same subtrees for a repeating sequence in the word. □

Next we will give a short example to illustrate how the proof works.

Example 34. Given the DPT A^t_ψ from Example 8 in Chapter 3, shown in Figure 4.5 again, with $Q^n_A = \{A, B, C\}$ and the word $\sigma = \binom{a=1}{r=1}\binom{a=0}{r=0}\dots$ then

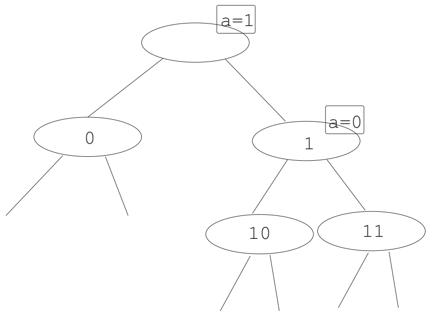
1. there exists an accepting run $\rho = ABB\dots$ of A^t_ψ on σ and
2. all the next states $\delta(A, a = 1, r = 0) = C$, $\delta(A, a = 1, r = 1) = B$ and $\delta(B, a = *, r = *) = B$ are in Q^n_A.

The accepting run of the DPT on the word σ can be seen in Figure 4.6 in the run tree, if you follow the arrows. The next states of the run ρ which are not on the run itself are C and B marked with a star in Figure 4.6. Since both C and B are in Q^n_A there exists an accepting input tree. We do not need to explicitly know which input tree, because we are interested in any accepting tree with the word σ.

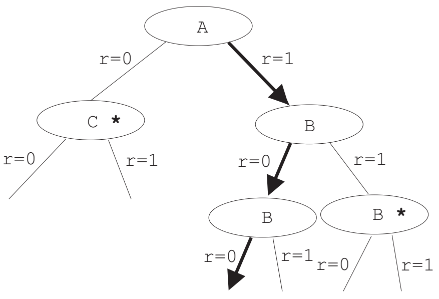


state	color
A	1
B	2
C	1
D	1

Figure 4.5: DPT for $\mathsf{X}\, a \vee \mathsf{F}\, r$



(a) Input tree with σ



(b) Run tree

Figure 4.6: Trees

Looking at the DPT, you can see that once in B you always stay there, so all the next states in ρ are B and also all the next states of ρ not in ρ (next states for other input directions) are B. Thus σ has an accepting run because $c(B) = 2$ and all the next states of ρ are in Q^n_A, which assures that there exist accepting input trees for all the next states of ρ not in ρ. Consequently there exists a tree with the infinite word $\sigma = \binom{a=1}{r=1}\binom{a=0}{r=0}\dots$ which is accepted by A^t_ψ. □

To sum up, given a DPT A and an ultimately periodic word σ as in Claim 33 above, we first calculate the nonempty states set Q^n_A. Then we check if the run of A on σ is accepting. If the run is accepting we check if all the next states for any input are in Q^n_A for all states of the accepting run. If and only if both checks hold there exists a regular tree t such that σ is a word of the tree and t is accepted by the DPT.

When we implement Claim 33 we will always do the two checks on the fly; that is, we do both checks simultaneously while reading σ.

4.3 Calculating the nonempty states set Q^n

In order to calculate the nonempty states set Q^n of a DPT, we exploit the close relationship to parity games. Every DPT can be converted into a parity game, such that the language of a DPT is not empty, if and only if the system (which controls the output variables) is the winner of the corresponding parity game. If the system is the winner of the parity game, a *winning strategy* for the system exists, meaning that no matter what the environment (controlling the input variables) does, the system can win the game, following the winning strategy. A winning strategy corresponds to a tree which is accepted by the DPT. Thus language emptiness of a DPT can be calculated via parity games. We will not go into more detail about games and only shortly explain how we will indirectly use them in our algorithm.

We want to calculate the set of states of the DPT with a nonempty language which corresponds to the set of states in the parity game which are winning for the system. This can be accomplished by the algorithm presented in [Jur00] by Jurdziński which calculates the winner in parity games. While calculating the winner, all the winning states of the game are marked and we can use this information to mark all the nonempty states of our DPT.

Running Jurdzińskis algorithm costs $O(dn)$ space and $O(dm\cdot(\frac{n}{\lfloor d/2 \rfloor})^{\lfloor d/2 \rfloor})$ time, where n is the number of vertices, m the number of edges and d the index in the parity game. As shown in Section 4.1 the number of vertices is $n = 2^{2^{O(|\varphi|)}}$ and the index is $d = 2^{O(|\varphi|)}$ in the DPT for φ. The corresponding parity game has the same number of vertices and edges and the same index. Hence it takes double exponential time and space in the size of φ to mark the states of the DPT with nonempty language.

4.4 Building the ABW for $\neg\psi$

In [Var96] Vardi shows that ABWs and NBWs have the same expressive power. But when we construct an automaton from an LTL formula the resulting ABW is a lot smaller than the corresponding NBW. Therefore we use an ABW.

Theorem 35. ([Var96]) *Given an LTL formula φ over a set of atomic propositions P, one can build an ABW $B = (\Sigma, Q, q_0, \delta, Acc)$, where $\Sigma = 2^P$ and $|Q| = O(|\varphi|)$, such that B accepts a word if and only if the word satisfies φ.*

Example 36. In Figure 4.8 the alternating automaton for the specification $\neg\varphi = \mathsf{X}\neg a \wedge \mathsf{X}\neg r$, which is the negation of the specification $\varphi = \mathsf{X}\,a \vee \mathsf{X}\,r$ from Example 7 in Section 2.6 is shown.

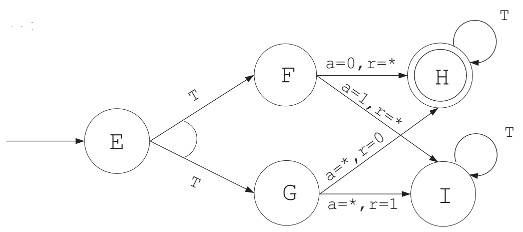


Figure 4.7: ABW for $\neg\varphi = \mathsf{X}\neg a \wedge \mathsf{X}\neg r$

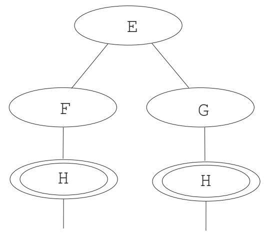


Figure 4.8: Run Tree

When an ABW is built from an LTL specification, every state of the automaton represents a sub-formula of the specification (the language the state accepts). The initial state always represents the whole formula, in our example $\neg\varphi = \mathsf{X}\neg a \wedge \mathsf{X}\neg r$. Additionally, F stands for $\neg a$, G for $\neg r$, H for true and I for false. The arc connecting the two transitions from the initial state in the automaton marks the alternating transition, it represents the $\wedge$ in the specification. This means that, on any input, the automaton goes both into state F and G in the first step. Consequently, the run of the automaton on a word is a tree. The tree in Figure 36 is the run tree of the automaton on the word $\sigma = \binom{a=1}{r=1}\binom{a=0}{r=0}\ldots$. Both branches of the run tree continue with accepting states H infinitely, thus it is an accepting run and the word σ satisfies $\neg\psi$. $\square$

4.5 Acceptance of the ABW

In this section we explain how to calculate if a given ultimately periodic word is accepted by an ABW. The algorithm is nondeterministic because in general an ABW has many different run trees for a single input word and we guess one possible run tree and check if it is accepting.

The algorithm works as follows. Given an ABW $B = (\Sigma \cup D, Q, q_0, \delta, Acc)$ and an ultimately periodic word $\sigma = u(v)^\omega$ over $(\Sigma \times D)$. The idea of the algorithm is to guess a run tree for the finite sequence uv such that there is an accepting state on every word of the run tree corresponding to the v part of the sequence. If this part of the run tree can be repeated infinitely often then we have an accepting run. Since the representation of an ultimately periodic word is not unique we also have to nondeterministically guess a representation u' and v' such that $\sigma = u(v)^\omega = u'(v')^\omega$. The algorithm will be described in more detail below and the corresponding pseudo code is depicted in Algorithm 1. A proof of its correctness is shown in Claim 37.

Start by guessing two arbitrary finite sequences u' and v' such that the infinite word is $\sigma = u(v)^\omega = u'(v')^\omega$. Next guess a run tree of u', where $L_{u'}$ denotes the set of current states starting with the initial state q_0 and following the transition function while reading the input u'. The last set of current states (the set of current states after reading the last letter of u') is stored in $L_{u'}$.

In the second part of the algorithm we look at the part of the run tree of σ which corresponds to v'. We also guess this part of the run tree starting with the set of current states $L_{v'} = L_{u'}$ and following the transition function while reading the input v'. Additionally we keep track of the words on this part of the run tree with no accepting state. This is done with the set R which is updated after every step. If R is empty after reading all of v', then there is an accepting state on every word of the part of the run tree corresponding to v'. If R is empty and $L_{v'} = L_{u'}$ again after reading v' then there exists an accepting run tree for σ.

Algorithm 1 nondet. check if the ABW $B = (\Sigma \cup D, Q, q_0, \delta, Acc)$ accepts the ultimately periodic word $\sigma = u(v)^\omega$

```
guess $u', v'$ in $(\Sigma \times D)^*$ such that $u'(v')^\omega = \sigma$
$L_{u'} = q_0$;
while $u' \neq \epsilon$ do
    guess $X \subseteq Q$ such that $X \models \bigwedge_{l \in L_{u'}} \delta(l, u'_0)$
    $L_{u'} = X$; $u' = u'_1 u'_2 \ldots$;
end while
$L_{v'} = L_{u'}$; $R = L_{u'} \setminus Acc$;
while $v' \neq \epsilon$ do
    guess $X, Y \subseteq Q$ such that $Y \models \bigwedge_{r \in R} \delta(r, v'_0) \subseteq X \models \bigwedge_{l \in L_{v'}} \delta(l, v'_0)$
    $L_{v'} = X$; $R = Y \setminus Acc$; $v' = v'_1 v'_2 \ldots$;
end while
if $(L_{v'} = L_{u'}) \wedge (R = \emptyset)$ then return true
end if
```

The idea for this algorithm stems from a proof shown by Miyano and Hayashi in [MH84]. The proof shows that ABWs and NBWs define the same

ω-languages by giving a similar subset construction as shown in Algorithm 1 above to convert any ABW into an equivalent NBW.

Claim 37. *Let $B = (\Sigma \cup D, Q, q_0, \delta, Acc)$ be an ABW and $\sigma = u'(v')^\omega$ an ultimately periodic word where u' and v' are arbitrary finite words over $(\Sigma \times D)$. The word σ is accepted by B if there exists a run tree for the finite word $u'v'$ such that*

1. *the set of states $L_{u'}$ which are reached after reading u' are reached again after reading v'*
2. *an accepting state occurs in every word of the part of the run tree corresponding to v'*

Proof. Assume $\sigma = u'(v')^\omega$ is given and there exists a run tree for the finite word $u'v'$ such that (1) and (2) hold. A run tree of an ABW is accepting if a state in *Acc* occurs infinitely often in every word of the run tree. This holds because if (1) holds we can reach the same states again and again reading v' and because if (2) holds we can do it in such a way that a state in *Acc* occurs on every word of the corresponding part of the run tree. □

Thus we showed that if our algorithm returns `true`, a fitting u' and v' was guessed and the word σ is accepted by the ABW. We also want to show that if the given word σ is accepted by the ABW then it is possible to guess "right" and prove the acceptance with the algorithm shown above.

Claim 38. *Every ultimately periodic word $\sigma = u(v)^\omega$ accepted by an ABW $B = (\Sigma \cup D, Q, q_0, \delta, Acc)$ can be brought into the form $\sigma = u'(v')^\omega$ such that there exists a run tree for the finite word $u'v'$ such that (1) and (2) from Claim 37 above hold.*

Proof. Assume σ is an ultimately periodic word accepted by an ABW then there exists a run tree such that a state in *Acc* occurs infinitely often in every word of the run tree. In Figure 4.9 such a run tree is shown. A set of states L_v^{Acc} denotes the set of states where an accepting state occurred on every word of the run tree since the last set of of states L_u^{Acc} or L_v^{Acc}. In terms of Algorithm 1, its the set of states where R is the empty set.

We will only look at the part of the run tree corresponding to $(v)^\omega$ because the finite part corresponding to u does not contribute to the acceptance. Meaning that no accepting state can occur infinitely often in the finite first part of the run tree.

Looking at the part of the run tree corresponding to $(v)^\omega$ we know that sets of states L_v^{Acc} will occur infinitely often on the run tree because the run tree is accepting. There are at most $n = 2^{|Q|}$ different sets of states L_v^{Acc}. The given sequence v is also finite, $|v| = m$. Thus after at most $n \cdot m$ occurrences of sets of states L_v^{Acc} a combination of a set of states L_v^{Acc} and a

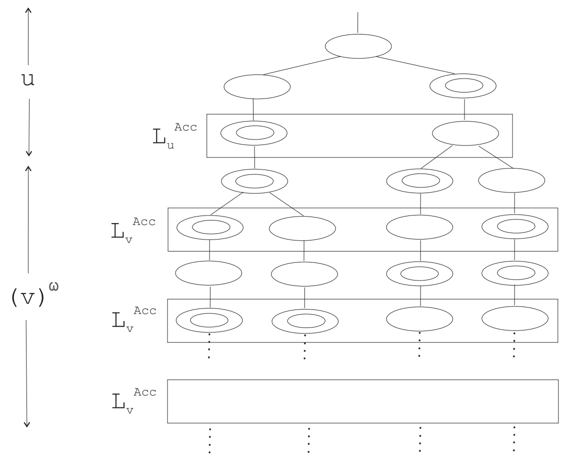


Figure 4.9: Accepting Run Tree

finite subsequence v' of v^ω must occur again. More precisely, there must be a set of states $L_{v'}^{Acc}$ and a subsequence v' of $(v)^*$ such that the set of states $L_{v'}^{Acc}$ is reached again after reading v' and the next sequence of σ is again v'. If we set u' to be the part of σ before reaching $L_{v'}^{Acc}$ for the first time in the run tree and v' as described above, then (1) and (2) from Claim 37 above hold. □

Consequently, if we determinize the algorithm we can calculate if a given ultimately periodic word σ is accepted by an ABW. In the next section we will show how the length of u' and v' can be limited to a certain size.

4.6 Deciding Open Implication

Finally we introduce the nondeterministic algorithm deciding open implication. Given two LTL specifications φ and ψ with input alphabet D and output alphabet Σ. Try to find a Moore machine realizing φ but not ψ by following the steps described below.

1. Construct a DPT $A = (\Sigma, D, Q_A, q_{A,0}, \delta_A, c)$ defining the tree language L_φ as shown in Section 4.1.

2. Calculate the set of nonempty states Q_A^n of the DPT A as shown in Section 4.3.

3. Construct an ABW $B = (\Sigma \cup D, Q_B, q_{B,0}, \delta_B, Acc)$ defining all the words satisfying $\neg\psi$ as shown in Section 4.4.

4. Call the function NDExistsWord(A, Q_A^n, B) shown in Algorithm 2.

Algorithm 2

```
 1: function NDExistsWord($A$,$Q^n_A$,$B$): $\mathbb{B}$
 2:     $(q_u, L_u), (q_v, L_v) = (0, \emptyset)$; // states in $Q_A \times 2^{Q_B}$
 3:     $p, b = 0$; // Boolean variables for parity and Büchi acceptance
 4:     $((q_u, L_u), p, b) =$ NDCheckRun$((q_{A,0}, \{q_{B,0}\}), |Q^n_A| \cdot 2^{|Q_B|}, Q^n_A)$;
 5:     if $(q_u, L_u) = (0, \emptyset)$ then return false;
 6:     end if
 7:     $((q_v, L_v), p, b) =$ NDCheckRun$((q_u, L_u), 3|Q^n_A| \cdot 2^{|Q_B|}, Q^n_A)$;
 8:     return $(((q_v, L_v) = (q_u, L_u)) \wedge p \wedge b)$;
 9: end function

    // $(q_0, L_0)$: starting states
    // $k \in \mathbb{N}, k > 0$: maximal length of run
    // $Q^n_A$: set of states of DPT $A$ with nonempty language
10: function NDCheckRun($(q_0, L_0), k, Q^n_A$): $Q_A \times 2^{Q_B} \times \mathbb{B} \times \mathbb{B}$
11:     $q = q_0$; // current state of $A$
12:     $L = L_0$; // current states of $B$
13:     $R = L_0 \setminus Acc$;
14:     $greatestc = c(q)$;
15:     if $q_0 \notin Q^n_A$ then return $((0, \emptyset)$, false, false$)$;
16:     end if
17:     for $(j = 0; j < k; j{+}{+})$ do
18:         guess $o \in \Sigma$;
19:         if $\exists i \in D : \delta_A(q, o, i) \notin Q^n_A$ then return $((0, \emptyset)$, false, false$)$;
20:         end if
21:         guess $i \in D$;
22:         $q = \delta_A(q, o, i)$;
23:         $greatestc = \max(greatestc, c(q))$;
24:         guess $X, Y$ such that
25:         * $X \models \bigwedge_{l \in L} \delta_B(l, o \cup i)$ // possible next states for $L$
26:         * $Y \models \bigwedge_{r \in R} \delta_B(r, o \cup i)$ // possible next states for $R$
27:         * $Y \subseteq X$
28:         $L = X$; $R = Y \setminus Acc$;
29:         guess $end$;
30:         if $end$ then
31:             return $((q, L)$, $greatestc$ is even, $R = \emptyset)$;
32:         end if
33:     end for
34:     return $((0, \emptyset)$, false, false$)$;
35: end function
```

Claim 39. *The function* NDExistsWord *guesses an arbitrary ultimately periodic word $\sigma = u(v)^\omega$ such that u is not longer than $|Q_A^n| \cdot 2^{|Q_B|}$ and v is not longer than $3|Q_A^n| \cdot 2^{|Q_B|}$ and simultaneously*

1. *checks if there exists a regular input tree t such that σ is a word of t and t is accepted by A*
2. *nondeterministically checks if σ is accepted by B*

Proof. In Line 4 the function NDCheckRun is called to guess the first part u of σ. Starting at the initial states NDCheckRun calculates the run of A on u and checks if all states of the run and all the next states for any input are in Q_A^n. It also calculates a possible run tree of B on u and returns the current state q_u of the automaton A and the current states L_u of the automaton B after reading u.

In Line 7 the function NDCheckRun is called again to guess the second part v of σ. Starting at the current states (q_u, L_u) of the automata after reading u the function NDCheckRun calculates the run of A on v and checks if all the states of the run and all the next states for any input are in Q_A^n. Additionally, the variable *greatestc* stores the greatest color of a state on the run. Again NDCheckRun also calculates a possible run tree of B on u and keeps track of the words of this part of the run tree with no accepting states. The function returns the current states of the automata after reading v and it returns true twice if the highest color in the run of A on v starting at the state q_u is even and if there is an accepting state in every word of the guessed run of B on v starting at the states L_u.

Finally, NDExistsWord returns true if the current states of the automata after reading v are (q_u, L_u) again and if the call of the function NDCheckRun in Line 7 returned true twice. Remembering Claim 33 in Section 4.2 and Algorithm 1 in Section 4.5 it is easy to see that the function NDExistsWord checks both (1) and (2). □

Lemma 40. *If our Algorithm returns* true *then $\varphi \not\to_o \psi$.*

Proof. If and only if our algorithm returns true it guessed "right" and verified that an ultimately periodic word exists which satisfies (1) and (2) of Corollary 26 from the beginning of this chapter. Thus according to Corollary 26, φ does not open imply ψ if our algorithm returns true. □

Example 41. Take the specifications $\psi = \mathsf{X}\,a \vee \mathsf{F}\,r$ and $\varphi = \mathsf{X}\,a \vee \mathsf{X}\,r$ from Example 8 in Chapter 3 where we already showed that $\psi \not\to_o \varphi$. The DPT for ψ and its accepting set of states Q_A^n can be found in Example 34 in Section 4.2. In the same example we showed that an accepting run for the word $\sigma = \binom{a=1}{r=1}\binom{a=0}{r=0}\ldots$ exists, where all the next states are in Q_A^n. The ABW for $\neg\varphi$ and its accepting run tree for σ is depicted in Example

36 in Section 4.4. This example shows how the nondeterministic function NDExistsWord verifies that $\psi \not\to_o \varphi$.

Assume NDExistsWord guesses $\sigma = u(v)^\omega = \binom{a=1}{r=1}\binom{a=0}{r=0}(\binom{a=0}{r=0})^\omega$ and the run tree shown in Example 36. The states after reading $u = \binom{a=1}{r=1}\binom{a=0}{r=0}$ are $(q_u, L_u) = (B, \{H\})$. Next we call NDCheckRun with starting states $(B, \{H\})$. Since we assume that the guessed v is of length one, we only need to go trough the loop once. All the next states in the DPT are in Q_A^n because the next state of B is always B independent of the values of the variables. The color of B is 2, thus *greatestc* is even. The next state of the run of the ABW on v starting at the state H is again H which is accepting. Thus R is the empty set and NDCheckRun returns $((B, \{H\}), \mathsf{true}, \mathsf{true})$. As (q_u, L_u) is also $(B, \{H\})$ the return value of the function NDExistsWord is true which proves that $\psi \not\to_o \varphi$. □

Next we argue why it suffices to check words $\sigma = u(v)^\omega$ where u is not longer than $|Q_A^n| \cdot 2^{|Q_B|}$ and v is not longer than $3|Q_A^n| \cdot 2^{|Q_B|}$.

Claim 42. *Given a DPT $A = (\Sigma, D, Q_A, q_{A,0}, \delta_A, c)$, its set of nonempty states Q_A^n and an ABW $B = (\Sigma \cup D, Q_B, q_{B,0}, \delta_B, Acc)$. If* NDCheckRun *can find a run for a word of arbitrary length from the initial states $(q_{A,0}, q_{B,0})$ to states (q_u, L_u) in $(Q_A^n \times 2^{Q_B})$ such that all states of the run of the DPT are in Q_A^n then it can also find such a run for a word not longer than $|Q_A^n| \cdot 2^{|Q_B|}$.*

Proof. $|Q_A^n| \cdot 2^{|Q_B|}$ is the number of different combinations of a state in Q_A^n and an arbitrary set of different states in Q_B. Assume NDCheckRun finds a run for a word longer than $|Q_A^n| \cdot 2^{|Q_B|}$. Then a combination of a state q_i in Q_A^n and states L_i in Q_B must occur twice on the run and thus it is possible to take a short cut and find a shorter word with a shorter run. See Figure 4.10 for an illustration of the idea.

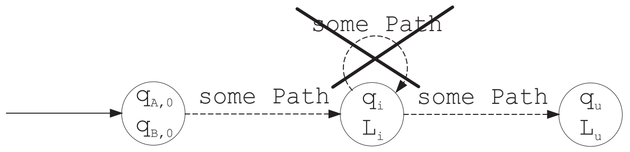


Figure 4.10: run with unnecessary loop

□

Claim 43. *Given a DPT $A = (\Sigma, D, Q_A, q_{A,0}, \delta_A, c)$, its set of nonempty states Q_A^n and an ABW $B = (\Sigma \cup D, Q_B, q_{B,0}, \delta_B, Acc)$. If* NDCheckRun *can find a run for a word of arbitrary length from the states (q_u, L_u) in $(Q_A^n \times 2^{Q_B})$ to the states (q_u, L_u) again such that the run satisfies all three items listed below, then it can find such a run for a word not longer than $3|Q_A^n| \cdot 2^{|Q_B|}$.*

1. *All the states of the run of the DPT are in Q_A^n.*

2. *The highest color of the run on the DPT is even.*

3. *There is a state in Acc in all words of the run of the ABW.*

Proof. Assume NDCheckRun finds a run for a word longer than $3|Q_A^n| \cdot 2^{|Q_B|}$ such that (1), (2) and (3) are satisfied. Then a combination of a state q_i in Q_A and states L_i in Q_B must be visited four times. We can divide the run into 3 parts. Without loss of generality we assume that we have to reach a combination of a state in Q_A^n and states in Q_B such that the run satisfies the Büchi condition of the ABW first. Then we have to reach a combination such that the run satisfies the parity condition and the third part of the run has to lead back to (q_u, L_u). Thus if a combination is visited four times it is again possible to take a short cut and find a shorter word with a shorter run. See Figure 4.11 for an illustration of the idea.

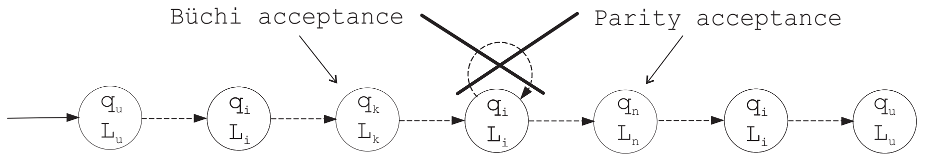


Figure 4.11: accepting loop with unnecessary loop

□

Example 44. In Figure 4.12 we give an example of a state space $(Q_A^n \times 2^{Q_B})$ and its transitions such that a run satisfying (1), (2) and (3) from Claim 43 must be almost $3|Q_A^n| \cdot 2^{|Q_B|}$ steps long. Thus we show that $3|Q_A^n| \cdot 2^{|Q_B|}$ is more or less a tight bound.

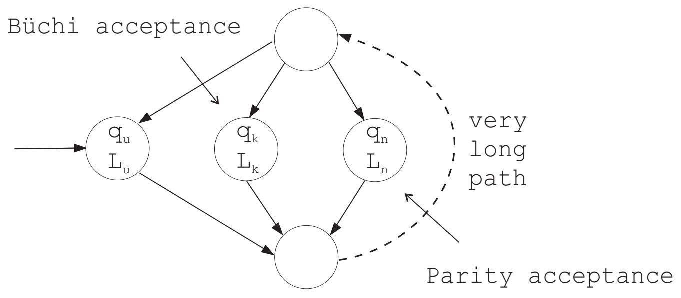


Figure 4.12: state space $Q_A^n \times 2^{Q_B}$ with very long run

Assume that the very long path, which is denoted by the dashed arrow in Figure 4.12 consists of all the combinations of states in $(Q_A^n \times 2^{Q_B})$ other than $(q_u, L_u), (q_k, L_k), (q_n, L_n)$. Then a run satisfying the parity acceptance condition and the Büchi accepting condition, starting and ending at (q_u, L_u) consists of $3|Q_A^n| \cdot 2^{|Q_B|} - 5$ states. □

Lemma 45. *If $\varphi \not\to_o \psi$ then our algorithm returns* `true` *for at least one possible set of guesses in* NDExistsWord.

Proof. Assume $\varphi \not\to_o \psi$ then, as shown in Corollary 27, there exists an ultimately periodic word σ such that

1. there exists a regular input tree t such that σ is a word of t and t is accepted by the DPT A for φ and
2. σ is accepted by the ABW B for $\neg\psi$.

If $\sigma = u(v)^\omega$ is such that u is not longer than $|Q_A^n| \cdot 2^{|Q_B|}$ and v is not longer than $3|Q_A^n| \cdot 2^{|Q_B|}$ then our algorithm returns `true` if it guesses the word, see Claim 39.

On the other hand, if $\sigma = u(v)^\omega$ such that u is longer than $|Q_A^n| \cdot 2^{|Q_B|}$ or v is longer than $3|Q_A^n| \cdot 2^{|Q_B|}$ then our function NDCheckRun can guess a word $\sigma' = u'(v')^\omega$ such that u' is not longer than $|Q_A^n| \cdot 2^{|Q_B|}$ and v' is not longer than $3|Q_A^n| \cdot 2^{|Q_B|}$ which also satisfies (1) and (2) and thus it also returns `true` in this case. $\square$

Summarizing Lemma 40 and Lemma 45, if we determinize the algorithm it returns `true` if and only if $\varphi \not\to_o \psi$.

The brute force method for determinization would be to try all possible guesses, but then we would have to store the guesses we tried already, which would be very space intensive. Savitch proposed in [Sav70] a nice algorithm for determinization, which only squares the space needed. Thus there is a deterministic algorithm deciding open implication which uses only quadratic as much space than our nondeterministic algorithm.

4.7 Time and Space Complexity

First we will analyze the complexity of the algorithm with respect to the first argument φ. To construct the DPT for φ and to mark the states of the DPT with nonempty language using Jurdzińskis algorithm we need double exponential time and space in the size of φ. See Section 4.1 and 4.3.

In the nondeterministic function NDExistsWord shown in Algorithm 2, we have to store two states of the DPT, q_u and q_v. We need $2^{O(|\varphi|)}$ bits to store these states and the remainder of the algorithm does not need any notable space with respect to φ. In [Sav70] Savitch showed that NSPACE($f(n)$) equals DSPACE($f(n)^2$). Thus when the algorithm is determinized, we need $2^{O(|\varphi|)^2}$ space which is still in EXPSPACE. Since EXPSPACE is a subset of 2EXPTIME, the determinized Algorithm 2 does not exceed the time and space complexity of Jurdzińskis algorithm. Using big-O notation everything adds up to the complexity classes 2EXPSPACE and 2EXPTIME in φ.

Second we will look at the complexity of the algorithm with regard to the second argument ψ. As shown in Section 4.4, the construction of the ABW for $\neg\psi$ can be done in linear time and space in the size of the formula. In Algorithm 2 we have to store two sets of states of the ABW, L_u and L_v. The ABW has $O(|\psi|)$ states, thus there are $2^{|\psi|}$ different set of states. In order to store the two sets we need $O(|\psi|)$ bits. All the subsequent calculations in Algorithm 2 do not take any notable space. Applying Savitchs determinization algorithm, we get a deterministic algorithm deciding open implication in DSPACE($|\psi|^2$) which is in PSPACE.

The time complexity of the algorithm is dominated by the for-loop, the largest k is $3|Q_A^n| \cdot 2^{|Q_B|}$ and as the number of states in the ABW is $|Q_B| = O(|\psi|)$, the time complexity is exponential in the size of ψ, which agrees with the well known fact that PSPACE $\subseteq$ EXPTIME.

To sum up, our algorithm needs polynomial space in the size of ψ and double exponential time in the size of φ. Thus it meets the lower bound derived in Section 3.5.

Chapter 5

Implementation

Since open implication for full LTL is 2EXP-hard we do not implement it. Our implementation is an add-on to a program called Anzu which was used for the case study on automatic hardware synthesis from specification described in [BGJ$^+$07a, BGJ$^+$07b, Job07]. Anzu is an implementation of the algorithm proposed in [PPS06] by Piterman, Pnueli and Sa'ar. The algorithm solves Mealy realizability for a special class of LTL formulas, that of generalized reactivity of rank 1 (GR(1) is defined in Section 5.1). GR(1) includes most useful specifications of digital designs.

In this chapter realizability and open implication denote Mealy realizability and Mealy open implication, the reason is explained in Section 5.2. Realizability of GR(1) formulas is solved in [PPS06] through games. The specifying GR(1) formula is transformed into a game and the specification is realizable if and only if there exists a winning strategy for the system, also see Section 4.3. All the states for which a winning strategy for the system exists are in the *winning region* of the game graph. The winning region of the system on the game graph is calculated using a triply nested fixpoint formula. A nice property of the algorithm is that it can be implemented symbolically. Anzu is a symbolic implementation using binary decision diagrams (BDDs [Bry86]) as data structure.

We propose a new fixpoint formula, to solve Mealy open implication. The main idea of the underlying algorithm is the same as described in Section 3.6 and used in Chapter 4. In order to show that φ does not open imply ψ, we look for a Mealy machine which realizes φ but not ψ by searching for a Mealy machine realizing φ with a word not satisfying ψ.

In the next section we will define the class of specifications considered. Subsequently we explain the winning and the reachable states. In Section 5.3 we show how the underlying fixpoint formula of the algorithm is derived from the specifying GR(1) formula. Then in Section 5.4 the algorithm and some implementational details are explained. After this we analyze the complexity of the algorithm and last but not least we will give an overview of the results of our case study.

5.1 GR(1)

An LTL formula φ over input variables I and output variables O is in the class of *generalized reactivity of rank 1* (GR(1)) if it can be written in the following form

$$\varphi = \varphi_i^e \wedge \varphi_t^e \wedge \varphi_f^e \rightarrow \varphi_i^s \wedge \varphi_t^s \wedge \varphi_f^s,$$

where

- φ_i^α for α in $\{e, s\}$ are Boolean formulas over I for $\alpha = e$ respectively over O for $\alpha = s$ which characterize the initial states of the environment respectively the system.

- φ_t^α for α in $\{e, s\}$ are formulas of the form $\bigwedge_{i \in I} \mathsf{G}\, p_i$ describing the transitions. Each p_i is a Boolean combination of values of input and output variables and expressions of the form $\mathsf{X}\, v$. For the environment transitions (φ_t^e) v is a value of an input variable, for the system transitions (φ_t^s) v can be the value of an arbitrary variable.

- φ_f^α for α in $\{e, s\}$ are formulas of the form $\bigwedge_{i \in I} \mathsf{G}\,\mathsf{F}\, p_i$ where each p_i is a Boolean formula over $I \cup O$. They encode the fairness constrains for the environment respectively the system.

In our implementation Anzu we consider LTL specifications of the form $\varphi = \bigwedge_{k=1}^m \varphi_k^e \rightarrow \bigwedge_{k=1}^n \varphi_k^s$, where each φ_k^e and φ_k^s can be represented by a DBW. How such specifications can be transformed into GR(1) formulas is shown below. Recall that not all LTL specifications can be converted into a DBW, see Example 30 in Section 4.1.

Given a DBW $A = (\Sigma, Q, q_0, \delta, Acc)$, a formula of the following form $\varphi = \varphi_i \wedge \varphi_t \wedge \varphi_f$ can be constructed. For each state q_j in Q define a new state variable q_j. The initial state of the automaton is encoded as $\varphi_i = q_0 \wedge \bigwedge_{q_j \in \{Q \setminus q_0\}} \neg q_j$. The transitions of the automaton are stored in φ_t. For every transition $\delta(q_1, s) = q_2$ with q_1, q_2 in Q and s in Σ add the formula $\mathsf{G}((q_1 \wedge s) \rightarrow \mathsf{X}\, q_2)$ with a conjunction to φ_t. Also add the formula $\mathsf{G}(\bigvee_{q_j \in Q}(q_j \wedge \bigwedge_{q_k \in \{Q \setminus q_j\}} \neg q_k))$ with a conjunction to φ_t in order to make sure that the automaton only has one current state at a time. For each accepting state q in Acc add $\mathsf{G}\,\mathsf{F}\, q$ to φ_f. Note that it is easy to find a more efficient encoding, for example if the states are represented bitwise.

Given an LTL formula of the form $\varphi = \bigwedge_{k=1}^m \varphi_k^e \rightarrow \bigwedge_{k=1}^n \varphi_k^s$ over input variables I and output variables O, where each φ_k^α can be represented by a DBW $A_k^\alpha = (2^I \cup 2^O, Q_k^\alpha, q_{k,0}^\alpha, \delta_k^\alpha, Acc_k^\alpha)$ for α in $\{e, s\}$. We convert φ into the following GR(1) formula:

- $\varphi_i^e = \mathsf{true}$

- $\varphi_i^s = \bigwedge_{k=1}^m \varphi_{k,i}^e \wedge \bigwedge_{k=1}^n \varphi_{k,i}^s$ where $\varphi_{k,i}^\alpha$ is the formula encoding the initial states of the automaton corresponding to the formula φ_k^α for α in $\{e, s\}$.
- $\varphi_t^e = \mathsf{true}$
- $\varphi_t^s = \bigwedge_{k=1}^m \varphi_{k,t}^e \wedge \bigwedge_{k=1}^n \varphi_{k,t}^s$ where $\varphi_{k,t}^\alpha$ is the formula encoding the transitions of the automaton corresponding to the formula φ_k^α for α in $\{e, s\}$.
- $\varphi_f^\alpha = \bigwedge_k \varphi_{k,f}^\alpha$ where $\varphi_{k,f}^\alpha$ is the formula encoding the accepting states of the automaton corresponding to the formula φ_k^α for α in $\{e, s\}$.

The resulting GR(1) formula is a formula over input variables I output variables O and state variables $\bigcup_{k=1}^m Q_k^e \cup \bigcup_{k=1}^n Q_k^s$.

A word σ in $(2^I \times 2^O)^\omega$ satisfies such a specification φ in GR(1) if and only if the corresponding paths in the DBWs satisfy the system fairness conditions if the environment fairness conditions are satisfied, e.g. the corresponding paths have to satisfy $\varphi_f^e \to \varphi_f^s$. The paths in the DBWs corresponding to a word σ are characterized by the formulas for the initial states and the transitions, e.g. they also have to satisfy $\varphi_i^e \wedge \varphi_i^s \wedge \varphi_t^e \wedge \varphi_t^s$. Because the DBWs are deterministic we sometimes refer to the paths in the DBWs as the path in the DBWs.

Example 46. We will construct the GR(1) formula for the specification $\varphi = \mathsf{X}\, a \vee \mathsf{X}\, r$ from Example 7 in Section 2.6. In order to get a formula of the form $\varphi = \bigwedge_{k=1}^m \varphi_k^e \to \bigwedge_{l=1}^n \varphi_l^s$ we change it to $\varphi = \mathsf{true} \to \mathsf{X}\, a \vee \mathsf{X}\, r$. Both formulas true and $\mathsf{X}\, a \vee \mathsf{X}\, r$ can be transformed into a DBW.

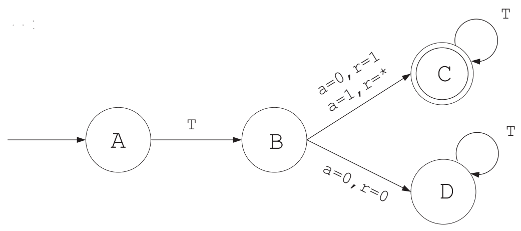


Figure 5.1: DBW for $\mathsf{X}\, a \vee \mathsf{X}\, r$

The resulting GR(1) formula is shown below, the variables q_A, q_B, q_C and q_D are the state variables for the states A, B, C and D of the automaton shown in Figure 5.1. We do not explicitly show the DBW for true, it is very easy to encode the formula without an automaton. Also note that $\mathsf{G}\, f \wedge \mathsf{G}\, g$ is equal to $\mathsf{G}(f \wedge g)$.

- $\varphi_i^e = \mathsf{true}$
- $\varphi_t^e = \mathsf{true}$

- $\varphi_f^e = \mathsf{G}\,\mathsf{F}\,\mathsf{true}$
- $\varphi_i^s = q_A \wedge \neg q_B \wedge \neg q_C \wedge \neg q_D$
- $\varphi_t^s = \mathsf{G}((q_A \rightarrow \mathsf{X}\,q_B) \wedge ((q_B \wedge (a \vee r))) \rightarrow \mathsf{X}\,q_C) \wedge$
  $((q_B \wedge \neg a \wedge \neg r) \rightarrow \mathsf{X}\,q_D) \wedge (q_C \rightarrow \mathsf{X}\,q_C) \wedge (q_D \rightarrow \mathsf{X}\,q_D) \wedge$
  $((q_A \wedge \neg q_B \wedge \neg q_C \wedge \neg q_D) \vee (\neg q_A \wedge q_B \wedge \neg q_C \wedge \neg q_D) \vee$
  $(\neg q_A \wedge \neg q_B \wedge q_C \wedge \neg q_D) \vee (\neg q_A \wedge \neg q_B \wedge \neg q_C \wedge q_D)))$
- $\varphi_f^s = \mathsf{G}\,\mathsf{F}\,q_C$

The word $\binom{a=1}{r=0}\binom{a=1}{r=0}\binom{a=1}{r=0}\ldots$ satisfies the GR(1) formula because the corresponding path $q_A q_B q_C q_C \ldots$ satisfying the initial conditions $\varphi_i^e \wedge \varphi_i^s$ and the transitions $\varphi_t^e \wedge \varphi_t^s$ satisfies the system fairness condition $\mathsf{G}\,\mathsf{F}\,q_C$. □

5.2 Winning region and reachable states

In order to calculate open implication of GR(1) formulas we need to calculate the winning region w which corresponds to the set of nonempty states Q^n used in Chaper 4, see Section 4.2 and Section 4.3. In this section we will explain how w is calculated by Anzu following the algorithm introduced in [PPS06]. Additionally we need to calculate the set of reachable states, which will be explained in the last part of this section.

Given a formula in GR(1) as described in Section 5.1, the algorithm introduced in [PPS06] calculating realizability works in three steps. It starts by constructing a game graph from the GR(1) formula. The states of the game graph correspond to the states of the specifying DBWs. The transitions in the graph reflect the possible choices of the system and the environment. The game is winning for the system if and only if there exist possible values for the output variables for any values of the input variables such that the evolving words satisfy the GR(1) formula. Thus if the game is winning for the system there exists a 2^O-labeled 2^I-tree such that all words of the tree satisfy the specifying formula. Consequently if the game is winning for the system then there exists a Mealy machine which realizes the GR(1) formula.

The next step of the algorithm after calculating the game graph is to calculate the winning region, because the specification is realizable if and only if the initial states are in the winning region. A state of the game graph is in the winning region if there is an accepting word for the corresponding states in the DBWs for any possible input of the environment. Thus there is a 2^O-labeled 2^I-tree such that all words of the tree are accepted by the DBWs starting at the states corresponding to the winning state in the game graph. The winning region is calculated with a triply nested fixpoint formula over the game graph. Let $|Q|$ be the number of states of all specifying DBWs, then the algorithm needs time proportional to $|Q|^3$ to calculate the winning region.

Last the algorithm checks if the initial states are in the winning region. If they are, then the GR(1) formula is realizable, else it is not.

Anzu can calculate the winning states of the game corresponding to a GR(1) formula and store the corresponding states of the DBWs in w. To be correct we would have to say that w is a μ-calculus formula denoting the set of states $[\![w]\!]$ which are winning. To make notation a little easier we sometimes use w short for $[\![w]\!]$.

For any state in the winning region w of the DBWs there exists a 2^O-labeled 2^I-tree where all words have a path in the DBWs satisfying the system fairness conditions if the environment fairness conditions are satisfied. If we can find a word σ with a path satisfying the system fairness condition if the environment fairness condition is satisfied and the path consists of states in the winning region of the DBWs then we can find a Mealy machine with the word σ which realizes the specification. This can be proved in the same way as Claim 33 in Section 4.2. Note that the nonempty states set Q^n of the DPT is identical to the winning region of the DBWs. We can work with DBWs here because we consider a smaller set of specifications. In Claim 33 we assumed that not only the states of the path itself are in Q^n but also all the next states for any input. This was required to calculate Moore realizability because Moore machines react one time step later. Contrarily Mealy machines can react immediately and thus only the states of the path itself must be in the winning region w when we calculate Mealy realizability.

Another set of states we will need later on is the set of reachable states. The formula r denotes all the states which are reachable from an initial state satisfying the transition functions and w. Again we often use r short for $[\![r]\!]$. How to calculate r will be described in Section 5.4.

5.3 The Formula

Given two formulas φ and ψ in GR(1) ($\varphi = \varphi_i^e \wedge \varphi_t^e \wedge \varphi_f^e \rightarrow \varphi_i^s \wedge \varphi_t^s \wedge \varphi_f^s$, $\psi = \psi_i^e \wedge \psi_t^e \wedge \psi_f^e \rightarrow \psi_i^s \wedge \psi_t^s \wedge \psi_f^s$) with input alphabet 2^I and output alphabet 2^O, representing DBWs as described in Section 5.1 above. First we calculate the formula w for the winning set of the DBWs of φ. Then we want to know if there is a word in $(2^O \times 2^I)^\omega$ with a path in the DBWs which

1. consists of states satisfying w,
2. satisfies the fairness condition of φ, i.e. satisfies $\varphi_f^e \rightarrow \varphi_f^s$ and
3. does not satisfy the fairness condition of ψ, i.e. satisfies $\neg(\psi_f^e \rightarrow \psi_f^s)$.

Claim 47. *If and only if a word with a path in the DBWs satisfying (1), (2) and (3) exists then there is a Mealy machine which realizes φ but not ψ.*

Proof. Assume there exists such a word then there exists a Mealy machine realizing φ because the path satisfies the fairness condition of φ and it consists of states in the winning region. Clearly the Mealy machine does not realize ψ because the path of the word does not satisfy the fairness condition of ψ.

Assume there exists a Mealy machine realizing φ but not ψ then there exists a 2^O-labeled 2^I-tree where all paths of the words of the tree satisfy the fairness condition of φ and at least one path does not satisfy the fairness condition of ψ. Thus there is a word in $(2^O \times 2^I)^\omega$ with a path in the DBWs satisfying (1), (2) and (3). □

Considering the fairness conditions of the given formulas:

$$\varphi_f^e = \mathsf{G}\,\mathsf{F}\, e_{\varphi,1} \wedge \mathsf{G}\,\mathsf{F}\, e_{\varphi,2} \wedge \ldots \wedge \mathsf{G}\,\mathsf{F}\, e_{\varphi,n_1}$$

$$\varphi_f^s = \mathsf{G}\,\mathsf{F}\, s_{\varphi,1} \wedge \mathsf{G}\,\mathsf{F}\, s_{\varphi,2} \wedge \ldots \wedge \mathsf{G}\,\mathsf{F}\, s_{\varphi,m_1}$$

$$\psi_f^e = \mathsf{G}\,\mathsf{F}\, e_{\psi,1} \wedge \mathsf{G}\,\mathsf{F}\, e_{\psi,2} \wedge \ldots \wedge \mathsf{G}\,\mathsf{F}\, e_{\psi,n_2}$$

$$\psi_f^s = \mathsf{G}\,\mathsf{F}\, s_{\psi,1} \wedge \mathsf{G}\,\mathsf{F}\, s_{\psi,2} \wedge \ldots \wedge \mathsf{G}\,\mathsf{F}\, s_{\psi,m_2}$$

We want to calculate if there exists a path in the DBWs which satisfies the LTL formula shown below.

$$\mathsf{G}\, w \wedge (\varphi_f^e \to \varphi_f^s) \wedge \neg(\psi_f^e \to \psi_f^s)$$

The formula can be transformed into:

$$\underbrace{\bigvee_{i=1,l=1}^{n_1,m_2} (\mathsf{G}\, w \wedge \neg\,\mathsf{G}\,\mathsf{F}\, e_{\varphi,i} \wedge \neg\,\mathsf{G}\,\mathsf{F}\, s_{\psi,l} \wedge \underbrace{\bigwedge_{k=1}^{n_2} \mathsf{G}\,\mathsf{F}\, e_{\psi,k})}_{\psi\_env\_part}}_{big\_or\_part1}$$

$$\vee$$

$$\underbrace{\bigvee_{l=1}^{m_2} (\mathsf{G}\, w \wedge \neg\,\mathsf{G}\,\mathsf{F}\, s_{\psi,l} \wedge \underbrace{\bigwedge_{j=1}^{m_1} \mathsf{G}\,\mathsf{F}\, s_{\varphi,j}}_{\varphi\_sys\_part} \wedge \underbrace{\bigwedge_{k=1}^{n_2} \mathsf{G}\,\mathsf{F}\, e_{\psi,k})}_{\psi\_env\_part}}_{big\_or\_part2}$$

This leads us to the fixpoint formula γ over the given DBWs. Solving this formula results in all the states which satisfy w, the fairness condition of φ and not of ψ.

$$\gamma = \underbrace{\bigvee_{i=1,l=1}^{n_1,m_2} (\nu Y . (w \wedge \neg e_{\varphi,i} \wedge \neg s_{\psi,l} \wedge \underbrace{\bigwedge_{k=1}^{n_2} \mathsf{EX}(\mathsf{E}(Y\,\mathsf{U}(Y \wedge e_{\psi,k}))))}_{\psi\_env\_part})}_{big\_or\_part1}$$

$$\vee$$

$$\underbrace{\bigvee_{l=1}^{m_2} (\nu Y . (w \wedge \neg s_{\psi,l} \wedge \underbrace{\bigwedge_{j=1}^{m_1} \mathsf{EX}(\mathsf{E}(Y\,\mathsf{U}(Y \wedge s_{\varphi,j})))}_{\varphi\_sys\_part} \wedge \underbrace{\bigwedge_{k=1}^{n_2} \mathsf{EX}(\mathsf{E}(Y\,\mathsf{U}(Y \wedge e_{\psi,k}))))))}_{\psi\_env\_part}}_{big\_or\_part2}$$

Claim 48. *There exists a path in the DBWs satisfying*

$$\mathsf{G}\,w \wedge \neg\,\mathsf{G}\,\mathsf{F}\,e_{\varphi,i} \wedge \neg\,\mathsf{G}\,\mathsf{F}\,s_{\psi,l} \wedge \bigwedge_{k=1}^{n_2} \mathsf{G}\,\mathsf{F}\,e_{\psi,k}$$

for some i smaller or equal to n_1 and some l smaller or equal to m_2 if and only if the set

$$[\![r \wedge \nu Y . (w \wedge \neg e_{\varphi,i} \wedge \neg s_{\psi,l} \wedge \bigwedge_{k=1}^{n_2} \mathsf{EX}(\mathsf{E}(Y\,\mathsf{U}(Y \wedge e_{\psi,k}))))]\!]$$

is not empty.

Proof. Assume there is a path ρ in the DBWs satisfying

$$\mathsf{G}\,w \wedge \neg\,\mathsf{G}\,\mathsf{F}\,e_{\varphi,i} \wedge \neg\,\mathsf{G}\,\mathsf{F}\,s_{\psi,l} \wedge \bigwedge_{k=1}^{n_2} \mathsf{G}\,\mathsf{F}\,e_{\psi,k}$$

for some i smaller or equal to n_1 and some l smaller or equal to m_2.

First of all, all states in ρ are in $[\![r]\!]$ because ρ is a path starting at an initial state following the transition functions and all states satisfy w according to the $\mathsf{G}\,w$ part of the formula.

Second we show that from some point onwards all the states of ρ are in the set $[\![\nu Y . (w \wedge \neg e_{\varphi,i} \wedge \neg s_{\psi,l} \wedge \bigwedge_{k=1}^{n_2} \mathsf{EX}(\mathsf{E}(Y\,\mathsf{U}(Y \wedge e_{\psi,k}))))]\!]$. Looking at the LTL formula of the assumption it is clear that from some point onwards all the states in ρ satisfy $w \wedge \neg e_{\varphi,i} \wedge \neg s_{\psi,l}$ because $\neg\,\mathsf{G}\,\mathsf{F}\,e_{\varphi,i} \wedge \neg\,\mathsf{G}\,\mathsf{F}\,s_{\psi,l}$ is equal to $\mathsf{F}\,\mathsf{G}(\neg e_{\varphi,i} \wedge \neg s_{\psi,l})$ and w holds for all states of ρ. The $\bigwedge_{k=1}^{n_2} \mathsf{G}\,\mathsf{F}\,e_{\psi,k}$ part of the LTL formula assures that states satisfying $e_{\psi,k}$ for any k smaller or equal to n_2 reoccur infinitely often from some point onwards in ρ. To sum up, from some point onwards the states in ρ satisfy $w \wedge \neg e_{\varphi,i} \wedge \neg s_{\psi,l}$

and states satisfying $e_{\psi,k}$ for any k smaller or equal to n_2 reoccur infinitely often and consequently the set

$$[\![r \wedge \nu Y\,.(w \wedge \neg e_{\varphi,i} \wedge \neg s_{\psi,l} \wedge \bigwedge_{k=1}^{n_2} \mathsf{EX}(\mathsf{E}(Y\,\mathsf{U}(Y \wedge e_{\psi,k}))))]\!]$$

is not empty.

On the other hand, assume that the set

$$[\![r \wedge \nu Y\,.(w \wedge \neg e_{\varphi,i} \wedge \neg s_{\psi,l} \wedge \bigwedge_{k=1}^{n_2} \mathsf{EX}(\mathsf{E}(Y\,\mathsf{U}(Y \wedge e_{\psi,k}))))]\!]$$

is not empty for some i smaller or equal to n_1 and some l smaller or equal to m_2.

Then states in $[\![\nu Y\,.(w \wedge \neg e_{\varphi,i} \wedge \neg s_{\psi,l} \wedge \bigwedge_{k=1}^{n_2} \mathsf{EX}(\mathsf{E}(Y\,\mathsf{U}(Y \wedge e_{\psi,k}))))]\!]$ can be reached from the initial states of the automata, following the transition functions and always satisfying w. The states in $[\![\nu Y\,.(w \wedge \neg e_{\varphi,i} \wedge \neg s_{\psi,l} \wedge \bigwedge_{k=1}^{n_2} \mathsf{EX}(\mathsf{E}(Y\,\mathsf{U}(Y \wedge e_{\psi,k}))))]\!]$ are all states from which it is possible to stay in states satisfying $w \wedge \neg e_{\varphi,i} \wedge \neg s_{\psi,l}$ and to reach states satisfying $e_{\psi,k}$ for any k smaller or equal to n_2 and still satisfying $w \wedge \neg e_{\varphi,i} \wedge \neg s_{\psi,l}$.

Consequently there exists a path in the DBWs which satisfies

$$\mathsf{G}\,w \wedge \neg\,\mathsf{G}\,\mathsf{F}\,e_{\varphi,i} \wedge \neg\,\mathsf{G}\,\mathsf{F}\,s_{\psi,l} \wedge \bigwedge_{k=1}^{n_2} \mathsf{G}\,\mathsf{F}\,e_{\psi,k}$$

. □

Corollary 49. *There exists a path in the DBWs satisfying the big_or_part1 of the LTL formula from above if and only if the set $[\![big\_or\_part1]\!]$ from the μ-calculus formula γ, intersected with $[\![r]\!]$ is not empty.*

Proof. This follows from Claim 48 and the fact that $\mathsf{E}\,f \vee \mathsf{E}\,g$ is equal to $\mathsf{E}(f \vee g)$ and $(r \wedge f) \vee (r \wedge g)$ is equal to $r \wedge (f \vee g)$.

If and only if there exists a path satisfying *big_or_part*1 of the LTL formula then there exists a path satisfying the LTL formula from Claim 48 for at least one i smaller or equal to n_1 and at least one l smaller or equal to m_2. If and only if a path satisfying the the LTL formula from Claim 48 for at least one i smaller or equal to n_1 and at least one l smaller or equal to m_2 exists then the set from Claim 48 is not empty. If and only if the set from Claim 48 is not empty for at least one i smaller or equal to n_1 and at least one l smaller or equal to m_2 then the set $[\![r \wedge big\_or\_part1]\!]$ is not empty. □

The property shown in Corrollary 49 for *big_or_part*1 can also be shown for *big_or_part*2 in the same way.

Again we use the fact that the existential path quantifier E distributes over $\vee$ and that $(r \wedge f) \vee (r \wedge g)$ is equal to $r \wedge (f \vee g)$. This leads us to the following equivalences:

$$\mathsf{E}(big\_or\_part1 \vee big\_or\_part2) = \mathsf{E}(big\_or\_part1) \vee \mathsf{E}(big\_or\_part2) \text{ and}$$

$$[\![r \wedge (big\_or\_part1 \vee big\_or\_part2)]\!] = [\![(r \wedge big\_or\_part1) \vee (r \wedge big\_or\_part2)]\!].$$

Since we showed above that $\mathsf{E}(big\_or\_part1) \vee \mathsf{E}(big\_or\_part2)$ is equal to $[\![(r \wedge big\_or\_part1) \vee (r \wedge big\_or\_part2)]\!]$, we just proved that there exists a path in the DBWs satisfying the LTL formula above if and only if the set $[\![r \wedge \gamma]\!]$ is not empty. We conclude that φ open implies ψ if and only if the set $[\![r \wedge \gamma]\!]$ is empty.

5.4 The Algorithm

In Algorithm 3 and 4 we show how to calculate big_or_part1 and big_or_part2 from the fixpoint formula above. Formulas of the form $\mathsf{E}(Y\,\mathsf{U}(Y \wedge f))$ can also be written as $\mu Z\,.(Y \wedge f \vee (Y \wedge \mathsf{EX}(Z)))$. Thus we have a least fixpoint nested in a greatest fixpoint in Algorithm 3 and two least fixpoints nested in a greatest fixpoint in Algorithm 4. The fixpoints are while loops which stop, when the variable of the fixpoint does not change any more. Greatest fixpoints start with the set of all states and least fixpoints start with the empty set.

The winning region w is calculated as described in [PPS06] and the $ex(z)$ function in Algorithm 3 and 4 calculates all states which can reach a state in z in one step.

Subsequently we have to determine, if it is possible to reach one of the state sets big_or_part1, big_or_part2 from the initial states, always staying in the winning region w. There are two ways of achieving this. In our implementation we calculated r as described in Algorithm 5 and checked if $[\![r \wedge (big\_or\_part1 \vee big\_or\_part2)]\!]$ is empty.

The other option would be to calculate reachability of $big\_or\_part1 \vee big\_or\_part2$ by using the formula proposed in Section 2.7:

$$s = \mu Y\,.((big\_or\_part1 \vee big\_or\_part2) \vee \mathsf{EX}(Y \wedge w))$$

and calculating if an initial state is in $[\![s]\!]$ by checking if $[\![s \wedge i]\!]$ is empty, where $i = \varphi_i^e \wedge \varphi_i^s \wedge \psi_i^e \wedge \psi_i^s$ and $[\![i]\!]$ are the initial states.

For both options the following holds. If the described set is not empty then there exists a path which satisfies the fairness condition for φ always staying in w and which does not satisfy the fairness condition for ψ. Consequently there exists a Mealy machine which realizes φ but not ψ and thus φ does not open imply ψ. On the other hand, if it is empty then such a path does not exist and φ open implies ψ.

Algorithm 3 big_or_part1

```
$big\_or\_part1 = 0$
for $i = 1$ to $n_1$ do
    for $l = 1$ to $m_2$ do
        greatest fixpoint y
            $\psi\_env\_part = 1$
            for $k = 1$ to $n_2$ do
                least fixpoint z
                    $z = (y \wedge e_{\psi,k}) \vee (y \wedge ex(z))$
                end least fixpoint z
                $\psi\_env\_part = \psi\_env\_part \wedge ex(z)$
            end for
            $y = w \wedge \neg e_{\varphi,i} \wedge \neg s_{\psi,l} \wedge \psi\_env\_part$
        end greatest fixpoint y
        $big\_or\_part1 = big\_or\_part1 \vee y$
    end for
end for
```

Algorithm 4 big_or_part2

```
$big\_or\_part2 = 0$
for $l = 1$ to $m_2$ do
    greatest fixpoint y
        $\varphi\_sys\_part = 1$
        for $j = 1$ to $m_1$ do
            least fixpoint z
                $z = (y \wedge s_{\varphi,j}) \vee (y \wedge ex(z))$
            end least fixpoint z
            $\varphi\_sys\_part = \varphi\_sys\_part \wedge ex(z)$
        end for
        $\psi\_env\_part = 1$
        for $k = 1$ to $n_2$ do
            least fixpoint z
                $z = (y \wedge e_{\psi,k}) \vee (y \wedge ex(z))$
            end least fixpoint z
            $\psi\_env\_part = \psi\_env\_part \wedge ex(z)$
        end for
        $y = w \wedge \neg s_{\psi,l} \wedge \varphi\_sys\_part \wedge \psi\_env\_part$
    end greatest fixpoint y
    $big\_or\_part2 = big\_or\_part2 \vee y$
end for
```

Algorithm 5 Calculating the reachable states r

```
$r = w \wedge \varphi_i^e \wedge \varphi_i^s \wedge \psi_i^e \wedge \psi_i^s$
$r_{old} = \mathsf{false}$
while $r_{old} \neq r$ do
    $r_{old} = r$
    $r = \{q \in w \mid \exists s \in (\Sigma \times D) : (r \wedge s \rightarrow \mathsf{X}\, q) \models \varphi_t^e \wedge \varphi_t^s \wedge \psi_t^e \wedge \psi_t^s\}$
end while
```

5.5 Complexity

As already mentioned in Section 2.7 the complexity of a μ-calculus formula depends on the alternation depth k of the fixpoint formula [EL86]. First of all we will analyze the complexity of Algorithm 3, starting with the inner least fixpoint over Z. Let $|Q|$ be the number of states in the DBWs of the specifications φ and ψ. We know that a fixpoint is a monotonic function and thus Z can only change $|Q|$ times, thus we have at most $|Q|$ calls to the $ex(z)$ function in one fixpoint calculation. The inner least fixpoint and another $ex(z)$ is calculated n_2 times. Then we have a greatest fixpoint over Y around the least fixpoint and the for loop. Again the fixpoint function is monotonic and thus Y can only change $|Q|$ times. Together with the last two for-loops everything adds up to:

$$n_1 \cdot m_2 \cdot |Q| \cdot n_2 \cdot (|Q| \cdot ex(z) + ex(z))$$

where n_1, m_2, n_2 are the number of accepting states. The function $ex(z)$ depends on the number of edges, thus it is in $O(|Q|^2)$ because the number of edges is at most quadratic in the number of states.

The complexity for Algorithm 4 can be derived in the same way, it is

$$m_2 \cdot |Q| \cdot (m_1 \cdot (|Q| \cdot ex(z) + ex(z)) + n_2 \cdot (|Q| \cdot ex(z) + ex(z)))$$

Using big O notation Algorithm 3 and 4 need $O(|Q|^4)$ time or $O(|Q|^2)$ $ex(z)$ computations.

Additionally we have to consider the time we need to calculate the winning region. The fixpoint formula from [PPS06] has alternation depth 3 and again we have to calculate a function depending on the edges of the specifying DBWs. Let $|Q_\varphi|$ be the number of states in the specifying DBWs of φ, then we need $O(|Q_\varphi|^6)$ time to calculate the winning region.

To sum up, our implementation deciding open implication for GR(1) formulas needs $O(|Q_\varphi^6| + |Q_\psi^4|)$ time where $|Q_\varphi|$ is the number of states in the DBWs of the first specification and $|Q_\psi|$ is the number of states in the DBWs of the second specification. Hence the algorithm is in P and needs linear space in the number of states of the specification.

5.6 Results

We started testing small specifications such as $\varphi = \mathsf{X}\,a \vee \mathsf{X}\,r$ and $\psi = \mathsf{X}\,a \vee \mathsf{F}\,r$ given in Example 8 in Chapter 3. With our program we showed that φ open implies ψ which confirms our intuition. On the other hand we could also calculate that ψ does not open imply φ if the initial value of r is 0 or not defined, which matches the result shown in Example 8.

A more interesting example with input variable r and output variable a is $\varphi = (\mathsf{G}\,\mathsf{F}\,r \rightarrow \mathsf{G}(a \rightarrow \mathsf{X}(\neg a\,\mathsf{U}\,r)))$ and $\psi = \mathsf{G}(a \rightarrow \mathsf{X}(\neg a\,\mathsf{W}\,r))$. Figure 5.2 and 5.3 show the DBW for φ^s respectively ψ^s and the GR(1) formulas for φ and ψ are also shown below.

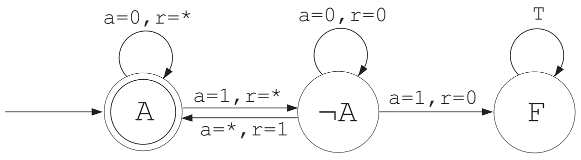


Figure 5.2: DBW for $\varphi^s = \mathsf{G}(a \rightarrow \mathsf{X}(\neg a\,\mathsf{U}\,r))$

$$\varphi = \mathsf{G}\,\mathsf{F}\,r \rightarrow ((\neg q_A) \wedge \mathsf{G}((\neg q_A \wedge \neg a \rightarrow \mathsf{X}\,\neg q_A) \wedge (\neg q_A \wedge a \rightarrow \mathsf{X}\,q_A) \wedge (q_A \wedge \neg a \wedge \neg r \rightarrow \mathsf{X}\,q_A) \wedge (q_A \wedge a \wedge \neg r \rightarrow \mathsf{X}\,q_F) \wedge (q_A \wedge r \rightarrow \mathsf{X}\,\neg q_A) \wedge (q_F \rightarrow \mathsf{X}\,q_F)) \wedge \mathsf{G}\,\mathsf{F}(q_A))$$

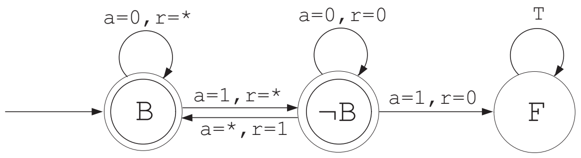


Figure 5.3: DBW for $\psi^s = \mathsf{G}(a \rightarrow \mathsf{X}(\neg a\,\mathsf{W}\,r)))$

$$\psi = \mathsf{true} \rightarrow ((\neg q_B) \wedge \mathsf{G}((\neg q_B \wedge \neg a \rightarrow \mathsf{X}\,\neg q_B) \wedge (\neg q_B \wedge a \rightarrow \mathsf{X}\,q_B) \wedge (q_B \wedge \neg a \wedge \neg r \rightarrow \mathsf{X}\,q_B) \wedge (q_B \wedge a \wedge \neg r \rightarrow \mathsf{X}\,q_F) \wedge (q_B \wedge r \rightarrow \mathsf{X}\,\neg q_B) \wedge (q_F \rightarrow \mathsf{X}\,q_F)) \wedge \mathsf{G}\,\mathsf{F}(\neg q_F))$$

Using our implementation of the algorithm described above, we showed that they are open equivalent.

The weak until is much easier to synthesize which can also be seen using the synthesis functionality of Anzu. The table below shows the number of bdd-nodes needed to represent the winning strategy which Anzu produces for the respective specification. In the last column of the table the number of bdd-nodes of the functional representation is shown.

formula	size of strategy	functional size
$\mathsf{G}\,\mathsf{F}\,r \rightarrow \mathsf{G}(a \rightarrow \mathsf{X}(\neg a\,\mathsf{U}\,r))$	31	9
$\mathsf{G}(a \rightarrow \mathsf{X}(\neg a\,\mathsf{W}\,r))$	22	5

Naturally smaller winning strategies can be converted into smaller realizing open modules. Since we know that open equivalence holds we can use the smaller realizing open module to realize the bigger specification.

Finally we looked at the specifications of the arbiter for ARM's AMBA AHB bus used in the case study discussed in [BGJ$^+$07a] and [BGJ$^+$07b] by Bloem et al.. The specification in [BGJ$^+$07a] could only be synthesized by Anzu for up to 4 masters. In [BGJ$^+$07b] the specification was improved by adding a new signal. For the improved specification which is shown in Table 5.1, Anzu can synthesize the arbiter for up to 10 masters.

Table 5.1: new LTL specification of the AMBA arbiter

G0	$\mathsf{G}(HMASTER = i \rightarrow (\neg HBUSREQ[i] \leftrightarrow \neg BUSREQ)$
A1	$\mathsf{G}((HMASTLOCK \wedge HBURST = INCR) \rightarrow \mathsf{X}\,\mathsf{F}\,\neg BUSREQ)$
A2	$\mathsf{G}\,\mathsf{F}\,HREADY$
A3	$\bigwedge_i \mathsf{G}(HLOCK[i] \rightarrow HBUSREQ[i])$
G1	$\mathsf{G}(\neg HREADY \rightarrow \mathsf{X}(\neg START))$
G2	$\mathsf{G}((HMASTLOCK \wedge HBURST = INCR \wedge START) \rightarrow$ $\mathsf{X}(\neg START\,\mathsf{W}(\neg START \wedge \neg BUSREQ)))$
G3	$\mathsf{G}((HMASTLOCK \wedge HBURST = BURST4 \wedge START \wedge$ $HREADY) \rightarrow \mathsf{X}(\neg START\,\mathsf{W}[3](\neg START \wedge HREADY)))$ $\mathsf{G}((HMASTLOCK \wedge HBURST = BURST4 \wedge START \wedge$ $\neg HREADY) \rightarrow \mathsf{X}(\neg START\,\mathsf{W}[4](\neg START \wedge HREADY)))$
G4	$\bigwedge_i \mathsf{G}(HREADY \rightarrow (HGRANT[i] \leftrightarrow \mathsf{X}(HMASTER = i)))$
G5	$\mathsf{G}(HREADY \rightarrow (LOCKED \leftrightarrow \mathsf{X}(HMASTLOCK)))$
G6	$\bigwedge_i \mathsf{G}(\mathsf{X}(\neg START) \rightarrow$ $((HMASTER = i \leftrightarrow \mathsf{X}(HMASTER = i)) \wedge$ $(HMASTLOCK \leftrightarrow \mathsf{X}(HMASTLOCK))))$
G7	$\bigwedge_i \mathsf{G}((DECIDE \wedge \mathsf{X}(HGRANT[i])) \rightarrow$ $(HLOCK[i] \leftrightarrow LOCKED))$
G8	$\mathsf{G}(\neg DECIDE \rightarrow \bigwedge_i (HGRANT[i] \leftrightarrow \mathsf{X}(HGRANT[i])))$ $\mathsf{G}(\neg DECIDE \rightarrow (LOCKED \leftrightarrow \mathsf{X}(LOCKED)))$
G9	$\bigwedge_i \mathsf{G}(HBUSREQ[i] \rightarrow \mathsf{F}(\neg HBUSREG[i] \vee HMASTER = i))$
G10	$\bigwedge_{i \neq 0} \mathsf{G}(\neg HGRANT[i] \rightarrow (\neg HRGANT[i]\,\mathsf{W}\,HBUSREQ[i]))$ $\mathsf{G}((DECIDE \wedge \bigwedge_i \neg HBUSREQ[i]) \rightarrow \mathsf{X}(HGRANT[0]))$
G11	$DECIDE \wedge START \wedge HGRANT[0] \wedge HMASTER = 0 \wedge$ $\neg HMASTLOCK \wedge \bigwedge_{i \neq 0} \neg HGRANT[i]$
A4	$\bigwedge_i (\neg HBUSREQ[i] \wedge \neg HLOCK[i]) \wedge \neg HREADY$

For a more detailed explanation of the specification see [BGJ$^+$07a] or [Job07]. Here we will only discuss some of the changes which led to the better specification. The signal which was added was *BUSREQ*, it stores the value of *HBUSREQ*$[i]$ for the current master i, see Guarantee G0 in Table 5.1. Also the environment assumption A1 and the system guarantee

G2 where changed. In Table 5.2 we show the original specification G2' of G2. Both the old G2' and the new G2 guarantee that no new access starts before the current master lowers its request after starting a locked unspecified length burst.

Table 5.2: old system guarantee G2'

G2'	$\mathsf{G}((HMASTLOCK \wedge HBURST = INCR \wedge START) \rightarrow$ $\mathsf{X}(\neg START\,\mathsf{W}(\neg START \wedge \neg HBUSREQ[HMASTER])))$

Next we illustrate the DBWs for G2 and G2' for 2 masters. It is easy to see that the number of states of the DBW for G2 is always two independent of the number of masters, whereas the number of states for G2' grows with the number of masters. Consequently the new specification G2 needs less space and is easier to synthesize.

In the DBWs in Figure 5.4 and 5.5 below PRE is an abbreviation for $(HMASTLOCK \wedge HBURST = INCR \wedge START)$.

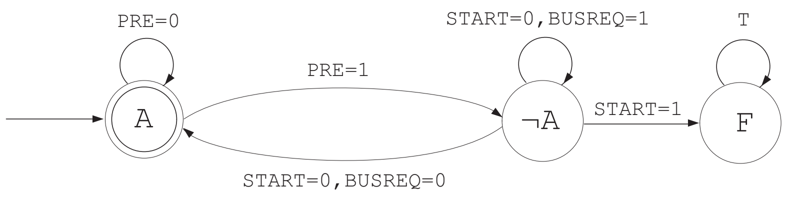


Figure 5.4: DBW for G2

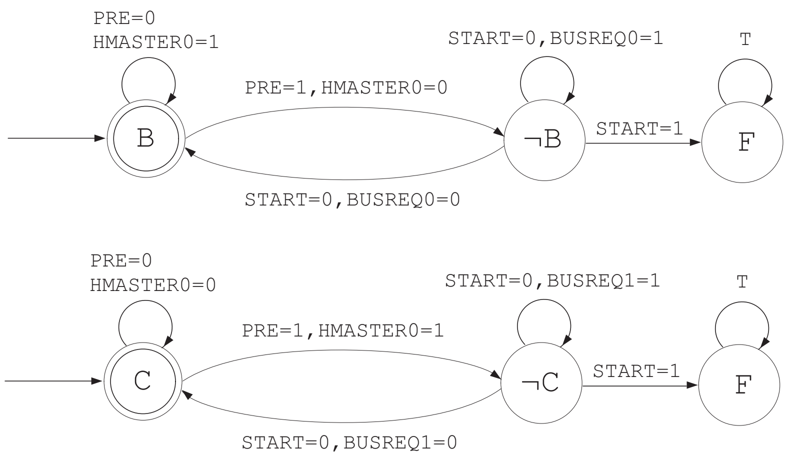


Figure 5.5: DBW for G2'

For our experiments testing open implication we took the new specification of the AMBA arbiter defined in Table 5.1 and compared it to the same specification with G2 replaced by G2' given in Table 5.2. Further on we use *new* to refer to the specification from Table 5.1 and *old* to refer to the same specification with G2'. Using Anzu we showed that *new* is open equivalent to *old*.

In order to compare the two specifications, realizing circuits were generated using Anzu. Subsequently the circuits were optimized and mapped to standard cells using ABC [Ber]. In Figure 5.6 we show the areas of the resulting arbiters as a function of the number of masters for the *new* and the *old* specification. For the *old* specification we could only calculate the needed area for up to six masters. For larger arbiters the computation ran out of memory, 2GB of memory were available.

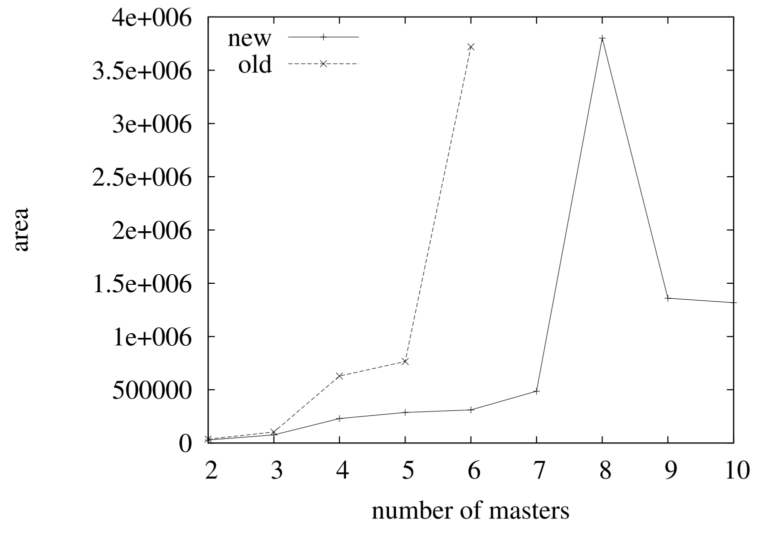


Figure 5.6: Size of the arbiter circuits

The results depicted in Figure 5.6 differ slightly from the results in [BGJ$^+$07b] although we use the same specification *new*. This can be explained by the following two facts. First of all we slightly modified the input file, because when we calculate open implication using Anzu both input files have to define the same sets of variables. Second, the BDD reordering which takes most of the time gives different results for different input files even when the specifications are the same.

Figure 5.6 shows very nicely how a small change in the specification can make a big difference. Although we only changed one signal (*BUSREQ*) in G2, a small part of the specification, the difference in the size of the resulting circuit is enormous.

In our last figure we compare the time Anzu takes to synthesize the *new* and the *old* specification with the time taken to calculate open implication. For the *old* specification Anzu can only calculate the circuits for up to 7 masters before running out of memory. In order to use the smaller circuits for the *old* specification we have to calculate if $\{M \in \mathcal{M} \mid M \models new\}$ is a subset of $\{M \in \mathcal{M} \mid M \models old\}$ because then we know that any small module realizing the *new* specification also realizes the *old* specification. Consequently the interesting question is: Does *new* open imply *old*? In Figure 5.7 the line for open implication denotes the time Anzu takes to calculate if $new \rightarrow_o old$. Since we only have to calculate the winning region for *new* and the fixpoint formula introduced in Section 5.3 above the time needed to decide open implication is even shorter than the time needed to synthesize *new*. We safe time because we do not have to build a strategy and we do not have to calculate the functional representation of the corresponding circuit.

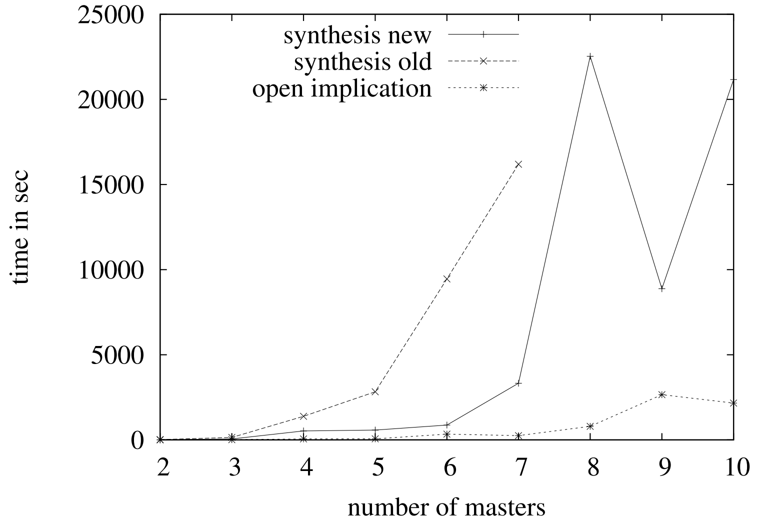


Figure 5.7: Time to synthesize and decide open implication

Figure 5.7 shows that the combined time needed to calculate open implication and to synthesize the *new* specification is shorter than the time needed to synthesize the *old* specification. Furthermore we could not synthesize the *old* specification for more than 7 masters but with the help of open implication we found a realizing circuit. Namely the circuit we obtain from synthesizing the *new* specification. Since we showed that *new* open implies *old* the smaller synthesized circuit also realizes the bigger specification.

To summarize, in this section we showed how open implication can be used to synthesize smaller circuits for big specifications or how open implication can help to find smaller specifications altogether.

Chapter 6

Summary and Conclusion

Although synthesis of open modules has been considered by researchers before, efforts to put the theory into practice have only been made in the last few years. With these new efforts new ideas arise automatically and one of those ideas is open implication.

In Chapter 3 we defined open implication and discussed its relation to trace inclusion. Comparing trace inclusion and open implication again, we know that trace inclusion is easier to calculate. On the other hand calculating open implication may lead to better results meaning that there are LTL formulas which are open equivalent but not trace equivalent. For example $\varphi = (\mathsf{G}\,\mathsf{F}\,r \rightarrow \mathsf{G}(a \rightarrow \mathsf{X}(\neg a\,\mathsf{U}\,r)))$ and $\psi = \mathsf{G}(a \rightarrow \mathsf{X}(\neg a\,\mathsf{W}\,r))$ are not trace equivalent because the infinite word $\sigma = \binom{a=1}{r=0}^{\omega}$ satisfies φ but not ψ but we showed that they are open equivalent in Section 5.6. Thus if we want to find smaller specifications defining the same set of open modules open equivalence might lead to better solutions.

In Chapter 4 we introduced an algorithm deciding open implication which meets the lower complexity bound. As in most algorithms in this research field we used automata to solve the problem. Automata are very popular because they are easier to understand and work with compared to a temporal logic. Unfortunately, in this case an algorithm which is easy to understand and meets the lower complexity bound does not entail an implementation because the time and space complexity is still way to high. Hence open implication for full LTL is more of theoretical interest than of practical at this time, but when synthesis for full LTL becomes a current topic in the "real world", we expect open implication to become of more practical interest too.

In the last part of this work we described our implementation which calculates open implication for a subset of LTL. We used the μ-calculus to describe our algorithm because it can easily be converted into a symbolic program. Symbolic calculations are usually more space economic than programs with explicit state representation, which makes them more applicable.

This was also shown in Section 5.6 where we calculated open implication for "real world" specifications. The astonishing part is that with the help of open implication we could synthesize a specification we were not able to synthesize before. The trick is to find a specification which open implies the original specification and which is easier to synthesize. This is not at all an easy task.

All in all writing correct specifications is already very hard, but writing good and small specifications is even harder. Thus we believe a lot of work still needs to be done to make it easier for developers to write correct and small specifications. Furthermore synthesis must be optimized further to become more practical. Fast synthesis tools will probably also lead to fast decision procedures for open equivalence.

Bibliography

[Ber] Berkeley Logic Synthesis and Verification Group. Abc: A system for sequential synthesis and verification. release 61208. http://www.eecs.berkeley.edu/∼alanmi/abc/.

[BGJ$^+$07a] R. Bloem, S. Galler, B. Jobstmann, N. Piterman, A. Pnueli, and M. Weiglhofer. Automatic hardware synthesis from specification: A case study. In *Proceedings of the Conference on Design, Automation and Test in Europe*, 2007.

[BGJ$^+$07b] R. Bloem, S. Galler, B. Jobstmann, N. Piterman, A. Pnueli, and M. Weiglhofer. Specify, compile, run: Hardware from PSL. In *6th International Workshop on Compiler Optimization Meets Compiler Verification*, 2007.

[Bry86] R. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. on Computers*, C-35:677–691, Aug. 1986.

[Chu62] A. Church. Logic, arithmetic and automata. In *Proceedings International Mathematical Congress*, 1962.

[EH86] E. A. Emerson and J. Y. Halpern. 'Sometimes' and 'not never' revisited: On branching time versus linear time temporal logic. *Journal of the Association for Computing Machinery*, 33(1):151–178, 1986.

[EL86] E. A. Emerson and C.-L. Lei. Efficient model checking in fragments of the propositional mu-calculus. In *Proceedings of the First Annual Symposium of Logic in Computer Science*, pages 267–278, June 1986.

[Eme90] E. A. Emerson. Temporal and modal logic. In van Leeuwen [vL90], chapter 16, pages 995–1072.

[Eme97] E. A. Emerson. Model checking and the mu-calculus. In Neil Immerman and Phokion G. Kolaitis, editors, *Descriptive Complexity and Finite Models*, volume 31 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science.* American Mathematical Society, 1997.

[ES84] E. A. Emerson and A. P. Sistla. Deciding full branching time logic. *Information and Control*, 61(3):175–201, June 1984.

[JB06] B. Jobstmann and Roderick Bloem. Optimizations for LTL synthesis. In *6th Conference on Formal Methods in Computer Aided Design (FMCAD'06)*, pages 117–124, 2006.

[Job07] B. Jobstmann. *Applications and Optimizations for LTL Synthesis.* PhD thesis, Technical University Graz, 2007.

[Jur00] M. Jurdziński. Small progress measures for solving parity games. In *STACS 2000, 17th Annual Symposium on Theoretical Aspects of Computer Science*, pages 290–301, Lille, France, February 2000. Springer. LNCS 1770.

[KPV06] O. Kupferman, N. Piterman, and M. Y. Vardi. Safraless compositional synthesis. In *Conference on Computer Aided Verification (CAV'06)*, pages 31–44, 2006.

[KV05] O. Kupferman and M. Vardi. Safraless decision procedures. In *Symposium on Foundations of Computer Science (FOCS'05)*, pages 531–542, 2005.

[MH84] S. Miyano and T. Hayashi. Alternating finite automata on ω-words. *Theoretical Computer Science*, 32:321–330, 1984.

[MP91] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems *Specification*.* Springer-Verlag, 1991.

[Pit06] N. Piterman. From nondeterministic Büchi and Streett automata to deterministic parity automata. In *21st Symposium on Logic in Computer Science (LICS'06)*, pages 255–264, 2006.

[Pnu77] A. Pnueli. The temporal logic of programs. In *IEEE Symposium on Foundations of Computer Science*, pages 46–57, Providence, RI, 1977.

[PPS06] N. Piterman, A. Pnueli, and Y. Sa'ar. Synthesis of reactive(1) designs. In *Proc. Verification, Model Checking, and Abstract Interpretation (VMCAI'06)*, pages 364–380, 2006.

[PR89] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Proc. Symposium on Principles of Programming Languages (POPL '89)*, pages 179–190, 1989.

[Rab72] M. O. Rabin. *Automata on Infinite Objects and Church's Problem.* Regional Conference Series in Mathematics. American Mathematical Society, Providence, RI, 1972.

[Ros92] R. Rosner. *Modular Synthesis of Reactive Systems.* PhD thesis, Weizmann Institute of Science, 1992.

[Saf88] S. Safra. On the complexity of ω-automata. In *Symposium on Foundations of Computer Science*, pages 319–327, October 1988.

[Sav70] W. J. Savitch. Relationships between nondeterministic and deterministic tape complexities. *J. Comput. Syst. Sci.*, 4(2):177–192, 1970.

[SC85] A. P. Sistla and E. M. Clarke. The complexity of propositional linear temporal logic. *Journal of the Association for Computing Machinery*, 3(32):733–749, 1985.

[Tho90] W. Thomas. Automata on infinite objects. In van Leeuwen [vL90], chapter 4, pages 133–191.

[Var96] M. Y. Vardi. An automata-theoretic approach to linear temporal logic. In *Proceedings of the VIII Banff Higher order workshop conference on Logics for concurrency : structure versus automata*, pages 238–266, Secaucus, NJ, USA, 1996. Springer-Verlag New York, Inc.

[vL90] J. van Leeuwen, editor. *Handbook of Theoretical Computer Science.* The MIT Press/Elsevier, Amsterdam, 1990.

[VW86] M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proceedings of the First Symposium on Logic in Computer Science*, pages 322–331, Cambridge, UK, June 1986.

[VW94] M. Y. Vardi and P. Wolper. Reasoning about infinite computations. *Information and Computation*, 115:1–37, 1994.